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
Department of Informatics, Kyushu University, Fukuoka, Japan
PRESTO, Japan Science and Technology Agency, Tokyo, Japan

Abstract

A trie $T$ 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 $T$ with $n$ edges, we show how to compute all distinct palindromes and all maximal palindromes on $T$ 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 $T$. Using our new algorithms, the eertree with suffix links for a given trie $T$ 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 $T$, 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 $T$ 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...
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 $T$ with $n$ edges and $l$ leaves, the number of maximal palindromes in $T$ is exactly $2n - l$ and the number of distinct palindromes in $T$ is at most $n + 1$. In addition, they showed how to find all maximal palindromes in a given trie $T$ in $O(n \log h)$ time with $O(h)$ space, where $h$ is the height of $T$. 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 $T$ 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 $\sigma$ in-coming suffix links to a single leaf of the suffix tree for a trie, where $\sigma$ is the alphabet size. Since we need to check whether the palindrome in question can be extended with one of at most $\sigma$
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 $T$ can be used for reporting all maximal/distinct palindromes in the strings stored in $T$, 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 $T$ can be as large as $O(n^2)$.

## Preliminaries

### Strings and palindromes

Let $\Sigma$ be the alphabet. An element of $\Sigma^*$ is called a string. The length of a string $S$ is denoted by $|S|$. The empty string $\epsilon$ is a string of length 0, namely, $|\epsilon| = 0$. For any non-negative integer $k$, let $\Sigma^k$ 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 $i$th 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 $S[i..] = \epsilon$ 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 $\epsilon$ is also considered to be a palindrome. Let $\mathbb{P}$ denote the set of all palindromes over $\Sigma$.

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 $S$.

### Tries and palindromes

A trie $T = (V, E)$ is a rooted tree where each edge in $E$ is labeled by a single character from $\Sigma$ and the out-going edges of each node are labeled by mutually distinct characters. For any non-root node $u$ in $T$, let $\text{parent}(u)$ denote the parent of $u$. For any node $v$ in $T$, let $\text{childchar}(v)$ denote the set of out-going edge labels of $v$. Let $\text{height}(T)$ denote the height of $T$. For convenience, we read the path labels in the input trie $T$ in the leaf-to-root direction. For any path $(u, v)$, we denote by $\text{str}(u, v)$ the path label from $u$ to $v$. The set $\text{Substr}(T)$ of (reversed) substrings in $T$ is $\text{Substr}(T) = \{ \text{str}(u, v) | u, v \in V, u \text{ is a descendant of } v \}$. Note that $v$ itself is a descendant of $v$, i.e., the empty string $\epsilon = \text{str}(v, v)$ is an element of $\text{Substr}(T)$. For a string $w$ and a node $u$ in $T$, the pair $(u, |w|)$ is called an occurrence of $w$ in $T$ if $w$ is a prefix of the substring starting at $u$ in $T$. Such representations of occurrences allow us to retrieve the substring $w$ in $O(|w|)$ time by taking the unique path from $u$ towards the root. Also, we enhance the input trie $T$ with a level ancestor data structure [2] so that any character in a given path $(u, v)$ 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 $u$ and ends at node $v$ is called a maximal palindrome in $T$ if (a) $u$ is not a leaf and $\text{str}(u', \text{parent}(v))$ is not a palindrome.
with any child $u'$ of $u$, (b) $u$ is a leaf, or (c) $v$ is the root $r$. Let $\text{MPal}(T)$ be the set of all maximal palindromes in $T$. For each $1 \leq k \leq \text{height}(T)$, let $\text{MPal}_k(T) = \{(v, k) \in \text{MPal}(T)\}$, and let $\text{MPal}_0(T) = \{\varepsilon\}$. Namely $\bigcup_{0 \leq k \leq \text{height}(T)} \text{MPal}_k(T) = \text{MPal}(T)$.

For any trie $T$, let $\text{DPal}(T)$ be the set of all substring palindromes in $T$, namely $\text{DPal}(T) = \text{Substr}(T) \cap \mathbb{P}$. We call the elements of $\text{DPal}(T)$ as distinct palindromes in $T$. For each $0 \leq k \leq \text{height}(T)$, let $\text{DPal}_k(T) = \text{DPal}(T) \cap \Sigma^k$. Namely $\bigcup_{0 \leq k \leq \text{height}(T)} \text{DPal}_k(T) = \text{DPal}(T)$.

See Figure 1 for an example of trie $T$ and its substring palindromes.

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

![Figure 1](image-url) An example for an input trie $T$ with $n = 15$ edges, in which the path labels are read from the leaves to the root $r$. The height of $T$ is $\text{height}(T) = 6$. Let us focus on the node $v_{14}$. Then $\text{parent}(v_{14}) = v_{10}$, $\text{childchar}(v_{14}) = \{a, c\}$. Also, $\text{str}(v_{14}, v_1) = \text{cba}$. The occurrence $(v_{10}, 1)$ of palindrome $b$ is not a maximal palindrome since it can be extended to a longer palindrome $\text{str}(v_4, v_1) = \text{aba}$. This occurrence $(v_1, 3)$ of $\text{aba}$ is a maximal palindrome. We have $\text{DPal}(T) = \{\varepsilon, a, b, c, \text{aa}, \text{bb}, \text{cc}, \text{aba}, \text{bbb}, \text{aca}, \text{abba}, \text{abba}\}$ 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 $r$ of the input trie $T$ has only a single child and its incoming edge from $r$ is labeled by a special character $\$$ that does not occur elsewhere in $T$.

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

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

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

For any reversed substring $w$ in the input trie $T$, the locus for $w$ in the respective suffix tree $\text{STree}(T)$ is the ending point of the path which spells out $w$ from the root of $\text{STree}(T)$. 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 $\text{STree}(T)$, let $\text{str}(v)$ denote the path string from the root of $\text{STree}(T)$ to $v$. Then, we say that the node $v$ represents the substring $\text{str}(v)$.
By the third property of \( \text{STree}(\mathcal{T}) \), there are exactly \( n \) leaves in \( \text{STree}(\mathcal{T}) \). Since each of the internal explicit nodes has at least two children, there are at most \( n - 1 \) internal nodes in \( \text{STree}(\mathcal{T}) \). Thus there are at most \( 2n - 1 \) nodes and \( 2n - 2 \) edges in \( \text{STree}(\mathcal{T}) \). Each edge label \( x \) of \( \text{STree}(\mathcal{T}) \) is represented by a pair \((u, v)\) of nodes in \( \mathcal{T} \) such that \( x = \text{str}(u, v) \). This allows us to represent \( \text{STree}(\mathcal{T}) \) in \( O(n) \) space. See Figure 2 for an example \( \text{STree}(\mathcal{T}) \).

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 \( \text{STree}(\mathcal{T}) \) 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 \([1..n] \), \( \text{STree}(\mathcal{T}) \) 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 \( \text{STree}(\mathcal{T}) \) are arranged in the lexicographical order from left to right. For each \( 1 \leq i \leq n \), let \( \ell_i \) denote the \( i \)-th leaf in \( \text{STree}(\mathcal{T}) \) which represents the lexicographically \( i \)-th suffix in \( \text{Suffix}(\mathcal{T}) \). For each \( 1 \leq i \leq n \), let \( v_i \) denote the node in the input trie \( \mathcal{T} \) that corresponds to the leaf \( \ell_i \), namely \( \text{suffix}(v_i) = \text{str}(\ell_i) \) (see Figure 2 for examples).

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

In our algorithms to follow, we will use only the suffix links of the leaves of \( \text{STree}(\mathcal{T}) \). 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 \( \ell_i \) in \( \text{STree}(\mathcal{T}) \), let \( \text{inSLchar}(\ell_i) \) denote the set of in-coming suffix link labels. Notice that \( \text{inSLchar}(\ell_i) = \text{childchar}(v_i) \) holds since \( \text{SL}(\ell_i) = \ell_j \) if and only if \( \text{parent}(v_i) = v_j \). See Figure 2 for examples. For any node \( v \) in \( \text{STree}(\mathcal{T}) \), let \( \text{subtree}(v) \) denote the subtree rooted at \( v \), and let \( \text{leaves}(v) \) denote the set of leaves in \( \text{subtree}(v) \). Given \( \text{STree}(\mathcal{T}) \), it is easy to compute the suffix links of all leaves of \( \text{STree}(\mathcal{T}) \) in \( O(n) \) time. Thus we can also compute \( \text{inSLchar}(\ell) \) for every leaf \( \ell \) in \( \text{STree}(\mathcal{T}) \) in a total of \( O(n) \) time.

We extend the notion of \( \text{childchar}(\cdot) \) to the suffix tree so that for each node \( v \) in \( \text{STree}(\mathcal{T}) \), \( \text{childchar}(v) \) 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 \( \text{eertree}(\mathcal{T}) \), is a pair of two rooted trees (the even-tree and odd-tree) where the root of the even-tree represents \( \varepsilon \) and the root of the odd-tree is an auxiliary node \( \bot \). Then, for each \( p \in \text{DPal}(\mathcal{T}) \), there is a path that spells out \( p[[|p|/2] + 1..|p|] \) 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 \( \text{eertree}(\mathcal{T}) \) 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 \( \varepsilon \) and \( \bot \) both point to \( \bot \). See Figure 3 for an example of the eertree for a trie.
3 Algorithm for computing \( \text{DPal}(T) \) and \( \text{MPal}(T) \)

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

3.1 Overview of our algorithm

We use \( \text{STree}(T) \) as our main tool for computing \( \text{DPal}(T) \) and \( \text{MPal}(T) \). An (implicit or explicit) node \( v \) on \( \text{STree}(T) \) such that \( \text{str}(v) \in \mathcal{P} \) 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 \( |\text{DPal}(T)| \in O(n) \) palindromic nodes in \( \text{STree}(T) \), and thus, computing \( \text{DPal}(T) \) can be reduced to computing all palindromic nodes in \( \text{STree}(T) \).

We compute \( \text{DPal}(T) \) together with \( \text{MPal}(T) \) in increasing order of the lengths of the palindromes. Namely, we visit all palindromic nodes in \( \text{STree}(T) \), 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 \( T \). This can be done as follows:

\[
\begin{align*}
\text{Base steps:} & \text{ Start from the root } r \text{ of } \text{STree}(T). \text{ Report } \varepsilon \text{ as a member of } \text{DPal}_0(T), \\
& \text{ and report all single characters occurring in } T, \text{ as members of } \text{DPal}_1(T).
\end{align*}
\]

\[
\begin{align*}
\text{Inductive step:} & \text{ For each increasing } k = 0, 1, \ldots, \text{height}(T) - 1:\n& \quad \text{ For each palindrome } p \in \text{DPal}_k(T), \text{ let } v \text{ be the locus for } p \text{ in } \text{STree}(T). \\
& \quad \quad \text{ For each character } c \in \text{childchar}(v), \text{ perform the following:} \\
& \quad \quad \quad \cdot \text{ If } cpc \in \text{Substr}(T), \text{ then report this new palindrome } cpc \text{ of length } k + 2 \text{ as a new member of } \text{DPal}_{k+2}(T). \\
& \quad \quad \quad \cdot \text{ For each occurrence of } p_c \text{ that is not preceded by } c \text{ in the reversed trie } T, \\
& \quad \quad \quad \quad \quad \text{ report this occurrence of } p \text{ as a new member of } \text{MPal}_k(T).
\end{align*}
\]
Figure 3 The palindromic tree $\text{eertree}(T)$ for the input trie $T$ of Figure 1, which represents $\text{DPal}(T) = \{\varepsilon, a, b, c, aa, bb, cc, aba, bbb, aca, abba, abba\}$. 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 $\text{STree}(T)$ and $c \in \text{childchar}(v)$, our task is to efficiently check whether or not $cpc \in \text{Substr}(T)$, and if so, then find the locus for $cpc$ in $\text{STree}(T)$. In the sequel, for simplicity, we identify the strings $p$ and $pc$ with their loci ( implicit or explicit nodes) in $\text{STree}(T)$.

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

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

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

Figure 4 Illustration for Observations 3 and 4.

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

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

(2) Check whether there exists a leaf $\ell$ under the locus for $pc$ such that $inSLchar(\ell)$ 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 $STree(\mathcal{T})$ 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 $destSL_c[i] = j$ if the leaf $\ell_j$ 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 $\infty$. Also, we sometimes regard $destSL_c$ as a list of the corresponding leaves $\ell_i$. We then define the jump-array $jump_c$ which leads us to the ending point of the run of adjacent leaves in $STree(\mathcal{T})$ for each given leaf $\ell_i$ in $destSL_c$. Formally, $jump_c[i] = i + 1$ if $destSL_c[i+1] - destSL_c[i] > 1$, and $jump_c[i] = \min\{j \mid j > i \text{ and } destSL_c[j+1] - destSL_c[j] > 1\}$ 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 $c \in inSLchar(\ell_i)$. Simultaneously, we associate each pair $(c, i)$ with its rank in $destSL_c$, i.e., the integer $r$ with $destSL_c[r] = i$.

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

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

In the computing phase, we traverse on $\text{STree}(T)$ 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 $T$. 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 $\ell_{\text{left}}$ and $\ell_{\text{right}}$ be the leftmost and the rightmost leaves in $\text{leaves}(v)$, respectively. Namely, they correspond
to the left-th and right-th smallest suffixes in $T$, respectively. We initialize the current-leaf-index $curr \leftarrow \text{destSL}_c[\text{leftDest}(v)]$ and the previous-leaf-index $prev \leftarrow \text{left} - 1$. While $curr \leq \text{right}$, output trie nodes $v_i$ for all $i$ with $prev < i < curr$ if $curr - 1 \notin \text{destSL}_c$, and then set $prev \leftarrow curr$ and $curr \leftarrow \text{destSL}_c[jump_c[\text{curr}]]$. At last, after breaking out from the while-loop condition, output trie nodes $v_i$ for all $i$ with $prev < i \leq \text{right}$ if $curr - 1 \notin \text{destSL}_c$. See also Figure 6 for illustration.

![Figure 6](image)

The black stars indicate the leaves in 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 $\text{destSL}_c$ corresponding to the leaves $\ell_\alpha$, $\ell_\beta$, $\ell_\gamma$ and $\ell_\delta$ 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 $\ell_\gamma$, we output nothing since none of the leaves between $\ell_\beta$ and $\ell_\gamma$ 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 subtree$(v)$. Then, at least two jumps are performed inside subtree$(v)$. (The last jump, which is toward the outside of subtree$(v)$, is the only exception.)

Let $i$ be the current leaf-index. By the definition of jump-pointers, if $\text{destSL}_c[i + 1] - \text{destSL}_c[i] > 1$, then $\text{jump}_c[i] = i + 1$. Then, our algorithm reports at least one maximal palindrome since we skip at least one leaf between $\text{destSL}_c[i]$ and $\text{destSL}_c[jump_c[i]] = \text{destSL}_c[i + 1]$ both exclusive. Otherwise, namely if $\text{destSL}_c[i + 1] - \text{destSL}_c[i] = 1$, then $\text{jump}_c[i]$ points to the ending point of the run of adjacent leaves in $\text{destSL}_c$. In this case, we report no maximal palindromes. However, in the next step, we track $\text{jump}_c[j]$ where $j = \text{jump}_c[i]$. Then, we fall into the first case $\text{destSL}_c[j + 1] - \text{destSL}_c[j] > 1$ 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 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 $\ell_R$ such that $\ell_R \in \text{leaves}(v)$ and $R \in \text{destSL}_c$. We move to the source leaves $\ell$ and $\ell'$ of the suffix links which respectively point to $\ell_L$ and to $\ell_R$, both labeled by $c$ where $L = \text{destSL}_c[\text{leftDest}(v)]$ is the leaf-index specified by $\text{leftDest}(v)$. We then compute the LCA $u'$ of $\ell$ and $\ell'$. 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 $T$, there is at most one implicit node $v$ on each edge of $STree(T)$ such that $\text{str}(v)$ is a palindrome.

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

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 $STree(T)$ and its following character is $c$, then $q$ corresponds to an implicit node on an edge in $\text{subtree}(v)$ 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 $T$. 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 $\text{subtree}(v)$.

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, $v.\text{mark} = 1$ means that we already visited an implicit ancestor representing some shorter prefix palindrome. Initially, we set $u.\text{mark} \leftarrow 0$ for every node $u$. Let $v$ be the shallowest explicit node below $p$ and $c$ be the character following $p$. If $v.\text{mark} = 0$, every $v' \in \text{subtree}(v)$ is not marked yet since we consider palindromes in non-decreasing order of the lengths. We traverse $\text{subtree}(v)$ and mark all nodes in the subtree. Also, while traversing the subtree, we compute the links $\text{left}(v')$ from each $v' \in \text{subtree}(v')$ defined as follows: $\text{left}(v')$ is the leftmost leaf in $\text{leaves}(v')$ such that its leaf-index is in $\text{destSL}_c$ if such a leaf exists, and $\text{nil}$ otherwise. In other words, $\text{left}(v')$ is a variant of $\text{leftDest}(v')$.
where the character $c$ to be considered is fixed. Traversing $\text{subtree}(v)$ and computing $\text{left}(v')$ for all nodes $v'$ in $\text{subtree}(v)$ can be done in time linear in the size of $\text{subtree}(v)$ plus the total number of incoming suffix links of $\text{leaves}(v)$ as in the computing of $\text{leftDest}$ described in the previous subsection. Once we create the links $\text{left}$, 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 $v\cdot \text{mark} = 1$, there is an implicit ancestor we already visited. By the above procedures, node $v$ already has link $\text{left}(v)$, and hence, we can output all maximal palindromes $p$ in output linear time. To summarize, the following theorem holds:

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

\textbf{Proof.} In the preprocessing phase, $\text{STree}(T)$ can be constructed in $O(n)$ time and space (Theorem 2). An LCA data structure on $\text{STree}(T)$ 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 $\text{STree}(T)$.

For the case (A) in the computing phase, most of the procedures can be done in constant time per an edge of $\text{STree}(T)$. 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 $O(n + |\text{MPal}(T)|) = O(n)$. 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($T$)

\textbf{Theorem 9.} For a given trie $T$ with $n$ edges over an integer alphabet $\Sigma = [1..n]$, the edge-sorted eertree($T$) with suffix links can be computed in $O(n)$ time and space.
Proof. Since our algorithm computes $\text{DPal}(T)$ in increasing order of the lengths, we can easily compute the tree-topology of $\text{eertree}(T)$ in conjunction with $\text{DPal}(T)$ 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 $\text{eertree}(T)$ 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 $\text{eertree}(T)$ by a standard traversal on $\text{STree}(T)$ in which all the palindromic nodes are explicitly inserted as in Figure 2.

4 Answering queries to find palindromes in sub-paths

A trie $T$ can be regarded as a compact representation of a set $S_T$ of strings that are the path strings from the leaves to the root of $T$. This section presents how to report all distinct/maximal palindromes in each string in $S_T$ upon query, in output-linear time with $O(n)$ space, where $n$ is the size of $T$. We remark that the total length of the strings in $S_T$ can be as large as $O(n^2)$, and thus, our $O(n)$-size representation of the palindromes for $S_T$ 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:

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

\textbf{Proof.} In the preprocessing, we transform the occurrence representation of maximal palindromes into the pair $(s, \alpha)$ of the starting node $s$ and its center $\alpha$, which is either a node or an edge on the suffix-path from $s$. Such transformation can be done in linear time by traversing $T$: We maintain an array $N$ of size $\text{height}(T)$ 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 $v$ and find a maximal palindrome $(v, k)$, we can compute the $k/2$-th shallower locus $\alpha$ by using array $N$ in constant time. Simultaneously, we store the pointer from each $\alpha$ to the starting node $s_\alpha$ of the maximal palindrome centered at $\alpha$.

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

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 $\text{DPal}(S)$ for a string $S$ in [10] reports their leftmost occurrences in $S$. Also, the algorithm that computes $\text{DPal}(T)$ for a trie $T$ in [7] reports the first occurrence of each palindrome during a depth-first traversal in $T$. Below, we show how to efficiently report $\text{DPal}(w)$ for any suffix-path string $w$ in the trie $T$. 

ISAAC 2022
Computing Palindromes on a Trie in Linear Time

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

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

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

Longest palindromes in suffix-paths

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

Corollary 12. We can compute a longest palindrome in each suffix-path $\text{suffix}(v) \in \text{Suffix}(T)$ in $O(n)$ total time.

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 $T$ 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 $\sigma$, one can first sort the edge labels of $T$ 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