Computing Palindromes on a Trie in Linear Time

Takuya Mieno
Department of Computer and Network Engineering, University of Electro-Communications, Tokyo, Japan

Mitsuru Funakoshi
Department of Informatics, Kyushu University, Fukuoka, Japan
Japan Society for the Promotion of Science, Tokyo, Japan

Shunsuke Inenaga[1]
Department of Informatics, Kyushu University, Fukuoka, Japan
PRESTO, Japan Science and Technology Agency, Tokyo, Japan

Abstract

A trie $\mathcal{T}$ is a rooted tree such that each edge is labeled by a single character from the alphabet, and the labels of out-going edges from the same node are mutually distinct. Given a trie $\mathcal{T}$ with n edges, we show how to compute all distinct palindromes and all maximal palindromes on $\mathcal{T}$ in $O(n)$ time, in the case of integer alphabets of size polynomial in n. This improves the state-of-the-art $O(n \log h)$-time algorithms by Funakoshi et al. [PSC 2019], where h is the height of $\mathcal{T}$. Using our new algorithms, the eertree with suffix links for a given trie $\mathcal{T}$ can readily be obtained in $O(n)$ time. Further, our trie-based $O(n)$-space data structure allows us to report all distinct palindromes and maximal palindromes in a query string represented in the trie $\mathcal{T}$, in output optimal time. This is an improvement over an existing (naïve) solution that precomputes and stores all distinct palindromes and maximal palindromes for each and every string in the trie $\mathcal{T}$ separately, using a total $O(n^2)$ preprocessing time and space, and reports them in output optimal time upon query.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases palindromes, suffix trees, tries, labeled trees, eertrees

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.15

Funding *Mitsuru Funakoshi*: JSPS KAKENHI Grant Number JP20J21147.
Shunsuke Inenaga: JST PRESTO Grant Number JPMJPR1922, JSPS KAKENHI Grant Number JP22H03551.

Acknowledgements We would like to thank anonymous referees for their helpful comments. We also thank Takuya Matsumoto for discussions.

1 Introduction

A string p is called a *palindrome* if p reads the same forward and backward, namely, $p = p^R$ where p^R denotes the reversed string of p. Finding palindromes in a given string is a classical problem in Theoretical Computer Science, with motivations and possible applications in Bioinformatics [11, 17]. A substring palindrome $p = S[i..j]$ in a string S is called a *maximal palindrome* if $S[i-1..j+1]$ is not a palindrome, $i = 1$, or $j = |S|$. Since any substring palindrome in S shares the same center with a unique maximal palindrome, one can essentially obtain all substring palindromes in string S by computing its maximal palindromes.

There are two well-known algorithms that compute all maximal palindromes in a given string S of length m. The algorithm of Manacher [14] finds all $2m-1$ maximal palindromes in S in $O(m)$ time and space. His algorithm scans an input string from left to right, and

[1] Corresponding author

works on a general (unordered) alphabet. The algorithm of Gusfield [11] consists in two steps: In the preprocessing step, it builds the suffix tree [18] of the concatenated string $S\$S^R\#$, where \$ and # are unique symbols not occurring inside S, and then enhances the suffix tree with the lowest common ancestor (LCA) data structure [1] that allows for LCA queries between two nodes in $O(1)$ time after linear-time preprocessing. Then, the algorithm finds all maximal palindromes by applying $2m-1$ *outward* longest common extension (outward LCE) queries in S via the LCA data structure on the suffix tree. Overall, the Gusfield algorithm works in $O(m)$ time and space in the case of *linearly-sortable alphabets*, including constant-size alphabets and integer alphabets of size polynomial in m.

While maximal palindromes tell us all *occurrences* of palindromes in a string S, the other important direction of research is the *vocabularies* of palindromes in S. The latter is called the *distinct palindromes* problem, where the task is to compute the set of all distinct palindromes in S. It is known that any string of length m can contain at most $m+1$ distinct palindromes (including the empty string) [5]. The algorithm of Groult et al. [10] finds all distinct palindromes in $O(m)$ time and space for linearly-sortable alphabets.

We consider extended versions of the above problems, in which the input is a *trie* (i.e. an edge-labeled rooted tree). Tries are a natural generalization of strings at least in two meanings: A trie can be seen as a generalized string with branches; A trie is a compact representation of a set of strings. Funakoshi et al. [7] showed that, for any trie $\mathcal{T}$ with n edges and l leaves, the number of maximal palindromes in $\mathcal{T}$ is exactly $2n-l$ and the number of distinct palindromes in $\mathcal{T}$ is at most $n+1$. In addition, they showed how to find all maximal palindromes in a given trie $\mathcal{T}$ in $O(n \log h)$ time with $O(h)$ space, where h is the height of $\mathcal{T}$. This algorithm is based on the periodicity of palindromes, and works on general alphabets. They also showed how to find all distinct palindromes in a given trie in $O(n \log h)$ time with $O(n)$ space, in the case of linearly-sortable alphabets.

In this paper, we propose the first $O(n)$-time algorithms for computing the maximal palindromes and distinct palindromes in a given trie $\mathcal{T}$ with n edges, in the case of integer alphabets of polynomial size in n. Our algorithm makes heavy use of the suffix tree of a backward trie [13, 3], but its design is quite different from the aforementioned approach by Gusfield [11]. Indeed, no $O(n)$-space $O(1)$-time outward LCE query data structures on tries are known to date (this is because the size of the suffix tree of a forward trie is $\Omega(n^2)$ [12]).

Technically speaking, our algorithm is most related to the suffix-tree based offline algorithm of Rubinchik and Shur (Proposition 4.10 in [15]) that builds the *eertree*, which is a trie-based data structure storing all distinct palindromes in a given string S. In the preprocessing phase of their method, the Manacher algorithm is used to compute all maximal palindromes in S. Our first finding in this paper is that this preprocessing phase can indeed be omitted, and instead one can use the suffix links of the leaves to check the maximality of given substring palindromes. This is helpful in our scenario since the application of the Manacher algorithm to a trie takes $O(n \log h)$ time [7]. The new suffix-link-based method can simultaneously compute all distinct palindromes (or alternatively the eertree) together with all maximal palindromes, in $O(m)$ time and space, for a given string S of length m. However, the time analysis of the suffix-link-based method is due to the fact that each leaf of the suffix tree of a string has exactly one in-coming suffix link (except for the leaf representing the whole string, which has no in-coming suffix link), and the cost of checking an in-coming suffix link to a leaf can be charged to either a unique distinct palindrome or a unique occurrence of a maximal palindrome in S. Unfortunately, in our trie case, there can be at most σ in-coming suffix links to a single leaf of the suffix tree for a trie, where σ is the alphabet size. Since we need to check whether the palindrome in question can be extended with one of at most σ

candidate characters, naïve suffix-link-based methods would require $\Omega(n \log \sigma)$ time for trie inputs, which is prohibitive for large alphabets. Still, we show an $O(n)$-time suffix-link-based method for computing all distinct/maximal palindromes in the input trie.

We also present how our trie-based $O(n)$-size data structure for storing all maximal/distinct palindromes in the input trie $\mathcal{T}$ can be used for reporting all maximal/distinct palindromes in the strings stored in $\mathcal{T}$, in *output optimal* time, without expanding the trie to the plain strings or extracting a string of interest from the trie. Note that the total length of the strings stored in $\mathcal{T}$ can be as large as $O(n^2)$.

2 Preliminaries

2.1 Strings and palindromes

Let Σ be the *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For any non-negative integer k, let Σ^k denote the set of strings of length k. For a string $S = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of S, respectively. A prefix (resp. suffix) of T of length less than $|S|$ is called a *proper prefix* (resp. *proper suffix*) of S.

For a string S and an integer $1 \leq i \leq |S|$, $S[i]$ denotes the ith character of S, and for two integers $1 \leq i \leq j \leq |S|$, $S[i..j]$ denotes the substring of S that begins at position i and ends at position j. For convenience, let $S[i..j] = \varepsilon$ when $i > j$.

Let S^R denote the reversed string of S, i.e., $S^R = S[|S|] \cdots S[1]$. A string S is called a *palindrome* if $S = S^R$. We remark that the empty string ε is also considered to be a palindrome. Let $\mathbb{P}$ denote the set of all palindromes over Σ.

For any substring palindrome $S[i..j]$ in S, $\frac{i+j}{2}$ is called its *center*. A substring palindrome $S[i..j]$ is said to be a *maximal palindrome* in S if (a) $S[i-1..j+1]$ is not a palindrome (or equivalently $S[i-1] \neq S[j+1]$), (b) $i = 1$, or (c) $j = |S|$. It is clear that for each $c = 1, 1.5, \ldots, |S|$, there is a one-to-one correspondence between $c = \frac{i+j}{2}$ and the maximal palindrome $S[i..j]$. Thus, there are exactly $2|S| - 1$ maximal palindromes in string S.

2.2 Tries and palindromes

A *trie* $\mathcal{T} = (V, E)$ is a rooted tree where each edge in E is labeled by a single character from Σ and the out-going edges of each node are labeled by mutually distinct characters. For any non-root node $\mathbf{u}$ in $\mathcal{T}$, let $\mathsf{parent}(\mathbf{u})$ denote the parent of $\mathbf{u}$. For any node $\mathbf{v}$ in $\mathcal{T}$, let $\mathsf{childchar}(\mathbf{v})$ denote the set of out-going edge labels of $\mathbf{v}$. Let $\mathsf{height}(\mathcal{T})$ denote the height of $\mathcal{T}$. For convenience, we read the path labels in the input trie $\mathcal{T}$ in the leaf-to-root direction. For any path $(\mathbf{u}, \mathbf{v})$, we denote by $\mathsf{str}(\mathbf{u}, \mathbf{v})$ the path label from $\mathbf{u}$ to $\mathbf{v}$. The set $\mathsf{Substr}(\mathcal{T})$ of (reversed) *substrings* in $\mathcal{T}$ is $\mathsf{Substr}(\mathcal{T}) = \{\mathsf{str}(\mathbf{u}, \mathbf{v}) \mid \mathbf{u}, \mathbf{v} \in V, \mathbf{u} \text{ is a descendant of } \mathbf{v}\}$. Note that $\mathbf{v}$ itself is a descendant of $\mathbf{v}$, i.e., the empty string $\varepsilon = \mathsf{str}(\mathbf{v}, \mathbf{v})$ is an element of $\mathsf{Substr}(\mathcal{T})$. For a string w and a node $\mathbf{u}$ in $\mathcal{T}$, the pair $(\mathbf{u}, |w|)$ is called an occurrence of w in $\mathcal{T}$ if w is a prefix of the substring starting at $\mathbf{u}$ in $\mathcal{T}$. Such representations of occurrences allow us to retrieve the substring w in $O(|w|)$ time by taking the unique path from $\mathbf{u}$ towards the root. Also, we enhance the input trie $\mathcal{T}$ with a *level ancestor* data structure [2] so that any character in a given path $(\mathbf{u}, \mathbf{v})$ can be accessed in $O(1)$ time, after linear-time preprocessing.

We can generalize the notions of maximal palindromes and distinct palindromes to tries, as follows. An occurrence of a substring palindrome p which starts at node $\mathbf{u}$ and ends at node $\mathbf{v}$ is called a *maximal palindrome* in $\mathcal{T}$ if (a) $\mathbf{u}$ is not a leaf and $\mathsf{str}(\mathbf{u}', \mathsf{parent}(\mathbf{v}))$ is not a palindrome

with *any* child $\mathbf{u}'$ of $\mathbf{u}$, (b) $\mathbf{u}$ is a leaf, or (c) $\mathbf{v}$ is the root $\mathbf{r}$. Let $\mathsf{MPal}(\mathcal{T})$ be the set of all maximal palindromes in $\mathcal{T}$. For each $1 \leq k \leq \mathsf{height}(\mathcal{T})$, let $\mathsf{MPal}_k(\mathcal{T}) = \{(\mathbf{v}, k) \in \mathsf{MPal}(\mathcal{T})\}$, and let $\mathsf{MPal}_0(\mathcal{T}) = \{\varepsilon\}$. Namely $\bigcup_{0 \leq k \leq \mathsf{height}(\mathcal{T})} \mathsf{MPal}_k(\mathcal{T}) = \mathsf{MPal}(\mathcal{T})$.

For any trie $\mathcal{T}$, let $\mathsf{DPal}(\mathcal{T})$ be the set of all substring palindromes in $\mathcal{T}$, namely $\mathsf{DPal}(\mathcal{T}) = \mathsf{Substr}(\mathcal{T}) \cap \mathbb{P}$. We call the elements of $\mathsf{DPal}(\mathcal{T})$ as *distinct palindromes* in $\mathcal{T}$. For each $0 \leq k \leq \mathsf{height}(\mathcal{T})$, let $\mathsf{DPal}_k(\mathcal{T}) = \mathsf{DPal}(\mathcal{T}) \cap \Sigma^k$. Namely $\bigcup_{0 \leq k \leq \mathsf{height}(\mathcal{T})} \mathsf{DPal}_k(\mathcal{T}) = \mathsf{DPal}(\mathcal{T})$. See Figure 1 for an example of trie $\mathcal{T}$ and its substring palindromes.

▶ **Theorem 1** ([7]). *For any trie $\mathcal{T}$ with n edges and l leaves, $|\mathsf{MPal}(\mathcal{T})| = 2n - l$ and $|\mathsf{DPal}(\mathcal{T})| \leq n + 1$.*

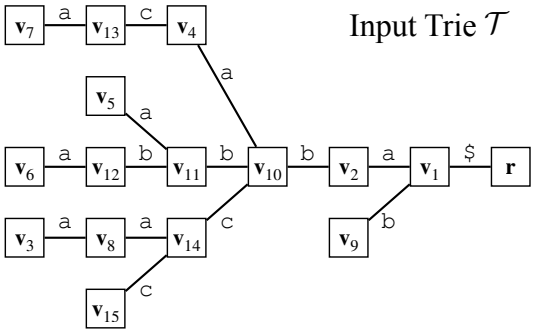

Figure 1 An example for an input trie $\mathcal{T}$ with $n = 15$ edges, in which the path labels are read from the leaves to the root $\mathbf{r}$. The height of $\mathcal{T}$ is $\mathsf{height}(\mathcal{T}) = 6$. Let us focus on the node $\mathbf{v}_{14}$. Then $\mathsf{parent}(\mathbf{v}_{14}) = \mathbf{v}_{10}$, $\mathsf{childchar}(\mathbf{v}_{14}) = \{\mathtt{a}, \mathtt{c}\}$. Also, $\mathsf{str}(\mathbf{v}_{14}, \mathbf{v}_1) = \mathtt{cba}$. The occurrence $(\mathbf{v}_{10}, 1)$ of palindrome $\mathtt{b}$ is not a maximal palindrome since it can be extended to a longer palindrome $\mathsf{str}(\mathbf{v}_4, \mathbf{v}_1) = \mathtt{aba}$. This occurrence $(\mathbf{v}_4, 3)$ of $\mathtt{aba}$ is a maximal palindrome. We have $\mathsf{DPal}(\mathcal{T}) = \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{bb}, \mathtt{cc}, \mathtt{aba}, \mathtt{bbb}, \mathtt{aca}, \mathtt{abba}, \mathtt{abbba}\}$ except for the special character \$.

2.3 Suffix tree and eertree of trie

Suffix tree of a trie. For convenience, we assume that the root $\mathbf{r}$ of the input trie $\mathcal{T}$ has only a single child and its incoming edge from $\mathbf{r}$ is labeled by a special character \$ that does not occur elsewhere in $\mathcal{T}$.

For any node $\mathbf{v}$ in $\mathcal{T}$, let $\mathsf{suf}(\mathbf{v}) = \mathsf{str}(\mathbf{v}, \mathbf{r})$. The set $\mathsf{Suffix}(\mathcal{T})$ of (reversed) suffixes of a given trie $\mathcal{T} = (V, E)$ is $\mathsf{Suffix}(\mathcal{T}) = \{\mathsf{suf}(\mathbf{v}) \mid \mathbf{v} \in V\}$.

The *suffix tree* of $\mathcal{T}$, denoted $\mathsf{STree}(\mathcal{T})$, is a compacted trie such that

- Each edge of $\mathsf{STree}(\mathcal{T})$ is labeled by a non-empty reversed substring of $\mathcal{T}$;
- The labels of the edges from each node of $\mathsf{STree}(\mathcal{T})$ begin with mutually distinct characters;
- There is a one-to-one correspondence between the leaves of $\mathsf{STree}(\mathcal{T})$ and the non-root nodes in $\mathcal{T}$ such that every suffix in $\mathsf{Suffix}(\mathcal{T})$ is represented by a unique leaf in $\mathsf{STree}(\mathcal{T})$.

For any reversed substring w in the input trie $\mathcal{T}$, the *locus* for w in the respective suffix tree $\mathsf{STree}(\mathcal{T})$ is the ending point of the path which spells out w from the root of $\mathsf{STree}(\mathcal{T})$. Note that the locus is either on a node or in the middle of an edge. The loci ending in the middle of edges are called *implicit nodes*, while the loci ending at nodes are called *explicit nodes*. For any implicit or explicit node v of $\mathsf{STree}(\mathcal{T})$, let $\mathsf{str}(v)$ denote the path string from the root of $\mathsf{STree}(\mathcal{T})$ to v. Then, we say that the node v *represents* the substring $\mathsf{str}(v)$.

By the third property of $\mathsf{STree}(\mathcal{T})$, there are exactly n leaves in $\mathsf{STree}(\mathcal{T})$. Since each of the internal explicit nodes has at least two children, there are at most $n-1$ internal nodes in $\mathsf{STree}(\mathcal{T})$. Thus there are at most $2n-1$ nodes and $2n-2$ edges in $\mathsf{STree}(\mathcal{T})$. Each edge label x of $\mathsf{STree}(\mathcal{T})$ is represented by a pair $(\mathbf{u}, \mathbf{v})$ of nodes in $\mathcal{T}$ such that $x = \mathsf{str}(\mathbf{u}, \mathbf{v})$. This allows us to represent $\mathsf{STree}(\mathcal{T})$ in $O(n)$ space. See Figure 2 for an example $\mathsf{STree}(\mathcal{T})$.

In what follows, we use a common assumption that our alphabet is an integer alphabet of polynomial size in n, where $n > 1$ is the number of edges (and thus the number of characters) in the input trie $\mathcal{T}$. We then sort the characters appearing in the input trie $\mathcal{T}$ in $O(n)$ time by radix-sort, and replace each character appearing in the input trie $\mathcal{T}$ with its lexicographical rank. This ensures that we can work on the integer alphabet $\Sigma = [1..\sigma]$, where $\sigma \leq n$ denotes the number of distinct characters in $\mathcal{T}$. Then $\mathsf{STree}(\mathcal{T})$ can be built in linear time:

▶ **Theorem 2** ([16]). *For any trie $\mathcal{T}$ with n edges where edge labels are drawn from an integer alphabet $[1..n]$, $\mathsf{STree}(\mathcal{T})$ can be built in $O(n)$ time and working space.*

We remark that the algorithm of Theorem 2 builds the *edge-sorted* suffix tree for a given trie $\mathcal{T}$, in which the out-going edges of each node are sorted in lexicographical order. Thus, we can naturally assume that the leaves in $\mathsf{STree}(\mathcal{T})$ are arranged in the lexicographical order from left to right. For each $1 \leq i \leq n$, let ℓ_i denote the ith leaf in $\mathsf{STree}(\mathcal{T})$ which represents the lexicographically ith suffix in $\mathsf{Suffix}(\mathcal{T})$. For each $1 \leq i \leq n$, let $\mathbf{v}_i$ denote the node in the input trie $\mathcal{T}$ that corresponds to the leaf ℓ_i, namely $\mathsf{suf}(\mathbf{v}_i) = \mathsf{str}(\ell_i)$ (see Figure 2 for examples).

The *suffix link* of any non-root explicit node v in $\mathsf{STree}(\mathcal{T})$ points to the explicit node u representing the longest proper suffix of $\mathsf{str}(v)$, i.e. $\mathsf{str}(u) = \mathsf{str}(v)[2..|\mathsf{str}(v)|]$, and we denote this by $\mathsf{SL}(v) = u$. Note that the destination explicit node u always exists. Also, by the definition, for each leaf ℓ_i in the suffix tree, $\mathsf{SL}(\ell_i) = \ell_j$ if and only if $\mathsf{parent}(\mathbf{v}_i) = \mathbf{v}_j$. The first character $\mathsf{str}(v)[1]$ that is dropped when moving from v to u is called the *suffix link label* of $\mathsf{SL}(v)$.

In our algorithms to follow, we will use only the suffix links of the leaves of $\mathsf{STree}(\mathcal{T})$. Since every suffix of $\mathcal{T}$ ends with the unique end-marker \$, the suffix link of a leaf points to another leaf, except for the suffix link of the leaf representing \$ which points to the root. For any leaf ℓ_i in $\mathsf{STree}(\mathcal{T})$, let $\mathsf{inSLchar}(\ell_i)$ denote the set of in-coming suffix link labels. Notice that $\mathsf{inSLchar}(\ell_i) = \mathsf{childchar}(\mathbf{v}_i)$ holds since $\mathsf{SL}(\ell_j) = \ell_i$ iff $\mathsf{parent}(\mathbf{v}_j) = \mathbf{v}_i$. See Figure 2 for examples. For any node v in $\mathsf{STree}(\mathcal{T})$, let $\mathsf{subtree}(v)$ denote the subtree rooted at v, and let $\mathsf{leaves}(v)$ denote the set of leaves in $\mathsf{subtree}(v)$. Given $\mathsf{STree}(\mathcal{T})$, it is easy to compute the suffix links of all leaves of $\mathsf{STree}(\mathcal{T})$ in $O(n)$ time. Thus we can also compute $\mathsf{inSLchar}(\ell)$ for every leaf ℓ in $\mathsf{STree}(\mathcal{T})$ in a total of $O(n)$ time.

We extend the notion of $\mathsf{childchar}(\cdot)$ to the suffix tree so that for each node v in $\mathsf{STree}(\mathcal{T})$, $\mathsf{childchar}(v)$ denotes the set of the first characters of the out-going edge labels of v.

Eertree of a trie. We can naturally generalize the *eertree* (a.k.a. *palindromic tree*) which represents the distinct palindromes of a single string [15] to the case of a trie $\mathcal{T}$, as follows: The eertree for a trie $\mathcal{T}$, denoted $\mathsf{eertree}(\mathcal{T})$, is a pair of two rooted tries (the even-tree and odd-tree) where the root of the even-tree represents ε and the root of the odd-tree is an auxiliary node $\perp$. Then, for each $p \in \mathsf{DPal}(\mathcal{T})$, there is a path that spells out $p[\lfloor |p|/2 \rfloor + 1..|p|]$ from the root of the even-tree if $|p|$ is even, or from the root of the odd-tree otherwise. The *suffix link* of a node representing a palindrome p in $\mathsf{eertree}(\mathcal{T})$ points to the node that represents the longest palindrome q, which is a proper suffix of p (the suffix links may bridge the nodes of the even-tree and the nodes of the odd-tree). The suffix links of the two roots ε and $\perp$ both point to $\perp$. See Figure 3 for an example of the eertree for a trie.

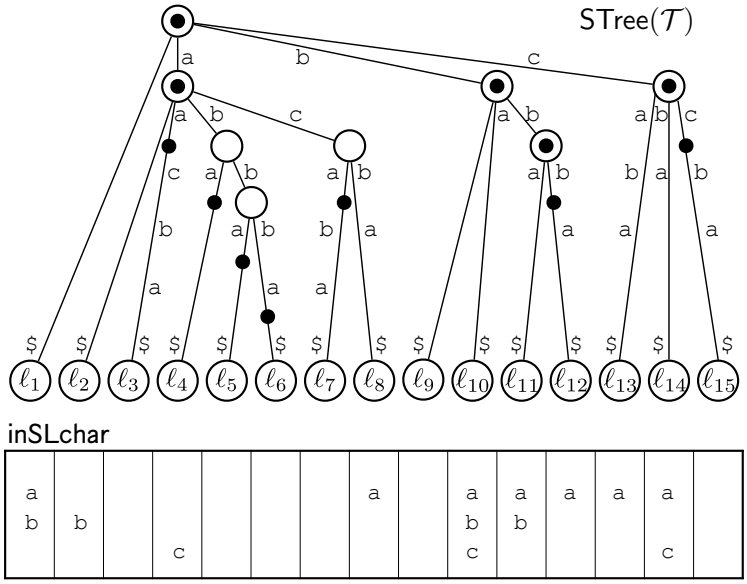

Figure 2 The suffix tree $\mathsf{STree}(\mathcal{T})$ of the input trie $\mathcal{T}$ of Figure 1. The explicit nodes are depicted by the white circles, while the palindromic nodes representing the elements of $\mathsf{DPal}(\mathcal{T})$ are depicted by the black circles. The sets $\mathsf{inSLchar}(\ell_i)$ for all ℓ_i are depicted below the leaves. Each $\mathsf{inSLchar}(\ell_i)$ stores the in-coming suffix link labels of leaf ℓ_i. The suffix links are omitted in this figure, however, we can verify this $\mathsf{inSLchar}$ is correct by looking at trie $\mathcal{T}$ since $\mathsf{inSLchar}(\ell_i) = \mathsf{childchar}(\mathbf{v}_i)$ holds.

3 Algorithm for computing $\mathsf{DPal}(\mathcal{T})$ and $\mathsf{MPal}(\mathcal{T})$

This section presents our algorithm for computing $\mathsf{MPal}(\mathcal{T})$ and $\mathsf{DPal}(\mathcal{T})$ for a given trie $\mathcal{T}$.

3.1 Overview of our algorithm

We use $\mathsf{STree}(\mathcal{T})$ as our main tool for computing $\mathsf{DPal}(\mathcal{T})$ and $\mathsf{MPal}(\mathcal{T})$. An (implicit or explicit) node v on $\mathsf{STree}(\mathcal{T})$ such that $\mathsf{str}(v) \in \mathbb{P}$ is called a *palindromic node*. See also Figure 2, in which the palindromic nodes are depicted by the black circles. By Theorem 1, there are exactly $|\mathsf{DPal}(\mathcal{T})| \in O(n)$ palindromic nodes in $\mathsf{STree}(\mathcal{T})$, and thus, computing $\mathsf{DPal}(\mathcal{T})$ can be reduced to computing all palindromic nodes in $\mathsf{STree}(\mathcal{T})$.

We compute $\mathsf{DPal}(\mathcal{T})$ together with $\mathsf{MPal}(\mathcal{T})$ in increasing order of the lengths of the palindromes. Namely, we visit all palindromic nodes in $\mathsf{STree}(\mathcal{T})$, in the level-wise breadth first manner. Also, for each found palindrome p in increasing order of their length, we check whether it has an occurrence as a maximal palindrome in $\mathcal{T}$. This can be done as follows:

Our strategy for computing $\mathsf{DPal}(\mathcal{T})$ and $\mathsf{MPal}(\mathcal{T})$

- **Base steps:** Start from the root $\mathbf{r}$ of $\mathsf{STree}(\mathcal{T})$. Report ε as a member of $\mathsf{DPal}_0(\mathcal{T})$, and report all single characters occurring in $\mathcal{T}$, as members of $\mathsf{DPal}_1(\mathcal{T})$.
- **Inductive step:** For each increasing $k = 0, 1, \ldots, \mathsf{height}(\mathcal{T}) - 1$:
  - For each palindrome $p \in \mathsf{DPal}_k(\mathcal{T})$, let v be the locus for p in $\mathsf{STree}(\mathcal{T})$.
    - For each character $c \in \mathsf{childchar}(v)$, perform the following:
      - If $cpc \in \mathsf{Substr}(\mathcal{T})$, then report this new palindrome cpc of length $k+2$ as a new member of $\mathsf{DPal}_{k+2}(\mathcal{T})$.
      - For each occurrence of pc that is not preceded by c in the reversed trie $\mathcal{T}$, report this occurrence of p as a new member of $\mathsf{MPal}_k(\mathcal{T})$.

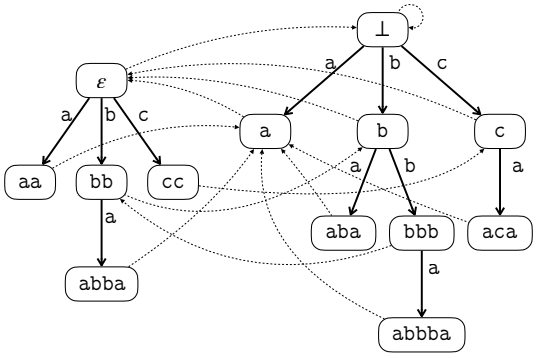

Figure 3 The palindromic tree $\mathsf{eertree}(\mathcal{T})$ for the input trie $\mathcal{T}$ of Figure 1, which represents $\mathsf{DPal}(\mathcal{T}) = \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{bb}, \mathtt{cc}, \mathtt{aba}, \mathtt{bbb}, \mathtt{aca}, \mathtt{abba}, \mathtt{abbba}\}$. The bold arcs represent the edges and the dotted arcs represent the suffix links.

Given a palindrome p of length k together with the corresponding palindromic node v in $\mathsf{STree}(\mathcal{T})$ and $c \in \mathsf{childchar}(v)$, our task is to efficiently check whether or not $cpc \in \mathsf{Substr}(\mathcal{T})$, and if so, then find the locus for cpc in $\mathsf{STree}(\mathcal{T})$. In the sequel, for simplicity, we identify the strings p and pc with their loci (implicit or explicit nodes) in $\mathsf{STree}(\mathcal{T})$.

For a palindrome p and a character c, $cpc \in \mathsf{Substr}(\mathcal{T})$ iff there exists a node $\mathbf{v}_i \in \mathcal{T}$ such that $\mathsf{suf}(\mathbf{v}_i)[1..|p|+1] = pc$ and $\mathbf{v}_i$ has a child that is reachable with character c. Recalling that this is equivalent to that $c \in \mathsf{inSLchar}(\ell_i)$ with the suffix tree leaf ℓ_i that corresponds to $\mathbf{v}_i$, we obtain the two following observations (see also Figure 4).

▶ **Observation 3.** *Let p be a locus in $\mathsf{STree}(\mathcal{T})$ such that $p \in \mathsf{DPal}_k(\mathcal{T})$, and let $c \in \mathsf{childchar}(p)$. Then, $cpc \in \mathsf{DPal}_{k+2}(\mathcal{T})$ iff there exists a suffix tree leaf $\ell \in \mathsf{leaves}(pc)$ such that $c \in \mathsf{inSLchar}(\ell)$.*

▶ **Observation 4.** *Let p be a locus in $\mathsf{STree}(\mathcal{T})$ such that $p \in \mathsf{DPal}_k(\mathcal{T})$, and let $c \in \mathsf{childchar}(p)$. Then, $(\mathbf{v}_i, |p|) \in \mathsf{MPal}_k(\mathcal{T})$ iff $\ell_i \in \mathsf{leaves}(pc)$ and $c \notin \mathsf{inSLchar}(\ell_i)$, where ℓ_i is the suffix tree leaf corresponding to $\mathbf{v}_i$.*

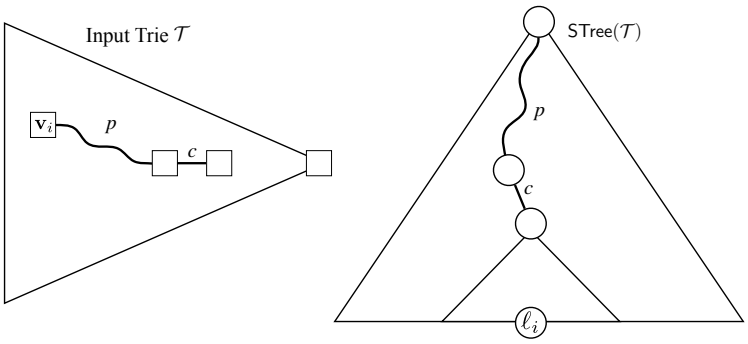

Figure 4 Illustration for Observations 3 and 4.

Following Observation 3 and Observation 4, our task is, given a palindromic node p in $\mathsf{STree}(\mathcal{T})$ and $c \in \mathsf{childchar}(p)$, to perform the following task in *(amortized) constant time.*

(1) Check whether there exists a leaf ℓ under the locus for pc such that $\mathsf{inSLchar}(\ell)$ contains character c, and if so, find it. Then, locate the locus for the extended palindrome cpc in $\mathsf{STree}(\mathcal{T})$.

(2) Check whether there exists a leaf ℓ under the locus for pc such that $\mathsf{inSLchar}(\ell)$ does not contain character c, and if so, retrieve all and only such leaves by skipping the other leaves which do not correspond to the output with respect to pc.

In what follows, we show the details of our algorithm that achieves the above goal, which consists of the preprocessing phase and the computing phase.

3.2 Preprocessing phase

We first enhance $\mathsf{STree}(\mathcal{T})$ with a lowest common ancestor (LCA) data structure in linear preprocessing time so that the LCA of two given nodes can be found in $O(1)$ time [1]. We then compute the *jump-arrays* and some links toward leaves, as follows.

destSL arrays and jump-arrays. For each character c occurring in $\mathcal{T}$, let destSL_c be an array such that $\mathsf{destSL}_c[r] = i$ if the leaf ℓ_i is the lexicographically r-th leaf that has an in-coming suffix link labeled c. For convenience, we add an extra element at the end of destSL_c that stores ∞. Also, we sometimes regard destSL_c as a list of the corresponding leaves ℓ_i. We then define the *jump-array* jump_c which leads us to the ending point of the run of adjacent leaves in $\mathsf{STree}(\mathcal{T})$ for each given leaf ℓ_i in destSL_c. Formally, $\mathsf{jump}_c[i] = i + 1$ if $\mathsf{destSL}_c[i{+}1] - \mathsf{destSL}_c[i] > 1$, and $\mathsf{jump}_c[i] = \min\{j \mid j > i \text{ and } \mathsf{destSL}_c[j{+}1] - \mathsf{destSL}_c[j] > 1\}$ otherwise. See Figure 5 for examples. Note that the total sizes of destSL_c and jump_c for all c occurring in $\mathcal{T}$ is $O(n)$ since these are linear in the number of suffix links of the leaves. Also, we can compute the arrays destSL_c and jump_c for all characters c occurring in $\mathcal{T}$ in linear total time, by radix-sorting all the pairs (c, i) of a character and a leaf-index such that $c \in \mathsf{inSLchar}(\ell_i)$. Simultaneously, we associate each pair (c, i) with its *rank* in destSL_c, i.e., the integer r with $\mathsf{destSL}_c[r] = i$.

Links toward leaves. For each non-leaf node v in $\mathsf{STree}(\mathcal{T})$, we store links from v to the leftmost and the rightmost leaves in $\mathsf{leaves}(v)$. Note that we can compute them easily by performing a depth-first traversal on $\mathsf{STree}(\mathcal{T})$. We further store another link defined below in every non-root node v. For each non-root node v in $\mathsf{STree}(\mathcal{T})$, let c_v be the first character of its in-coming edge label, and further let $\mathsf{leftDest}(v)$ be (some representation of) the leftmost leaf in $\mathsf{leaves}(v)$ such that its leaf-index is in destSL_{c_v} if such a leaf exists, and *nil* otherwise. For technical reasons, we implement $\mathsf{leftDest}(v)$ as the index of the entry of destSL_{c_v} corresponding to the specified leaf. See Figure 5 for examples. Whenever we access an entry of destSL_c, we always take this link from a given node v such that $c_v = c$. Our implementation of $\mathsf{leftDest}(v)$ ensures that we can access the corresponding entry in $O(1)$ time, without the need to search the array $\mathsf{inSLchar}(\ell)$ on some leaf $\ell \in \mathsf{leaves}(v)$ for character c_v, which would use $O(\log \sigma)$ time. Next, we show how to compute the $\mathsf{leftDest}$ for all nodes in linear total time.

As a warm-up, we fix a character c and describe an algorithm which computes $\mathsf{leftDest}(v)$ for all nodes v such that $c_v = c$, in $O(n)$ time. We execute left-to-right depth-first traversal on $\mathsf{STree}(\mathcal{T})$ with a stack st_c storing nodes. When we descend to a node v, push v to st_c if $c_v = c$. When we ascend from a node v, pop v and set $\mathsf{leftDest}(v) \leftarrow nil$ if the top of st_c is v. When we arrive at a leaf ℓ_k, scan $\mathsf{inSLchar}(\ell_k)$ and if $c \in \mathsf{inSLchar}(\ell_k)$ (i.e., $k \in \mathsf{destSL}_c$), pop all nodes v in st_c, and set $\mathsf{leftDest}(v) \leftarrow r$ with $\mathsf{destSL}_c[r] = k$. Recall that such index r has been associated with (c, k) for $c \in \mathsf{inSLchar}(\ell_k)$. During the traversal, st_c stores all nodes v on the path from the root to the current node s.t. $c_v = c$ but $\mathsf{leftDest}(v)$ is not determined yet. Thus, when arriving at a leaf ℓ_k with $c \in \mathsf{inSLchar}(\ell_k)$, we can correctly compute $\mathsf{leftDest}(v)$ for all v s.t. $\mathsf{destSL}_c[\mathsf{leftDest}(v)] = k$.

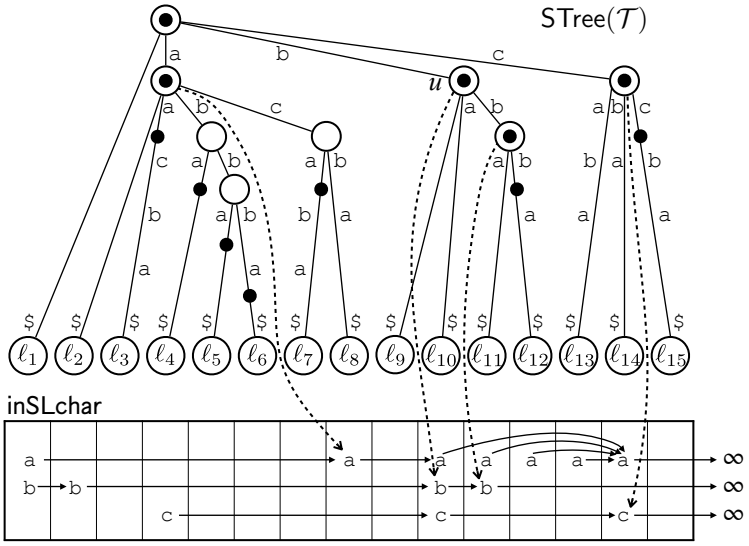

Figure 5 The suffix tree $\mathsf{STree}(\mathcal{T})$ of Figure 2 with some auxiliary data structures. By reading array $\mathsf{inSLchar}$ horizontally for each character c, we can obtain the array destSL_c. Here, $\mathsf{destSL_a} = (1, 8, 19, 10, 11, 12, 13, 14, \infty)$, $\mathsf{destSL_b} = (1, 2, 10, 11, \infty)$, and $\mathsf{destSL_c} = (4, 10, 14, \infty)$. Each element in destSL_c is represented by the character c and is aligned along the corresponding leaf. Also, the arc from each element of destSL_c represents the value stored in the corresponding entry of the jump_c array. Each dotted arrow from an internal node v denotes $\mathsf{leftDest}(v)$. For example, $\mathsf{leftDest}(u)$ of the node u representing string $\mathtt{b}$ points to the third entry of $\mathsf{destSL_b}$, which is represented by the $\mathtt{b}$ under the leaf ℓ_{10}, since the first character c_u of the label of u's parent edge is $c_u = \mathtt{b}$, and the leaf ℓ_{10} is the leftmost leaf in $\mathsf{leaves}(u)$ such that its leaf-index is in $\mathsf{destSL_b}$.

Now, let us remove the constraint of fixing the character c, and perform linear-time computation of $\mathsf{leftDest}(v)$ for all internal nodes v. To compute $\mathsf{leftDest}(v)$ for *all* non-root nodes v, we run the above algorithm for all $c \in \Sigma$ simultaneously in a single depth-first traversal. For this, we use a length-n array $\mathbf{S}$ of stacks in which each entry $\mathbf{S}[c]$ plays the role of st_c as the above. While performing a left-to-right depth-first traversal, when we descend to a node v, push v to $\mathbf{S}[c_v]$, and when we ascend from a node v, pop v and set $\mathsf{leftDest}(v) \leftarrow nil$ if the top of $\mathbf{S}[c_v]$ is v. When we arrive at a leaf ℓ_k, then scan $\mathsf{inSLchar}(\ell_k)$ and for every $c \in \mathsf{inSLchar}(\ell_k)$, pop all nodes v in $\mathbf{S}[c]$ and set $\mathsf{leftDest}(v) \leftarrow r$ with $\mathsf{destSL}_c[r] = k$. The correctness relies on the aforementioned algorithm, and the running time is linear in the size of $\mathsf{STree}(\mathcal{T})$ including the suffix links, that is, $O(n)$.

3.3 Computing phase

In the computing phase, we traverse on $\mathsf{STree}(\mathcal{T})$ and visit the loci representing palindromes in non-decreasing order of the lengths of palindromes. Let us assume that we are now located at a node representing a palindrome p occurring in $\mathcal{T}$. There are the two following cases: (A) node p is an explicit node, and (B) node p is an implicit node.

Algorithm for case (A)

For the case (A), we can enumerate all trie nodes whose length-$|p|$ prefixes are maximal palindromes p in output-sensitive time by using data structures precomputed. Let v be a child of node p and c be the first character of the label of edge (p, v). Let ℓ_{left} and ℓ_{right} be the leftmost and the rightmost leaves in $\mathsf{leaves}(v)$, respectively. Namely, they correspond

to the *left*-th and *right*-th smallest suffixes in $\mathcal{T}$, respectively. We initialize the current-leaf-index $curr \leftarrow \mathsf{destSL}_{c_v}[\mathsf{leftDest}(v)]$ and the previous-leaf-index $prev \leftarrow left - 1$. While $curr \leq right$, output trie nodes $\mathbf{v}_i$ for all i with $prev < i < curr$ if $curr - 1 \notin \mathsf{destSL}_c$, and then set $prev \leftarrow curr$ and $curr \leftarrow \mathsf{destSL}_c[\mathsf{jump}_c[curr]]$. At last, after breaking out from the while-loop condition, output trie nodes $\mathbf{v}_i$ for all i with $prev < i \leq right$ if $curr - 1 \notin \mathsf{destSL}_c$. See also Figure 6 for illustration.

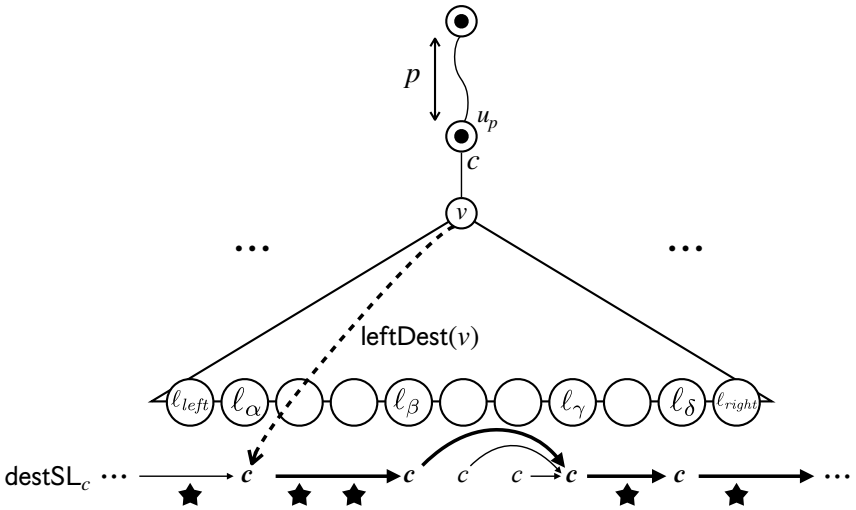

Figure 6 The black stars indicate the leaves in $\mathsf{leaves}(v)$ corresponding to occurrences of palindrome p which are followed by character c but cannot be expanded with c, i.e., occurrences of p as maximal palindromes. In the computing phase, starting from node v, we visit the entries in destSL_c corresponding to the leaves ℓ_α, ℓ_β, ℓ_γ and ℓ_δ in this order, and output all the leaves indicated by black stats. The jump-arrays values that are used in this step are depicted in bold. Note that when we arrive at ℓ_γ, we output nothing since none of the leaves between ℓ_β and ℓ_γ inclusive corresponds to an occurrence of a maximal palindrome.

By Observation 4, this algorithm correctly reports all the occurrences of pc (without duplication) such that the occurrence of p is a maximal palindrome. Next, to evaluate the running time, we show the next lemma:

▶ **Lemma 5.** *In the above algorithm for a node v, when we use jump-pointers three times, at least one maximal palindrome is reported.*

Proof. We assume that we use jump-pointers at least three times on leaves in $\mathsf{subtree}(v)$. Then, at least two jumps are performed inside $\mathsf{subtree}(v)$. (The last jump, which is toward the outside of $\mathsf{subtree}(v)$, is the only exception.)

Let i be the current leaf-index. By the definition of jump-pointers, if $\mathsf{destSL}_c[i+1] - \mathsf{destSL}_c[i] > 1$, then $\mathsf{jump}_c[i] = i+1$. Then, our algorithm reports at least one maximal palindrome since we skip at least one leaf between $\mathsf{destSL}_c[i]$ and $\mathsf{destSL}_c[\mathsf{jump}_c[i]] = \mathsf{destSL}_c[i+1]$ both exclusive. Otherwise, namely if $\mathsf{destSL}_c[i+1] - \mathsf{destSL}_c[i] = 1$, then $\mathsf{jump}_c[i]$ points to the ending point of the run of adjacent leaves in destSL_c. In this case, we report no maximal palindromes. However, in the next step, we track $\mathsf{jump}_c[j]$ where $j = \mathsf{jump}_c[i]$. Then, we fall into the first case $\mathsf{destSL}_c[j+1] - \mathsf{destSL}_c[j] > 1$ since j is the ending position of a run (see also Figure 6). Thus, a maximal palindrome is reported while the second jump. Therefore, performing jumps twice inside $\mathsf{subtree}(v)$ results in a maximal palindrome being reported. ◀

Thus, the running time of the above algorithm for node v is linear in the output size, or is constant if there is no maximal palindrome to output.

After these processes for v, we retrieve the node representing palindrome cpc if it exists. While computing maximal palindromes we can obtain the rightmost leaf ℓ_R such that $\ell_R \in \mathsf{leaves}(v)$ and $R \in \mathsf{destSL}_c$. We move to the source leaves ℓ and ℓ' of the suffix links which respectively point to ℓ_L and to ℓ_R, both labeled by c where $L = \mathsf{destSL}_{c_v}[\mathsf{leftDest}(v)]$ is the leaf-index specified by $\mathsf{leftDest}(v)$. We then compute the LCA u' of ℓ and ℓ'. If the string-depth of u' equals $|cpc|$, the node u' is the desired node corresponding to cpc. Otherwise, the locus on the parent-edge of u' with the string-depth of $|cpc|$ is the (implicit) node corresponding to cpc.

Algorithm for case (B)

For the case (B), the precomputed data structures are not helpful. Instead, we employ the following lemmas for designing an efficient algorithm.

▶ **Lemma 6.** *For any trie $\mathcal{T}$, there is at most one implicit node v on each edge of $\mathsf{STree}(\mathcal{T})$ such that $\mathsf{str}(v)$ is a palindrome.*

Proof. Suppose on the contrary that there are two implicit nodes v and u on the same edge of $\mathsf{STree}(\mathcal{T})$ such that both $\mathsf{str}(v)$ and $\mathsf{str}(u)$ are palindromes. Assume without loss of generality that $|\mathsf{str}(v)| < |\mathsf{str}(u)|$. Since v and u are on the same edge, we have $\mathsf{leaves}(v) = \mathsf{leaves}(u)$ which means that the number of occurrences of $\mathsf{str}(v)$ and $\mathsf{str}(u)$ in $\mathcal{T}$ must be equal. However, since $\mathsf{str}(v)$ is a proper prefix of $\mathsf{str}(u)$ and both of them are palindromes, $\mathsf{str}(v)$ is also a proper suffix of $\mathsf{str}(u)$, meaning that $\mathsf{str}(v)$ occurs at least twice in $\mathsf{str}(u)$. This contradicts that the number of occurrences of $\mathsf{str}(v)$ and $\mathsf{str}(u)$ in $\mathcal{T}$ are equal. Thus there exists at most one implicit node on each edge of $\mathsf{STree}(\mathcal{T})$. ◀

The following lemma states that implicit palindromic nodes have a sort of monotonicity. See also Figure 7 for illustration.

▶ **Lemma 7.** *Let p and q be palindromes such that p is a proper prefix of q. If p corresponds to an implicit node on an edge (u, v) in $\mathsf{STree}(\mathcal{T})$ and its following character is c, then q corresponds to an implicit node on an edge in $\mathsf{subtree}(v)$ and its following character is also c.*

Proof. Since p corresponds to an implicit node, the only character following p is c for all occurrences of p in $\mathcal{T}$. Also, since the palindrome p is a proper prefix of the palindrome q, p occurs in q also as a proper suffix. If we assume that q corresponds to an explicit node, then there have to be two distinct characters following q, and p as well, a contradiction. Thus, q corresponds to an implicit node and its following character is c. Also, by Lemma 6, the implicit node cannot exist on the edge (u, v), and thus, it is in $\mathsf{subtree}(v)$. ◀

Namely, once an implicit node u corresponding to a palindrome is found, any palindrome corresponding to a descendant of u is guaranteed to be followed by the same character (see also Figure 7). Based on this monotonicity, we design an algorithm for the case (B).

At each step of the algorithm, $v.\mathsf{mark} = 1$ means that we already visited an implicit ancestor representing some shorter prefix palindrome. Initially, we set $u.\mathsf{mark} \leftarrow 0$ for every node u. Let v be the shallowest explicit node below p and c be the character following p. If $v.\mathsf{mark} = 0$, every $v' \in \mathsf{subtree}(v)$ is not marked yet since we consider palindromes in non-decreasing order of the lengths. We traverse $\mathsf{subtree}(v)$ and mark all nodes in the subtree. Also, while traversing the subtree, we compute the links $\mathsf{left}(v')$ from each $v' \in \mathsf{subtree}(v')$ defined as follows: $\mathsf{left}(v')$ is the leftmost leaf in $\mathsf{leaves}(v')$ such that its leaf-index is in destSL_c if such a leaf exists, and *nil* otherwise. In other words, $\mathsf{left}(v')$ is a variant of $\mathsf{leftDest}(v')$

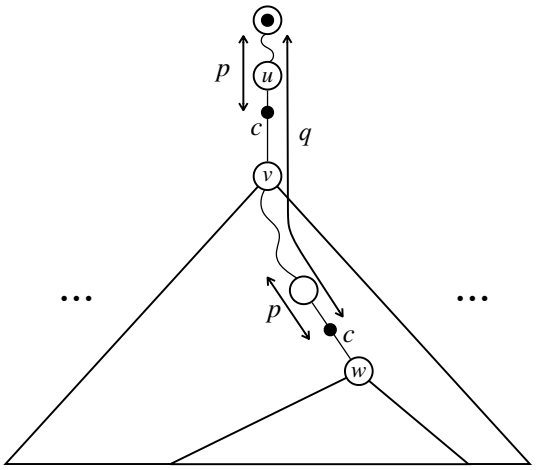

Figure 7 Illustration for the case (B). In our algorithm, we visit the locus of palindrome p before that of palindrome q since q is longer than p. After we have visited the locus of p, each node w' in $\mathsf{subtree}(v)$ have links $\mathsf{left}(w')$ w.r.t. the character c. When we visit the locus of q later, we use $\mathsf{left}(w)$ and destSL_c to report all occurrences of q which are maximal palindromes.

where the character c to be considered is fixed. Traversing $\mathsf{subtree}(v)$ and computing $\mathsf{left}(v')$ for all nodes v' in $\mathsf{subtree}(v)$ can be done in time linear in the size of $\mathsf{subtree}(v)$ plus the total number of incoming suffix links of $\mathsf{leaves}(v)$ as in the computing of $\mathsf{leftDest}$ described in the previous subsection. Once we create the links left, we can compute all occurrences of maximal palindrome p and find the next node for cpc in a similar way to the case (A).

If $v.\mathsf{mark} = 1$, there is an implicit ancestor we already visited. By the above procedures, node v already has link $\mathsf{left}(v)$, and hence, we can output all maximal palindromes p in output linear time. To summarize, the following theorem holds:

▶ **Theorem 8.** *For a given trie $\mathcal{T}$ with n edges over an integer alphabet $\Sigma = [1..n]$, $\mathsf{MPal}(\mathcal{T})$ and $\mathsf{DPal}(\mathcal{T})$ can be computed in $O(n)$ time and space.*

Proof. In the preprocessing phase, $\mathsf{STree}(\mathcal{T})$ can be constructed in $O(n)$ time and space (Theorem 2). An LCA data structure on $\mathsf{STree}(\mathcal{T})$ can be built in $O(n)$ time and space [1], and the remaining data structures can also be built in $O(n)$ time by radix-sort and depth-first traversal on $\mathsf{STree}(\mathcal{T})$.

For the case (A) in the computing phase, most of the procedures can be done in constant time per an edge of $\mathsf{STree}(\mathcal{T})$. The exception is the number of times jump-arrays are used can not be bounded by a constant per an edge. However, by Lemma 5, that can be charged to the maximal palindromes to output, and thus, the total execution time is $O(n + |\mathsf{MPal}(\mathcal{T})|) = O(n)$. For the case (B) in the computing phase, traversing the subtree of a node appears to be the bottleneck. However, the total time to traverse the subtrees is $O(n)$ since each node will be marked at most once throughout the algorithm. The remaining procedures can be done in a total of $O(n)$ time (the analysis is the same as that for the case (A)). ◀

3.4 Computing eertree($\mathcal{T}$)

▶ **Theorem 9.** *For a given trie $\mathcal{T}$ with n edges over an integer alphabet $\Sigma = [1..n]$, the edge-sorted $\mathsf{eertree}(\mathcal{T})$ with suffix links can be computed in $O(n)$ time and space.*

Proof. Since our algorithm computes $\mathsf{DPal}(\mathcal{T})$ in increasing order of the lengths, we can easily compute the tree-topology of $\mathsf{eertree}(\mathcal{T})$ in conjunction with $\mathsf{DPal}(\mathcal{T})$ in $O(n)$ time and space. Recall that the algorithm of Theorem 2 builds the edge-sorted suffix tree. Thus, the edges of the obtained $\mathsf{eertree}(\mathcal{T})$ are already sorted.

Recall that a palindrome q is the longest proper suffix palindrome of another palindrome p iff q is the longest proper prefix palindrome of p. Thus, we can compute the suffix links of the nodes of $\mathsf{eertree}(\mathcal{T})$ by a standard traversal on $\mathsf{STree}(\mathcal{T})$ in which all the palindromic nodes are explicitly inserted as in Figure 2. ◀

4 Answering queries to find palindromes in sub-paths

A trie $\mathcal{T}$ can be regarded as a compact representation of a set $\mathcal{S}_{\mathcal{T}}$ of strings that are the path strings from the leaves to the root of $\mathcal{T}$. This section presents how to report all distinct/maximal palindromes in each string in $\mathcal{S}_{\mathcal{T}}$ upon query, in output-linear time with $O(n)$ space, where n is the size of $\mathcal{T}$. We remark that the total length of the strings in $\mathcal{S}_{\mathcal{T}}$ can be as large as $O(n^2)$, and thus, our $O(n)$-size representation of the palindromes for $\mathcal{S}_{\mathcal{T}}$ is space-efficient. In addition, our algorithms which follow permit us to query on any sub-path for maximal palindromes, and on any suffix-path for distinct palindromes.

4.1 Maximal palindromes in a sub-path

As for maximal palindromes in any path of a given trie, we obtain the following result:

▶ **Theorem 10.** *After $O(n)$-time preprocessing on the input trie $\mathcal{T}$, given a sub-path $(\mathbf{u}, \mathbf{v})$ in $\mathcal{T}$ and a position α that is either a node or an edge on the path $(\mathbf{u}, \mathbf{v})$, the maximal palindrome centered at α in string $\mathsf{str}(\mathbf{u}, \mathbf{v}) \in \mathsf{Substr}(\mathcal{T})$ can be computed in $O(1)$ time.*

Proof. In the preprocessing, we transform the occurrence representation of maximal palindromes into the pair $(\mathbf{s}, \alpha)$ of the starting node $\mathbf{s}$ and its center α, which is either a node or an edge on the suffix-path from $\mathbf{s}$. Such transformation can be done in linear time by traversing $\mathcal{T}$: We maintain an array $\mathcal{N}$ of size $\mathsf{height}(\mathcal{T})$ that stores nodes in the suffix path from the current node so that we can access any ancestor of given depth in constant time. When we visit a node $\mathbf{v}$ and find a maximal palindrome $(\mathbf{v}, k)$, we can compute the $k/2$-th shallower locus α by using array $\mathcal{N}$ in constant time. Simultaneously, we store the pointer from each α to the starting node $\mathbf{s}_\alpha$ of the maximal palindrome centered at α.

Given nodes $\mathbf{u}$, $\mathbf{v}$ and center position α as a query, we first compute the lowest common ancestor of $\mathbf{s}_\alpha$ and $\mathbf{u}$. Let $\mathbf{w}$ be the LCA node. If the center position α is a node $\mathbf{z}$, then the length of the maximal palindrome centered at α in $\mathsf{str}(\mathbf{u}, \mathbf{v})$ is $\min\{2|(\mathbf{w}, \mathbf{z})|, 2|(\mathbf{z}, \mathbf{v})|\}$. Otherwise, namely, if the center position α is an edge $(\mathbf{x}, \mathbf{y})$, then the length of the maximal palindrome centered at α in $\mathsf{str}(\mathbf{u}, \mathbf{v})$ is $\min\{2|(\mathbf{w}, \mathbf{x})|, 2|(\mathbf{y}, \mathbf{v})|\} + 1$. ◀

4.2 Distinct palindromes in a suffix-path

In general, a substring palindrome p in a string S may occur more than once in S. It suffices to report a representative of the occurrences of each palindrome for computing the set of distinct palindromes. The algorithm that computes $\mathsf{DPal}(S)$ for a string S in [10] reports their leftmost occurrences in S. Also, the algorithm that computes $\mathsf{DPal}(\mathcal{T})$ for a trie $\mathcal{T}$ in [7] reports the first occurrence of each palindrome during a depth-first traversal in $\mathcal{T}$. Below, we show how to efficiently report $\mathsf{DPal}(w)$ for any suffix-path string w in the trie $\mathcal{T}$.

▶ **Theorem 11.** *After $O(n)$-time preprocessing on the input trie $\mathcal{T}$, given a node $\mathbf{u}$ in $\mathcal{T}$, all the distinct palindromes in the suffix-path string $\mathsf{suf}(\mathbf{u}) \in \mathsf{Suffix}(\mathcal{T})$ can be enumerated in output-linear time.*

Proof. An occurrence $(\mathbf{v}, |w'|)$ of string w' is called a *rightmost* occurrence in $\mathcal{T}$ if there is no occurrence of w' in $\mathsf{suf}(\mathbf{v})$ other than $(\mathbf{v}, |w'|)$. As the representative of the occurrences of a palindrome p in $\mathcal{T}$, we choose a rightmost one. Our task is to enumerate rightmost occurrences of palindromes in a given suffix $\mathsf{suf}(\mathbf{u}) \in \mathsf{Suffix}(\mathcal{T})$. Let $lpp_{\mathbf{v}}$ be the longest prefix palindrome starting at a trie node $\mathbf{v}$. If $(\mathbf{v}, |p|)$ is a rightmost occurrence of a palindrome p, then $p = lpp_{\mathbf{v}}$ holds (if not, p occurs in $\mathsf{suf}(\mathbf{v})$ as a proper suffix of $lpp_{\mathbf{v}}$ since they are palindromes, a contradiction). Therefore, once we have computed the set $\mathcal{R} = \{\mathbf{v} \mid (\mathbf{v}, |lpp_{\mathbf{v}}|)$ is a rightmost occurrence of $lpp_{\mathbf{v}}$ in $\mathcal{T}\}$ and have *marked* the trie nodes in $\mathcal{R}$, the set $\mathsf{DPal}(w)$ for $w = \mathsf{suf}(\mathbf{u})$ can be computed by searching all the marked nodes on the suffix-path from $\mathbf{u}$ in $\mathcal{T}$ when a query node $\mathbf{u}$ is given. The search can be done in $O(|\mathsf{DPal}(w)|)$ time by recursively querying the nearest marked ancestor (NMA) from $\mathbf{u}$ until it reaches the root. Note that $O(1)$-time static NMA queries can be preprocessed by an $O(n)$-time standard traversal on $\mathcal{T}$.

In the following, we show how to precompute the set $\mathcal{R}$ in linear time. First, we compute $lpp_{\mathbf{v}}$ for each node $\mathbf{v}$. This can be done in linear time by traversing on $\mathsf{STree}(\mathcal{T})$ since $lpp_{\mathbf{v}_i}$ corresponds to the nearest palindromic ancestor node of the leaf ℓ_i in $\mathsf{STree}(\mathcal{T})$. Also, we store a stack in each palindromic node in $\mathsf{STree}(\mathcal{T})$. Then, we check whether $lpp_{\mathbf{v}}$ is the rightmost occurrence for each node $\mathbf{v}$ by performing a depth-first traversal on $\mathcal{T}$ as follows: During the traversal, when descending from node $\mathbf{v}$, we push the node onto the stack of the palindromic node representing $lpp_{\mathbf{v}}$. If the stack is empty just before the push, then $\mathbf{v} \in \mathcal{R}$ holds since $lpp_{\mathbf{v}}$ is the rightmost occurrence, and thus, we mark the node $\mathbf{v}$. When ascending from node $\mathbf{v}$, we pop the node from the stack of the palindromic node representing $lpp_{\mathbf{v}}$. ◀

Longest palindromes in suffix-paths

Let $L(\mathbf{v})$ denote the length of a longest palindrome in $\mathsf{suf}(\mathbf{v})$ in the input trie $\mathcal{T}$. For each non-root node $\mathbf{v}$, $L(\mathbf{v}) = \max\{|lpp_{\mathbf{v}}|, L(\mathsf{parent}(\mathbf{v})\}$ holds. Thus, after computing the lengths of all $lpp_{\mathbf{v}}$, we can compute all $L(\mathbf{v})$ in a top-down manner. Also, we can associate the length $L(\mathbf{v})$ with the corresponding occurrence while computing them. As we proved in Theorem 11, we can compute $lpp_{\mathbf{v}}$ for each node $\mathbf{v}$ in linear time. The next corollary holds:

▶ **Corollary 12.** *We can compute a longest palindrome in each suffix-path $\mathsf{suf}(\mathbf{v}) \in \mathsf{Suffix}(\mathcal{T})$ in $O(n)$ total time.*

5 Conclusions and open questions

This paper proposed the *first* $O(n)$-time algorithm which computes all distinct palindromes and all maximal palindromes in a given trie $\mathcal{T}$ of size n, where the edge labels are drawn from an integer alphabet of size polynomial in n.

In the case of a general ordered alphabet of size σ, one can first sort the edge labels of $\mathcal{T}$ in $O(n \log \sigma)$ time with $O(n)$ space and replace the edge labels with their lexicographical ranks in range $[1..n]$, so our algorithms work in $O(n \log \sigma)$ time and $O(n)$ space.

It is open whether there exists an $O(n)$-time algorithm for computing distinct/maximal palindromes in a trie for general ordered alphabets. To achieve this goal, it is prohibited to construct edge-sorted suffix trees since there is an $\Omega(n \log \sigma)$-time lower bound [6].

It is known that there can be $\Theta(n^{3/2})$ distinct palindromes in an *unrooted* edge-labeled tree of size n [4, 8], and all of them can be computed in $O(n^{3/2} \log n)$ time [9]. It is open whether there is an $O(n^{3/2})$-time solution for this problem.

References

1. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *4th Latin American Theoretical Informatics Symposium, LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
2. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.
3. Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theor. Comput. Sci.*, 191(1-2):131–144, 1998. doi:10.1016/S0304-3975(96)00319-2.
4. Srecko Brlek, Nadia Lafrenière, and Xavier Provençal. Palindromic complexity of trees. In *19th International Conference on Developments in Language Theory, DLT 2015*, volume 9168 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2015. doi:10.1007/978-3-319-21500-6_12.
5. Xavier Droubay, Jacques Justin, and Giuseppe Pirillo. Episturmian words and some constructions of de Luca and Rauzy. *Theor. Comput. Sci.*, 255(1-2):539–553, 2001. doi:10.1016/S0304-3975(99)00320-5.
6. Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
7. Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing maximal palindromes and distinct palindromes in a trie. In *Proceedings of the Prague Stringology Conference, PSC 2019*, pages 3–15. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019. URL: http://www.stringology.org/event/2019/p02.html.
8. Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Tight bound for the number of distinct palindromes in a tree. In *22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015*, volume 9309 of *Lecture Notes in Computer Science*, pages 270–276. Springer, 2015. doi:10.1007/978-3-319-23826-5_26.
9. Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Tight bound for the number of distinct palindromes in a tree. *CoRR*, abs/2008.13209, 2020. doi:10.48550/arXiv.2008.13209.
10. Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010. doi:10.1016/j.ipl.2010.07.018.
11. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
12. Shunsuke Inenaga. Towards a complete perspective on labeled tree indexing: New size bounds, efficient constructions, and beyond. *J. Inf. Process.*, 29:1–13, 2021. doi:10.2197/ipsjjip.29.1.
13. S. Rao Kosaraju. Efficient tree pattern matching (preliminary version). In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 178–183. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63475.
14. Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.
15. Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018. doi:10.1016/j.ejc.2017.07.021.
16. Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 86-A(5):1061–1066, 2003. URL: http://search.ieice.org/bin/summary.php?id=e86-a_5_1061.
17. Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. Chapman and Hall/CRC, 2009.
18. Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.