SAT-based Local Improvement for the Closest String Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Logic and Computation

eingereicht von

Florentina Voboril, B.Sc.
Matrikelnummer 11730664

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider
Mitwirkung: Dipl.-Ing. Dr.techn. Andre Schidler
Dr.techn. Vaidyanathan P. R.

Wien, 30. Jänner 2024

Florentina Voboril [redacted] Stefan Szeider

TU WIEN Informatics

SAT-based Local Improvement for the Closest String Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Logic and Computation

by

Florentina Voboril, B.Sc.
Registration Number 11730664

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider
Assistance: Dipl.-Ing. Dr.techn. Andre Schidler
Dr.techn. Vaidyanathan P. R.

Vienna, January 30, 2024

Florentina Voboril [redacted] Stefan Szeider

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Florentina Voboril, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Jänner 2024

Florentina Voboril

Danksagung

An dieser Stelle will ich mich gerne bei all jenen bedanken, die mich während des gesamten Prozesses meiner Diplomarbeit - von der Themenfindung bis zum letzten Feinschliff - unterstützt und motiviert haben.

Besonderer Dank gilt meinem Betreuer Stefan Szeider und meinen beiden Co-Betreuern Andre Schidler und Vaidyanathan Peruvemba Ramaswamy, die mir mit hilfreichen Ratschlägen und wertvollem Feedback beiseite gestanden sind.

Acknowledgements

I would like to take this opportunity to thank all those who supported and motivated me throughout the entire process of my thesis - from finding the topic to the final touches.

Special thanks go to my supervisor Stefan Szeider and my two co-supervisors Andre Schidler and Vaidyanathan Peruvemba Ramaswamy, who were able to assist me with helpful advice and valuable feedback.

Kurzfassung

Das Closest String Problem (CSP) hat also Eingabe n Strings der Länge m über einem Alphabet A. Das Ziel ist es, einen String gleicher Länge zu finden, die den maximalen Hamming-Abstand d zu allen Eingabe-Strings minimiert. In dieser Masterarbeit fokussieren wir uns auf das CSP mit dem Alphabet $A = \{A, C, G, T\}$. Die vier Buchstaben stehen für die vier Basen, die die genetische Information in DNA-Molekülen kodieren: Adenin (A), Cytosin (C), Guanin (G) und Thymin (T). Eine praktische Anwendung besteht darin, neue DNA-Sequenzen zu erstellen, die allen vorgegebenen Eingabesequenzen ähnlich sind. Es wurde gezeigt, dass CSP NP-vollständig ist [KLLM+03]. Es stellt sich also die Frage, wie dieses Problem effizient gelöst werden kann. Für NP-vollständige Probleme können exakte Algorithmen nicht in polynomieller Zeit laufen. Heuristische Algorithmen haben hingegen den Nachteil, dass sie nicht immer die optimale Lösung liefern.

Die SAT-based Local Improvement Method (SLIM) ist ein neuer Ansatz, der exakte und heuristische Methoden kombiniert, um die Vorteile beider Methoden zu nutzen. In früheren Forschungsarbeiten wurde SLIM erfolgreich auf verschiedene Probleme angewandt, wie z.B. treewidth, branchwidth und Graphenfärbung. Es hat also das Potenzial, auch für das CSP geeignet zu sein. Der SLIM-Ansatz besteht aus zwei Schritten: Im *initiation step* wird eine bestehende Heuristik verwendet, um eine Anfangslösung effizient zu berechnen. Danach wird im *local improvement step* ein lokaler Teil der bestehenden Lösung ausgewählt und durch einen SAT-Solver verbessert wird. Dieser Schritt kann beliebig oft wiederholt werden. Die Forschungsfrage dieser Arbeit lautet

Können wir SLIM nutzen, um die Lösungen aktueller Heuristiken für CSP zu verbessern?

Die Masterarbeit beschäftigt sich auch damit, was die Leistung von SLIM beeinflusst und welche Parametereinstellungen die besten Ergebnisse liefern. Um die Forschungsfrage zu beantworten, werden wir mehrere Experimente durchführen. Zunächst lassen wir eine bestehende Heuristik so lange laufen, bis sie eine erste Lösung findet. Dann verwenden wir diese Lösung als Ausgangslösung für SLIM und lassen beide Methoden parallel laufen. Nach Ablauf der vorgegebenen Zeit werden die Ergebnisse der beiden Methoden verglichen. Die Ergebnisse zeigen, dass es bereits eine gut funktionierende Heuristik gibt. Dennoch konnte unser SLIM-Ansatz die Lösung für einige der Instanzen verbessern.

Abstract

The Closest String Problem (CSP) has as input n strings of length m over an alphabet A. The task is to find a string of the same length that minimizes the maximal Hamming distance d to all the input strings. In this thesis, we focus on the CSP with the alphabet $A = \{A, C, G, T\}$. The four characters represent the four bases that encode the genetic information in DNA molecules: adenine (A), cytosine (C), guanine (G), and thymine (T). One of its applications is to create new DNA sequences similar to all given input sequences. It was shown that CSP is NP-complete [KLLM+03]. Thus, the question arises of how this problem can be solved efficiently. For NP-complete problems, exact algorithms cannot run in polynomial time. On the other hand, heuristic algorithms have the downside that they do not consistently deliver the optimal solution.

The SAT-based Local Improvement Method (SLIM) is a new approach that combines exact and heuristic methods to use the advantages of both methods. In previous research SLIM was successfully applied SLIM to different problems, like treewidth, branchwidth, and graph coloring. So it also has the potential to be suitable for CSP. The SLIM approach consists of two steps: In the *initiation step*, we use an existing heuristic to compute an initial solution efficiently. Afterward, we execute the *local improvement step*, in which a local part of the existing solution is selected and improved by an SAT solver. This step can be executed as often as desired. The research question of this thesis is

Can we use SLIM to improve the state of the art in solving the Closest String Problem?

The scope of this thesis also includes finding out what influences the performance of SLIM and which parameters and adjustments deliver the best results. To answer the research question, we will conduct multiple experiments. First, we run an existing state-of-the-art heuristic until it finds one solution. Then, we use this solution as the initial solution for SLIM and let both methods run in parallel. After a global timeout, the results of both methods are compared. The results show that there already exists a well working state-of-the-art heuristic. Nevertheless, our SLIM approach could improve the solution for some of the benchmark instances.

Contents

CHAPTER 1

Introduction

1.1 The Closest String Problem

The Closest String Problem (CSP) was introduced in 2003 by Kevin Lanctot at al. [KLLM+03]. It is also known as Center String Problem and for binary inputs as Hamming Center Problem [GJL99] or Covering Radius problem [CLLM94]. Given n input strings of length m over an alphabet A, the CSP tries to find a string of the same length that minimizes the maximal Hamming distance d to all the input strings. The Hamming distance and the formal definition are given in Section 2.1. However, it should not be confused with the Constraint Satisfaction Problem, which is also abbreviated as CSP.

The CSP has some applications: In Coding Theory, it can be used to encode a set of messages, for write-once memories, testing, and data compression [CLLM94]. It can also be applied in Computational Biology, an interdisciplinary field that applies techniques from Mathematics and Computer Science, like computational simulations and data analysis, to understand biological systems. One area in Computational Biology that can make use of the CSP is drug target design. Here, it could help to find a genetic sequence that combats bacteria but does not harm human beings. The problem can also be used to create new DNA sequences similar to all given input sequences [KLLM+03]. It can even help to construct phylogenetic trees by analyzing protein regions of different species. [LW67]

It was shown that the CSP is NP-hard [KLLM+03]. This is even the case for binary strings [FL97]. Thus, the question arises of how this problem can be solved efficiently. For NP-complete problems, exact algorithms cannot run in polynomial time. On the other hand, heuristic algorithms have the downside that they do not consistently deliver the optimal solution.

1.2 State of the art

There are different approaches to solving NP-hard problems: Exact, fixed-parameter tractable (FPT), approximation, and heuristic algorithm.

Finding an *exact algorithm* for CSP that runs in polynomial time is impossible unless P=NP. One possible approach is to use a straightforward enumerative method that tries all possible candidate strings. This has a running time of $\mathcal{O}^*(|A|^m)$, which can be quite time-consuming. Alternatively, Dalpasso [DL18] et al. introduce an integer linear programming and a dynamic programming procedure that give exact solutions.

Fixed-parameter tractability is a different approach to deal with NP-hard problems. The underlying idea is to isolate exponential terms to a specific parameter. When this parameter has a small, bounded value, the algorithm can find the exact solution of the instance fast [DF99]. There exists an FPT algorithm parameterized by the maximal distance d. Bulteau et al. [BHKN14] describes the following search tree strategy: We assume that there exists a string with Hamming distance of at most d to all other input strings. We can start with any of the input strings as a candidate string. If, for the current node, there is no input string s which has a distance of more than d to the candidate string we are done. Otherwise, we have to branch into $d+1$ cases, where in each case a different position is chosen that is changed to the character of s. Since it must be possible to reach the closest string by changing at most d characters from our chosen first string, our search tree has a depth of at most d and the branching factor is bounded by $d+1$. That yields a running time of $\mathcal{O}^*((d+1)^d)$. This approach is assumed to be basically optimal. For other parameters, there are not so good chances of success. Since CSP is NP-complete even for binary strings, using the alphabet size $|A|$ as a parameter for tractability does not make sense. Using parameter m would only allow short strings in the input, which is unsuitable for many input instances. FPT algorithms parameterized on n exist, but this result is only of theoretical interest due to a huge combinatorial explosion.

Lanctot et al. [KLLM+03] introduced an $(4/3+\epsilon)$-*approximation algorithm* for any $\epsilon > 0$.

Because of the characteristics of SLIM (see Chapter 3), *heuristics approaches* are of special interest in this thesis. Liu et al. [LHS05] introduced a sequential genetic algorithm (GA) structure and a sequential simulated annealing (SA) algorithm for CSP. Faro et al. [FP10] proposed an ant colony optimization algorithm, which outperformed the results of GA and SA. Another approach for heuristic algorithms includes a combination of an largest distance decreasing algorithm and local search strategies by Liu et al. [LLHM11]. One of the more recent approaches is a heuristic that uses wave function collapse techniques introduced by Xu and Perkins [XP22]. Their algorithm outperforms the algorithms mentioned above in solution quality, run time, or even both metrics. Their experiments used benchmark instances with up to 30 strings of length up to 5,000 characters. Another interesting approach is an algorithm based on Lagrangian relaxation by Tanaka [Tan12], which calculates solutions for CSP instances with up to 50 strings of length up to 5,000 in only about 21 seconds. On average, the solutions of such large instances only differ by 0.04 % from the optimal solution. By providing lower bounds, this heuristic can also prove the optimality of the solution. What makes our SLIM approach especially interesting is that it utilizes the results of any existing heuristics and further tries to improve its results.

1.3 Research Question

In this thesis, we will utilize the SAT-based Local Improvement Method (SLIM) to solve the CSP. SLIM is a new approach that combines exact and heuristic methods to use the advantages of both methods. It will be described in more detail in Section 3.1. It has already been successfully applied SLIM to different problems, like treewidth [FLS17], branchwidth [LOS19], and graph coloring [Sch22]. So it also has the potential to be suitable for CSP.

This thesis investigates whether SLIM is a useful approach to solving CSP. The research question is

Can we use SLIM to improve the state of the art in solving the Closest String Problem?

This improvement could include different objectives, like better results, shorter running times, or the capability to deal with more input strings or longer input strings. The corresponding working hypothesis is

SLIM can indeed improve solutions of state-of-the-art heuristics for solving the CSP.

The scope of this thesis will also include finding out what influences the performance of SLIM and which parameters and adjustments deliver the best results.

1.4 Outlook

The outlook of this thesis will be as follows: Chapter 2 will provide relevant background information about the CSP and the propositional satisfiability problem (SAT) problem. The next chapter will describe SLIM in more detail. It will explain the used encoding and give a pseudocode on how it can be used. It will also describe which choices can be made in the encoding and will describe the heuristics we used for initial solutions. The experimental results are presented in Chapter 4. The last chapter summarizes the thesis and gives an outlook for future work.

CHAPTER 2

Preliminaries

2.1 The Closest String Problem

Let x and y be strings of the same length. Then, $d(x, y)$ denotes the *Hamming Distance*, which is the number of positions in which x and y differ. For example, the strings "GATAC" and "AGTAA" have a Hamming Distance of 3, since they differ at the first, second, and fifth position. Now, we can define the CSP as follows.

Definition 2.1.1 (Closest String Problem)**.**

- **Instance:** Given a set $S = \{s_1, s_2, ..., s_n\}$ of n strings over an alphabet A of length m each.
- **Objective:** Find a string x of length m over A minimizing d such that for every string s_i in S, $d(x, s) \leq d$.

In this paper, we want to focus on the CSP with an alphabet size of 4, namely $A = \{A, C, G, T\}$. The four characters represent the four bases that encode the genetic information in DNA molecules: adenine (A), cytosine (C), guanine (G), and thymine (T). Table 2.1 shows an example of the CSP with three input strings of length 5 and a corresponding closest string x. x has Hamming distance of 2 to the input strings s_1 and s_3 and $d(x, s_2) = 1$. There is no string with Hamming distance ≤ 1 to all input strings. Thus, the minimal $d = 2$.

There is a trivial lower bound and upper bound for the CSP. The trivial upper bound is given by m, the length of the strings. The trivial lower bound is given by the halve of the *Hamming diameter*. The Hamming diameter is the maximal Hamming distance between any two input strings. In our example, the hamming distance between any two strings is 3. Thus, we cannot find a string with a Hamming distance of less than 2 to all strings.

						$d(x, s_i)$
string s_1	G	A	T	A	C	2
string s_2	A	G	T	A	A	1
string s_3	T	G	C	A	C	2
string x	A	G	T	A	C	

Table 2.1: Example of the CSP with three input strings of length 5 and output string x

Besides the CSP there are also other Consensus String Problems, including the Close to Most String Problem and Farthest String Problem.

2.2 SAT

The propositional satisfiability problem (SAT) is one of the most important problems in Computer Science. It was the first problem shown to be NP-complete. Nevertheless, there are efficient heuristic SAT algorithms, which significantly improve the running time. Modern SAT solvers have the capability to successfully deal with formulas with thousands of variables. By reducing an NP problem to SAT, one can utilize the efficiency of these SAT solvers. This leads to a variety of practical applications. Amongst the problems, that can be solved by a reduction to SAT, are for example the vertex cover problem, the graph coloring problem, or the clique problem [BHMW09].

A propositional formula in conjunctive normal form (CNF) consists of a conjunction of *clauses*. A clause is a disjunction of *literals*, where a literal is a possibly negated propositional variable, which can be assigned to true or false. For example,

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

is a propositional formula with the propositional variables x_1, x_2, and x_3. A variable x_i or its negated form $\neg x_i$ are literals. The clauses in F are $(x_1 \vee x_2 \vee x_3)$, $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $(\neg x_1 \vee x_2)$, and $(x_2 \vee x_3)$. [FBHS23]

The mapping of the variables to $\{\text{true}, \text{false}\}$ is called *assignment*. An assignment is *satisfying* when each clause contains a variable assigned to true or a negated variable assigned to false. A formula F is called *satisfiable* if there exists a satisfying assignment [BHMW09]. In the example above, F is satisfiable, since mapping x_1 and x_2 to true and x_3 to false is a satisfying assignment. The formula

$$H = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1) \wedge (x_1 \vee \neg x_2)$$

is not satisfiable, since there exists no satisfying assignment.

The SAT problem determines whether a given propositional formula is satisfiable.

There are different variants of the SAT problem, including k-SAT and SAT-f. k-SAT considers k-CNF formulas. Those are CNF formulas with at most two literals per clause. In SAT-f, every variable occurs at most f times. k-SAT-f can be seen as an intersection between k-SAT and SAT-f, where each clause has a maximum length of k, and every variable appears no more than f times.

Some variants of SAT, like 2-SAT are solvable in polynomial time. For others, strategies, like branching, clause shortening, or resolution are used to improve the running time.

CHAPTER 3

SLIM

3.1 What SLIM is?

The idea behind the SAT-based Local Improvement Method (SLIM) is to make use of SAT solvers to locally improve an existing heuristic. Even though SAT [FBHS23] is NP-complete, there are SAT solvers that are highly efficient in solving SAT problems. Nevertheless, we cannot solve the whole problem only with SAT solvers to obtain an optimal solution because the encoding has to be sufficiently small. Otherwise, it is infeasible.

The SLIM approach consists of two steps:

- *Initiation Step.* We use an existing heuristic to compute an initial solution efficiently. For example, this can be done with a fast and simple greedy algorithm.
- *Local Improvement Step.* We select a local part of the existing solution using a *local selection strategy* (see Section 3.3). Then, we use our SAT encoding to obtain an exact solution for this smaller instance. Afterwards, we replace the selected part of the original solution with the new solution.

The local improvement step can be repeated until a global timeout is reached. A local improvement is aborted if the *local timeout lt* has been reached without finding an improvement. The *local budget lb* describes the maximum size of a local instance. For the CSP, the size refers to the number of positions in the local instances.

Another nice thing about SLIM is that the heuristic for the initial solution, and the local selection strategy work more or less independently. So we can exchange them. For example, we can run SLIM on a greedy initial solution, as well as on an initial solution from another algorithm, without changing the code for SLIM. Section 3.2 describes possible heuristics in more detail.

3.2 Used Heuristics

3.2.1 Our Simple Greedy Algorithm

In order to have an algorithm that delivers a quick result, we decided to implement a simple greedy algorithm that just assigns each position of the output string the character that appears most often at that position in the input strings. This program `greedy_simple.py` can be found at https://www.ac.tuwien.ac.at/files/resources/software/csp_slim_code.zip.

3.2.2 Heuristic Algorithm based on Lagrangian Relaxation

Another heuristic algorithm with promising results is introduced by Tanaka [Tan12]. The underlying concept is to formulate the CSP as a mixed integer programming (MIP) problem. Using just a MIP solver would not be feasible for large instances. Thus, Tanaka applied the Lagrangian relaxation technique. This makes it possible to simultaneously obtain an approximate solution as well as a lower bound.

The used benchmark instances in the computational experiments had a size of $n \in \{10, 20, ..., 50\}$ strings of length $m \in \{1000, 2000, ..., 5000\}$. It was shown Tanaka's algorithm is able to find solutions, which are close to the optimal solution, very quickly. It even outperforms the existing heuristic algorithms.

Fortunately, the author was kind enough to provide me with the code for his algorithm so that I can use it for the experiments in my thesis.

3.3 Local Selection Strategie

The local selection strategy for the CSP describes, which positions of the current candidate string are picked for the local improvement step. There are different ways to do it.

The probably easiest way is to just pick random positions. This has the advantages that it is easy to understand and easy to implement. The downside is that it does not make use of the properties of the current solution.

The strategy that was used for this thesis is the following: First, we determine the input string that is the furthest away from the candidate string. The positions at which that string differs from the candidate string are 100 times more likely to be chosen. We decided to include randomness in the selection process because, for a deterministic strategy, it might happen that the same subsequences are selected multiple times when the local improvement step did not find an improvement.

There are many other local selection strategies one could use, including taking all strings into account or using different probabilities that might depend on the number or length of the input strings.

3.4 Initialization and Update Strategy for the Distance Array

In our SLIM approach for the CSP, we have a distance array, which is used to specify the maximum local distances for each string in the local SAT encoding. This array is initialized after selecting the subsequences for the local improvement step. Whenever the local SAT encoding cannot find a solution with the current distance array, it will be increased. There are different ways how the array could be initialized. In this thesis, we look at the remaining distances, i.e. the distances from all strings to the candidate string after removing the local subsequences. From these remaining distances, we store the maximum d_{max}. The initialize the array for the local distances, we subtract the remaining distances from d_max for each element of the array. By doing that, the distances of the string with the highest remaining distance will always be initialized with 0. So it is more likely that the distance to that string will improve the most. When the array is increased, we just increase every distance by 1. Here, also different strategies would have been possible. For example, one could just increase one distance by 1. That distance could be chosen randomly or deterministically. For bigger inputs, it might also be possible to increase the distances by more than 1.

3.5 Local Encoding

For the encoding of the local instances, we used PySAT [IMMS18], a toolkit for Python that provides a simple interface to work with state-of-art SAT solvers. We used the Glucose3 solver. The SAT encoding for the local instances is based on a paper by Kelsey and Kotthoff [KK10], where the exact Closest String Problem is formulated as a Constraint Satisfaction Problem.

We have $LS = \{ls_1, ..., ls_n\}$ given, which is a set of n strings of length lb composed of the alphabet A. We also have a distance array `local_d` of length n given, which specifies which output string should not exceed which maximal local distance to the local candidate string.

The general idea behind the encoding is easy: We represent the input strings and the candidate string in a binary way. Then, we have an array `hamming_distances`, which indicates for each string the position that differs from the candidate string. In the end, we just have to check that for each string at most a specific number of positions, which is given by the distance array, differ from the candidate string. In more detail:

First, we want to represent the input strings as binary arrays. So we need an array `binary_strings` of size $n \cdot lb \cdot 4$. The first dimension is the string, the second dimension is the character and the third dimension is the character which is mapped to a number. Let $f = \{0 \mapsto A, 1 \mapsto C, 2 \mapsto G, 3 \mapsto T\}$ be a mapping from numbers to the input alphabet. Then for all n' in $0 \leq n' < n$, n' in $0 \leq lb' < lb$, and k in $0 \leq k < 4$:

$$\texttt{binary\_strings}[n', lb', k] = \begin{cases} 1 & \text{if string } ls_{n'} \text{ at position } lb' \text{ is equal to } f(k); \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, we have a matrix `candidate_string` of size $lb \cdot 4$, which is the binary representation of the candidate string. One constraint is that exactly one of the four characters at one position is true.

We also have the array `hamming_distance` of size $n \cdot lb$, which specifies for all local strings the positions that differ from the candidate string. For every n' in $0 \leq n' < n$ and every lb' in $0 \leq lb' < lb$ we have:

$$\texttt{hamming\_distance}[n', lb'] = \begin{cases} 1 & \text{if } \texttt{binary\_strings}[n', lb'] \neq \texttt{candidate\_string}[lb']; \\ 0 & \text{otherwise.} \end{cases}$$

In the end, we just have to ensure that for each string at most a specific number of positions, which is given by the distance matrix, differ from the candidate string. For all n' in $0 \leq n' < n$:

$$\sum_{i=0}^{lb} (\texttt{hamming\_distance}[n'][i]) \leq \texttt{local\_d}[n']$$

To implement the cardinality constraint $\leq$, we proceed as follows: Assume we have a set of literals given and want to bound them such that at most b of them are true. Then, we search for all possible combinations to pick $b+1$ literals out of the given set of literals. For each of the combinations, we create a new clause, which contains all literals in the clause in their negated form. As an example, assume we have the four literals $\{a, b, c, d\}$ and want to have at most 2 of them true. Our implementation finds all possible combinations to pick $2+1=3$ literals out of the four given literals. Here, it would find the combinations $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, and $\{b, c, d\}$. The resulting formula would be $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg d)$. It is easy to see, that not more than 2 of the used variables can be true to satisfy the formula.

3.6 Pseudocode

The pseudocode over the whole SLIM process is given in Algorithm 3.1. As input, the input strings $s_1, ...s_n$ and an initial candidate string from any heuristic are given. The code runs until the stopping criterion is reached. This can, for example, be a global timeout or a certain number of iterations.

The function `choose_subsequence(current_cs, strings)` returns the indices chosen by the local selection strategy, which is described in Section 3.3. `local_distances` is an array of size n that specifies the local distances as described in Section 3.4. The function `slim(subsequence_indices, strings, local_distances)` is the local SAT encoding as described in Section 3.5. We store the results of the local SAT encoding in the variable `local_cs`. If the SAT solver cannot find a solution, the distance array `local_distances` is increased and the loop repeats. Otherwise, we replace the indices used for the local instance in the string `cs` with `local_cs`. Then, we check whether the new candidate solution `new_cs` is better than the previous candidate solution. If that is the case, we replace the old solution. The implemented program can be found in the file `slim.py` at https://www.ac.tuwien.ac.at/files/resources/software/csp_slim_code.zip.

Algorithm 3.1: SLIM process

```
Input : n input strings of length m; initial candidate string of length m
strings[n] ← array of input strings;
current_cs ← initial candidate string from heuristic;
while stopping criterion not reached do
    subsequence_indices ← choose_subsequence(current_cs,
     strings);
    initialize local_distances[n];
    solved ← False;
    while not solved do
        local_cs ← slim(subsequence_indices, strings,
         local_distances);
        if local_cs is empty then
            increase local_distances;
        else
            solved ← True;
            new_cs ← replace current_cs at the indices given by
             subsequence_indices with local_cs;
            if new_cs is better that current_cs then
                current_cs ← new_cs;
            end
        end
    end
end
```

CHAPTER 4

Experimental Results

4.1 Benchmark instances

To the best of the author's knowledge, there are no commonly used and publicly available benchmark instances to compare the results for different algorithms for the CSP with alphabet size 4. Most other authors just create random instances of their own. There are also real DNA sequences of various species available online from the National Center for Biotechnology Information, NCBI (`https://www.ncbi.nlm.nih.gov/gene/`). First, we thought about using real DNA sequences for the experiments, but since we are not experts in Genetics, we did not know which datasets were useful to compare. Furthermore, not all DNA strings have the same length, which is an input constraint for the CSP. Of course, truncating strings would have been possible, but an offset by one would already change the results drastically. So we decided to just use strings, which were randomly generated as follows: Given the number of strings n, the length of the strings m, and the input alphabet $A = \{A, C, G, T\}$ we randomly selected a character from the set A for every position in the resulting string. The program used for generating random strings called `create_random_strings.py` and can be found at `https://www.ac.tuwien.ac.at/files/resources/software/csp_slim_code.zip`. The randomly generated benchmark instances for the following experiments can be found at `https://www.ac.tuwien.ac.at/files/resources/instances/csp_slim.zip`.

4.2 Experimental Setup

The experiments have been conducted on a server with 16 GB RAM and an Intel Xeon Processor (Cascadelake) with 4 cores and a CPU clock rate of 2494.120 MHz. The used Python version was Python 3.6.9.

4.3 Tuning SLIM

There are different parameter settings that can be tuned in order to improve the performance of SLIM. These parameters include the following:

- Budget: Number of positions in the local instances. The optimal value may depend on the number of input strings.
- Local timeout: How long does it take the solver to run in one iteration? When the SAT encoding does not find a satisfiable solution after a certain time it is likely that this instance is not solvable and thus, SLIM can stop earlier.
- Selection strategy: How are the positions chosen that are improved in one iteration step?
- Initialization and Update Strategy for the Distance Array: How is the distance array initialized? How is it increased after we found an unsolvable solution?

In this Master's thesis, we only focused on the local timeout and budget. The used selection strategy and the initialization and update strategy for the distance array are described in Sections 3.3 and 3.6.

4.3.1 Experiment 1 - Local Timeouts

The goal of this experiment was to find a suitable local timeout for different combinations of the number of strings n and the budget lb. For n we used the values 5, 10, 20, 30, 50, 75, and 100. For lb we used values from 8 to 18. For each combination of n and lb, we randomly generated local instances with n strings of length lb. Since we did not know in advance for which maximum distances such a local instance is satisfiable or unsatisfiable, all instances started with a uniform distance array. Whenever the local instance was not solvable with the given distance array, one randomly picked distance in this array was increased by 1. Otherwise, one randomly picked distance in the array was decreased by 1. This was done iteratively until we found at least 10 distance matrices for which the instance was satisfiable and at least 10 distance matrices for which the instance was unsatisfiable. For each execution of the SAT encoding the time was measured. This was done with 50 different instances. At the end, we had the running times for at least 500 satisfiable instances and 500 unsatisfiable instances for each combination of n and lb.

Table 4.1 shows the average running times for the satisfiable instances. The different columns represent the different numbers of strings. The rows represent the different budgets. For example, it took on average 0.7075 seconds to solve a satisfiable instance with 10 strings of length 12. "-" means that not for all 50 instances 10 satisfiable and 10 unsatisfiable instances could be solved within a timeout of 10 minutes. For the empty cells, the local times were not measured for those combinations. Table 4.2 shows the average running times for the unsatisfiable instances. Comparing with Table 4.1, one can

lb \ n	**5**	**10**	**20**	**30**	**50**	**75**	**100**
8	0.0253	0.0460	0.0857	0.1277	0.2021	0.2941	0.4011
9	0.0334	0.0590	0.1120	0.1697	0.2971	0.4831	0.7617
10	0.0462	0.0853	0.2124	0.5152	1.5366	1.9079	1.8414
11	0.0739	0.1860	1.4206	2.4584	-	-	-
12	0.1339	0.7075	3.2805	-	-	-	-
13	0.3038	2.2216	-				
14	0.8704	-	-	-			
15	2.3563	-	-				
16	-	-	-				
17	-	-					
18	-	-					

Table 4.1: Average running times (in seconds) for solving satisfiable local instances. The rows indicate the budgets and the columns indicate the number of strings.

lb \ n	**5**	**10**	**20**	**30**	**50**	**75**	**100**
8	0.0274	0.0514	0.0982	0.1502	0.2397	0.3482	0.4903
9	0.0380	0.0737	0.1707	0.2759	0.6216	1.0563	1.6471
10	0.0578	0.1427	0.4761	1.5310	4.0490	4.4029	3.6103
11	0.1135	0.3970	3.9715	8.2663	-	-	-
12	0.2283	1.9135	12.419	-	-	-	-
13	0.5794	7.5763	-				
14	2.0559	-	-	-			
15	7.0091	-	-				
16	-	-	-				
17	-	-					
18	-	-					

Table 4.2: Average running times (in seconds) to find out that a local instance is unsatisfiable. The rows indicate the budgets and the columns indicate the number of strings.

see that for all combinations, the average running time of unsatisfiable instances is a bit longer than the average time for unsatisfiable instances. However, the difference is for most values in the same order of magnitude.

Since the local timeout should be chosen such that most of the instances can be solved, we decided to not consider the average running time, but the running time in the 90th percentile. This is the running time such that 90% of the satisfiable instances could be solved within that timeout. The corresponding values can be found in Table 4.3.

lb \ n	**5**	**10**	**20**	**30**	**50**	**75**	**100**
8	0.0658	0.0689	0.1264	0.1368	0.2564	0.3503	0.4726
9	0.0699	0.0838	0.1393	0.2532	0.4453	0.7932	1.2910
10	0.0754	0.1337	0.3833	1.1326	3.3889	3.9233	3.8386
11	0.1288	0.3185	3.4788	6.2246	-	-	-
12	0.2020	1.4712	8.6212	-	-	-	-
13	0.4643	6.0085	-				
14	1.5818	-	-	-			
15	5.0410	-	-				
16	-	-	-				
17	-	-					
18	-	-					

Table 4.3: Running times (in seconds) for solving 90% of the satisfiable local instances. The rows indicate the budgets and the columns indicate the number of strings.

Figure 4.1 shows the values from Table 4.3 in a more visual way. The growth of the running time seems to be exponential in relation to the budget. One can see that the fewer strings there are, the higher the budget can be until solving the local instances becomes infeasible. Interestingly, the running times for instances with 50, 75, and 100 strings seem to be relatively similar. With a budget of 10, the instances with 100 strings even have a lower local running time than the instances with 75 strings. This might be due to outliers.

The program for measuring the running times, as well as all measured times, can be found in the folder `local_timeouts` at `https://www.ac.tuwien.ac.at/files/resources/instances/csp_slim.zip`.

4.3.2 Experiment 2 - Budgets

In this experiment, we wanted to test which budget lb is suitable for which number of strings n. As a heuristic, we implemented a simple greedy algorithm as described in Section 3.2.1. Because of the greedy manner, this algorithm had a running time of less than one second for each of the benchmark instances we used in this experiment.

We conducted the experiment with benchmark instances of different numbers of strings with $n \in \{5, 10, 20, 30, 50, 75, 100\}$ and different lengths of strings. We also used different budgets. We only used budgets for which we got feasible results in Section 4.3.1. For this experiment, we used a uniform local timeout of 60 seconds. This generous timeout was compared with other timeouts in Section 4.3.3. For each combination of a number of strings n, length of strings m, and budget bl, we executed 20 benchmark instances each. The benchmark instances were generated randomly. For each of the benchmark instances, we used a global timeout of 10 minutes.

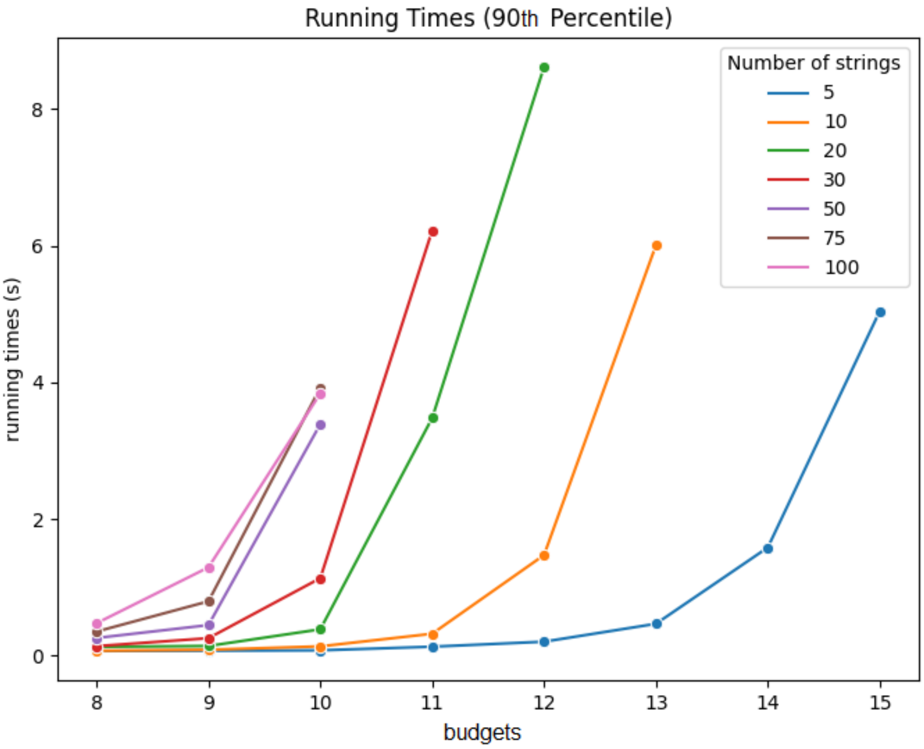
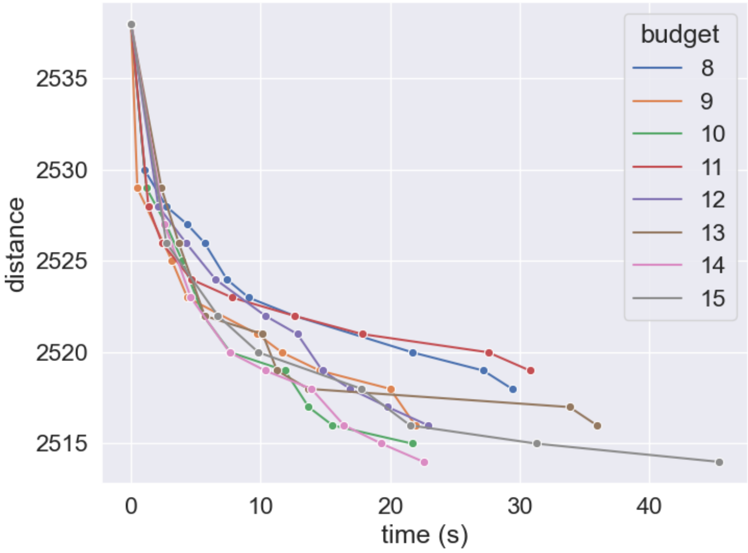


Figure 4.1: Running times in seconds for solving 90% of the satisfiable local instances. The x-axis indicates the different budgets.

First, let us have a look at the 20 benchmark instances with 5 strings of length 5,000. The greedy algorithm gave the initial candidate strings with an average distance of 2,560.8 and a standard deviation of 21.8. We used SLIM with different budgets from 8 to 15 to improve the results. Figure 4.2 shows the improvements with SLIM on two different benchmark instances with 5 strings of length 5,000. One can see, that the original distances we got from the greedy heuristic are 2,538 and 2,590. The first improvements were found after only a few seconds. A bit later, only some more small improvements were found. The last improvement was found after 45.34 seconds resp. 52.4 seconds. After that, no more improvements were found by SLIM, even though it was running for 10 minutes.

When we compare the results of the 20 benchmark instances with $n = 5$ and $m = 5,000$, we can see that SLIM improved the distances of the initial candidate string on average by 27.7. The boxplot in Figure 4.3 shows the achieved improvements for the eight different budgets. Interestingly, we cannot see significant differences between the different budgets.

Also for the other combinations of n and m the different budgets do not show big



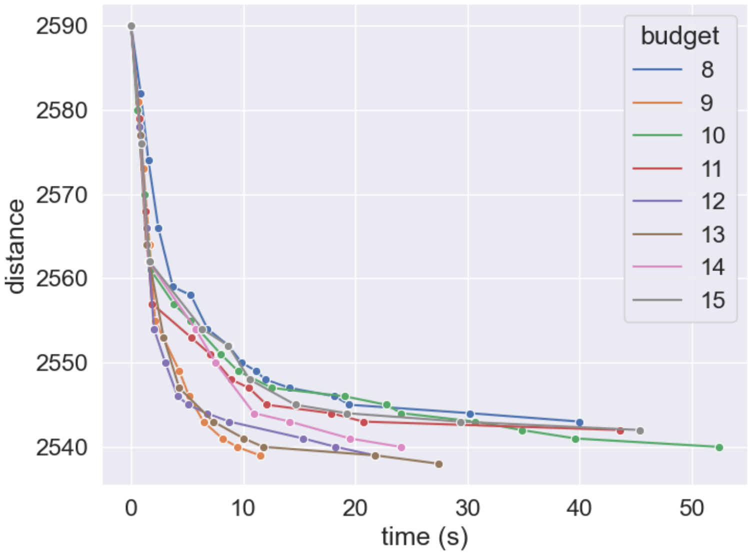


Figure 4.2: The distance improvements of SLIM with different budgets on two different benchmark instances with 5 strings of length 5,000. Even though SLIM was running for 10 minutes, no more improvements were found after 45.34 resp. 52.4 seconds.

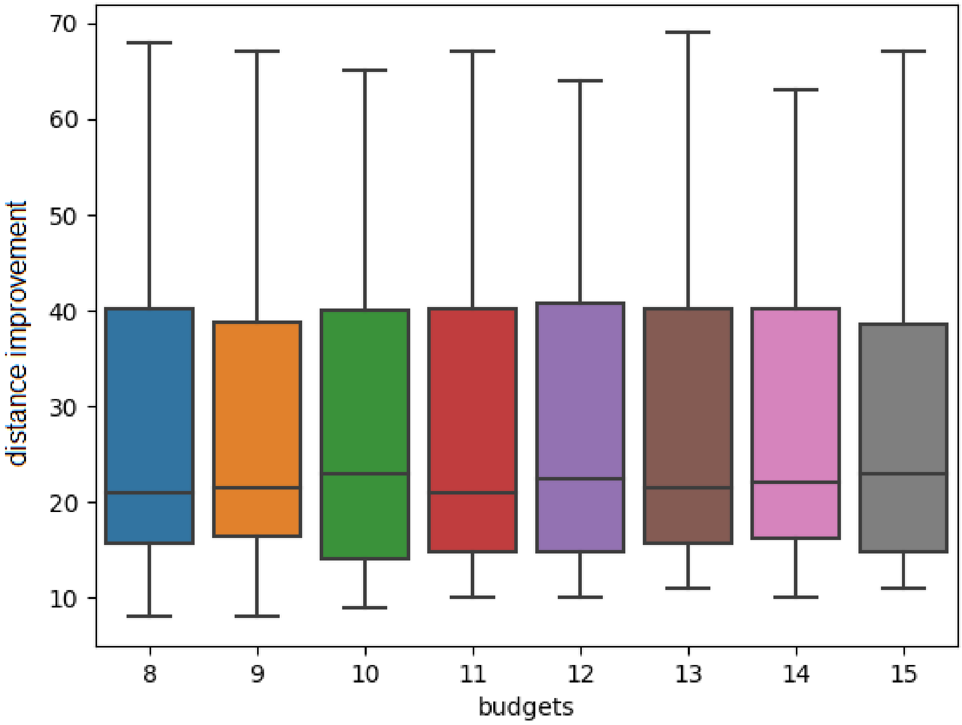
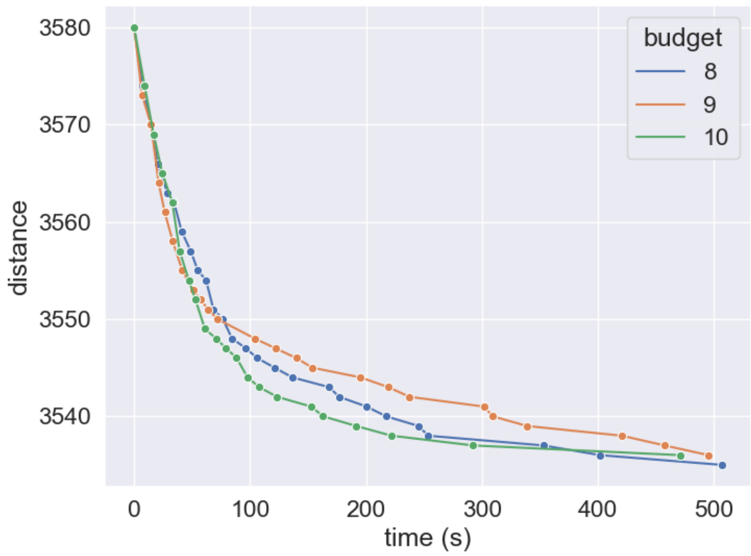


Figure 4.3: Boxplots on the distance improvement for 20 benchmark instances with 5 strings of length 5,000 for different budgets

differences in the improvements. Figure 4.4 shows the distance improvements for two different benchmark instances with $n = 100$ and $m = 5,000$. What is notable is that for benchmark instances with more strings, SLIM was running longer until it found the last improvement. Overall, the 20 benchmark instances with $n = 100$ and $m = 5,000$ have an initial distance of 3,457.7, and SLIM could improve that by 41.7 on average.

Figure 4.5 shows the improvements on two of the benchmark instances with $n = 5$ and $m = 100,000$. The benchmark instance on the top has an initial distance of 50,630 and improves the distances by 143 on average over the different budgets. It is interesting to have a closer look at the shapes of the curves. In some parts of the graphs, we have an almost linear line. This is because the improvement per step is bounded by the budget. With a higher budget one can potentially make a higher improvement in every step. Afterward, it is harder to find improvements. After 600 seconds the curve is already rather flat, but with a higher timeout, the results might improve a little bit more.

The findings of this experiment are that SLIM can indeed improve the initial candidate strings from a simple greedy heuristic. The number of strings and the length of strings



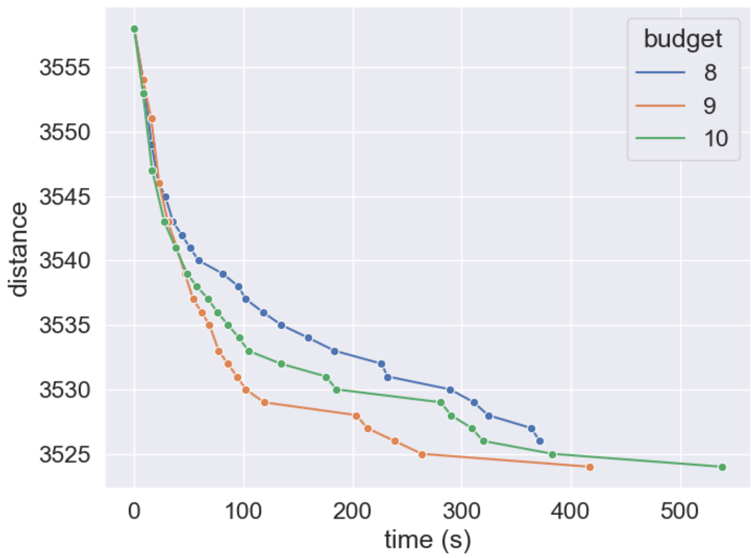
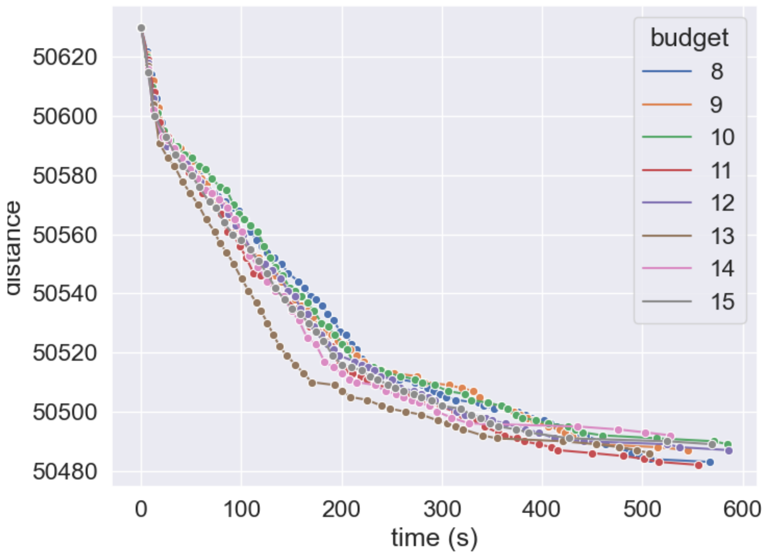


Figure 4.4: Improvement of distance over time using SLIM with different budgets on two different benchmark instances with 100 strings of length 5,000



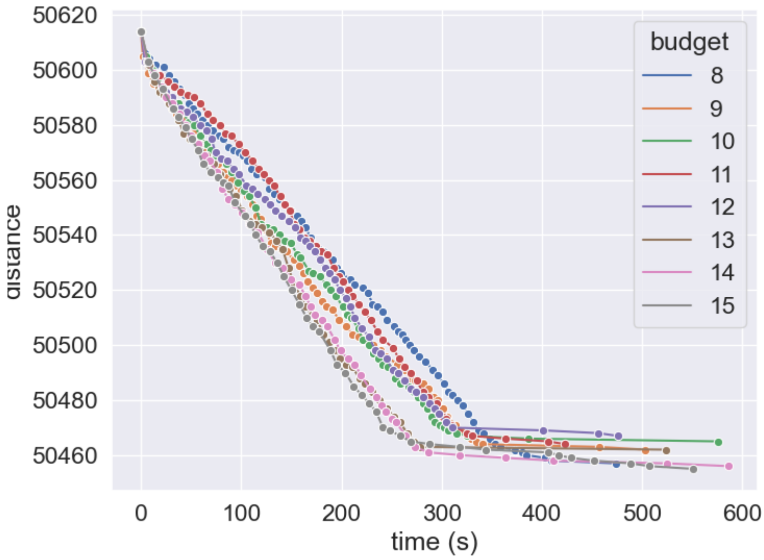


Figure 4.5: Improvement of distance over time using SLIM with different budgets on two different benchmark instances with 5 strings of length 100,000

have an impact on the running time until SLIM finds the last improvement. For our used benchmark instances the different budgets do not yield a significant difference in the performance of SLIM.

The benchmark instances, initial solutions, final solutions and log files for experiments 2 and 3 can be found in the folder `budgets` at https://www.ac.tuwien.ac.at/files/resources/instances/csp_slim.zip.

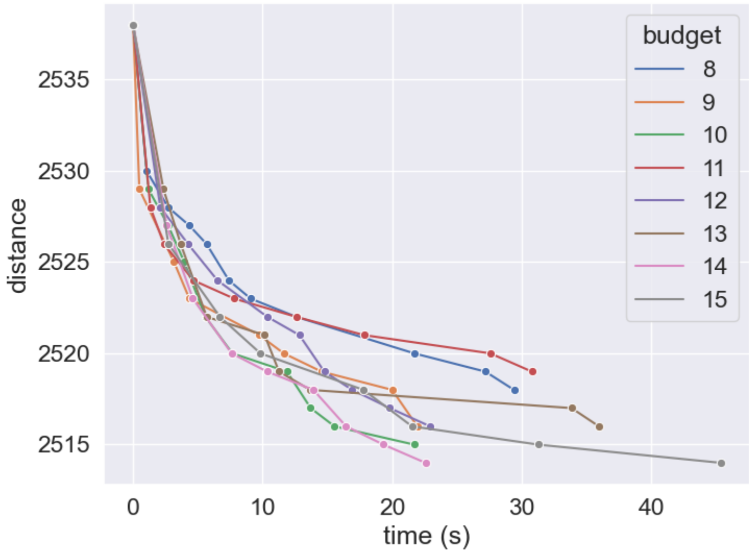
4.3.3 Experiment 3 - Comparison of SLIM with Different Local Timeout

In this experiment, we compared the results of SLIM using adaptive local timeouts. For the experiment in Section 4.3.2 a generous timeout of 60 seconds was used. From the experiment in Section 4.3.1 we already know that that is much higher than needed. So we limited the local timeout to a value such that 90 percent of satisfiable instances with the given budget and number of strings can be solved. See Table 4.3 for the adaptive timeouts depending on the number of strings n and the budget lb. Figure 4.6 compares the improvement of the distance for one specific benchmark instance of size with 5 strings of length 5,000. The graph on the top shows SLIM with a local timeout of 60 seconds. For the graph on the bottom, an adaptive local timeout based on the 90th percentile, according to Table 4.3, was used. Figure 4.7 compares the improvement of the distance for one benchmark instance with 100 strings of length 5,000 and Figure 4.8 does that for a benchmark instance with 5 strings of length 100,000. For all of these three comparisons, one can see that the improvements are almost the same, but the running times show clear differences. When a smaller local timeout was used, SLIM found the improvements quicker.

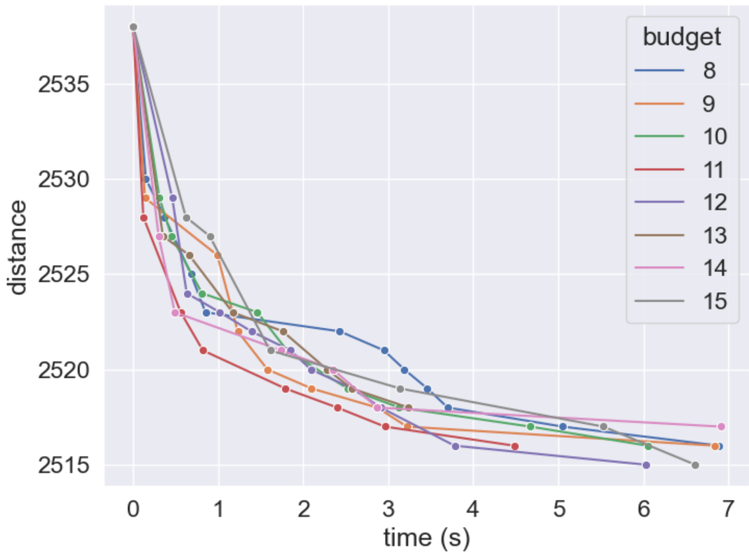
To support this observation, we summarized the result of all 20 benchmark instances in Tables 4.4 and 4.5. Table 4.4 compares the results of benchmark instances with $m = 5,000$ over different n. Table 4.5 compares the results of benchmark instances with $n = 5$ over different m. The tables show the average initial distances and the average improvements with SLIM using a local timeout of 60 seconds and the local timeout based on the 90th percentile according to Table 4.3. The average improvement for both local timeouts was almost the same. However, one can see differences in the running time until the last improvement was found. On average, SLIM with the fixed timeout needed approximately 4.4 times as long to find the solution.

4.3.4 Discussion

In this section, we could see that it is indeed possible to improve an initial candidate string from our simple greedy heuristic. We have seen that using an adaptive local timeout, i.e. such that 90% of the satisfiable instances can be solved, led to almost the same improvement but in a shorter running time. Thus, it makes sense to use a local timeout, especially if the global timeout is bounded.

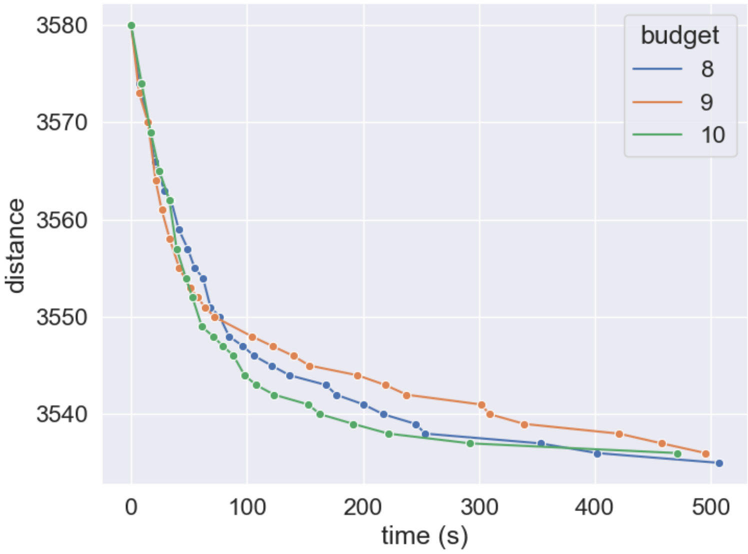


(a) Local timeouts of 60 seconds

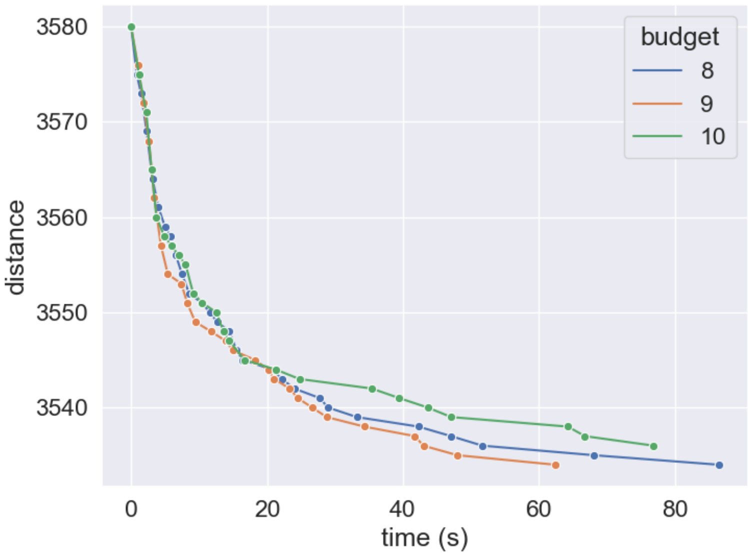


(b) Adaptive local timeouts according to Table 4.3

Figure 4.6: The distance improvements of SLIM with different budgets on a benchmark instance with 5 strings of length 5,000. Observe the different scales for the x-axis

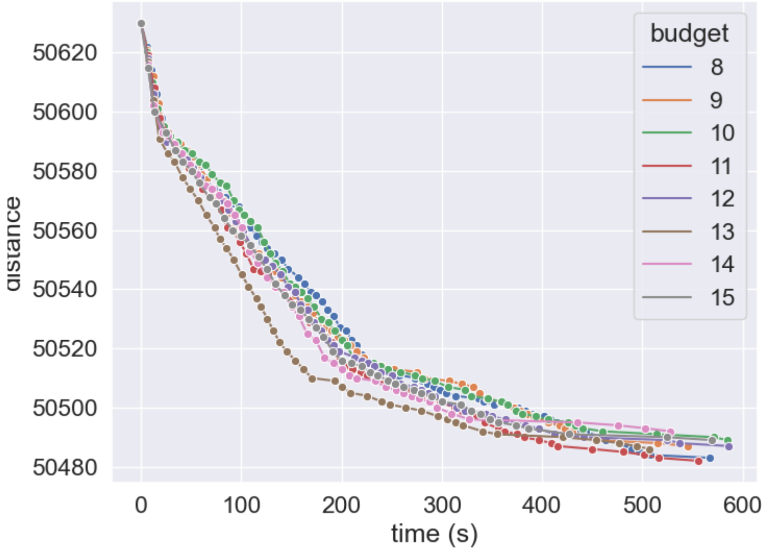


(a) Local timeouts of 60 seconds

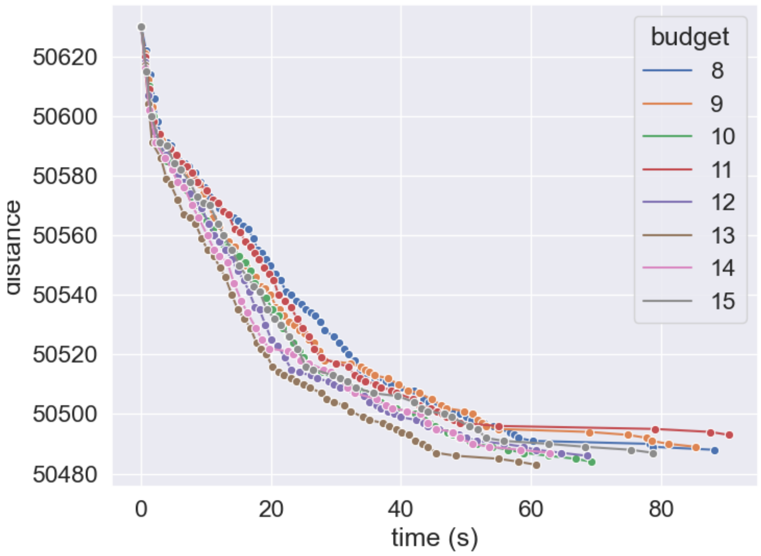


(b) Adaptive local timeouts according to Table 4.3

Figure 4.7: The distance improvements of SLIM with different budgets on a benchmark instance with 100 strings of length 5,000. Observe the different scales for the x-axis



(a) Local timeouts of 60 seconds



(b) Adaptive local timeouts according to Table 4.3

Figure 4.8: The distance improvements of SLIM with different budgets on a benchmark instance with 5 strings of length 100,000. Observe the different scales for the x-axis

number of strings	5	10	20	30	50	75	100
avg. initial dist.	2,560.80	2,954.10	3,218.20	3338.95	3,457.70	3524.65	3,567.80
avg. impr. (fixed LTO)	27.69	34.17	36.31	39.69	41.72	38.27	37.82
avg. impr. (adaptive LTO)	27.76	34.12	36.51	40.11	42.40	38.30	37.88
runtime (fixed LTO) (s)	38.44	80.04	130.08	176.99	273.34	367.51	418.38
runtime (adaptive LTO) (s)	10.98	16.20	23.83	31.20	49.23	75.26	87.36

Table 4.4: Comparison of SLIM improvements using either a fixed local timeout or adaptive local timeout. The table shows the average initial distance, average SLIM improvement, and average running time (in seconds) for benchmark instances with different numbers of strings of length 5,000. The average SLIM improvement is given for a fixed local timeout (fixed LTO) of 60 seconds and an adaptive local timeout (adaptive LTO) according to Table 4.3. The average running time is the time until the last improvement was found. It is also given for a fixed and an adaptive local timeout

length of strings	1,000	5,000	10,000	20,000	50,000	100,000
avg. initial dist.	523.85	2,560.80	5,100.70	10,174.25	25,325.50	50,566.50
avg. impr. (fixed LTO)	15.06	27.69	39.71	65.42	84.75	116.58
avg. impr. (adaptive LTO)	15.40	27.76	39.44	65.10	84.64	117.49
runtime (fixed LTO) (s)	57.26	38.44	58.9	112.28	222.77	454.78
runtime (adaptive LTO) (s)	99.97	10.98	15.34	18.84	37.82	74.75

Table 4.5: Comparison of SLIM improvements using either a fixed local timeout or adaptive local timeout. The table shows the average initial distance, average SLIM improvement, and average running time (in seconds) for benchmark instances with 5 strings of different lengths. The average SLIM improvement is given for a fixed local timeout (fixed LTO) of 60 seconds and an adaptive local timeout (adaptive LTO) according to Table 4.3. The average running time is the time until the last improvement was found. It is also given for a fixed and an adaptive local timeout

In contrast to the author's expectations, the results of the experiments suggest that the used budget does not make a difference, as long as it is not so high that solving the local instances becomes infeasible.

4.4 SLIM on a State-of-the-Art Algorithm by Tanaka

To answer the question of whether SLIM can improve a state-of-the-art heuristic, we used a heuristic algorithm based on Lagrangian relaxation by Tanaka [Tan12]. This algorithm is described in more detail in Section 3.2.2. One big advantage of this algorithm is that it gives a lower bound for the distance of the closest string. By that, it can sometimes know whether the solution is optimal.

4.4.1 Results for Initial Candidate Strings

We randomly created benchmark instances for different numbers of strings with $n \in \{5, 10, 20, 30, 50, 75, 100\}$ and different lengths of strings from 5,000 to 50,000,000. For each examined combination, we created 20 benchmark instances. For each benchmark instance, the algorithm got a timeout of 60 minutes. The results were divided into the following categories:

- **optimal:** The algorithm found a optimal solution
- **best:** The algorithm terminated because it found a currently best solution and could not improve it for a certain number of iterations
- **timeout:** The algorithm was terminated by the given timeout of 60 minutes. Within that time at least one solution was found
- **no solution:** The algorithm was terminated by the given timeout of 60 minutes. Within that time not a single solution was found
- **error:** The algorithm was aborted because of a segmentation fault

The results can be viewed in Table 4.6. Table 4.7 shows the average running times, excluding the benchmark instances that ran into a timeout. It can be seen that for several benchmark instances, especially those with small n, the optimal solution was found within some seconds. For example, all benchmark instances with 10 strings of lengths from 5,000 to 100,000 could be solved optimally within less than 3.5 seconds on average. On the other hand, no benchmark instance with 75 or 100 strings could be solved optimally. The algorithm could find a solution for most of the tested benchmark instances. It only could not find a solution for some of the benchmark instances instances with a string length of more than 5 million characters. In many of the benchmark instances with 10 strings of length 20,000,000 and 50,000,000, a segmentation fault occurred. Nevertheless, the results were quite impressive. Especially because the optimality could be confirmed for many instances. All benchmark instances and resulting initial strings for this experiment can be found at https://www.ac.tuwien.ac.at/files/resources/instances/csp_slim.zip in the folder tanaka_initial_solutions.

4.4.2 SLIM Improvement

As shown in Section 4.4.1, Takana's algorithm already achieves great results. Out of the 1,160 benchmark instances, it could find the optimal solution for 540. For 218 of the instances, the currently best solution was returned after an internal stopping criterion was reached. For those instances, we do not know, whether it is possible to improve them or whether they are already optimal. For 288 of the instances, at least one solution was found, before the algorithms were terminated after a timeout. For 114 instances the algorithm did not even return a single solution. Either because of the timeout or because of an error.

For SLIM, it does not make sense to run on instances that are already optimal. It can also not improve the benchmark instances for which no solution was found, because it needs an initial solution. So we let SLIM run on the $218 + 288 = 506$ remaining instances, for which we have at least one solution, which is not already shown to be the optimal one.

We used a budget of 10 and an adaptive local timeout according to Table 4.3 plus a buffer of 0.1 seconds. As the initial solution, we used the first solution that Tanaka's algorithm returned. As global timeout, we took 60 minutes minus the time until the first solution was found. Out of the solutions of the 506 benchmark instances, SLIM could improve 21. One has to mention that some of the benchmark instances were already very close to the optimal solution or were even optimal. So there was not much space for improvement. The improvements of these 21 benchmark instances are shown in Table 4.8.

To ensure comparability, Tanaka's algorithm did not had to stop after it found the first solution but had until the timeout to improve its initial solution. For five out of the 21 instances mentioned above, Tanaka's algorithm was able to improve the results in the remaining time. For the remaining 16 instances, SLIM could indeed bring an improvement.

At https://www.ac.tuwien.ac.at/files/resources/instances/csp_slim.zip one can find the folder tanaka_improvement, which contains the 506 benchmark instances and corresponding initial solutions used for this experiment. Furthermore, there are the solutions for the 21 benchmark instances improved by SLIM and log files.

4.4.3 Beyond the Limits

As one can see in Section 4.4.1, for Tanaka's algorithm it was hardly possible to solve benchmark instances with huge strings of length more than 10,000,000. For the 20 benchmark instances of 5 strings with a length of 50,000,000 and the 20 benchmark instances of 10 strings with a length of 10,000,000, it was not able to find a single solution.

m \ n	5				10				30			
	NS	TO	B	OPT	NS	TO	B	OPT	NS	TO	B	OPT
5,000	0	0	0	20	0	0	0	20	0	0	8	12
10,000	0	0	1	19	0	0	0	20	0	0	10	10
20,000	0	0	1	19	0	0	0	20	0	0	6	14
50,000	0	4	0	16	0	0	0	20	0	5	0	15
100,000	0	3	0	17	0	0	0	20	0	8	0	12
200,000	0	6	0	14	0	1	0	19	0	5	0	15
300,000	0	6	0	14	0	1	0	19	0	5	0	15
500,000	0	3	0	17	0	9	0	11	0	3	0	17
1,000,000	0	8	0	12	0	11	0	9	0	6	0	14
2,000,000	0	2	0	18	0	9	0	11				
5,000,000	0	7	0	13	8	7	0	5				
10,000,000	7	3	0	10	20	0	0	0				
20,000,000	19	0	0	1	5	0	0	0				
50,000,000	20	0	0	0	0	0	0	0				

m \ n	50				75				100			
	NS	TO	B	OPT	NS	TO	B	OPT	NS	TO	B	OPT
5,000	0	0	19	1	0	0	20	0	0	0	20	0
10,000	0	0	19	1	0	0	20	0	0	0	20	0
20,000	0	0	15	5	0	0	20	0	0	0	20	0
50,000	0	0	15	5	0	16	4	0	0	20	0	0
100,000	0	17	0	3	0	20	0	0	0	20	0	0
200,000	0	12	0	8	0	20	0	0	0	20	0	0
300,000	0	13	0	7								
500,000	0	11	0	9								
1,000,000	0	7	0	13								

Table 4.6: Results from the heuristic by Tanaka. The rows specify the length of strings in the benchmark instances. The columns specify the result categories for different numbers of strings. The categories are optimal (OPT), best (B), timeout (TO), and no solution (NS). For the empty cells, no experiments were conducted. For many of the benchmark instances with 10 strings of length 20,000,000 and 50,000,000, a segmentation fault occurred. So they are not listed in the table.

m \ n	5	10	30	50	75	100
5,000	0.58	0.02	10.7	21.8	34.2	49.3
10,000	5.57	0.06	42.9	76.4	78.3	120
20,000	22.9	0.59	114	262	263	258
50,000	92.8	1.48	123	385	1031	422
100,000	67.8	14.5	153	440	1651	401
200,000	145	43.3	120	256	1339	303
300,000	105	104	93.2	265		
500,000	5.8	14.9	57.1	111		
1,000,000	18	41	155	297		
2,000,000	47.2	111				
5,000,000	186	304				
10,000,000	359	-				
20,000,000	395	-				
50,000,000	-	-				

Table 4.7: Average running times (in seconds) of the algorithm by Tanaka on different benchmark instances. The rows specify the numbers of strings and the columns specify the length of the strings of the corresponding benchmark instances. If a benchmark instance reached the timeout of 60 minutes, its running time was not taken into account for this table. "-" means that none of the 20 benchmark instances finished before the timeout. For the empty cells, no experiments were conducted.

This is, where the power of SLIM can be shown. Since it can use the solutions of any CSP algorithm as an initial solution, it is more flexible with the size of the input instances. So we used our simple greedy algorithm as described in Section 3.2.1 to give an initial solution.

For the instances with 5 strings, we used a budget of 12. For the instances with 10 strings, we used a budget of 10. Even though the results from the experiment in Section 4.3.3 suggest that local timeouts as small as in Table 4.3 lead to better results, we decided on a local timeout of 1 second. This is because finding the local instances already takes a relatively long time when we have long strings. Compared to selecting the local instances, solving the local instances takes less time and we want to be able to solve almost all local instances.

To be able to deal with such big instances, we made little adjustments to our SLIM programming. First of all, we decided to use a completely random selection strategy. This has the advantage that it works quickly, regardless of the string length. The strategy described in Section 3.3 took about 90 seconds to find one local instance for benchmark instances with a string length of 10,000,000 and 260 seconds for the benchmark instances with a string length of 50,000,000. In some cases, it even led to Memory Errors. Furthermore, we changed the code in such a way that it did not print every intermediate

n	m	initial d	d after SLIM	further improvement by Tanaka
5	50,000	25,170	25,168	25,155
5	50,000	25,207	25,203	25,189
5	200,000	100,746	100,741	100,718
5	200,000	100,828	100,827	-
5	200,000	100,861	100,850	100,812
5	300,000	151,229	151,227	151,222
5	300,000	151,179	151,178	-
5	1,000,000	503,998	503,997	-
5	1,000,000	503,954	503,948	-
5	1,000,000	503,909	503,894	-
5	1,000,000	503,524	503,515	-
5	1,000,000	503,868	503,865	-
5	2,000,000	1,007,394	1,007,384	-
5	2,000,000	1,008,316	1,008,284	-
5	5,000,000	2,519,362	2,519,349	-
5	5,000,000	2,519,639	2,519,627	-
5	5,000,000	2,519,707	2,519,702	-
5	10,000,000	5,039,661	5,039,657	-
5	10,000,000	5,038,136	5,038,134	-
10	500,000	290,075	290,074	-
10	1,000,000	580,390	580,387	-

Table 4.8: The 21 benchmark instances that could be improved by SLIM. The table shows the number of strings n, and the length of strings m of the benchmark instances. The initial distance (initial d) is the distance of the first solution which was given by Tanaka's algorithm. The distance after SLIM (d after SLIM) is the distance of the solution after the SLIM improvement. At the same time, Tanaka's algorithm was still running and might also improved its initial solution. The distance of the new solution is given in the last column of the Table. "-" means that Tanaka's algorithms were not able to further improve its initial solution.

result, since this also takes some minutes.

To be able to compare our SLIM approach to the results from Tanaka's algorithm, we also wanted to have an overall global timeout of 60 minutes. Since generating the initial solutions with the greedy algorithm took approximately 12 minutes, the SLIM part was restricted to a global timeout of 48 minutes.

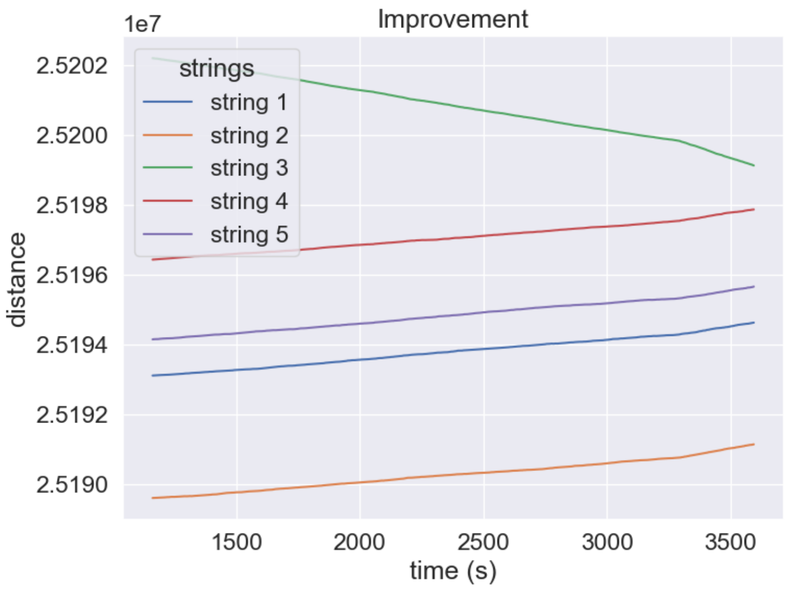
For the benchmark instances with 5 strings of length 50,000,000, the distance of the initial candidate string was on average 25,200,122. SLIM could improve that on average by 1,324.8. One instance was even improved by 3,083. The graphic on the top of Figure 4.9 shows that benchmark instance. One can see how the distances from the currently best string to the five input distances changed over time. At the beginning, the furthest string had a distance of 25,202,219 and the string with the smallest distance had a distance of 25,189,602. Over time, the candidate string got closer to the furthest input string but the distance to the other strings got higher. The maximum distance after the one-hour timeout was 25,199,136, which is an improvement of 3,083. As the figure shows, there is still a difference in the distances. With a higher global timeout, SLIM is likely to find better results. On the graphic on the bottom of the figure, one can see another benchmark instance, where in the end a string was found which has the same distance for three of the input strings.

For the benchmark instances with 10 strings of length 10,000,000, the distance of the initial candidate string was on average 5,805,182. SLIM could improve that on average by 228.1. Here, the average improvement is lower, because some of the instances ran into a memory error after a while. At this point, there might be some more space for improvements in the SLIM code that prevent memory problems. The best improvement for one instance was 1,962, which is shown in Figure 4.10 on the top. It is interesting to see, how the distances of the currently closed string were the same for most of the input strings at the end of the timeout. The benchmark instance on the bottom was aborted earlier because a memory error occurred.

The reworked SLIM version, all the used benchmark instances, the corresponding initial solutions from the greedy heuristic, and the log files for this experiment can be found at `https://www.ac.tuwien.ac.at/files/resources/results/csp_slim.zip` in the folder `beyond_tanaka`.

4.4.4 Discussion

The experiments in this section showed that Tanaka's algorithm is already quite powerful in solving the CSP. Due to the described benchmark instances from the literature, we initially did not think that there exists an algorithm that could solve the CSP with input strings of length more than one million. For example, Gomes et al. [GMPV08] used benchmark instances with up to 30 strings and up to 5,000 characters. Meneses et al. [MLOP04] used 10 to 30 strings of length of 300 to 800 characters. Not even Tanaka [Tan12] tested his own algorithm on strings with more than 5,000 characters.



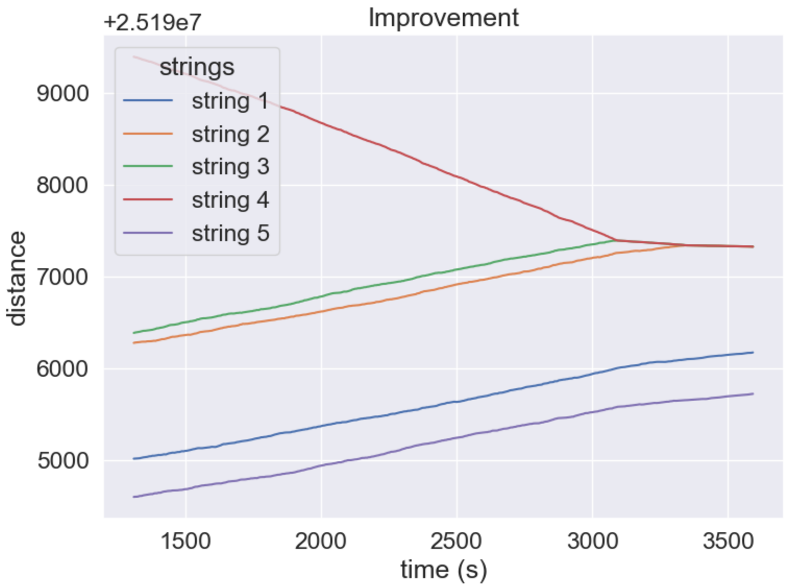


Figure 4.9: The distances of to candidate string to all input strings of two benchmark instances with 5 strings of length 50,000,000. The improvements only started after about 1,300 seconds, because it took some time for the greedy algorithm until the initial solution was created and until SLIM read in the input strings

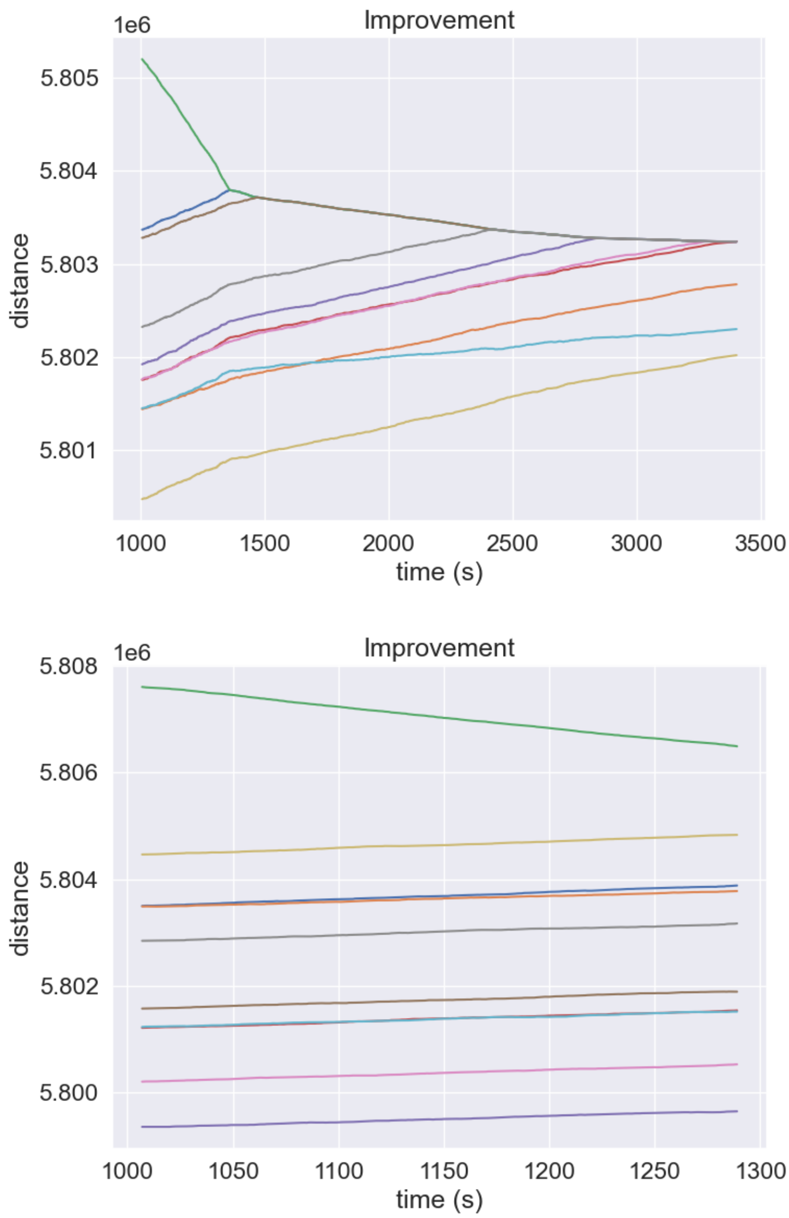


Figure 4.10: The distances of to candidate string to all input strings of two benchmark instances with 10 strings of length 10,000,000. The improvements only started after about 1,000 seconds, because it took some time for the greedy algorithm until the initial solution was created and until SLIM read in the input strings. The benchmark instance on the bottom was aborted earlier because a memory error occurred

One further advantage of this algorithm is that it gives a lower bound for the distance of the closest string. For about half of the used benchmark instances it already gave the optimal solution. Most of them within only a few seconds. That is really impressive for an NP-complete problem and did not leave much room for improvement. Nevertheless, out of the 506 instances, for which Tanaks's algorithm found at least one solution but could not show optimality, our SLIM approach could improve 16 instances - even if only a little.

On the other hand, one big advantage of SLIM is that it works with the initial candidate strings from various heuristics. For benchmark instances that were too big for Tanaka's algorithm to solve, we could just use another heuristic that works better on big input instances and then improve those results with SLIM. In the experiments in Section 4.4.3 SLIM could improve the distances of the initial solutions from our simple greedy algorithm by up to 3,083.

CHAPTER 5

Conclusion

5.1 Summary

In this thesis, we had a look at the SAT-based Local Improvement Method (SLIM) and how to use it for solving the Closest String Problem (CSP). Chapter 3 describes what SLIM is and which design choices we made in the programming. In Chapter 4 we did some experiments to find the best parameter settings and to compare it to another existing heuristic. What was interesting to see is that the choice of the budget does not make a big difference, as long as it is not so high that the SAT encoding becomes infeasible. However, what made a difference was a different local timeout. We did some experiments that measured the times needed to solve the local instances. In another experiment, we took the running times, within which 90% of the solvable instances could be found, as local timeouts. When we compared the results with SLIM results from the same benchmark instances with a higher local timeout of 60 seconds, we could see that the quality of the results was the same, but with the longer timeout, it took approximately 4.4 times as long to find the solution.

Furthermore, we did some experiments that investigated how well SLIM works with a state-of-the-art algorithm by Tanaka [Tan12]. The results showed that Tanaka's algorithm already worked quite well. Since it even offers a lower bound for the benchmark instances, it could show optimality for almost half of the used benchmark instances. So it did not leave much room for improvement. Nevertheless, out of the 506 instances, for which Tanaks's algorithm found at least one solution, but could now show optimality, our SLIM approach could at least improve 16 instances - even if only a little.

One big advantage of SLIM is that it works with the initial candidate strings from various heuristics. For benchmark instances that were too big for Tanaka's algorithm to solve, we could just use another heuristic, like our simple greedy algorithm, that works better on big input instances. Afterward, we could just improve those results with SLIM.

The short answer to our initial research question “Can we use SLIM to improve the state of the art in solving the Closest String Problem?” is that there are already algorithms that work well with the CSP and thus, SLIM could only improve them on a few instances or work better only with some special cases, like very big benchmark instances with input strings with a length of more than 10,000,000 characters.

5.2 Further work

There are many further things which could be investigated in relation to this topic.

First of all, it would be interesting to test SLIM on different benchmark instances. In this thesis, we only used randomly created instances. It also would be interesting to use benchmark instances with real DNA or instances that have a certain structure. For example, Tanaka [Tan12] also used randomly generated instances in which the possibility that C and G are chosen is 36% each, and the probability for A and T is 14% each. This simulates the genome of Streptomyces coelicolor, which is a bacterium whose content of C and G is 72%. Another interesting idea, used by Xu et al. [XP22] for example is to generate benchmark instances for which a maximal distance d is guaranteed. This can be done by first generating an answer string s of length m over the alphabet A. For generating the input strings, a copy of s is used in which d randomly chosen positions were overwritten by another character. Knowing the maximal distance d could help to evaluate different approaches regarding the quality of the results.

Furthermore, for tuning our SLIM process, some more experiments could be made. In our experiments, we only compared a local timeout of 60 seconds to the adaptive local timeout based on the 90th percentile. One could also try out other local timeouts, for example by adding a buffer to the adaptive local timeout. It might also be the case that the optimal local timeout depends on the performance of the used computer. For the budget, it would be interesting to find out whether there is a minimum budget. One could also implement different selection strategies and strategies to initialize and update the distance array and compare the choices against each other. Especially for the selection strategy, there are many possibilities and it might be possible to find underlying structures in the input strings that have a particularly high chance to be good candidates for local improvement. Another question to ask is whether there are good deterministic strategies or if it is always better to include randomness. For the bigger benchmark instances, it might be possible to improve SLIM such that it can deal better with big strings and not get memory problems. What might also be interesting is to compare different SAT solvers, like CaDiCaL 153, for example.

Further research could not only look at SLIM for an input alphabet of size 4 but also look at binary strings or strings with an alphabet size of 20. The CSP on binary strings has applications in Coding Theory. The CSP on strings with an alphabet size of 20 can be used in biology to investigate amino acid sequences.

One big advantage of SLIM is that the local encoding is quite flexible and other variants of the CSP can be easily built in. For example, one could look at the CSP with wildcards. Here, there are some positions in the input strings that are not known. Another variant of SLIM is to not only minimize the maximal distance to the strings but also to maximize the number of occurrences of a certain character. For other algorithms implementing such variants might need major changes, but with SLIM this could be implemented with only a few constraints in the SAT encoding.

List of Figures

List of Tables

List of Algorithms

Acronyms

CNF conjunctive normal form. 6, 7

CSP Closest String Problem. 1–3, 5, 6, 9–11, 15, 32, 34, 39–41, 45

FPT fixed-parameter tractable. 2

GA genetic algorithm. 2

MIP mixed integer programming. 10

SA simulated annealing. 2

SAT propositional satisfiability problem. 3, 6, 7, 9, 11, 13, 16, 39–41

SLIM SAT-based Local Improvement Method. 3, 9, 11, 12, 16, 19–30, 32–34, 36, 37, 39–41, 43, 45, 46

Bibliography

[BHKN14] L. Bulteau, Falk Hüffner, Christian Komusiewicz, and R. Niedermeier. Multivariate Algorithmics for NP-Hard String Problems. *Bull. EATCS*, October 2014.

[BHMW09] Armin Biere, Marijn Heule, H. Maaren, and Toby Walsh. Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. January 2009.

[CLLM94] G. Cohen, S. Litsyn, Antoine Lobstein, and H. Mattson. Covering Radius 1985-1994. *Electrical Engineering and Computer Science - Technical Reports*, November 1994.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, NY, 1999.

[DL18] Marcello Dalpasso and Giuseppe Lancia. New Modeling Ideas for the Exact Solution of the Closest String Problem. In Mourad Elloumi, Michael Granitzer, Abdelkader Hameurlain, Christin Seifert, Benno Stein, A Min Tjoa, and Roland Wagner, editors, *Database and Expert Systems Applications*, Communications in Computer and Information Science, pages 105–114, Cham, 2018. Springer International Publishing.

[FBHS23] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The Silent (R)evolution of SAT. *Communications of the ACM*, 66(6):64–72, May 2023.

[FL97] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, April 1997.

[FLS17] Johannes K. Fichte, Neha Lodha, and Stefan Szeider. Sat-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 401–411, Cham, 2017. Springer International Publishing.

[FP10] Simone Faro and Elisa Pappalardo. *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*. January 2010. Pages: 381.

[GJL99] Leszek Gasieniec, Jesper Jansson, and Andrzej Lingas. *Efficient Approximation Algorithms for the Hamming Center Problem*. January 1999. Journal Abbreviation: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms Pages: 906 Publication Title: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms.

[GMPV08] Fernando C. Gomes, Cláudio N. Meneses, Panos M. Pardalos, and Gerardo Valdisio R. Viana. A parallel multistart algorithm for the closest string problem. *Computers & Operations Research*, 35(11):3636–3643, November 2008.

[IMMS18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python Toolkit for Prototyping with SAT Oracles. pages 428–437. June 2018.

[KK10] Tom Kelsey and Lars Kotthoff. The Exact Closest String Problem as a Constraint Satisfaction Problem, May 2010. arXiv:1005.0089 [cs].

[KLLM+03] J. Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, August 2003.

[LHS05] Xuan Liu, Hongmei He, and Ondrej Sýkora. Parallel Genetic Algorithm and Parallel Simulated Annealing Algorithm for the Closest String Problem. In Xue Li, Shuliang Wang, and Zhao Yang Dong, editors, *Advanced Data Mining and Applications*, Lecture Notes in Computer Science, pages 591–597, Berlin, Heidelberg, 2005. Springer.

[LLHM11] Xiaolan Liu, Shenghan Liu, Zhifeng Hao, and Holger Mauch. Exact algorithm and heuristic for the closest string problem. *Computers & operations research*, 38(11):1513–1520, 2011.

[LOS19] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A sat approach to branchwidth. *ACM Transactions on Computational Logic (TOCL)*, 20(3):1–24, 2019.

[LW67] G. N. Lance and W. T. Williams. A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. *The Computer Journal*, 9(4):373–380, February 1967.

[MLOP04] Claudio Meneses, Zhaosong Lu, Carlos Oliveira, and Panos Pardalos. Optimal Solutions for the Closest String Problemvia Integer Programming. *Informs Journal on Computing - INFORMS*, 16:419–429, November 2004.

[Sch22] André Schidler. Sat-based local search for plane subgraph partitions (cg challenge). In *38th International Symposium on Computational Geometry (SoCG 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[Tan12] Shunji Tanaka. A heuristic algorithm based on lagrangian relaxation for the closest string problem. *Computers & operations research*, 39(3):709–717, 2012.

[XP22] Shirley Xu and David Perkins. A Heuristic Solution to the Closest String Problem Using Wave Function Collapse Techniques. *IEEE Access*, 10:115869–115883, 2022. Conference Name: IEEE Access.