Double String Tandem Repeats

Amihood Amir
Department of Computer Science, Bar Ilan University, Ramat-Gan, 52900, Israel
http://u.cs.biu.ac.il/~amir/
amir@esc.biu.ac.il

Ayelet Butman
Department of Computer Science, Holon Institude of Technology, Golomb St 52, Holon, 5810201, Israel
ayeletb@hit.ac.il

Gad M. Landau
Department of Computer Science, University of Haifa, Haifa 31905, Israel
NYU Tandon School of Engineering, New York University, Six MetroTech Center, Brooklyn, NY 11201, USA
http://www.cs.haifa.ac.il/~landau/
landau@univ.haifa.ac.il

Shoshana Marcus
Department of Mathematics and Computer Science, Kingsborough Community College of the City University of New York, 2001 Oriental Boulevard, Brooklyn, NY 11235
shoshana.marcus@kbcc.cuny.edu

Dina Sokol
Department of Computer and Information Science, Brooklyn College and The Graduate Center, City University of New York
http://www.sci.brooklyn.cuny.edu/~sokol
sokol@sci.brooklyn.cuny.edu

Abstract

A *tandem repeat* is an occurrence of two adjacent identical substrings. In this paper, we introduce the notion of a *double string*, which consists of two parallel strings, and we study the problem of locating all tandem repeats in a double string. Double strings are ubiquitous in nature, as molecules such as DNA and RNA come in pairs. However, the problem introduced here has applications beyond actual double strings, as we illustrate by solving two different problems with the algorithm of the double string tandem repeats problem. The first problem is that of finding all corner-sharing tandems in a 2-dimensional text, defined by Apostolico and Brimkov. The second problem is that of finding all scaled tandem repeats in a 1d text, where a scaled tandem repeat is defined as a string UU' such that U' is discrete scale of U. In addition to the algorithms for exact tandem repeats, we also present algorithms that solve the problem in the inexact sense, allowing up to k mismatches. We believe that this framework will open a new perspective for other problems in the future.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Pattern matching

Keywords and phrases double string, tandem repeat, 2-dimensional, scale

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Amihood Amir*: Partially supported by Binational Science Foundation Grant 2018141 and Israel Science Foundation Grant 1475-18.
Gad M. Landau: Partially supported by Binational Science Foundation Grant 2018141 and Israel Science Foundation Grant 1475-18.
Dina Sokol: Partially supported by Binational Science Foundation Grant 2018141.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A *tandem repeat*, or *square*, is a string which consists of two consecutive identical occurrences of a substring or *root*, e.g. *abab*. Finding all tandem repeats in a given string is a well-studied problem with many applications in diverse areas such as biological sequence analysis and data compression. A *maximal run* in a string S is a substring of S that is periodic and cannot be extended at all to the right or left, e.g. *ababa* is a maximal run in the string aba*ababa*c. A maximal run in a string represents contiguous tandem repeats, all with periods conjugates of each other, and as such, maximal runs have been used to succinctly encode all tandem repeats. For example, the maximal run *ababa* represents consecutive tandem repeats with roots *ab* and *ba*.

In this paper, we consider the problem of finding tandem repeats in input that consists of two parallel strings. We define a *double string*, and introduce the corresponding notions of tandem repeat and run in a double string. Double strings are ubiquitous in nature, as molecules such as DNA and RNA come in pairs.[1] Hence, the problem considered is interesting from both a theoretical and practical perspective. However, the strength of this paper's contribution lies in its applicability to unrelated variants of the tandem repeats problem. We show how the solution to the double string tandem repeats problem can be used to solve two different problems. The first is finding 2D corner-sharing tandems, and the second is finding all scaled tandem repeats. We are confident that more applications of double string pattern matching will be discovered in the future.

In Section 2 we present precise definitions and examples of tandem repeats and runs in a double string, and then prove upper and lower bounds on the number of occurrences of such runs. In Section 3 we present a $O(n \log n)$ time algorithm for locating all double string tandem repeats. We then extend this algorithm to deal with double string tandem repeats while allowing k mismatches. In Section 4 we provide a reduction of the 2-dimensional (2D) *corner-sharing tandem problem* to the double string tandem repeats problem. We thus obtain a more efficient algorithm for locating all corner-sharing tandems in a 2D text, both with and without mismatches. Finally, in Section 5 we solve the *scaled tandem repeats problem* by reducing it to a tandem repeats problem on double strings.

2 Definition and Characterization of Double String TR's

We use $S[i]$ to denote the ith character of a string S, and $S[i \dots j]$ to denote the substring of S from $S[i]$ through $S[j]$.

▶ **Definition 1.** *A* double string *of length n consists of two parallel sequences over a given alphabet, each of length n, indexed by $1 \dots n$. We call the two strings S_1 and S_2.*

Example 1: a double string of length 5, with $S_1 = aabca$ and $S_2 = ccbba$.

1	2	3	4	5
a	a	b	c	a
c	c	b	b	a

▶ **Definition 2.** *A* double string tandem repeat *(2-str TR) is a substring of S_1 and a substring of S_2 that are identical and consecutive. As in one string, we call the repeating*

[1] In DNA and RNA there are specific relationships between corresponding bases, while our definition of a double string does not imply any such relationship.

substring the root *or* period *of the 2-str TR. Specifically, a 2-str TR with length* $2p$ *beginning at location* i *in* S_1*, implies that the substring* $S_1[i \dots i+p-1]$ *is identical to the substring* $S_2[i+p \dots i+2p-1]$*. A 2-str TR beginning at location* j *in* S_2 *implies that* $S_2[j \dots j+p-1]$ *is identical to the substring* $S_1[j+p \dots j+2p-1]$.

Example 2: A double string tandem repeat with root *abc* begins at location 2 in S_1.

1	2	3	4	5	6	7	8
a	**a**	**b**	**c**	a	a	b	b
c	c	b	b	**a**	**b**	**c**	d

For the remainder of the paper, we assume that the 2-str TR begins in S_1; all lemmas and algorithms apply with minor modifications to indices for those beginning in S_2.

▶ **Definition 3.** *A* 2-str run (i, j, p) *in a double string* (S_1, S_2) *of length* n*,* $1 \le i \le j \le n-2p+1$*,* $1 \le p \le n/2$*, is a sequence of one or more 2-str TR's with* period size p *beginning at each location* $i \le \ell \le j$ *in* S_1*. The run is said to be* maximal *if it cannot be extended to the left or right, i.e. both* $(i-1, j, p)$ *and* $(i, j+1, p)$ *are not 2-str runs.*

Example 3: A maximal run with period size 3 occurs at locations $1 \dots 6$ in S_1 and $4 \dots 9$ in S_2. It can be represented by the triple $(1, 4, 3)$, since 1 is the start of the leftmost tandem, 4 is the start of the rightmost tandem, and 3 is the period size.

1	2	3	4	5	6	7	8	9
a	**b**	**c**	**x**	**y**	**z**	z	z	z
a	a	a	**a**	**b**	**c**	**x**	**y**	**z**

Although all of the consecutive 2-str TR's in a 2-str run have the same period *size*, the actual characters in the periods can be different for different tandems in the same run (as is evident in Example 3 which shows 2-str TRs with roots *abc, bcx, cxy, xyz*). Thus, transitivity in equality of location i with location $i-p$ and $i+p$, for period p, which holds trivially for a run in a string, does **not** hold for a 2-str run. Nevertheless, 2-str runs can still be used as an efficient encoding of consecutive 2-str TR's, and as we show in the next subsection, there cannot be too many of them.

2.1 The number of maximal 2-str runs in a double string

▶ **Lemma 4.** *Two distinct maximal 2-str runs in a double string, with the same period size, cannot overlap within* S_1 *or* S_2*.*

Proof. Let p be the period size of two distinct maximal 2-str runs in a given double string, and let j be the rightmost location of the 2-str run that has the leftmost starting location. Due to the maximality, there must be a mismatch following the first run, thus $S_1[j+1] \neq S_2[j+p+1]$, and location $j+1$ cannot be included in any 2-str run with period p due to the mismatch. Therefore, the second 2-str run must start to the right of location $j+1$ in S_1 and hence cannot overlap. ◀

Note that in one string, two maximal runs with the same period size may overlap, as long as the overlap is shorter than the period size, for e.g. *abcabcxbcx*.

▶ **Lemma 5.** *There can be* $O(n \log n)$ *maximal 2-str runs in a double string of length* n*.*

Proof. For a given period p there are no more than n/p maximal 2-str runs since they cannot overlap by Lemma 4. Since p can be $1 \dots n/2$, this yields $\Sigma_p^{n/2} n/p$, a harmonic series which is bound by $O(n \log n)$. ◀

▶ **Lemma 6.** *There can be $\Omega(n)$ maximal 2-str runs in a double string of length n.*

Proof. If we take any S_1 that contains $O(n)$ runs (e.g. Fibonacci string [9]), and then set $S_1 = S_2$, we will get a double string with $O(n)$ 2-str runs, since the first period of each run in S_1 will pair up with the second period in S_2. ◀

Remarks: We point out that the gap of $\log n$ between the upper and lower bound remains an open problem. Further, we note that the definitions of 2-str TR and run ignore the concept of primitivity, and we include reasoning for this in the appendix. Thus, for example, the double string $S_1 = S_2 = a^n$ contains $\lfloor \frac{n}{2} \rfloor$ 2-str runs, one for each period size.

3 The Algorithm

A common idea used in algorithms that find tandem repeats in a string, is to search for all tandem repeats that cross a given point (see for e.g. [11, 15]). Instead of fixing the starting point of a tandem, and searching for xx, the algorithm fixes certain points that the set of contiguous tandems must cross, and searches for all tandems that cross that point. We use this idea, searching for each period size separately, and reporting consecutive tandem repeats as a single run. We follow the framework of the Main-Lorentz algorithm [16] (see pseudocode in Algorithm 1). Given an input double string (S_1, S_2) of length n, in the first iteration, all runs that cross the center of the string are found. In the following iteration, (S_1, S_2) is split into two halves, and each one is searched individually. (To simplify the presentation we assume that n is a power of 2.) As implemented in Algorithm 1, this continues for $\log n$ iterations.

The runs that cross the center are classified into two groups. A *right* run has more than half of its characters to the right of the center of the string, and a *left* run has the majority of its characters to the left of the center. Algorithm 2, together with Figure 1, describes the procedure that finds all right runs; by symmetry, all left runs can be found.

The novel idea of Algorithm 2 is that computing the longest common extensions using *two different strings* yields the desired results. The forward comparisons are done with a substring of S_1 against a substring S_2, and the same for the reverse comparisons. These extensions define which runs occur in the double string crossing the midpoint. The standard KMP algorithm [10] is used to compute all of the forward and reverse extensions in linear time. The input pattern to KMP for the forward extensions is the string $S_1[\frac{n}{2} \dots n]$ and the text is $S_2[\frac{n}{2} + 1 \dots n]$. Conversely the reverses of $S_1[1 \dots \frac{n}{2} - 1]$ and $S_2[1 \dots n]$ are used as input to KMP for the reverse extensions.

Algorithm 1 Find Runs in a Double String

Input: double string (S_1, S_2) of length n
Output: all runs that occur in the double string

```
for i = log2 n downto 1 do                          ▷ for log n iterations of ML framework
    for ℓ = 0 to n/2^i − 1 do                       ▷ for each piece of the input of width 2^i
        FindRightRuns((S1, S2) , ℓ2^i + 1, (ℓ + 1)2^i)
        FindLeftRuns((S1, S2) , ℓ2^i + 1, (ℓ + 1)2^i)
    end for
end for
```

▶ **Lemma 7.** *Algorithm 1 finds all 2-str runs in a double string (S_1, S_2) in $O(n \log n)$ time.*

Algorithm 2 FindRightRuns

Input: double string (S_1, S_2), beg, end (beginning and end indexes of substring to search)
Output: all right runs that occur in the double string that cross the midpoint.
$n = end - beg + 1$
$mid = (beg + end)/2$
for $p = 1$ to $n/2$ **do** ▷ find runs with period p

δ_1 = length of longest common prefix of $S_1[mid \dots n]$ and $S_2[mid + p \dots n]$
▷ Forward Extension
δ_2 = length of longest common suffix of $S_1[1 \dots mid - 1]$ and $S_2[1 \dots mid + p - 1]$
▷ Reverse Extension
if $\delta_1 + \delta_2 \geq p$ **then**
report run $(mid - \delta_2, mid + \delta_1 - 1, p)$
end if
end for

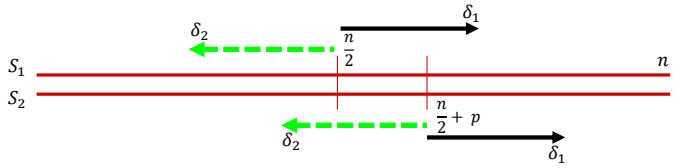

Figure 1 Computing right runs: The figure shows the first iteration, where $beg = 1$ and $end = |S_1|$. δ_1 is the length of the forward extension that results from matching $S_1[\frac{n}{2} \dots n]$ to $S_2[\frac{n}{2} + p \dots n]$. δ_2 is the length of the reverse extension of $S_1[1 \dots \frac{n}{2} - 1]$ and $S_2[1 \dots \frac{n}{2} + p - 1]$. If $\delta_1 + \delta_2 \geq p$, then there are tandem repeats with period size p beginning from location $S_1[\frac{n}{2} - \delta_2 \dots \frac{n}{2} + \delta_1 - 1]$. These are reported by the algorithm as a single run.

Proof. Every run within (S_1, S_2) crosses the center of a substring of S_1 at some point in the algorithm. As proof of this, consider a run that does not cross the center of S_1, and hence is not found in the first iteration. The run will be divided among different substrings at some point since in the final iteration the input strings are of length 1. In the step prior to its division, a given run must cross the center since the center becomes the splitting point of the following iteration. In each iteration, only one 2-str run of a given period can cross the center, since no two runs of the same period size can overlap by Lemma 4. Since the algorithm checks each possible period size, all 2-str runs will be found by the algorithm. Since there are $O(\log n)$ iterations, and each iteration takes $O(n)$ time, the total time complexity of Algorithm 1 is $O(n \log n)$. ◀

3.1 Tandem Repeats in a Double String with k-mismatches

The Hamming distance between two strings is defined as the number of mismatching characters between the two strings. Allowing a Hamming distance up to k between the two occurrences of the root results in a k-mismatch 2-str TR. The concept of a k-mismatch run applies as well, where a run includes consecutive k-mismatch tandem repeats (i.e. each repeat in the run has at most k mismatches, and overall the number of mismatches in the run is not relevant).

Example 6: A double string tandem repeat with $k = 1$ mismatch begins at location 2 in S_1.

1	2	3	4	5	6	7	8
a	**a**	**b**	**c**	a	a	b	b
c	c	b	b	**b**	**b**	**c**	d

Just as we were able to directly extend the Main and Lorentz idea in the previous section, we are able to extend the algorithm of [12] which solves the tandem repeats with k-mismatches problem in 1 string. First, instead of using KMP to find the longest common extenstions, the algorithm uses the "kangaroo method" that relies on suffix trees and Lowest Common Ancestor (LCA) queries to give the position of the first mismatch between strings [6].

Hence, suffix trees in both the forward and reverse direction must be constructed for each S_1 and S_2, and preprocessed for LCA to allow constant time Longest Common Prefix (LCP) queries [8, 13].

As in the previous algorithm, there are $O(\log n)$ iterations and in each iteration, the repeats that cross the center are found by using the forward and reverse extensions. However, in this case the comparisons are done *allowing up to k errors* in each direction. Specifically, each possible period p is searched for separately. For a given p, each LCP query returns a position of mismatch, and when the $k+1st$ mismatch is encountered, we stop. Finally, the algorithm considers each pair, $(k', k-k')$ for $0 \leq k' \leq k$. For each pair, we check whether a 2-str TR exists when allowing k' mismatches in the reverse extension and $k-k'$ mismatches in the forward extension.

Time Complexity: The number of iterations is slightly smaller than in the previous algorithm, since for substrings with length $\leq k$, our algorithm should not be run, but a simple $O(k)$ time method should be used. In each iteration, there are $O(k)$ LCP queries done for each possible period size. In addition, before reporting in a particular period, we consider $O(k)$ pairs, allowing a number of mismatches to the left and right. Hence, each iteration takes $O(nk)$ time, and the overall runtime is $O(nk \log n/k)$.

4 Application 1 - Corner Sharing Tandems

▶ **Definition 8.** *A* 2D corner-sharing tandem *(*cs-tandem*) in a 2D array, is a configuration consisting of two occurrences of the same subarray that share one corner (see Figure 2).*

In [2], Apostolico and Brimkov mention that all primitive corner-sharing tandems can be found in $O(n^4)$ time using similar techniques to their algorithm that they presented for side-sharing tandems. In this section, we reduce the problem of finding all corner-sharing tandems in a 2D array to the problem of finding tandems in a double string. We thus obtain a $O(n^3 \log n)$ time algorithm for this problem. Although the actual output may be of size $O(n^4)$ cs-tandems, we can reasonably represent the set of cs-tandems with the set of maximal cs-runs, which has size at most $O(n^3 \log n)$. For the special case of tandems that are square (i.e. of size $p \times p$), the algorithm acheives $O(n^2 \log n)$. Finally, the algorithm that allows mismatches in a 2-str TR is also extended to 2D cs-tandems with mismatches, as described in Section 4.3.

▶ **Definition 9.** *A* 2D corner-sharing horizontal run *(cs-run) is a sequence of one or more corner sharing tandems with the same period size occurring consecutively.*

▶ **Lemma 10.** *There can be* $O(n^3 \log n)$ *and* $\Omega(n^3)$ *maximal cs-runs in a 2D array of size* n^2*.*

Proof. The proof has been omitted due to lack of space.

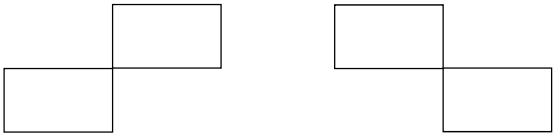

Figure 2 The two configurations of a 2D cs-tandem.

4.1 Reduction

The technique of *naming* in 1d is that of consistently replacing identical substrings with an integer called the *name*. We use the following 2D naming technique to reduce the 2D corner sharing tandem problem to the 2-str tandem problem. Given an input 2D text T, we construct $n/2$ 2D texts by naming all subcolumns of T. We create a new text called T_h for each $1 \leq h \leq n/2$, such that $T_h[r, c]$ is the name of the height h substring in column c beginning at row r.

Each two rows, i and $i + h$, in each text of names T_h, $1 \leq h \leq n/2$, is input as a double string to the algorithm that finds tandem repeats in a double string. Since we have a text of names for each height, every corner sharing tandem of height h', will appear as a 2-str tandem in the text of names for $T_{h'}$. See Figure 3 for an example.

The time complexity for the reduction is $O(n^3)$ since we construct $O(n)$ texts, each in time linear to the size of T, as the naming can be done during construction of a suffix tree of all columns [17]. Algorithm 3 presents pseudocode for the corner-sharing tandem problem. Algorithm 1 is called $O(n)$ times for each of the $O(n)$ texts, and each running of Algorithm 1 takes $O(n \log n)$ time. Overall, the 2D corner-sharing tandem problem is solved in $O(n^3 \log n)$ time.

Algorithm 3 Corner Sharing Tandems Algorithm in 2D Text

Input: 2D text T of size $n \times n$
Output: all corner-sharing tandems in T

Preprocessing: Construct $n/2$ texts of names, T_h, $1 \leq h \leq n/2$

Text Scanning:
for $h = 1$ to $n/2$ **do** ▷ for each height h
  for $r = 1$ to $n - 2h + 1$ **do** ▷ for each row r in T_h
    call Algorithm 1 with rows r and $r + h$ in T_h as S_1, S_2 respectively.
  end for
end for

4.2 Corner-Sharing *Square* Tandems

If the problem of finding all corner-sharing tandems is limited to those tandems whose roots are of size $p \times p$, we can improve our algorithm to run in $O(n^2 \log n)$ time. We will have to show two things: 1: a transformation of the input 2D text into input to the double string problem in less time. 2. The search phase of the algorithm can be improved. The

transformation can be done using the techniques of [7] for finding 2D palindromes, while the search phase can be shown to be faster using a counting trick. Details are omitted due to lack of space.

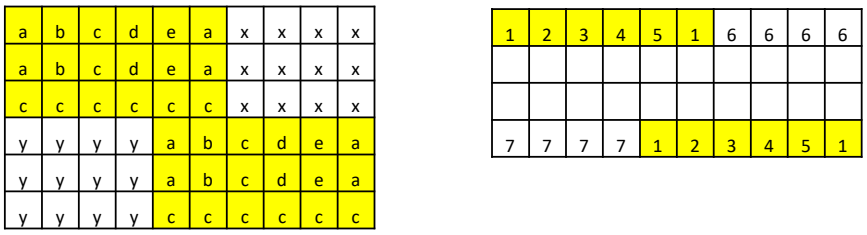

Figure 3 Input text T is shown on the left, containing a run with period size 3×4, beginning at its upper left corner. The two corresponding rows in T_3 can be viewed as a double string, and the 2-str run with period 4 found with substring "123451" in S_1 directly corresponds to the cs-run in T.

4.3 Corner Sharing Tandems with k-mismatches

A 2D corner sharing tandem that allows up to k mismatches between copies is called a k-mismatch cs-tandem. A k-mismatch cs-run can be defined analogously as a set of contiguous k-mismatch cs-tandems, such that each individual cs-tandem contains at most k mismatches. The algorithm described in Section 3.1 searches for tandem repeats in a double string allowing k mismatches. The reduction of Section 4.1 can be used in a similar manner to reduce the k-mismatch cs-tandem problem to the k-mismatch double string problem. However, each mismatch between names in the 2D text may consist of one or more mismatches in the column, and will therefore need further investigation. Hence, we will need to process the mismatching columns when attempting to discover the actual tandems. Details will be included in the full version of the paper.

5 Application 2 - Scaled Tandem Repeats

5.1 Definitions and Properties

Denote the string $aa \dots a$, where a repeated r times, by a^r. Let $S = a_1^{r_1} a_2^{r_2} \dots a_j^{r_j}$ be a string for which $a_i \neq a_{i+1}$. Let $e \in N$, we say that $S^{[e]}$ is an *e-scaling* of S if $S^{[e]} = a_1^{r_1 \cdot e} a_2^{r_2 \cdot e} \dots a_j^{r_j \cdot e}$.

▶ **Definition 11.** *A* scaled tandem repeat *is a string UU' where U' is an* e-scaling *of U for some integer e, i.e. $U' = U^{[e]}$. We call the* period *of a scaled tandem repeat the length of the first copy, i.e. $|U|$.*

We say that a scaled tandem repeat is *sharp* if the the last letter of U is not equal to the first letter of U'. Similarly, we say that scaled tandem repeat UU' occurring within text T is a sharp occurrence, if the character in T prior to U differs from the first character of U, and the following character in T differs from the last character of U'. Using the techniques of [1] it is possible to show that any solution to the problem of finding sharp occurrences of sharp scaled tandem repeats yields a solution to the general scaled tandem problem with the same complexity. Thus, we solve the following problem.

Problem Definition: Given a 1-dimensional text $T = t_1 \dots t_n$, find all sharp occurrencces of sharp scaled tandem repeats (SSTR) that are substrings of T.

We assume that the number of distinct characters in a sharp tandem repeat is at least two, otherwise it would not be sharp. We also assume that the scaled tandem repeats we are seeking are of scale $e > 1$, since for $e = 1$ this is the known case of regular tandem repeats.

Define T' as the run-length encoding (RLE) of the given text T, where each sequence of characters is replaced with a character and exponent. T'_{char} is the string of characters of the RLE of T, and T'_{exp} is the string of exponents of T'. In a similar manner to [5] we define the *quotient array* $S_Q[1..n-1]$ of array of numbers $S[1..n]$ as follows: $S_Q[i] = S[i+1]/S[i]$.

▶ **Lemma 12.** *No more than $O(n \log n)$ SSTR can occur in a string T of length n.*

Proof. Every SSTR in T must correspond to a tandem repeat in T'_{char}. By the Three Squares Lemma [4], each location in T'_{char} can have at most $O(\log n)$ tandem repeats. We conclude that there are $O(n \log n)$ SSTR's in T. ◀

The naive algorithm for the problem would consider every substring of the input text T and check whether it is an SSTR resulting in time $O(n^3)$. Known methods of using suffix trees and LCA's on the character and quotient arrays of the string (see e.g. [14]) , allow checking in constant time, for every substring U of T, whether the subsequent substring of T is a scaled copy of U. Thus the time complexity of straight-forward improvements for finding all scaled tandem repeats would be $O(n^2)$.

In the next subsection we solve the SSTR problem in a more efficient way by reducing the problem into a tandem problem on double strings. To this end, we first generalize the definition of a *run* as a concatenated string of repeats, so that the problem can fit into the framework described in Sections 2 and 3.

▶ **Definition 13.** *Let T be a string, $1 \le i < j \le n$. We say that there is a* scaled run *from $T[i]$ to $T[j]$ if there are k, ℓ; $i < k \le \ell < j$, for which $\exists e, T[k \dots j] = T[i \dots \ell]^{[e]}$. e is called the* scale of the run. *The* period *of the run is the period of the leftmost scaled tandem repeat in the run. A scaled run with scale e is* maximal *if it cannot be extended by one character either to the right or the left, i.e. there are no scaled runs from $T[i]$ to $T[j+1]$, from $T[i-1]$ to $T[j]$, nor from $T[i-1]$ to $T[j+1]$ with scale e.*

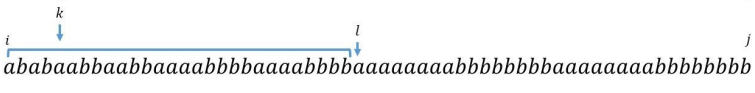

Figure 4 An example of a scaled run.

5.1.1 The Compact Region Idea for Scaling

In [3], Butman, Eres and Landau showed a linear-sized data structure of compact regions of text T that enables efficient work on scaled matching problems. The idea is to construct $n/2$ collections of strings $T_1, ..., T_{n/2}$, where the sum of the lengths of the substrings in all T_i's is $O(n)$. We will then seek dual tandems of each such substring S in the T_i's and a substring of T whose length is $O(|S|)$.

We provide below the definition of the compact regions data structure, which is based upon the following observation.

▷ Observation 1. If a substring S scaled to e occurs sharply in ${\sigma_i}^{j_i} \cdots {\sigma_k}^{j_k}$ then $j_i, \ldots, j_k$ are multiples of e.

Following the above observation, the compact regions structure computes for each scale e a compact text T_e in the following two steps:
Step 1: Locate all the regions in T where the symbols appear in scale e. Add the symbol \$ as a separator between the regions.
Step 2: Expand these regions to include the symbols on their boundaries. In order to simplify the computation of Stage 2, a symbol ${t_j}^{r_j}$ of T is replaced in T_e by ${t_j}^{\lfloor \frac{r_j}{e} \rfloor}$. Butman et al. [3] showed that the total length of all regions in $O(n)$, and that the compact regions data structure can be constructed in linear time.

5.2 The Reduction

The reduction is based on the following lemma.

▶ **Lemma 14.** *Let T be a text and assume that there is a scaled tandem to scale $e > 1$ starting in index i of T, where the length of the period is p. Then the scaled part of the tandem is represented by a substring of a single compact region in T_e. In fact, the substring in T_e is precisely the period.*

Proof: Since the scale of the period is e, then e divides the exponent of every symbol in the scaled part of the tandem. We write in T_e the scales divided by e therefore what is written in T_e is precisely the period. ◀

Assume that a compact region C in T_e starts at location i of the RLE T' of T. Lemma 14 assures us that any scaled tandem whose scaled repetition occurs in C cannot start in any index smaller than $i-|C|$ and cannot end in any index larger than $i+|C|$. Let X be the string composed of $|C|$ occurrences of σ^1, where σ is a symbol not in the alphabet. Let $C' = XC$. Then every double string tandem between the strings $T'[i-|C|..i+|C|]$ and C' is an e-scaled tandem in T. The figure below illustrates this. Both *abaabb* and *abaabbccaabbaaaabbbbcccc* are 2-scale tandems. They both appear as double string tandems between the appropriate substring of T' and C'.

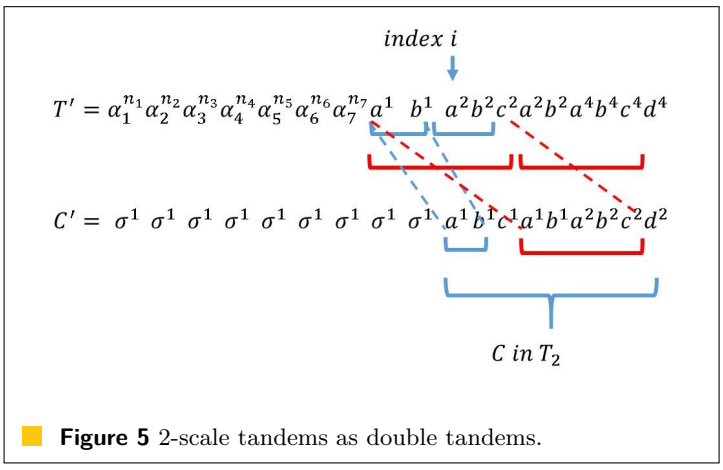

Figure 5 2-scale tandems as double tandems.

Time: The compact regions data structure is created in time $O(n)$. For every region C the double string tandem repeats are found in time $O(|C| \log |C|)$. Since $\sum_{\forall C} |C| = O(n)$ the total time is $O(n \log n)$.

References

1. A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. *Information Processing Letters*, 70(4):185–190, 1999.
2. Alberto Apostolico and Valentin E. Brimkov. Optimal discovery of repetitions in 2d. *Discrete Applied Mathematics*, 151(1-3):5–20, 2005.
3. A. Butman, R. Eres, and G.M. Landau. Scaled and permuted string matching. *Information processing letters*, 92(6):293–297, 2004.
4. Maxime Crochemore, Lucian Ilie, and Wojciech Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227 – 5235, 2009. Mathematical Foundations of Computer Science (MFCS 2007). URL: http://www.sciencedirect.com/science/article/pii/S0304397509006070, doi:10.1016/j.tcs.2009.08.024.
5. T. Eilam-Tsoreff and U. Vishkin. Matching patterns in a string subject to multilinear trasformations. *Procedings of the International Workshop on Sequences, Combinatorics, Compression, Security and Transmission, Salerno, Italy*, June 1988.
6. Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.
7. Sara H. Geizhals and Dina Sokol. Finding maximal 2-dimensional palindromes. *Information and Computation*, 266:161 – 172, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0890540119300197, doi:https://doi.org/10.1016/j.ic.2019.03.001.
8. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*, 13(2):338–355, 1984.
9. Costas S. Iliopoulos, Dennis Moore, and W.F. Smyth. A characterization of the squares in a fibonacci string. *Theoretical Computer Science*, 172(1):281 – 291, 1997. URL: http://www.sciencedirect.com/science/article/pii/S0304397596001417, doi:https://doi.org/10.1016/S0304-3975(96)00141-7.
10. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.
11. Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. URL: http://dx.doi.org/10.1109/SFFCS.1999.814634, doi:10.1109/SFFCS.1999.814634.
12. G. M. Landau, J.P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. Comput. Biol.*, 8:1–18, 2001.
13. G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10(2):157–169, 1989.
14. Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.
15. J.J. Liu, G.S. Huang, and Y.L. Wang. A fast algorithm for finding the positions of all squares in a run-length encoded string. *Theoretical Computer Science*, 410(38):3942 – 3948, 2009. URL: http://www.sciencedirect.com/science/article/pii/S0304397509003946, doi:https://doi.org/10.1016/j.tcs.2009.05.032.
16. M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
17. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

A Primitivity in 2-str TR's and Runs

A string S is *primitive* if it cannot be expressed in the form $s = u^j$, for some integer $j > 1$ and some prefix u of S. For example, *ababa* is primitive, but *abab* is *non-primitive*. The notion of primitivity is very relevant to tandem repeats, since tandem repeats with primitive roots are really the only interesting tandem repeats. In fact, in a string, a maximal run with

a primitive root encodes the information about all tandem repeats that span its substring, for e.g. *abababababa* encodes consecutive tandems with periods 2, 4, and 6. We can encode this output as a triple (i, j, p), where i is the start location of the leftmost tandem, j is the start of the rightmost tandem, and p is the smallest period (in the above example it is $(1, 10, 2)$). This encoding is commonly used in algorithms that report all tandem repeats in a string.

On the other hand, the concept of primitivity in a 2-str TR is more subtle. We cannot say that we are only interested in TR's with primitive roots, as we will miss some TR's in the double string (see Example 4). Furthermore, a non-primitive TR may be a substring of a longer run as in Example 5. This non-primitivity certainly should not disqualify the run.

Example 4: The 2-str TR beginning at location 1, of length 8, has non-primitive root *abab*. This is not implied by the 2-str TR at location 3 with primitive root *ab*.

1	2	3	4	5	6	7	8	9	10
a	**b**	**a**	**b**	c	c	c	c	c	c
c	c	c	c	**a**	**b**	**a**	**b**	a	b

Example 5: The TR at location 1 has primitive root *xbab*, the TR's at locations 2, 3, and 4 have non-primitive roots *baba, abab*. There is also a 2-str run of period 2 beginning at location 4, which in a sense encodes the TR of period 4 beginning at location 4.

1	2	3	4	5	6	7	8	9	10	11
x	**b**	**a**	**b**	**a**	**b**	**a**	b	a	b	c
c	c	c	c	**x**	**b**	**a**	**b**	**a**	**b**	**a**

We conclude that since some non-primitive TR's must be reported, an algorithm that locates all 2-str TR's must search for these TR's. Hence, our algorithm finds and reports all 2-str TR's, including those that have non-primitive roots. If necessary, those that are not interesting can be filtered out by finding all 1d runs in each string, and merging this with the output of our algorithm, since every 2-str TR that has a non-primitive root will be part of a run in each individual string of the double string.

B Example of a text T and the compact regions data structure

$T = a^2b^5a^8c^6b^4a^8b^9a^3c^7b^2a^4c^8a^9b^1a^3b^6c^9b^9$

$T_2 = a^1b^2\$b^2a^4c^3b^2a^4b^4\$b^4a^1\$a^1c^3\$c^3b^1a^2c^4a^4\$a^4\$a^1\$a^1b^3c^4\$c^4b^4\$b^4$

$T_3 = \$b^1\$b^1a^2\$a^2c^3b^1\$b^1a^2\$a^2b^3a^1c^2\$c^2\$a^1\$a^1c^2\$c^2a^3\$a^1b^2c^3b^3$

$T_4 = \$b^1\$b^1a^2c^1\$c^1b^1a^2b^2\$b^2\$c^1\$c^1\$a^1c^2a^2\$a^2\$b^1\$b^1c^2\$c^2b^2\$b^2$

$T_5 = \$b^1a^1\$a^1c^1\$c^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1\$c^1a^1\$a^1\$b^1\$b^1c^1\$c^1b^1\$b^1$

$T_6 = \$a^1\$a^1c^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1\$c^1\$c^1a^1\$a^1\$b^1c^1\$c^1b^1\$b^1$

$T_7 = \$a^1\$a^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1\$c^1a^1\$a^1\$c^1\$c^1b^1\$b^1$

$T_8 = \$a^1\$a^1b^1\$b^1\$c^1a^1\$a^1\$c^1\$c^1b^1\$b^1$

$T_9 = \$b^1\$a^1\$c^1b^1$

Figure 6 compact regions data structure example.