An Algorithm for Approximate Tandem Repeats

Gad M. Landau* Jeanette P. Schmidt† Dina Sokol‡
Polytechnic University Polytechnic University Polytechnic University
and and and
Haifa University Incyte Pharmaceuticals Bar-Ilan University

Abstract

A *perfect single tandem repeat* is defined as a nonempty string that can be divided into two identical substrings, e.g. *abcabc*. An *approximate* single tandem repeat is one in which the substrings are similar, but not identical, e.g. *abcdaacd*.

In this paper we consider two criterions of similarity: the Hamming distance (k mismatches) and the edit distance (k differences). For a string S of length n and an integer k our algorithm reports all locally optimal approximate repeats, $r = \bar{u}\hat{u}$, for which the Hamming distance of $\bar{u}$ and $\hat{u}$ is at most k in $O(nk\log(n/k))$ time, or all those for which the edit distance of $\bar{u}$ and $\hat{u}$ is at most k, in $O(nk\log k\log(n/k))$ time.

This paper concentrates on a more general type of repeat called *multiple* tandem repeats. A multiple tandem repeat in a sequence S is a (periodic) substring r of S of the form $r = u^a u'$, where u is a prefix of r and u' is a prefix of u. An *approximate* multiple tandem repeat is a multiple repeat with errors; the repeated subsequences are similar but not identical.

We precisely define approximate multiple repeats, and present an algorithm that finds all repeats that concur with our definition. The time complexity of the algorithm, when searching for repeats with up to k errors in a string S of length n, is $O(nka\log(n/k))$ where a is the maximum number of periods in any reported repeat. We present some experimental results concerning the performance and sensitivity of our algorithm.

The problem of finding repeats within a string is a computational problem with important applications in the field of molecular biology. Both exact and inexact repeats occur frequently in the genome, and certain repeats occurring in the genome are known to be related to diseases in the human.

1 Introduction

In recent years much work has been invested into developing computer algorithms to facilitate research in the field of molecular biology. A large focus has been on string algorithms, since the data for molecular genetics is naturally represented as sequences of characters. In this paper we address a specific computational problem that has biological significance, namely, the search for repeated patterns within a sequence.

A *perfect single tandem repeat* is defined as a nonempty string that can be divided into two identical substrings, *e.g. abcabc*. It is a well-studied problem. Main and Lorentz [ML84] present an $O(n\log n)$ algorithm,

*Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, phone: (718) 260-3154, FAX: (718) 260-3906; email: landau@poly.edu; partially supported by NSF grant CCR-9610238, by NATO Science Programme grant PST.CLG.977017, and by the Israel Science Foundation, founded by the Israel Academy of Sciences and Humanities.

†Incyte Pharmaceuticals, 3174 Porter Drive, Palo Alto, CA 94304; phone:(650) 845-4878, FAX: (650) 855-0572; email: jschmidt@incyte.com; partially supported by NSF grants CCR-9305873.

‡Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel; email: sokold@macs.biu.ac.il; partially supported by NSF grants CCR-9305873.

which reports all *perfect* tandem repeats and Apostolico [Ap92] describes an optimal speed-up parallel algorithm for the problem. Motivations for the exact repeat problem can be found in research in formal languages (see a survey in [ML85]).

Repeats occur frequently in biological sequences, yet they are seldom exact. Hence, we focus our attention on approximately repeated patterns. An *approximate single repeat* is a nonempty string that can be divided into two similar substrings. The distance between the two substrings must be less than a given threshold k, in order for the parts to be considered similar. We can measure the distance between the parts of a repeat using the standard ways of measuring distance between two strings. The simplest measure is the Hamming distance, defined as the number of mismatches between two strings of equal length. The string s=*bearbeer* is an approximate single repeat with a Hamming distance of 1. A more commonly used measure is the edit distance, defined by Levenshtein [Lev66] as the minimum number of insertions, deletions, and substitutions necessary to transform one string into the other. An example of a repeat with edit distance of 2 is, $s = actta|gctt$. The most rigorous measure is the scoring matrix, which can be viewed as a weighted edit distance. A match or substitution receive a score, as do deletions and insertions. In this paper we deal with the Hamming and Levenshtein measures, since these measures give a combinatorial leverage not afforded by the general scoring scheme, leading to much more efficient algorithms.

Algorithms for finding all approximate single repeats, using the weighted edit distance as the similarity measure, can be found in [KM96, B95, S98, M92] and [BW94]. Kannan and Meyers [KM93, KM96] describe a recursive algorithm for finding all highest scoring non-overlapping alignments in $O(n^2 \log^2 n)$ time and $O(n^2 \log n)$ space. Benson [B95] simplifies this algorithm and reduces the space to $O(n^2)$. Schmidt [S98] uses weighted grid graphs to find all locally optimal approximate repeats, improving the time to $O(n^2 \log n)$, with $O(n^2)$ space. Practical use of these algorithms for searching large databases is problematic due to their runtime. Therefore, Benson and Waterman [BW94] developed a heuristic method for fast database search for tandem repeats. Their algorithm, however, requires that the size of the period of the repeat be specified prior to the search.

The problem of finding all approximate single repeats is a subproblem of finding all approximate *multiple* repeats in a string. A *perfect multiple repeat* is a nonempty string that can be divided into a number of identical adjacent substrings, or *periods*. The last period of the repeat can be partial, *e.g. tgtgtgt* is a perfect multiple repeat with 3.5 periods. An *approximate* multiple repeat (or *multirepeat* for short) is a multiple repeat in which the periods of the repeat are approximate. The difficulties of defining approximate multiple repeats are addressed in Section 3, where we present a simple and precise definition. A similar notion is discussed in [SPIS99] where approximate periodicity of strings is defined as follows. Given two strings x and p, p is a *t-approximate period* of x if there exists a partition of x into disjoint blocks of substrings $p_1 \dots p_r$ such that the distance between p and every p_i $(1 \leq i \leq r)$ is less than or equal to t. The algorithm presented in [SPIS99] finds the substring p of the input string x that is the period of x with the minimum distance. When the Hamming distance is used, the complexity of the algorithm is $O(n^3)$, and with the (weighted) edit distance it is $O(n^4)$. Our definition of approximate periodicity is simpler, yielding a more efficient algorithm to solve a problem that is in a sense more general. Our goal is to find *all substrings* of the input string that are "approximately periodic."

A different approach, recently developed by Benson [B99, BS98], defines tandem repeats based on a probabilistic model. The algorithm to find all tandem repeats within a string first finds all runs of k matches at a common distance, called *k-tuple matches*. Then, using a collection of statistical criteria, the k-tuple matches are used to detect the tandem repeats. This algorithm has already been used in conjunction with a sequence alignment algorithm [B97].

Multiple repeats occur frequently in both DNA and protein sequences. Some multiple repeats have been associated with human genetic diseases. For example, the triplet CGG is tandemly repeated 6 to 54 times in a normal FMR-l gene. In patients with the Fragile X Syndrome, the pattern occurs more than 200 times. Kennedy disease and Myotonic Dystrophy(DM) are two other diseases that have been associated with triplet repeats [C92]. Another important application for finding multiple repeats in biological sequences is related to the multiple sequence alignment. Producing multiple alignments becomes very complicated when the

sequences to be aligned contain multiple repeats, because matches may be present in numerous places. As a precursor to multiple alignments, it is helpful to recognize all multiple repeats within the set of strings that must be aligned.

This paper continues work started in [LS93]. In the next section we describe an algorithm that finds all approximate single repeats within a sequence. In the case of the Hamming distance our algorithm runs in $O(nk\log(n/k))$ time. For the case of the edit distance our algorithm runs in $O(nk\log k\log(n/k))$ time. In Section 3 we formally define multiple repeats with and without errors. Section 4 presents an algorithm to find all approximate multiple repeats with at most k errors within a string of length n. The algorithm presented runs in $O(nka\log(n/k))$ where a is the maximum number of periods in any reported repeat. Some implementation details will be given in Section 5.

2 Algorithms for Approximate Single Repeats

The framework of our algorithm is similar to the algorithm for *perfect* repeats described by Main and Lorentz [ML84, ML85]. In section 2.1 we briefly describe the algorithm of [ML84], followed by an overview of the algorithm for approximate single repeats.

2.1 Algorithm Overview

Given a string, $S = s_1 \ldots s_n$, the algorithm of Main and Lorentz [ML84, ML85] finds all *perfect* single repeats within S. The algorithm is a recursive, divide-and-conquer algorithm. At the highest level, S is viewed as the concatenation of two strings, $S = uv$, $u = s_1 \ldots s_{n/2}$ and $v = s_{n/2+1} \ldots s_n$. (to simplify the presentation we assume that n is a power of 2.) The algorithm finds all repeats that span both u and v (i.e. cross the center of S), and then calls itself recursively on both u and v. Every repeat within S crosses the center of a substring of S at some point in the recursion. As proof of this, consider a repeat that does not cross the center of S, and hence is not found at the first level. The repeat will be divided among different substrings at some point in the recursion since in the final step of the recursion the input strings are of length 1. In the step prior to its division, a given repeat must cross the center of uv as the center is the splitting point.

The repeats that span uv are classified into two groups according to where their centers lie. A *right* repeat has its center at or to the right of the boundary between u and v. A *left* repeat has its center to the left of this boundary. Following we describe the procedure to find all right repeats at the highest level, for $S = s_1 \ldots s_n$. By symmetry, all left repeats can be found.

Find right repeats

For $p = 1$ to $n/2$ do // find repeats with period p

(1) $j = n/2 + p$

(2) Forward Direction: Find the longest prefix of $s_j \ldots s_n$ that matches a prefix of $s_{n/2} \ldots s_n$. Let the length of the prefix be ℓ_1 (see figure 1).

(3) Reverse Direction: Find the longest suffix of $s_{n/2} \ldots s_{j-1}$ that matches a suffix of $s_1 \ldots s_{n/2-1}$. Let the length of the suffix be ℓ_2.

(4) If $\ell_1 + \ell_2 \geq p$ then there is at least one tandem repeat of length $2p$. Tandem repeats of period p begin at positions $n/2 - \ell_2 \ldots n/2 + \ell_1 - p$.

od

The algorithm for finding *approximate* tandem repeats follows the framework of the Main and Lorentz [ML84, ML85] algorithm. Given a string $S = s_1 \ldots s_n$, the algorithm has $O(\log n)$ iterations. We assume for simplicity of presentation that n is a power of 2. (No difficulty arises in the general case). In iteration i the string is divided into the $n/2^i$ substrings, $(s_1 \ldots s_{2^i}), \ldots (s_{\ell 2^i+1} \ldots s_{(\ell+1)2^i}), \ldots (s_{n-2^i+1} \ldots s_n)$. For each

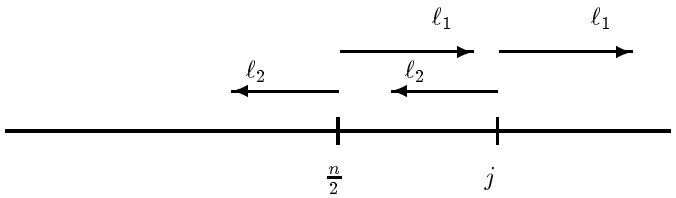

Figure 1: Computing right repeats: ℓ_1 is the length of the forward direction comparisons, and ℓ_2 is the length of the reverse direction comparisons. Repeats with period $p = j - n/2$ (of length $2p$) that include $s_{n/2}$ begin at positions in the range $n/2 - \ell_2 + 1 \ldots n/2 + \ell_1 - p$.

substring, $s_{\ell 2^i+1} \ldots s_{(\ell+1)2^i}$, separately, all approximate tandem repeats $r = \bar{u}\hat{u}$ are found, for which r is contained in the substring and for which the two middle characters of the substring ($s_{\ell 2^i+2^{i-1}}$ and $s_{\ell 2^i+2^{i-1}+1}$) are in r. Each such iteration has two steps. In the **first step** one finds all approximate tandem *right* repeats. An approximate tandem right repeat includes $s_{\ell 2^i+2^{i-1}}$ in $\bar{u}$. In the **second step** the approximate tandem *left* repeats, where $s_{\ell 2^i+2^{i-1}}$ is in $\hat{u}$, are identified. For the Hamming distance our algorithm will start with iteration $\log k$, i.e. with strings of length k. The repeats within these strings of length k will be found by a straightforward algorithm in $O(k^2)$ time per string.

In what follows, we are going to describe the first step of any given iteration. Since iterations only differ in that they consider strings of different sizes, consider the iteration which finds an approximate tandem repeat $r = \bar{u}\hat{u}$ in the string $S = s_1 \ldots s_n$, for which $s_{n/2} \in \bar{u}$. As in [ML84, ML85] for each j, $n/2 < j \leq n$, we determine the indexes ℓ, for which $s_{n/2+1} \ldots s_\ell$, $\ell \leq j$ is similar to some prefix $s_{j+1} \ldots s_\pi$ of $s_{j+1} \ldots s_n$, and $s_{\ell+1} \ldots s_j$ is similar to some suffix $s_\sigma \ldots s_{n/2}$ of $s_1 \ldots s_{n/2}$.

We then have: $\begin{array}{lll} \bar{u}= & s_\sigma \ldots s_{n/2} & s_{n/2+1} \ldots s_\ell \\ \hat{u}= s_{\ell+1} \ldots s_j & & s_{j+1} \ldots \quad s_\pi \end{array}$ is an *approximate* repeat if $\bar{u}$ is "sufficiently" similar to $\hat{u}$. To find all indices ℓ which might be candidates for such a repeat, (for a given j, $n/2 < j \leq n$), we will determine for each $k_1 \leq k$ the longest suffix $s_{\ell_{k_1}} \ldots s_j$ of $s_1 \ldots s_j$ whose Hamming (or edit) distance with some suffix of $s_1 \ldots s_{n/2}$ is no more than k_1. We also determine for each $k_2 \leq k$ the *longest* prefix $s_{n/2+1} \ldots s_{\ell_{k_2}}$ of $s_{n/2+1} \ldots s_n$ whose Hamming (or edit) distance with some prefix of $s_{j+1} \ldots s_n$ is no more than k_2:

$$\left.\begin{array}{ll} \ldots & \ldots s_{n/2} \\ s_{\ell_{k_1}} .. s_{\ell+1} & \ldots s_j \end{array}\right\} \quad \textit{distance } k_1, \qquad \left.\begin{array}{lll} s_{n/2+1} \ldots & & s_\ell .. s_{\ell_{k_2}} \\ s_{j+1} & \ldots & \ldots \end{array}\right\} \quad \textit{distance } k_2.$$

For any pair k_1, k_2 for which $k_1 + k_2 \leq k$, and $\ell_{k_1} - 1 \leq \ell \leq \ell_{k_2}$ an *approximate repeat* is identified. Note that for the Hamming distance $|\bar{u}| = |\hat{u}| = j - n/2$.

2.2 Hamming distance

In this section we describe the use of the Hamming distance as the similarity measure. We also translate the alignments described in the previous section to alignment paths, (or in fact diagonals) in the dynamic programming matrix. Although this is not essential to follow the computation of the Hamming distance it does set the ground for a very easy understanding of the computations necessary when the measure of similarity is the edit distance. We describe the algorithm for identifying all approximate tandem repeats in S for which $s_{n/2}$ is in $\bar{u}$, (identifying those for which $s_{n/2}$ is in $\hat{u}$ requires simple straightforward modifications). The algorithm has two parts.

Part 1. We align the string S with itself and consider the dynamic programming matrix $D[0 \ldots n, 0 \ldots n]$. We divide the bottom half of D into two sub-matrices: $PR[n/2 \ldots n; n/2 \ldots n] = D[n/2 \ldots n; n/2 \ldots n]$, (to compute common prefixes), and $SU[n/2 \ldots n; 0 \ldots n/2] = D[n/2 \ldots n; 0 \ldots n/2]$, (to compute common

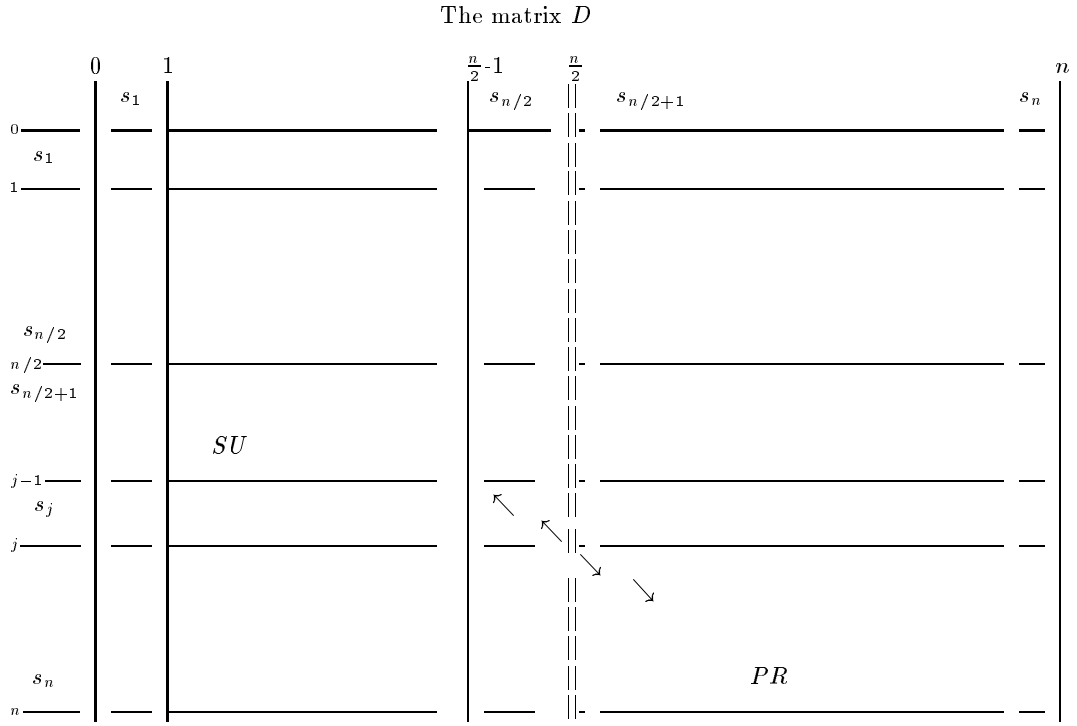

Figure 2: Subdivision of the matrix D; shown are the matrices PR and SU in the computation for index j. The point $D[i,i]$ corresponds to the intersection point of the vertical line labeled i and the horizontal line labeled i. In PR, for $c', c > 0$, the value of an alignment $s_{j+1} \ldots s_{j+c}$ and $s_{n/2+1} \ldots s_{n/2+c'}$ is stored in the bottom right corner of the square $s_{j+c}, s_{n/2+c'}$, (i.e. in $PR[j+c, n/2+c']$); in SU, for $c', c \geq 0$, an alignment $s_j \ldots s_{j-c}$ and $s_{n/2} \ldots s_{n/2-c'}$ is stored in the upper left corner of the square $s_{j-c}, s_{n/2-c'}$, (i.e. in $SU[j-c-1, n/2-c'-1]$).

suffixes). SU and PR share the middle column of D, which both use for the purpose of initialization, (see Figure 2). For any $j > n/2$ any sub-diagonal through $D[j, n/2]$ of length $j - n/2$ represents a tandem repeat $r = \bar{u}\hat{u}$, with $s_{n/2} \in \bar{u}$. (The algorithm then, as in [ML84, ML85], recursively continues with sub-matrices $D[0 \ldots n/2, 0 \ldots n/2]$ and $D[n/2 \ldots n, n/2 \ldots n]$.) Figure 3 shows the linear representation of the similar prefixes for a given j.

It remains to identify those diagonals which represent tandem repeats with less than k mismatches. Each diagonal of PR and SU is computed separately. In PR the computation starts from the left side of the matrix, with $PR[j+c, n/2+c]$ holding the value of the alignment of $s_{j+1} \ldots s_{j+c}$ with $s_{n/2+1} \ldots s_{n/2+c}$. In SU the computation starts from the right side of the matrix, with $SU[j-c, n/2-c]$ holding the value of the alignment of $s_j \ldots s_{j-c+1}$ with $s_{n/2} \ldots s_{n/2-c+1}$. ($SU[j, n/2] = PR[j, n/2] = 0$.) The computation is similar for all diagonals.

Let $j > n/2$ and consider the diagonal through $D[j, n/2]$. It can be computed in the following straightforward manner:

Computing PR

$e = 0$;
For $h = 0$ to k do
$PR[j+e, n/2+e] = h$ (an auxiliary value when $h = 0$)
While $(j + e + 1 \leq n)$ and $(s_{j+e+1} = s_{n/2+e+1})$ do
$e = e + 1$; $PR[j+e, n/2+e] = h$

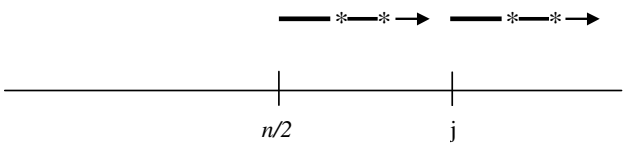

Figure 3: The forward and reverse extensions are computed from positions $n/2$ and j. However, we allow up to k mismatches between the similar prefixes/suffixes. The figure shows the similar prefixes, with the asterisks representing the positions of mismatches.

od
$P^*[j,h] = e$; $e = e + 1$;
od

Computing SU

$e = 0$;
For $h = 0$ to k do
$SU[j - e, n/2 - e] = h$ (an auxiliary value when $h = 0$)
While $(j - e > n/2)$ and $(s_{j-e} = s_{n/2-e})$ do
$e = e + 1$; $SU[j - e, n/2 - e] = h$
od
$S^*[j,h] = e$; $e = e + 1$;
od

Successive values along the diagonals increase and differ by at most one and only the last row (respectively column) in each diagonal that contains a value h less or equal to k is of interest. These values are recorded in the matrices P^* and S^*. The last row with value h on the diagonal that goes through $D[j, n/2]$ is $(j + P^*[j,h])$. Similarly $(j - S^*[j,h])$ is the last row, (upward from j), with value h on that diagonal. $P^*[j,h]$ (resp. $S^*[j,h]$) can be computed from $P^*[j,h-1]$ (resp. $S^*[j,h-1]$) in $O(1)$ time using suffix trees, as shown in [LV88], in place of the above while loop. Therefore, for a given j, Part 1 of step 1 (resp. 2) is computed in $O(k)$ time. It remains to combine the pieces.

Part 2. We can now identify all repeats $r = \bar{u}\hat{u}$ for which $s_{n/2} \in \bar{u}$ and in which s_j $(j > n/2)$ is aligned with $s_{n/2}$. Define $j^* = j - n/2$, and note that all the indexes on this diagonal are of the form $[m, m - j^*]$. The repeats correspond to all the sub-diagonals of length j^* through $D[j, n/2]$, i.e. from any $[\ell, \ell - j^*]$ to $[\ell + j^*, \ell]$, $\ell > n/2$, for which $SU[\ell, \ell - j^*] + PR[\ell + j^*, \ell] \leq k$.

It is reasonable to report only those repeats $r = \bar{u}\hat{u}$ through $D[j, n/2]$ for which the Hamming distance between $\bar{u}$ and $\hat{u}$ is minimal among such repeats. More generally we compute for each $h \leq k$ the value $f(h) = \min_{h' \leq k-h}\{h' \mid (P^*[j,h] + S^*[j,h'] \geq j^*)\}$. If no such h' can be found $f(h)$ is undefined. These values can be easily computed by a merge like procedure from the data collected in Part 1 in $O(k)$ time for each point j, as illustrated below.

Computing optimal repeats

$h' = k + 1$;
for $h = 0$ to k

while $(h' > 0)$ and $(P^*[j,h] + S^*[j,h'-1] \geq j^*)$ do
$h' = h' - 1$;
od
if $((h'+h) \leq k)$ then $f(h) = h'$ else $h' = h' - 1$;
od

For all those h for which $h + f(h)$ is minimal we can now report all sub-diagonals of length j^* that start between $D[j - S^*[j, f(h)], n/2 - S^*[j, f(h)]]$ and $D[j + P^*[j,h] - j^*, n/2 + P^*[j,h] - j^*]$. (A sub-diagonal from $D[j-c, n/2-c]$ to $D[j+j^*-c, j-c]$ corresponds to a repeat $r = \bar{u}\hat{u}$, with $\hat{u} = s_{j-c+1} \ldots s_{j+j^*-c}$ and $\bar{u} = s_{n/2-c+1} \ldots s_{j-c}$.)

Strings of size k. (Sketch) For small strings of size $\ell \leq k$ we use a simple $O(\ell^2)$ algorithm. For a given $t \leq \ell/2$, it gives the number of mismatches corresponding to all repeats $r = \bar{u}\hat{u}$ of length $2t$ in $O(\ell)$ time.

Complexity. Our iterations follow the $O(n \log n)$ Main and Lorentz algorithm, but our algorithm starts with strings of size k, and has therefore $\log(n/k)$ iterations. For each substring of length ℓ, our algorithm takes $O(\ell k)$, and hence each iteration takes $O(nk)$ time. The initial iteration, (for strings of length k), takes $O((n/k)k^2) = O(nk)$ time. The total running time of our algorithm is therefore $O(nk \log(n/k))$.

2.3 Edit distance

The computation of the edit distance will proceed similarly to the computation of the Hamming distance. The nature of the repeats is somewhat different than before in that now a repeat $r = \bar{u}\hat{u}$ no longer has the property that $|\bar{u}| = |\hat{u}|$. We will still carry out the computation using the matrices D, (with its sub-matrices SU and PR) and consider all repeats for which the alignment between $\bar{u}$ and $\hat{u}$ goes through $D[j, n/2]$, $j > n/2$. The repeats in this case correspond to paths in the matrix which start at $D[j - r_1, \ell_1]$, $(j - r_1 \geq n/2,\ \ell_1 \leq n/2)$, and end at $D[\ell_2, j - r_1]$, for some $\ell_2 \geq j$. In the previous section (Hamming distance) one diagonal in D was computed for each j. Here, the computation for each j progresses along several diagonals. Therefore, we are obliged to compute a matrix D for each j.

Consider the computation of PR for j. A straightforward, $O(n^2)$, computation would be:

for $a = j$ to n
$PR[a, n/2] = a - j$
for $b = n/2$ to n
$PR[j, b] = b - n/2$
for $a = j + 1$ to n
for $b = n/2 + 1$ to n do
$PR[a,b] = \min\{PR[a, b-1] + 1, PR[a-1, b] + 1, ((PR[a-1, b-1]$ if $s_a = s_b)$ and $(PR[a - 1, b-1] + 1$ otherwise$)\}$
od

As before the computation in PR starts from the left side of the matrix, and the computation of SU from the right side. In PR for each $h \leq k$ we identify the longest prefix of $s_{n/2+1} \ldots s_n$ which matches some prefix of $s_{j+1} \ldots s_n$ with h differences. This will correspond to a path starting at $[j, n/2]$ which stretches as far right as possible. In SU, for each $h' \leq k$, we identify the longest suffix of $s_{n/2} \ldots s_j$ that matches some suffix of $s_1 \ldots s_{n/2}$ with h' differences. This will correspond to a path in SU that starts at $[j, n/2]$ and stretches as high as possible. (See Figure 2 for the direction of these paths). Two such paths can be combined to an approximate repeat with at most k differences if $h + h' \leq k$.

Consider the computation of PR. We compute the edit distance matrix for the strings $s_{n/2+1} \ldots s_j$ and $s_{j+1} \ldots s_n$. The values along the diagonals in PR increase and successive values differ by at most one, it is

hence sufficient to record, for each $h \leq k$, the last element on each diagonal whose value is h, as observed in [U83]. In order to simplify the explanation we give the diagonals fixed numbers, which are independent of the reference point j. Define the diagonal through $[d, n/2]$ as diagonal d. (Note that all diagonals in any PR have positive numbers.) Let $p_j^*(d,h)$ be the column number, (in PR for a given j), of the last element with value h on diagonal d. Since the smallest value on diagonals $j \pm x$ is $|x|$ it is sufficient to compute the values on the $2k+1$ diagonals corresponding to $x \in [j+k, j-k]$. The list $p_j^*(j+h,h) \ldots p_j^*(j,h) \ldots p_j^*(j-h,h)$ is defined as *wave* h, and denoted by L_j^h, $(p_j^*(\ell,h) = L_j^h(\ell))$. Note that by slight abuse of notation, but without ambiguity, $L_j^h(\ell)$ will denote both the point on diagonal ℓ on wave h, as well the column number of that point. Also note that for a given j the information stored in PR, is now stored in k waves each of size (at most) $2k+1$.

We define $P^*[j,h] = max_\ell\ L_j^h(\ell) - n/2$, the maximum number of columns to the right of $n/2$ reached with at most h differences. Since there might be more than one such point we define $\pi(j,h)$ as the *set* of corresponding end points on the wave.

In [LMS98] we discuss how to compute the k waves (wave 0, wave 1, ... wave k). Section 2.3.1 describes an algorithm to find $P^*[j,h]$ and $\pi(j,h)$ for each wave.

Until the end of this subsection assume that $P^*[j,h]$ and $\pi(j,h)$ have already been computed for all the waves.

The computations for SU are very similar to the ones of PR. The diagonals are defined exactly as before, and $s^*(d,h)$ will be the number of rows upwards from j, of the last element with value h on diagonal d. $S^*[j,h] = max_d\ s^*(d,h)$, is the maximum number of rows upwards reached with at most h differences by a path that starts on $[j, n/2]$; the set of corresponding end points on the wave is denoted by $\sigma(j,h)$.

Tandem repeats $r = \bar{u}\hat{u}$ with $s_{n/2} \in \bar{u}$ aligned with $s_j \in \hat{u}$ with at most k differences correspond to pairs $S^*[j,h'], P^*[j,h]$, for which $h' + h \leq k$ and $S^*[j,h'] + P^*[j,h] \geq j - n/2$.

As before to obtain the tandem repeats with the least number of differences we perform the same merge like procedure as described for the Hamming distance. In particular we compute for each $h \leq k$ the value $f(h) = \min_{h' \leq k-h}\{h' \mid (P^*[j,h] + S^*[j,h'] \geq j - n/2)\}$ and consider only those pairs $(h, f(h))$ for which $h + f(h)$ is minimal. The computation here is the same as in "Computing optimal repeats" in Section 2.2. For such optimal pairs (h,h'), the repeats are given by the sub-paths that start at any point in $\sigma(j,h')$, go through $[n/2, j]$ and end at any point in $\pi(j,h)$. Since these pairs are optimal, whenever the corresponding values $P^*[j,h] + S^*[j,h'] > j - n/2$, these paths start with matches along a diagonal and end with matches along a diagonal. Let $(i, i_1) \in \sigma(j,h')$ and let $(i', i_1') \in \pi(j,h)$. Notice that by definition all points in $\sigma(j,h')$ have the same coordinate $i = j - S^*[j,h']$ and those in $\pi(j,h)$ have the same coordinate $i_1' = n/2 + P^*[j,h]$. In addition if $P^*[j,h] + S^*[j,h'] = j - n/2 + c$, for $c > 0$ then $i_1' = i + c$ for all such points, and the repeats correspond to the sub-paths starting at $(i + r, i_1 + r)$ and ending at $(i' - c + r, i_1' - c + r)$, for $0 \leq r \leq c$.

2.3.1 Finding the maximum on each wave

In [LMS98] we described an $O(k)$ algorithm that computes the waves (L_j^h, for $h \leq k$) in the dynamic programming matrix of PR for a given j. In this section we compute $P^*[j,h]$ - the maximum value on L_j^h. Computing $P^*[j,h]$ in $O(1)$ time, would provide a $O(nk)$ running time for the entire iteration. Unfortunately, we were only able to compute $P^*[j,h]$ in $O(\log k)$ time.

In wave h for index j, $h \leq k$, the rightmost column c on each diagonal d, $j + h \leq d \leq j - h$, with value h is reported. The reports are concatenated in a linked list starting with the report of diagonal $j + h$ and ending with report of diagonal $j - h$. In [LMS98], we showed how to compute wave h for index j by concatenating up to three sub-waves from the computation of PR for index $j + 1$. Since the computation of the waves for index j are done after those for index $j + 1$, we refer to L_{j+1}^h as the old h wave and L_j^h as the new h wave. The new h wave was computed by concatenating a prefix of the old $h - 1$ wave, a sublist of the old h wave, and a suffix of the old $h + 1$ wave, and adding up to two new points. This was done in $O(1)$ time, by

changing $O(1)$ pointers.

Finding the maximum of the concatenated list can be reduced to the following more general problem. Given three unsorted linked lists L^1, L^2 and L^3 of $O(h)$ elements each and their respective maximum, as well as two pointers p_1^i, p_2^i to list L^i, $(i = 1..3)$, the problem is to construct the new list $L^* = p_1^1 \dots p_2^1, p_1^2 \dots p_2^2, p_1^3 \dots p_2^3$ and to find its maximum. Since the problem of sorting the elements of a linked list L can easily be reduced to the above problem, there is little hope for a faster algorithm for this general problem. It is not clear however that the special circumstances we are dealing with could not be somehow exploited.

In the current paper we are using mergeable 2-3 heaps to compute $P^*[j,h]$ in $O(\log k)$ time, as for example described in [AHU74], Sections 4.11-4.12. Each list corresponds to the list of leaves of a 2-3 tree. The leaves of each subtree are hence reports of consecutive diagonals. The information stored in each internal node is the maximum value of the leaves in its subtree. To build a 2-3 heap on the linked list of the new wave h, the algorithm merges a prefix of the 2-3 heap of the old $h-1$ wave with a sublist of the 2-3 heap of the old h wave and the suffix of the 2-3 heap of the old $h+1$ wave. In addition at most 2 new leaves, (reports) need to be inserted.

The algorithm described in [LMS98] provides the pointers to each starting and end point of the sub-waves. The 2-3 heap corresponding to the new h wave is constructed in $O(\log h)$ time. $P^*[j,h]$ is therefore obtained by accessing the root of the 2-3 heap. The total time to find all new maxima for index j is hence $O(k \log k)$.

Strings of size k. For small strings of size $\ell \leq k$, the number of errors is at most ℓ. Hence, the time to find the new maxima for a given index is $O(\ell \log \ell)$, and for all indices is $O(\ell^2 \log \ell)$. There are n/ℓ such strings in a given iteration, hence each iteration takes $O(n\ell \log \ell)$ time. The total time for all iterations $0 \dots \log k$ is $\Sigma_{i=0}^{\log k} n\frac{k}{2^i} \log \frac{k}{2^i} = O(nk \log k)$.

Complexity. As in the Hamming distance case the number of iterations is $\log n/k$. Each iteration takes $O(nk \log k)$ time. The total running time of the algorithm for the edit distance is therefore $O(nk \log k \log(n/k))$.

3 Multiple Repeats - Definitions

3.1 Perfect Multiple Repeats

Definition 1 (multiple repeat) *A* multiple repeat *is a string r of the form $r = u^a u'$, where u is a prefix of r, u' is a prefix of u, and $a \geq 2$.*

A multiple repeat is a periodic string where some substring u occurs consecutively two or more times. A multiple repeat $r = u^a$ can be viewed as a set of overlapping single repeats, where a distinct single repeat begins at each position of $u_1 u_2 \dots u_{a-2}$. For many applications, this is not the preferred way of viewing the multiple repeat. Rather, a multiple repeat should be reported as one unit, the substring u repeated a times.

Within a string S, there may be many substrings that satisfy the above definition of a multiple repeat. However, a complete solution need not report every such substring. We discuss two more criteria that will substantially decrease the number of reported repeats, without detracting from the essential output.

Definition 2 (maximal) *A multiple repeat r within a string S is* maximal *if it cannot be extended at either end. If s_i is the preceding character in S, and s_j is the following character, then both $s_i r$ and $r s_j$ are not multiple repeats.*

Note that it is legitimate for more than one maximal repeat to begin at a single position. For example, in the string *abababcababababc* two distinct repeats begin at position 1. (*ab* occurring 3 times, and *abababc* occurring twice.) However, it is also possible (according to the definition) for two or more maximal repeats to share a starting and ending point. When several repeats span the exact same characters, the only difference

between the repeats is in the grouping of the characters. To illustrate, the string *abababababab* represents three different multirepeats, *ab* occurring 6 times, *abab* occurring 3 times, and *ababab* twice. In such a case, we define the multirepeat with the smallest period to be *primitive*, since it contains the information about all other possible multirepeats.

Definition 3 (primitive) *A repeat $r = u^a$ is* primitive *if u is not a multiple repeat.*

Definition 4 (canonical) *A multiple repeat is* canonical *if it is both maximal and primitive.*

The canonical multiple repeats within a string S implicitly encode all repeats that occur in S.

3.2 Approximate Multiple Repeats

In this section we introduce approximate multiple repeats, and we define a way of counting the number of errors in a multirepeat. To obtain an exact error count of an approximate *single* repeat, we simply measure the distance between the two parts of the repeat. As discussed previously, the Hamming distance or the edit distance can be used for this purpose. When we attempt to extend this definition to include approximate *multiple* repeats, we are faced with the problem of measuring the distance among several strings. Many measures have been proposed that capture the similarity among all strings within a set. Unfortunately, the computation of many of the most intuitive measures is NP-complete. We are therefore proposing a measure that is easily computable, yet captures many of the biologically relevant relationships between biosequences.

For simplicity, we would like to use a variation of the Hamming distance. There are several ways that the definition of Hamming distance can be extended to apply to several strings. Following we consider a few options, and then present our definition, which uses elements of the following definitions.
Possible Definitions for k mismatches:

1. If the maximum Hamming distance between all adjacent parts of the repeat is k, then we say that the repeat has k mismatches. (Note that when using this definition the first substring of the repeat can degenerate into a substring that has nothing in common with itself. e.g. *abcd*|*fbcd*|*fgcd*|*fghd*|*fghi*).
2. If the maximum Hamming distance between *any* pair of substrings is k, then we say that the repeat has k mismatches.
3. If there exists some string whose Hamming distance with every substring of the repeat is no more than k, then the repeat has no more than k mismatches. (Note the similarity to the *consensus* string, used to score multiple alignments, and see [SPIS99] for a similar definition of approximate periods of a string.)

Our approach, formalized in definition 6, is to view the approximate multiple repeat as a multialignment of its parts. Each row in the alignment is a period of the repeat, without any inserted gap characters. Column j consists of all characters at position j in each period. We count one mismatch for every column in the alignment that contains occurrences of two or more distinct characters. Thus, there are k mismatches in a multirepeat if its alignment contains k *error columns*, i.e. k columns that are not uniform.

It is convenient to introduce the (known) notion of wildcard when defining approximate repeats.

Definition 5 (wildcard character) *A wildcard in a string (denoted by *) is a character that can be replaced by any character. The strings represented by a string u that contains wildcards will be called instances of u.*

Definition 6 (approximate multiple repeat) *An* approximate multiple repeat *with k error columns, is a pair (r, u), also denoted by $r = u^a u'$, that satisfies the following conditions:*

1) r is a string over Σ and u is a string with k wildcards in positions $e_1, \ldots, e_k$.

2) r can be written as $u_1, u_2, \ldots u_a, u'$, where $a \geq 2$, each u_i is an instance of u, and u' is an instance of a prefix of u.

3) None of the k columns $e_1, \ldots, e_k$ in the multiple alignment of $u_1, u_2, \ldots, u_a, u'$ is uniform, (i.e. each contain at least two different characters).

Note that given a string r the choice of $|u|$ may not be unique. This creates some difficulty when dealing with multiple repeats. These difficulties are addressed in the following definitions.

Definition 7 (canonical k-repeat) *An approximate multiple repeat (r, u) is a* canonical k-repeat *if u has $\leq k$ wildcards, (and hence (r, u) has at most k error columns) and further satisfies the following:*

1. *r contains no* extraneous *characters.*
2. *(r, u) is* k-maximal.
3. *(r, u) is* k-primitive.

Definitions of the three terms: extraneous, k-maximal, and k-primitive, immediately follow.

Definition 8 (extraneous) *Let $r = u^a u' = r_1, \ldots, r_n$ be a multiple repeat with k error columns and let u'' be the prefix of u of which u' is an instance.*

r has extraneous *error characters if $|r| > 2|u|$ and one of the following two conditions hold:*

1. *$u = *, v$ (the first element of u is a wildcard, hence the first column is an error column).*
2. *$u'' = v'', *$ (the last element of u'' is a wildcard, hence the last column is an error column).*

Note that a simple repeat, i.e. a repeat with period p and length $2p$, cannot have any extraneous characters. Although the last (or first) character of such a repeat may be in an error column, if it were to be removed, the repeat would become too short to have two equal length parts.

Definition 9 (k-maximal) *Let (r, u) be an approximate multiple repeat* **without** *extraneous characters, containing e error columns. (r, u) is* k-maximal *if, by further extending the repeat to get $k - e$ new errors on either side of r, no additional matching characters will be added to a uniform column of (r, u).*

For example, consider the repeat that appears in the string *aggctacgctaccct*. The following repeat is *not* 2-maximal since with the addition of a single error (at the end) we can add characters to 2 uniform columns.

```
                        2-maximal repeat:
aggct                   aggct
 |                       ||
acgct                   acgct
                         ||
a                       accct
```

We say that two repeats, (r_1, u) and (r_2, v), *span the same substring* if $r_1 = r_2$. If two repeats either begin or end at the same position, but they do not span the same substring, then the longer of the two repeats is said to *include* the shorter repeat. Note that several approximate multirepeats can span a single substring even if the period lengths are not multiples of each other.

Definition 10 (k-primitive) *Let (r, u) be a repeat with e error columns, and let $|u| = p$. (r, u) is* k-primitive *if e is strictly minimal over all k-maximal repeats with period $< p$ that either span the same substring as (r, u), or include (r, u).*

Definition 10 is based on the observation that a repeat that overlaps with smaller period repeats, but has fewer errors than they do, has information that is not captured in all other repeats. The following example illustrates this observation.

```
big        bigbag
 |         bigbag
bag
 |
big
 |
bag
```

Note that when the number of errors is zero, the definition of k-primitive reduces to the definition of primitive repeats without errors (definition 3).

4 The Algorithm for Approximate Multiple Repeats

4.1 Algorithm's Framework

The algorithm for the approximate multiple repeat problem is an extension of the one described in section 2.2, for the Hamming distance. The framework of the algorithm is the same. There are $\log(n/k)$ iterations; in each iteration i, the input string S is divided into $n/2^i$ substrings of length 2^i. In the first iteration ($i = \log n$), the entire string S is processed and the repeats that include both $s_{n/2}$ and $s_{n/2+1}$ are reported. In each subsequent iteration i, the input string is divided into the substrings, $(s_1 \ldots s_{2^i}), \ldots (s_{\ell 2^i+1} \ldots s_{(\ell+1)2^i}), \ldots (s_{n-2^i+1} \ldots s_n)$. Each of these substrings is searched individually for all repeats that span its center characters.

Every repeat within S must span the center characters of some substring in some iteration. However, in some iteration, a maximal repeat may be split among different substrings of the input. When examining an isolated substring, a part of the maximal repeat may appear to be a maximal repeat. For example, in the second iteration, the input string $S = abab|abff$ is divided into 2 substrings. The maximal repeat $ababab$ is also divided among the two substrings. When searching the first substring, the algorithm must recognize that the repeat $abab$ is not maximal.

Claim 1 *Any repeat that extends into the previous or next substring (of the input) by at least 1 character, is a sub-repeat of a multirepeat that will have been reported in an earlier iteration.*

Proof: We show that a repeat that extends into the following substring spans the center characters of a larger substring, and hence had to have been reported in an earlier iteration. (The same logic applies for repeats extending into the previous substring.)

In iteration i the input string is divided into $n/2^i$ substrings numbered $S_0 \ldots S_{n/2^i-1}$. Each substring S_π includes the following characters from the input string: $S_\pi = s_{\pi 2^i+1} \ldots s_{(\pi+1)2^i}$. Let us assume that some repeat is found in iteration i in the substring S_π, and it extends by 1 character into the following substring, $S_{\pi+1}$. We show that this repeat was already found in the previous iteration.

Since we begin our iterations with $i = \log n$, and decrement i by one between iterations, the previous iteration to i is iteration $i+1$. In iteration $i+1$ the number of substrings is exactly half of the number of substrings in iteration i ($\frac{n}{2^{i+1}} = \frac{1}{2} \cdot \frac{n}{2^i}$). Therefore, if π is even, the substring $S_{\pi/2}$ in iteration $i+1$ is the concatenation of S_π and $S_{\pi+1}$ from iteration i. The center characters of this substring correspond to the last character of S_π and the first character of $S_{\pi+1}$. ($S_{\pi/2} = s_{(\pi/2)2^{i+1}+1} \ldots s_{(\pi/2+1)2^{i+1}}$.) Thus, a repeat in iteration i that includes the last character of S_π and the first character of $S_{\pi+1}$, also includes the center characters of the substring $S_{\pi/2}$ in iteration $i+1$, and hence is reported in iteration $i+1$. (If π is odd, then this is true for $S_{\pi/4}$ in iteration $i+2$.) □

Hence, to determine whether a repeat is maximal it is necessary to check whether the repeat extends into a neighboring substring with characters in matching columns. Up to $k+1$ comparisons may be necessary, however, for each period size at most one repeat can extend in either direction (specifically, the repeat with the leftmost starting point and the repeat with the rightmost endpoint).

4.2 Finding Repeats within a Substring

In each iteration of the algorithm the input string S is divided into $n/2^i$ substrings of length 2^i and the repeats within each substring, that span the center characters of the substring, are found. Since the processing of each substring is the same in every iteration, we describe the first iteration, where $i = \log n$, and the entire string S is processed. The search for repeats is done in two steps, in the **first step** one finds all approximate tandem repeats in which $s_{n/2+p}$ is included as the corresponding character to $s_{n/2}$ (for repeats with period p), and in the **second step** the approximate tandem repeats in which $s_{n/2-p}$ corresponds to $s_{n/2}$ are identified. The first step of the algorithm uses the bottom half of the dynamic programming matrix $D[0 \ldots n, 0 \ldots n]$, with $n/2 < j \leq n$, and the second step is similar, with $1 \leq j < n/2$, to complete the top half of the matrix D. As in section 2, we describe only the first step. The first step is divided into two parts: computing the matrix, and identifying the repeats.

Part 1: Compute P^* and S^*

Recall that we divide the bottom half of D into two sub-matrices:
$PR[n/2 \ldots n; n/2 \ldots n] = D[n/2 \ldots n; n/2 \ldots n]$, (to compute common prefixes), and
$SU[n/2 \ldots n; 0 \ldots n/2] = D[n/2 \ldots n; 0 \ldots n/2]$, (to compute common suffixes). For each j, the diagonal through $PR[j, n/2]$ represents the alignment of the string $s_{j+1} \ldots s_{j+c}$ with $s_{n/2+1} \ldots s_{n/2+c}$. When searching for multiple repeats, we cannot simply record the position of every mismatch in the alignment. Several mismatches, if occurring in a single column in the multiple repeat, count as a single error. For example, consider the string $s_{n/2} \ldots s_n = abcgbcfbchbci$, with $j = n/2+3$ (the position of g). The comparisons begin with $s_{n/2+1}$ and s_{j+1}. In the previous algorithm, three mismatches were recorded, at positions $j+3$, $j+6$, and $j+9$. Since all three mismatches are in the same column of the multiple repeat, only the first one is relevant to the current algorithm.

The computation of PR follows the computation in section 2.2, however, a mismatch is recorded in PR only if it is the first mismatch in its column. We define $j^* = j - n/2$, where j^* is the period size of the repeats on the diagonal through $D[j, n/2]$. For each j, we use an array ($cols$) of length j^*, initialized to zeros, to store the columns that are marked as error columns. $cols[i] = 1$ if column i contains a mismatch. A mismatch is recorded in $PR[j+e, n/2+e]$ only if $cols[e \bmod j^*] = 0$. This ensures at most 1 recorded error per column in the repeat. (In the following algorithm, the additions to the previous computation appear in boldface.)

Computing PR

$e = 0$;
For $h = 0$ to k do
$PR[j + e, n/2 + e] = h$ (an auxiliary value when $h = 0$)
While $(j + e + 1 \leq n)$ and $((s_{j+e+1} = s_{n/2+e+1})$ **or** $(\mathbf{cols}[\mathbf{e} \bmod \mathbf{j}^*] = \mathbf{1}))$ do
$e = e + 1$; $PR[j + e, n/2 + e] = h$;
od
$P^*[j, h] = e$; $\mathbf{cols}[\mathbf{e} \bmod \mathbf{j}^*] = \mathbf{1}$; $e = e + 1$;
od

The modifications for computing SU are identical.

When P^* and S^* are computed using suffix trees, the modification can be done as a post-processing step in $O(nk)$ time. Assume that P^* and S^* are computed as explained in section 2.2; each array is of size $n/2 \times k$. We modify the arrays to contain only the first error position in each column, individually. For each row j, we initialize the array *cols* to zeros, and iterate through the k error positions stored in row j: if an error is the first in its column (i.e. the error occurs in column i and $cols[i] = 0$), we set $cols[i] = 1$, otherwise the error is deleted.

As a result of this processing, some rows may store fewer than k errors, and additional computation will be necessary to determine the k errors for each such row. In the worst case, the number of error positions that will be computed is $O(ka)$, where $r = u^{a-1}u'$ is the repeat with the maximum number of periods (over all j). However, in practice we have found that it suffices to initially compute ck errors, where c is a small constant, resulting in no penalty in the time complexity.

Part 2: Identifying the repeats

Each sub-diagonal through $D[j, n/2]$ of length *at least* j^* represents an approximate multiple repeat. We use the pair of integers, $[h', h]$, to denote the sub-diagonal that extends from the first position with the value h' to the last position with the value h (i.e. from row $j - S^*[j, h']$ to row $j + P^*[j, h]$). The substring of S that corresponds to the sub-diagonal $[h', h]$ is, $s_{n/2-S^*[j,h']+1} \cdots s_{j+P^*[j,h]}$. An error position in a repeat is called an extraneous error if it only adds extraneous characters to the repeat (see definition 8). The k-maximal multiple repeats correspond to all sub-diagonals $[h', h]$ for which $h' + h = k$, and h', h are not extraneous errors. The following algorithm uses the values in P^* and S^* to identify all k-maximal multiple repeats in S.

Identify the repeats

For $j = n/2 + 1$ to n do
$h' = k$; $h = k - h'$; $j^* = j - n/2$;
While $(h \leq k)$ do
if $(P^*[j, h] + S^*[j, h'] \geq j^*)$ and (error number h' is not extraneous)
then report the repeat (r, u) where $r = s_{n/2-S^*[j,h']+1} \cdots s_{j+P^*[j,h]}$, and $|u| = j^*$.
$h' = h' - 1$; // subtract one *critical* error from h'.
$h = h + 1$; // add one *critical* error to h.
od
od

Since P^* and S^* are computed separately, it is possible for a mismatch in P^* to be in the same column as a mismatch in S^*. If this is so, then although $h' + h = k$, there may be fewer than k column errors in the repeat. Therefore, between iterations, rather than simply adding one error to h and subtracting one from h', the addition and subtraction must be "intelligent," and only add/subtract a *critical* error. The $h + 1st$

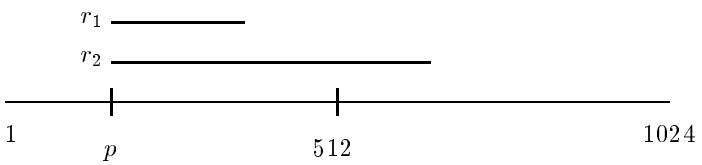

Figure 4: The 2 repeats (r_1, u_1) and (r_2, u_2) are found in different iterations, however, only the k-primitive ones are reported.

(resp. h'th) error in row j of P^* (resp. S^*) is called *critical* with regard to the pair $[h', h]$, if it is not in the same column as any of the first h' (resp. h) errors in row j of S^* (resp. P^*).

Another technical problem that must be dealt with is the inclusion of extraneous characters at the beginning and end of the repeat. A sub-diagonal $[h', h]$ for which error number h' is preceded only by characters in error columns, does not represent a k-maximal repeat. The extraneous characters must be removed and an additional error must be allowed in the opposite direction. This results in the pair $[h' - 1, h + 1]$, which is a pair that will be considered in the following iteration. Thus, the 'if' statement in the algorithm checks for an extraneous error in S^*, and simply skips all repeats in which the leftmost error is extraneous.

Similarly, a pair $[h', h]$ in which the hth error is extraneous, is equivalent to the previous iteration, $[h'+1, h-1]$. Such pairs however, are not skipped, since this may result in the omission of pairs with extraneous errors in both directions. Instead, before reporting a repeat, the extraneous errors are removed from the right end, and the test for k-primitivity (as explained in what follows) ensures that the same repeat is not reported twice.

Note that it takes $O(k)$ time to test whether h' is extraneous. Similarly, removing the extraneous characters from the end of a repeat may take $O(k)$ time. However, over all repeats with period j^*, the total work done will be $O(ka)$, where a is the maximum number of periods over all u^a.

Before reporting a k-maximal repeat, we must check whether it is k-primitive, that is, whether it has fewer errors than all smaller period repeats that span its substring/include it (see definition 10). Recall that repeats that span the same substring or include one another either begin or end at the same position. Thus, we can store information about previous repeats in a simple 1-D array according to their starting and ending positions. For each starting (ending) position, the length of the longest repeat found, and the minimum number of errors of all repeats found beginning (ending) at that position is recorded. Information on all repeats that are found, even those that are not reported, will be recorded in the array.

It is clear that within each substring a 1-D array ensures k-primitivity, since within each substring repeats are found in order of increasing period size. Now, consider two repeats (r_1, u_1) and (r_2, u_2), that are found in different iterations, and for which r_2 *includes* r_1 (see figure 4). Since the arrays are isolated to the individual substrings, the two repeats do not know about each other. However, we show that only the primitive ones will be reported.

- if $|u_1| < |u_2|$ then both repeats are k-primitive (by definition 10).
- if $|u_1| > |u_2|$ then, since r_1 is included in r_2, (r_1, u_1) should only be reported if it improves the error count of (r_2, u_2). This is exactly what happens, since a prefix of r_2 is found in the smaller substring, and although it is not reported (because it is not k-maximal), its information is stored in the array.

Theorem 1 *The algorithm finds all canonical k-repeats in an input string S.*

Proof: All k-maximal repeats in S cross the center of some substring in some iteration. This is clear in the recursive description of [ML84]). Every k-maximal repeat that spans the center characters of a substring can be represented by a pair of integers $[h', h]$. The algorithm "Identify the repeats" considers every possible pair $[h', h]$ with at most k error columns. (The only pairs that are skipped, are those that are considered in a different iteration.) Hence, the algorithm finds all k-maximal repeats within S. As shown, a 1-D array suffices to ensure k-primitivity, thus the reported repeats are canonical. □

Complexity: The time complexity of the algorithm for approximate multiple repeats is $O(nka \log(n/k))$, where a is the maximum over all repeats $r = u^a$.

5 The Program

The algorithm that we presented in the previous section to find all approximate multiple repeats within a string, has been implemented in 'C'[1]. The program has the following input and output.
INPUT: 1) a biological sequence S, 2) an integer k.
OUTPUT: all canonical k-repeats that occur in S.

In the implementation, P^* and S^* are computed using the algorithm of [LV86], with complexity $O(k(m \log m + n))$ where m is the length of the pattern and n is the length of the text. Since we call [LV86] with the pattern and text of length $O(n)$, the complexity of each iteration becomes $O(nk \log n)$. Although the theoretical time complexity of [LV89] is better, the suffix trees are difficult to program, and the practical runtime may not be significantly faster. As mentioned, we must compute ck positions of mismatches to comply with our definition of k errors. We tested the program for $k \leq 10$, and $c = 5$ proved to be adequate.

When dealing with very large sequences, as is the case with biological sequences, the runtime of the algorithm is proportional to the largest repeat that one wants to report. If n is the largest repeat that the algorithm is searching for, then the input string is searched in intervals of length $2n$. Thus, if N is the length of the input file, and n the largest repeat to search for, the algorithm that the program follows runs in $O(Nk \log n \log(n/k))$ time.

Although a repeat is allowed to have up to k errors, we impose an additional condition, which maintains that repeats are 70% error free (that is, at most 30% of the columns can be error columns). It would be meaningless to report a repeat with period 1, and 1 error. Thus, regardless of the value of k, all reported repeats with periods 1 and 2 do not contain any errors, repeats with periods 3,4 contain at most 1 error, and so on.

The final output is sorted according to entire length of each repeat. See the appendix for some sample output from a run of the program with the following input: 1: the human germline T-cell receptor β chain sequence (sections 1, 2 and 3) accession numbers U66059, U66060, U66061, (formerly corresponding to accession L36092), of length 685K,
2: $k = 2$.

6 Further Research

Our definition of approximate multiple repeats is built upon the measure of the Hamming distance. We chose this measure primarily for its simplicity. It would be interesting to search for a definition (and an algorithm) that is based upon the more commonly used measure of the edit distance. In such a case, an

[1] The program can be found at the site http://csweb.haifa.ac.il/~landau/library/.

alignment that corresponds to a repeat, may contain insertions and deletions (i.e. gap characters) within some periods of the repeat.

References

[Ap92] A. Apostolico. Fast Parallel Detection of Squares in Strings. *Algorithmica*, 8:285–319, 1992.

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[B95] G. Benson. A space-efficient algorithm for finding best scoring non-overlapping alignments. *Theoretical Computer Science*, 145:357–369, 1995.

[B97] G. Benson. Sequence alignment with tandem duplication. *J. Computational Biology*, 4:351–367, 1997.

[B99] G. Benson. Tandem repeats finder – a program to analyze DNA sequences. *Nucleic Acids Research*, 27:573–580, 1999.

[BS98] G. Benson and X. Su. On the distribution of k tuple matches for sequence homology: a constant time-exact calculation of the variance. *J. Computational Biology*, 5:87–100, 1998.

[BW94] G. Benson and M. Waterman. A method for fast database search for all k-nucleotide repeats. *Nucleic Acids Research*, 22:4828–4836, 1994.

[C92] C. T. Caskey et al. An unstable triplet repeat in a gene related to Myotonic Dystrophy. *Science*, 255:1256–1258, 1992.

[Doo90] R. F. Doolittle. Searching through sequence databases. *Methods in Enzymology*, 183:99–110, 1990.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[KM93] S. K. Kannan and E. W. Myers. An Algorithm for Locating Nonoverlapping Regions of Maximum Alignment Score. In *Proc. Fourth Combinatorial Pattern Matching Conference*, Lecture Notes in Computer Science 684, pages 74–86. Springer-Verlag, 1993.

[KM96] S. K. Kannan and E. W. Myers. An Algorithm for Locating Nonoverlapping Regions of Maximum Alignment Score. *SIAM J. Comput.*, 25(3):648–662, 1996.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

[Les89] A. M. Lesk. *Computational Molecular Biology.* Oxford University Press, 1989.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Phys. Dokl*, 10:707–710, 1966.

[LMS98] G. M. Landau, E. W. Myers and J. P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.

[LS93] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proc. Fourth Combinatorial Pattern Matching Conference*, Lecture Notes in Computer Science 684, pages 120–133. Springer-Verlag, 1993.

[LV86] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[LV88] G. M. Landau and U. Vishkin. Fast string matching with k differences. *JCSS*, 37:63–78, 1988.

[LV89] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[ML84] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.

[M92] W. Miller. An algorithm for locating a repeated region. *manuscript*, 1992.

[ML85] M. G. Main and R. J. Lorentz. Linear time recognition of square free strings. A. Apostolico and Z. Galil (editors), *Combinatorial Algorithms on Words*, NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, Springer-Verlag, 272–278, 1985.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[RKH96] L. Rowan, B. Koop, L. Hood. The complete 685-kilobase DNA sequence of the Human β T-cell receptor locus. *Science*, 272:1755–1768, 1996.

[S98] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27(4):972–992, 1998.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[SPIS99] J. S. Sim, K. Park, C. S. Iliopoulos, W. F. Smyth. Approximate Periods of Strings. In *Proc. Tenth Combinatorial Pattern Matching Conference*, Lecture Notes in Computer Science 1645, pages 123–133. Springer-Verlag, 1999.

[U83] E. Ukkonen. On approximate string matching. *Proc. Int. Conf. Found. Comp. Theor.*, Lecture Notes in Computer Science 158, Springer-Verlag, 487-495, 1983.

Appendix - Results

This appendix shows some sample repeats reported when the program was run on the following input: 1: the human germline T-cell receptor β chain sequence (sections 1, 2 and 3) accession numbers U66059, U66060, U66061, (formerly corresponding to accession L36092), of length 685K,
2: $k = 2$.

```
110 characters covered by repeat of size 31,
    repeated  3.5 times with 2 error(s)

377636 tatatatatatacactctcctatatatatag 377666
                     |       |
377667 tatatatatatacactctcctatatatatag 377697
                     |       |
377698 tatatatatatacactctcctgtatatatag 377728
                     |
377729 tatatatatatactctc               377745

107 characters covered by repeat of size 28,
    repeated  3.8 times with 1 error(s)

198007 ataatatagaatatagaatatatattct 198034
                                   |
198035 ataatatagaatatagaatatatattat 198062
                                   |
198063 ataatatagaatatagaatatatattat 198090

198091 ataatatagaatatagaatatat      198113

80 characters covered by repeat of size 7,
   repeated 11.4 times with 2 error(s)

198034 tataata 198040
         | |
198041 tagaata 198047
         | |
198048 tagaata 198054
         | |
198055 tatatta 198061
         | |
198062 tataata 198068
         | |
198069 tagaata 198075
         | |
198076 tagaata 198082
         | |
198083 tatatta 198089
         | |
198090 tataata 198096
         | |
198097 tagaata 198103
         | |
198104 tagaata 198110
```

```
198111 ta      198112

22 characters covered by repeat of size 1,
   repeated 22.0 times

403893 ttttttttttttttttttttt 403914

21 characters covered by repeat of size 6,
   repeated  3.5 times with 1 error(s)

427545 gggaca 427550
        |
427551 gagaca 427556
        |
427557 gggaca 427562
        |
427563 ggg    427565
```