Bachelor's Thesis:
Suffix Tree Based Palindrome Detection

Felix Heeger

04.08. - 28.09.2009

Freie Universität Berlin
Bachelor of Sience in Bioinformatics

Supervisors: Prof. Dr. Knut Reinert, Dr. Roland Krause
Advisor: David Weese

Contents

Declaration on Honour 1

1 Introduction 2

2 Methods 3

2.1 Definitions . . . 3
2.2 Gusfield-Algorithm . . . 4
2.2.1 Basic Idea . . . 4
2.2.2 Finding the LCP using the LCP-table . . . 4
2.2.3 Finding the LCP using a top-down iteration . . . 5
2.3 Bottom-Up-Algorithm . . . 5
2.4 Naive Algorithm . . . 6
2.5 Implementation . . . 7
2.5.1 SeqAn . . . 7
2.5.2 Enhanced Suffix Array . . . 7
2.5.3 Inverted Suffix Array . . . 8
2.5.4 Input and Output . . . 8
2.5.5 Gusfield-Algorithm with LCP-table . . . 9
2.5.6 Gusfield-Algorithm with Top-Down Iteration . . . 9
2.5.7 Bottom-Up Algorithm . . . 9
2.5.8 Naive Algorithm . . . 10
2.5.9 Palindrome Iterator . . . 10
2.6 Runtime and Memory Usage . . . 11
2.6.1 Dataset . . . 11
2.6.2 General Runtime Comparison . . . 11
2.6.3 The Parameters' Effect on the Runtime . . . 11
2.6.4 Runtime on Extreme Cases . . . 11
2.6.5 Memory Usage . . . 12

3 Results 12

3.1 Implementation . . . 12
3.2 Runtime and Memory Usage . . . 12
3.2.1 General Runtime Comparison . . . 12
3.2.2 The Parameters' Effect on the Runtime . . . 13
3.2.3 Runtime on Extreme Cases . . . 14
3.2.4 Memory Usage . . . 15

4 Discussion 17

4.1 Implementation . . . 17
4.2 Runtime . . . 17
4.2.1 Gusfield Algorithm with LCP-table . . . 17

4.2.2 Gusfield Algorithm with Top-Down Iteration 18
4.2.3 Bottom-Up Algorithm 18
4.2.4 Naive Algorithm 19
4.3 Memory Usage 19
4.4 Conclusion 20

5 Acknowledgement **20**

References **22**

Declaration on Honour

This is to solemnly declare that I have produced this work all by myself. Ideas taken directly or indirectly from other sources are marked as such.
This work has not been shown to any other board of examiners so far and has not been published yet.
I am fully aware of the legal consequences of making a false declaration.

1 Introduction

Palindromes in natural languages have fascinated men since ancient times and there are different opinions about what should be considered a palindrome. Some call a phrase a palindrome if its words are the same in forward and backward direction (figure 1 A). Other define a phrase a palindrome if it is the same forward and backward letter by letter, but ignore the spaces and punctuation (figure 1 B). There are also palindromic sentences with respect to spacing (figure 1 C). Words are called palindromes if they stay the same no matter if read forward or backward (figure 1 D).

A: King, are you glad you are king?
B: Was it a car or a cat I saw?
C: Step on no pets.
D: level

Figure 1: Different palindromes in English. A: Palindromic sentence word by word. B: Palindromic sentence letter by letter. C: Palindromic sentence letter by letter with respect to spacing. D: Palindromic word.

In amino acid strings and nucleic acid strings palindromes are defined in two different ways, considering their biochemical characteristics. A substring of a protein is called palindrome if its one-letter representation is the same read forward or backward, ignoring a certain number of letters in the middle. A part of a nucleic acid string is called palindrome if its is the same as its reverse complement, ignoring a certain number of bases in the middle.

In DNA and RNA palindromes have shown to have various functions. Some viruses use a palindrome which is forming a hairpin at the end of their genome as a primer for the DNA replication [16]. In eukaryotic RNA, a palindrome forming a hairpin seems to be often involved in terminating the transcription [5]. Palindromes in the DNA provide binding sites for different DNA-binding, homodimeric proteins [12] [4]. They can also induce insertion of a bigger palindrome into the DNA if a double-strand break occurs next to them. This may lead to gene amplification resulting in uncontrolled cell growth and thus, to a tumor [15].

In proteins the importance of palindromes is less clear. In some cases palindromes seem to have a role in the binding side of DNA binding [7] and metal binding [11] proteins. However, other studies suggest that palindromes in proteins don't influence their structure but seem to encourage the forming of α-helices [13].

So it is interesting to find palindromes in biological sequences *in silico*. Due to the different definitions of palindromes there are many possible algorithmic ways to find them.

For example for non-gapped palindromes in DNA a seed and extend algorithm can be used [10]. First all possible palindromes of a certain small length are generated and searched in the string of interest. Second these *seeds* are extended. This algorithm could also be used for gapped-palindromes if the method for searching the seeds in the string is able to search for gapped patterns, but it would not be adequate for bigger alphabets like amino acids because the number of seeds would be to big.

In other cases the naive algorithm may be sufficient.

An uncommon, but nevertheless interesting, way to identify palindromes and other sequence motifs in DNA was presented by Larionov et al. [9]. They illustrate a DNA sequence as a path in a plane in which the four bases of the DNA are represented as the four directions (up, down, left an right). In this path big palindromes can be found with the naked eye.

In this work I will present two different ways of finding maximal palindromes using suffix trees. The first is a implementation of a solution described in "Algorithms on strings, trees, and sequences" by Dan Gusfield [8]. The second is based on work about finding tandem repeats using suffix trees by Jens Stoye and Dan Gusfield [14]. The second way includes finding the *longest common prefix* of two strings. For this task two different methods was used.

All three approaches were implemented using the generic C++ library SeqAn [6]. It provides *enhanced suffix arrays* which can be used like suffix trees and have some very beneficial properties for the applied algorithms.

With the implemented algorithms, every string can be searched for gapped, ungapped and complemented palindromes if a adequate suffix array is provided.

For comparison purpose also a naive algorithm for searching palindromes was implemented.

In addition the SeqAn Bottom-Up iterator was specialized to a palindrome iterator.

2 Methods

2.1 Definitions

Definition 1 (The Reverse of a String)
For a string $S = s_0 s_1 \ldots s_{n-1} s_n$ the string $S^R = s_n s_{n-1} \ldots s_1 s_0$ will be called its **reverse**.

Definition 2 (The Reverse Complement of a Nucleic Acid String)
For a nucleic acid string $S = s_0 s_1 \ldots s_{n-1} s_n$ the String $\bar{S} = \bar{s}_0 \bar{s}_1 \ldots \bar{s}_{n-1} \bar{s}_n$ where $\bar{s}$ is the complementary base to s is called complement of S. The string $\bar{S}^R = \bar{s}_n \bar{s}_{n-1} \ldots \bar{s}_1 \bar{s}_0$ is called **reverse complement** of S.

Definition 3 (Palindromes)
A string consisting of three consecutive substrings A, G and B will be called a

palindrome with respect to gap length g if A is the reverse of B (and vice versa) and $|G| \leq g$. If $|G| > 0$ the palindrome will be called *gapped.* G will be called the *gap* of the palindrome. The length of a palindrome is defined as $|A| + |B|$.

Definition 4 (Palindromes in Nucleic Acid Strings)
In a nucleic acid sequence a string consisting of three consecutive substrings A, G and B will be defined as a **palindrome**, like in other strings except that A should be the reverse complement of B.

Definition 5 (Maximal Palindrome)
In a string $S = s_0 s_1 \ldots s_{n-1} s_n$ a substring $S_{i,j} = s_i s_{i+1} \ldots s_{j-1} s_j$ will be called a **maximal palindrome** if it is a palindrome which is not outward extendible. That means $S_{i-1,j+1} = s_{i-1} s_i s_{i+1} \ldots s_{j-1} s_j s_{j+1}$ is not a palindrome.

2.2 Gusfield-Algorithm

The first approach will be called Gusfield-Algorithm in this work, because it was described by Dan Gusfield in his book "Algorithms on strings, trees, and sequences" [8]

2.2.1 Basic Idea

The basic idea of the Gusfield-Algorithm is simple. If we want to find the maximal palindrome which is centred between a given position i and $i-1$ in a string S, we have to find the longest common prefix of the suffix of S starting at i and the reverse of the prefix of S ending at $i-1$. If we don't want to revert the prefix for every position in S, we can revert the whole string once to get S^R and instead of looking at the prefix of S ending at $i-1$, look at the suffix of S^R beginning at $|S| - i$.

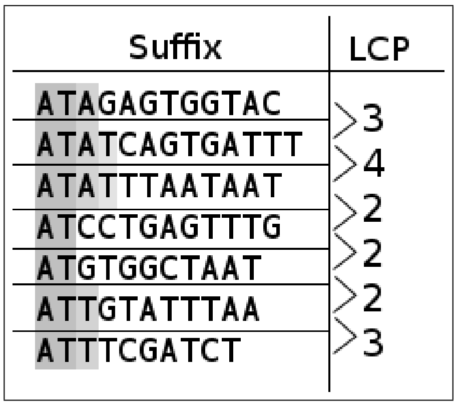
So far the Algorithm can only find ungapped palindromes. If we want to find a palindrome with gap length g we have to find the longest common prefix of the suffix starting at i and the prefix ending at $i-1-g$ or the longest common prefix of the suffix starting at i in S and the suffix starting at $|S| - i + g$ in S^R respectively.

If S and S^R are put into one *enhanced suffix array* (see section 2.5.2), this gives us two different possibilities to find the longest common prefix (LCP).

2.2.2 Finding the LCP using the LCP-table

The obvious way to find the LCP of two suffixes in an enhanced suffix array would be using the LCP-table. However, the LCP-table just gives us the length of the LCP of two consecutiv entries in the suffix array. To get the length of the LCP of two arbitrary entries in the suffix array, the minimum of all entries in the LCP-table for suffixes located between them in the suffix array must be found.

For example in figzre 2 the first suffix and the last suffix have the LCP `AT`, which has length two. Because of the lexicographical sorting of the suffix array the entries between them also start with `AT` so the LCP entries for them are at least two. Vice versa because the minimum of all LCP-table entries between the two suffixes is two, all of them must start with the same two characters. The minimum LCP-table entry between the first an third suffix is three. Their LCP `ATA` has length three.



Suffix	LCP
ATAGAGTGGTAC	
ATATCAGTGATTT	3
ATATTTAATAAT	4
ATCCTGAGTTTG	2
ATGTGGCTAAT	2
ATTGTATTTAA	2
ATTTCGATCT	3

Figure 2: Lexicographically sorted suffixes with LCP-table. The LCP of the first and last suffix is `AT` with length two. Two is also the minimum of all corresponding LCP-table entries

2.2.3 Finding the LCP using a top-down iteration

Another way to find the LCP of two suffixes a and b represented by two leafs in a suffix tree is to traverse trough the suffix tree from the root into the subtree containing the leafs representing a and b. At some knot a and b will be in different subtrees. The representation string of this knot is the LCP of the two suffixes.

For example in the word “lotto” whose suffix tree is shown in figure 3 the palindrome “otto” is centred between position 2 and 3. So we have to find the LCP of “to” and “tol”, which correspond to the leafs $(0, 3)$ and $(1, 2)$. At the root knot both are in the subtree beneath the knot representing “t” therfor we move down this edge. From there both are in the subtree of the knot corresponding to “to”. Beneath this knot they are in different subtrees. The LCP is “to”. Its length is 2.

2.3 Bottom-Up-Algorithm

The second approach uses a suffix tree of S and S^R, too. It visits every knot in the tree and searches for palindromes. Because it can - and will in this implementation - traverse the tree bottom-up it is called Bottom-Up-Algorithm in this work.

The algorithm is based on an idea that was originally developed to find *tandem repeats* i. e. direct consecutive repeats [14].

For every knot v in the suffix tree a leaf-list ($LL(v)$) is build, which contains all leafs which are part of the sub tree beneath v. Then the child w of v with the biggest leaf-list is found. For every leaf in $LL(v) \setminus LL(w)$ its *partner* is searched in $LL(v)$. The *partner* of a leaf corresponding to a suffix a in S is defined as the leaf corresponding to the suffix $b = |S| - a$ in S^R. If the partner u' of a leaf u is found, u and u' are in the same subtree above v. This means there is a palindrome whose second half starts at a in S. But to find out if it is a maximal palindrome, the next characters in S and S^R are compared. If they are different, the palindrome just found is maximal and can be reported.

For example in the suffix tree of the word "lotto" shown in figure 3 there are two palindromes "tt" and "otto". The first is not maximal. The palindromes will be dtected by finding their second half. So "tt" would be found at the knot representing "t". The leaf-list of this knot is $\{(0, 3), (1, 2), (0, 2), (1, 1)\}$. The partner for $(0, 3)$ is $(1, 2)$, because it is in the other string (not 0 but 1) and $b = |S| - a = 5 - 3 = 2$. It is part of the leaf-list so a palindrome is found, but we don't know if it is maximal yet. The next character in S is "o" at position 4 and the next in S^R is also "o" at position 3. They are not different, hence, the palindrome is not maximal.

The point of excluding $LL(w)$ from the search is that it is not necessary. If one of the leafs in $LL(w)$ would lead us to find a maximal palindrome, its partner is in $LL(v) \setminus LL(w)$ and we will find the corresponding palindrome this way. Therefore, for efficiency reason $LL(w)$ can be omitted.

In contrast to the algorithm of Stoye and Gusfield [14] the partner has not to be searched forward and backward, because in this case S and S^R are both in the suffix tree and a backward case in S will be a forward case in S^R, but we have to include the constraint that a partner of a suffix tree entry in S must be in S^R and vice versa.

To make this algorithm work for gapped palindromes with gap length g we just have to change the calculation of the suffix for the partner to $b = |S| - a + g$

2.4 Naive Algorithm

For every position i in S the naive algorithm starts to compare s_i and s_{i+1}. If they are equal, it compares s_{i-1} and s_{i+2} and so on until it finds two different characters. If there were enough equal characters before, it reports a palindrome. To find gapped palindromes with gap length g, it starts the comparison with s_i and s_{i+g+1} and continuous with s_{i-1} and s_{i+g+2} and so on.

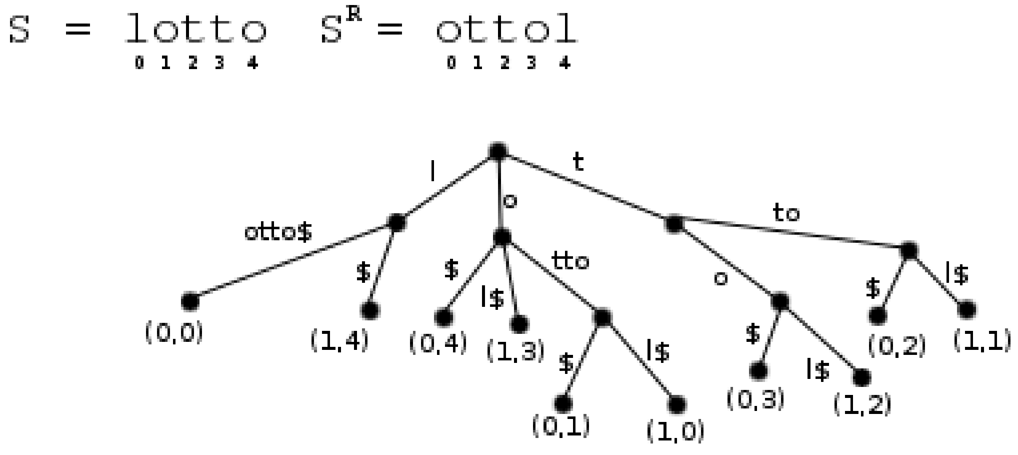


Figure 3: Suffix tree of the word "lotto" and its reverse. The leafs are denoted by a pair of numbers where the first is the sequence the suffix origins from (0 means S and 1 measn S^R) and the second is the start position of the suffix.

2.5 Implementation

2.5.1 SeqAn

All three ways to find palindromes using suffix trees plus the naive Algorithm was implemented in C++ using the generic library SeqAn [6]. SeqAn was developed for biological sequence analysis and contains different structures and functions which are useful for implementing the algorithms, for example a generic string class, different alphabet types for DNA and amino acids, functors to represent the complement or the reverse complement of a string with minimal additional space usage and suffix trees represented by enhanced suffix arrays.

2.5.2 Enhanced Suffix Array

Suffix trees are very useful structures for string processing. For many problems like finding repeats there are smart and easy to implement algorithms using a suffix tree. Suffix trees can be constructed in time linear to the length of the string but they consume a lot of memory. This is a problem espacially when dealing with long sequences like genomes. Suffix trees can be substituted by the more space efficient *enhanced suffix arrays* [3]. A suffix array is a list of all positions in the string sorted by the lexicographical order of the suffix starting at this position. However, a suffix array alone cannot do all the things a suffix tree can. To make it an enhanced suffix array which can replace the suffix tree in every algorithms other tables must be added.

In this work the LCP-table will be used explicit, other tables will be used implicit for traversing the tree. The LCP-table stores for every entry in the suffix array how long its longest common prefix with the next entry in the suffix array is.

2.5.3 Inverted Suffix Array

All three implemented algorithms need to get the position of a certain suffix in the suffix array by its start position in the text. The easiest way to do this is by an *inverted suffix array* which simply is an index of the suffix array. In the case of a suffix array which contains more than one string, the entries of the suffix array will be pairs. The first number denotes the string the suffix belongs to, the second one is the start position of the suffix in this string. Therefore, the inverted suffix array needs to have two dimensions, one for the string and the other for the suffix' start position in this string.

The enhanced suffix arrays of SeqAn do not contain such an inverted suffix array so it was implemented as an external structure. It needs an enhanced suffix array to be build and is designed to handle suffix arrays containing one or more strings.

2.5.4 Input and Output

Because the SeqAn string class is generic the algorithm could be implemented ignoring the alphabet. However the different definition of a palindrome in nucleic acid strings and other strings are a problem for this approach. This problem was solved by the fact that the only difference between searching for a palindrome or a complemented palindrome in the suffix tree based algorithms, is if the reverse or the reverse complement of the string is put into the suffix tree alongside the normal string. All three algorithms expect the same four parameters:

1. the enhanced suffix array containing S and S^R **or** S and $\bar{S}^R$
2. the minimum length the found palindromes should have
3. the maximal gap length that should be accepted in the palindromes
4. an out parameter for the list of found palindromes

So the user of the algorithm can define if he wants complemented or normal palindromes to be found by passing the right suffix array to the algorithm.

The found palindromes are stored in a string of palindrome structures which consist of position, length and gap length of the palindrome. With these informations the palindrome can be found in the original string.

2.5.5 Gusfield-Algorithm with LCP-table

The implementation of the Gusfield-Algorithm using the LCP-table for finding the LCP uses the fact, that the enhanced suffix array of SeqAn already has a LCP-table, which is not constructed by default, but can be easily requested. For every position i in the string S the position a of i in the suffix array is obtained via the inverted suffix array. Then for every gap length g from null to the given maximum the position b of the suffix starting at $|S| - i + g$ in S^R in the suffix array is identified. The minimum of all LCP-table entries between a and b is found by walking over the LCP-table from the smaller one of a and b to the bigger one. To increase the effectiveness of this search, it will be terminated if the minimum falls below the half of the given minimum palindrome length. If the found LCP length exceeds the half of the given minimum, an according palindrome structure is appended to the result string.

2.5.6 Gusfield-Algorithm with Top-Down Iteration

The implementation of the Gusfield-Algorithm using the top-down iteration to find the LCP obviously does not differ from the above one until the point where the LCP is searched. To find it, the iteration is started at the root knot. Starting from there the iterator is moved down the lexicographical smallest edge. It is tested if the leaf a corresponding to the current suffix is in the subtree beneath this knot. If so the subtree is searched for b. If none of a or b is in the subtree the iterator is moved to the next lexicographical bigger knot. If only a is in the subtree, the parent knot of the current one represents the LCP.

Checking if a certain suffix position u is in the subtree beneath a knot can be done in constant time, because the enhanced suffix array representing the suffix tree in SeqAn provides the opportunity to get the range of all positions in the suffix array of suffixes beneath a certain knot. If u is inside this range it is beneath the tree. The property that all suffix array entries representing leafs in a subtree are consecutive is intrinsic to the representation of a suffix tree by an enhanced suffix array.

2.5.7 Bottom-Up Algorithm

The Bottom-Up Algorithm uses a bottom-up iterator to traverse all knots of the suffix tree containing S and S^R. For every knot v first of all is tested if it is a leaf or if it is so close to the root of the tree, that its representative would be to short to match half of the given minimal palindrome length. If so, it is skipped. Otherwise the child w of v with the biggest leaf-list is searched. This is done by an iteration over all children of v. Determining of the length of the leaf-list of a certain knot can be done in constant time because it can be calculated by subtracting the start of the range of the knot from its end.

Then all suffix array positions in $LL(v) \setminus LL(w)$ are iterated by first counting from the start of $LL(v)$ to the start of $LL(w)$ and then from the end of $LL(w)$ to the end of $LL(v)$. This is done by two for-loops following each other. This is valid because w is located in the suffix tree beneath v which means $LL(w) \subseteq LL(v)$.

It was also tried to iterate over all children's leaf-lists except w. This method did not significantly change the runtime.

For every suffix array position a in $LL(v) \setminus LL(w)$ the suffix array position b of the partner corresponding to the leaf represented by a is determined with the help of the inverted suffix array. Checking if b is in $LL(v)$ can be done in constant time, because of the property of the enhanced suffix array mentioned above.

The characters after the so far found palindrome in S and before the so far found palindrome in S^R are compared. If they are different, a maximal palindrome is reported. The position of the palindrome must be computed with respect to the fact if a was in S or in S^R.

2.5.8 Naive Algorithm

In contrast to the algorithms described above the naive algorithm works directly on the string rather than on a suffix tree. For this reason there are two different implementations. One for finding normal palindromes and one for finding complemented palindromes in nucleic acid strings. However, if the function `findPalinNaive` is used for a nucleic acid string, the internal function `findComplPalinNaive`, which searches for complemented palindromes, is called.

2.5.9 Palindrome Iterator

The palindrome iterator is a specialization of the SeqAn Bottom-Up Iterator. It uses the same approach as the Bottom-Up Algorithm described above. The main difference to the normal algorithm is he fact that the iterator needs to store all information about which entries of the leaf-list have already been checked with which gap length and so. This leads to the Problem that the two for-loops following each other used in the Bottom-Up Algorithm were not usable any more. For this reason the already mentioned alternative of iterating over all children's leaf-lists was applied.

All other loops had to be "turned upside down", which means first search four a palindrome with the already set parameters and after that if none was found increase the parameters.

In the constructor the parameters have to be set for the starting knot. After that the first palindrome is searched. Searching the next palindrome may mean search the next palindrome which can be found at this knot or, if there is no next palindrome at this knot, find the next knot representing the second half of a palindrome.

Different functions were implemented to get the position, length and gap length of the palindrome the iterator is currently pointing at. Another function returns a palindrome structure representing the last found palindrome. When computing the properties of the palindrome, it has to be considered, that the parameter for the tested gap length was incremented after the palindrome was found.

2.6 Runtime and Memory Usage

2.6.1 Dataset

For the runtime tests on DNA the sequence of the chromosome 21 of the human reference genome released at March 2, 2009 by the Genome Reference Consortium was used. It was downloaded from the NCBI ftp server [2]. The individual contigs of the chromosome were concatenated to a single sequence. This will obviously produce biological wrong results, but for testing the runtime the data is sufficient.

For tests on natural language the English text part of the Pizza&Chilli Corpus was downloaded [1]. It contains English texts of the Gutenberg Project from May 4, 2005. For comparability reason only the first 35 MB of the text were used, because the size of the sequence of chromosome 21 is 35 MB.

2.6.2 General Runtime Comparison

All test runs were done one Intel Xeon Processors with 3.2 GHz.

First of all the runtime of all four algorithms was tested with default parameters (minimal palindrome length: 4, maximal gap length: 2) on the datasets to find out which algorithm is the fastest on average data.

They were also tested on smaller subsets of the chromosome 21 sequence to find out which influence the size of the input data has to the runtime of the different algorithms. The subset where the first 50, 100 and 500 thousand and the first 1, 2, 5, 10 and 20 million characters of the sequence.

2.6.3 The Parameters' Effect on the Runtime

Second the effect of different parameter combinations was tested for the four algorithms. They were run on the first million characters of the chromosome 21 with all possible combinations of minimal palindrome length from 4 to 30 in steps of size two and maximal gap length from 0 to 8.

2.6.4 Runtime on Extreme Cases

Third the effect of extreme data on the four algorithms was tested. For this test small datasets were used, because it was likely that the normal dataset would lead to runtimes to long to wait for them. The three following sequences were tested:

- a random palindrome of length 50,000 on an alphabet containing all letters and several special characters like space, newline, exclamation mark and question mark.
- a random DNA palindrome of length 50,000
- a 25,000 repeat of "AT", which is a ungapped complemented palindrome of length 50,000 itself and also contains a complemented palindrome stretching to the nearer end of the sequence at every position. Each of these palindromes contains gapped palindromes with even gap lengths.

All four algorithms were run on these strings with default parameters. The results were compared to runtimes of the algorithms on the 50,000 characters long random datasets of the according alphabet.

2.6.5 Memory Usage

The memory usage of every algorithm was measured for the different sized subsets of the chromosome 21 mentioned in section 2.6.2.

3 Results

3.1 Implementation

The header files including the implementations of all four algorithms and the inverted suffix array along with the example programs for finding palindromes in DNA and natural texts and for the use of the palindrome iterator can be downloaded at: http://page.mi.fu-berlin.de/fheeger/bsc_code.tar.gz

A maximal palindrome of length $l > 2$ and gap length g always contains $\frac{l-(g+2)}{2}$ even length gapped palindromes with lengths $l-2, l-4, \ldots, 4, 2$ and gap lengths $g+2, g+4\ldots, g+l-2$. For example the palindrome `abccba` contains the gapped palindrome `ab--ba`.

The implemented algorithm will find and report all of these $s$ubpalindromes.

3.2 Runtime and Memory Usage

3.2.1 General Runtime Comparison

On the 35 MB English text with default parameters the naive algorithm took 3.023 seconds. The Bottom-Up Algorithm was noticeable slower. It took 261 seconds. The Gusfield Algorithm using the top-down iteration took about twice the time (469 seconds). The Gusfield Algorithm using the LCP-table was canceled after 11 hours.

In the test on the sequence of chromosome 21 the naive algorithm was slower than on the English text. It needed 4.760 seconds to finish. In contrast to the run on the English text the Bottom-Up Algorithm was slower than the Gusfield using the top-down iteration. It took 310 seconds while the Gusfield Top-Down Algorithm only took 233 seconds. The Gusfield Algorithm using the LCP-table again was canceled after running 11 hours without terminating.

The effect of the input data's size can be seen in figure 4. Beside the test runs done especially for this analysis, the runs on the complete chromosome 21 mentioned above, were compared. The runtime of the Gusfield Algorithm using the LCP-table increases exponential with the data size. The runs for 10, 20 and 35 MB was canceled, because they did not finish for 10 hours. The other three algorithms' runtimes increases linear with the input data's size.

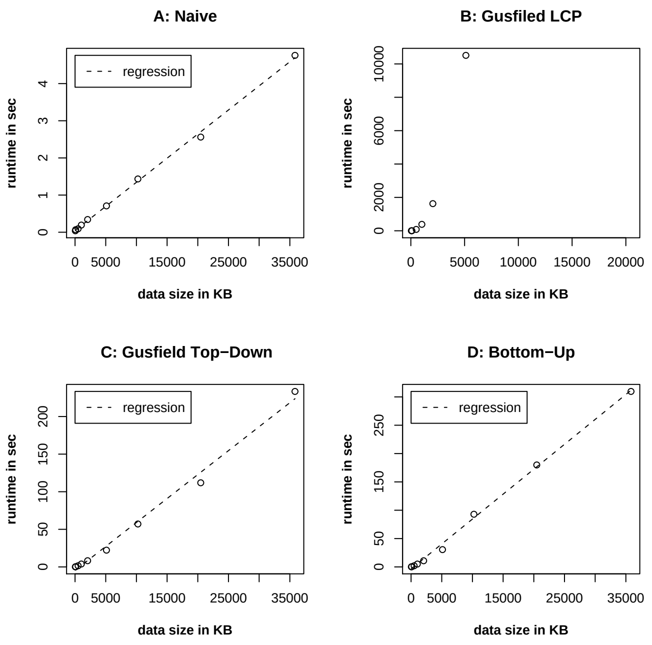


Figure 4: Effect of the input data size (in KB) to the runtime (in sec) for DNA data.

3.2.2 The Parameters' Effect on the Runtime

The choice of the parameter greatly effects the runtime of the suffix tree based algorithms. Its effect on the runtime of the naive algorithm is far less considerable.

The runtime of the naive algorithm increases slightly for greater gap lengths (figure 6 A), but is not effected by the minimal palindrome length (figure 5 A).

The Gusfield Algorithm using the LCP-table to find the LCP is highly dependent on both parameters. Small palindrome lengths result in in very long runtimes which fall rapidly for greater lengths (figure 5 B). The runtime increases linear with the maximal gap length (figure 6 B).

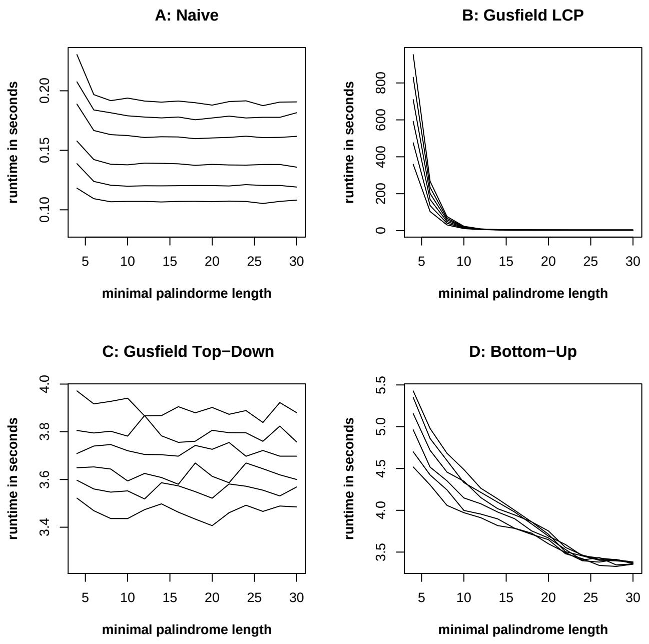


Figure 5: Effect of different palindromes lengths to the runtime (in seconds) of all four implemented algorithms. Note that the maximal gap length also was varied. The data points of one line have the same gap length.

The Top-Down approach makes the Gusfield Algorithm's runtime independent of the maximal palindrome length (figure 5 C) but it still increases linear with the gap length (figure 6 C).

The Bottom-Up Algorithm's runtime falls rapidly if the minimal palindrome length is increased (figure 5 D). Above a minimal palindrome length of 24 it seems to almost stay the same. The runtime also increases linear with the maximal gap length (figure 6 D).

3.2.3 Runtime on Extreme Cases

All algorithms run faster on the random palindrome than on an equal length random string (table 1). On the DNA palindrome all algorithms except the Gusfield Algorithm using the LCP-table are faster than on the equal length DNA sequence.

The `AT` repeat significantly slows the naive algorithm and both Gusfield Algorithms down. In contrast the Bottom-Up Algorithm is even faster than on the

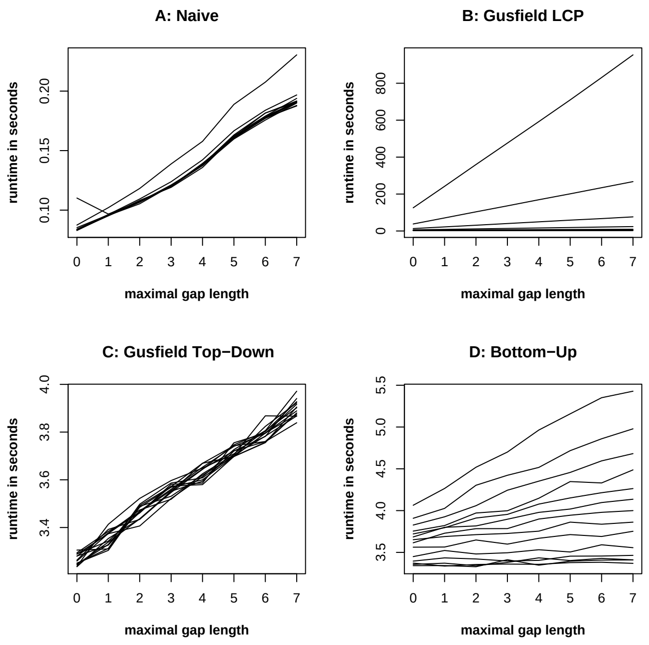


Figure 6: Effect of different gap lengths to the runtime (in seconds) of all four implemented algorithms. Note that the minimal palindrome length also was varied. The data points of one line have the same palindrome length.

random DNA sequence. The Gusfield Algorithm using the top-down iteration is by far the slowest on this dataset.

	naive	Gusfield LCP	Gusfield Top-Down	Bottom-Up
random DNA	0.029	1.469	0.202	0.225
DNA palindrome	0.008	1.694	0.135	0.151
AT repeat	4.466	10.700	67.484	0.181
random “natural” alphabet	0.0122	0.224	0.285	0.226
“natural” palindrome	0.009	0.161	0.194	0.147

Table 1: Runtime (in seconds) of all four algorithms on “normal” and extreme datasets of size 50 KB

3.2.4 Memory Usage

The results of the memory usage analysis can be seen on table 2. It increases linear with the size of the input data for all four algorithms and is almost equal for both Gusfield Algorithms and the Bottom-Up approach. For 1 MB input data size they use about 73 MB of RAM. The naive algorithm uses far less memory (about 6.5 MB per 1 MB input data). The measurements for 1 and 2 MB input data were omitted for the Gusfield LCP, because of its extreme runtime.

	naive	Gusfield LCP	Gusfield Top-Down	Bottom-Up
50 KB	376,546	3,549,700	3,549,700	3,549,572
100 KB	602,328	7,103,344	7,103,344	7,103,216
500 KB	3,039,738	35,518,018	35,518,018	35,517,890
1 MB	6,557,767	NA	73,285,403	73,285,275
2 MB	13,733,829	NA	145,936,747	145,936,619

Table 2: Peak memory usage (in Byte) of all four algorithms with different sized input data. Measurement for 1 and 2 MB with Gusfield LCP have been skipped because of the extreme runtime of this algorithm.

4 Discussion

4.1 Implementation

The reporting of the $s$ubpalindromes described in section 3.1 may be the subject of criticism. If they are maximal is arguable. At least they are not outward extendible and so they are maximal according to the definition used in this work (see definition 5). But most importantly excluding them is very difficult, because of the way the implemented algorithms work. For a given position they do not extend the gap symmetrically which would include moving the start position of the second half of the palindrome. Thus for a given position the actual middle and start position of the palindrome are different for different gap lengths. This leads to the problem that the subpalindromes described above will not be found in one position iteration of the algorithm and so the question if there was already found a palindrome with the same start position and length but with a smaller gap can not be answered without searching the result list. Doing this on the fly would be very inefficient. However, after the algorithm finished the result list can be sorted by position, length and gap length. After this the subpalindromes can be filtered easily.

4.2 Runtime

4.2.1 Gusfield Algorithm with LCP-table

The test runs on chromosome 21 and the English text show, that the method of finding the LCP with the LCP-table does lead to not acceptable runtimes on large data, especially when compared to the runtimes of the other version of the Gusfield Algorithm. The iteration over the LCP table for every pair of positions i in S and $|S| - i$ in S^R will take $\mathcal{O}(n^2)$ steps in worst case and if not terminated because the minimum LCP falls below the given minimal palindrome length. The average case's runtime seems to be near the one of the worst case, because the runtime increases exponential with the input data size as can be seen in figure 4 B.

Introducing the termination of the run over the LCP-table if the found minimum falls below the given minimal palindrome length, obviously has no great effect if the given minimal length is small (figure 5 B). If a really big minimal palindrome is requested the iteration could likely be terminated after a short while. But this is mainly the case if very few or no palindromes are found and thus not very useful.

A small alphabet also makes it more likely that the iteration over the LCP-table will not be terminated early, because many suffixes will have the same start characters by chance. In the test for extreme datasets the algorithm showed to be slower on random data with an alphabet of size four (DNA) than on random data with a bigger alphabet (see table 1).

For bigger accepted gap lengths the runtime of the Gusfield LCP increases linear. This is because the run over the LCP-table has to be repeated for every possible gap length.

The test on the `AT` repeat shows that the Gusfield Algorithm using the LCP-table also is very vulnerable to this kind of extreme data. Every single suffix of this string is by itself a repeat of `AT` or `TA`. So there are two groups of suffixes in which the individual entries differ only in their length. For this reason the early termination of the iteration over the LCP-table is very unlikely.

4.2.2 Gusfield Algorithm with Top-Down Iteration

The top-down iteration is the much faster way to implement the Gusfield Algorithm. It leads to acceptable runtimes on the chromosome 21 and on the English text. It is twofold slower on the English text than on the chromosome 21. This may be due to the fact that a bigger alphabet leads to more different children at one knot in the suffix tree. So the algorithm has to iterate over more children to find the right subtree to proceed with.

The runtime increases linear with the input data's size (figure 4 C), because for every position in the string the top-down iteration has to be executed. Compared to this the effect to the time for one iteration seems to be small enough to not be noticed.

The given minimal palindrome length has no effect to the runtime, because the algorithm will finish the iteration and afterwards check if the found palindrome is long enough. So a bigger minimal length does not lead to earlier termination like in the case of the LCP-table iteration. The given maximal gap length effects the runtime of the algorithm linearly, because the top-down iteration has to be repeated for the different gap lengths.

The Gusfield Algorithm with top-down iteration is massively slowed down by the `AT` repeat. The repeat sequence leads to a very "deep" tree i. e. the distance from the root to the leafs is very long. This slows the algorithm down because it has to go down a long way.

4.2.3 Bottom-Up Algorithm

The Bottom-Up Algorithm is faster than the Gusfield Algorithm with top-down iteration on the English text, but is slower on the DNA sequence. This is mostly because the Gusfield algorithm was faster on the DNA data but also because the Bottom-Up Algorithm is slightly slower on DNA. This may be because a bigger alphabet leads to less internal knots in the suffix tree because more child knots can have the same parent knot. The Bottom-Up Algorithm iterates over all internal knots. So fewer knots lead to a shorter runtime for the same data size.

If the inputs data's size is increased, the runtime of the algorithm increases linearly. It seems the number of internal knots increases linear with the data size and with number of knots the runtime of the algorithm increases.

Like the Gusfield Algorithms, the Bottom-Up Algorithm's runtime increases linear with given maximal gap length. This is because at every knot for every gap length has to be tested whether there is a palindrome with thes parameters. So doubling the maximal gap length means doubling the number of tests for every knot.

For longer minimal palindrome lengths the Bottom-Up Algorithm's runtime decreases. The algorithm skips knots which are so close to the root, that their representative is too small to be the second half of a long enough palindrome. If the requested minimal palindrome length is bigger, more knots can be skipped. For small minimal lengths the decrease in runtime is bigger because many knots are near to the root in a suffix tree. For bigger minimal length the effect of increasing the length is smaller because less knots are deep in the tree and therefore few more will be skipped.

In contrast to the other algorithms the Bottom-Up Algorithm is not slowed down by the `AT` repeat. It even is faster than on identical sized random DNA data. The repeat sequence contains many similar suffixes. this leads to fewer internal knots in the suffix tree and speeds up the Bottom-Up Algorithm.

4.2.4 Naive Algorithm

The naive algorithm is much faster than the suffix based approaches. It is slower on the DNA sequence than on the equal long English text. The bigger alphabet may lead to less small palindromes occuring by chance. This speeds up the naive algorithm because, when there is no palindrome at a certain position, this means only one comparison has to be made per accepted gap length.

The runtime of the naive algorithm increases linear with the input data's size, because the algorithm has to check for every text position if there is a palindrome.

For bigger accepted gap size the runtime increases linear. This is due to the fact that the check for a palindrome has to be done for every possible gap length.

A bigger minimal palindrome length does not effect the runtime of the algorithm, because the check if the palindrome is long enough to be reported is performed after the palindrome is found. This means palindromes not exceeding the given minimal length are also found, but not reported.

The `AT` repeat slows the naive algorithm significantly down. At every position in this string there is a complemented palindrome which reaches to the nearer end of the sequence. Thus the algorithm has to do many comparisons at every position.

4.3 Memory Usage

Longer input sequences lead to linear increasing memory usage for all four algorithms. In case of the naive algorithm the memory usage is mainly influenced by the size of the input data itself. For the suffix based approaches the enhanced suffix array and inverted suffix array have to be stored in addition. Their size is also linear to that of the input string.

Because it needs no enhanced suffix array and inverse suffix array the naive algorithm uses much less memory.

The memory usage of the three suffix based algorithms is very similar, because its manly influenced by the suffix arrays, which are obvious of equal size in every algorithm.

4.4 Conclusion

The naive algorithm is faster than the suffix based approaches for every parameter combination and also uses less memory. Only on the `AT` repeat sequence the Bottom-Up Algorithm is faster.

The naive algorithm has a long worst case runtime. If a sequence contains many palindromes (like the `AT` repeat), the naive algorithm has to do very many comparisons for every text position. In this case the Bottom-Up algorithm whose runtime does not depend on the found palindrome length is faster. But because an average English text or DNA sequence does not contain many palindromes the naive algorithm is faster on them.

The suffix based methods theoretical worst case runtime may be smaller than that of the naive algorithm, but in praxis the time to build the indices and the overhead of handling them slows them down too much. So if a “normal“ sequence without a lot of palindromes is expected the naive algorithm is the better choice after all.

5 Acknowledgement

I want to thank Prof. Knut Reinert and Dr. Roland Krause who made it possible for me to write this work. Big thanks also to David Weese for his help and advise.

I apologize to the people who had to suffer from my bad mood during the last weeks. Thank you for your patients and your support.

References

[1] English texts of the Pizza&Chilli Corpus. http://pizzachili.dcc.uchile.cl/texts/nlang/.

[2] human chromosome 21. ftp.ncbi.nih.gov/genomes/H_sapiens/CHR_21/.

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.

[4] David Baillat, Agnès Bègue, Dominique Stéhelin, and Marc Aumercier. Ets-1 transcription factor binds cooperatively to the palindromic head to head ets-binding sites of the stromelysin-1 promoter by counteracting autoinhibition. *J Biol Chem*, 277(33):29386–29398, Aug 2002.

[5] W. M. Chu, R. E. Ballard, and C. W. Schmid. Palindromic sequences preceding the terminator increase polymerase III template activity. *Nucleic Acids Res*, 25(11):2077–2082, Jun 1997.

[6] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9:11, 2008.

[7] Malgorzata Giel-Pietraszuk, Marcin Hoffmann, Sylwia Dolecka, Jacek Rychlewski, and Jan Barciszewski. Palindromes in proteins. *J Protein Chem*, 22(2):109–113, Feb 2003.

[8] Dan Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge Univ. Press, 1997.

[9] Sergei Larionov, Alexander Loskutov, and Eugeny Ryadchenko. Chromosome evolution with naked eye: palindromic context of the life origin. *Chaos*, 18(1):013105, Mar 2008.

[10] Le Lu, Hui Jia, Peter Dröge, and Jinming Li. The human genome-wide distribution of dna palindromes. *Funct Integr Genomics*, 7(3):221–227, Jul 2007.

[11] P. K. Pan, Z. F. Zheng, P. C. Lyu, and P. C. Huang. Why reversing the sequence of the alpha domain of human metallothionein-2 does not change its metal-binding and folding characteristics. *Eur J Biochem*, 266(1):33–39, Nov 1999.

[12] Sean P Riley, Tomasz Bykowski, Anne E Cooley, Logan H Burns, Kelly Babb, Catherine A Brissette, Amy Bowman, Matthew Rotondi, M. Clarke Miller, Edward DeMoll, Kap Lim, Michael G Fried, and Brian Stevenson. Borrelia

burgdorferi ebfc defines a newly-identified, widespread family of bacterial dna-binding proteins. *Nucleic Acids Res*, 37(6):1973–1983, Apr 2009.

[13] Armita Sheari, Mehdi Kargar, Ali Katanforoush, Shahriar Arab, Mehdi Sadeghi, Hamid Pezeshk, Changiz Eslahchi, and Sayed-Amir Marashi. A tale of two symmetrical tails: structural and functional characteristics of palindromes in proteins. *BMC Bioinformatics*, 9:274, 2008.

[14] Jens Stoye and Dan Gusfield. *Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, chapter Simple and flexible detection of contiguous repeats using a suffix tree Preliminary Version, pages 140–152. Springer Berlin / Heidelberg, 1998.

[15] Hisashi Tanaka, Stephen J Tapscott, Barbara J Trask, and Meng-Chao Yao. Short inverted repeats initiate gene amplification through the formation of a large dna palindrome in mammalian cells. *Proc Natl Acad Sci U S A*, 99(13):8772–8777, Jun 2002.

[16] K. Willwand, E. Mumtsidu, G. Kuntz-Simon, and J. Rommelaere. Initiation of dna replication at palindromic telomeres is mediated by a duplex-to-hairpin transition induced by the minute virus of mice nonstructural protein ns1. *J Biol Chem*, 273(2):1165–1174, Jan 1998.