Contents lists available at ScienceDirect

European Journal of Combinatorics

journal homepage: www.elsevier.com/locate/ejc

EERTREE: An efficient data structure for processing palindromes in strings

Mikhail Rubinchik, Arseny M. Shur

Ural Federal University, Ekaterinburg, Russia

ARTICLE INFO

Article history:
Available online xxxx

ABSTRACT

We propose a new linear-size data structure which provides a fast access to all palindromic substrings of a string or a set of strings. This structure inherits some ideas from the construction of both the suffix trie and suffix tree. Using this structure, we present simple and efficient solutions for a number of problems involving palindromes.

1. Introduction

Palindromes are one of the most important repetitive structures in strings. During the last decades they were actively studied in formal language theory, combinatorics on words and stringology. Recall that a palindrome is any string $S = a_1a_2\cdots a_n$ equal to its reversal $\overleftarrow{S} = a_n\cdots a_2a_1$.

There are a lot of papers concerning the palindromic structure of strings. The most important problems in this direction include the search and counting of palindromes in a string and the factorization of a string into palindromes. Manacher [18] came up with a linear-time algorithm which can be used to find all maximal palindromic substrings of a string, along with its palindromic prefixes and suffixes. The problem of counting and listing distinct palindromic substrings was solved offline in [8] and online in [16]. Knuth, Morris, and Pratt [14] gave a linear-time algorithm for checking whether a string is a product of even-length palindromes. Galil and Seiferas [6] asked for such an algorithm for the *k-factorization* problem: decide whether a given string can be factored into exactly k non-empty palindromes, where k is an arbitrary constant. They presented an online algorithm for $k = 1, 2$ and an offline one for $k = 3, 4$. An online algorithm working in $O(kn)$ time for the length n string and any k was designed in [17]. Close to the k-factorization problem is the problem of finding the *palindromic length* of a string, which is the minimal k in its k-factorization. This problem was solved by Fici et al. and independently by I et al. in $O(n \log n)$ time [4,10]. Palindromic factorizations also

E-mail addresses: mihail.rubinchik@urfu.ru (M. Rubinchik), arseny.shur@urfu.ru (A.M. Shur).

http://dx.doi.org/10.1016/j.ejc.2017.07.021
0195-6698/© 2017 Elsevier Ltd. All rights reserved.

attracted some interest in combinatorics; note the paper [5] on the palindromic length of infinite words.

In this paper we present a new tree-like data structure, called eertree,[1] which simplifies and speeds up solutions to search, counting and factorization problems as well as to several other palindrome-related algorithmic problems. This structure can also cope with Watson–Crick palindromes [11] and other palindromes with involution and may be interesting for the RNA studies along with the affix trees [19] and affix arrays [26]. The current paper is an extended version of the conference paper Rubinchik and Shur (2016) [23].

In Section 2 we first recall the problem of counting distinct palindromic substrings in an online fashion. This was a motive example for inventing eertree. This data structure contains the digraph of all palindromic factors of an input string S and supports the operation $\mathtt{add}(c)$ which appends a new symbol to the end of S. Thus, the number of nodes in the digraph equals the number of distinct palindromes inside S. Maintaining an eertree for a length n string with σ distinct symbols requires $O(n\log\sigma)$ time and $O(n)$ space (for a random string, the expected space is $O(\sqrt{n\sigma})$). After introducing the eertree we discuss some of its properties and simple applications.

In Section 3 we study some advanced questions related to eertrees. We consider joint eertree of several strings and name a few problems solved with its use. Then we design two "smooth" variations of the algorithm which builds eertree. These variations require at most logarithmic time for each call to $\mathtt{add}(c)$ and thus allow one to support an eertree for a string with two operations: appending and deleting the last symbol. Using one of these variations, we design a fast backtracking algorithm enumerating all *rich* strings over a fixed alphabet up to a given length. (A string is rich if it contains the maximum possible number of distinct palindromes.) Finally, we show that eertree can be efficiently turned into a persistent data structure.

The use of eertrees for factorization problems is described in Section 4. Namely, new fast algorithms are given for the k-factorization of a string and for computing its palindromic length. We also conjecture that the palindromic length can be found in linear time and provide some argument supporting this conjecture.

1.1. Definitions and notation

We study finite strings, viewing them as arrays of symbols: $S = S[1..n]$. The notation σ stands for the number of distinct symbols of the processed string. We write ε for the empty string, $|S|$ for the length of a string S, $S[i]$ for the ith symbol of S and $S[i..j]$ for $S[i]S[i+1]\ldots S[j]$, where $S[i..i-1] = \varepsilon$ for any i. A string T is a *substring* of S if $T = S[i..j]$ for some i and j. A substring $S[1..j]$ (resp., $S[i..n]$) is a *prefix* [resp. *suffix*] of S. If a substring (prefix, suffix) of S is a palindrome, it is called a *subpalindrome* (resp. *prefix-palindrome*, *suffix-palindrome*). A subpalindrome $S[l..r]$ has *center* $(l+r)/2$ and *radius* $\lceil (r-l+1)/2\rceil$. Throughout the paper we do not count ε as a palindrome.

A *trie* is a rooted tree in which (a) some nodes, including all leaves, are marked as terminal and (b) all edges are labeled by symbols such that no node has two outgoing edges with the same label. Each trie represents a finite set of strings, which label the paths from the root to the terminal nodes.

2. Building an eertree

2.1. Motive problem: distinct subpalindromes online

Well-known online linear-time Manacher's algorithm [18] outputs maximal radiuses of palindromes in a string for all possible centers, thus encoding all subpalindromes of a string. Another interesting problem is to find and count all *distinct* subpalindromes. Groult et al. [8] solved this problem offline in linear time and asked for an online solution. Such a solution in $O(n\log\sigma)$ time and $O(n)$ space was given in [16], based on Manacher's algorithm and Ukkonen's suffix tree algorithm [27]. As was proved in [16], this solution is asymptotically optimal for a general ordered alphabet. But in

[1] This structure can be found, with the reference to the first author, in a few IT blogs under the name "palindromic tree". See, e.g., http://adilet.org/blog/25-09-14/.

spite of a good asymptotics, this algorithm is based on two rather "heavy" data structures. It is natural to try finding a lightweight structure for solving the analyzed problem with the same asymptotics. Such a data structure is described below. Its further analysis revealed that it is suitable for coping with many algorithmic problems involving palindromes.

2.2. *Eertree: structure, interface, construction*

The basic version of eertree supports a single operation $\mathsf{add}(c)$, which appends the symbol c to the processed string from the right, updates the data structure respectively, and returns the number of new palindromes that appeared in the string. According to the next lemma, $\mathsf{add}(c)$ returns 0 or 1.

Lemma 2.1 ([3]). *Let S be a string and c be a symbol. The string Sc contains at most one palindrome which is not a substring of S. This new palindrome is the longest suffix-palindrome of Sc.*

From inside, eertree is a directed graph with some extra information. Its nodes numbered with positive integers starting with 1 are in one-to-one correspondence with subpalindromes of the processed string. Below we denote a node and the corresponding palindrome by the same letter. We write $\mathsf{eertree}(S)$ for the state of eertree after processing the string S symbol by symbol, left to right.

Remark 2.2. To report the number of distinct subpalindromes of S, just return the maximum number of a node in $\mathsf{eertree}(S)$.

Each node v stores the length $\mathsf{len}[v]$ of its palindrome. For the initialization purpose, two special nodes are added: with the number 0 and length 0 for the empty string, and with the number -1 and length -1 for the "imaginary string".

The edges of the graph are defined as follows. If c is a symbol, v and cvc are two nodes, then an edge labeled by c goes from v to cvc. The edge labeled by c goes from the node 0 (resp. -1) to the node labeled by cc (resp., by c) if it exists. This explains why we need two initial nodes. The outgoing edges of a node v are stored in a dictionary, which, given a symbol c, returns the destination vertex $\mathsf{to}[v][c]$ of the edge labeled by c. Such a dictionary is implemented as a binary balanced search tree and thus answers each query in $O(\log \sigma)$ time.

An unlabeled *suffix link* $\mathsf{link}[u]$ connects u to v if v is the longest proper suffix-palindrome of u. By definition, $\mathsf{link}[c] = 0$, $\mathsf{link}[0] = \mathsf{link}[-1] = -1$. The resulting graph, consisting of nodes, edges, and suffix links, is the eertree; see Fig. 1 for an example.

Lemma 2.3. *A node of positive length in an eertree has exactly one incoming edge.*

Proof. An edge leading to a node u is labeled by $c = u[1]$. Then its origin must be the node v such that $u = cvc$ or the node -1 if $u = c$. □

Proposition 2.4. *The eertree of a string S of length n is of size $O(n)$.*

Proof. The eertree of S has at most $n + 2$ nodes, including the special ones (by Lemma 2.1), at most n edges (by Lemma 2.3), and at most $n + 2$ suffix links (one per node). □

Proposition 2.5. *For a string S of length n, $\mathsf{eertree}(S)$ can be built online in $O(n \log \sigma)$ time.*

Proof. We start by defining $\mathsf{eertree}(\varepsilon)$ as the graph with two nodes (0 and -1) and two suffix links. Then we make the calls $\mathsf{add}(S[1]), \ldots, \mathsf{add}(S[n])$ in this order. By Lemma 2.1 and the definition of add, after each call we know the longest suffix-palindrome $\mathsf{palSuf}(T)$ of the string T processed so far. We support the following invariant: after a call to add, all edges and suffix links between the existing nodes are defined. In this case, adding a new node u one must build exactly one edge (by Lemma 2.3) and exactly one suffix link: any suffix-palindrome of a palindrome u is its prefix as well, and hence the destination node of the suffix link from u already exists.

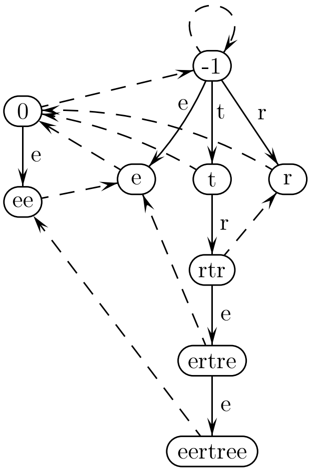

Fig. 1. The eertree of the string *eertree*. Edges are solid, suffix links are dashed.

Consider the situation after i calls. We have to perform the next call, say add(a), to $T = S[1..i]$. We need to find the maximum suffix-palindrome P of Ta. Clearly, $P = a$ or $P = aQa$, where Q is a suffix-palindrome of T. Thus, to determine P we should find the longest suffix-palindrome of T preceded by the symbol a. To do this, we traverse the suffix-palindromes of T in the order of decreasing length, starting with palSuf(T) and following suffix links. For each palindrome we read its length k and compare the symbol $T[i-k]$ against a until we get an equality or arrive at the node -1. In the former case, the current palindrome is Q; we check whether it has an outgoing edge labeled by a. If yes, this edge leads to $aQa = P$, and P is not new; if no, we create the node P of length $|Q| + 2$ and the edge (Q, P). In the latter case, $P = a$; as above, we check the existence of P in the graph using the dictionary of the current node (which is now -1) and create the node if necessary, together with the edge $(-1, P)$ and the suffix link $(P, 0)$.

It remains to create the suffix link from P if $|P| > 1$. It leads to the second longest suffix-palindrome of Ta. This palindrome can be found similar to P: just continue traversing suffix-palindromes of T starting with the suffix link of Q.

Now estimate the time complexity. During a call to add(a), one checks the existence of the edge from Q with the label a in the dictionary, spending $O(\log \sigma)$ time. The path from the old to the new value of palSuf requires one transition by an edge (from Q to P) and $k \geq 0$ of transitions by suffix links, and is accompanied by $k+1$ comparisons of symbols. In order to estimate k, follow the position of the first symbol of palSuf: a transition by a suffix link moves it to the right, and a transition by an edge moves it one symbol to the left. During the whole process of construction of eertree(S), this symbol moves to the right by $\leq n$ symbols. Hence, the total number of transitions by suffix links is $\leq 2n$. The same argument works for the second longest suffix-palindrome, which was used to create suffix links. Thus, the total number of graph transitions and symbol comparisons is $O(n)$, and the time complexity is dominated by checking the existence of edges, $O(n \log \sigma)$ time in total. □

2.3. Some properties of eertrees

We call a node *odd* (resp., even) if it corresponds to an odd-length (resp., even-length) palindrome. By *suffix path* we mean a path consisting of suffix links.

Lemma 2.6. (1) *Nodes and edges of an eertree form two weakly connected components: the tree of odd (resp., of even) nodes rooted at* -1 *(resp., at* 0*).*

Please cite this article in press as: M. Rubinchik, A.M. Shur, EERTREE: An efficient data structure for processing palindromes in strings, European Journal of Combinatorics (2017), http://dx.doi.org/10.1016/j.ejc.2017.07.021.

(2) *The tree of even (resp., odd) nodes is precisely the trie of right halves of even-length palindromes (resp., the trie of right halves, including the central symbol, of odd-length palindromes).*

(3) *Nodes and inverted suffix links of an eertree form a tree with a loop at its root* -1.

Proof. (1) If an edge (u, v) exists, then $|v| = |u| + 2$. Hence, the edges of an eertree constitute no cycles, odd nodes are unreachable from even ones, and vice versa. Further, Lemma 2.3 implies that each even (resp., odd) node can be reached by a unique path from 0 (resp., -1). So we have the two required trees.

(2) This is immediate from the definitions of a trie and an edge.

(3) The suffix link decreases the length of a node, except for the node -1. So the only cycle of suffix links is the loop at -1. Each node has a unique suffix link and is connected by a suffix path to the node -1. So the considered graph is a tree (with a loop on the root) by definition. □

Remark 2.7. Tries are convenient data structures, but a trie built from the set of all suffixes (or all factors) of a length n string is usually of size $\Omega(n^2)$. For a linear-space implementation, such a trie should be compressed into a more complicated and less handy structure: suffix tree or suffix automaton (DAWG). On the other hand, eertrees are linear-size tries and need no compression. Moreover, the size of an eertree is usually much smaller than n, because the expected number of distinct palindromes in a length n string is $O(\sqrt{n\sigma})$ [24]. This fact explains high efficiency of eertrees in solving different problems.

Remark 2.8. A θ*-palindrome* is a string $S = a_1 \cdots a_n$ equal to $\theta(a_n \cdots a_1)$, where θ is a function on the alphabet of S such that θ^2 is the identity. For example, if θ is an identity, then θ-palindromes are just palindromes, and if θ swaps A with T and G with C on the 4-letter nucleotide alphabet, then θ-palindromes are Watson–Crick palindromes (see, e.g., [11]). Clearly, an eertree containing all θ-palindromes of a string can be built in the way described in Proposition 2.5. The only difference in the $\mathsf{add}(a)$ function is that we search through the suffix-θ-palindromes of the current string until a suffix preceded by $\theta(a)$ is found.

2.4. First applications

We demonstrate the performance of eertrees on two test problems taken from student programming contests. The first problem is Palindromic Refrain [20], stated as follows: for a given string S find a palindrome P maximizing the value $|P| \cdot \mathsf{occ}(S, P)$, where $\mathsf{occ}(S, P)$ is the number of occurrences of P in S. The solution to this problem, suggested by the jury of the contest, included a suffix data structure and Manacher's algorithm.

Proposition 2.9. *Palindromic Refrain can be solved by an eertree with the use of* $O(n)$ *additional time and space.*

Proof. In order to find $\mathsf{occ}[v]$ for each node of $\mathsf{eertree}(S)$, we store an auxiliary parameter $\mathsf{occAsMax}[v]$, which is the number of i's such that $\mathsf{palSuf}(S[1..i]) = v$. This parameter is easy to compute online: after a call to add, we increment occAsMax for the current palSuf. After building $\mathsf{eertree}(S)$, we compute the values of occ as follows:

$$\mathsf{occ}[v] = \mathsf{occAsMax}[v] + \sum_{u:\mathsf{link}[u]=v} \mathsf{occ}[u]. \qquad (1)$$

Indeed, if v is a suffix of $S[1..i]$ for some i, then either $v = \mathsf{palSuf}(S[1..i])$ and this occurrence is counted in $\mathsf{occAsMax}[v]$, or $v = \mathsf{palSuf}(u)$ for some suffix-palindrome u of $S[1..i]$; in the latter case, $\mathsf{link}[u] = v$, and this occurrence of v is counted in $\mathsf{occ}[u]$. To compute the values of occ in the order prescribed by (1), one can traverse the tree of suffix links bottom-up:

```
for (v = size; v ≥ 1; v--)
  occ[v] = occAsMax[v]
for (v = size; v ≥ 1; v--)
  occ[ link[v] ] += occ[v]
```

Here size is the maximum number of a node in eertree(S). Note that the node link[v] always has the number less than v, because link[v] exists at the moment of creation of v. After computing occ for all nodes, $P = \mathrm{argmax}(\mathrm{occ}[v] \cdot \mathrm{len}[v])$. □

The second problem is Palindromic Pairs [21, Problem B]: for a string S, find the number of triples i, j, k such that $1 \le i \le j < k \le |S|$ and the strings $S[i..j]$, $S[j+1..k]$ are palindromes.

Proposition 2.10. *Palindromic Pairs can be solved by an eertree with the use of $O(n \log \sigma)$ additional time and $O(n)$ space.*

Proof. Let palSuf[j] = palSuf($S[1..j]$) and sufCount[v] be the number of suffix-palindromes of the subpalindrome v of S, including v itself. Note that sufCount[v] = 1 + sufCount[link[v]]. Hence, sufCount[v] can be stored in the node v of eertree(S) and computed when this node is created. In addition, we memorize the values palSuf[1], . . . , palSuf[n] in a separate array. The number of palindromes ending in position j of S is the number of suffix-palindromes of $S[1..j]$ or of palSuf($S[1..j]$). So this number equals sufCount[palSuf[j]].

Further, let prefCount[v] be the number of prefix-palindromes of v and palPref[j] be the longest prefix-palindrome of $S[j..n]$. The values of prefCount and palPref can be found when building eertree($\overleftarrow{S}$).[2] Similar to the above, the number of palindromes beginning in position j of S is prefCount[palPref[j]]. Note that all additional computations take $O(1)$ time for each call of add, except for the second eertree, which requires $O(n \log \sigma)$ time.

For a fixed j, the number of triples (i, j, k) defining a palindromic pair is the number of palindromes ending at position j times the number of palindromes beginning at position $j + 1$. Hence, the answer to the problem is

$$\sum_{j=1}^{n-1} \mathrm{sufCount}[\mathrm{palSuf}[j]] \cdot \mathrm{prefCount}[\mathrm{palPref}[j+1]].$$

Since this is also a linear-time computation, we are done with the proof. □

3. Advanced modifications of eertrees

3.1. Joint eertree for several strings

When a problem assumes the comparison of two or more strings, it may be useful to build a joint data structure. For example, a variety of problems can be solved by joint (“generalized”) suffix trees, see [9]. Here we introduce the *joint eertree* of a set of strings and name several problems it can solve.

A joint eertree eertree($S_1, \ldots, S_k$) is built as follows. We build eertree(S_1) in a usual fashion; then reset the value of palSuf to 0 and proceed with the string S_2, addressing the add calls to the currently built graph; and so on, until all strings are processed. Each created node stores an additional k-element Boolean array flag. After each call to add, we update flag for the current palSuf node, setting its ith bit to 1, where S_i is the string being processed. As a result, flag[v][i] equals 1 if and only if v is contained in S_i.

[2] The strings S and $\overleftarrow{S}$ have exactly the same subpalindromes, so there is no need to build the second eertree. We just perform calls to add on eertree(S) and fill prefCount and palPref.

Some problems easily solved by a joint eertree are gathered below.

Problem	Solution
Find the number of subpalindromes, common to all k given strings.	Build $\mathsf{eertree}(S_1, \ldots, S_k)$ and count the nodes having only 1's in the flag array.
Find the longest subpalindrome contained in all k given strings.	Build $\mathsf{eertree}(S_1, \ldots, S_k)$. Among the nodes having only 1's in the flag array, find the node of biggest length.
For strings S and T find the number of palindromes P having more occurrences in S than in T.	Build $\mathsf{eertree}(S, T)$, computing occ_S and occ_T in its nodes (see the proof of Proposition 2.9). Return the number of nodes v such that $\mathsf{occ}_S[v] > \mathsf{occ}_T[v]$.
For strings S and T find the number of equal palindromes, i.e., of triples (i, j, k) such that $S[i..i + k] = T[j..j + k]$ is a palindrome.	Build $\mathsf{eertree}(S, T)$, computing the values occ_S and occ_T in its nodes. The answer is $\sum_v \mathsf{occ}_S[v] \cdot \mathsf{occ}_T[v]$.

3.2. *Coping with deletions*

In the proof of Proposition 2.5, an $O(n \log \sigma)$ algorithm for building an eertree is given. Nevertheless, in some cases one call of add requires $\Omega(n)$ time, and this kills some possible applications. For example, we may want to support an eertree for a string which can be changed in two ways: by appending a symbol on the right ($\mathsf{add}(c)$) and by deleting the last symbol ($\mathsf{pop}()$). Consider the following sequence of n calls:

$$\underbrace{\mathsf{add}(a), \ldots, \mathsf{add}(a),}_{n/3 \text{ times}} \underbrace{\mathsf{add}(b), \mathsf{pop}(), \mathsf{add}(b), \mathsf{pop}(), \ldots, \mathsf{add}(b), \mathsf{pop}()}_{n/3 \text{ times}}$$

Since each appending of b requires $n/3$ suffix link transitions, the algorithm from Proposition 2.5 will process this sequence in $\Omega(n^2)$ time independent of the implementation of the operation $\mathsf{pop}()$.

Below we describe two algorithms which build eertrees in a way that provides an efficient solution to the problem with deletions.

3.2.1. *Searching suffix-palindromes with quick links*

Consider a pair of nodes v, $\mathsf{link}[v]$ in an eertree and the symbol $b = v[|v| - |\mathsf{link}[v]|]$ preceding the suffix $\mathsf{link}[v]$ in v. In addition to the suffix link, we define the *quick link*: let $\mathsf{quickLink}[v]$ be the longest suffix-palindrome of v preceded in v by a symbol different from b.

Lemma 3.1. *As a node v is created, the link $\mathsf{quickLink}[v]$ can be computed in $O(1)$ time.*

Proof. The two longest suffix-palindromes of v are $u = \mathsf{link}[v]$ and $u' = \mathsf{link}[\mathsf{link}[v]]$. Assume that v has suffixes bu and cu'. If $c \neq b$, then $\mathsf{quickLink}[v] = u'$ by definition. If $c = b$, then clearly $\mathsf{quickLink}[v] = \mathsf{quickLink}[u]$. Thus we need a constant number of operations. □

Recall that appending a letter c to a current string S, we scan suffix-palindromes of S to find the longest suffix-palindrome Q preceded by c; then $\mathsf{palSuf}(Sc) = cQc$. (If cQc is a new palindrome, then this scan continues until $\mathsf{link}[cQc]$ is found.) The use of quick links reduces the number of scanned suffix-palindromes as follows. When the current palindrome is v, we check both v and $\mathsf{link}[v]$. If both are not preceded by c, then all suffix-palindromes of S longer than $\mathsf{quickLink}[v]$ are not preceded by c either; so we skip them and check $\mathsf{quickLink}[v]$ next.

Example 3.2. Let us call $\mathsf{add}(b)$ to the eertree of the string $S = aabaabaaba$. The longest suffix-palindrome of S is the string $v = abaabaaba$. Since the symbols preceding v and $\mathsf{link}[v] = abaaba$ in S are distinct from b, we jump to $\mathsf{quickLink}[v] = a$, skipping the suffix-palindrome aba preceded by the same letter as $\mathsf{link}[v]$. Now $\mathsf{quickLink}[v]$ is preceded by b, so we find $\mathsf{palSuf}(Sb) = bab$. Note that v "does not know" which symbol precedes its particular occurrence, and different occurrences can be preceded by different symbols. So there is no way to avoid checking the symbol preceding $\mathsf{link}[v]$.

Constructing an eertree with quick links, in each step we add $O(1)$ time and space for maintaining these links and possibly reduce the number of processed suffix-palindromes. So the overall time and space bounds from Proposition 2.5 are in effect. Let us estimate the number of operations per step. The statements on "series" of palindromes, analogous to the next proposition, were proved in several papers (see, e.g., [17, Lemmas 5,6] and [4, Lemma 5]).

Proposition 3.3. *In an eertree, a path consisting of quick links has length $O(\log n)$.*

Corollary 3.4. *The algorithm constructing an eertree using quick links spends $O(\log n)$ time and $O(1)$ space for any call to* add.

Direct links wont be used

3.2.2. *Using direct links*

Now we describe the fastest algorithm for constructing an eertree which, however, uses more than $O(1)$ space for creating a node. Still, the space requirements are quite modest, so the algorithm is highly competitive:

Proposition 3.5. *There is an algorithm which constructs an eertree spending $O(\log\sigma)$ time and $O(\min(\log\sigma, \log\log n))$ space for any call to* add.

Proof. For each node we create up to σ *direct links*: $\mathsf{directLink}[v][c]$ is the longest suffix-palindrome of v preceded in v by c.

Let Q be the longest suffix-palindrome of a string S, preceded by c in S. Then either $Q = \mathsf{palSuf}(S)$ or $Q = \mathsf{directLink}[\mathsf{palSuf}(S)][c]$; note that Q does not exist if and only if this direct link is undefined. Similarly, the longest suffix-palindrome of Q, preceded by c, is $\mathsf{directLink}[Q][c]$ (or does not exist in the case of undefined link). Thus, we scan suffixes in constant time, and the time per step is now dominated by the $O(\log\sigma)$ search for an edge in the dictionary plus the time for creating direct links for a new node.

Note that the arrays $\mathsf{directLink}[v]$ and $\mathsf{directLink}[\mathsf{link}[v]]$ coincide for all symbols except for the symbol c preceding $\mathsf{link}[v]$ in v. Hence, creating a node v we first find $\mathsf{link}[v]$, then copy $\mathsf{directLink}[\mathsf{link}[v]]$ to $\mathsf{directLink}[v]$ and assign $\mathsf{directLink}[v][c] = \mathsf{link}[v]$. However, storing or copying direct links explicitly would cost a lot of space and time. Instead, we do this implicitly, using a fully persistent balanced binary search tree (*persistent tree* for short; see [2] for the full description of this and some other persistent data structures). We will not fall into details of the construction of the persistent tree, taking it as a blackbox. The persistent tree provides full access to any of its *versions*, which are balanced binary search trees.[3] The versions are ordered by the time of their creation. An update of any version results in creating a new version, which is also fully accessible; the updated version remains unchanged. Such an update as adding a node or changing the information in a node takes $O(\log k)$ time and space, where k is the size of the updated version.

We store direct links from all nodes of the eertree in a single persistent tree. Each version corresponds to a node. Direct links $\mathsf{directLink}[v][c]$ in a version v are stored as a balanced binary search tree, with the symbol c serving as the key for sorting (we assume an ordered alphabet). Creation of a node v requires an update of the version corresponding to the node $\mathsf{link}[v]$. It remains to estimate the size of a single search tree. It is at most σ by definition, and it is $O(\log n)$ by Proposition 3.3. Thus, the update time and space is $O(\min(\log\sigma, \log\log n))$, as required. □

3.2.3. *Comparing different implementations*

The three methods of building an eertree are gathered in the following table.

[3] Full access means that any information available from a "usual" search tree can be retrieved from the corresponding version of the persistent tree within the same time bounds.

Method	Time for n calls	Time for one call	Space for one node
basic	$\Theta(n \log \sigma)$	$\Omega(\log \sigma)$, $O(n)$	$\Theta(1)$
quickLink	$\Theta(n \log \sigma)$	$\Omega(\log \sigma)$, $O(\log n)$	$\Theta(1)$
directLink	$\Theta(n \log \sigma)$	$\Theta(\log \sigma)$	$O(\min(\log \sigma, \log \log n))$

The basic version is the simplest one and uses the smallest amount of memory. Quick and direct links work somewhat faster, but their main advantage is that any single call is cheap, and thus can be reversed without much pain. Hence, one can easily maintain an eertree for a string with both operations add(c) and pop(). Indeed, let add(c) push to a stack the node containing $P = \mathsf{palSuf}(Sc)$ and, if P is a new palindrome, the node containing Q such that $P = cQc$. This takes $O(1)$ additional time and space. Then pop() reads this information from the stack and restores the previous state of the eertree in constant time.

The table above also suggests the question whether some further optimization of the obtained algorithms is possible.

Question 1. *Is there an online algorithm which builds an eertree spending $O(\log \sigma)$ time and $O(1)$ space for any call to* add*?*

3.3. *Enumerating rich strings*

By Lemma 2.1, the number of distinct subpalindromes in a length n string is at most n. Such strings with exactly n distinct subpalindromes are called *rich*. Rich strings possess a number of interesting properties; see, e.g., [3,7]. The sequence A216264 in the Online Encyclopedia of Integer Sequences [25] is the growth function of the language of binary rich strings, i.e., the nth term of this sequence is the number of binary rich strings of length n. J. Shallit computed this function up to $n = 25$, thus enumerating several millions of rich strings. Using the results of Section 3.2, we were able to raise the upper bound to $n = 60$, enumerating several *trillions* of rich strings in 10 h using an average laptop. The new numerical data shows that this sequence grows much slower than it was expected before.

Proposition 3.7 serves as the theoretical basis for such a breakthrough in computation. It is based on the following obvious corollary of Lemma 2.1.

Lemma 3.6. *Any prefix of a rich string is rich.*

Proposition 3.7. *Suppose that R is the number of k-ary rich strings of length $\le n$, for some fixed k and n. Then the trie built from all these strings can be traversed in time $O(R)$.*

Proof. For simplicity, we give the proof for the binary alphabet. The extension to an arbitrary fixed alphabet is straightforward. Consider the following code, which uses an eertree on a string with deletions.

```
void calcRichString(i)
  ans[i]++
  if (i < n)
    if (add('0') )
      calcRichString(i + 1)
    pop()
    if (add('1') )
      calcRichString(i + 1)
    pop()
```

Here i is the length of the currently processed rich string. Recall that add(c) appends c to the current eertree and returns the number of new palindromes, which is 0 or 1. Hence the modified string is rich if and only if add returns 1. Note that any added symbol will be deleted back with pop(). So we exit every particular call to calcRichString with the same string as the one we entered this call. As a result, the call calcRichString(0) traverses depth-first the trie of all binary rich strings of length $\leq n$.

As was mentioned in Section 3.2, the pop operation works in constant time. For add we use the method with direct links. Since the alphabet is constant-size, the array directLink[v] can be explicitly stored in $O(1)$ space and copied in $O(1)$ time. Hence, add also works in $O(1)$ time. The number of pop's equals the number of add's, and the latter is twice the number of rich strings of length $< n$. The number of other operations is constant per call of calcRichString, so we have the total $O(R)$ time bound. □

Remark 3.8. Visit http://pastebin.com/4YJxVzep for an implementation of the above algorithm. In 10 h, it computed the first 58 terms of the sequence A216264. To increase the number of terms to 60, we used a few optimization tricks which reduce the constant in the O-term. We do not discuss these tricks here, because they make the code less readable.

3.4. Persistent eertrees

In Section 3.2 we build an eertree supporting deletions from a string. A natural generalization of this approach leads to *persistent* eertrees. Recall that a persistent data structure is a set of "usual" data structures of the same type, called *versions* and ordered by the time of their creation. A call to a persistent structure asks for the access or update of any specific version. Existing versions are neither modified nor deleted; any update creates a new (latest) version.

Consider a *tree of versions* $\mathcal{T}$ whose nodes, apart from the root, are labeled by symbols. The tree represents the set of versions of some string S: each node v represents the string read from the root to v. Recall that we denote a node of a data structure by the same letter as the string related to it. Note that some versions can be identical except for the time of their creation (i.e., for the number of a node). The problem we study is maintaining an eertree for each version of S. More precisely, the function addVersion(v, c) to be implemented adds a new child u labeled by c to the node v of $\mathcal{T}$ and computes eertree(u). The data structure which performs the calls to addVersion, supporting the eertrees for all nodes of $\mathcal{T}$, will be called a *persistent eertree*. Surprisingly enough, this complicated structure can be implemented efficiently in spite of the fact that the current string cannot be addressed directly for symbol comparisons.

Proposition 3.9. *The persistent eertree can be implemented to perform each call to* addVersion(v, c) *in* $O(\log|v|)$ *time and space.*

Proof. We use the method with direct links and build, as in Section 3.1, a joint eertree for all versions. However, now we do not use flags; instead, each node of the tree $\mathcal{T}$ stores links to the palindromes of the corresponding version of S. Overall, the node v of $\mathcal{T}$ contains the following information: a binary search tree searchTree[v], containing links to all subpalindromes of v; link palSuf[v] to the maximal suffix-palindrome of v; array pred[v], whose ith element is the link to the predecessor z of v such that the distance between z and v in $\mathcal{T}$ is 2^i ($i \geq 0$); and the symbol symb[v] added to the parent of v to get v. All listed parameters except for searchTree[v] use $O(\log|v|)$ space. For search trees we use, as in Section 3.2, the persistent tree [2], reducing both time and space for copying the tree and inserting one element to $O(\log|v|)$. (Recall that another persistent tree is used inside the eertree for storing direct links of all nodes.)

Now we implement addVersion(v, c) in time $O(\log|v|)$. Note that for any i the symbol $v[i]$ can be found in $O(\log|v|)$ time. Indeed, this symbol is symb[z], where z is the predecessor of v such that the distance between z and v is $h = |v| - i$. Using the binary representation of h, we can reach z from v in at most $\log|v|$ steps following the appropriate pred links.

Let V be the current number of versions (at any time). Creating a new version u with the parent v, we increment V by one and compute all parameters for u. First we compute pred[u]. This can be done in $O(\log|v|)$ time because pred[u][0] $= v$ and pred[u][i] $=$ pred[pred[u][$i-1$]][$i-1$] for $i > 0$.

To compute the palindrome $y =$ palSuf[u], we call add(c) for the string v. Let x be the node in the eertree such that $cxc = y$ (and hence to[x][c] $= y$ if defined). Then $x =$ palSuf[v] if palSuf[v] is preceded by c in v and $x = \mathsf{directLink}[\mathsf{palSuf}[v]][c]$ otherwise. Hence, to compute y we access exactly one symbol of v. Further, if y is not in the eertree, a new node of the eertree should be created for y. It is easy to see that $\mathsf{link}[y] = \mathsf{to}[\mathsf{directLink}[x][c]][c]$. Next, $\mathsf{directLink}[u]$ is copied from $\mathsf{directLink}[\mathsf{link}[u]]$, with one element replaced by $\mathsf{link}[u]$. To find this element, we need to know the letter of v preceding x. Therefore, to find $\mathsf{palSuf}[u]$ and modify eertree if necessary, we need $O(\log|v|)$ time for accessing a constant number of symbols in v and $O(\log\sigma)$ time for the rest of computation in $\mathsf{add}(c)$. Finally, we create a version of the search tree for u, updating the version for v with y (if y is in the search tree for v, this tree is copied to the new version without changes). This operation takes $O(\log|v|)$ as well. The proposition is proved. The code for $\mathsf{addVersion}(v, c)$ is given below. □

```
void getpred(v, par)
  pred[v][0] = par
  i = 1
  while (pred[v][i] > 0)
    pred[v][i + 1] = pred[ pred[v][i] ][i]
    i++
int addVersion(v, c)
  V++ // the number of versions, initialized by 0
  u = V
  symb[u] = c
  pred[u] = getpred(u, v)
  if (c == v[len[v] - len[palSuf[v]]])
    x = palSuf[v]
  else
    x = directLink[palSuf[v]][c]
  palSuf[u] = to[x][c] //created if does not exist
  searchTree[u] = insert(searchTree[v], palSuf[u])
  return u
```

4. Factorizations into palindromes

As was mentioned in the introduction, the k-factorization problem can be solved online in $O(kn)$ time for the length n string and any k [17]. In this section we aim at solving this problem in time independent of k. This setting is motivated by the fact that the expected palindromic length of a random string is $\Omega(n)$ [22], and for such values of k the complexity $O(kn)$ is not satisfactory. On the positive side, the palindromic length of a string S, which is the minimum k such that a k-factorization of S exists, can be found in $O(n\log n)$ time [4,10].

4.1. Palindromic length versus k-factorization

Lemma 4.1. *Given a k-factorization of a length n string S, it is possible, in $O(n)$ time, to factor S into $k+2t$ palindromes for any positive integer t such that $k+2t \le n$.*

Proof. Let $P_1, \ldots, P_k$ be palindromes, $S = P_1 \cdots P_k$, $k \le n-2$. It is sufficient to show how to factor S into $k+2$ palindromes. If $|P_i| \ge 3$ for some i, then we split P_i into three palindromes: the first letter, the last letter, and the remaining part. Otherwise, there are some P_i, P_j of length 2, each of which can be split into two palindromes. □

Thus, k-factorization problem is reduced in linear time to two similar problems: factor a string into the minimum possible odd (resp. even) number of palindromes. We solve these two problems using an eertree. To do this, we first describe an algorithm, based on an eertree and finding the palindromic length in time $O(n\log n)$. While its asymptotics is the same as of the algorithm of [4], its constant under the O-term is much smaller (see Remark 4.8) and its code is simpler and shorter.

Proposition 4.2. *Using an eertree, the palindromic length of a length n string can be found online in time* $O(n \log n)$.

Proof. For a length n string S we compute online the array ans such that $\mathsf{ans}[i]$ is the palindromic length of $S[1..i]$. Note that any k-factorization of $S[1..i]$ can be obtained by appending a suffix-palindrome $S[j+1..i]$ of $S[1..i]$ to a $(k-1)$-factorization of $S[1..j]$. Thus,

$$\mathsf{ans}[i] = 1 + \min\{\mathsf{ans}[j] \mid S[j+1..i] \text{ is a palindrome}\}.$$

To compute ans efficiently, we store two additional parameters in the nodes of the eertree: *difference* $\mathsf{diff}[v] = \mathsf{len}[v] - \mathsf{len}[\mathsf{link}[v]]$ and *series link* $\mathsf{seriesLink}[v]$, which is the longest suffix-palindrome of v having the difference unequal to $\mathsf{diff}[v]$. Series links are similar to quick links, which are not suitable for the problem studied. Clearly, the difference is computable in $O(1)$ time and space on the creation of a node; the following code shows that the same is true for the series link.

```
if (diff[v] == diff[link[v]])
  seriesLink[v] = seriesLink[link[v]]
else
  seriesLink[v] = link[v]
```

The following “naive” implementation computes $\mathsf{ans}[n]$ in $O(n)$ time.

```
ans[n] = ∞
for (v = palSuf[v]; len[v] > 0; v = link[v])
  ans[n] = min(ans[n], ans[n - len[v]] + 1)
```

With series links, the same idea can be rewritten as follows:

```
int getMin(u)
  res = ∞
  for (v = u; len[v] > len[seriesLink[u]]; v = link[v])
    res = min(res, ans[n - len[v]] + 1)
  return res
ans[n] = ∞
for (v = palSuf[v]; len[v] > 0; v = seriesLink[v])
  ans[n] = min(ans[n], getMin(v))
```

The getMin function has linear worst-case time complexity, and we are going to speed it up to a constant time. By the *series* of a palindrome u we mean the sequence of nodes in the suffix path of u from u (inclusive) to $\mathsf{seriesLink}[u]$ (exclusive). Note that $\mathsf{getMin}[u]$ loops through the series of u. Comparing $\mathsf{diff}[u]$ and $\mathsf{diff}[\mathsf{link}[u]]$, we can check whether the series of u contains just one palindrome. If this is the case, then $\mathsf{res} = \mathsf{ans}[n - \mathsf{len}[u]] + 1$ can be computed in $O(1)$ time. Hence, below we are interested in series of at least two elements. A suffix-palindrome u of S is called *leading* if either $u = \mathsf{palSuf}(S)$ or $u = \mathsf{seriesLink}[v]$ for some suffix-palindrome v of S. We need four auxiliary lemmas. These lemmas have direct analogs in the previous work on palindromic factorizations; see [4,10,17]. However, for the convenience of the reader and self-containment of the paper, we provide their proofs.

Lemma 4.3. *If a palindrome v of length $l \geq n/2$ is both a prefix and a suffix of a string $S[1..n]$, then S is a palindrome.*

Proof. Let $i \leq n/2$. Then $S[i] = v[i] = v[l-i+1] = S[n-i+1]$, i.e., S is a palindrome by definition. □

Lemma 4.4. *Suppose v is a leading suffix-palindrome of a string $S[1..n]$ and $u = \mathsf{link}[v]$ belongs to the series of v. Then u occurs in v exactly two times: as a suffix and as a prefix.*

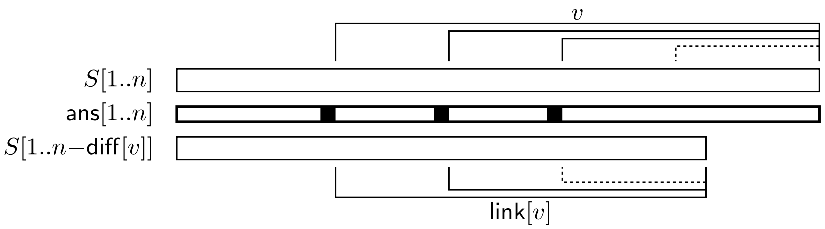

Fig. 2. Series of a palindrome v in $S[1..n]$ and of $\mathsf{link}[v]$ in $S[1..n-\mathsf{diff}[v]]$. Leading palindromes of the next series are shown by dash lines. The function $\mathsf{getMin}(v)$ returns the minimum of the values of ans in the marked positions, plus one.

n - |link[v]| + 1 == i + diff[v] == n - |v| + 1 + |v| - |link[v]|

Proof. Let $i = n - |v| + 1$. Then $v = S[i..n]$, $u = S[i + \mathsf{diff}[v]..n] = S[i..n - \mathsf{diff}[v]]$. Since $\mathsf{diff}[u] = \mathsf{diff}[v]$, we have $\mathsf{diff}[v] \le |v|/2$, so that the two mentioned occurrences of u touch or overlap. If there exist k, t such that $i < k < i + \mathsf{diff}[v]$ and $S[k..t] = u$, then $S[k..n]$ is a palindrome by Lemma 4.3. This palindrome is a proper suffix of v and is longer than $\mathsf{link}[v]$, which is impossible. □

Lemma 4.5. *Suppose v is a leading suffix-palindrome of a string $S[1..n]$ and $u = \mathsf{link}[v]$ belongs to the series of v. Then u is a leading suffix-palindrome of $S[1..n - \mathsf{diff}[v]]$.*

since diff[z] = diff[u] = diff[v] => |z| = |v|

Proof. If u is not leading, then the string $S[1..n - \mathsf{diff}[v]]$ has a suffix-palindrome $z = S[j..n - \mathsf{diff}[v]]$ with $\mathsf{link}[z] = u$ and $\mathsf{diff}[z] = \mathsf{diff}[u]$. Since u is both a prefix and a suffix of z and $|z| = |v| \le 2|u|$, clearly $z = v$. Then $w = S[j..n]$ is a palindrome by Lemma 4.3. Assume that w has a suffix-palindrome v' which is longer than v. Then v' begins with u, and this occurrence of u is neither prefix nor suffix of $z = S[j..n - \mathsf{diff}[v]]$, contradicting Lemma 4.4. Therefore, $v = \mathsf{link}[w]$ and $\mathsf{diff}[w] = \mathsf{diff}[v]$, which is impossible because v is leading. This contradiction proves that u is leading. □

Lemma 4.6. *In an eertree, a path consisting of series links has length $O(\log n)$.*

Proof. First note that $\mathsf{diff}[\mathsf{link}[v]] \le \mathsf{diff}[v]$. Indeed, let $v = v[1..m]$ be a subpalindrome of S, $t = \mathsf{diff}[v]$, $u = \mathsf{link}[v]$. If $t \ge m/2$, then $\mathsf{diff}[u] < |u| \le t$, as desired. So assume $t < m/2$. We have $u = v[t+1..m]$ by definition and $u = v[1..m-t]$ since v is a palindrome. Hence the string $z = v[2t+1..m]$ equals $v[t+1..m-t]$ as a suffix of u. Now the fact that z is both a prefix and a suffix of the palindrome u implies that z is a palindrome. Therefore $\mathsf{diff}[u] \le |u| - |z| = t$.

Assume that $u = \mathsf{seriesLink}[v']$ and $u = \mathsf{link}[v]$ for a suffix $v = v[1..m]$ of v' (possibly $v = v'$). From the definition of series link and the previous paragraph we have $s = \mathsf{diff}[u] < \mathsf{diff}[v] = t$. Let $z = \mathsf{link}[u]$. Then $z = v[t+s+1..m] = v[s+1..m-t]$ as a suffix of u. Since the string $v[s+1..m]$ is not a palindrome by the definition of link but has the palindrome z as its prefix and suffix, Lemma 4.3 implies $m - t < t + s + 1$. Hence $m < 3t$ and $|u|/|v'| \le |u|/|v| = 1 - t/m < 2/3$. Thus, every series link transition shrinks the length of a string multiplicatively, whence the result. □

By Lemma 4.6, the function $\mathsf{ans}(n)$ calls getMin $O(\log n)$ times. Now consider an $O(1)$ time implementation of getMin. Recall that it is enough to analyze non-trivial series of palindromes; they look like in Fig. 2. The first positions of all palindromes in the depicted series of v and $\mathsf{link}[v]$ match (because $\mathsf{diff}[v] = \mathsf{diff}[\mathsf{link}[v]]$) except for the last palindrome in the series of v.

We see that $\mathsf{diff}[v]$ steps before we already computed the minimum of all but one required numbers. If we memorize the minimum at that moment, we can use it now to obtain getMin in constant time. We store such a minimum as an additional parameter dp of the node of the eertree, updating it each time the palindrome represented by a node becomes a leading suffix-palindrome. Lemmas 4.4 and 4.5 ensure that when we access $\mathsf{dp}[\mathsf{link}[v]]$ to compute $\mathsf{getMin}[v]$, it is exactly the value computed $\mathsf{diff}[v]$ steps before. The computations with dp can be performed inside the getMin function:

Please cite this article in press as: M. Rubinchik, A.M. Shur, EERTREE: An efficient data structure for processing palindromes in strings, European Journal of Combinatorics (2017), http://dx.doi.org/10.1016/j.ejc.2017.07.021.

```
int getMin(v)
  dp[v] = ans[n - (len[seriesLink[v]] + diff[v])] //last
  if (diff[v] == diff[link[v]])// non-trivial series
    dp[v] = min(dp[v], dp[link[v]])
  return dp[v] + 1
```

Here dp[v] is initialized by the value of ans in the position preceding the last element of the series of v. There is nothing to do if this series does not have other elements; if it has, the minimum value of ans in the corresponding positions is available in dp[link[v]]. □

Remark 4.7. Series links can replace quick links in the construction of eertrees. Recall that in the method of quick links (Section 3.2) after checking the symbols in S preceding v and link[v] we assign quickLink[v] to v and repeat the process until the required symbol is found or the node -1 is reached. With series links, the termination condition is the same, but the process is a bit different. We first put $v = \mathsf{palSuf}(S)$ and check the symbol before v. Then we keep repeating two operations: check the symbol preceding link[v] and assign seriesLink[v] to v. In this way, all "skipped" symbols, including the symbol preceding v, equal the symbol preceding the previous value of link[v]. (This is due to periodicity of v; for details see, e.g., [17, Sect. 2].) The number of iterations of the cycle equals the number of series of suffix-palindromes of S, which is $O(\log n)$ by Lemma 4.6.

Remark 4.8. Let t_i be the number of series of suffix-palindromes for the string $S[1..i]$. Our computation of palindromic length[4] performs, in each step, the following operations. For the eertree: at most $t_i + 1$ symbol comparisons (Remark 4.7) and one $(\log\sigma)$-time access to a dictionary. For palindromic length: t_i calls to getMin, each filling one cell in dp and one cell in ans.

The algorithm by Fici et al. [4, Figure 8] uses the arrays dp and ans (under different names) and processes them in a like manner, filling $2t_i$ cells per step. In addition, this algorithm in each step builds three arrays (G, G', G''), each containing t_i triples of numbers; this gives totally $9t_i$ cells to be filled. So, our algorithm should work significantly faster. The space used by both algorithms is linear.

Now we return to the k-factorization problem.

Proposition 4.9. *Using an eertree, the k-factorization problem for a length n string can be solved online in time $O(n \log n)$.*

Proof. The above algorithm for palindromic length can be easily modified to obtain both minimum odd number of palindromes and minimum even number of palindromes needed to factor a string. Instead of ans and dp, one can maintain in the same way four parameters: ans_o, ans_e, dp_o, dp_e, to take parity into account. Now ans_o (resp., ans_e) uses the values of dp_e (resp., dp_o), while dp_o (resp., dp_e) uses the values of ans_o (resp., ans_e). The reference to Lemma 4.1 finishes the proof. □

4.2. Towards a linear-time k-factorization

A big question is whether palindromic length can be found faster than in $O(n \log n)$ time. First of all, it may seem that the bound $O(n \log n)$ for our algorithm is imprecise. Indeed, for building an eertree we scan only $O(n)$ suffix palindromes even when we use just suffix links (see the proof of Proposition 2.5). For palindromic length, in each step we run through all suffix-palindromes, but possibly skipping many of them due to the use of series links. Can this number of scanned palindromes be $O(n)$ as well? As was observed in [4], the answer is "yes" on average, but "no" in the worst case: processing any length n prefix of the famous *Zimin word*, one should analyze $\Theta(n \log n)$ series of palindromes (all of them 1-element, but this does not help).

[4] See http://ideone.com/xE2k6Y for an implementation.

Below we design an $O(n)$ offline algorithm for building an eertree of a length n string S over the alphabet $\{1, \ldots, n\}$, getting rid of the $\log \sigma$ factor in online algorithms. Then we discuss ideas which may help to obtain the palindromic length from an eertree in linear time. The offline algorithm consists of four steps.

1. Using Manacher's algorithm, compute arrays oddR and evenR, where oddR$[i]$ (resp. evenR$[i]$) is the radius of the longest subpalindrome of S with the center i (resp., $i + 1/2$).

2. Compute the longest and the second longest suffix-palindromes for any prefix of S. We use variables ℓ, ℓ', and r such that after the rth iteration the string $S[\ell..r]$ (resp., $S[\ell'..r]$) is the longest (resp., second longest) suffix-palindrome of $S[1..r]$. Consider the following code:

```
ℓ = 2
for (r = 1; r ≤ n; r++)
  ℓ--
  while ( !isPal(S[ℓ..r] )
    ℓ++
  ℓ′ = max(ℓ′ - 1 , ℓ + 1)
  while ( !isPal(S[ℓ′..r] ) && (ℓ′ ≤ r) )
    ℓ′++
  C[(ℓ + r) / 2].push(1, r)
  C[(ℓ′ + r) / 2].push(2, r)
```

The function isPal checks whether a given substring is a palindrome and works in $O(1)$ time using the value obtained on step 1 for the center $(\ell + r)/2$. Each element of the array C is a connected list; the indices are both integers and half-integers. The internal cycles make at most $2n$ increments of each of the variables ℓ, ℓ'; hence, the whole step works in linear time.

3. Build the suffix array SA and the LCP array for S; for the alphabet $\{1, \ldots, n\}$, this can be done in linear time (see, e.g., [15]). Recall that $LCP[i]$ is the length of the longest common prefix of $S[SA[i]..n]$ and $S[SA[i-1]..n]$.

4. Recall from Section 2.3 that an eertree consists of two tries, containing right halves of odd-length and even-length palindromes, respectively. Build each of them using a variation of the algorithm, constructing a suffix tree from a suffix array and its LCP array [12]. The algorithm for odd-length palindromes is given below; the algorithm for even lengths is essentially the same, so we omit it.

```
path = (-1) // stack for the current branch of the trie
for (i = 1; i ≤ n; i++)
  k = SA[i] // start processing palindromes centered at k
  while (path.size() > LCP[i] + 1)
    path.pop()
  for (j = path.size(); j ≤ oddR[k]; j++) //can be empty
    path.push( newNode(path.top(), S[k + j - 1]) )
  for (j = 1; j ≤ C[k].size(); j++)
    (rank, r) = C[k][j]
    node[rank][r] = path[r - k + 1]
```

In the above code, the function newNode(v, a) returns a new node attached to the node v with the edge labeled by a. Array node$[1][1..n]$ (resp., node$[2][1..n]$) contains links to the longest (resp., second longest) palindromes ending in given positions. We now estimate the working time of this algorithm. The outer cycle works $O(n)$ time plus the time for the inner cycles. The number of pop operations is bounded by the number of pushes, and the latter is the same as the number of nodes in the resulting eertree, which is $O(n)$. The total number of iterations of the third inner cycle is the number of palindromes stored in the whole array C; this is exactly $2n$, see step 2. Thus, the algorithm works in $O(n)$ time.

After running both the above code and its modification for even-length palindromes, we obtain the eertree without suffix links plus the auxiliary arrays node$[1]$, node$[2]$. From these arrays, the suffix links can be computed trivially:

```
for (i = 1; i ≤ n; i++)
  link[ node[1][i] ] = node[2][i];
```

Thus we have proved

Proposition 4.10. *The eertree of a length n string over the alphabet* $\{1, \ldots, n\}$ *can be built offline in* $O(n)$ *time.*

Now return to the palindromic length. Even with an $O(n)$ preprocessing for building the eertree, we still need $O(n \log n)$ time for factorization. Note that in [17] an $O(kn \log n)$ algorithm for k-factorization was transformed into a $O(kn)$ algorithm using bit compression (the so-called *method of four Russians* [1]). That algorithm produced a $k \times n$ bit matrix (showing whether a jth prefix of the string is i-factorable), so such a speed up method was natural. In our case we work with integers, so the direct application of a bit compression is impossible. However, we have the following property.

Lemma 4.11. *If* S *is a string of palindromic length* k *and* c *is a symbol, then the palindromic length of* Sc *is* $k-1$, k, *or* $k+1$.

Proof. Any k-factorization of S plus the substring c give a $(k+1)$-factorization of Sc. Suppose Sc also has a t-factorization $P_1 \cdots P_t$ for a smaller t. Then $P_t = Pc$ has length > 1. Hence, either $P = c$ and S has the t-factorization $P_1 \cdots P_{t-1}c$ or $P = cQ$ for a palindrome Q and S has the $(t+1)$-factorization $P_1 \cdots P_{t-1}cQ$. The result now follows. □

Consider a $n \times n$ bit matrix M such that $M[i, j] = 1$ if and only if $S[1..j]$ is i-factorable. For jth column, we have to compute just two values: in the rows $k-1$ and k, where k is the palindromic length of $S[1..j-1]$ (if $M[k-1, j] = M[k, j] = 0$, we write $M[k+1, j] = 1$ by Lemma 4.11). For each value we should apply the OR operation to $\log n$ bit values, to the total of $2n \log n$ bit operations. If we will be able to arrange these operations naturally in groups of size $\log n$, we will use the bit compression to get just $O(n)$ operations. Overall, Lemma 4.11, eertree, and the method of four Russians are promising tools to attack the following conjecture.

Conjecture 1. *The palindromic length of a string can be found in* $O(n \log \sigma)$ *time online and in* $O(n)$ *time offline.* [5]

5. Conclusion

In this paper, we proposed a new tree-like data structure, named eertree, which stores all palindromes occurring inside a given string. The eertree has linear size (even sublinear on average) and can be built online in nearly linear time. We also proposed some advanced modifications of the eertree, including the joint eertree for several strings, the version supporting deletions from a string, and the persistent eertree.

Then we provided a number of applications of the eertree. The most important of them are the new online algorithms for k-factorization, palindromic length, the number of distinct palindromes, and also for computing the number of rich strings up to a given length.

For further research we formulated a conjecture on the linear-time factorization into palindromes and an open problem about the optimal construction of the eertree.

Acknowledgments

The authors thank A. Kul'kov, O. Merkurev and G. Nazarov for helpful discussions. They also thank the referees for careful reading and useful remarks.

[5] The conjecture was resolved positively [13] when this article was in press.

References

[1] V. Arlazarov, E. Dinic, M. Kronrod, I. Faradzev, On economical construction of the transitive closure of a directed graph, Dokl. Akad. Nauk SSSR 194 (11) (1970) 1209–1210.

[2] J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan, Making data structures persistent, J. Comput. System Sci. 38 (1) (1989) 86–124.

[3] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de luca and Rauzy, Theoret. Comput. Sci. 255 (2001) 539–553.

[4] G. Fici, T. Gagie, J. Kärkkäinen, D. Kempa, A subquadratic algorithm for minimum palindromic factorization, J. Discrete Algorithms 28 (2014) 41–48.

[5] A.E. Frid, S. Puzynina, L.Q. Zamboni, On palindromic factorization of words, Adv. Appl. Math. 50 (2013) 737–748.

[6] Z. Galil, J. Seiferas, A linear-time on-line recognition algorithm for "Palstar", J. ACM 25 (1978) 102–111.

[7] A. Glen, J. Justin, S. Widmer, L. Zamboni, Palindromic richness, European J. Combin. 30 (2) (2009) 510–531.

[8] R. Groult, E. Prieur, G. Richomme, Counting distinct palindromes in a word in linear time, Inform. Process. Lett. 110 (2010) 908–912.

[9] D. Gusfield, Algorithms on strings, trees and sequences, Computer Science and Computational Biology, Cambridge University Press, 1997.

[10] T. I, S. Sugimoto, S. Inenaga, H. Bannai, M. Takeda, Computing palindromic factorizations and palindromic covers on-line, in: Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014. Proceedings, in: Lecture Notes in Computer Science, vol. 8486, Springer, 2014, pp. 150–161.

[11] L. Kari, K. Mahalingam, Watson-Crick palindromes in DNA computing, Nat. Comput. 9 (2010) 297–316.

[12] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching, in: LNCS, vol. 2089, Springer, Berlin, 2001, pp. 181–192.

[13] Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, Arseny M. Shur, Palindromic length in linear time, in: Juha Kärkkäinen, Jakub Radoszewski, Wojciech Rytter (Eds.), 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 78, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 23:1–23:12 URL: http://drops.dagstuhl.de/opus/volltexte/2017/7338.

[14] D.E. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.

[15] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Proceedings, in: LNCS, vol. 2676, Springer, 2003, pp. 200–210.

[16] D. Kosolobov, M. Rubinchik, A.M. Shur, Finding distinct subpalindromes online, in: Proc. Prague Stringology Conference. PSC 2013, Czech Technical University in Prague, 2013, pp. 63–69.

[17] D. Kosolobov, M. Rubinchik, A.M. Shur, Pal^k is linear recognizable online, in: Proc. 41th Int. Conf. on Theory and Practice of Computer Science, SOFSEM 2015, in: LNCS, vol. 8939, Springer, 2015, pp. 289–301.

[18] G. Manacher, A new linear-time on-line algorithm finding the smallest initial palindrome of a string, J. ACM 22 (3) (1975) 346–351.

[19] G. Mauri, G. Pavesi, Algorithms for pattern matching and discovery in RNA secondary structure, Theoret. Comput. Sci. 335 (2005) 29–51.

[20] Problems of Asia–Pacific Informatics Olympiad 2014 (2014), KBTU, Almaty, Kazakhstan zzProbApio, available at http://olympiads.kz/apio2014/apio2014_problemset.pdf.

[21] Problems of the MIPT Fall Programming Training Camp 2014 (2014) Contest 12, MIPT, Moscow, Russia, zzProbMIPT, available at https://drive.google.com/file/d/0B_DHLY8icSyNUzRwdkNFa2EtMDQ.

[22] O. Ravsky, On the palindromic decomposition of binary words, J. Autom. Lang. Comb. 8 (1) (2003) 75–83.

[23] M. Rubinchik, A.M. Shur, EERTREE: An efficient data structure for processing palindromes in strings, in: Combinatorial Algorithms - 26th International Workshop, IWOCA 2015, Revised Selected Papers, in: LNCS, vol. 9538, Springer, 2016, pp. 321–333.

[24] M. Rubinchik, A.M. Shur, The number of distinct subpalindromes in random words, Fund. Inform. 145 (2016) 371–384.

[25] N.J.A. Sloane, The on-line encyclopedia of integer sequences, available at http://oeis.org.

[26] D. Strothmann, The affix array data structure and its applications to RNA secondary structure analysis, Theoret. Comput. Sci. 389 (2007) 278–294.

[27] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

Please cite this article in press as: M. Rubinchik, A.M. Shur, EERTREE: An efficient data structure for processing palindromes in strings, European Journal of Combinatorics (2017), http://dx.doi.org/10.1016/j.ejc.2017.07.021.