BIOINFORMATICS

Vol. 23 ECCB 2006, pages e30–e35
doi:10.1093/bioinformatics/btl309

Tandem repeats over the edit distance

Dina Sokol[1,*], Gary Benson[2] and Justin Tojeira[1]

[1]Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY, USA and [2]Departments of Biology and Computer Science, Boston University, Boston, USA

ABSTRACT

Motivation: A tandem repeat in DNA is a sequence of two or more contiguous, approximate copies of a pattern of nucleotides. Tandem repeats occur in the genomes of both eukaryotic and prokaryotic organisms. They are important in numerous fields including disease diagnosis, mapping studies, human identity testing (DNA fingerprinting), sequence homology and population studies. Although tandem repeats have been used by biologists for many years, there are few tools available for performing an exhaustive search for all tandem repeats in a given sequence.

Results: In this paper we describe an efficient algorithm for finding all tandem repeats within a sequence, under the edit distance measure. The contributions of this paper are two-fold: theoretical and practical. We present a precise definition for tandem repeats over the edit distance and an efficient, deterministic algorithm for finding these repeats.

Availability: The algorithm has been implemented in C++, and the software is available upon request and can be used at http://www.sci.brooklyn.cuny.edu/~sokol/trepeats. The use of this tool will assist biologists in discovering new ways that tandem repeats affect both the structure and function of DNA and protein molecules.

Contact: sokol@sci.brooklyn.cuny.edu

1 INTRODUCTION

A tandem repeat, or tandem array, in DNA, is a sequence of two or more contiguous, approximate copies of a pattern of nucleotides. Tandem repeats appear in biological sequences with a wide variety. They are important in numerous fields including disease diagnosis, mapping studies, human identity testing (DNA fingerprinting), sequence homology and population studies.

Most genomes have a high content of repetitive DNA. Repeated sequences make up 50% of the human genome (Collins *et al*., 2003). The repeats in the human genome are important as genetic markers (Kannan and Myers, 1996), and they are also responsible for a number of inherited diseases involving the central nervous system. For example, in a normal FMR-1 gene, the triplet CGG is tandemly repeated 6 to 54 times, while in patients with Fragile X Syndrome the pattern occurs >200 times. Kennedy disease and Myotonic Dystrophy are two other diseases that have been associated with triplet repeats (Frazier *et al*., 2003). In addition, tandem repeats are used in population studies (Uform and Wayne, 1993), conservation biology (The International Human Genome Mapping Consortium, 2001) and in conjunction with multiple sequence alignments (Benson, 1997; Kolpakov and Kucherov, 2001).

Many of the repeats appear in non-coding regions of DNA. Although useful for identity testing, these regions were thought to carry no function. Recently, however, it has been shown that repeats in the genome of a rodent provide code for its sociobehavioral traits (Jeffreys, 1993). Scientists currently believe that the non-coding tandem repeats do affect the function of the DNA in ways yet unknown.

Although tandem repeats have been used by biologists for many years, there are few tools available for performing an exhaustive search for all tandem repeats in a given sequence, while allowing for mutations. Due to the recent sequencing of the human genome by the Human Genome Project (Groult *et al*., 2003; Fu *et al*., 1992), it is now possible to analyze the sequence of the human genome and create a listing of all tandem repeats in the genome. Detecting all tandem repeats in protein sequences is an important goal as well (Kitada *et al*., 1996).

In this paper we describe an efficient algorithm for finding all tandem repeats within a sequence. We have already implemented the algorithm in C++, and a website for the program is under construction. Our program automates the task of listing all repeats that occur in a biological sequence. It is our hope that with the use of our program, biologists will discover new ways that tandem repeats affect both the structure and function of DNA and protein molecules.

The contributions of this paper are 2-fold. The theoretical contributions consist of an extension of the algorithm of Landau and Schmidt (Landau *et al*., 1998) to locate evolutive tandem repeats over the edit distance and a more careful analysis of the algorithm eliminating the need for suffix trees. The practical contribution is an efficient program for finding tandem repeats that will become available on the web for all to use.

1.1 Problem definition

We define tandem repeats over the edit distance using the model of evolutive tandem repeats (Groult *et al*., 2004; Hammock and Young, 2005). The model assumes that each copy of the repeat, from left to right, is derived from the previous copy through zero or more mutations. Thus, each copy in the repeat is similar to its predecessor and successor copy.

Let $ed(s_1, s_2)$ denote the minimum edit distance between two strings, s_1 and s_2.

DEFINITION 1. *A string R is a k-edit repeat if it can be partitioned into consecutive substrings,* $R = r_1, r_2, \ldots, r_\ell$, *such that* $\sum_{i=1}^{n-1} ed(r_i, r_{i+1}) \leq k$. *The last copy of the repeat does not have to be complete, and therefore ed* $(r_{\ell-1}, r_\ell)$ *refers to the edit distance between a prefix of* $r_{\ell-1}$ *and* r_ℓ.

A k-edit repeat is an evolutive tandem repeat in which there are at most k insertions, deletions and mismatches, overall, between all consecutive copies of the repeat. For example, the string $R = caagct\,|\,cagct\,|\,ccgct$ is a 2-edit repeat. A k-edit repeat is maximal

*To whom correspondence should be addressed.

if it cannot be extended to the right or left. Maximal repeats can be overlapping, as shown in the following example.

EXAMPLE. Repeat of length 14 with 2 errors:

```
157  CA-CAGG  162
163  CACCGGG  169
170  C        170
```

Repeat of length 18 with 2 errors:

```
166  CGGGCTGC-  173
174  CGGCCTGCA  182
183  C          183
```

In this paper, we address the *k*-edit Repeat Problem, defined as follows. Given a string, *S*, and an integer *k*, find all maximal *k*-edit repeats in *S*. From the theoretical viewpoint, we provide an efficient algorithm that locates all maximal *k*-edit repeats in a string. We implemented this algorithm and found that our program locates many repeats that are similar one to another. Hence, we incorporated several heuristics in our program, to make the output more succinct and useful. The heuristics are described in Section 2.3. The result is a concise listing of the *k*-edit repeats occurring in the input sequence.

1.2 Related work

The early work on finding tandem repeats in strings dealt with simple repeats, i.e. repeats that contained exactly two parts. Kannan and Meyers (Katti *et al*., 2000), Benson (Benson, 1995) and Schmidt (Spong and Hellborg, 2002) present algorithms for finding simple repeats using the weighted edit distance.

The true goal of biologists, which turns out to be a much more difficult problem, is to find all maximal repeats in a string. A maximal repeat within a string is a repeat that contains two or more consecutive copies, and it cannot be extended further to the left or to the right. Each part of the repeat is called a period. When considering maximal repeats without errors, all periods (except possibly the last one) have equal length. For example, in the string *aatgtgtgt* the string *tgtgtgt* is a maximal repeat with 3.5 periods.

More recently, the concentration has been on searching for maximal repeats with errors. In this case, however, the definition is not obvious. Given a repeat with several periods, what does it mean for the parts to be 'similar?' Benson (Benson, 1999) requires that a consensus string must exist which is similar to all periods of the repeat. Using this approach, it is difficult to provide a deterministic algorithm to find tandem repeats. Hence, Tandem Repeats Finder (TRF), developed by Benson[1] uses a collection of statistical criteria in combination with *k*-tuple matches to detect statistically significant tandem repeats. Similar criteria are used by Wexler *et al*. (2004).

The goal of this paper is to provide a rigorous definition of tandem repeats and to provide a deterministic algorithm to detect all repeats that satisfy the definition. All of the algorithms that use this approach build upon the Hamming distance, which measures the number of mismatching characters between two strings, maintaining the property that each period of a repeat has the same length.

[1]The program and Tandem Repeats Database are available at http://tandem.bu.edu/TRDB.html.

In (Groult *et al*., 2004; Hammock and Young, 2005), evolutive tandem repeats are defined over the Hamming distance as follows. A string is an evolutive tandem repeat if it can be broken up into substrings $r_1 \ldots r_\ell$ such that the Hamming distance between r_i and r_{i+1} is smaller than a given threshold, for $1 \leq i < \ell$. The definition also allows for a small gap between copies of the repeat. The algorithm presented in (Hammock and Young, 2005) has worst-case quadratic runtime.

Kolpakov and Kucherov (Kolpakov *et al*., 2003, http://www.loria.fr/mreps/) present two different definitions. The first definition sums the mismatches between all neighboring copies of the repeat. Formally, a repeat $r[1..n]$ is called a *k-repetition* if the Hamming distance between $r[1..n-p]$ and $r[p+1..n]$ is $\leq k$. The second definition allows *k* mismatches between each pair of consecutive copies, and this is called a *k*-run, and is similar to the evolutive tandem repeats over the Hamming distance. The algorithm of (Kolpakov *et al*., 2003, http://www.loria.fr/mreps/) has $O(nk \log k)$ time. The algorithm has been implemented in the `mreps` software (Landau).

Landau, Schmidt and Sokol (Landau *et al*., 2001) define a tandem repeat to have *k* mismatches if the alignment constructed from its periods has *k* nonuniform columns. Their algorithm runs in $O(nka \log (n/k))$ where *n* is the sequence, *k* is the error bound and *a* is the maximum number of periods in any reported repeat. The algorithm has been implemented and is available at (Landau and Vishkin, 1989 http://csweb.cs.haifa.ac.il/library/ and http://www.sci.brooklyn.cuny.edu/~sokol/trepeats/).

The disadvantage of the Hamming distance is that it only accounts for point mutations and does not allow insertions and deletions. To model mutations more generally, it is preferable to use the edit distance, defined by Levenshtein (Main and Lorentz, 1984) as the minimum number of insertions, deletions and substitutions necessary to transform one string into the other. Ideally, we would like to differentially weight mutations; substitutions are typically scored more permissively than insertions and deletions. In this paper, we weight all differences uniformly, i.e. we use an edit distance scheme.

2 METHODS

In this section we describe the algorithm for finding *k*-edit repeats. We first describe a straightforward solution, following which we describe three speedups to achieve a time and space efficient algorithm. The speedups use similar ideas to those in (Landau *et al*., 2001).

2.1 A straightforward solution

The classical method for calculating the edit distance between two strings $S = s_1 \ldots s_m$ and $S' = s'_1 \ldots s'_n$ is to construct an $n \times m$ dynamic programming matrix, to initialize (row 0, column *j*) to *j* (column 0, row *i*) to *i*, and to fill it in for $i, j > 0$ using the following formula.

$$\text{edit\_dist}[i,j] = \text{MIN}\begin{cases} \text{edit\_dist}[i-1,\ j]+1, \\ \text{edit\_dist}[i,\ j-1]+1, \\ \text{edit\_dist}[i-1,\ j-1]+0 \quad \text{if } s_i = s'_j, \\ \text{edit\_dist}[i-1,\ j-1]+1 \quad \text{if } s_i \neq s'_j \end{cases}$$

We define a *p*-restricted edit distance alignment as an edit distance alignment between two strings that disallows *p* insertions into the first string (alternatively, *p* deletions in the second string). This definition is necessary, since

we would like to align a prefix of a string with a proper suffix of the same string. If insertions into the prefix are not restricted, then the string may eventually 'catch up' with itself, aligning characters in the prefix with the exact corresponding characters in the suffix.

In terms of the edit distance matrix, if we assume that the first input string is to the left of the matrix, a p-restricted alignment corresponds to the regular edit distance matrix, allowing only $p-1$ diagonals to the left of the main diagonal.

LEMMA 1. *A string R of length q is a k-edit repeat if and only if a p-restricted edit distance alignment can be constructed between a suffix of R, beginning at location $p+1$, and a prefix of R, with $\leq k$ differences, for some $p < q/2$.*

PROOF. The proof follows from the fact that we can obtain a k-edit repeat from a 2-sequence alignment of the prefix and suffix, and alternatively, given a k-edit repeat, the prefix/suffix can be derived from the alignment of the repeat. Given a p-restricted alignment of a suffix/prefix of a string R, the copies of the repeat can be derived from the alignment as follows. The first copy consists of the first p characters of the prefix. The second copy consists of all characters in the suffix that appear in the alignment against the first p characters in the prefix. Each successive copy is calculated by using the previous known copy in the prefix to demarcate the following copy in the suffix.

If a string R is a k-edit repeat, the prefix/suffix alignment can be obtained from the copy-to-copy alignments. The suffix can be taken as the start of the second copy of the repeat. The prefix is the string R minus the last copy of the repeat. Each has length $\geq q/2$ since only one copy of the repeat is removed. The prefix/suffix alignment can be constructed by aligning copy i in the prefix with copy $i+1$ in the suffix.

We can use lemma 1 to derive a straightforward algorithm for finding all k-edit repeats within a string. The algorithm finds all substrings of the string for which a 2-sequence alignment exists between a proper prefix and a proper suffix of the substring. It then breaks up the 2-sequence alignment into its copies using the method in the first part of the proof of the lemma.

Simple Algorithm

Input: A string S, and an integer k.

Output: All k-edit repeats in S.

(1) Consider each index $1 \leq i < n$ as the starting point of a potential repeat.

(2) Consider each index $i < j \leq \frac{n+i}{2}$ (to the right of location i) as the start of the proper suffix of the repeat.

(3) For each pair (i, j) perform a $(j-i)$-restricted alignment of the two strings $S_1 = s_i \ldots s_n$ and $S_2 = s_j \ldots s_n$ using the classical dynamic programming method (allowing only $j-i-1$ diagonals to the left of the main diagonal).

(4) If there is a match between S_1 and S_2, of length at least $j-i$, with $\leq k$ errors, then a repeat exists, beginning at location i and ending at the location before the $k+1$st error found in step 3.

Complexity analysis: The time complexity of the naive algorithm is $O(n^4)$. There are $O(n^2)$ iterations, and in each iteration we compute the dynamic programming matrix of size $O(n^2)$. The space used is that of the edit distance matrix, which is of size $O(n^2)$.

In the following section we explain the efficient algorithm. We present it as a three-tier modification to the simple algorithm. The new algorithm reduces the time to $O(nk \log k \log(n/k))$ and it reduces the space from $O(n^2)$ to $O(k^2)$.

2.2 An efficient algorithm

We present a three-tier speedup to the naive algorithm. First, we reduce the number of iterations. Then, we show how each iteration can be improved by pruning the edit distance matrix. The third speedup fine-tunes the computation time of the partial edit distance matrix.

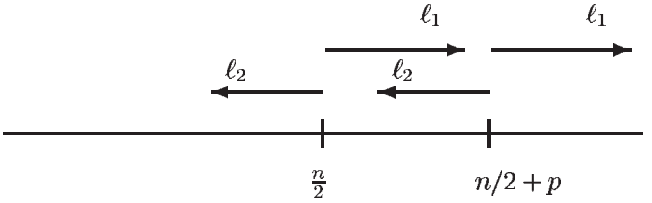

Fig. 1. Computing repeats: ℓ_1 is the length of the forward direction comparisons with $k_1 < k$ errors, and ℓ_1 is the length of the backward direction comparison with $k_2 < k$ errors. If $k_1 + k_2 \leq k$ and $\ell_1 + \ell_2 \geq p$, then there is a k-edit repeat at $s_{n/2-\ell_2+1} \ldots s_{n/2+\ell_1}$.

Speedup #1: *reduce the number of iterations.* We use the idea[2] of Main and Lorentz (Schmidt) to reduce the number of iterations from $O(n^2)$ to $O(n\log(n/k))$. The algorithm, rather than considering all possible starts, anchors the comparisons at the center of the string. In the first iteration, the input is $S = s_1, \ldots, s_n$, and all repeats that cross the center of the string (i.e. include character s_{n2}) are found. In the following iteration, S is divided into two substrings, $S = uv$, $u = s_1 \ldots s_{n/2}$ and $v = s_{n/2+1} \ldots s_n$. The algorithm which finds repeats crossing the center is applied recursively to both u and v.

In order to locate all repeats that include the character $s_{n/2}$, it is necessary to consider all alignments in which $s_{n/2}$ corresponds to another character in the string. Following we describe the procedure which aligns location $n/2$ with all indices $p > n/2$. By symmetry, alignments for values $p < n/2$ can be produced.

Find repeats

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

(1) Forward Direction: Find the longest prefix of $s_{n/2+p} \ldots s_n$ that p-restricted matches a prefix of $s_{n/2} \ldots s_n$ with $\leq k$ errors. (Fig. 1).

(2) Backward Direction: Find the longest suffix of $s_1 \ldots s_{n/2+p-1}$ that p-restricted matches a suffix of $s_1 \ldots s_{n/2-1}$ with $\leq k$ errors.

(3) Consider all pairs k_1, k_2 for which $k_1 + k_2 = k$. Let ℓ_1 be the length of the backward direction comparison with k_1 errors, and ℓ_2 be the length of the forward direction comparison with k_2 errors. If $\ell_1 + \ell_2 \geq p$ then there is a tandem repeat extending from $s_{n/2-\ell_2+1} \ldots s_{n/2+\ell_1}$

We illustrate this idea with the following example.

Let $k = 4$, find the k-edit repeats in the string $S = ctcgagctcctgacctcgtga$.

We show the iteration of $p = 6$, assuming that the first appearance of 't' in the string is location $n/2$. We omitted the first part of the string since it is not necessary for the example. The forward direction comparison is done by aligning the string $s_{n/2} \ldots sn$ with the string $s_{n/2+6} \ldots sn$. The optimal alignment (i.e. obtaining the minimum edit distance) is shown in Figure 2. Two gaps are introduced, and there are two mismatches.

The length of the forward extension with 4 errors is 14 (i.e. the alignment extends up until location $n/2+6+14-1$). Since we allowed 4 errors in the forward extension, and $k = 4$, we do not allow any errors in the backward extension. The length of the backward extension with 0 errors is 1 (the single character c).

The copies of the repeat are derived from the 2-sequence alignment as follows:

Repeat of length 21 with 4 errors.

```
Copy 1: CTC--GAG
copy 2: CTCCTGAC
copy 3: CTCGTGA
```

[2]Main and Lorentz use $O(\log\ n)$ iterations, and we state that there are $O(\log(n/k))$ iterations, since we deal with strings of length $\leq$k separately, using a straightforward algorithm.

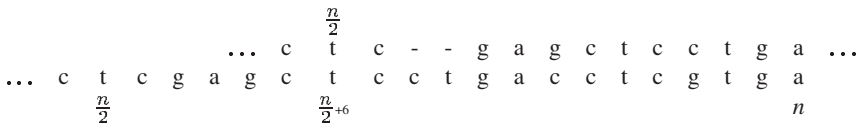

Fig. 2. Optimal alignment obtaining the minimum edit distance.

Speedup #2: *Reduce the Size of the Dynamic Programming Matrix*. In the straightforward algorithm, the forward and reverse direction extensions are computed by building the dynamic programming edit distance matrix for each pair of substrings. The second idea for speeding up the algorithm uses the observation that it is not necessary to compute the entire edit distance matrix, since we are only looking for k errors.

Using the ideas of Ukkonen (1983) and Landau/Vishkin (Levenshtein, 1966), we can reduce the size of the matrix from $O(n^2)$ to $O(k^2)$. Consider the diagonals in the edit distance matrix, where diagonal d corresponds to the diagonal with indices $\{(i, j) | i - j = d\}$. The main diagonal is diagonal 0. Any diagonal in the edit distance matrix that has distance $>k$ from diagonal 0 is not of interest since all of its values will be $>k$. Thus, it is only necessary to compute the values on $2k + 1$ diagonals. Furthermore, on a given diagonal, successive values differ by at most 1. Therefore, it is only necessary to store the location of the last row on each diagonal that has value h for each $0 \leq h \leq k$. Instead of the *edit dist* matrix, we compute a $k \times k$ matrix L such that:

$L[d, h]$ = largest row in the *edit_dist* matrix on diagonal d that has the value h.

Analysis: The matrix L can be computed by filling in $n \times 2k$ entries in the dynamic programming matrix. Each row needs $2k$ values, and potentially n rows will have to be computed. Using the classical dynamic programming edit distance algorithm (as in Section 2.1), this can be done in $O(nk)$ time. Only a constant number of rows need be stored, hence the space complexity is $O(k^2)$. This looks excellent, however, consider the fact that the matrix L needs to be computed, in a given iteration, for each possible $1 \leq p < n/2$. This results in the matrix being computed potentially $O(n)$ times, resulting in $O(n^2k)$ time per iteration. This is where the third speedup comes into play.

Speedup #3: *reduce the computation time of the dynamic programming matrix*. The third speedup uses information from one period size for the following period size. In each iteration, the algorithm computes the edit distance matrix separately for each period $1 \leq p \leq n/2$. For $p = 1$, this translates into the computation of an edit distance matrix for the two strings $s_{n/2} \ldots s_n$ and $s_{n/2+1} \ldots s_n$. For $p = 2$, a new matrix is computed for the strings $s_{n/2} \ldots s_n$ and $s_{n/2+2} \ldots s_n$, and so on. The only change from one period size to the next is the deletion of the first character in the second string. There is a lot of overlapping computation between different period sizes. Landau, Myers and Schmidt (LMS) (Landau and Schmidt, 1993) called this problem 'incremental string comparison', and showed how to get from one matrix to the next in $O(k)$ time, assuming we are only searching for up to k errors.

Thus, we can compute the first $O(k^2)$ matrix using the previous speedup (Levenshtein, 1966), and then for each period size spend $O(k)$ time using LMS to modify the matrix.

The only remaining issue is that when using the LMS algorithm to update the matrix it is not trivial to figure out the values for ℓ_1 and ℓ_2, i.e. the longest common extension forward and backward. These values correspond to the rightmost column in the dynamic programming matrix, for which a certain value $k_1 < k$ appears. In (Landau *et al*., 2001) this is solved by maintaining a heap for each value $k_1 < k$, which contains all columns having the value k_1. Thus, the rightmost column can be located and updated in $O(\log k)$ time.

Complexity analysis: There are $O(\log (n/k))$ iterations. In each iteration, we construct a matrix of size $O(k^2)$ in time $O(k^2 + nk)$. The matrix is modified for each period size in $O(k \log k)$ time. Overall, each iteration takes $O(nk \log k)$ time, and the algorithm has time complexity $O(nk \log k \log(n/k))$. The space complexity is $O(n + k^2)$.

2.3 Implementation

We have implemented the program in C++ including the first two speedups. The third speedup is complicated, and it is our guess that in practice it will not significantly affect the program. This is based on the observation that in practice we almost never compute the full $n \times k$ matrix, as we stop calculating each of the diagonals after $k + 1$ errors are found. Our program can process a string of several thousand characters in a fraction of a second. Currently, the running time of the program is proportional to the time it takes to print the output.

An inherent flexibility exists in the definition of approximate repeats, with the goal of allowing for mutations. This flexibility poses an issue when the goal is to detect all substrings of the input string that satisfy the definition. The issue is that in practice there is too much output. Each repeat that is reported satisfies the mathematical definition; however, many reported repeats differ only slightly one from another. Hence, our challenge was to modify our algorithm in a way that it reports a meaningful and significant subset of the found repeats. To this end, we filter out all repeats that prove to be redundant. The filtering is incorporated into the algorithm itself, which is much more efficient than doing a post-processing phase to filter the output.

We deal with the issue of filtering by augmenting the different phases of our algorithm. The first filtering technique filters repeats found within an individual iteration, while the second technique filters repeats found in different iterations.

Filtering a given iteration: 1. *Choose the Optimal Period using the Local Maximum*. The first point of comparison is anchored at the center position of the string, but the second point varies, and only by one character at a time. Suppose there exists an alignment for some p with h errors. Then, this alignment can be converted into an alignment for $p + 1$ with at most $h + 1$ errors. Thus, we do not want to report every possible k-edit repeat. We are only interested in the value p that produces the best matching of the surrounding string segments. This leads to a local maximum idea. As the first point is anchored, and the second point moves forward, the comparison produces better matches as the second point nears a corresponding position in a different part of the repeat. The comparison gives worse results as the second point moves away from that 'optimal' position, until it begins nearing the corresponding position of another repeat.

2. Combine several neighboring repeats. Once a local maximum, p, is found, we have potentially k different repeats with period p, each shifted over several characters. This is because we consider k_1 errors to the left and k_2 errors to the right, for all values of k_1, k_2 such that $k_1 + k_2 = k$. For purposes of reporting the repeats, it is much clearer to see the shifting repeats as a single repeat. Hence, in this case, we combine all repeats with period p into a single repeat. We note that the combined repeat has at most $2k$ errors.

Filtering between iterations: The idea for filtering between iterations is a simple one. If a repeat reaches either end of the string segment being examined by the current iteration, it will also be detected by a different iteration at a higher level in the recursion and thus need not be reported in the current iteration.

3 RESULTS

The table on the left in Figure 3 contains a summary of the repeats that our program detected in the first 14 000 bp of human chromosome 18 (May 2004). The exact filtering criteria are still being improved, and thus we manually filtered the repeats to clarify

K-edit Repeats				
Start	End	Length	Errors	Period
1	633	633	49	6
795	**885**	91	12	44
929	1147	219	6	29
5017	5082	66	3	32
10690	**10789**	100	15	50
13119	13221	103	9	42

TRF				
Start	End	Length	Errors	Period
1	631	631	41	6
929	1129	201	6	29
5017	5082	66	3	32
13119	13223	105	13	14

Fig. 3. The repeats found by our program are compared and contrasted with the repeats found by TRF (Benson, 1999).

Repeat of length 91, from beg pos 795 to end pos 885 with 12 errors.

```
795 CCGG--GTCTGTGCTGAGGAGAACGCTGCTCCGCCTCCGCGGTACT   838
839 CCGGACATCTGTGCAGAGAAGAACGCAGCTGCGCCCTCGCCATGCT   884
885 C                                 885
```

Repeat of length 100, from beg pos 10690 to end pos 10789 with 15 errors.

```
10690 TTCACAGCAGAATTCTACCAGACATTCAAAGAAGAAATGATACCAATCCT   10739
10740 TTCACA-CT-A-TTCCACAAGACAGAGAAAGAAGAAACCCTTCCAATTCA   10786
10787 TTC                              10789
```

Fig. 4. Alignments of the copies of two new repeats found by our program.

results. The criterion for this run allows up to $k = 40$ errors and requires a minimum length of at least 100 characters more than the number of errors. This particular criterion was tested on pseudorandom strings created using the C++ function rand( ), which produces statistically random output in a single run. The function rand( ) was initialized using the system clock time, ensuring a different set of statistically random strings for each execution. It was tested on 20 strings, each 20 000 in length, and returned a single repeat in 6 of them, and no repeats in the rest. The table on the right of Figure 3 shows the results of TRF (Benson, 1999) on the same sequence.

In Figure 4 we show the alignments of the copies of the two new repeats found by our program, beginning at positions 795 and 10 690.

4 DISCUSSION

Many of the repeats that are found by our program are also found by TRF (Benson, 1999), which uses a non-evolutive definition. This is due to the fact that the consensus type repeat often satisfies the evolutive definition as well. Sometimes, our program will detect more errors than TRF, because in the case of a deviation in a single period, we count each deviation twice: once from the period before to the period with the deviation and once again from the period with the deviation to the period after. However, more often, changes will occur and be repeated for several periods before they change again. It is in these situations that the evolutive definition is most applicable. A non-evolutive definition causes an error to be reported for each period in which the deviation occurs, whereas an evolutive definition counts it as only one error until it is changed again.

We conclude that our novel definition and efficient program provide a useful tool for analyzing whole genomes. It is our hope that with its widespread use new scientific discoveries and inferences will be achieved.

Conflict of Interest: none declared.

ACKNOWLEDGEMENTS

This work was partially supported by NSF grant DBI-0542751 to J.T. and D.S., and by PSC-CUNY grant 67217-0036 to J.T.

Conflict of Interest: none declared.

REFERENCES

Benson,G. (1995) A space efficient algorithm for finding best scoring non-overlapping alignments. *Theor. Comput. Sci.*, **145**, 357–369.

Benson,G. (1997) Sequence alignment with tandem duplication. *J. Comput. Biol.*, **4**, 351–367.

Benson,G. (1999) Tandem repeats finder—a program to analyze DNA sequences. *Nucleic Acids Res.*, **27**, 573–580.

Collins,F.S. *et al.* (2003) The Human Genome Project: lessons from large-scale biology. *Science*, **300**, 286–290.

Frazier,M.E. *et al.* (2003) Realizing the potential of the genome revolution: the genomes to life program. *Science*, **300**, 290–293.

Fu,Y.H. *et al.* (1992) An unstable triplet repeat in a gene related to myotonic muscular dystrophy. *Science*, **255**, 1256–1258.

Groult,R., Leonard,M. and Mouchard,L. (2003) Speeding up the detection of evolutive tandem repeats. In *Proceedings of The Prague Stringology Conference '03*.

Groult,R. *et al.* (2004) Speeding up the detection of evolutive tandem repeats. *Theor. Comput. Sci.*, **310**, 309–328.

Hammock,E. and Young,L.J. (2005) Microsatellite instability generates diversity in brain and sociobehavioral traits. *Science*, **308**, 1630–1634.

Jeffreys,A.J. (1993) DNA typing: approaches and applications. *J. Forensic Sci. Soc.*, **33**, 204–211.

Kannan,S.K. and Myers,E.W. (1996) An algorithm for locating regions of maximum alignment score. *SIAM J. Comput.*, **25**, 648–662.

Katti,M.V. *et al.* (2000) Amino acid repeat patterns in protein sequences: their diversity and structural-functional implications. *Protein Sci.*, **9**, 1203–1209.

Kitada,H. *et al*. (1996) Multiple alignment of biological sequences containing tandem repeats. *Genome Inform.*, **7**, 276–277.

Kolpakov,R. and Kucherov,G. (2001) Finding approximate repetitions under hamming distance. In *Proceedings of the 9th European Symposium on Algorithms (ESA), Lecture Notes in Computer Science,* Vol. 2161, pp. 170–181.

Kolpakov,R *et al*. (2003) mreps: Efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Res*., **31**, 3672–3678.

Landau,G.M. and Vishkin,U. (1989) Fast parallel and serial approximate string matching. *J. Algorithm*., **10**, 157–169.

Landau,G.M. and Schmidt,J.P. (1993) An algorithm for approximate tandem repeats. In *Proceedings of the. 4th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science,* Springer-Verlag, Vol. 648, pp. 120–133.

Landau,G.M. *et al*. (1998) Incremental string comparison. *SIAM J. Comput*., **27**, 557–582.

Landau,G.M. *et al*. (2001) An algorithm for approximate tandem repeats. *J. Comput. Biol*., **8**, 1–18.

Levenshtein,V.I. (1966) Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl*., **10**, 707–710.

Main,M.G. and Lorentz,R.J. (1984) An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithm*., 422–432.

Schmidt,J.P. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput*., **27**, 972–992.

Spong,G. and Hellborg,L. (2002) A near-extinction event in lynx: do microsatellite data tell the tale? *Conserv. Ecol*., **6**, 15.

The International Human Genome Mapping Consortium. (2001), Initial sequencing and analysis of the human genome. *Nature* [Erratum (2001) *Nature*, 411, 720]. **409**, 860–921.

Uform,M.W. and Wayne,R.K. (1993) Microsatellites and their application to population genetic studies. *Curr. Opin. Genet. Dev*., **3**, 939–943.

Ukkonen,E. (1983) On approximate string matching. In *Proceedings of the International Conference Foundations of Computation Theory, Lecture Notes in Computer Science 158,* Springer-Verlag, pages 487–495.

Wexler,Y., Yakhini,Z., Kashi,Y. and Geiger,D. (2004) Finding approximate tandem repeats in genomic sequences. In *Proceedings of the 8th Annual Conference on Research in Computational Molecular Biology (RECOMB).*