CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Short Tandem String Duplication Sequences

Master's Thesis in Computer Science - Algorithms, Languages and Logic

BELMIN DERVISEVIC
MATEO RASPUDIC

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Short Tandem String Duplication Sequences

BELMIN DERVISEVIC
MATEO RASPUDIC

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Short Tandem String Duplication Sequences

BELMIN DERVISEVIC
MATEO RASPUDIC

Supervisor: Peter Damaschke, Department of Computer Science and Engineering
Examiner: Thierry Coquand, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Short Tandem String Duplication Sequences

BELMIN DERVISEVIC
MATEO RASPUDIC
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

It has been shown that tandem repeats are involved in several neurological and genetic disorders. Additionally, tandem repeats and the Tandem Duplication (TD) operation have been studied in other areas, such as information theory and formal languages. Given a string `AXB`, where `A`, `X`, and `B` denote substrings, a tandem duplication operation on the substring `X` copies it and places its copy adjacently, which produces the string `AXXB`.

Contemporary work shows that the Tandem Duplication Distance Problem (TDDP) is NP-complete for alphabets of size five. Therefore, we study the idea of overlapping squares and introduce a new algorithm, which solves the k-TD problem in time $\mathcal{O}((n \log n)^k)$, where n denotes the length of the target string T, and k denotes the number of contraction. Primarily, we prove that the Tandem Duplication Distance Problem is fixed-parameter tractable in the natural parameter d. We introduce a dynamic programming algorithm that locally exhausts independent disjoint substrings and calculates the optimal edit distance through minimization. The algorithm has a running time of $\mathcal{O}(nd^3(d \log d)^d)$, where n denotes the length of the target string T and d denotes the length difference of the two strings S and T.

Keywords: algorithms, tandem duplication, squares, tandem repeats, edit distance, dynamic programming, FPT algorithms.

Acknowledgements

First and foremost, we would like to direct a special acknowledgment to our supervisor, Professor Peter Damaschke, for his unwavering support and guidance throughout the project. This project would not have been possible without his input, valuable feedback, and the stimulating discussions we have had. We would also like to express our gratitude to Professor Thierry Coquand, who has shown interest in the project and taken the time to be our examiner.

Belmin Dervisevic & Mateo Raspudic, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1
Introduction

Research in fields like computational and molecular biology has shown that repeating patterns of nucleotide in the DNA, known as tandem repeats, are often involved in neurological and genetic disorders [1]. Huntington's disease is an example of a neurological disorder associated with tandem repeats [2]. Furthermore, gene duplication is an essential mechanism in not only human evolution but also in other organisms. Consequently, studying such patterns using algorithms can help better understand DNA structure.

DNA structures can be thought of as strings, where each symbol in the string corresponds to one nucleotide. Therefore, mutations in DNA are analogous to operations on a string. For example, a tandem duplication repeats some substring in a given string. In the string `AXAXBC`, the sequence `AX` is repeated twice. Therefore, `AXAXBC` can be generated from `AXBC` by duplicating the substring `AX` once.

1.1 Background

Section 1.1.1 formally introduces the tandem duplication operation and relevant subproblems presented in [3]. Thereafter, Section 1.1.2 introduces related work that inspired this project.

1.1.1 Problem definitions

A tandem duplication (TD) on a string S is a transformative operation that replaces some substring A in S with its square AA. We need to know whether a string S can be transformed into another string T using tandem duplication. Given two strings S and T, we write $S \Rightarrow T$ if a sequence of transformations exists from S to T using tandem duplication. If no such transformation exists, we write $S \not\Rightarrow T$.

Example 1. Let $S =$ `AXB` where `A`, `X`, and `B` in turn are strings. Tandem duplication of `X` transforms the string S to $S' =$ `AXXB`, and $S \Rightarrow S'$.

Problem definition 1 (TD-Exist). Given two strings S and T over the same alphabet Σ, where Σ is a fixed finite set, if it is possible to transform S to T, we write $S \Rightarrow T$.

Tandem Duplication Existence (TD-Exist)

Input: Strings S and T over the same alphabet Σ.

Question: $S \Rightarrow T$?

Example 2. Let S and T be strings containing the substrings `A`, `X`, `B` such that $S =$ `AXB` and $T =$ `AXBAX`. Since the second instance of `AX` in T is located after the substring `B`, it is impossible to transform S to T, i.e., $S \not\Rightarrow T$.

Definition 1. Given two strings S and T such that $S \Rightarrow T$, the minimum number of duplications from S to T is defined as $dist_{TD}(S,T)$. If $S \not\Rightarrow T$, then $dist_{TD}(S,T) = \infty$.

Example 3. Let S and T be strings containing the substrings `A`, `X`, `B` such that $S =$ `AXB` and $T =$ `AAXBXB`. The sequence of duplications can then be expressed as `AXB` $\Rightarrow_1$ `AAXB` $\Rightarrow_2$ `AAXBXB`, where $\Rightarrow_i$ denotes the i:th tandem duplication. It follows that $S \Rightarrow T$, and $dist_{TD}(S,T) = 2$.

Problem definition 2 (k-TD). Given two strings S and T over the same alphabet Σ, and a positive integer k, we check if $dist_{TD}(S,T) \leq k$.

Tandem Duplication Edit Distance (k-TD)

Input: Strings S and T over the same alphabet Σ and an integer k.

Question: Is $dist_{TD}(S,T) \leq k$?

1.1.2 Related work

The issue of whether the edit distance problem can be solved in polynomial time was originally presented by Leupold et al. in 2004 [4]; it was not until 2019 that Lafond et al. proved that the problem is, in fact, NP-hard [5]. The main point of reference for our project is a recent paper [3] by Cicalese and Pilati, published in 2021, where they prove that the TD-Dist problem is NP-complete for alphabets of 5 symbols. Furthermore, Cicalese and Pilati present a linear-time algorithm for the TD-Exist problem for a special class of strings, namely *purely alternating* strings (PA), with an alphabet size of 5 symbols. To explain, a string S is purely alternating if there is a fixed cyclic order of the symbols of the alphabet such that every symbol in S is followed by the same symbol or the next symbol in that cyclic order. For example, the string 0112012 is purely alternating, whereas 011212 is not.

A complete overview of the known complexity classes for each subproblem can be viewed in Table 1.1.

Table 1.1: Complexity classes for the problems introduced in Section 1.1.1.

	$\|\Sigma\| = 2$	$\|\Sigma\| = 3$	$\|\Sigma\| = 4$	$\|\Sigma\| = 5$
TD-Exist	P	?	?	?
TD-Exist (PA)	P	P	P	P
TD-Dist	?	?	?	NP-complete
k-TD	?	?	?	?

Note: Open problems are marked "?".

1.2 Notations

The notations introduced in this section build on [3, 5] and are used consistently. Let Σ be a fixed finite set of symbols, also known as an alphabet. Let S be a string, where $|S|$ denotes the length of string S and $S[i]$ denotes the ith symbol in S and $S[i, j] = S[i]S[i + 1]..S[j]$ denotes a substring in S from position i to j. A string XX consisting of two identical substrings X is called a *square*. If no squares exist in X we call XX a *primitive* square. Also, if the string S is a square-free string, we say that S is a *root*. Moreover, for a given problem instance (S, T), d denotes the length difference between the two strings, i.e., $d = |T| - |S|$. Note that it is always the case that $|S| \leq |T|$ for any given problem instance.

Furthermore, let $dist_{TD}(S, T)$ denote the minimum number of tandem duplications needed to transform S to T. The positive integer k denotes the maximum number of duplications allowed from S to T. If there exists a sequence of tandem duplication from S to T, we write $S \Rightarrow T$. If no such sequence exists, we write $S \not\Rightarrow T$ and $dist_{TD}(S, T) = \infty$. We call this the *Tandem Duplication Distance Problem* (TDDP).

Removing one of the occurrences of X in a square XX is an operation called a *contraction*. If there exists a sequence of tandem contractions from T to S we write $T \mapsto S$. If no such sequence exists, we write $T \not\mapsto S$. Note that the minimum number of tandem contractions such that $T \mapsto S$ is the same as the minimum number of tandem duplications such that $S \Rightarrow T$.

Finally, fixed-parameter tractability is investigated in this thesis, see Section 2.1.

1.3 Problem

We pose the following research questions:

1. For an alphabet of size q, where $q \geq 2$, can we develop an algorithm for k-TD with fixed k, as large as possible?

2. Are we able to produce a fixed-parameter tractable (FPT) algorithm for k-TD with the parameter d?

3. If the development of the FPT algorithm fails, can we identify the difficulties and gain a specific understanding of them?

1.4 Contribution

The contribution of the project is twofold. First, an algorithm that solves the k-TD problem in time $\mathcal{O}((n \log n)^k)$ is presented, where n denotes the length of the target string T and k the maximum number of contractions. The algorithm's run-time is achieved by proving that same-length overlapping squares yield the same contraction result. Thus only disjoint substrings of the same length must be considered for contraction. Second, a dynamic programming algorithm, which evaluates disjoint substrings, is presented together with correctness proofs and run-time analysis. In particular, we show that TDDP is FPT in the parameter $d = |T| - |S|$. Finally, we also introduce structural properties for overlapping squares, which are not utilized by the algorithms. However, these properties could be used to further reduce the branching number of TDDP algorithms.

1.5 Delimitations

This project will only cover theoretical aspects of tandem string duplication. We will therefore disregard applications of algorithms derived from the project. Also, as it has been proven that all tandem repeats can be identified in linear time [6], we have decided not to investigate tandem repeat identification further. Finally, we will only consider short edit sequences for finite strings.

1.6 Ethical considerations

Basic research and design of algorithms have no ethical issues, generally. Instead, the application of algorithms can be of more interest. In the future, one such application could be in active gene editing. An ethical dilemma, in that case, is whether the benefits outweigh the risks or not. According to a study by Baylis et al. in 2020, where 106 countries were surveyed, no country was documented to permit heritable human genome editing explicitly [7]. However, some countries prohibit gene editing with exceptions. An example of that is Finland, where gene editing is allowed to cure or prevent a serious hereditary disease [7].

Furthermore, it is essential to distinguish between editing genes in an embryo to prevent a baby from being born with sickle cell anemia and editing genes to alter the appearance of future generations. To conclude, if gene editing could be used to prevent severe diseases and suffering, it could be worth researching further.

2
Preliminaries

This chapter introduces the fundamental ideas of combinatorial algorithms and an overview of combinatorial issues related to strings.

2.1 Fixed-parameter tractability

A parameterized problem is a language $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet and $\mathbb{N}$ is the parameter of the problem. A problem is *fixed-parameter tractable* (FPT) if $(x, k) \in \mathcal{L}$ and it can be decided in running time $f(k) \cdot |x|^{\mathcal{O}(1)}$, where k is the parameter and f is an arbitrary function depending on k. An algorithm $\mathcal{A}$ is a parameterized algorithm for $\mathcal{L}$ if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ and a constant c such that for every $(x, k) \in \mathcal{L}$, the algorithm $\mathcal{A}$ correctly decides if $(x, k) \in \mathcal{L}$ in time $\mathcal{O}(f(k) \cdot n^c)$, where f is a function depending only on k and n is the size of the instance x.

A parameterized problem can take the following form:

> A Parameterized Problem
> **Input:** an instance of the problem and a parameter.
> **Question**: a YES-NO question about the instance and the parameter

2.1.1 Kernelization

Kernelization is the process of taking a problem input instance (x, k) and mapping it to another smaller instance (x', k'), called the *kernel*. This processing stage is in time polynomial in $|x|$, where the size of x' is bounded by some computable function f in k [8, p. 39]. The kernel is then analyzed, and if a solution for the kernel exists, then a solution for the original problem instance must also exist [8, p. 39]. In other words, $(x', k') \in \mathcal{L}$ if and only if $(x, k) \in \mathcal{L}$.

2.1.2 Complexity classes

If n is the instance size and k is the parameter, then the complexity class P is the class of problems that can be solved in polynomial time, i.e., $n^{\mathcal{O}(1)}$. As previously explained in Section 2.1, the complexity class FPT is the class of parameterized

problems solved in $f(k) \cdot n^{\mathcal{O}(1)}$ time. Finally, the complexity class XP is the class of parameterized problems that can be solved in $\mathcal{O}(f(k) \cdot n^{g(k)})$ time, where $g : \mathbb{N} \to \mathbb{N}$, i.e., it can be solved in polynomial time when k is a constant.

2.2 String combinatorics

This section presents important key results for related string combinatorial problems, which may be considered a literature review.

2.2.1 Squares

Let A be a string over an alphabet Σ, where the length of A is denoted as $|A|$. If $|A| = 0$, then $A = \emptyset$. For an integer $i \geq 0$, the i:th power of A is defined inductively as $A^0 = \emptyset$, $A^n = AA^{n-1}$. A square is a tandem repeat of the form AA, where $A \neq \emptyset$. Finally, two squares XX and YY are *distinct* if $XX \neq YY$.

Theorem 2.2.1. *Any string of length n has at most $2n$ distinct squares [9].*

Fraenkel and Simpson prove this in [9], and the best bound to date, $2n - \mathcal{O}(\log n)$, is given in [10] by Ilie. Based on numerical evidence, the conjectured bound is n [10].

Theorem 2.2.2. *All tandem repeats in a string of length n can be identified in $\mathcal{O}(n)$ time [6].*

Lemma 2.2.3. *Any string of length n has at most $\frac{n^2}{4}$ squares [6, 9].*

2.2.2 Periods

Once again, we start by giving the notation for a period. We say that a string S has a period p if $S[i] = S[i+p]$, for all i, $1 \leq i \leq |S| - p$. Furthermore, a *run*, sometimes referred to as a *maximal repetition*, in a string S is an interval $[i..j]$ such that the substring $S[i..j] = S[i]S[i+1]..S[j]$ is periodic and has period $p \leq (j - i + 1)/2$ where the periodicity cannot be extended to the right nor left. More precisely, $S[i-1] \neq S[x+p-1]$ and $S[j-p+1] \neq S[j+1]$. For example, the substring `BABAB` in the string $S =$ `BABBABABBABBA` is a maximal repetition, with period 2, while the substring `BABA` is not. Another example of a run in S is `BABBABBA` with period 3. Runs encode all repetitions in the string S, in a most compact way, hence their importance [11].

Theorem 2.2.4. *Any string of length n has $\mathcal{O}(n)$ runs [11].*

2.3 Exhaustive search

Given strings S and T, a way to find whether $S \Rightarrow T$ or not is to apply tandem contractions on T. Intuitively, contracting existing squares in T might be more efficient than applying tandem duplication on S, where it would be possible to duplicate arbitrary substrings that are not part of the resulting string T. The k-TD problem can also be solved by tandem contraction from T to S.

By first identifying all squares in T, we contract every single one independently and then repeat this process on each resulting substring. This exhaustive approach will attempt every combination of contractions possible on a string T. If in some step it is discovered that the resulting substring is identical to S, there must exist a sequence of transformations such that $S \Rightarrow T$.

The exhaustive approach will generate a tree-like structure with a depth bound by the parameter k. If a match between a resulting substring and S is not found after k levels in the search tree, the search terminates, and we have that $S \not\Rightarrow_k T$. The search tree has a depth of at most k and a degree of at most $\mathcal{O}(n^2)$, see Lemma 2.2.3. Consequently, the tree has at most $\mathcal{O}(n^{2k})$ nodes, and the entire tree-like structure can be exhausted in time $\mathcal{O}(n^{2k})$ [5].

2.4 Dynamic programming

Dynamic programming is an optimization method for solving complex problems by dividing them into a sequence of smaller subproblems. By dividing the problem into smaller sequential decision problems, the result can be re-used for future evaluations where similar subproblems could be re-evaluated, and solutions to some sequential decision problems may affect future problems.

The *optimal substructure* property of problems in dynamic programming means that an optimal solution to a problem can be obtained by a combination of optimal solutions to its subproblems. When an optimal solution to a subproblem has been found, *memoization* is utilized, an optimization technique used to store the results of time-inefficient calculations. Before solving a new subproblem, the algorithm checks if it has been previously computed and memoized. If that is the case, the algorithm can use the result directly without recomputing the same subproblem. This approach is called the *top-down approach* [12].

Another way of solving dynamic programming problems is to apply a *bottom-up* approach [12]. This technique involves solving smaller subproblems first and using their solutions to solve larger subproblems. Once all the solutions to all subproblems are known, it is easy to backtrack and find the solution to the main problem.

3

Overlapping squares

Damaschke suggested a new bound $\mathcal{O}((n \log n)^k)$ for an algorithm by only considering pairwise disjoint substrings for contraction. In Section 3.2, we give details and prove this new bound for the k-TD problem.

3.1 The algorithm

The algorithm recursively calls the resulting string while traversing down the tree-like structure, similarly to the exhaustive approach in Section 2.3. However, the key difference with this approach is the utilization of memoization and properties of strings to avoid unnecessary evaluations. Furthermore, the algorithm further reduces the branching number by terminating a branch if a previous branch contains a solution with a lower branching number.

The algorithm works as follows. First, given an instance (S, T), all substrings of any fixed length j in T, only pairwise disjoint substrings need to be considered for being the left half of the square to be contracted, as suggested by Damaschke. This idea reduces the degree of the search tree to $\mathcal{O}(n \log n)$, see Section 3.2. Second, if the number of contracted symbols exceeds $|T| - |S|$ in a branch, no sequence of tandem contractions to S exists, and the branch can be safely terminated. In each step, if S is not equal to T, the algorithm checks if the resulting substring has previously been computed and use that contraction result. If it has not been computed, the algorithm recursively evaluates the instance and memoizes the result. Finally, if the resulting substring in some branch is equivalent to S at some depth $k' \leq k$, we return k'. Otherwise, the current depth exceeds k and the algorithm terminates the branch. If all branches have been evaluated and no solutions have been found, we know that $S \not\Rightarrow_k T$.

3.2 Overlapping squares

In Section 2.3, we learned about the $\mathcal{O}(n^{2k})$ bound for the exhaustive algorithm. Here, we significantly improve it by showing that a worst-case running time of $\mathcal{O}((n \log n)^k)$ can be achieved by utilizing the concept of overlapping squares. Consider the string $S =$ `ABABA` that contains the substrings `A` and `B`. There are two squares in S, `ABAB` and `BABA`, and contracting any of them will yield the same re-

sulting substring $S' = \texttt{ABA}$.

Essentially, for a given problem instance (S, T) where we ask if $S \Rightarrow T$, we do not need to consider all $\mathcal{O}(n^2)$ squares since many contractions could result in the same strings. Instead, we only consider pairwise disjoint substrings of the same length in T.

Lemma 3.2.1. *For any given problem instance (S, T), only pairwise disjoint substrings in T need to be evaluated since contracting same-length overlapping squares yields the same result.*

Proof. Let XX and YY be two arbitrary same-length squares in a string T, where the first Y overlaps the first X and Y starts later than X. Let $\texttt{A}$ denote the prefix of X before Y starts, and let $\texttt{B}$ be the rest of X, that is, $X = \texttt{AB}$ and $XX = \texttt{ABAB}$. Now, since the first Y starts with the first $\texttt{B}$, and the two squares have the same length, we get $Y = \texttt{BA}$. As a result, the second instance of Y starts with the second $\texttt{B}$; therefore, the substring $\texttt{ABABA}$ appears in T. Now, it is enough to contract either XX or YY since the contraction of either of them will yield the same resulting substring $\texttt{ABA}$. In other words, for all substrings of T of any fixed length, only pairwise disjoint substrings need to be considered candidates for being the left half of a square to be contracted. All others would lead to the same contraction results. Thus, we only need to evaluate pairwise disjoint substrings for contraction.

□

Lemma 3.2.2. *For any given problem instance (S, T), only $\mathcal{O}(n \log n)$ squares need to be evaluated.*

Proof. From Lemma 3.2.1 we conclude that only pairwise disjoint substrings need to be evaluated for branching. Naturally, there exists at most n/j pairwise disjoint substrings of length j in a string of length n. Therefore, we can express the total number of all pairwise disjoint substrings from length 1 to n in the following manner:

$$\sum_{j=1}^{n} \frac{n}{j} = \frac{n}{1} + \frac{n}{2} + ... + \frac{n}{n} = n * \left(1 + \frac{1}{2} + ...\right) = n * H_n \tag{3.1}$$

The harmonic series is an estimate of the natural logarithm [13]. We use calculus to show this fact.

$$\int_1^n \frac{1}{x}\, dx = ln(n) \tag{3.2}$$

We use the following inequality to show that $H_N \approx \ln N$.

$$\sum_{x=1}^{n} \frac{1}{x+1} \leq \int_1^n \frac{1}{x}\, dx = ln(n) \leq \sum_{x=1}^{n} \frac{1}{x} \tag{3.3}$$

The bound is tight and it follows that $n * H_N$ is in $\mathcal{O}(n \log n)$.

□

Theorem 3.2.3. *The k-TD problem can be solved in $\mathcal{O}((n \log n)^k)$ time.*

Proof. The search tree has a depth of at most k, and from Lemma 3.2.2, we conclude that the search tree has a degree of $\mathcal{O}(n \log n)$. Thus, the entire search tree can be exhausted in time $\mathcal{O}((n \log n)^k)$. □

This new bound improves our previous XP bound, $O(n^{2k})$, first introduced in Section 2.3.

3.2.1 Impossible overlaps

We identify impossible overlaps to reduce the possible number of eligible contractions further. Impossible overlaps are not utilized in the algorithm but could be further explored in future research.

Proposition 3.2.4. *Let X and Y be two substrings of the string S and let l_A, p_A be two positive integers denoting the length and starting position, respectively, of a string A in S. Given two squares XX and YY in S, we say that the tuples $(l_X, p_X), (l_Y, p_Y)$ describes the length and starting position of X and Y. If $p_x + 2l_X - 2l_Y < p_Y < p_X + 2l_X$ does not hold, then an overlap between XX and YY does not exist.*

3.2.2 Variable-length overlaps

By looking at equivalence on positions and variable-length overlaps, we can express which equation of symbols is implied by an overlap of squares. We conjecture that the upper bound of the algorithm can be improved since overlapping squares imply short periods. Consider the case where the overlapping squares XX and YY are of different lengths. Let l_X and l_Y denote the length of the substrings X and Y, where $l_X, l_Y \geq 2$ and $l_X > l_Y$. Let p_X and p_Y be the start index of the first half of the square, where $p_X < p_Y$.

Let $XX = x_1x_2..x_nx_1x_2..x_n$ and $YY = y_1y_2..y_my_1y_2..y_m$ be two overlapping squares in the string T for a given problem instance (S, T). The first y_1 is aligned with the first x_{k+1}, and $m < n$. Furthermore, the inequality $n < k + m < 2n$ must be satisfied, meaning that YY exists in XX. Then it follows that:

$$x_{k+1} = x_{k+1+(n-m)}, x_{k+2} = x_{k+2+(n-m)}, ..., x_{k+(n-k)} = x_{k+(n-k)+(n-m)}$$

where the last equation simplifies to

$$x_n = x_m$$

This entails that the overlap $x_{k+1}..x_n$ has a period of length $n - m$.

3.3 Length difference

For a given problem instance (S, T), comparing the difference in length between the two strings introduces a new parameter. One immediate observation is that it can help us further limit the search space by disregarding certain contractions.

Proposition 3.3.1. *For any given problem instance (S, T), all contractions that remove a number of symbols that is greater than $|T|-|S|$ or greater than the remaining number of symbols to contract can be disregarded.*

This new parameter naturally introduces a new bound to consider. For any given problem instance (S, T), we cannot delete more than $d = |T| - |S|$ symbols from T since a tandem contraction at the very least deletes one symbol from T. Therefore, we know that we will at most perform d contractions on T. If $k > d$, we can tighten the bound to $k = d$ to further limit the depth of the search tree.

4

Dynamic Programming

This chapter presents our algorithmic contributions to the Tandem Duplication Distance Problem, together with correctness proofs and runtime analysis.

4.1 An FPT algorithm for TDDP

We propose a new FPT algorithm for the Tandem Duplication Distance Problem utilizing the concept of dynamic programming over substrings. In particular, the algorithm considers the length difference $d = |T| - |S|$, initially introduced in Section 3.3, as the parameter. First, we prove that the problem can be solved with the help of dynamic programming. Second, we reintroduce the algorithm, originally introduced in Section 3.2, with a minor modification. The algorithm is used locally on pairwise disjoint substrings, also referred to as *subproblems*. Finally, the FPT algorithm is presented with correctness proof and runtime analysis.

Recall the k-TD problem originally introduced in Section 1.3.

> Tandem Duplication Edit Distance (k-TD)
> **Input:** Strings S and T over the same alphabet Σ and the integer k.
> **Question**: Is $dist_{TD}(S, T) \leq k$?

First of all, we prove that the k-TD problem can be solved with the help of dynamic programming on disjoint substrings, henceforth referred to as *windows*, each of some length at most $2d$, such that contractions only occur inside these windows.

Lemma 4.1.1. *In a string T, where any sequence of contractions deletes at most d symbols, T can be partitioned into disjoint windows of size at most $2d$ such that contractions are contained within the windows and will not influence other windows.*

Proof. Given a string T where some sequence of contractions deletes d symbols, we mark all symbols in T that appear in any contracted squares. There are at most $2d$ such symbols as we are only allowed to delete at most d symbols from T in total. Consider all maximal substrings of marked symbols, i.e., the longest substrings that do not contain unmarked symbols. Every such substring has a length of at most $2d$, and any contractions must happen within them. Furthermore, unmarked symbols can be divided into substrings of length at most $2d$. Thus, the string T can be partitioned into windows of length at most $2d$, where any contractions are contained within the windows. □

Lemma 4.1.2. *A string S can be partitioned into disjoint windows of size at most $2d$ such that each window in T can be contracted to an equivalent window in S.*

Proof. Lemma 4.1.1 shows that a string T can be partitioned into windows of size at most $2d$. Each window in T can be contracted to an equivalent window in S. Given the two first windows v and w in S and T, respectively, there must exist a sequence of contractions such that w can be contracted to v. This follows for any subsequent pair of windows as well. Note that some pairs of windows in S and T may already be equivalent. Since windows in T can never grow larger, the window size in S is also at most $2d$. □

It follows from Lemma 4.1.1 that all contractions that are part of the solution can be partitioned into windows of length at most $2d$, and there could be multiple contractions to be made within the same window. Furthermore, new squares can emerge that overlap windows as contractions are made within a window. However, it has been shown that the contractions that are part of the solution are already contained in the windows, so that no such overlapping square can be part of the solution. Thus, new windows can be appended to a sequence of windows.

Furthermore, for each window in T, the algorithm computes every combination of contractions possible on a string, using the algorithm introduced in Section 3.2 with a minor modification. Let $\delta(S, T, d)$ denote the minimum number of tandem duplications from S to T given d symbols to remove. Let q denote the total number of distinct squares in T. Then, the recurrence formula can be expressed as:

$$\delta(S, T, d) = \sum_{i=1}^{q} min\{1 + \delta(S, T\backslash\{x_i\}, d - \frac{|x_i|}{2})\} \tag{4.1}$$

Recall, that if two squares of the same length overlap, only the leftmost square is contracted since both of them yield the same contraction result. Therefore, the size of q is $\mathcal{O}(n \log n)$. Furthermore, x_i denotes the i:th square in T from the left, and $T\backslash\{x_i\}$ denotes the contraction result. Finally, $|x_i|$ denotes the length of the square. The base cases for δ can be viewed in (4.2).

$$\delta(S, T, k) = \begin{cases} \infty & \text{iff } S \neq T \text{ and } d \leq 0 \\ 0 & \text{iff } S = T \text{ and } d \geq 0 \end{cases} \tag{4.2}$$

4.1.1 The algorithm

The idea is to build the solution gradually, in a growing prefix by evaluating independent windows of T and S with lengths of at most $2d$ using dynamic programming. The windows are locally and independently exhausted from left to right, and the minimum edit distance is retrieved through backtracking. Let $OPT(i, j)$ denote the minimum number of contractions needed to transform the first j symbols in T to the first i symbols in S. The results are stored in a $|S| \times |T|$ matrix M, where each entry $M[i][j]$ corresponds to the edit distance between the prefixes $S[1, i]$ and $T[1, j]$. The matrix is presented in Figure 4.1.

$$\begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{ccccc} T_1 & T_2 & T_3 & \cdots & T_n \end{array} \\ \begin{bmatrix} 0 & v_{1,2} & v_{1,3} & \dots & v_{1,j} \\ \infty & v_{2,2} & v_{2,3} & \dots & v_{2,j} \\ \infty & \infty & v_{3,3} & \dots & v_{3,j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \dots & v_{i,j} \end{bmatrix} \end{array}$$

Figure 4.1: The algorithm generates an upper triangular matrix M of size $|S| \times |T|$ since there cannot exist a sequence of contractions from a shorter to a longer string. The value $v_{i,j}$ denotes the minimum number of contractions from $S[1, i]$ to $T[1, j]$.

The core idea of the algorithm is that it locally exhausts each window and memoizes its contraction results. Then, it adds the edit distance result of the current window to the next window's edit distance result. Correctness of this approach follows from Lemma 4.1.1 and Lemma 4.1.2, which allows us to solve the k-TD problem by using dynamic programming. Therefore, we propose the following recurrence.

$$OPT(i, j) = \min_{\substack{1 \le i' \le 2d \\ 1 \le j' \le 2d}} \{\delta(S[i - i', i], T[j - j', j], d) + OPT(i - i' - 1, j - j' - 1)\} \quad (4.3)$$

Claim 4.1.3. *$OPT(i, j)$ satisfies the above recurrence.*

Proof. It follows from Lemma 4.1.1 that the string T can be partitioned into windows of length at most $2d$ such that all contractions are contained within independent windows. Furthermore, Lemma 4.1.2 shows that S also can be partitioned into windows of length at most $2d$. The recurrence $OPT(i, j)$ is defined as the number of contractions needed to transform the first j symbols in T to the first i symbols in S. For each i, j, the algorithm looks back $2d$ steps in T to some j' and $2d$ steps in S to some i'. The algorithm δ is executed with the following parameters $\delta(S[i - i', i], T[j - j', j], d)$ and a previous OPT value $OPT(i - i' - 1, j - j' - 1)$ is added to the result. This approach will consider all partitions of S and T and return the optimal value through minimization. For correctness, let $V = v_1 ... v_l$ denote the windows of the partitioned string S, and let $W = w_1 ... w_l$ denote the windows of the partitioned string T. For each window $v_a \in V$ and $w_a \in W$, the algorithm calculates the minimum number of contractions needed such that $w_a \mapsto v_a$. Initially, it computes the minimum edit distance between the first pair of windows w_1, v_1. For all windows where $a > 1$, the edit distance between the previous pair of windows is added to the edit distance of the current pair of windows. □

Claim 4.1.4. *The algorithm computes the tandem edit distance between two strings S and T in time $\mathcal{O}(nd^3(d \log d)^d)$.*

Proof. Let n denote the length of a string T. The total number of index pairs (i, j) in T is bound by $\mathcal{O}(nd)$ since each j can deviate $\mathcal{O}(d)$ steps from i. In each iteration, there are $\mathcal{O}(d^2)$ pairs of window lengths to check. Thus, the total number of windows in T is $\mathcal{O}(nd^3)$. Furthermore, it follows from Theorem 3.2.3 that the time complexity of the local exhaustive algorithm is $\mathcal{O}(n \log n^k)$, where n denotes the length of the target string T and k denotes the number of contractions. Since

the length of each subproblem in T is $1, .., 2d$, the longest length of a window from T is $2d$. Also, since the algorithm can remove at most d symbols from the substrings of length at most $2d$, the exhaustive local approach will generate a tree-like structure with a depth bound by d. The search tree has a degree of at most $\mathcal{O}(d \log d)$. Thus, the entire search tree can be exhausted in time $\mathcal{O}((d \log d)^d)$, and the runtime of the algorithm is $\mathcal{O}(nd^3(d \log d)^d)$.

□

Theorem 4.1.5. *The Tandem Distance Problem is fixed-parameter tractable in the natural parameter d, where $d = |T| - |S|$.*

Proof. In the local exhaustion phase, the algorithm fixes the windows lengths specified in Lemma 4.1.1 and Lemma 4.1.2 by $\mathcal{O}(d)$ and computes each window in time $\mathcal{O}((d \log d)^d)$ according to Claim 4.1.4. Additionally, it also follows from Claim 4.1.4 that the total number of windows is $\mathcal{O}(nd^3)$. Thus, the algorithm solves the k-TD problem in $f(d) \cdot n^{\mathcal{O}(1)}$ time. □

5

Discussion

The following chapter serves two purposes. First, in Section 5.1, we discuss and highlight the results initially presented in Chapters 3 and 4. Then, in Section 5.2, future work is suggested.

5.1 Results

In this thesis, we have presented algorithms for the Tandem Duplication Distance Problem. First, a branch and bound algorithm was designed and developed by identifying the properties of strings that are utilized in the algorithm. The main idea that enabled us to reduce the degree of the search tree from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ is that the algorithm does not have to consider even-length overlapping squares for contraction. Furthermore, in Chapter 4, a dynamic programming algorithm was presented. The algorithm is fixed-parameter tractable for the length difference parameter $d = |T| - |S|$ which is used to partition the strings into windows of length at most $2d$. Within these windows, we apply the branch and bound algorithm. This allows us to exhaust substrings locally within windows where the degree is further reduced to $\mathcal{O}(d \log d)$. We will get a time-bound that is exponential in the parameter d, but only polynomial in length n. By utilizing the properties above, we have improved the runtime of previous TDDP algorithms.

The length difference parameter d is important since it indicates exactly how many symbols to remove from all windows together. The parameter also enabled us to prove that k-TD is FPT in d. The FPT bound is generous and can be further improved because, for each contraction, d decreases, and contraction lengths that are close to d could imply fewer options for subsequent contractions. Other properties, such as variable-length overlaps and impossible overlaps, have also been identified. Although these properties do not affect the overall time-bound, we conjecture that they can be utilized to reduce the total branching number of the algorithm.

We conclude the discussion by answering the research questions originally posed in Section 1.3.

For an alphabet of size q, where $q \geq 2$, can we develop an algorithm for k-TD with fixed k, as large as possible?
An exhaustive algorithm has been proposed with a running time of $\mathcal{O}((n \log n)^k)$ and is presented in Chapter 3.

Are we able to produce a fixed-parameter tractable (FPT) algorithm for k-TD with the parameter d?
We managed to show that TDDP is FPT in the parameter d. The original research question was whether k-TD was FPT in the parameter k. However, the research direction was slightly altered.

If the development of the FPT algorithm fails, can we identify the difficulties and gain a specific understanding of them?
The development of the FPT algorithm was successful after the natural parameter d was selected over the parameter k.

5.2 Future work

We suggest opportunities to improve the previously presented algorithms, which have not been pursued due to the project's time constraints.

5.2.1 Overlapping squares

We believe that the search space can be further pruned by considering the parameter k in combination with the parameter d. More specifically, we can limit the branch's depth by considering the number of removed symbols.

Furthermore, the idea of variable-length overlapping squares might be a promising research direction for further restricting the search space. Also, Section 2.2 introduces the concept of periods and, more importantly, runs, and they are essential since they encode all repetitions in a string in a compact matter. What does the existence of only $\mathcal{O}(n)$ runs imply for the branching number?

5.2.2 Kernelization

Damaschke suggested that if the string T of length n contains a long substring of length $\mathcal{O}(n/d)$ with a period at most $d = |T| - |S|$, then there cannot be very many possible contraction results. Hence, it is possible that any such substring can be shortened to a substring with the same period, but now with a length that depends only on d, without destroying all optimal solutions. Due to the limited time frame, we could not pursue this idea further, so it is left as an open problem. Finally, we conjecture that the idea of independent windows might be an interesting angle to tackle kernelization, but no further analysis has been conducted.

5.2.3 The FPT algorithm

As previously mentioned, the exhaustive algorithm used on independent windows has a running time of $\mathcal{O}((d \log d)^d)$. We conjecture that the exponent might be tightened since the number of remaining symbols to remove is not decremented linearly; instead, it depends on the length of the contracted square.

Finally, the question remains; is the Tandem Duplication Distance Problem fixed-parameter tractable for the parameter k?

6
Conclusion

The primary purpose of the thesis project was to study string patterns by designing and proving new algorithms and, more specifically, to research if the k-TD problem is fixed-parameter tractable for the parameter d.

Consequently, a new algorithm was presented, which utilizes the idea of overlapping squares to prove that it is enough to consider same-length pairs of disjoint substrings for contraction since we prove that overlapping squares of the same-length yield the same contraction results. This idea reduces the number of squares to consider from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, which results in a running time of $\mathcal{O}((n \log n)^k)$.

The natural parameter $d = |T| - |S|$ was considered over the number of contractions k. As a result, an FPT algorithm was designed, developed, and presented together with correctness proof and run-time analysis. The algorithm is of the dynamic programming flavor where the exhaustive algorithm is utilized internally on independent disjoint substrings called windows of length at most $2d$. As a result, the run time of the local exhaustion is significantly improved. Therefore, the FPT algorithm has a run-time of $\mathcal{O}(nd^3(d \log d)^d)$ and is exactly solvable in practice.

Additionally, other string properties were also identified and presented, but are not used by the algorithms. We conjecture that the exhaustive algorithm can be further improved by exploring variable-length overlapping squares and considering the number of deletions d as the tree depth instead of the number of contractions k.

Conclusively, two algorithms are presented together with correctness proofs and run-time analysis. However, we conjecture that the exponent of the exhaustive algorithm, with parameter d, can be reduced further — the longer squares the algorithm contracts, the fewer eligible squares to contract remain. Finally, the question remains for the k-TD problem: Is k-TD fixed-parameter tractable in the parameter k?

Bibliography

[1] G. R. Sutherland and R. I. Richards, "Simple tandem dna repeats and human genetic disease." *Proceedings of the National Academy of Sciences*, vol. 92, no. 9, pp. 3636–3641, 1995.

[2] A. J. Hannan, "Tandem repeats mediating genetic plasticity in health and disease," *Nature Reviews Genetics*, vol. 19, no. 5, pp. 286–298, 2018.

[3] F. Cicalese and N. Pilati, "The tandem duplication distance problem is hard over bounded alphabets," in *International Workshop on Combinatorial Algorithms*. Springer, 2021, pp. 179–193.

[4] P. Leupold, V. Mitrana, and J. M. Sempere, "Formal languages arising from gene repeated duplication," in *Aspects of Molecular Computing*. Springer, 2004, pp. 297–308.

[5] M. Lafond, B. Zhu, and P. Zou, "The tandem duplication distance is np-hard," 2019.

[6] D. Gusfield and J. Stoye, "Linear time algorithms for finding and representing all the tandem repeats in a string," *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 525–546, 2004.

[7] F. Baylis, M. Darnovsky, K. Hasson, and T. M. Krahn, "Human germline and heritable genome editing: the global policy landscape," *The CRISPR Journal*, vol. 3, no. 5, pp. 365–377, 2020.

[8] R. G. Downey and M. R. Fellows, *Parameterized Complexity*. Springer New York, 1999. [Online]. Available: https://doi.org/10.1007%2F978-1-4612-0515-9

[9] A. S. Fraenkel and J. Simpson, "How many squares can a string contain?" *Journal of Combinatorial Theory, Series A*, vol. 82, no. 1, pp. 112–120, 1998.

[10] L. Ilie, "A note on the number of squares in a word," *Theoretical Computer Science*, vol. 380, no. 3, pp. 373–376, 2007.

[11] R. Kolpakov and G. Kucherov, "Finding maximal repetitions in a word in linear time," in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 1999, pp. 596–604.

[12] S. R. Eddy, "What is dynamic programming?" *Nature biotechnology*, vol. 22, no. 7, pp. 909–910, 2004.

[13] J. H. Conway and R. Guy, *The book of numbers*. Springer Science & Business Media, 1998, p. 143.