Bachelor's thesis

ALGORITHMS RELATED TO GENERALIZED PALINDROMES IN SAGEMATH

Ivan Romanenko

Faculty of Information Technology
Department of Computer Science
Supervisor: doc. Ing. Štěpán Starosta, Ph.D.
May 10, 2023

Citation of this thesis: Romanenko Ivan. *Algorithms related to generalized palindromes in SageMath.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **v**

Declaration **vi**

Abstract **vii**

Summary **viii**

Notations **ix**

1 Introduction **1**

2 Preliminaries **3**
- 2.1 Palindromic defect . . . 3
- 2.2 Manacher's algorithm and its generalization . . . 5
- 2.3 Dynamic level ancestor problem . . . 9

3 Algorithms **11**
- 3.1 Computing generalized palindromic defects . . . 12
  - 3.1.1 Θ-defect . . . 12
  - 3.1.2 G-defect . . . 19
- 3.2 Computing classical palindromic defect . . . 25

4 SageMath **27**
- 4.1 Survey on development processes in SageMath . . . 27
  - 4.1.1 Tools . . . 27
  - 4.1.2 Development workflow . . . 27
  - 4.1.3 Code structure . . . 28
- 4.2 State of algorithms integration into SageMath . . . 28

5 Conclusion **29**

List of code listings

2.1 Slow algorithm for finding the longest palindromic substring 6
2.2 $\mathcal{O}\left(n^2\right)$ algorithm for finding the longest palindromic substring 6
2.3 Modified $\mathcal{O}\left(n^2\right)$ algorithm for finding maximal palindromic substring radiuses . 7
2.4 Manacher's algorithm for finding maximal palindromic substring radiuses 7
2.5 Manacher's algorithm for finding maximal Θ-palindromic substring radiuses . . . 8

I want to thank all my friends and all the good people. Thanks to my friends in Prague for all the psychological support since I moved to Prague. Thanks to all math professors I studied from in Russia. Special thanks to my first math professor Vladimir Vladimirovich Nyu, Cand.Sc.(Russia) who has always been a huge inspiration for me. Thanks to my parents for financial support during my studies. Finally, thanks to doc. Ing. Štěpán Starosta, Ph.D. for great work on supervising this thesis and not letting me go into any wrong directions.

Declaration

In Prague on May 10, 2023 . . . .

Abstract

A palindrome is a word which reads the same from the left and from the right. Palindromic defect of word w is the difference between $|w|+1$ and the amount of pairwise distinct palindromic substrings of w. Concepts of palindrome and palindromic defect can be generalized to generalized palindrome, Θ-defect and G-defect, where Θ is an antimorphism, G is a finite group consisting of morphisms and antimorphisms (see [1]). The free open-source mathematics software system SageMath [2] contains a developed library containing numerous algorithms dealing with words. The first goal of the thesis is to present and prove several newly discovered algorithms for computing palindromic defect and its generalizations. As a special case of one of these algorithms, linear time algorithm for computing classical palindromic defect will be shown. The second goal of the thesis is to start adding some of these algorithms into SageMath.

Keywords Word, Word algorithms, Palindromic defect, Generalized palindrome, Generalized palindromic defect, SageMath, Python

Abstrakt

Palindrom je slovo, které se čte stejně zleva doprava i zprava doleva. Palindromický defekt slova w je rozdíl mezi $|w|+1$ a počtem dvojic odlišných palindromických podřetězců slova w. Koncepty palindromu a palindromického defektu lze zobecnit na generalizovaný palindrom, Θ-defekt a G-defekt, kde Θ je antimorfismus, G je konečná grupa skládající se z morfismů a antimorfismů (viz [1]). Open-source matematický softwarový systém SageMath [2] obsahuje vyvinutou knihovnu podporující různé algoritmy na slova. Prvním cílem této práce je prezentovat a dokázat několik nově objevených algoritmů pro výpočet palindromického defektu a jeho zobecnění. Jako speciální případ jednoho z těchto algoritmů bude ukázán lineární časový algoritmus pro výpočet klasického palindromického defektu. Druhým cílem této práce je začít přidávat některé z těchto algoritmů do SageMath.

Klíčová slova Řetězce, Algoritmy nad řetězci, Palindromický defekt, Generalizované palindromy, Generalizovaný palindromický defekt, SageMath, Python

Summary

Algorithms

This thesis contains description and proof of several new algorithms for computing generalized palindromic defects.

All algorithms are meant to be single-threaded, if not stated otherwise.

First three algorithms compute Θ-defect in $\mathcal{O}(|w| \log |w|)$, $\mathcal{O}(|w|)$ and $\mathcal{O}(|w|)$ time respectively.

The first algorithm uses Manacher's algorithm (see [3]) combined with jump-pointers concept (see [4]).

The second algorithm uses Manacher's algorithm combined with advanced solution for dynamic level ancestor problem by Alstrup; Holm from [5].

The third algorithm uses only Manacher's algorithm and iteration over tree graphs.

The next algorithm computes G-defect in $\mathcal{O}(|G| \cdot |w|)$ time. This algorithm is asymptotically fastest known at time of writing this thesis.

The last algorithm is a multi-threaded algorithm which computes G-defect in $\mathcal{O}(|w|)$ time using $\mathcal{O}(|G|)$ threads.

Here w stands for word: $w \in A^*$, Θ stands for antimorphism which is a letter permutation, G stands for finite group of morphisms and antimorphisms.

SageMath implementation

Last mentioned $\mathcal{O}(|w|)$ algorithm for Θ-defect and $\mathcal{O}(|G| \cdot |w|)$ algorithm for G-defect described in the thesis will be added into existing code of SageMath word library. These algorithms will be implemented in Python because all existing related code is using Python and switching existing code to Cython should be done as separate work in separate branch/ticket.

Multi-threaded algorithm for computing G-defect won't be implemented in SageMath because it requires multi-thread hash map (a.k.a. concurrent hash map) data structure. This data structure does not exist in any standard Python libraries and adding multi-threaded algorithms into SageMath is outside the scope of this thesis.

GitHub issue on SageMath [2] has number 35495.

Notations

The following notations are assumed, if not stated otherwise:

$\mathcal{A}$	Alphabet
w	Word, $w \in \mathcal{A}^*$
f	Morphism and a letter permutation, $f \in M(\mathcal{A}^*)$, $\forall a \in \mathcal{A} : f(a) \in \mathcal{A}$
Θ	Antimorphism and a letter permutation, $\Theta \in AM(\mathcal{A}^*)$, $\forall a \in \mathcal{A} : \Theta(a) \in \mathcal{A}$
G	Finite group of morphisms and antimorphisms

Chapter 1
Introduction

This thesis serves mainly fundamental research purpose. Even though there are some known applications of generalized palindromes in several areas mentioned in [1], author of this thesis is not aware of any direct applications of algorithms, which will be presented.

It was shown in [6] by Droubay that amount of pairwise distinct palindrome substrings of word w does not exceed $|w|+1$. This limit is reached, for example, for words formed by pairwise distinct letters (empty word ε is included). Palindromic defect of word w is then defined as difference between maximal possible amount of pairwise distinct palindrome substrings for words of length $|w|$ (which is equal to $|w|+1$ for the classical palindromes) and actual amount of pairwise distinct palindrome substrings of w.

After some time, generalization of palindrome to Θ-palindrome was introduced in [7]. Θ-defect is then a generalization of classical palindromic defect, defined the same way as classical palindromic defect, but the $|w|+1$ term is slightly modified.

Then in [8], definition of G-defect was given in similar manner to Θ-defect.

For readers, who are only interested in algorithms for computing classical palindromic defect (not generalized ones), it is recommended to study Manacher's algorithm (several sources can be used for that, e.g. [3]), optionally see section 2.3 for theory on $\mathcal{O}(|w|)$ algorithm using advanced dynamic level ancestor solution and skip whole thesis except section 3.2.

From now on, let's concentrate on algorithms for generalized palindromic defects.

In this work several algorithms to compute Θ-defect will be presented in section 3.1, which are theoretically backed by chapter 2. Recapitulation of some definitions, lemmas and theorems from [1] is done in section 2.1. Main purpose of section 2.2 is to generalize Manacher's algorithm ([3]) to work with Θ-palindromes instead of classical palindromes. Several already existing solutions of dynamic level ancestor problem when leafs can be added to tree dynamically ([4], [5]) will be mentioned in section 2.3.

Three algorithms for computing Θ-defect will be presented: with $\mathcal{O}(|w|\log|w|)$, $\mathcal{O}(|w|)$ and $\mathcal{O}(|w|)$ time complexities. First two of these algorithms use generalized Manacher's algorithm from section 2.2 and data structures from section 2.3, while the last algorithm uses only generalized Manacher's algorithm. Readers interested only in this algorithm can safely skip section 2.3.

The only non-trivial part of these algorithms is to find amount of pairwise different palindrome substrings of w. To achieve it, each of these algorithms builds such tree graph T, that each node in T represents a palindrome different from palindromes represented by other nodes of T.

The first algorithm uses jump-pointers to transfer up in T, while building it during Manacher's algorithm. Each transfer up the T costs $\mathcal{O}(\log|w|)$ time.

The second algorithm works conceptually the same as the above algorithm but it uses data structures which are superior to jump-pointers (see [5]). Each transfer up the T costs constant time.

The third algorithm does not build T from the start, but instead it builds a forest graph H (forest is a graph formed by several trees) using Manacher's algorithm. Then it simplifies each tree in H, resulting in H'. And finally, it consecutively traverses each tree of H' while building T.

As for algorithms to compute G-defect, single-threaded algorithm with $\mathcal{O}\left(|G| \cdot (S(w) + |w|)\right)$ time complexity will be shown, where S is an algorithm which builds the T graph, $S(w)$ is time complexity of S applied on w. If we choose S as the above $\mathcal{O}\left(|w|\right)$ algorithm, then we get algorithm with $\mathcal{O}\left(|G| \cdot |w|\right)$ time complexity for computing G-defect.

Lastly, multi-threaded algorithm to compute G-defect in $\mathcal{O}\left(S(w) + |w|\right)$ time using $\mathcal{O}\left(|G|\right)$ threads will be presented. It is conceptually same as the previous algorithm. It simply integrates classical multi-thread techniques and uses concurrent hash map data structure with minor implementation details. Concurrent hash map can be implemented in different ways and our algorithm has some extra requirements on concurrent hash map implementation to achieve promised time complexity.

The second goal of this thesis is to implement some of the above algorithms in SageMath [2].

SageMath is a free open-source mathematics software system, based mainly on Python.

In section 4.1 we will do a quick overlook of current development processes of SageMath, which were significantly simplified at the beginning of 2023 by complete transfer into GitHub repository. Current state (at the time of presenting this thesis) for algorithms integration into SageMath is given in section 4.2. For readers not interested in details of SageMath development processes and implementation of the algorithms in SageMath system, it is recommended to completely skip whole chapter 4.

Chapter 2
Preliminaries

This chapter contains theoretical notes on several topics. These notes are then used in following chapters.

2.1 Palindromic defect

An *alphabet* $\mathcal{A}$ is a finite set. Elements of $\mathcal{A}$ are usually called *letters*. A *finite word* w over $\mathcal{A}$ is a finite string $w = w_1 w_2 \cdots w_n$ of letters $w_i \in \mathcal{A}$. Its length, denoted by $|w|$, is n. The set of all finite words over $\mathcal{A}$ equipped with the operation of concatenation is the free monoid $\mathcal{A}^*$. Its neutral element is the empty word ε. A word $v \in \mathcal{A}^*$ is a *factor* of a word $w \in \mathcal{A}^*$ if there exist words $s, t \in \mathcal{A}^*$ such that $w = svt$. If $s = \varepsilon$, then v is a *prefix* of w, if $t = \varepsilon$, then v is a *suffix* of w. Alternatively, a factor of w is called a *substring* of w.

A mapping φ on $\mathcal{A}^*$ is called

- a *morphism* if $\varphi(vw) = \varphi(v)\varphi(w)$ for any $v, w \in \mathcal{A}^*$
- an *antimorphism* if $\varphi(vw) = \varphi(w)\varphi(v)$ for any $v, w \in \mathcal{A}^*$.

We denote the set of all morphisms and antimorphisms on $\mathcal{A}^*$ by $AM(\mathcal{A}^*)$. Together with composition, it forms a monoid with the identity mapping Id as the unit element. The set of all morphisms, denoted by $M(\mathcal{A}^*)$, is a submonoid of $AM(\mathcal{A}^*)$. The reversal mapping R is defined by

$$R(w_1 w_2 \cdots w_n) = w_n w_{n-1} \cdots w_2 w_1 \quad \text{for all } w = w_1 \cdots w_n \in \mathcal{A}^*$$

It is obvious that any antimorphism is a composition of R and a morphism. Thus

$$AM(\mathcal{A}^*) = M(\mathcal{A}^*) \cup R(M(\mathcal{A}^*))$$

A fixed point of a given antimorphism Θ is called Θ-palindrome, i.e., a word w is a Θ-palindrome if $w = \Theta(w)$. If Θ is the reversal mapping R, we say palindrome or classical palindrome instead of R-palindrome.

In the rest of the thesis, it will be assumed for all antimorphisms that they are letter permutations as well, because this limitation simplifies all results of this thesis and similar results for antimorphisms which are not letter permutations were not researched in this thesis.

▶ **Definition 2.1** (Θ-defect). *Let Θ be an antimorphism and letter permutation, $w \in A^*$. Then Θ-defect of w written as $\mathrm{D}_\Theta(w)$ is defined as:*

$$\mathrm{D}_\Theta(w) = |w| + 1 - \#\,\mathrm{Pal}_\Theta(w) - \gamma_\Theta(w)$$

where $\mathrm{Pal}_\Theta(w)$ is the set of Θ-palindromic factors occurring in w, and

$$\gamma_\Theta(w) := \#\{\{a, \Theta(a)\} \mid a \in \mathcal{A}, a \text{ occurs in } w \text{ and } a \neq \Theta(a)\}$$

▶ **Theorem 2.2.**

$$\mathrm{D}_\Theta(w) \geq 0$$

Proof. See [1]. ◀

In the rest of the thesis, the symbol G stands exclusively for a subset of $AM(\mathcal{A}^*)$ satisfying the two following requirements:

1. G is a finite group
2. G contains at least one antimorphism

The first requirement on G implies the following for an element ν of G. The element ν is non-erasing, i.e., $\nu(a) \neq \varepsilon$ for all $a \in \mathcal{A}$ (otherwise ν has no inverse in G). Moreover, $\nu(a)$ is a letter for all $a \in \mathcal{A}$ (otherwise $\nu^n \neq \mathrm{Id}$ for all $n \geq 1$). We can conclude that ν restricted to $\mathcal{A}$ is a permutation of letters.

The second requirement on G comes from the fact that results of this thesis are based on generalized palindromes and one gets only trivial or no results when dealing with groups consisting of morphisms only.

▶ **Lemma 2.3.** *The following set of properties apply to G:*

1. *every element of G is either a morphism or an antimorphism determined by a permutation of letters of $\mathcal{A}$*
2. *G may contain elements of order greater than 2*
3. *G need not be abelian;*
4. *the set of antimorphisms of G generates the group G*
5. *the number of morphism in G equals the number of antimorphisms in G*

Proof. See [1]. ◀

We say that finite words $w, v \in \mathcal{A}^*$ are G-equivalent if there exists $\mu \in G$ such that $w = \mu(v)$. The class of equivalence containing a word w is denoted

$$[w] := \{\mu(w) \mid \mu \in G\}$$

As already mentioned, since the group G is finite, any $\mu \in G$ preserves length of words and thus equivalent words have the same length.

A word $w \in \mathcal{A}^*$ is said to be G-palindrome if there exists an antimorphism $\Theta \in G$ such that $w = \Theta(w)$.

▶ **Definition 2.4** (G-defect)**.** *Let $w \in A^*$. Then G-defect of w written as $\mathrm{D}_G(w)$ is defined as:*

$$\mathrm{D}_G(w) = |w| + 1 - \#\,\mathrm{Pal}_G(w) - \gamma_G(w)$$

where

$$\mathrm{Pal}_G(w) := \{[v] \mid v \textit{ is a factor of } w \textit{ and a } G\textit{-palindrome}\}$$

and

$$\gamma_G(w) := \#\{[a] \mid a \in \mathcal{A}, a \textit{ occurs in } w, \textit{and } a \neq \Theta(a) \textit{ for every antimorphism } \Theta \in G\}$$

▶ **Theorem 2.5.**

$$\mathrm{D}_G(w) \geq 0$$

Proof. See [1]. ◀

On top of above theory we will need two additional lemmas in this thesis.

▶ **Lemma 2.6.** *Let $u, v \in [w]$ and u is a G-palindrome. Then v is a G-palindrome as well.*

Proof. By definition of $[w]$, $\exists \varphi \in G : \varphi(u) = v$. By definition of G-palindrome, $\exists \psi \in G : \psi(u) = u$ and ψ - antimorphism. Let's notice that in both cases when φ is morphism and antimorphism, $\varphi^{-1} \circ \psi \circ \varphi$ is an antimorphism and $\varphi^{-1} \circ \psi \circ \varphi(v) = v$. This means that v is G-palindrome. ◀

It follows from above lemma that either all elements of $[w]$ are G-palindromes or none of $[w]$ elements are G-palindromes.

▶ **Lemma 2.7.** *Let $u, v \in [w]$ and w is a G-palindrome. Then $\exists f \in G : f(u) = v, f$ - morphism.*

Proof. By definition of $[w]$, $\exists \varphi \in G : \varphi(u) = v$. If φ is morphism, then this lemma is proven. Otherwise, φ is an antimorphism. In such case, let's notice that from the previous lemma follows u is G-palindrome so $\exists \psi \in G : \psi(u) = u$ and ψ is an antimorphism. Then $\psi \circ \varphi$ is a morphism and $\psi \circ \varphi(u) = v$. This is exactly what we wanted. ◀

2.2 Manacher's algorithm and its generalization

Manacher's algorithm was first described by Manacher in [3]. This algorithm finds longest palindromic substring of given word in linear time. In reality it does even more. For word w of length n, this is a $\mathcal{O}(n)$ time algorithm which finds maximum radiuses for all positions of w such that for position p and its radius r_p, substring of w starting at $p - r_p$ and ending at $p + r_p$ is a palindrome. Position p can either be position of one of w letters of it could be position in-between two consecutive letters of w (in such case we say that empty string has radius 0, two letter palindrome has radius 1 and so on).

For purposes of this thesis it is not enough to just mention this algorithm. As we are going to build different data structures during run of Manacher's algorithm, it is important to understand how Manacher's algorithm actually functions internally.

Let's start with more simple algorithms. All algorithms will be presented in Python. Naive $\mathcal{O}(n^3)$ time algorithm to find longest palindromic substring works by iterating over all $\mathcal{O}(n^2)$ substrings and checking for each substring if it is palindrome in $\mathcal{O}(n)$ time. Here is an example of algorithm:

Code listing 2.1 Slow algorithm for finding the longest palindromic substring

```python
def longest_palindromic_substring(s):
    n = len(s)
    max_len = 0
    start = 0
    for i in range(n):
        for j in range(i+1, n+1):
            sub = s[i:j]
            if sub == sub[::-1] and len(sub) > max_len:
                max_len = len(sub)
                start = i
    return s[start:start+max_len]
```

It is easy to see that we could actually find all maximum radiuses instead using same idea.

One can notice that we can reuse knowledge about substring starting on a position and ending on b position being a palindrome to find out if substring starting at $a-1$ and ending at $b+1$ is a palindrome. With this observation we get $\mathcal{O}\left(n^2\right)$ time algorithm to find longest palindromic substring. Here is an example algorithm:

Code listing 2.2 $\mathcal{O}\left(n^2\right)$ algorithm for finding the longest palindromic substring

```python
def longest_palindromic_substring(s):
    n = len(s)
    max_len = 1
    start = 0
    for i in range(n):
        # odd-length substrings
        l, r = i, i
        while l >= 0 and r < n and s[l] == s[r]:
            if r - l + 1 > max_len:
                max_len = r - l + 1
                start = l
            l -= 1
            r += 1
        # even-length substrings
        l, r = i, i+1
        while l >= 0 and r < n and s[l] == s[r]:
            if r - l + 1 > max_len:
                max_len = r - l + 1
                start = l
            l -= 1
            r += 1
    return s[start:start+max_len]
```

For sake of simplification, let's notice that we can add a special character s into our alphabet A and insert it between each pair of consecutive letters of w. Word w did not contain s character before we added it into alphabet, which means that all palindromic substrings are now of odd length (except empty string ε). This idea is important and by default we assume that it is used in all further algorithms.

Here is updated version of above algorithm computing radiuses instead:

Code listing 2.3 Modified $\mathcal{O}\left(n^2\right)$ algorithm for finding maximal palindromic substring radiuses

```python
def maximal_palindromic_substring_radiuses(s):
    s = '#'.join(s) # insert special character '#'
    p = [0] * len(s) # stores radiuses of palindromic substrings
    for i in range(len(s)):
        l, r = i, i
        while l >= 0 and r < len(s) and s[l] == s[r]:
            p[i] += 1
            l -= 1
            r += 1
    odd_pos_results = [(x + 1) // 2 for x in p[::2]]
    even_pos_results = [x // 2 for x in p[1::2]]
    p[::2] = odd_pos_results
    p[1::2] = even_pos_results
    return p
```

Manacher's algorithm goes even further to achieve $\mathcal{O}(n)$ time complexity. It is based on fact that we can reuse knowledge about found palindromes centered on different positions. More precisely, let's say we have two maximal palindromic substrings w and v. w has center position $center$, start position $left$ and end position $right$. v has center position $oldCenter$ such that $left \leq oldCenter < center$, start position $oldLeft$ and radius $oldRadius$. Then for maximal radius $newRadius$ in center position $newCenter = center + (center - oldCenter)$ the following applies:

- if $oldLeft > left$, then $newRadius = oldRadius$. (otherwise, v is not maximal)
- if $oldLeft < left$, then $newRadius = right - newCenter + 1, newRadius < oldRadius$. (otherwise, w is not maximal)
- if $oldLeft = left$, then $newRadius \geq oldRadius$.

Manacher's algorithm iterates string from left to right and uses above observations. Here is an example code:

Code listing 2.4 Manacher's algorithm for finding maximal palindromic substring radiuses

```python
def maximal_palindromic_substring_radiuses(s):
    if len(s) == 0:
        return []
    s = '#'.join(s)
    p = [0] * len(s)
    center, right = 0, 0
    for i in range(len(s)):
        if i < right:
            p[i] = min(right-i, p[2*center-i]) # use symmetry
        l, r = i - p[i], i + p[i]
        while l >= 0 and r < len(s) and s[l] == s[r]:
            p[i] += 1
            l -= 1
            r += 1
        if i + p[i] > right:
            center = i
            right = i + p[i]
    odd_pos_results = [(x + 1) // 2 for x in p[::2]]
    even_pos_results = [x // 2 for x in p[1::2]]
    p[::2] = odd_pos_results
    p[1::2] = even_pos_results
    return p
```

Let's notice using "two pointers technique" that this algorithm indeed has $\mathcal{O}(n)$ time complexity. We can choose i and $right$ as two pointers. They both start with 0 value. On each iteration of outside loop i is increased by 1, while on each iteration of inside loop $right$ is going to be increased by 1 in the following if-statement. Both i and $right$ can not exceed length of modified string. This proves $\mathcal{O}(n)$ time complexity.

For purposes of this thesis we are going to need Manacher's algorithm which works for Θ-palindromes instead of standard palindromes. Actually, there is not much work to do for generalizing Manacher's algorithm on Θ-palindromes.

Firstly, let's says that the special character s which we are adding in our alphabet in Manacher's algorithm works with Θ as follows: $\Theta(s) = s$.

Secondly, let's notice that all observations of Manacher's algorithm on standard palindromes apply on Θ-palindromes as well. Namely, it is not hard to check that previously mentioned list of statements:

- if $oldLeft > left$, then $newRadius = oldRadius$. (otherwise, v is not maximal)
- if $oldLeft < left$, then $newRadius = right - newCenter + 1, newRadius < oldRadius$. (otherwise, w is not maximal)
- if $oldLeft = left$, then $newRadius \geq oldRadius$.

works for Θ-palindromes.

It is important to notice, though, a small technical change for Θ-palindromes compared to standard ones - maximal radius values for modified string are not necessary odd values now, they can be zeros as well.

Here is an example code of Manacher's algorithm which works with Θ-palindromes:

Code listing 2.5 Manacher's algorithm for finding maximal Θ-palindromic substring radiuses

```
def maximal_palindromic_substring_radiuses(s, f):
    if len(s) == 0:
        return []
    s = '#'.join(s)
    p = [0] * len(s)
    center, right = 0, 0
    for i in range(len(s)):
        if i < right:
            p[i] = min(right-i, p[2*center-i])
        l, r = i - p[i], i + p[i]
        while l >= 0 and r < len(s) and f(s[l]) == s[r] and s[l] == f(s[r]):
            p[i] += 1
            l -= 1
            r += 1
        if i + p[i] > right:
            center = i
            right = i + p[i]
    odd_pos_results = [(x + 1) // 2 for x in p[::2]]
    even_pos_results = [x // 2 for x in p[1::2]]
    p[::2] = odd_pos_results
    p[1::2] = even_pos_results
    return p
```

2.3 Dynamic level ancestor problem

There are multiple versions of dynamic level ancestor problem and known solutions for them. Let's start from recapitulation of standard level ancestor problem.

Standard level ancestor problem is a problem in which we are given a static tree graph T with n nodes and chosen root node r. The goal is to spend some time preprocessing and then answer queries of type "for node u from T return node which is k nodes higher in T (higher means going into direction of r)".

You can find known solutions for standard level ancestor problem in [4], including the best known $\mathcal{O}(n)$ preprocessing time solution which uses $\mathcal{O}(1)$ time answering queries. This solution is not suitable for our purposes, though. Instead we will use solution (again from [4]) which uses jump-pointers and achieve $\mathcal{O}(\log n)$ query time by $\mathcal{O}(n \log n)$ preprocessing time.

Dynamic version of level ancestor problem which has interest for purposes of this thesis is modification of standard level ancestor problem for non-static trees. Given a tree graph T with n initial nodes and chosen root node r, spend some time for preprocessing and then answer two types of queries: "for node u from T return node which is k nodes higher in T" and "add new leaf node into T".

It is quite trivial that jump-pointer solution for standard level ancestor problem can be used for the above dynamic version of level ancestor problem. This algorithm is using $\mathcal{O}(n \log n)$ preprocessing time and answers both types of queries in $\mathcal{O}(\log n)$ time (n is node count of T before query).

The best known solution for such dynamic level ancestor problem which we found is solution by Alstrup; Holm described in [5]. They claim their algorithm uses $\mathcal{O}(n)$ preprocessing time and answers both types of our queries in constant time.

Chapter 3

Algorithms

In this chapter the following algorithms will be presented in section 3.1:

1. Single-threaded algorithm computing Θ-defect in $\mathcal{O}(n \log n)$ time by combining Manacher's algorithm and jump-pointers
2. Single-threaded algorithm computing Θ-defect in $\mathcal{O}(n)$ time by combining Manacher's algorithm with advanced solution of dynamic level ancestor problem
3. Single-threaded algorithm computing Θ-defect in $\mathcal{O}(n)$ time by using Manacher's algorithm and tree traversal
4. Single-threaded algorithm computing G-defect in $\mathcal{O}(n \cdot |G|)$ time by using previous algorithm and tree traversal
5. Multi-threaded algorithm computing G-defect in $\mathcal{O}(n)$ time using $\mathcal{O}(|G|)$ threads

After that in section 3.2, the following algorithms will be presented:

1. Single-threaded algorithm computing classical palindromic defect in $\mathcal{O}(n \log n)$ time by combining Manacher's algorithm and jump-pointers
2. Single-threaded algorithm computing classical palindromic defect in $\mathcal{O}(n)$ time by combining Manacher's algorithm with advanced solution of dynamic level ancestor problem
3. Single-threaded algorithm computing classical palindromic defect in $\mathcal{O}(n)$ time by using Manacher's algorithm and tree traversal

These algorithms are special cases of algorithms computing Θ-defect from section 3.1 in case when Θ is the reversal mapping R. R is antimorphism and $\forall a \in \mathcal{A} : \mathrm{R}(a) = a$.

Important notice is that all mentioned algorithms have promised time complexity only in cases when alphabet size is small compared to n or we are using hashing algorithm on the alphabet which performs queries in $\mathcal{O}(1)$ time. Alternatively, these time complexities can be achieved in case when all letters are enumerated by increasing space complexity of the algorithms.

In this chapter, we will use special notation w' to refer to a word built from word w by adding special character s between each pair of consecutive letters of w as it is done in Manacher's algorithm for Θ-palindromes.

3.1 Computing generalized palindromic defects

3.1.1 Θ-defect

Θ-defect is defined in definition 2.1. Just from observing this definition, we can conclude the following lemma.

▶ **Lemma 3.1.** *Let S be a single-threaded algorithm which finds amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}(f(n))$ time, then there exists a single-threaded algorithm P which computes Θ-defect in $\mathcal{O}(f(n)+n)$ time.*

Proof. Following definition,

$$\mathrm{D}_\Theta(w) = |w| + 1 - \#\operatorname{Pal}_\Theta(w) - \gamma_\Theta(w),$$

the $|w|$ and 1 terms are trivial. If we show that a single-thread algorithm K exists which computes the $\gamma_\Theta(w)$ term in $\mathcal{O}(n)$ time, then this lemma will be proven, because we can just take P as a consecutive run of S and K, followed by one addition and two subtractions.

By observing definition of the term,

$$\gamma_\Theta(w) := \#\{\{a, \Theta(a)\} \mid a \in \mathcal{A}, a \text{ occurs in } w \text{ and } a \neq \Theta(a)\},$$

it is not hard to see that we can simply iterate over all letters of w and for each $a \in w$ check if $a \neq \Theta(a)$. Then we would need to deal with duplicate letters. This can practically be done in several ways.

For example, hash table would give us $\mathcal{O}(1)$ time to add every of $\mathcal{O}(n)$ letters considering alphabet size is small compared to n or chosen hashing algorithm behaves well for our letters.

Another alternative, which is possible if all letters are enumerated, is to just use static array of counters of size $|\mathcal{A}|$. Initially we fill the array with zeros. When iterating over letter a, if $\Theta^2(a) \neq a$, then we increase counter for a. Otherwise, we increase counter for letter with minimal index: a or $\Theta(a)$. In the end, we count the amount of non-zero counters and get $\gamma_\Theta(w)$.

So we have built an algorithm K computing the $\gamma_\Theta(w)$ term in $\mathcal{O}(n)$ time, which ends proof of this lemma.

◀

If we assume that word w is given to us as sequence of letters, then $f(n)$ in above lemma is asymptotically at least linear.

Also it follows from the above lemma, that we can forget about algorithms for computing Θ-defect and talk about algorithms for counting amount of pairwise distinct Θ-palindromic substrings of a word instead. This is exactly what we are going to do.

We will be presenting algorithms counting pairwise distinct Θ-palindromic substrings of word w. All these algorithms will have following steps in common:

1. Insert a special character s between each pair of consecutive letters of w to get modified word w'. Reasoning here is the same as in section 2.2 when special character is added before main part of Manacher's algorithm for Θ-palindromes. $\Theta(s) = s$, w does not contain s.

2. While applying Manacher's algorithm to w', build some data structures.

3. By using data structures from previous step, build a tree graph T_w such that set of all nodes of T_w is in bijection with set of all pairwise distinct Θ-palindromic substrings of w'. The amount of nodes of T_w is same as the amount of pairwise distinct Θ-palindromic substrings of w'.

4. By making some simple observations on T_w structure, find amount of pairwise distinct Θ-palindromic substrings of w from it in linear time. This can also be done dynamically while building T_w.

Let's describe structure of T_w. As already mentioned, each node of T_w will represent a Θ-palindromic substring of w' which is different from other such substrings. Here are rules describing T_w structure:

- Nodes of T_w do not contain any data inside them.
- Edges of T_w contain one letter each.
- If there is an edge e with letter a from non-root node h to node l and h is higher in the tree than l and h represents word v, then node l represents word $av\,\Theta(a)$.
- If there is an edge e with letter a from root node to node l, then l represents word a.
- Root of T_w represents empty string ε.
- Every pairwise distinct Θ-palindromic substring of w' is represented in T_w by exactly one node.

For every possible w there always exists exactly one tree graph to which all these rules apply. This can be seen from the fact that w' contains only odd-length Θ-palindromic substrings (except empty string ε), and the fact that if w' contains Θ-palindromic substring $u = avb$, where $a, b \in \mathcal{A}$, then v is a Θ-palindromic substring of w' as well.

The next lemma helps with finding amount of pairwise distinct Θ-palindromic substrings of w from T_w:

▶ **Lemma 3.2.** *Amount of pairwise distinct Θ-palindromic substrings of w is equal to amount of nodes of T_w which do not represent word starting with special character s. In other words, it is equal to amount of edges of T_w which do not contain s character plus one.*

Proof. Let's show that there exists a bijection between all nodes of T_w which do not represent word starting with special character s and all pairwise distinct Θ-palindromic substrings of w.

From the way we defined w' by adding special character s between each pair of consecutive letters of w, it follows that a word v is a Θ-palindromic substring of w iff word v', which does not start with s, is a Θ-palindromic substring of w', where v' is v with s character added between each pair of consecutive letters. This gives us bijection between all Θ-palindromic substrings of w' not starting with s and all Θ-palindromic substrings of w.

There is an obvious bijection between all nodes of T_w which do not represent word starting with special character s and all Θ-palindromic substrings of w' not starting with s.

Combining these two bijections give us exactly what we wanted. ◀

Now considering above lemma, we can state another important lemma:

▶ **Lemma 3.3.** *Let S be a single-threaded algorithm which builds T_w in $\mathcal{O}(f(n))$ time, $f(n)$ is asymptotically at least linear. Then there exists a single-threaded algorithm P which finds amount of pairwise distinct Θ-palindromic substrings of w in $\mathcal{O}(f(n))$ time.*

Proof. Let's run S algorithm to build T_w. Then, using lemma 3.2, traverse T_w and find amount of pairwise distinct Θ-palindromic substrings of w in $\mathcal{O}(n)$ time. ◀

So being able to compute T_w fast would provide us with a way to compute amount of pairwise distinct Θ-palindromic substrings of a word fast. The following lemma will help us achieve it by establishing a link between structure of T_w and Manacher's algorithm for Θ-palindromes:

▶ **Lemma 3.4.** *T_w can be built in the following way. At the start T_w contains only root node, which represents empty string. Then Manacher's algorithm for Θ-palindromes is run on w' (special character is not added, w' already has it) and we try to add a new node to T_w each time maximal radius of current position is increased in Manacher's algorithm. The node we are adding to T_w is the one, which represent Θ-palindromic substring in the current position with the currently known maximal radius in this position of running Manacher's algorithm. In case there is already a node in T_w, which represents such Θ-palindrome, we do not add a new node.*

Proof. To prove this lemma we need to prove two things.

Firstly, we need to prove that adding a node in the described way is always possible. This means we need to prove that parent node already exists in T_w when we are adding a new node.

Secondly, we need to prove that in the end T_w will contain exactly one node for every pairwise distinct Θ-palindromic substring of w'. We can not have two nodes representing same Θ-palindrome simply by the way we are adding nodes to T_w. So we only need to show that we will try to add to T_w every pairwise distinct Θ-palindromic substring of w' at least once.

The following observation will help us to prove these statements.

By behaviour of Manacher's algorithm (see section 2.2), it can be seen that Manacher's algorithm iterates over w' letter positions from left to right. For each position it first defines initial radius as either zero or by coping it by symmetry from some previous position and possibly reducing copied radius. Then it iterates from the initial radius to actual maximal radius in the position (possibly this iteration has zero steps).

Now we can prove both statements by mathematical induction over the following statements:

- $A(k)$ is "When current position of Manacher's algorithm reaches $k+1$, then T_w contains nodes representing all Θ-palindromes centered at any of the first k letters of w'"
- $B(k)$ is "When adding all nodes for position $k+1$ into T_w, parent of each new node will already exist in T_w"

We start numerating letter positions with 1.

To prove that $B(k)$ is true, it is enough to show that parent node for the first node added in position $k+1$ exists in T_w before we try to add this node. For the rest of the added nodes, parent node already exists because we added it on the previous step of Manacher's algorithm.

Let's prove the statements $A(k)$ and $B(k)$ together by mathematical induction for k from 0 to $|w'|$. $B(|w'|)$ is considered to be true from the start. For $k = 0$, $A(0)$ is trivial, $B(0)$ is true, because T_w contains empty string from the start and initial radius in position 1 is zero.

For step $k \to k+1$, let's notice that $A(k+1)$ follows from $A(k)$ and $B(k)$. Indeed, based on $A(k)$, we only need to prove that all Θ-palindromes centered at $k+1$ are contained in T_w after Manacher's algorithm finishes iterating from initial radius to actual maximal radius at $k+1$. From $B(k)$ it follows that all Θ-palindromes centered at $k+1$ with radius less than initial radius are already represented in T_w before the iteration starts. After iteration is done, the rest of Θ-palindromes centered at $k+1$ are added to T_w. This proves $A(k+1)$.

Now we will prove $B(k+1)$ when we already know all $A(l), l \leq k+1$. If $k+1 = |w'|$, then there is nothing to prove. Let's see what happens when current position of Manacher's algorithm is $k+2 \leq |w'|$. If initial radius at $k+2$ is zero, then $B(k+1)$ is proven, because T_w contains node for empty string from the start. Otherwise, initial radius at $k+2$ was copied from actual maximal radius of some previous position $m \leq k+1$ and possibly reduced after copy. In both cases, $B(k+1)$ follows from $A(k+1)$ and $m \leq k+1$. This ends induction.

It is not hard to see that statements which we initially needed to prove are followed from statements we just proved by mathematical induction.

◀

Now we will present an algorithm which finds amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}\left(n^2\right)$ time.

▶ **Lemma 3.5.** *There exists an algorithm finding amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}\left(n^2\right)$ time using Manacher's algorithm for Θ-palindromes.*

Proof. It follows from lemma 3.3 that it is enough for us to present an algorithm which builds T_w in $\mathcal{O}\left(n^2\right)$ time. We can build T_w using lemma 3.4, but we still need to answer the question of how we will find parent node in T_w of a node we are currently trying to add.

We can just take Θ-palindrome represented by the parent node and find this Θ-palindrome in T_w in $\mathcal{O}(n)$ time by going down from the root of T_w. Doing it every time we try to add a node in T_w would give us an algorithm to build T_w in $\mathcal{O}\left(n^2\right)$ time.

◀

It is hard to improve time complexity of the above algorithm, so we need to introduce an additional data structure, which we will be building while running Manacher's algorithm for Θ-palindromes.

For every letter position p in w' we are going to remember two things:

1. Node v_p in T_w
2. Non-negative integer t_p

The limitations on choosing v_p and t_p pair are that v_p currently exists in T_w, depth of v_p in T_w is at least t_p and t_pth ancestor of v_p in T_w is a node representing Θ-palindrome centered at p with maximal possible radius.

▶ **Lemma 3.6.** *Pairs (v_p, t_p) for all positions p can be computed in $\mathcal{O}(n)$ time while running Manacher's algorithm for Θ-palindromes.*

Proof. We will set (v_p, t_p) pair right after Manacher's algorithm finishes iterating from initial radius to maximal radius in position p.

If initial radius was less than maximal radius in p, then at the last step of iteration we added node into T_w which represents Θ-palindrome centered at p with maximal possible radius, so we can set t_p as 0 and v_p as this node right after we added it.

If initial radius was equal to maximal radius in p, then there are two possibilities. First possibility is that both initial radius and maximal radiuses are 0. In this case we set v_p as root node of T_w and t_p as 0. Second possibility is that initial radius in p was copied from maximal radius in some previous position l and reduced by non-negative integer value d. In this case we can set $v_p = v_l$ and $t_p = t_l + d$. Limitations on pair (v_p, t_p) are not broken here. Limitations on t_p are not broken because maximal radius in position l was at least d simply by definition of d and correctness of Manacher's algorithm. Limitations on v_p are not broken because of behaviour of Manacher's algorithm for Θ-palindromes.

We can notice that in all cases we spend constant time to compute every next pair (v_p, t_p). And plain Manacher's algorithm for Θ-palindromes runs in $\mathcal{O}(n)$ time. So we find pairs (v_p, t_p) for all positions p in $\mathcal{O}(n)$ time.

◀

We are now ready to present three algorithms which find amount of pairwise distinct Θ-palindromic substrings of a word and have time complexities $\mathcal{O}(n \log n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n)$ respectively.

▶ **Theorem 3.7.** *There exists an algorithm finding amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}(n \log n)$ time by combining Manacher's algorithm for Θ-palindromes and jump-pointers.*

Proof. Our algorithm will follow the same basic structure as the algorithm from lemma 3.5. On top of that we will be building T_w with jump-pointers (see section 2.3), so adding new nodes into T_w will cost us $\mathcal{O}(\log n)$ time now. And we will be additionally computing (v_p, t_p) pairs while running Manacher's algorithm for Θ-palindromes (see lemma 3.6). Another possibility is to compute all (v_p, t_p) pairs beforehand in $\mathcal{O}(n)$ time.

Now let's show that finding parent node in T_w of a node we are currently trying to add can be done in $\mathcal{O}(\log n)$ time. Then following same logic as in algorithm from lemma 3.5, we will get the $\mathcal{O}(n \log n)$ time algorithm we promised. As can be seen from lemma 3.4, we are trying to add new nodes into T_w only when our Manacher's algorithm is currently in position p, which has initial radius less than maximal radius, and we increase current radius for position p.

Let's show how we can find in $\mathcal{O}(\log n)$ time node representing Θ-palindrome centered in p with radius equal to initial radius in p. If initial radius in p is 0, then we take the root node of T_w in constant time. If initial radius in p is greater than 0, then it was copied from some previous position l and reduced by non-negative integer d. Pair (v_l, t_l) was already computed, so we can take $(t_l + d)$th ancestor of v_l in $\mathcal{O}(\log n)$ time using jump-pointers. This is exactly the node we are looking for.

So we can find parent nodes for nodes, which are added when current radius in some position p is increased from initial radius, in $\mathcal{O}(\log n)$ time. For the nodes, which are added when current radius in p is increased not from initial radius, their parent node was found or added on the previous step of Manacher's algorithm for Θ-palindromes. So finding the parent node can be done in constant time in such case.

In worst case, we find parent node in T_w of a node we are currently trying to add in $\mathcal{O}(\log n)$ time. As already mentioned above, the promised algorithm with $\mathcal{O}(n \log n)$ time complexity follows from this. ◀

▶ **Theorem 3.8.** *There exists an algorithm finding amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}(n)$ time by combining Manacher's algorithm for Θ-palindromes and advanced solution for dynamic level ancestor problem.*

Proof. Let's notice that algorithm described in theorem 3.7 does not use any properties of jump-pointers expect that we use jump-pointers as a solution for dynamic level ancestor problem (see section 2.3).

So if we had a solution for dynamic level ancestor problem, such that we can add new leaf nodes in constant time and find ancestors in constant time, then we would build the promised algorithm based on algorithm from theorem 3.7.

Such solution is mentioned in section 2.3. ◀

▶ **Theorem 3.9.** *There exists an algorithm finding amount of pairwise distinct Θ-palindromic substrings of a word in $\mathcal{O}(n)$ time by using Manacher's algorithm for Θ-palindromes and tree traversal.*

Proof. From lemma 3.3 it follows that it is enough for us to present an algorithm which builds T_w in $\mathcal{O}(n)$ time.

And from lemma 3.4 follows that T_w can be built in the following way. Initially T_w contains only root node. Then we run Manacher's algorithm for Θ-palindromes on w', while collecting all pairs (p, r) of positions with radiuses such that Manacher's algorithm increased radius in position p from $r - 1$ to r. And then iterate over all these pairs (p, r) in some order and for each pair add node into T_w which represents palindrome centered in p with radius r. The only limitation on the order of (p, r) pairs is that when we are adding node into T_w, the parent node of this node must already exist in T_w.

To use the above observation we will first need to build an additional data structure.

While running Manacher's algorithm for Θ-palindromes, we will build an oriented forest graph H which will contain one node h_p for each letter position p of w'.

Every node h_p will contain two non-negative integers $incr_p$ and $decr_p$. $incr_p$ is equal to maximal radius in position p minus initial radius in p as in Manacher's algorithm. If Manacher's algorithm copied initial radius in position p from some previous position and reduced it by non-negative integer d, then $decr_p = d$. Otherwise, initial radius in position p is 0 and we define $decr_p = 0$.

Additionally every node h_p will contain index of position p in w', so for every node h_p we can go to position p in w' in constant time.

Edges of H will be added based on how Manacher's algorithm copies radiuses of previous positions. An edge e from node h_l to node h_p exists in H iff Manacher's algorithm copied initial radius in position p from maximal radius in previous position l.

It is not hard to see that H can be built in $\mathcal{O}(n)$ time by running Manacher's algorithm for Θ-palindromes, because every $incr_p$ and $decr_p$ can be computed in constant time in all cases and every edge can be added in constant time. Details of implementation are left to readers as an exercise.

The fact, that H is indeed an oriented forest, follows from how edges of H were defined.

Actually, we do not need all the data contained in H, so we will build another oriented forest H' by removing some nodes from H.

H' starts off as a copy of H and then we consecutively remove all nodes h_p from H' such that $incr_p = 0$, while adding $decr_p$ values lost from removal of the nodes to descendant nodes. In other words, when node h_p is removed, then for each direct descendant h_l of h_p we modify $decr_l = decr_l + decr_p$. If h_p is a root node for some tree in H', then we simply remove node h_p and all of its edges, meaning amount of trees in H' might change after this operation. If h_p is not a root node and has a parent node h_t, then for each direct descendant h_l of h_p we add an edge from h_t to h_p, and finally, we remove node h_p and all of its edges.

The order in which nodes are removed from H' is not arbitrary. At the start we choose an arbitrary order in which trees of H' will be traversed. Then we traverse each tree from the root node using depth-first traversal, while removing the nodes as mentioned above. Here it is important that we traverse the trees in such way that after node h_p was visited, all ascendants of h_p were already visited before and none of them will be removed later.

Let's notice that in the described way of building H' we spend $\mathcal{O}(\deg_{\text{in}}(h_p) + \deg_{\text{out}}(h_p))$ time to remove node h_p and the value $\deg_{\text{in}}(h_l)+\deg_{\text{out}}(h_l)$ is not increased for any descendent nodes h_l of h_p. From this observation follows, that the described way of building H' from H has time complexity $\mathcal{O}\left(\#\,\text{nodes}(H) + \#\,\text{edges}(H) + \sum_{h_p\in\text{nodes}(H)} \deg(h_p)\right) = \mathcal{O}(n + n + \#\,\text{edges}(H))$ which in fact is $\mathcal{O}(n)$.

In reality we can build H' instead of H from the start during run of Manacher's algorithm for Θ-palindromes. Details of implementation and proof are left to readers as an exercise.

As can be seen from definition of H', H' contains one node h'_p for each position p of w' such that maximal radius in p is bigger than initial radius in p in Manacher's algorithm. All $incr_p$ values in H' are positive.

H has one property which is important for us. For each node h_p in H, initial radius in position p is equal to sum of all $incr_a - decr_a$ values for all ancestors h_a of h_p minus $decr_p$. H follows this property because of how it is defined based on Manacher's algorithm. Moreover, H' also follows this property. It can be seen from the fact that when we are building H', it starts off as H, which follows the property, and the property is not broken after each node removal in H' (notice how we modify $decr_p$ values for this).

From now on we will not need H anymore, so any mentions of $incr_p$ and $decr_p$ values will be referring to the values in nodes of H'.

Now let's build T_w using H'. Initially, T_w contains only root node, which represents an empty string ε.

We will present an algorithm S for building part of T_w while traversing one tree F of H'. This algorithm is then applied to every tree of H' consecutively. The order, in which trees are traversed, is arbitrary.

While traversing F, we will be remembering some path $curPath$ of T_w, which start at the root node of T_w. We will also have a dynamic array $nodesLeft$, which has the same size as amount of nodes in $curPath$. $nodesLeft$ will contain as its elements dynamic arrays of references to some nodes of F.

Let's describe how $curPath$ and $nodesLeft$ will be changing during traversal of F and how they will affect traversal of F.

When algorithm S comes to a node $h'_p \in F$ during traversal and it wants to mark h'_p as visited, then it follows the next procedure. By the way S will behave, at this moment $curPath$ will end with a node from T_w which represents Θ-palindrome centered in p with radius equal to initial radius in p. Before marking h'_p as visited, S iterates from initial radius in p plus one to maximal radius in p and tries to add new node into T_w representing Θ-palindrome centered in p with current radius. S also pushes references to these new nodes or already existing nodes into $curPath$, and it fills $nodesLeft$ with empty dynamic arrays so it has the same size as node count in $curPath$. After this part is done, S iterates over all direct descendants h'_l of h'_p in F and it adds a reference to h'_l in $nodesLeft$ array into dynamic array on index $(\operatorname{len}(nodesLeft) - decr_l)$. And finally, S marks h'_p as visited and continues. Time complexity of this operation is $\mathcal{O}\left(incr_p + \deg_{\text{out}}(h'_p)\right)$, because we get p from F in constant time, we get initial radius from node count of $curPath$, then we do $incr_p$ constant time operations of trying to add new node into T_w, and finally we add reference into $nodesLeft$ for each of $\deg_{\text{out}}(h'_p)$ direct descendants of h'_p, adding each reference costs constant time.

When algorithm S needs to choose node h'_p to apply the above procedure of marking h'_p as visited, it follows the next procedure. If the last dynamic array in $nodesLeft$ is not empty, then S pops the last node from it and takes this node as h'_p. If the last dynamic array in $nodesLeft$ is empty, then S decreases size of $nodesLeft$ by 1 and removes the last node from $curPath$, and after that S repeats this procedure of choosing h'_p. If $nodesLeft$ has size 0, then S finishes.

Initially S starts with procedure of marking root node of F as visited.

As can be seen from definition of S, all nodes of F will indeed be marked as visited at the end of S. It follows from the fact that after we mark some node h'_p as visited, then we add references to all direct descendants of h'_p somewhere into $nodesLeft$ and so we will mark all of them as visited some time later.

From definition of S and properties of H' mentioned above, it follows that for node h'_p from F, before S marked h'_p as visited, S tried to add into T_w nodes representing all Θ-palindromes centered in p with radius bigger than initial radius in p and not bigger than maximal radius in p.

So if we apply S for all trees of H' in some arbitrary order, then we will try to add to T_w nodes representing all Θ-palindromes centered in position p with radius r, which is bigger than initial radius in p and not bigger than maximal radius in p, for all pairs (p, r).

Then it follows from the observation on how T_w can be built, which was made at the start of this proof, that the above algorithm actually builds whole T_w.

Now it is only left to show that the above algorithm of building T_w from H' has $\mathcal{O}(n)$ time complexity. This will finish the proof, because we can build H in $\mathcal{O}(n)$ time, then build H' from H in $\mathcal{O}(n)$ time, and finally build T_w from H' in $\mathcal{O}(n)$ time. Combining all three algorithms, we get an algorithm building T_w in $\mathcal{O}(n)$ time, which is exactly what we wanted, as mentioned at the very start of this proof.

Let's notice that all calls of procedure of S to choose node h'_p do the same amount of constant-time steps as amount of times $curPath$ is increased by 1 or a some reference is added into $nodesLeft$ somewhere during run of S. Indeed, at the start of S and at the end of S, both

$curPath$ and $nodesLeft$ are empty. Nodes are removed from $curPath$ and references are removed from $nodesLeft$ only during procedure of choosing node h'_p.

It follows that time spent on choosing nodes during run of S is not asymptotically bigger than time spent on marking nodes as visited. So time complexity of S applied to F is equal to $\mathcal{O}\left(\sum_{h'_p\in F} incr_p + \sum_{h'_p\in F} \deg_{\text{out}}(h'_p)\right)$. Then time complexity of algorithm for building T_w from H' by using algorithm S is $\mathcal{O}\left(\sum_{h'_p\in H'} incr_p + \sum_{h'_p\in H'} \deg_{\text{out}}(h'_p)\right) = \mathcal{O}\left(n + \sum_{h'_p\in H'} incr_p\right)$. Sum $\sum_{h'_p\in H'} incr_p$ is equal to amount of time radius in current position is increased during Manacher's algorithm, so $\mathcal{O}\left(\sum_{h'_p\in H'} incr_p\right) = \mathcal{O}(n)$. Then it follows that $\mathcal{O}\left(n + \sum_{h'_p\in H'} incr_p\right)$ is same as $\mathcal{O}(n)$. This proofs that our algorithm to build T_w from H' has $\mathcal{O}(n)$ time complexity, which is exactly what we wanted.

◀

Going back to computing Θ-defect, let's present the initially promised algorithms.

▶ **Lemma 3.10.** *There exists an algorithm computing Θ-defect in $\mathcal{O}(n\log n)$ time by combining Manacher's algorithm for Θ-palindromes and jump-pointers.*

Proof. Combine theorem 3.7 and lemma 3.1.

◀

▶ **Lemma 3.11.** *There exists an algorithm computing Θ-defect in $\mathcal{O}(n)$ time by combining Manacher's algorithm for Θ-palindromes and advance solution of dynamic level ancestor problem.*

Proof. Combine theorem 3.8 and lemma 3.1.

◀

▶ **Lemma 3.12.** *There exists an algorithm computing Θ-defect in $\mathcal{O}(n)$ time by using Manacher's algorithm for Θ-palindromes and tree traversal.*

Proof. Combine theorem 3.9 and lemma 3.1.

◀

3.1.2 G-defect

For purposes of multi-threaded algorithms presented in this subsection, we will require an implementation of concurrent hash table/map which has additional properties that any amount of parallel queries to remove the same element from it has constant time complexity and that any amount of parallel queries to add the same element to it has constant time complexity. Implementation details of such version of concurrent hash table/map are left to readers as an exercise.

In this subsection, we will be using symbol r specifically to represent amount of antimorphisms contained in G. It follows from lemma 2.3 that $r = \mathcal{O}(|G|)$.

Any mention of concurrent hash table/map data structure in this subsection will be referring to implementation of concurrent hash table/map mentioned above.

We will also use $T_{w,\Theta}$ notation to refer to tree T_w defined in subsection 3.1.1 built for word w and antimorphism Θ.

G-defect is defined in definition 2.4. Just from observing this definition, we can conclude the following lemma.

▶ **Lemma 3.13.** *Let S be a single-threaded algorithm which finds $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}(f(n,r))$ time, then there exists a single-threaded algorithm P which computes G-defect in $\mathcal{O}(f(n,r) + r\cdot n)$ time.*

Proof. Following definition,

$$\mathrm{D}_G(w) = |w| + 1 - \#\,\mathrm{Pal}_G(w) - \gamma_G(w)$$

the $|w|$ and 1 terms are trivial. If we show that a single-thread algorithm K exists which computes the $\gamma_G(w)$ term in $\mathcal{O}\left(r \cdot n\right)$ time, then this lemma will be proven, because we can just take P as a consecutive run of S and K, followed by one addition and two subtractions.

By observing definition of the term,

$$\gamma_G(w) := \#\{[a] \mid a \in \mathcal{A}, a \text{ occurs in } w, \text{and } a \neq \Theta(a) \text{ for every antimorphism } \Theta \in G\},$$

it is not hard to see that we can simply iterate over all letters of w and for each $a \in w$ check if $a \neq \Theta(a)$ for every antimorphism $\Theta \in G$ and mark all such letters a. This requires $\mathcal{O}\left(r \cdot n\right)$ time. Then we would need to deal with duplicate letters. This can practically be done in several ways.

We will show how to do it using hash table. Adding one letter to hash table would require $\mathcal{O}\left(1\right)$ time considering alphabet size is small compared to n or chosen hashing algorithm behaves well for our letters. So we try to add all letters, which we marked above, into hash table. It takes $\mathcal{O}\left(n\right)$ and hash table does not contain duplicate letters in the end.

Now when we have the hash table, we need to find how many pairwise distinct classes of equivalence $[a]$ there are among letters a of our hash table. We can do it by taking an arbitrary element a of our hash table and trying to remove elements $\varphi(a)$ from hash table for all $\varphi \in G$. Then repeat this for another arbitrary element of hash table until hash table is empty. Overall time complexity of this is $\mathcal{O}\left(r \cdot n\right)$. Amount of iterations of this algorithm is equal to amount of pairwise distinct classes of equivalence $[a]$ in our hash table. This is exactly what we want to compute.

Summing up, we have built an algorithm K computing the $\gamma_G(w)$ term in $\mathcal{O}\left(r \cdot n\right)$ time, which ends proof of this lemma.

◀

We can also prove similar lemma for multi-threaded algorithms.

▶ **Lemma 3.14.** *Let S be a multi-threaded algorithm which uses $k \leq r$ threads and finds $\#\,\mathrm{Pal}_G(w)$ in $\mathcal{O}\left(f(n,r,k)\right)$ time, then there exists a multi-threaded algorithm P which computes G-defect in $\mathcal{O}\left(f(n,r,k) + \frac{r \cdot n}{k}\right)$ time.*

Proof. Following definition,

$$\mathrm{D}_G(w) = |w| + 1 - \#\,\mathrm{Pal}_G(w) - \gamma_G(w)$$

the $|w|$ and 1 terms are trivial. If we show that a multi-thread algorithm K using k threads exists which computes the $\gamma_G(w)$ term in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time, then this lemma will be proven, because we can just take P as a consecutive run of S and K, followed by one addition and two subtractions.

By observing definition of the term,

$$\gamma_G(w) := \#\{[a] \mid a \in \mathcal{A}, a \text{ occurs in } w, \text{and } a \neq \Theta(a) \text{ for every antimorphism } \Theta \in G\},$$

it is not hard to see that we can iterate over all letters of w and for each $a \in w$ check if $a \neq \Theta(a)$ for every antimorphism $\Theta \in G$ and mark all such letters a. This can be done in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time using k threads. Indeed, let's first setup an atomic boolean for each letter of w as true. This setup can be done in $\mathcal{O}\left(n\right)$ time using one thread. Then we numerate all pairs (a, Θ) and make thread with index i iterate all pairs with indices j such that $j \equiv i \pmod{k}$. If some thread finds that $a = \Theta(a)$ for some Θ, then it sets the atomic boolean of letter a as false. In the end, we are interested only in letters with atomic boolean with true value.

Now we need to deal with duplicate letters. This can practically be done in several ways.

We will show how to do it using concurrent hash table. Adding one letter to concurrent hash table would require $\mathcal{O}(1)$ time considering alphabet size is small compared to n or chosen hashing algorithm behaves well for our letters. So we try to add all letters, which have atomic boolean with true value, into concurrent hash table. It takes $\mathcal{O}(n)$ using one thread and concurrent hash table does not contain duplicate letters in the end.

Now when we have the concurrent hash table, we need to find how many pairwise distinct classes of equivalence $[a]$ there are among letters a of our concurrent hash table. We can do it by taking an arbitrary element a of our concurrent hash table and trying to remove elements $\varphi(a)$ from concurrent hash table for all $\varphi \in G$. We do it using k threads by numerating all $\varphi_j \in G$ and making thread with index i iterate all φ_j with indices j such that $j \equiv i \pmod{k}$. Then we repeat this for another arbitrary element of concurrent hash table until concurrent hash table is empty. Overall time complexity of this is $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$. Amount of iterations of this algorithm is equal to amount of pairwise distinct classes of equivalence $[a]$ in our concurrent hash table. This is exactly what we want to compute.

Summing up, we have built an algorithm K computing the $\gamma_G(w)$ term in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time, which ends proof of this lemma.

◀

It follows from the above lemmas, that we can forget about algorithms for computing G-defect and talk about algorithms for computing $\#\operatorname{Pal}_G(w)$ instead. This is exactly what we are going to do.

To compute $\#\operatorname{Pal}_G(w)$, our algorithms will be building a special tree graph to which we will refer as $T_{w,G}$. It will be built using trees $T_{w,\Theta}$ built for all antimorphisms $\Theta \in G$.

Before describing structure of $T_{w,G}$, we need to pick an arbitrary order of all morphisms $f \in G$. We will refer to morphism with index i as f_i.

It is also important to say that special character s, which is used to build w' from w, is chosen the same for all antimorphisms $\Theta \in G$, and $\forall \Theta \in G : \Theta(s) = s$. It follows from definition that all G-palindromic substrings of w' have odd length (except empty string ε).

Now let's describe structure of $T_{w,G}$. Every node h of $T_{w,G}$ will represent a G-palindromic class of equivalency $[v']$, v' is a substring of w', which is different from G-palindromic classes of equivalency represented by other nodes of $T_{w,G}$. We will refer to node representing a class of equivalency $[v']$ as $h_{v'}$. Here are rules describing $T_{w,G}$ structure:

- Edges of $T_{w,G}$ contain two letters each. For edge e, we will refer to the first letter as e_l and to the second letter as e_r. $e_l = e_r$ for all edges from root node of $T_{w,G}$.
- If there is an edge e with $e_l = a$ and $e_r = b$ from non-root node h to node l and h is higher in the tree than l and h represents $[v]$, then node l represents $[avb]$.
- If there is an edge e with $e_l = e_r = a$ from root node to node l, then l represents $[a]$.
- Root of $T_{w,G}$ represents $[\varepsilon]$.
- Every pairwise distinct G-palindromic class of equivalency $[v']$, such that v' is substring of w', is represented in $T_{w,G}$ by exactly one node.
- Node $h_{v'}$ of $T_{w,G}$ will contain a dynamic array of size r of references to hash maps $mapRef_{i,v'}$. Two references $mapRef_{i,v'}$ and $mapRef_{j,v'}$ might reference the same hash map. We will refer to hash maps by $map_{i,v'}$. Keys of the hash maps are pairs of letters, and values are references to other such hash maps. All these hash maps are maintained in one dynamic array outside of $T_{w,G}$. We will be saying that hash map $map_{i,v'}$ represents string $f_i(v')$ and $map_{i,\varepsilon}$ represents ε.

- For node $h_{v'}$ with parent edge e with $e_l = a$ and $e_r = b$, $\mathit{mapRef}_{i,v'} = \mathit{mapRef}_{j,v'} \iff (f_i(a), f_i(b)) = (f_j(a), f_j(b))$. For root node h_ε of $T_{w,G}$, $\forall i, j : \mathit{mapRef}_{i,\varepsilon} = \mathit{mapRef}_{j,\varepsilon}$.
- There is an edge e with $e_l = a$ and $e_r = b$ from node $h_{v'}$ to node $h_{av'b}$ iff hash map $\mathit{map}_{i,v'}$ in node $h_{v'}$ contains an element with the key $(f_i(a), f_i(b))$ and the value $\mathit{mapRef}_{i,av'b}$.

For every possible w there always exists exactly one tree graph to which all these rules apply. Uniqueness of data inside nodes follows from how data inside nodes is defined. The rest can be seen from the fact that w' contains only odd-length G-palindromic substrings (except empty string ε), and the fact that if w' contains G-palindromic substring $u = avb$, where $a, b \in \mathcal{A}$, then v is a G-palindromic substring of w' as well.

Every hash map map of $T_{w,G}$ is always referenced only in one node h of $T_{w,G}$. We will refer to h as node containing map and put a reference to h alongside map so we can access it in constant time.

The next lemma helps with finding $\#\,\mathrm{Pal}_G(w)$ from $T_{w,G}$:

▶ **Lemma 3.15.** *$\#\,\mathrm{Pal}_G(w)$ is equal to amount of nodes of $T_{w,G}$ which do not represent $[v']$ such that v' starts with special character s. In other words, it is equal to amount of edges e of $T_{w,G}$ such that $e_l \neq s$ plus one.*

Proof. Let's show that there exists a bijection between all nodes $h_{v'}$ of $T_{w,G}$ such that v' does not start with s and all pairwise distinct G-palindromic classes of equivalency $[v]$, v is a substring of w.

From the way we defined w' by adding special character s between each pair of consecutive letters of w, it follows that a word v is a G-palindromic substring of w iff word v', which does not start with s, is a G-palindromic substring of w', where v' is v with s character added between each pair of consecutive letters. This gives us bijection between all G-palindromic substrings of w' not starting with s and all G-palindromic substrings of w.

Also from definition of s it follows that for any v, u two G-palindromic substrings of $w : v \in [u] \iff v' \in [u']$. This gives us bijection between all pairwise distinct G-palindromic classes of equivalency $[v']$, where v' is a substring of w' and v' does not start with s, and all pairwise distinct G-palindromic classes of equivalency $[v]$, v is a substring of w.

By definition of $T_{w,G}$, there is an obvious bijection between all nodes $h_{v'}$ of $T_{w,G}$ such that v' does not start with s and all pairwise distinct G-palindromic classes of equivalency $[v']$, where v' is a substring of w' and v' does not start with s.

Combining the last two bijections give us exactly what we wanted.

◀

Now considering above lemma, we can state several other important lemmas.

▶ **Lemma 3.16.** *Let S be a single-threaded algorithm which builds $T_{w,G}$ in $\mathcal{O}(f(n,r))$ time. Then there exists a single-threaded algorithm P which computes $\#\,\mathrm{Pal}_G(w)$ in $\mathcal{O}(f(n,r)+n)$ time.*

Proof. Let's run S algorithm to build $T_{w,G}$. Then, using lemma 3.15, traverse $T_{w,G}$ and compute $\#\,\mathrm{Pal}_G(w)$ in $\mathcal{O}(n)$ time.

◀

▶ **Lemma 3.17.** *Let S be a multi-threaded algorithm which uses k threads and builds $T_{w,G}$ in $\mathcal{O}(f(n,r,k))$ time. Then there exists a multi-threaded algorithm P which uses k threads and computes $\#\,\mathrm{Pal}_G(w)$ in $\mathcal{O}(f(n,r,k)+n)$ time.*

Proof. Let's run S algorithm to build $T_{w,G}$. Then, using lemma 3.15, traverse $T_{w,G}$ with one thread and compute $\#\,\mathrm{Pal}_G(w)$ in $\mathcal{O}(n)$ time.

◀

Now we are ready to present single-threaded algorithms for computing $\#\operatorname{Pal}_G(w)$ and G-defect.

▶ **Theorem 3.18.** *Let S be a single-threaded algorithm which builds $T_{w,\Theta}$ for arbitrary antimorphism $\Theta \in G$ in $\mathcal{O}(f(n))$ time, then there exists a single-threaded algorithm P which computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}(r \cdot f(n) + r \cdot n)$ time.*

Proof. We start by running S for all antimorphisms $\Theta \in G$ and by doing so we build $T_{w,\Theta}$ for all antimorphisms $\Theta \in G$ in $\mathcal{O}(r \cdot f(n))$ time.

We also compute all f_i^{-1} in $\mathcal{O}(r \cdot n)$ time.

If we now present a single-threaded algorithm which builds $T_{w,G}$ in $\mathcal{O}(r \cdot n)$ time using $T_{w,\Theta}$ trees, then by summing up we get a single-threaded algorithm which builds $T_{w,G}$ in $\mathcal{O}(r \cdot f(n) + r \cdot n)$ time. By combining this algorithm with lemma 3.16, we get a single-threaded algorithm which computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}(r \cdot f(n) + r \cdot n)$ time, which is exactly what we want.

Now we will present a single-threaded algorithm K which builds $T_{w,G}$ in $\mathcal{O}(r \cdot n)$ time using $T_{w,\Theta}$ trees, which will finish this theorem.

K will consecutively traverse all trees $T_{w,\Theta}$ in arbitrary order by depth-first traversal while trying to add new nodes into $T_{w,G}$.

When K is visiting node in $T_{w,\Theta}$ which represents Θ-palindrome $u' = u_1u_2 \dots u_{2n-1}$, then it will remember hash maps in $T_{w,G}$ which represents $\varepsilon, u_n, \cdots, u_2u_3 \dots u_{2n-2}$ and u'.

If K goes up one node in $T_{w,\Theta}$, then it forgets the last of hash maps. This operation is done in constant time.

If K goes down one node in $T_{w,\Theta}$ from node h to node l using edge e with letter a, then there are two cases.

If the last of hash maps contains key $(a, \Theta(a))$ with value being a reference to *map*, then K adds *map* as the last hash map it currently remembers. This is done in constant time.

If the last of hash maps does not contain key $(a, \Theta(a))$, then K adds a new node into $T_{w,G}$. $b = \Theta(a)$. This new node $h_{a'v'b'}$ is a direct descendant of node $h_{v'}$ of the last hash map. Edge e from $h_{v'}$ to $h_{a'v'b'}$ has $a' = e_l = f_i^{-1}(a)$ and $b' = e_r = f_i^{-1}(b)$. To choose i in constant time, K needs to additionally remember for every hash map $map_{i,v'}$ one of indices i. After $h_{a'v'b'}$ is added, K needs to fill data for it and create new hash maps. To do this, K builds a hash map with keys being tuples $(mapRef_{i,v'}, f_i(a'), f_i(b'))$ for all i and values being a new hash map $newMap_i$. Then, for all i, K sets $mapRef_{i,a'v'b'} = ref(newMap_i)$ and adds to $map_{i,v'}$ element with key $(f_i(a'), f_i(b'))$ and value $mapRef_{i,a'v'b'}$. Overall time complexity of this operation of K is $\mathcal{O}(r)$.

Let's notice that in total K adds $\mathcal{O}(n)$ nodes into $T_{w,G}$ and operation of K which adds a node into $T_{w,G}$ costs $\mathcal{O}(r)$ time, so total time spent on such operations is $\mathcal{O}(r \cdot n)$. At the same time, all other operations of K have constant time complexity and their amount is $\mathcal{O}(r \cdot n)$. It follows that time complexity of K is $\mathcal{O}(r \cdot n)$.

As can be seen from definition of $K, T_{w,\Theta}$ and $T_{w,G}$, after K finishes traversal of $T_{w,\Theta}$ for some antimorphism Θ, all Θ-palindromic substrings of w' are represented by some hash map in $T_{w,G}$ and so all G-palindromic classes of equivalency $[v']$, where v' is a substring of w' and v' is a Θ-palindrome, are represented by some node in $T_{w,G}$.

Summing up, after K finishes traversal of all $T_{w,\Theta}$, $T_{w,G}$ contains nodes representing all G-palindromic classes of equivalency $[v']$, where v' is a substring of w'. ◀

▶ **Lemma 3.19.** *There exists an algorithm computing G-defect in $\mathcal{O}(r \cdot n)$ time.*

Proof. By combining algorithm described inside proof of theorem 3.9 and theorem 3.18, we get an algorithm which computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}(r \cdot n)$ time. By combining this algorithm with lemma 3.13, we get an algorithm which we want. ◀

Now we will present multi-threaded algorithms for computing $\#\operatorname{Pal}_G(w)$ and G-defect.

▶ **Theorem 3.20.** *Let S be a single-threaded algorithm which builds $T_{w,\Theta}$ for arbitrary antimorphism $\Theta \in G$ in $\mathcal{O}(f(n))$ time, then there exists a multi-threaded algorithm P using $k \le r$ threads which computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}\left(\frac{r \cdot f(n) + r \cdot n}{k}\right)$ time.*

Proof. We start by running S for all antimorphisms $\Theta \in G$ using k threads, up to k instances of S are running at the same. By doing so we build $T_{w,\Theta}$ for all antimorphisms $\Theta \in G$ in $\mathcal{O}\left(\frac{r \cdot f(n)}{k}\right)$ time.

We also compute all f_i^{-1} in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time.

If we now present a multi-threaded algorithm which uses $k \le r$ threads and builds $T_{w,G}$ in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time using $T_{w,\Theta}$ trees, then by summing up we get a multi-threaded algorithm which uses $k \le r$ threads and builds $T_{w,G}$ in $\mathcal{O}\left(\frac{r \cdot f(n) + r \cdot n}{k}\right)$ time. By combining this algorithm with lemma 3.17, we get a multi-threaded algorithm which uses $k \le r$ threads and computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}\left(\frac{r \cdot f(n) + r \cdot n}{k} + n\right) = \mathcal{O}\left(\frac{r \cdot f(n) + r \cdot n}{k}\right)$ time, which is exactly what we want.

Now we will present a multi-threaded algorithm M which uses $k \le r$ threads and builds $T_{w,G}$ in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time using $T_{w,\Theta}$ trees, which will finish this theorem.

We will be using an algorithm K which builds $T_{w,G}$ in $\mathcal{O}(r \cdot n)$ time described in theorem 3.18. K consecutively traverses all trees $T_{w,\Theta}$ by algorithm L.

M also traverses all trees $T_{w,\Theta}$ by algorithm L, but it runs up to k instances of L.

M uses concurrent hash maps in $T_{w,G}$ and for the additional hash map created when a new node is being added to $T_{w,G}$.

It is important to understand how M synchronizes between threads to achieve $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time complexity.

When a thread detects that it needs to add a new node, then it does not do it by itself, but synchronizes with other threads and reports that a new node needs to be added. This operation of reporting is done in constant time.

After each instance of L in M makes one step of traversal of its $T_{w,\Theta}$, all threads synchronize and check if there is at least one new node to add, reported by one of the threads. If there is a new node to add, then all threads participate in adding this new node, and then all threads rollback their state to state before the last step of traversal. Steps of traversal of $T_{w,\Theta}$ include ascending by one node, descending by one node without new node in $T_{w,G}$, descending by one node with new node in $T_{w,G}$.

Algorithm for adding a new node into $T_{w,G}$ is slightly modified in M compared to how to works in K. Instead of initializing as many as needed new hash maps, M initialized exactly r new concurrent hash maps. Some of these new concurrent hash maps will not be used at all after operation of adding new node to $T_{w,G}$ is finished, it is decided by which thread is faster to add a key into the concurrent hash map with keys $(mapRef_{i,v'}, f_i(a'), f_i(b'))$.

Synchronized operations in M all have constant time complexity. Initially, amount of synchronized operations is $\mathcal{O}(r \cdot n)$. Considering $\mathcal{O}(n)$ rollbacks of k operations, time spent on synchronized operations using k threads is $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$.

Operation in M to add new node into $T_{w,G}$ has $\mathcal{O}\left(\frac{r}{k}\right)$ time complexity.

It follows that time complexity of M is $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$, which is exactly what we wanted.

◀

▶ **Lemma 3.21.** *There exists a multi-threaded algorithm which uses $k \le r$ threads and computes G-defect in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time.*

Proof. By combining algorithm described inside proof of theorem 3.9 and theorem 3.20, we get a multi-threaded algorithm which uses k threads and computes $\#\operatorname{Pal}_G(w)$ in $\mathcal{O}\left(\frac{r \cdot n}{k}\right)$ time. By combining this algorithm with lemma 3.14, we get an algorithm which we want.

◀

3.2 Computing classical palindromic defect

Let's notice that the whole subsection 3.1.1 does not make any actual use from working with Θ-palindromes, except at the very start in lemma 3.1 and at the very end in lemma 3.10, lemma 3.11 and lemma 3.12. So if we skip these parts and read the rest while changing all mentions of Θ-palindromes into classical palindromes, then all statements will work for classical palindromes as well.

We are not going to rewrite whole subsection 3.1.1 here. Instead we will simply present results for classical palindromes as special cases of results, which were already achieved in subsection 3.1.1.

▶ Note 3.22. Classical palindromic defect of word w is defined as $|w|+1$ minus amount of pairwise distinct palindromic substrings of w. So to compute classical palindromic defect in $\mathcal{O}(f(n))$ time, where $f(n)$ is at least linear, it is enough to find amount of pairwise distinct palindromic substrings of w in $\mathcal{O}(f(n))$ time and then perform one addition and one subtraction.

▶ **Lemma 3.23.** *There exists an algorithm finding amount of pairwise distinct palindromic substrings of a word in $\mathcal{O}(n \log n)$ time by combining Manacher's algorithm and jump-pointers.*

Proof. Proof follows as special case of theorem 3.7 when Θ is chosen as the reversal mapping R. R is an antimorphism and $\forall a \in \mathcal{A} : \mathrm{R}(a) = a$. ◀

▶ **Lemma 3.24.** *There exists an algorithm finding amount of pairwise distinct palindromic substrings of a word in $\mathcal{O}(n)$ time by combining Manacher's algorithm and advanced solution for dynamic level ancestor problem.*

Proof. Proof follows as special case of theorem 3.8 when Θ is chosen as the reversal mapping R. R is an antimorphism and $\forall a \in \mathcal{A} : \mathrm{R}(a) = a$. ◀

▶ **Lemma 3.25.** *There exists an algorithm finding amount of pairwise distinct palindromic substrings of a word in $\mathcal{O}(n)$ time by using Manacher's algorithm and tree traversal.*

Proof. Proof follows as special case of theorem 3.9 when Θ is chosen as the reversal mapping R. R is an antimorphism and $\forall a \in \mathcal{A} : \mathrm{R}(a) = a$. ◀

▶ **Lemma 3.26.** *There exists an algorithm computing classical palindromic defect in $\mathcal{O}(n \log n)$ time by combining Manacher's algorithm and jump-pointers.*

Proof. Combine lemma 3.23 and note 3.22. ◀

▶ **Lemma 3.27.** *There exists an algorithm computing classical palindromic defect in $\mathcal{O}(n)$ time by combining Manacher's algorithm and advance solution of dynamic level ancestor problem.*

Proof. Combine lemma 3.24 and note 3.22. ◀

▶ **Lemma 3.28.** *There exists an algorithm computing classical palindromic defect in $\mathcal{O}(n)$ time by using Manacher's algorithm and tree traversal.*

Proof. Combine lemma 3.25 and note 3.22. ◀

Chapter 4

SageMath

4.1 Survey on development processes in SageMath

Here I am going to go through development processes used in SageMath development, including tools which I used during coding, development workflow (how to merge you changes, how they are reviewed and so on) and basic structure of code in SageMath (how to add/modify code, how to add/modify tests and so on).

4.1.1 Tools

Among non-trivial tools, which I used during coding, there is user version of SageMath and conda. Having user version of SageMath is logical to check existing functionality and check its behaviour in some corner cases. But it can be tricky to have a plain user version of SageMath installed, so I used conda for creating a preset environment in which I then run SageMath. This way if some package gets updated on local machine, it is not updated in conda environment and running SageMath is still safe there. I followed SageMath guidelines to set it up in its own conda environment.

Obviously, there was some trivial toolkit as well. Some IDE, git and such. I am not going to talk about those.

4.1.2 Development workflow

SageMath development workflow was significantly improved at the start of 2023 by complete transfer from its own ticket tracking system to GitHub. Right now SageMath has its own public repository on GitHub and development workflow in SageMath is close to standard development workflow for public repositories on GitHub. Here are steps of this workflow:

1. Create or choose open issue(s) on SageMath repository
2. Fork SageMath repository
3. Make changes on your fork: clone, make changes locally, commit and push
4. Open pull request
5. Go through code review process and finish it successfully
6. Merge your changes into SageMath repository

4.1.3 Code structure

Most code of SageMath is written in Python and I chose to make all my changes in Python as well, so I am going to describe a common Python code structure in SageMath.

When writing Python code in SageMath, almost everything is done directly in the source files you are working on. This includes testing and documenting your code. When you are making changes for some class in Python, then for each method you need to have a Python docstring in some format. It should contain both tests and documentation of this method. The docstring is then passed through setup Sphinx [9] documentation generator when needed. So tests are started during pipelines and updated documentation in LaTeX is generated when merging changes into SageMath repository.

The only thing which is not done directly in code of methods in SageMath is dealing with bibliographical references. To make changes on bibliographical references you need to modify master bibliography file of SageMath and follow defined citation format. These citations can then be used in docstrings of methods for documentation purposes.

4.2 State of algorithms integration into SageMath

The algorithms which I am going to add to SageMath are:

- Algorithm computing Θ-defect in $\mathcal{O}(n)$ time using only Manacher's algorithm and tree traversal described in lemma 3.12
- Algorithm computing G-defect in $\mathcal{O}(n \cdot |G|)$ time described in lemma 3.19

For several technical reasons, I am not going to add any multi-threaded algorithms among my changes. This would require going deep into SageMath tooling for multi-threading and researching if this tooling is advanced enough to support my changes. On top of that I would require concurrent hash map with implementation details, while there is no built-in concurrent hash map in Python. Finally, because adding multi-threaded algorithms could be done as part of a separate work, it is simply not worth investing into as part of already proposed set of changes.

Currently I have created an issue number 35495 on SageMath main GitHub repository, then I forked the repository and started implementing the algorithms inside the fork.

I started my work on the fork by researching surrounding scope of changes I planned to make. This scope was mainly a part of code inside *WordMophism* class. Turned out there were quite a lot of algorithms with poor time complexity. There even was an already existing issue number 16366 for improving time complexity of some of these algorithms. So I marked all places which needed improvements and adjusted my estimation of amount of work I was going to do on my fork.

As *WordMophism* class was already implemented in Python and there was a big amount of code unrelated to my changes, I made a quick decision to make all my changes in Python. It would make a lot of sense to transfer the whole class to Cython in a separate issue, if there ever will be an actual need or desire for this.

After I concluded my researches, I implemented both fix of existing algorithms and addition of promised algorithms. During this implementation I was following development guidelines of SageMath.

Currently most of the work is done, but I need to reference this thesis in the SageMath code inside my changes. So it follows that I can add my changes to SageMath only once this thesis is finished and submitted.

Chapter 5

Conclusion

This thesis has contributed to the growing body of knowledge on the topic of algorithms dealing with generalized palindromes and generalized palindromic defects, and offers valuable insights for future researchers in this field.

There are still some open theoretical questions worth mentioning left at the time of writing this thesis. Here is a list of some of these questions:

1. Let's take the algorithm described in theorem 3.7, but instead of using jump-pointers just go m nodes up the T_w in $\mathcal{O}(m)$ time when getting mth ancestor of a node in T_w. This algorithm finds amount of pairwise distinct Θ-palindromic substrings of w. For sure it has time complexity $\mathcal{O}\left(n^2\right)$. Does this algorithm actually have $\Theta\left(n^2\right)$ time complexity or is its actual time complexity asymptotically faster than $\Theta\left(n^2\right)$? If it turned out that actual time complexity of this algorithm is $\mathcal{O}(n)$, then there would be a much more simple version of $\mathcal{O}(n)$ time algorithm computing Θ-defect compared to algorithms presented in this thesis.

2. What is the best achievable time complexity of a single-threaded algorithm computing G-defect? The fastest algorithm presented in this thesis has $\mathcal{O}(|w| \cdot |G|)$ time complexity.

3. What time complexities can be achieved for multi-threaded algorithms computing Θ-defect? Based on these algorithms, what promising multi-threaded algorithms computing G-defect can be presented and what time complexities will they have?

4. Can algorithms for computing Θ-defect be generalized for antimorphisms which are not letter permutations? Should definition of Θ-defect be modified for such antimorphisms compared to definition 2.1?

5. During the writing of the thesis we discovered an article by Rubinchik and Shur [10] which describes a rather sophisticated algorithm counting palindromes in a word in $\mathcal{O}(n \log n)$ time. As this article is very technical, we have not included any details from it. It is a natural question whether this approach can be generalized to count G-palindromes, or if it can be used to improve our approach.

Bibliography

1. PELANTOVA, Edita; STAROSTA, Štěpán. Palindromic richness for languages invariant under more symmetries. *Theoretical computer science*. 2014, vol. 518, pp. 42–63. ISBN 0304-3975.
2. THE SAGEMATH DEVELOPERS. *SageMath*. 2022. Version 9.5. Available from DOI: `10.5281/zenodo.593563`.
3. MANACHER, Glenn. A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *Journal of the ACM*. 1975, vol. 22, no. 3, pp. 346–351. ISBN 0004-5411.
4. BENDER, Michael A.; FARACH-COLTON, Martín. The Level Ancestor Problem simplified. *Theoretical computer science*. 2004, vol. 321, no. 1, pp. 5–12. ISBN 0304-3975.
5. ALSTRUP, Stephen; HOLM, Jacob. Improved Algorithms for Finding Level Ancestors in Dynamic Trees. In: ed. by MONTANARI, U.; ROLIM, JDP; WELZL, E. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1853, pp. 73–84. AUTOMATA LANGUAGES AND PROGRAMMING. ISBN 0302-9743.
6. DROUBAY, Xavier; JUSTIN, Jacques; PIRILLO, Giuseppe. Episturmian words and some constructions of de Luca and Rauzy. *Theoretical computer science*. 2001, vol. 255, no. 1, pp. 539–553. ISBN 0304-3975.
7. GARZON, Max H.; YAN, Hao. Watson-Crick Conjugate and Commutative Words. In: Germany: Springer Berlin / Heidelberg, 2008, vol. 4848. DNA Computing. ISBN 9783540779612;3540779612;
8. PELANTOVA, Edita; STAROSTA, Štěpán. Languages invariant under more symmetries: Overlapping factors versus palindromic richness. *Discrete mathematics*. 2013, vol. 313, no. 21, pp. 2432–2445. ISBN 0012-365X.
9. BRANDL, Georg. *Sphinx*. 2021. Available also from: `https://www.sphinx-doc.org/en/master/`.
10. RUBINCHIK, Mikhail; SHUR, Arseny M. Counting Palindromes in Substrings. In: FICI, Gabriele; SCIORTINO, Marinella; VENTURINI, Rossano (eds.). *String Processing and Information Retrieval*. Cham: Springer International Publishing, 2017, pp. 290–303. ISBN 978-3-319-67428-5.