Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree

Jens Stoye* Dan Gusfield†

Department of Computer Science
University of California, Davis
Davis, CA 95616

Abstract

We study the problem of detecting all occurrences of (primitive) tandem repeats and tandem arrays in a string. We first give a simple time- and space-optimal algorithm to find all tandem repeats, and then modify it to become a time and space-optimal algorithm for finding only the primitive tandem repeats. Both of these algorithms are then extended to handle tandem arrays. The contribution of this paper is both pedagogical and practical, giving simple algorithms and implementations based on a suffix tree, using only standard tree traversal techniques.

1 Introduction

Suffix trees are a fundamental data structure supporting a wide variety of efficient string searching algorithms. Their "myriad virtues" are well known [1], and more than 30 non-trivial applications have been collected [6, 12]. Although alternative algorithms based on other data structures exist for many of these applications, it is remarkable that this single data structure allows so many efficient – and often surprisingly simple and elegant – solutions to so many string searching and matching problems. In particular, suffix trees are well known to allow efficient and simple solutions to many problems concerning the identification and location of repeated substrings, where the substrings are either *not* required to be contiguous, or where the substrings form the two halves of a palindrome (see [12] for a description of several of such problems). For example, the simple method described in [12] to enumerate occurrences of all maximal

---

[*] Present Address: German Cancer Research Center (DKFZ), Theoretical Bioinformatics (H0300), Im Neuenheimer Feld 280, D–69120 Heidelberg, Germany. Research supported by the German Academic Exchange Service (DAAD).

[†] Corresponding Author. E-mail: gusfield@cs.ucdavis.edu. Research partially supported by grant DBI-9723346 from the National Science Foundation, and by grant DE-FG03-90ER60999 from the Department of Energy.

pairs of repeated substrings in time proportional to their number, has been independently found by several people [15, 21, 28].

Despite the enormous versatility of suffix trees and their natural application to problems concerning non-contiguous repeats and palindromes, problems concerning *contiguous* repeated substrings have not previously had simple, natural solutions based on suffix trees. This is both surprising and disappointing, making it more difficult to teach efficient algorithms for a wide range of string problems, and complicating the long-term project (at U.C. Davis) of building practical, easily understood software for many different string tasks, based around a single resident data structure, the suffix tree. Such tools are being developed for applications in bio-sequence analysis.

In this paper we are primarily concerned with finding, in a long string, embedded substrings of the form α^i, where α is any (unspecified) substring and i is at least two. When i is two, α^i is called a "tandem repeat", and when i is greater than two, it is called a "tandem array". Note that in these definitions, all copies of α must be identical and this will be implied by our use of "tandem repeat" and "tandem array" unless stated otherwise. There is a large and current literature on studying and finding tandem repeats and tandem arrays in three fields: computer science, mathematics, and biology. A tandem repeat is also called a "square" in some computer science and mathematics literature, and is also called a "direct repeat" in biological literature. The computer science literature on tandem repeats and arrays spans almost two decades [4, 2, 23, 24, 1, 5, 7, 19, 13, 16, 17, 18]. There is also a large literature on approximate (non-identical) tandem repeats, and some of those papers (for example [22]) also discuss the problem of tandem repeats. Moreover, algorithmic issues of finding tandem repeats are discussed in depth in two current textbooks [6, 12] and reasons for interest in tandem repeats in biology are discussed in [12]. We know of two computer companies that use tandem repeat finders for certain web-related tasks. As for mathematical results, there is an extensive discussion of square-free strings, but deep and current results about strings containing tandem repeats have also been obtained, and appear for example in [7, 29, 14, 9, 10, 11]. In the biological literature, algorithms for finding identical repeats were discussed early on in [26], and more recently, a system for finding identical (but not specifically tandem) repeats in DNA and protein sequences was developed [20], and other algorithms for finding tandem repeats in biosequences (as well as related problems) were discussed in [31]. There is extensive interest in many types of repeats in molecular biology with an enormous literature. Overwhelmingly, that literature concerns interspersed repeats, or approximate (non-identical) tandem repeats, but many papers do concern (identical) tandem repeats. For example, tandem repeats are discussed in [32]: "There are sequence designs that promote evolution. One such design suitable for fast adaptation is the tandem repetition of identical sequences, so that their copy numbers in the repeat arrays would modulate (tune) the expression of nearby genes." As another example, an analysis of the adenovirus type 8 genome [3] finds "Two sets of tandem repeats, one with five identical 33 bp repeats and the other with more than ten identical 135 bp repeats". Note that the use of "tandem repeat" here is what we call "tandem array". Additional examples of complex (identical) tandem repeats can be found in the biological literature.

The existing algorithmic literature contains methods for locating tandem repeats and array [4, 23, 24, 22] that are not based on suffix trees, although the method in [22] uses a suffix tree to solve certain subproblems. There are also two technically impressive papers, [19] and [2], which present time- and space-optimal methods using suffix trees for problems concerning tandem repeats. The methods in both of those papers are quite complex (in algorithmic detail, needed auxiliary data structures, embellishments required for optimal space use, or time and correctness proofs). The first of those papers concerns problems not addressed here, while the second paper does concern the same problems addressed here. The second paper processes a suffix tree from the bottom up and requires considerable auxiliary data structures.

In this paper we present simple, time- and space-optimal algorithms for problems of locating tandem repeats and arrays in a string S. Our methods only use standard tree traversal techniques, assuming the suffix tree for S is available. Our methods process a single suffix tree top down with only the addition of an array the size of the input string. These simple methods have both pedagogical and practical value. The algorithms are based on the fact that suffix trees allow the efficient location of what we call *branching* occurrences of tandem repeats in a string. Once these occurrences are found, almost all other repetitive structures of interest can be determined with little additional effort. Hence our various algorithms are not only simple, they are all derivatives of a single, basic algorithm. The earlier, conference version of this paper appeared in [30].

In Section 2 we introduce our terminology and state basic facts about the repeated substrings we will search for. In Section 3 we present the basic algorithm and three extensions. In Section 4 we sketch a bound on the number of occurrences of primitive tandem arrays. Section 5 concludes with an open question.

2 Strings, Suffix Trees, and Tandem Arrays

2.1 Terminology and Basic Facts

We assume a finite alphabet Σ of a fixed size. Throughout this paper, a, b, c, x, and y denote single characters from Σ; S, w, α, β, γ, δ denote strings from Σ^*.

We fix attention to a string S of length $n = |S|$; for convenience, we assume S ends with a character '\$' not occurring elsewhere in S. For $1 \leq i \leq j \leq n$, $S[i..j]$ denotes the substring of S beginning with the ith and ending with the jth character of S; we say there is an *occurrence* of $S[i..j]$ at position i in S. When the substring consists of only one letter we simply write $S[i]$ rather than $S[i..i]$.

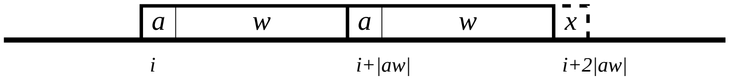

Figure 1: Occurrences of branching and non-branching tandem repeats $(i, aw, 2)$; when $x = a$, the occurrence is non-branching, when $x \neq a$, the occurrence is branching.

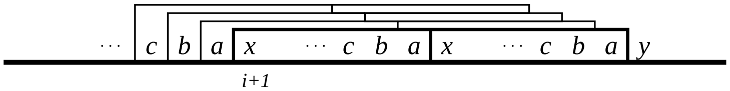

Figure 2: Chain of non-branching tandem repeats.

A string w is a *tandem array* if it can be written as $w = \alpha^k$ for some $k \geq 2$; otherwise w is called *primitive*. An occurrence of a tandem array $w = \alpha^k = S[i..i + k|\alpha| - 1]$ is represented by a triple (i, α, k). Such an occurrence is called *primitive* if α is primitive; it is called *right-maximal* if there is no additional occurrence of α immediately after w in S; it is called *left-maximal* if there is no additional occurrence of α immediately preceding w in S. A *tandem repeat* (in the literature also called a *square*) is a tandem array $w = \alpha^k$ with $k = 2$.

An occurrence $(i, \alpha, 2)$ of a tandem repeat is *branching* if and only if the character in S immediately to the right end of this occurrence, $S[i + 2|\alpha|]$, differs from $S[i + |\alpha|]$ (which must equal $S[i]$, the first character of the repeat). Figure 1 illustrates this definition.

String aw is called the *left-rotation* of string wa.

Branching repeats and left-rotations are the keys to the algorithms presented in this paper. A first indication of their importance is contained in the following fact.

Lemma 1. Any non-branching occurrence $(i, aw, 2)$ of a tandem repeat is the left-rotation of another tandem repeat, $(i + 1, wa, 2)$, starting one place to its right. The tandem repeat $(i + 1, wa, 2)$ may or may not be branching.

By repeatedly applying Lemma 1, it follows that every tandem repeat is either branching, or is contained in a chain of tandem repeats created by successive left-rotations starting from a branching tandem repeat. (Recall that string S ends with a termination symbol \$). Furthermore, if $(i + 1, wa, 2)$ is an occurrence of a tandem repeat (branching or not), then we can test in constant time if there is a tandem repeat of the same length starting at position i: simply test if $S[i] = a$. Hence, starting from a branching tandem repeat $(i + 1, wa, 2)$, the chain of tandem repeats with $(i+1, wa, 2)$ at its right end can be determined in time proportional to the length of the chain (see Figure 2).

The basic algorithm we will present in Section 3, first finds branching repeats, and then generates any desired non-branching repeats from the branching repeats. To prepare for that algorithm, we need to connect suffix trees with tan-

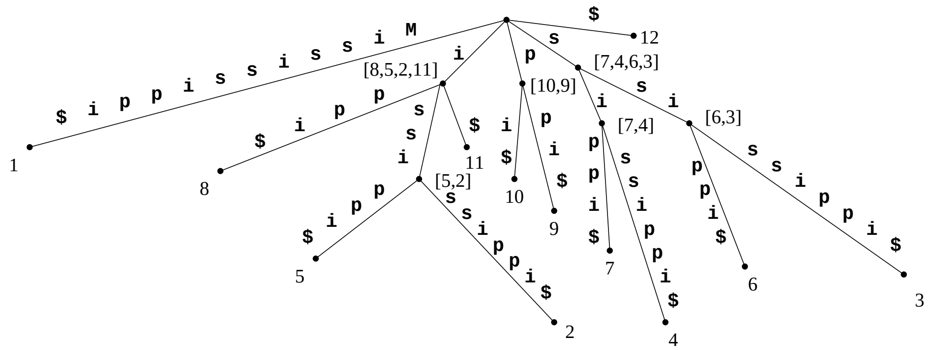

Figure 3: Suffix tree of string `Mississippi` with leaf-list $LL(v)$ at each internal node.

dem repeats.

2.2 Suffix Trees and Tandem Repeats

We assume that the reader is familiar with the basic definitions of a suffix tree. Efficient, linear time methods are known to construct a suffix tree, e.g. [34, 27, 33, 8].

We denote by $T(S)$ the suffix tree of S, i.e., the compacted trie of all the suffixes of S; $L(v)$ denotes the *path-label* of node v in $T(S)$, i.e., the concatenation of the edge labels along the path from the root to v. $D(v) = |L(v)|$ is the *string-depth* of v. Leaf v of $T(S)$ is labeled with index i if and only if $L(v) = S[i..n]$. At an internal node v of $T(S)$, we define a *leaf-list* of v as a list of the leaf-labels in the subtree below v. We denote this list by $LL(v)$. Figure 3 shows an example of a suffix tree with its leaf-lists.

The following key fact about the relationship of tandem repeats and suffix trees follows easily from the definitions, and can be found (explicitly or implicitly) in [4, 2, 19, 12].

Lemma 2. Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) There is an occurrence of a tandem repeat of length $2l$ starting at position i in S;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) \geq l$.

Lemma 2 is easily extended to characterize *branching* tandem repeats.

Lemma 3. Consider two positions i and j of S, $1 \leq i < j \leq n$, let $l = j - i$. Then the following assertions are equivalent:

(a) There is an occurrence of a branching tandem repeat of length $2l$ starting at position i in S;

(b) i and j occur in the same leaf-list of some node v in $T(S)$ with depth $D(v) = l$, but do not appear in the same leaf-list of any node with depth greater than l. Equivalently, they do not appear together in the leaf-list of any single child of v.

3 Algorithms

We will find all occurrences of branching tandem repeats in $O(n \log n)$ time, all occurrences of tandem repeats in $O(n \log n + z)$ time, where z is the number of occurrences, and all occurrences of primitive tandem repeats in $O(n \log n)$ time. All methods require just $O(n)$ space. With respect to worse case analysis, these bounds are time- and space optimal. All occurrences of tandem *arrays* of repeats (primitive or not) will be found in linear space, and in time equal or less than these bounds.

The basic algorithm and its variations are based on dividing the occurrences of tandem repeats in S into the two disjoint sets, the branching and non-branching occurrences. The branching occurrences of tandem repeats are found first, and then the non-branching occurrences are reported by successive left-rotations as suggested by Lemma 1.

3.1 The Basic Algorithm

Given Lemma 3, all occurrences of *branching* tandem repeats can be found in the following direct way:

Basic Algorithm. All nodes of $T(S)$ begin unmarked. Step 1 is repeated until all nodes are marked.

1. Select an unmarked internal node v. Mark v and execute steps 2a and 2b for node v.

2a. Collect the leaf-list, $LL(v)$, of v.

2b. For each leaf i in $LL(v)$, test whether leaf $j = i + D(v)$ is in $LL(v)$. If so, test whether $S[i] \neq S[i + 2D(v)]$. There is a branching tandem repeat of length $2D(v)$ starting at position i if and only if both tests return true. (The first test determines if $L(v)^2$ is a tandem repeat, and the second test determines if it is branching.)

The leaf-list of v is collected via any linear time traversal of the subtree rooted at v. Assuming (as is standard) a representation of the suffix tree that allows the algorithm to move from a node to a child in constant time, that traversal takes time proportional to the size of $LL(v)$.

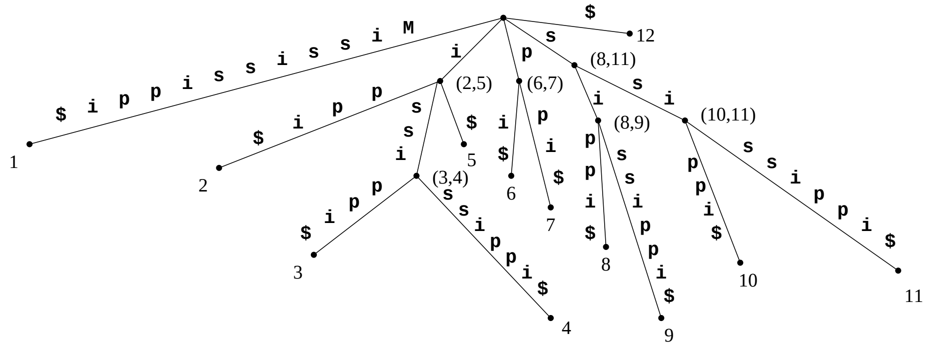

Figure 4: Suffix tree of string `Mississippi` with dfs numbers at internal nodes.

Given a leaf i in that leaf-list, we can test in constant time if $j = i + D(v)$ is also in $LL(v)$, provided we have preprocessed the suffix tree in the following standard way: During a depth-first traversal of the suffix tree (starting at the root), assign successive numbers (called *dfs numbers*) to the leaves in the order that they are encountered, and record these numbers in an array DFS, indexed by the original leaf numbers.[1] Additionally, when the depth-first traversal first visits an internal node v, record at v the next dfs number which will be given to a leaf, and when the depth-first traversal backs up from v, record at v the most recent dfs number assigned (see Figure 4). It is easy to establish that all the leaves in $LL(v)$ are assigned dfs numbers (inclusively) between the two dfs numbers recorded at v. Hence to determine if a leaf $j = i + D(v)$ is in $LL(v)$ just check if $DFS[j]$ is between the two dfs numbers recorded at v.

The above basic algorithm finds all occurrences of branching tandem repeats in time proportional to the total size of all the leaf-lists. That total size is $O(n^2)$. However, a simple modification leads to the desired time bound $O(n \log n)$.

3.2 Speeding Up the Basic Algorithm

For each node v, let v' denote the child of v whose leaf-list is largest over all the children of v (breaking ties arbitrarily). Let $LL'(v)$ denote the leaf-list of v minus the leaf-list of v', i.e., $LL'(v) = LL(v) - LL(v')$. By Lemma 3 (part b), if a branching tandem repeat starting at position i is detected by the basic algorithm during an examination of node v, then positions i and $j = i + D(v)$ must be in the leaf-lists of two distinct children of v. Hence if one of those positions is in the leaf-list of v', the other position must be in $LL'(v)$. Therefore, we need execute step 2b of the basic algorithm only for each position in $LL'(v)$, provided we look both forward from that position (as in the above basic algorithm) and backward from it (as we will do below). These ideas are formalized in the following optimized basic algorithm.

---

[1] As a side remark for those who know about suffix arrays [25], note that the array DFS is the inverse of the suffix array of S.

Optimized Basic Algorithm. All nodes of $T(S)$ begin unmarked. Step 1 is repeated until all nodes are marked.

1. Select an unmarked internal node v. Mark v and execute steps 2a, 2b and 2c for node v.

2a. Collect the list $LL'(v)$ for v.

2b. For each leaf i in $LL'(v)$, test whether leaf $j = i + D(v)$ is in $LL(v)$, the leaf-list of v. If so, test whether $S[i] \neq S[i + 2D(v)]$. There is a branching tandem repeat of length $2D(v)$ starting at that position i if and only if both tests return true.

2c. For each leaf j in $LL'(v)$, test whether leaf $i = j - D(v)$ is in $LL(v)$. If so, test whether $S[i] \neq S[i + 2D(v)]$. There is a branching tandem repeat of length $2D(v)$ starting at that position i if and only if both tests return true.

Clearly, $LL'(v)$ can be found by a traversal from v that never visits v', and that traversal takes time proportional to the size of $LL'(v)$. Moreover, from the dfs numbers at each node, the size of that node's leaf-list can be obtained (it is simply the difference of the dfs numbers plus one), so that the child of any node v with the largest leaf-list can be easily identified when needed. Hence the time for the optimized algorithm is proportional to $\sum_v |LL'(v)|$. It is a well-known fact that this sum is at most $n \log_2 n$. To see this, note that if a leaf i is in $LL'(v)$ and is also in $LL'(u)$ for some ancestor u of v, then the size of $LL'(v)$ is at most half the size of $LL'(u)$. Hence, leaf i can be counted in $\sum_v |LL'(v)|$ at most $\log_2 n$ times. In summary,

Theorem 1. All the branching tandem repeats are found in $O(n \log n)$ time and $O(n)$ space by the optimized basic algorithm.

There are additional obvious ways to improve the running time of the algorithm in practice (such as combining traversals from the internal nodes). But for simplicity of exposition, and because these improvements don't reduce the worst case running time, we omit a discussion of them.

3.3 Finding All Occurrences of Tandem Repeats

From the set of branching occurrences of tandem repeats, the non-branching occurrences are obtained by a simple enumeration procedure, based on Lemma 1. In detail, the following is executed at each occurrence of a branching tandem repeat discovered by the optimized basic algorithm.

Starting with an occurrence $(i, wa, 2)$ of a branching tandem repeat, test if $S[i-1] = a$. If they are equal, $(i-1, aw, 2)$ is reported as a non-branching tandem repeat. This process, called the *rotation procedure*, is continued to the left until an inequality is observed, at which point the procedure stops. It is obvious that the additional time used by the rotation procedure is proportional to the total number, z, of occurrences of tandem repeats in S. Hence,

Theorem 2. All occurrences of tandem repeats are found in $O(n \log n + z)$ time. No additional space is needed since all comparisons can be done directly on the string S.

The same time and space bounds were also obtained for this problem, without the use of suffix trees, in [23, 24, 22].

3.4 Primitive Tandem Repeats

A tandem repeat $\alpha\alpha$ is called a *primitive* tandem repeat if string α is primitive, i.e., α cannot itself be expressed as the repeat of some substring. It is well known that there can be at most $O(n \log n)$ occurrences of primitive tandem repeats in a string of length n. We will sketch a proof of this fact in Section 4. Because the size of the output is smaller, and because any tandem repeat can be expressed as an array of primitive tandem repeats, it is often desirable to only report primitive tandem repeats. Prior algorithms which find all occurrences of primitive tandem repeats in $O(n \log n)$ time and linear space appear in [4] and [2].

We extend the basic algorithm of the previous section to report only the primitive tandem repeats. We begin by stating a general property of primitive strings.

Lemma 4. A string wa is primitive if and only if its left-rotation aw is primitive. Hence, if $(i + 1, wa, 2)$ is an occurrence of a *primitive* tandem repeat, and $(i, aw, 2)$ is also an occurrence of a tandem repeat, then $(i, aw, 2)$ is an occurrence of a *primitive* tandem repeat.

Proof. If aw is non-primitive then $aw = \alpha^k$ for some α and $k > 1$. That means that each of the first $|\alpha|(k-1)$ characters in wa is equal to the character $|\alpha|$ places to its right. In particular, character $|\alpha| + 1$ in aw is a. Therefore, wa $= \beta^k$ where β consists of the last $k - 1$ characters of α followed by character a. Hence wa is non-primitive.

The converse, that when wa is non-primitive, then aw is also non-primitive, is proved in essentially the same way. □

The algorithmic importance of Lemma 4 is that when the (optimized) basic algorithm identifies a branching tandem repeat associated with a node v, the tandem repeats generated by the rotation procedure at node v will either all be primitive, or will all be non-primitive. So to exclude all and only the non-primitive tandem repeats, it suffices to exclude every branching tandem repeat which is not primitive. Since branching tandem repeats are identified only at nodes, it suffices to identify every node u whose path-label $L(u) = \alpha^k$ for some $k \geq 2$, where α is primitive. Clearly, such a string α will be the path-label of some ancestor node v of u. Moreover, the basic algorithm will identify the primitive branching tandem repeat $L(v)^2 = \alpha^2$ at node v. We will show next that, at that point in its execution, the basic algorithm can be extended to efficiently locate and mark all nodes below node v whose path-labels are $L(v)^k = \alpha^k$ for $k \geq 2$. That extension will also identify some other nodes that may be marked for exclusion.

To exclude all non-primitive tandem repeats (but no primitive tandem repeats) we first modify the (optimized) basic algorithm to process the nodes in a top- down order, so that no node is selected in step 1 until all of its ancestors have been selected. This ensures that a node with path-label α will be selected before a node with path-label α^k for $k \geq 2$.

Second, we combine the rotation procedure with the (optimized) basic algorithm, so that when a branching primitive repeat $L(v)^2 = \alpha^2$ is found at a node v, the algorithm next executes a rotation procedure from each branching occurrence of α^2. Each such execution rotates left through each character in a chain of consecutive α's. As a side-effect of this computation, the algorithm can determine (in essentially no extra time) the largest value of k (call it k_v) such that α^k is a substring of S. Once k_v is determined, the algorithm walks from v to the end of the path labeled α^{k_v} in the suffix tree. That path exists (and will extend from v) since α^{k_v} is a substring in S. Moreover, since the path labeled α ends at a node (v), each string α^k, for $k < k_v$, will also end at a node. During the walk, the algorithm marks each node whose path-label is α^k, meaning that that node will not be selected in step 1 of the basic algorithm. (Recognizing that the node has that label is a trivial exercise.) Note that the number of steps in the walk from v is bounded by the number of left-rotations done in the rotation procedure that discovers k_v.

Clearly, any node corresponding to a branching non-primitive tandem repeat will become marked in such a way, and hence never selected in step 1. Therefore the algorithm, as modified above, will enumerate all and only occurrences of primitive tandem repeats. The number of steps in all the extra walks is bounded by the number of left-rotations, and each left-rotation identifies a distinct occurrence of a primitive tandem repeat. Hence, the time for the algorithm is $O(n \log n + z)$, where z is the number of occurrences of primitive tandem repeats. However, z is $O(n \log n)$ in any string of length n. Hence,

Theorem 3. The method described above finds all occurrences of primitive tandem repeats in $O(n \log n)$ time and $O(n)$ space.

The time for the extra walks can be further reduced by using the skip/count trick that is known from suffix tree construction methods. That reduces the number of steps for a walk from the number of characters on the walk to the number of nodes on the walk, but, in this application, does not improve the worst case running time.

3.5 Primitive Tandem Arrays

Finally we extend the algorithm to locate all right-maximal occurrences of primitive tandem arrays. The idea is, for each branching primitive tandem repeat $(i, \alpha, 2)$ observed at a node v with $L(v) = \alpha$, successively test for $k = 1, 2, \ldots$ if leaf $i - k|\alpha|$ is also in the subtree below v. Each successful test corresponds to a branching tandem array $(i - k|\alpha|, \alpha, k + 2)$. (Here it is not necessary to test explicitly if the tandem array is branching: ¿From the fact that tandem repeat $(i, \alpha, 2)$ is branching, it follows immediately that all tandem arrays we find this way are also branching.) Once the test fails, the procedure stops.

To also find the non-branching occurrences, the rotation procedure is applied to each of the branching occurrences $(i - k|\alpha|, \alpha, k + 2)$. If we stop the rotations after $|\alpha| - 1$ steps, all and only the right-maximal occurrences of primitive tandem arrays will be obtained; otherwise all occurrences of primitive tandem arrays are obtained, and there may be as many as $n(n-1)/2$ of these. Hence in the latter case the procedure runs in time $O(n \log n + z)$ where z is the output size.

The procedure can also easily be extended to find only those primitive tandem arrays which are simultaneously left- and right-maximal if for each of the chains of right-maximal primitive tandem repeats, only the last one (when the rotation procedure stops) is reported. This procedure takes time $O(n \log n)$ as well.

4 The Number of Occurrences of Primitive Tandem Repeats

In this section we sketch a proof that there can be at most $O(n \log n)$ occurrences of primitive tandem repeats in a string of length n. This fact is well established [4, 5, 7] (in fact, it is known [29] that the number of occurrences of primitive tandem repeats is bounded by $1.45(n+1)\log_2 n - 3.3n + 5.87$). We present here the $O(n \log n)$ bound to make the paper self-contained, and because the proof given here is simpler than previously published proofs.

We say two positions i and j in the leaf-list $LL(v)$ of some node v are *adjacent in* $LL(v)$ if there is no position strictly between i and j that is also in $LL(v)$. The key fact we need is the following:

Lemma 5. Assume $i < j = i+l$, and that there is an occurrence of a *primitive* tandem repeat of length $2l$ starting at position i in S. Then (a) i and j both occur in the leaf-list $LL(v)$ of some node v in $T(S)$ with depth $D(v) \geq l$, and (b) i and j are adjacent in $LL(v)$.

Condition (a) simply repeats the necessary condition from Lemma 2 for an occurrence of a tandem repeat of length $2l$ starting at position i. Condition (b) distinguishes a primitive from a non-primitive tandem repeat. The key to proving this lemma is to show that if condition (a) is satisfied, and yet i and j are not adjacent in $LL(v)$, then the tandem repeat of length $2l$ starting at i is not primitive.

Proof (of Lemma 5). Let $\alpha\alpha$ be a tandem repeat of length $2l$ beginning at position i, and let $j = i + l$. Assume condition (a) is satisfied but (b) is not.

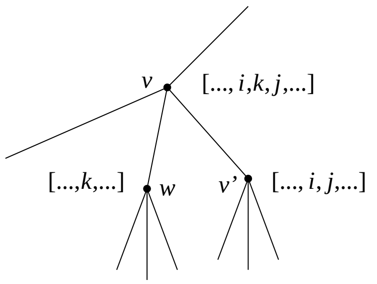

Figure 5: Scenario where (i, j) is an adjacent pair in $LL(v')$ but not in $LL(v)$.

That means there is another position k in $LL(v)$ strictly between i and j. So a copy of α occurs starting at position $k < i+l$. That copy of α can be expressed as a suffix, β, of α (from the copy starting at i) followed by a prefix, γ, of α (from the copy starting at j). It follows that $\alpha = \beta\gamma = \gamma\beta$, and by a well-known fact (Lemma 3.2.1 in [12]), α can be expressed as δ^q for some substring δ, and $q > 1$. Therefore, α is not primitive. □

A pair (i, j) is said to be an *adjacent pair* if there is some node v such that i and j are adjacent in $LL(v)$.

By Lemma 5, each occurrence of a primitive tandem repeat is associated with some adjacent pair. But each adjacent pair (i, j) is associated with at most one occurrence of a primitive tandem repeat, because that repeat is of length $2(j - i)$ and starts at i. Hence we can bound the number of occurrences of primitive tandem repeats in S by the total number of distinct adjacent pairs in all the leaf-lists of $T(S)$. For any node u, let $N(u)$ be the number of adjacent pairs that are in the leaf-list of u but not in the leaf-list of the parent of u. Define $N(r) = n - 1$, for the root r of $T(S)$. Any adjacent pair is adjacent in the leaf-lists of nodes that form a descending path in $T(S)$ (maybe only a single node in length), so the total number of distinct adjacent pairs is $\sum_u N(u)$.

Consider an internal node v' and its parent node v. Assume positions i and j are adjacent in $LL(v')$ but are not adjacent in $LL(v)$ (see Figure 5). That means that in $LL(v)$ there is some position k strictly between i and j, and that k is not in $LL(v')$. So k must be contained in the leaf-list of some other child w of v. Since for each such pair (i, j) in $LL(v')$ there is a different such "witness" k, the value of $N(v')$ can not be larger than the number of entries in the lists $LL(w)$ summed over all children w of v other than v', so $N(v') \leq \sum_w |LL(w)| = |LL(v) - LL(v')|$.

Now for any internal node v, define (as in Section 3.2) v' to be the child of v with the largest leaf-list. It follows that $\sum_u N(u)$, and the total number of occurrences of primitive tandem repeats, is bounded by $(n-1)+\sum_v |LL(v)-LL(v')|$. That sum is bounded by $O(n \log n)$ following the discussion in Section 3.2.

5 Summary and an Open Question

The time and space bounds for the methods presented here have been obtained earlier. Therefore, the contribution of this paper is the simplicity of the algorithms, which use only standard traversals of a suffix tree. The success of this effort must therefore be gauged by comparing the methods in this paper with earlier methods (particularly those in [2]) that use suffix trees to find contiguous repeated substrings.

We leave it as an open question whether the use of branching tandem repeats also allows linear-time solutions for related problems which are solvable within that time bound (e.g. the problem of finding the shortest tandem repeat beginning at each position of a string, cf. [19]). A positive indication is that the number of occurrences of branching tandem repeats in a string of length n is bounded by n, as shown in [16, Theorem 3].

Moreover, the number of branching tandem repeats is identical to the number of chains of tandem repeats in a string, as shown in Lemma 1. Hence, they give a linear-space encoding for all occurrences of tandem repeats in a string, a question posed by Iliopoulos *et al.* [14]. (Another linear-space encoding, namely the end-locations of all the strings which occur as tandem repeats in the suffix tree, is introduced in [13].)

The C source code of an implementation of the algorithms presented in this paper (and many more) is available from
http://www.cs.ucdavis.edu/~gusfield/strmat.html.

References

[1] A. Apostolico, The myriad virtues of subword trees, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer Verlag, Berlin, 1985) 21–39.

[2] A. Apostolico and F. P. Preparata, Optimal off-line detection of repetitions in a string, *Theor. Comput. Sci.* **22** (1983) 297–315.

[3] J.X. Cao and P.J. Krell and E. Nagy. Sequence and transcriptional analysis of terminal regions of the fowl adenovirus type 8 genome. *J Gen Virol* 1998 Oct;79 (Pt 10):2507-2516

[4] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Inf. Process. Lett.* **12** (1981) 244–250.

[5] M. Crochemore and W. Rytter, Periodic prefixes in texts, in: R. Capocelli, A. De Santis, and U. Vaccaro, eds., *Sequences II* (Springer Verlag, Berlin, 1993) 153–165.

[6] M. Crochemore and W. Rytter, *Text Algorithms* (Oxford University Press, New York, 1994).

[7] M. Crochemore and W. Rytter, Squares, cubes, and time-space efficient string searching, *Algorithmica* **13** (1995) 405–425.

[8] M. Farach, Optimal suffix tree construction with large alphabets, in: *Proc. 38th Annu. Symp. Found. Comput. Sci., FOCS 97* (IEEE Press, 1997) 137–143.

[9] A. S. Fraenkel and J. Simpson. How many squares must a binary string contain? *Electronic J. of Combinatorics, vol. 2, p. 1-9, 1995*

[10] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Comb. Theory Ser. A*, 82:112–120, 1998.

[11] A. S. Fraenkel and J. Simpson. The exact number of squares in Fibonacci words. *Theoretical Computer Science* 218 (1999) 95-106.

[12] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (Cambridge University Press, New York, 1997).

[13] D. Gusfield and J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, Technical Report CSE-98-4, Department of Computer Science, UC Davis, 1998.

[14] C. S. Iliopoulos, D. Moore and W. F. Smyth, A characterization of the squares in a Fibonacci string, *Theor. Comput. Sci.* **172** (1997) 281–291.

[15] R. W. Irving, personal communication.

[16] R. Kolpakov and G. Kucherov, On the sum of exponents of maximal repetitions in a word, Report 99-R-034, LORIA, France, 1999.

[17] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. Proceedings of IEEE Foundations of Computer Science, October 1999.

[18] R. Kolpakov and G. Kucherov. On maximal repetitions in words Proceedings of Foundamentals of Computation Theory, p. 374-385 (1999).

[19] S. R. Kosaraju, Computation of squares in a string, in: M. Crochemore and D. Gusfield, eds., *Proc. 5th Annu. Symp. Combinatorial Pattern Matching, CPM 94* (LNCS 807, Springer Verlag, Berlin, 1994) 146–150.

[20] S. Kurtz and C. Schleiermacher, REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, Vol. 15, issue 5, 1999, p. 426-427.

[21] G. M. Landau, personal communication.

[22] G. M. Landau and J. P. Schmidt, An algorithm for approximate tandem repeats, in: A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., *Proc. 4th Annu. Symp. Combinatorial Pattern Matching, CPM 93* (LNCS 684, Springer Verlag, Berlin, 1993) 120–133.

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

[24] M. G. Main and R. J. Lorentz, Linear time recognition of squarefree strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer Verlag, Berlin, 1985) 271–278.

[25] U. Manber and E. W. Myers, Suffix arrays: A new method for on-line search, *SIAM J. Computing* **22** (1993) 935–948.

[26] H. Martinez, An Efficient Method for Finding Repeats in Molecular Sequences, *Nucleic Acids Research* **11** (1983), 4626-4634.

[27] E. M. McCreight, A space-economical suffix tree construction algorithm. *J. ACM* **23** (1976) 262–272.

[28] J. P. Schmidt, personal communication.

[29] P. F. Stelling, *Applications of Combinatorial Analysis to Repetitions in Strings, Phylogeny, and Parallel Multiplier Design*, Ph.d. dissertation, Department of Computer Science, University of California, Davis, 1995.

[30] J. Stoye and D. M. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree, in: M. Farach-Colton, ed., *Proc. 9th Annu. Symp. Combinatorial Pattern Matching, CPM 98* (LNCS 1448, Springer Verlag, Berlin, 1998) 140–152.

[31] T. Tsunoda and M. Fukagawa and T. Takagi. Time and memory efficient algorithm for extracting palindromic and repetitive subsequences in nucleic acid sequences. Pacific Symposium on Biocomputing 4:202-213 (1999).

[32] E.N. Trifonov. Elucidating sequence codes: three codes for evolution. N Y Acad Sci 1999, May 18;870:330-338.

[33] E. Ukkonen, On-line construction of suffix trees, *Algorithmica* **14** (1995) 249–260.

[34] P. Weiner, Linear pattern matching algorithms, in: *IEEE 14th Annual Symposium on Switching and Automata Theory* (IEEE Press, 1973) 1–11.