FEDERAL UNIVERSITY OF AMAZONAS - UFAM
Institute of Computing - ICOMP
Post-graduate Program in Informatics - PPGI

COMBINATORIAL APPROACHES FOR THE CLOSEST STRING PROBLEM

Omar Latorre Vilca

A thesis submitted to the Post-graduate Program in Informatics of the Institute of Computing of the Federal University of Amazonas in partial fulfillment of requirements for the degree of Doctor of Science. Concentration area: Informatics.

Advisor: Eduardo Luzeiro Feitosa, D.Sc.

August 2019
Manaus - AM

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Vilca, Omar Latorre
V699c Combinatorial Approaches for the Closest String Problem / Omar Latorre Vilca. 2019
106 f.: il. color; 31 cm.

Orientador: Eduardo Luzeiro Feitosa
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. Combinatorial Optimization. 2. Integer Programming. 3. Heuristics. 4. Consensus String. I. Feitosa, Eduardo Luzeiro II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Combinatorial Approaches for the Closest String Problem"

OMAR LATORRE VILCA

Tese de Doutorado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Eduardo Luzeiro Feitosa - PRESIDENTE

Prof. Juan Gabriel Colonna - MEMBRO INTERNO

Profa. Fabiola Guerra Nakamura - MEMBRO EXTERNO

Profa. Renata da Encarnação Onety - MEMBRO EXTERNO

Prof. Joaquim Maciel da Costa Craveiro - MEMBRO EXTERNO

Manaus, 23 de Agosto de 2019

Abstract of thesis presented to IComp/UFAM as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

COMBINATORIAL APPROACHES FOR THE CLOSEST STRING PROBLEM

Omar Latorre Vilca

August/2019

Advisor: Eduardo Luzeiro Feitosa

Program: Postgraduate in Informatic

The closest string problem (CSP) that arises in computational molecular biology and coding theory is to find a string that minimizes the maximum Hamming distance from a given set of strings, the CSP is an NP-hard problem. The main aim of this work is to propose exact methods for this problem, for this purpose, we characterize special cases for this problem with emphasis in the number of strings. Until now our contribution is: linear-time algorithms for CSP with up to three strings and for four binary strings, in addition to an heuristic greedy algorithm and a recursive exact algorithm for CSP for the general case. Furthermore, for each proposed algorithm formal proofs will be presented, also numerical experiments will show the effectiveness of the proposed algorithms.

Keywords: Combinatorial Optimization, Integer Programming, Heuristics.

Contents

Acknowledgements

I have always believed that the satisfaction that comes from doing what you want, in the way you want, is a key point in the life of everyone. I think it is the greatest source of inspiration and encouragement to really try to give the best. Hoping that my thesis will be just a start, I am really proud of the result, because this is the kind of work that I wanted to do.

This has been possible thanks to prof. Mário Salvatierra Júnior, to whom I want to express my gratitude for helping me review the correctness proofs of the different proposed algorithms, also thanks to Elena Yudovina that helped me to revise our main articles in English, correcting probably millions of errors in my drafts, as well as proposing new ideas in making the articles. I was really happy to have the opportunity of working with them for all these months, it was a real learning experience. I also want to thank prof. Eduardo Luzeiro Feitosa, thesis adviser, for his guidance and specially for his confidence in me.

A special thanks go to all the fellow students that shared their knowledge with me. Thanks to Marcos Salvatierra for being an excellent study partner, and to Marcela Pessoa for encouraging and stimulating to be a better person in these years.

Thank to the Almighty God, thank you for the guidance, strength, power of mind, defense, skills and for giving us a healthy life.

And last but not least, I wish to thank all my family, to which this work is dedicated. I am sure that my belated mother and father would be proud to know that we are very united despite the physical distance that separates us.

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Combinatorial optimization is an emerging field at the cutting edge of combinatorial and theoretical computer science that aims to use combinatorial techniques to solve discrete optimization problems. A discrete optimization problem seeks to determine the best possible solution from a finite set of possibilities.

From a computer science perspective, combinatorial optimization seeks to improve an algorithm by using mathematical methods either to reduce the size of the set of possible solutions or to make the search itself faster. From a combinatorial perspective, it interprets complicated questions in terms of a fixed set of objects about which much is already known: sets, graphs, polytopes, and strings. Combinatorial optimization refers primarily to the methods used to approach such problems and, mainly, does not provide guidelines on how to turn real-world problems into abstract mathematical questions, or vice versa.

A classic combinatorial optimization problem is the Closest String Problem (CSP) since in the literature there are much articles using different techniques to solve this problem, CSP belongs to a set of string comparison and selection problems. It sometimes called the Center String Problem, CSP has many practical application such as computational biology, coding theory, and web searching; the aim of a CSP instance is to find the geometric center of the given set of strings.

In the CSP, the objective is to find a string t that minimizes the number of differences among elements in a given set $\mathcal{S}$ of strings. The distance is defined as the Hamming distance between t and each $s^i \in \mathcal{S}$. The Hamming distance between two strings a and b of equal length is calculated by simply counting the character

positions in which a and b differ. For instance, if s =CCACT and t =TACCA, then $d_H(s,t) = 4$. Let us formally state the CSP:

Closest String Problem (CSP)

Input: strings $\mathcal{S} = \{s^1, \ldots, s^k\}$ of length m.

Output: a closest string t such that $\max_{\forall s \in \mathcal{S}} d_H(t,s) \leq d_{opt}$.

We are interested in optimal solutions for the general case of the CSP where k the number of strings is a fixed parameter. With this objective, we explored the literature concern to the CSP, there are several approximation algorithms, exact methods, and heuristics derived from the literature review. These gotten methods from the literature have a theoretical interest, however, new practical methods, to solve CSP in polynomial time using parameterized complexity, are still needed.

1.1 Motivation

The Human Genome Project made available a draft map of the Deoxyribonucleic acid (DNA) sequence in human beings. The completion of the enormous effort in sequencing the human DNA sequence developed several tools that were used for sequencing the DNA of other species.

Several byproducts of the Human Genome Project are now commonly used in real life, for example, DNA testing to identify criminals and paternity. But the benefits do not stop here. The DNA of many plants has been sequenced and with those, best pesticides could be discovered, which resulted in better productivity for farmers. With all these advances, new research areas have been created and/or old ones have re-appeared.

One area that has re-appeared is Bioinformatics, also called computational biology, that has almost the same age as computer science. Among many definitions for Bioinformatics, we particularly like the following two. Bioinformatics derives knowledge from computer analysis of biological data and Bioinformatics or computational biology is the use of techniques from applied mathematics, informatics, statistics, and computer science to solve biological problems [Shaik et al., 2019].

String comparison problems using Hamming distance have important applications in diverse areas of computational biology. One area where this is useful, for

example, is in drug target design. The objective can be to find a target genetic sequence that kills, for instance, all pathologic bacteria but does not affect human beings [Lanctot et al., 2003].

Another application in the same sense is multiple alignments. Multiple sequence alignments can be helpful in many circumstances like detecting historical and familial relations between sequences of proteins or amino acids and determining certain structures or locations on sequences [McClure et al., 1994].

Its application can also be found in the analysis of large conserved regions in the protein of diverse species in molecular evolution analysis using phylogenetic methods, and the construction of phylogenetic trees [Lance and Williams, 1967]. Figure 1.1 shows 8 different protein sequences from diverse species and its consensus string. Consensus methods attempt to find the optimal multiple sequence alignment given multiple different alignments of the same set of sequences.

Furthermore, in coding theory [Roman, 1992], the CSP is a key problem because in a lot of situations the objective is to find the sequences of characters which are closest to some given set of strings. This process aims to find the best way of encoding a set of messages. Figure 1.2 [Cryptography] presents an application in cryptography for data encryption.

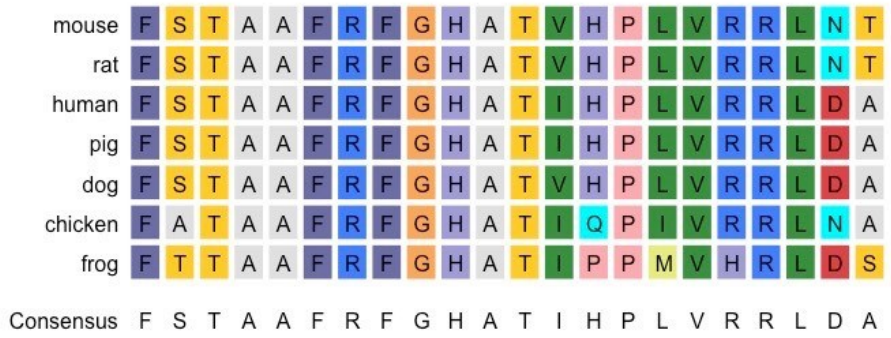
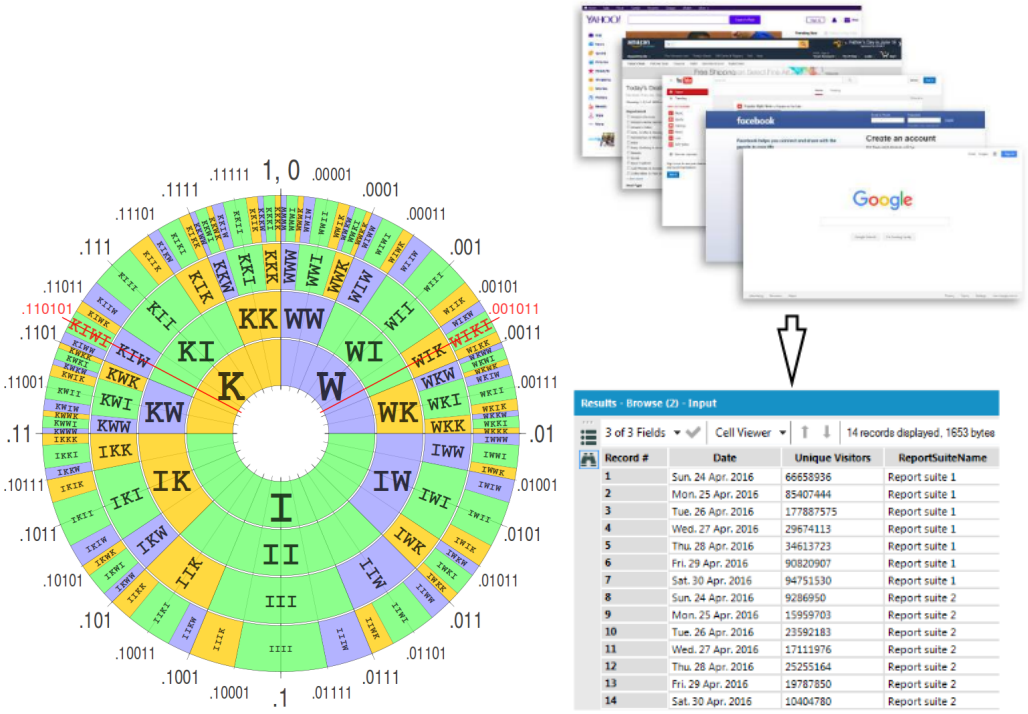


Figure 1.1: Illustration of application in the construction of phylogenetic trees in Multiple Alignment of Protein Sequences taken from various species. Source [Cornell, 2016].

Finally, in web searching one of the challenges is multiple occurrences of the same data, whether in exact duplicates or with minor changes [Amir et al., 2016]. Figure 1.2 [Web searching] shows an application in the analysis alongside in history of the same data accessed by each user using different search engines.



(a) Cryptography **(b)** Web searching

Figure 1.2: [Cryptography] Illustration of application of an arithmetic coding visualized as a circle, the values in red encoding "WIKI" and "KIWI". [Web searching] Illustration of application for web searching: a user chooses different search engines to search the target data, the dates accessed alongside is registered in a log. Source [Roman, 1992] and the authors.

1.2 Hypothesis

To the best of the author knowledge, there is one fixed-parameter algorithm for CSP parameterized by the number k of given strings proposed by [Gramm et al., 2003]. In particular, they showed an Integer Program has a number of variables dependent only on k, and thus by Lenstra's theorem [Lenstra, 1983] they obtain an algorithm with $O(k!^{4.5k!}m)$ integer operations on integer size $O(k!^{2k!}m)$. In their paper, they admit that because of the huge constants, the algorithm is not feasible for $k > 4$.

In [Bulteau et al., 2014] a challenge was launched, which consists in creating a fixed-parameter tractable algorithm for CSP parameterized by k (number of strings), thus, avoiding the use of integer linear programming, since that several attempts have been made to overcome this challenge.

In this context, this doctoral thesis is based on the following hypothesis:

Desing and implement a direct combinatorial fixed-parameter algorithm for Closest String parameterized by the number k of input strings (thus avoiding integer linear programming).

1.3 Objectives

The main objective of this work is to elaborate combinatorial algorithms for the Closest String Problem, treating different parameters with emphasis on the number of strings and the size of the alphabet due to practical interest in Bioinformatics and Cryptography. Thus, the general objective is broken down into the following specific objectives:

1. Analyze and design a fixed-parameter algorithm for the CSP treating any parameter;
2. Elaborate combinatorial algorithms for polynomial cases, involving a small number of strings;
3. Design and implement a heuristic algorithm for the general case;
4. Develop a combinatorial approach for the CSP parameterized by the number k (number of input strings).

1.4 Thesis contribution

This thesis aims to propose a fixed-parameter algorithm parameterized by k (number of input strings). To solve this challenge, we have progressed through three different stages. A first strategy was to study CSP for a small k (number of input strings), that is, 3-strings, since for $k = 2$ we have a trivial solution for CSP. In the second strategy, we add the complexity to $k = 4$ with a binary alphabet. Finally, for the third strategy, we extended our technique to the general case so we proposed a greedy algorithm that gives a good approximation for an optimal solution, consequently, we were able to establish an exact method, also for the general case. As a result, overall the objectives were achieved. The following will explain each solution strategy in more detail.

The first proposed strategy is a linear-time algorithm for the closest string problem with up to three strings. Called Minimization First Algorithm (MFA), it solves the CSP for up to three strings. The key idea is to identify column-position types in the input-instance, allowing to decompose it into five different tuples, corresponding to the position of each string in the set of strings, and to determine all unfixed column positions by simple evaluation through the different cases. A formal proof of the correctness and the computational complexity of the proposed algorithm are given. The proposed algorithm is compared with an integer programming formulation for 3-CSP. Furthermore, computational experiments in comparison tables will show the effectiveness of the proposed algorithm.

In the literature there is an algorithm to solve the CSP for $k = 3$ with an arbitrary alphabet proposed by [Gramm et al., 2001] whose implementation uses the concept of tuples and has the space complexity usage of $O(km \log m)$, meanwhile, the MFA has memory usage $O(k \times m)$. Both algorithms always find optimal solution values with execution time $O(m)$. For a more detailed description of the algorithms, see Sections 4.1 and 5.2. The original MFA idea was published in the 1st Computing Theory Meeting, a satellite event of the 2016 Brazilian Society of Computing Congress [Latorre and de Freitas, 2016].

The second strategy is a linear time algorithm for the Closest String Problem with four binary strings. We propose an efficient algorithm for solving the CSP for four strings with a binary alphabet called Minimization Second Algorithm (MSA). The key idea is to apply normalization for the CSP instances, allowing to decompose the problem in eight different cases corresponding to the position of each string in the set of strings and to determine all unfixed column positions by simple evaluation through the different cases. MSA is compared with an integer programming formulation and Boucher's method for 4-CSP with a binary alphabet. Furthermore, computational experiments in comparison tables will show the effectiveness of the proposed algorithm. For a more detailed explanation of the approaches, see Sections 4.2 and 5.3. The original MSA idea was published in CLAIO 2018 [Latorre and de Freitas, 2018].

The third strategy is a recursive exact algorithm for the Closest String Problem. We propose a recursive exact method for the CSP (general case) called

CSP-R. The CSP-R algorithm is a recursive exact method over k (number of input strings) to solve the CSP. Let $\mathcal{S} = \{s^1, \ldots, s^k\}$ be the input instance, it solves recursively sub instances with $\mathcal{S} \setminus \{s^i\}$ $(i = 1, \ldots, k)$ that is, $k - 1$ strings taken from the input instance. Let x^i be the optimal solution for $\mathcal{S} \setminus \{s^i\}$, based on these partial solutions, it builds a partially filled candidate. Finally, it fixes the unfixed column positions by a greedy heuristic algorithm (GREEDY). Furthermore, computational experiments in comparison tables will show the effectiveness of the proposed algorithm.

Sections 4.3 and 5.4 give a more detailed account of the proposed algorithm. The results of this algorithm appeared in JCMCC - 2019 "Journal of Combinatorial Mathematics and Combinatorial Computing" (accepted paper on July 1st, 2019) [Latorre and Salvatierra, 2019].

1.5 Thesis structure

Concerning organization, this thesis proposal is divided into chapters. Chapter 2 presents formally string selection problems that are related to the CSP, and it poses basic definitions. Chapter 3 shows a benchmarking among methods and algorithms for the CSP. Chapter 4 shows two linear-time algorithms for the Closest String Problem, one with up to three strings, and others with four binary strings, also a recursive approach for the CSP with k strings. Chapter 5 demonstrates the computational experiments in comparison tables for the proposed algorithms. Finally, the conclusion chapter poses all main findings, they are listed and summarized.

Chapter 2

THEORETICAL FOUNDATIONS

Comparison problems are among the most important faced by researchers in the area of computational biology. When working with the genome, or other types of amino-acid sequences, one of the main issues is how to correctly determine the similarities and differences occurring in two given sequences. These are in essence combinatorial problems and can be solved, as we shall discuss in this chapter, using techniques developed by the operations research community. This chapter defines string selection problems associated with the CSP and explains some notation that will be useful to describe problems on strings.

2.1 String selection problems

This section defines formally five string selection problems linked to the CSP, whose decision versions belong to NP-complete, according to [Lanctot et al., 2003]. Each problem is explained by an instance and an optimal solution. As can be defined later in the next chapter, the study of solving the CSP instances will be useful to solve instances from those problems. The formal definition of those problems are the following:

Farthest String Problem (FSP)

Let a finite set $\mathcal{S} = \{s^1, s^2, \cdots, s^k\}$ with k strings, each one of length m, under an alphabet Γ, the objective is to find a string x of length m under Γ, that maximizes d such that for each string $s^i \in \mathcal{S}$, we have $d_H(x, s^i) \geq d$.

As an example let the instance of FSP be the following: consider a set of strings

$\mathcal{S}$ = {AAACA,GTCTA,AATGC,CTTAC}. An optimal solution is given by the string x =TCGAG with $d = 4$.

Closest Substring Problem (CSubSP)

Given a finite set $\mathcal{S} = \{s^1, s^2, \cdots, s^k\}$ with k strings of length at least m under an alphabet Γ, the objective is to find a string x of length m under Γ, that minimizes d such that for each string s^i in $\mathcal{S}$, the relation $d_H(x, y) \leq d$ is truth for some substring y, of length m, from s^i.

As an example, consider $\mathcal{S}$ = {AAT,CCAA,CCTA,TCA}. In this case, an optimal solution is ACA with $d = 2$.

Farthest Substring Problem (FSubSP)

Given a finite set $\mathcal{S} = \{s^1, s^2, \ldots, s^k\}$ with k strings of length at least m under an alphabet Γ, the objective is to find a string x of length m under Γ, that maximizes d such that for each string s^i in $\mathcal{S}$ and for all substring y, of length m from s^i, we have $d_H(x, y) \geq d$.

For example consider $\mathcal{S}$ = {AAT,CCAA,CCTA,TCA}. In this case, an optimal solution is ACA with $d = 1$.

Close to Most String Problem (CMSP)

Given a finite set $\mathcal{S} = \{s^1, s^2, \ldots, s^k\}$ with k strings of length m under an alphabet Γ and a threshold $l > 0$, the objective is to find a string x of length m under Γ, maximizing the number of strings $s^i \in \mathcal{S}$ such that $d_H(x, s^i) \leq l$.

For example consider $\mathcal{S}$ = {AATCC,CCAAT,CCTAC,TCACC}. If $l = 3$, then an optimal solution is CCTCT with four strings satisfying $d_H(x, s^i) \leq 3$. If $l = 2$, then an optimal solution is ACAAC and three strings satisfying $d_H(x, s^i) \leq 2$.

Distinguishing String Selection Problem (DSSP)

Given two finite sets of strings $\mathcal{S}_c$ and $\mathcal{S}_f$, all the strings of length at least m, under an alphabet Γ, and two positive integer numbers l_c and l_f, the objective is to find a string x of length m under Γ such that for each string $s_c \in \mathcal{S}_c$, there is some substring y_c, of length m, from s_c that satisfied $d_H(x, y_c) \leq l_c$, and for all substring y_f, of length m, from $s_f \in \mathcal{S}_f$ we have $d_H(x, y_f) \geq l_f$.

For example, consider $\mathcal{S}_c$ = {AATCC,CCAAT,CCTAC,TCACC} and a set $\mathcal{S}_f$ = {AATAA,CCACT,GGTAC,TCAAC}. If $l_c = 3$ and $l_f = 2$, then ACACC is an

optimal solution.

2.1.1 Complexity hierarchy

Notice that the four first problems described above are optimization problems, meanwhile the last problem is a decision problem. As mentioned later at the beginning of this chapter, these five problems have an interesting relationship with the CSP. Figures 2.1 and 2.2 show the relationship between those problems, in terms of reduction and computational complexity.

Figure 2.1 presents a set of NP-hard problems that are closely related, this hierarchy of complexities were proposed in [Lanctot et al., 2003]. Figure 2.2 presents some reductions from classic NP-hard problems to string selection problems [Boucher et al., 2012]. Furthermore, each of them has equivalent complexity. As a consequence an optimal solution for one of them will be used to solve the others, hence finding a method to solve one of them can be used to solve the other related problems.

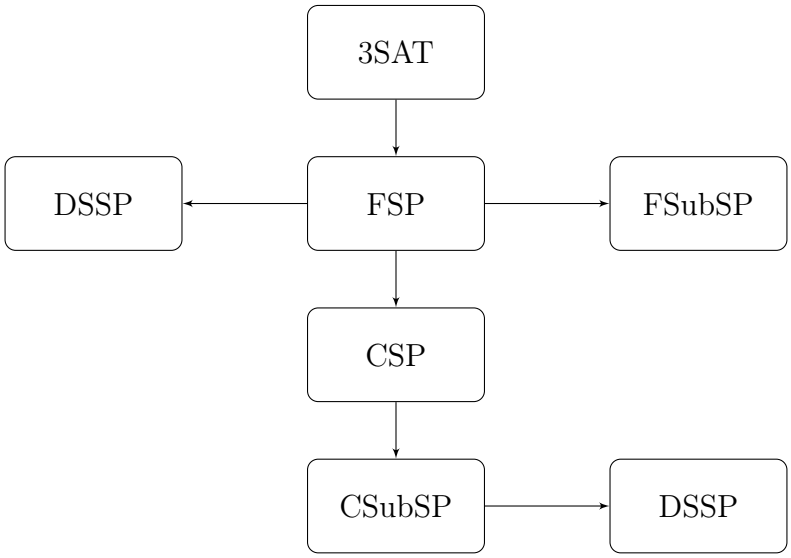


Figure 2.1: Hierarchy of complexities in the consensus sequence problem: the relationships between problems, in terms of reductions, and consequent computational complexities. By 3SAT, we mean 3-Satisfaction. Source: the authors.

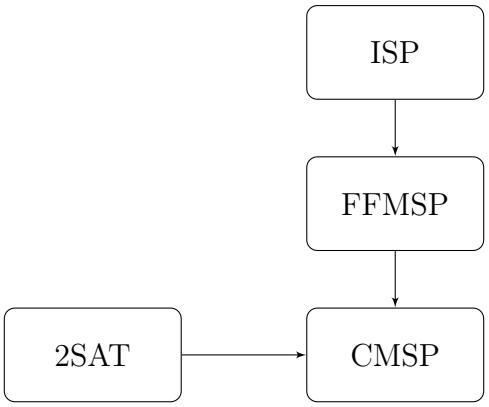


Figure 2.2: A set of NP-complete problems. By 2SAT, and ISP we mean 2-Satisfaction, and Independent Set Problem, respectively. Remember that there is a well known deterministic algorithm of polynomial time for the 2SAT, but the 3SAT and ISP belong to NP-complete. Source: the authors.

2.1.2 Hamming distance

According to [Hill, 1986], a metric on a set X is a function, called distance function or distance is given by:

$$d : X \times X \to \Re,$$

where $\Re$ is the set of real numbers. For all $x, y, z \in X$, this function needs to satisfy the following conditions:

1. $d(x, y) \geq 0$ (non-negativity)
2. $d(x, y) = 0$ if and only if $x = y$ (identity)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

Let $(F_q)^n$ the set of all n-tuples ordered $a = a_1a_2...a_n$ where every $a_i \in F_q$. F_q is an alphabet q-ith element is obtained from the set of sequences of symbols where each symbol is chosen from the set $F_q = \{\lambda_1, \lambda_2, ..., \lambda_q\}$ q different elements. The Hamming distance between two vectors x and y of $(F_q)^n$ is the number of positions in which differ. It is denoted by $d(x, y)$.

For example,

a. In the $(F_2)^5$ we have $d(00111, 11001) = 4$,

b. While in $(F_4)^5$ we get $d(AACGG, AGACG) = 3$.

2.1.3 Normalized process

In general, an instance can be reduced to its isomorphic instance [Gramm et al., 2001] by a bijective function. More formally we have the following definition:

Definition 1 (Normalized instance)**.** *Let $\mathcal{S}$ be an instance, that is, $\mathcal{S} = \{s^1, \ldots, s^k\}$, where $|s^i| = m$, with $1 \leq i \leq k$ and $1 \leq j \leq m$. Let $M_{k\times m}$ be a matrix of characters from $\mathcal{S}$ each column is a position of the strings in $\mathcal{S}$, so we have $M[c_1, c_2, \ldots, c_m]$. Let Γ_j be the alphabet of c_j we ordered the symbols in Γ_j according to their frequency in c_j in non-decreasing order, we get, $f_{\sigma_1} \geq f_{\sigma_2} \cdots \geq f_{\sigma_q}$. Let Γ'_j be an alphabet such that $|\Gamma_j| = |\Gamma'_j|$, let $\phi : \Gamma \to \Gamma'$ be a bijective function such that,*

$$\phi(c_j)_{j=1,\ldots,m} = \begin{cases} \lambda_1 & \text{if} \quad c_j^i = \sigma_1 \quad i = 1, \ldots, k \\ \lambda_2 & \text{if} \quad c_j^i = \sigma_2 \\ \cdots & \\ \lambda_q & \text{if} \quad c_j^i = \sigma_q \quad \sigma_q \in c_j; \quad q = 1, \ldots, |\Gamma_j| \end{cases} \tag{2.1}$$

Let $M'[c'_1, \ldots, c'_m]$ be a matrix of characters composed of $c'j$, and $\mathcal{S}'$ be an instance based on characters from M', as a result $\mathcal{S}'$ is called normalized instance from $\mathcal{S}$.

2.2 Combinatorial optimization

Robert E. Bixby, in [Bixby, 1987], defines a combinatorial optimization problem as: Let E be a finite set, $\mathbb{S}$ a family of subsets of E and $w \in \mathbb{R}^{|E|}$ a weight function of real values defined under the elements of E. The combinatorial optimization problem associated is to find a set $S^* \in \mathbb{S}$ such that

$$w(S^*) = \max_{S\in\mathbb{S}} w(S)$$

where $w(S) = \sum_{e\in S} w(e)$.

2.2.1 Approximation algorithms

The available techniques for solving optimization problems can be roughly classified into three main categories: exact, approximation, and heuristic methods. While

exact methods allow finding solutions with theoretical guarantees, their run-time increases dramatically with the instance size. Heuristic approaches, on the other side, sacrifice optimality guarantees to find solutions more efficiently, by spending only a 'reasonable' amount of time to compute them. Unlike heuristics, where there is no knowledge of the quality of the solution returned, approximation methods provide solutions within a small constant factor of the optimal solution. The combination of these approaches defines the category of hybrid methods.

Definition 2 (Approximation algorithm)**.** *According to [Cormen et al., 2009], let $\mathcal{C}_{opt}$ be the cost of the optimal algorithm for a problem of size n. An approximation algorithm for this problem has an approximation ratio $\varrho(n)$ if, for any input, the algorithm produces a solution of cost $\mathcal{C}$ such that:*

$$max(\frac{\mathcal{C}}{\mathcal{C}_{opt}}, \frac{\mathcal{C}_{opt}}{\mathcal{C}}) \leq \varrho(n)$$

such an algorithm is called a $\varrho(n)-$approximation algorithm.

An approximation scheme that takes as input $\epsilon > 0$ and produces a solution such that $\mathcal{C} = (1+\epsilon)\mathcal{C}_{opt}$ for any fixed ϵ, is a $(1+\epsilon)-$approximation algorithm.

A polynomial time approximation scheme (PTAS) is an approximation algorithm that runs in time polynomial in the size of the input, n. A fully polynomial time approximation scheme (FPTAS) is an approximation algorithm that runs in time polynomial in both n and ϵ. For example, a $\mathcal{O}(n^{\frac{2}{\epsilon}})$ approximation algorithm is a PTAS but not a FPTAS. A $\mathcal{O}(\frac{n}{\epsilon^2})$ approximation algorithm is a FPTAS.

2.2.2 Integer programming

Wolsey in [Wolsey, 1998] defined an integer programming as follows, suppose that we have a linear programming

$$max\{cx : Ax \leq b, x \geq 0\}$$

where A is an m by n matrix, c an n-dimensional row vector, b an m-dimensional column vector, and x an n-dimensional column vector of variables or unknowns. Now we add in the restriction that certain variables must take integer values. If some but not all variables are integer, we have a (Linear) Mixed Integer Program

(MIP), written as

$$\begin{aligned} \max \quad & cx + hy \\ \text{s.a.:} \quad & Ax + Gy < b \\ & x \geq 0, y \geq 0 \textit{ and integer} \end{aligned}$$

Where A is again m by n, G is m by p, h is a p row-vector, and y is a p column-vector of integer variables. If all variables are integer, we have a (Linear) Integer Program (IP), written as

$$\begin{aligned} \max \quad & cx \\ \text{s.a.:} \quad & Ax \leq b \\ & x \geq 0 \textit{ and integer} \end{aligned}$$

2.3 Parameterized complexity

Parameterized complexity is a branch of computational complexity theory that focuses on classifying computational problems according to their inherent difficulty to multiple parameters of the input or output. The complexity of a problem is then measured as a function in those parameters. This allows the classification of NP-hard problems on a finer scale than in the classical setting, where the complexity of a problem is only measured by the number of bits in the input. The first systematic work on parameterized complexity was done by [Downey and Fellows, 1999].

2.3.1 Fixed parameter algorithm

According to [Downey and Fellows, 1999] a Fixed Parameter Algorithm is an alternative way to deal with NP-hard problems instead of approximation algorithms. There are three general desired features of an algorithm:

1. Solve NP-hard problems;
2. Run in polynomial time (fast);
3. Get exact solutions.

The idea is to aim for an exact algorithm but isolate exponential terms to a specific parameter. When the value of this parameter is small, the algorithm gets fast instances. Hopefully, this parameter will be small in practice.

A parameter is a nonnegative integer $k(x)$ where x is the problem input. Typically, the parameter is a natural property of the problem (some k in input). It may not necessarily be efficiently computable (e.g., OPT).

A parameterized problem is simply the problem plus the parameter or the problem as seen with respect to the parameter. There are potentially many interesting parameterizations for any given problem.

The goal of fixed-parameter algorithms is to have an algorithm that is polynomial in the problem size n but possibly exponential in the parameter k and still get an exact solution. Fixed Parameter Tractability and kernelization algorithm are defined by [Downey and Fellows, 1999] as follows.

Definition 3 (Fixed Parameter Tractability)**.** *A parameterized problem is fixed-parameter tractable (FPT) if there is an algorithm with running time* $\leq f(k)n^{\mathcal{O}(1)}$*, such that* $f : \mathbb{N} \to \mathbb{N}$ *(non negative) and* k *is the parameter, and the* $\mathcal{O}(1)$ *degree of the polynomial is independent of* k *and* n*.*

Definition 4 (kernelization algorithm)**.** *A decidable problem is FPT if and only if it is kernelizable: a kernelization algorithm for a problem* $\mathcal{Q}$ *takes an instance* (x, k) *and in polynomial in* $|x| + k$ *produces an equivalent instance* (x', k') *(i.e.,* $(x, k) \in \mathcal{Q}$ *iff* $(x', k') \in \mathcal{Q}$*) such that* $|x'| + k' \leq g(k)$ *for some computable function* g*. The function* g *is the size of the kernel, and if it is polynomial, we say that* $\mathcal{Q}$ *admits a polynomial kernel.*

2.4 Remarks

This chapter has provided the theoretical foundation, it is necessary for solving Combinatorial Optimization Problems. The study of these concepts was extremely important for the development of this work. Mainly the fixed parameter algorithm, we use it to analyze the principal characterization of the CSP parameterized by k (number of the given strings). The next chapter presents the main related works applied to CSP.

Chapter 3

CLOSEST STRING PROBLEM

The Closest String Problem has been extensively studied since its introduction by [Frances and Litman, 1997]. The problem was studied both in computational biology and theoretical computer science. In this way, it made possible a series of articles applying several existing techniques, they can be divided into four categories such as formulations in integer programming, approximation algorithms, fixed parameter tractability (exact solutions), meta-heuristics, and matheuristics for the CSP.

3.1 Formal definition

To formally state the problem, the following notation is needed: An alphabet $\Gamma = \{\sigma_1, \sigma_2, \ldots, \sigma_q\}$ is a finite set of elements, called characters, $s^i = \{s^i_1, s^i_2, \ldots, s^i_m\}$ is a sequence of length m ($|s^i| = m$) on $\Gamma(s^i_j \in \Gamma, i = 1, 2, \ldots, k)$. Given two sequences s^i and s^j on Γ such that $|s^i| = |s^j|$, $d_H(s^i, s^j)$ denotes their Hamming distance and is given by $d_H(s^i, s^j) = \Sigma_{l=1}^{|s^i|}\Delta(s^i_l, s^j_l)$ where s^i_l and s^j_l are the characters in position l in s^i and s^j, respectively and $\Delta : \Gamma \times \Gamma \to \{0, 1\}$ is the predicate function such that

$$\Delta(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases}$$

So the CSP can be formulated as a follows.

$$d = min\{s : max_{i=1}^{k}\Sigma_{l=1}^{|s^i|}\Delta(s^i_l, s_l) : s_l \in \Gamma\}$$

Furthermore, there are other names that point out the same problem, we show a benchmarking among articles that tackle the CSP in a different point of view receiving a different name instead of CSP, each of them highlights main characteristics

of the problem, through the time-line these names receive especially interest with reference to technique used to solve the CSP. In Table 3.1, the column (Technique) aims the approach using in the article, the Column (Name) mentions the particular name referring to CSP, and finally, the Column (Reference) presents the pointing out to that article.

Table 3.1: A list of articles that concerns other names referring to CSP.

Technique	Name	Reference
Metrical characterization (2-vectors set)	Minimum Radius Problem	[Frances and Litman, 1997]
Efficient approximation algorithms	Hamming Center Problem	[Gasieniec et al., 1999]
Exact solutions for Closest String	Center String Problem	[Gramm et al., 2001]
Polynomial-time approximation scheme	Closest String Selection Problem	[Deng et al., 2002]
Fixed-parameter algorithms	Consensus String Problem	[Gramm et al., 2003]
Polynomial-time approximation scheme	Hitting String Problem	[Lanctot et al., 2003]
Sample-driven and pattern-driven	Motif Finding Problem	[Sze et al., 2004]
Small Motifs Structure	Consensus Sequence Problem	[Boucher et al., 2009]
Hardness of Approximation	Consensus Patterns	[Boucher et al., 2015]
Configurations and minority	String Consensus Problem	[Amir et al., 2016]

3.2 Formulations in integer programming

An instance of CSP consists of $\mathcal{S} = \{s^1, \ldots, s^k\}$ such that $s^i = s^i_1 s^i_2 \ldots s^i_m \in \Gamma^m$, for $i = 1, \ldots, k$. The objective of the CSP is to find a string $x \in \Gamma^m$ such that $\max_{i=1,\ldots,k} d_H(x, s^i)$ is minimized.

Here we introduce the third Integer Programming (IP) formulation in [Meneses et al., 2004] which is thought as the strongest; it always provides optimal solution for the general case of CSP, and it converges more quickly for small instances.

Define a 0-1 variable $x_{\sigma,j}$ as $x_{\sigma,j} = 1$ if the character σ of U_j is used in the j-th position of a solution and 0 otherwise.

$$x_{\sigma,j} = \begin{cases} 1 & \text{if } \sigma \in U_j \text{ is used in the position } j \text{ in a solution} \\ 0 & \text{otherwise.} \end{cases}$$

The CSP can be reduced to the following IP problem:

$$\min \quad d \tag{3.1}$$

$$\text{s.t.:} \quad \sum_{\sigma \in U_j} x_{\sigma,j} = 1 \qquad j = 1, \ldots, m \tag{3.2}$$

$$d \geq m - \sum_{j=1}^{m} x_{s^i_j, j} \qquad i = 1, \ldots, k \tag{3.3}$$

$$x_{\sigma,j} \in \{0, 1\} \qquad \sigma \in U_j; j = 1, \ldots, m \tag{3.4}$$

$$d \in \mathbb{Z}_+. \tag{3.5}$$

The equations in (3.2) guarantee that only one character in U_j is used in each position of a solution. The inequalities in (3.3) say that if a character in a string s^i is not in the solution x^*, then that character will contribute to increasing the Hamming distance for x^* to s^i. Constraints (3.4) are binary inequalities, and (3.5) forces d to assume non-negative integer value. In truth we are interested in the minimized value.

Also, in [Gramm et al., 2003] was suggested an IP-formulation based on column-position types, this yields fixed-parameter algorithm (FPT) for CSP concerning parameter k (number of input strings):

$$\min \quad d \tag{3.6}$$

$$\text{s.t.:} \quad \sum_{t \in T(S)} \sum_{\sigma \in \Gamma \setminus \{\sigma_{t,i}\}} x_{t,\sigma} \leq d \qquad \forall i \in \{1, \ldots, k\} \tag{3.7}$$

$$\sum_{\sigma \in \Gamma} x_{t,\sigma} = \#t \qquad \forall t \in T(S) \tag{3.8}$$

$$x_{t,\sigma} \in \{0, 1, \ldots, \#t\} \qquad \forall \sigma \in \Gamma, \forall t \in T(S) \tag{3.9}$$

In this formulation, for every $t \in T(S)$ and $\sigma \in \Gamma$, the variable $x_{t,\sigma}$ is the number of occurrences that σ has in the closest string at locations that correspond to column type t. Let $s^i \in S$ and let $\sigma_{t,i}$ be the character of string s^i in column type t. The restrictions (3.7) calculate the Hamming distance of s^i from the center string, that is, for each column type t we sum the characters in the center string at locations that correspond to t that are different from the character s^i has in these locations. Constraints (3.8) impose the number of column types existed in each tuple, and (3.9) makes $x_{t,\sigma}$ to have integer values. Finally, the objective is minimized the value d.

Furthermore, in [Chimani et al., 2011] is posed an IP-formulation. Let σ be an arbitrary but fixed character, then $\Gamma' = \Gamma \setminus \{\sigma\}$.

$$\min \quad d \tag{3.10}$$

$$\text{s.t.:} \quad \sum_{i:s|_i \in \Gamma'} x_{i,s|_i} + \sum_{i:s|_i \notin \Gamma'} (1 - \sum_{\sigma' \in \Gamma'} x_{i,\sigma'}) + d \geq m \qquad \forall s \in \mathcal{S} \tag{3.11}$$

$$\sum_{\sigma' \in \Gamma'} x_{i,\sigma'} \leq 1 \qquad \forall i \in \{1, \ldots, m\} \tag{3.12}$$

$$x_{i,\sigma'} \geq 0 \qquad \forall i \in \{1, \ldots, m\}, \sigma' \in \Gamma' \tag{3.13}$$

Observe, this formulation has $|\Gamma|k$ constraints and $(|\Gamma| - 1)m + 1$ decision variables. The inequalities (3.11) guarantee that d is at least as large as the number of mismatches between t optimal solution and any $s \in \mathcal{S}$. The constraints (3.12) and (3.13) are a complete characterization of the feasible solution strings and are facets in the CSP polyhedron. In conclusion, the goal is minimized the value d.

Similarly, in [Arbib et al., 2017], is presented an integer-programming formulation for the binary case. Let I_0^i and I_1^i denote the set of indexes j such that $s_j^i = 0$ and $s_j^i = 1$, respectively, the Hamming distance is $2d$ and $k^i = \sum_{j=1}^{m} s_j^i$.

$$\min \quad d \tag{3.14}$$

$$\text{s.t.:} \quad 2d - \sum_{j \in I_0^i} x_j + \sum_{j \in I_1^i} x_j \geq k^i \qquad i = 1, \ldots, k \tag{3.15}$$

$$x_j \in \{0, 1\} \qquad j = 1, \ldots, m \tag{3.16}$$

$$d \geq 0 \tag{3.17}$$

Note that, this formulation has k restrictions and $1 + m$ decision variables. The inequalities (3.15) guarantee that the only one character in each x_j is selected, at the same time, calculate the Hamming distance between a vector solution x and the strings in $\mathcal{S}$. Constraints (3.16) are binary inequalities, and (3.17) imposes d to get non-negative value. The target is minimized the value d.

Table 3.2 shows a benchmarking among articles related to formulations in integer programming for CSP. The first formulation is for the general case, in the time-line has been developed to gain a good feasible solution for biggest instances, in the other hand for smallest instances that formulation is efficient in running time. The last two formulations are for binary cases, in [Gramm et al., 2003] presents a fixed parameter algorithm on the number of strings, and in [Arbib et al., 2017] aims that formulation focused only in the binary alphabet.

Table 3.2: A list of articles involving formulations in integer programming.

Year	Alphabet	Restrictions	Variables	References
1997	q	$m+k$	$1+\sum_{j=1}^{m}\|V_j\|$	[Ben-Dor et al., 1997]
				[Li et al., 2002]
				[Meneses et al., 2004]
2003	q	$(1+\|\Gamma\|)\#t$	$\|\Gamma\|\#t$	**[Gramm et al., 2003]**
2011	q	$\|\Gamma\|k$	$(\|\Gamma\|-1)m+1$	[Chimani et al., 2011]
2017	2	k	$1+m$	[Arbib et al., 2017]

Since our proposed methods are demarcated in the field of parameterized complexity, a fair way of comparison would be to use the fixed parameter algorithm proposed by [Gramm et al., 2003], that is, an FTP parameterized by k (number of strings). On the other hand for comparison purposes, this algorithm was only possible to test it for 3 and 4 strings, specifically for 5 strings we come across an exponential explosion of variables.

For that reason, Meneses' IP-formulation will be used [Meneses et al., 2004] as a baseline for comparison, because this algorithm is proven to be robust, it has fast convergence to find optimal solutions for larger instances, and it always finds an exact solution.

3.3 Approximation algorithms

The first approximation algorithm for CSP, with a performance guarantee of two, firstly was presented in the Symposium on Discrete Algorithms (SODA, 1999), then was published in [Lanctot et al., 2003]. This very simple algorithm can be described as in Algorithm 1. The technique employed is to take one of the strings in the input (without loss of generality we can take the first string) and return it as the solution.

This very simple algorithm can be shown to give a performance guarantee of two. To do this, let x denote an optimal solution, which has a minimum distance at most d from each solution in $\mathcal{S}$, i.e., $d_H(x, s^i) \leq d$, for $i \in \{1, \ldots, k\}$. If we take s^1 as the solution, as shown in Algorithm 1, then it is clear that $d_H(s^1, s^i) \leq d_H(s^1, x) + d_H(x, s^i) \leq 2d$, for $i \in \{2, \ldots, k\}$. Thus, s^1 is a solution with cost at most twice the optimum.

Algorithm 1: A 2-Approximation algorithm for the CSP.

1 Procedure Algorithm 1 $(k, m, \mathcal{S})$

2 k : number of strings

3 m : size of strings

Input : $\mathcal{S} = \{s^1, \ldots, s^k\}$

4 return s^1 as a solution

A better solution was introduced by [Lanctot et al., 2003], who presented an approximation algorithm with a performance guarantee of $\frac{4}{3}(1+\epsilon)$, for any small $\epsilon > 0$. The basic idea of the algorithm consists of formulating the problem as an integer programming problem, solving the linear programming (LP) relaxation, and using the results of the LP to find an approximate solution. To find the approximate solution from the LP solution, the technique of randomized rounding is employed.

Randomized rounding is a technique for getting an integer $0-1$ solution from a continuous solution for an LP problem. This method works by defining the value of the variable $x \in \{0, 1\}$ to be $x = 1$ with probability $\bar{x}$, where $\bar{x}$ is the value of the continuous variable corresponding to x in the relaxation of the original integer programming problem.

In [Lanctot et al., 2003], the solution found as described above is proved to be never more than $\frac{4}{3}$ of the optimum solution. This is shown by taking the expected value of the solution and using some probabilistic bounds over the result.

Another algorithm was requested by [Li et al., 1999], a polynomial time approximation scheme (PTAS) for the CSP. A PTAS is a special type of approximation algorithm that, for each $\epsilon > 1$, yields a guaranteed performance in polynomial time. Thus, this can be viewed as a way of getting solutions with guaranteed performance, for any desired threshold greater than one. The PTAS presented in [Li et al., 1999] is also based on randomized rounding, and their basic steps are presented in Algorithm 2. The scheme is similar to the algorithm in [Lanctot et al., 2003], but the idea of randomized rounding is refined to check the results for a large (but polynomial) number of subsets of indices. The analysis of the PTAS presented in this subsection is also similar to the one used in [Lanctot et al., 2003]. However, here there are a large number of iterations where the algorithm needs to solve a linear relaxation of an integer program. This makes the resulting algorithm impractical for any instance with large strings. It is interesting to note that a similar result appeared independently in the context of coding theory in [Gasieniec et al., 1999].

Algorithm 2: A PTAS for closest string problem.

```
Procedure Algorithm 2 (k, m, S)
k : number of strings
m : size of strings
for each r-element subset of the input strings do
    Find a subset P of positions in a string
    Solve an IP relaxation to find the optimum solution for the positions in P
    Calculate the cost of the resulting partial solution
for i ∈ {1, ..., k} do
    compute the cost of the i-th string as a solution to the problem
return the best of the solutions in the last two steps
```

In addition, [Andoni et al., 2006] showed a $(1+\epsilon)$-approximation algorithm with running time of $k^{\Omega(\epsilon^{-2+\gamma})}$ for any $\gamma > 0$, the technique applied is basically a review of the Algorithm 2 using the dimensionality reduction method developed by the authors. Also, in [Ma and Sun, 2008] was presented a $(1+\epsilon)-$approximation algorithm with a running time of $mk^{\mathcal{O}(\epsilon^{-2})}$. That last result was obtained using their new fixed-parameter algorithm. The main results of this category are summarized in Table 3.3.

Table 3.3: A list of articles that tackles approximate algorithms to CSP.

Year	Approx.	Running time	References
1999	2	$\mathcal{O}(1)$	[Lanctot et al., 2003]
1999	$\frac{4}{3}+\epsilon$	LP	[Lanctot et al., 2003]
1999	$\frac{4}{3}+\epsilon$	LP	[Gasieniec et al., 1999]
2002	$1+\epsilon+\frac{1}{2r-1}$	$(km)^r k^{\mathcal{O}(\log\lvert\Gamma\rvert\times\frac{r^2}{\epsilon^2})}$	[Li et al., 1999, Li et al., 2002]
2006	$1+\epsilon$	$mk^{\mathcal{O}(\epsilon^{-2}\log(\frac{1}{\epsilon}))}$	[Andoni et al., 2006]
2008	$1+\epsilon$	$mk^{\mathcal{O}(\epsilon^{-2})}$	[Ma and Sun, 2008]

3.4 Fixed parameter complexity

Another direction explored by researchers to solve the CSP is the use of parameterized complexity [Downey and Fellows, 1999]. The central idea of parameterized complexity is to study how difficult a problem remains after some of its parameters are fixed. Depending on the problem, different behaviours can arise. For example, the problem can become solvable in polynomial time. This is the case of the vertex cover problem, which can be

solved with practical algorithms, with time complexity in the order of $\mathcal{O}(1.3^c + cn)$, when the size c of the cover is fixed [Downey and Fellows, 1999, Fellows, 2002]. On the other hand, some problems remain NP-hard even for a fixed parameter. For example, finding a c-colouring of a graph is NP-complete even when the number of colors c considered is equal to three only.

3.4.1 Fixed parameter algorithms for the CSP

For the CSP, parameterized complexity results have shown [Gramm et al., 2001] that when the value of the minimum distance is fixed to d a polynomial algorithm can be used to give exact solutions. In this case, the Algorithm 3 was proved to run in time $\mathcal{O}(km + kd^{d+1})$. Although this result is not attractive for large values of d, sometimes in practice d is small (less than 20). This can be justified since the objective in genetic applications is to find strings which are quite similar to another, and therefore the resulting distance is small.

Most works deal with a decision version of the problem, rather than the optimization version defined above. In the decision version, in addition to the set of strings, a distance d is also part of the input. The objective is to decide whether there is a consensus whose Hamming distance to each of the input string does not exceed d. Stojanovic [Berman et al., 1997] proposed a linear-time algorithm for $d = 1$. Ma and Sun [Ma and Sun, 2008] presented another algorithm running in $\mathcal{O}(km + kd(16|\Gamma|)^d)$ time, where Γ denotes the alphabet.

Furthermore, there have been some efficient algorithms for a small constant k (number of input-strings). [Gramm et al., 2001] proposed a direct combinatorial algorithm for finding a consensus string t for three strings. [Sze et al., 2004] showed a condition for the existence of a string whose radius is less than or equal to d. [Boucher et al., 2009] proposed a linear algorithm for finding a string t such that $max_{1\leq i\leq 4} d(t, s^i) \leq d$ for four binary strings. [Amir et al., 2009] presented a linear algorithm finding a consensus string minimizing both distance sum and radius for 3 strings. Finally,[Amir et al., 2016] posed a quadratic time algorithm for 5 strings under binary alphabet. The main results in this category are presented in Table 3.4.

In [Bulteau et al., 2014] a challenge was launched, which consists in creating a fixed-parameter tractable algorithm for CSP parameterized by k (number of strings), thus, avoiding the use of integer linear programming, since that several attempts have been made to overcome this challenge. A first attempt was proposed by [Amir et al., 2016] with running time $O(k^2 m^k)$, it does not fit the definition of fixed-parameter tractability, because

k the fixed-parameter is not isolated, that is, $O(k^2m^k) \leq f(k)m^{O(1)}$, $f(k) = k^2$, $m^{O(1)} \approx m^k$, $k \notin O(1)$. A second attempt was presented by [Dalpasso and Lancia, 2018] with time complexity $O(|\Gamma|m^{k+1})$, also this approach does not fit the previous definition, that is, $O(|\Gamma|m^{k+1}) \leq f(k)|\Gamma|m^{O(1)}$, $|\Gamma|m^{k+1} \leq |\Gamma|m^{O(1)}$, $k+1 \notin O(1)$. As result, we can conclude that until the present moment such a challenge is still open.

Algorithm 3: A FPT algorithm for CSP parameterized by d.

```
1  Recursive procedure CSd (s, Δd)
2  Global variables: Set of strings S = {s^1, s^2, ..., s^k}, integer d
   Input  : Candidate string s and integer Δd.
   Output: A string ŝ with max_{i=1,...,k} d_H(ŝ, s^i) ≤ d and d_H(ŝ, s^i) ≤ Δd, if it exists,
           and "not found" otherwise.
3  if Δd < 0 then return "not found";
4  if d_H(s, s^i) > d + Δd for some i ∈ {1, ..., k} then return "not found";
5  if d_H(s, s^i) ≤ d for all i = 1, ..., k then return s;
6  foreach i ∈ {1, ..., k} such that d_H(s, s^i) > d do
7      P := {p | s[p] ≠ s^i[p]}
8      Choose any P' ⊆ P with |P'| = d + 1
9      for all p ∈ P' do
10         s' := s
11         s'[p] := s^i[p]
12         s_ret := CSd(s', Δd − 1)
13         If s_ret ≠ "not found" then return s_ret
14 Return "not found"
```

To know the characteristics of CSP with k as a fixed-parameter, we made an exploratory analysis for special cases, a first case was for instances with $k = 3$ strings and an arbitrary alphabet, for that case, we implemented a linear-time algorithm called Minimization First Algorithm (MFA). In the literature, there can be found an exact method that solves for the same case developed by [Gramm et al., 2003], it will be a baseline approach for comparison, both methods are efficient and their running time is $O(m)$.

Another special case explored was for instances with $k = 4$ and a binary alphabet, for that case we designed and implemented a linear-time algorithm called Minimization Second Algorithm (MSA). For the same case, [Boucher et al., 2009] presented an efficient method, it will be a baseline method for comparison. Both methods have running time $O(m)$.

Finally, [Amir et al., 2016] and [Dalpasso and Lancia, 2018] proposed exact methods for the general case, both methods set the parameter k (number of strings), for that reason, they will be baseline methods for comparison. In those articles only the theoretical part were presented, such as proof of correctness, and a very high-level script of their methods, in this way, above the similarities and differences with the recursive exact method were reported.

Table 3.4: A list of articles that uses exact methods to solve the CSP.

Year	#Strings	Alphabet	Running time	References
2003	k	q	$\mathcal{O}(km + kd^{d+1})$	[Gramm et al., 2003]
2008	k	q	$\mathcal{O}(km + kd(16\|\Gamma\|)^d)$	[Ma and Sun, 2008]
2016	k	q	$\mathcal{O}(k^2m^k)$	**[Amir et al., 2016]**
2018	k	q	$\mathcal{O}(\|\Gamma\|m^{k+1})$	**[Dalpasso and Lancia, 2018]**
2019	k	q	$\mathcal{O}(k^{k+1}m^2)$	**[Latorre and Salvatierra, 2019]**
2001	3	q	$\mathcal{O}(m)$	**[Gramm et al., 2001]**
2016	3	q	$\mathcal{O}(m)$	**[Latorre and de Freitas, 2016]**
2011	3	2	$\mathcal{O}(m)$	[Liu et al., 2011]
2009	4	2	$\mathcal{O}(m)$	**[Boucher et al., 2009]**
2018	4	2	$\mathcal{O}(m)$	**[Latorre and de Freitas, 2018]**
2016	5	2	$\mathcal{O}(m^2)$	[Amir et al., 2016]

3.4.2 Kernelization algorithms of the CSP

A fundamental and very powerful technique in designing FPT algorithms is kernelization. In a nutshell, a kernelization algorithm for a parameterized problem is a polynomial-time transformation that transforms any given instance to an equivalent instance of the same problem, with size and parameter bounded by a function of the parameter in the input. Typically this is done using so-called reduction rules, which allow the safe reduction of the instance to an equivalent 'smaller' instance.

In [Gramm et al., 2001] the authors showed that CSP is FPT parameterized by k (number of strings), they also proved a polynomial time reduction $\mathcal{O}(k^2 d \log k)$. Also, in [Basavaraju et al., 2014] the authors presented that the CSP is not likely to have polynomial kernels when parameterized by d (Hamming distance), they arrived at the results by showing a polynomial parameter transformation from CNF-SAT parameterized by the

number of variables.

In Table 3.5, it can be observed that when the CSP is parameterized by k (number of strings) is still open, as a consequence, it was not possible to find a reduction in polynomial time. In this sense, kernelization can be viewed as polynomial time preprocessing which has universal applicability, not only in the design of efficient FPT algorithms but also in the design of approximation and heuristic algorithms. Our research also fits this challenge.

Table 3.5: A polynomial kernelization status for the CSP.

Parameters	Kernel	References
d	None	
k	**Open**	**[Latorre and Salvatierra, 2019]**
(d, k)	$\mathcal{O}(k^2 d \log k)$	[Gramm et al., 2001]
(d, m)	$\nexists$	[Basavaraju et al., 2014]
m	-	
(k, m, d)	-	

3.5 Metaheuristics

Normally, metaheuristics are applied at an early stage to make an exploratory analysis of an optimization problem. In general, metaheuristics require the iterative application of a sampling method and a selection criterion; first, the algorithm generates viable solutions belonging to the search space using a predefined strategy, and, successively, it selects one solution from the sampled space to use in the next iteration.

One of the most successful metaheuristics applied to an optimization problem is simulated annealing (SA), it has been presented in [Keith et al., 2002]. Simulated Annealing is a generalization of Monte Carlo methods, originally proposed by [Metropolis and Ulam, 1949] as a means of finding the equilibrium configuration of a collection of atoms at a given temperature.

The basic idea of SA was taken from an analogy with the annealing process used in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Methods based on SA apply a probabilistic mechanism to escape local minima: the underlying idea is to accept, under certain conditions, not only transitions that improve the objective function value, but also transitions that do not. The probability of accepting worsening steps varies during the search phase,

and it slowly decreases to zero. In the original Metropolis scheme, an initial state (or solution) is chosen, having energy E and temperature T. Keeping T constant, such initial configuration is perturbed, and the energy change ΔE is computed. If ΔE is negative, the new solution is always accepted. Otherwise, it is accepted with a probability given by the Boltzmann factor $e^{-(\Delta E/T)}$. This process is repeated L times for the current temperature, then the temperature is decremented and the entire process is repeated until a frozen state is reached at $T = 0$. At the beginning of the search, when temperatures are high, the algorithm behaves like a random search and therefore bad solutions can be accepted; whereas for lower values of T, solutions are located in promising regions of the search space. Like genetic algorithms for the CSP, a solution in SA is a sequence of characters, and solutions minimizing the Hamming distance value have a higher probability of being accepted for the next iteration of the algorithm.

One of the first meta heuristic algorithm for CSP, a Genetic Algorithm (GA), was proposed by Mauch et al. [Mauch et al., 2003], which was combined with simulated annealing and applied parallel strategies by Liu et al. in 2005 [Liu et al., 2005]. Moreover, an Integer Programming (IP) formulation and a heuristic (Heur) for CSP was posed by Meneses et al. [Meneses et al., 2004], improved their heuristic using parallel multi-start strategy by Gomes et al. [Gomes et al., 2008]. In addition Liu et al. further improved their heuristic in a compounded genetic and simulated annealing algorithm[Liu et al., 2008]. A more recent Genetic Algorithm, based on a technique called data-based coding, was proposed by Julstrom [Julstrom, 2009]. Faro and Pappalardo proposed an Ant Colony Optimization (ACO) algorithm, called Ant-CSP, which was shown to outperform individual GA and SA algorithms over extensive random benchmarks [Faro and Pappalardo, 2010]. Also, in the same year, a Memetic Algorithm (MA) was proposed for the problem by Babaie and Mousavi [Babaie and Mousavi, 2010]. Liu et al. in 2011 proposed a polynomial-time heuristic resulting from a combination of local search strategies and an approximation algorithm called Largest Distance Decreasing Algorithm (LDDA)[Liu et al., 2011]. At the same time, a new algorithm for the problem based on a GRASP using the probabilistic heuristic function was proposed by Mousavi and Esfahani [Mousavi and Esfahani, 2012]. Pappalardo et al. proposed a new heuristic for the problem that allows one to locate a good starting solution for the SA method [Pappalardo et al., 2014]. Finally, Nienkötter et al. proposed a heuristic using the distance-preserving vector space, it is an embedding method applied to CSP [Nienkötter and Jiang, 2016]. Table 3.6 provides a summary for those heuristics ranked by year of their publications.

One of the last published articles belonging to this category was proposed by Nienkötter et al., in essence, this article presents a prototype-embedding approach [Ferrer et al., 2010] for generalized median computation. In this method, the objects are embedded into a Euclidean metric space using an embedding function. In the end, the median in vector space is transformed back into the original problem space (reconstruction), resulting in an approximation of the searched generalized median. The vector space embedding framework is a general approach to consensus computation without knowing the structure of space. It is an application of this framework.

Moreover, for purposes of comparison between the metaheuristics and our proposed method, we must have access to the instances tested from those articles. Those instances were not published by the authors. For that reason, we could not make a comparison between those methods. On the other hand, our proposed algorithm called polynomial greedy algorithm (GREEDY) exploit the mathematical structure of CSP, it is not fair to compare any metaheuristic with GREEDY. The computational results demonstrate the good approximation for an optimal solution.

Table 3.6: A list of articles related to heuristics for CSP.

Title	Strategy	Cited
Genetic algorithm	GA	[Mauch et al., 2003]
Branch and Bound algorithm and heuristic	IP & Heur	[Meneses et al., 2004]
Parallel genetic and simulated annealing algorithm	PGSA	[Liu et al., 2005]
A compounded genetic and simulated annealing algorithm	CGSA	[Liu et al., 2008]
A parallel local-search heuristic	PLS	[Gomes et al., 2008]
A genetic algorithm and data-based coding	GA	[Julstrom, 2009]
An ant colony optimization algorithm	ACO	[Faro and Pappalardo, 2010]
Memetic algorithm	MA	[Babaie and Mousavi, 2010]
Heuristic and local search strategies	HLSS	[Liu et al., 2011]
A GRASP algorithm using a probability-based heuristic	GRASP	[Mousavi and Esfahani, 2012]
A greedy-walk heuristic and simulated annealing algorithm	GW & SA	[Pappalardo et al., 2014]
Heuristic based on distance-preserving vector space	DPVS	[Nienkötter and Jiang, 2016]

3.6 Matheuristics

A new category of metaheuristics is formed by matheuristics, that use the solution of a relaxed model to create a feasible solution following a specific heuristic criterion. Matheuris-

tics are often known to give good results, without considering complex approaches. They perform quite well in the CSP context, especially the ones that are based on the linear continuous relaxation of the model. Table 3.7 provides a summary for those matheuristics ranked by year of their publications.

The first algorithm of this category is a greedy approach to solve an Integer Programming (GIRT) called greedy iterative rounding technique was proposed by Chen [Chen, 2007], in this algorithm, if the number of strings is limited to 3, the algorithm is probably at most 1 away from the optimum, in many cases it can find an exact solution, even though it fails to find an exact solution, the solution found is very close to optimal solution. Della Croce and Salassa described three relaxation-based procedures. One procedure (RA) rounds up the result of continuous relaxation, while the other two approaches (BCPA and ECPA) fix a subset of the integer variables in the continuous solution at the current value and let the solver run on the remaining (sub)problem [Della Croce and Salassa, 2012]. A new technique Lagrangian relaxation to the problem formulated as an integer programming using Tabu Search was proposed by Tanaka [Tanaka, 2012]. Moreover, Croce and Garraffa designed a multi-start relaxation-based algorithm (called the selective fixing algorithm) that for a predetermined number of iterations takes a feasible solution as input and iteratively selects variables to be fixed at their initial value until the number of free variables is small enough that the remaining subproblem can be efficiently solved to optimality by an ILP solver [Croce and Garraffa, 2014].

Table 3.7: A list of articles related to matheuristics for CSP.

Title	Strategy	Cited
Relaxation-based and iterative rounding technique	LP & IRT	[Chen, 2007]
Relaxation-based procedures	LP & RBP	[Della Croce and Salassa, 2012]
Linear programming and lagrangian relaxation	LP & LR	[Tanaka, 2012]
Relaxation-based and selective fixing algorithm	LP & SFA	[Croce and Garraffa, 2014]

3.7 Remarks

In the literature review, we learned many important techniques for implementing and designing the proposed algorithms. By example, the column-position type concept, also called tuple was proposed by [Gramm et al., 2001, Amir et al., 2016], it contributes to implementing the MFA approach. The 1-mismatch block posed by [Amir et al., 2016],

the normalization process presented by [Gramm et al., 2001], and the identification of tuples for four binary strings posed by [Boucher et al., 2009], they contribute to implementing the MSA approach. Finally to implement the recursive exact method we use the greedy walk posed by [Pappalardo et al., 2014], the pre-processing method proposed by [Dalpasso and Lancia, 2018], and the minority and majority identification for each column-position type presented by [Amir et al., 2016].

This chapter has provided the literature review necessary for solving the Closest String Problem, the formal definition for the problem, formulations in IP, approximation algorithms, fixed parameter complexity, meta-heuristics, and matheuristics were summarized.

Chapter 4

PROPOSED METHODS

One strategy to tackle with NP-hard problems is to solve using fixed-parameter complexity to get optimal solutions in polynomial running time for some fixed parameter, with this objective, this chapter reports the proposed algorithms theoretically suggested by formal proofs, and two IP formulations for 3-CSP and 4-CSP.

4.1 A linear-time algorithm with up to three strings

In this section we are interested in optimal solutions for the Closest String with three sequences. With this objective, this work gives the proof of correctness for a linear-time algorithm to solve 3-CSP with an arbitrary alphabet, it always calculates the optimal solution value.

Notation. Throughout this section, we will be considering our input instance as a matrix. If Set $\mathcal{S}$ has n strings, $s^1, \ldots, s^n$, each of length m, we view $\mathcal{S}$ as an $n \times m$ matrix. We can thus refer to columns and rows. Thus, e.g., the element in the second row of column j, is the jth symbol of s^2. The distance of a string s' from $\mathcal{S}$ is $\max_{s \in \mathcal{S}} d_H(s', s)$. Given a string s we denote with (s_j) the character in the jth position of s. We refer to two identical columns as having the same column-position type. We denote with $V(\mathcal{S})$ the set of column-position types from $\mathcal{S}$. For a column type $v_i \in V(\mathcal{S})$ we denote with n_i the number of column-position types v_i in $\mathcal{S}$ with $i = 0, \ldots, |V(\mathcal{S})|$. We indicate with $v_0, \ldots, v_4$ the five column-position types obtained from the input 3-CSP instance with an arbitrary alphabet, a column-position type is also named as tuple (previously-defined in [Gramm et al., 2001]).

In order to compare our proposed algorithm, we elaborate an IP formulation for 3-CSP, and also we report the theoretical analysis of the Minimization First Algorithm (MFA)

4.1.1 An IP formulation for 3-CSP

Gramm et al. solved the decision version of this IP directly using the algorithm of Lenstra [Lenstra, 1983] that has an exponential dependency in the number of variables. Thus, they were not able to solve the IP for more than four strings. The authors suggested an IP-formulation based on column-position types, this yields fixed-parameter tractability for CLOSEST STRING with respect to parameter k (number of strings)[Gramm et al., 2001].

Define n_0 through n_4 as the number of column-position types v_0 through v_4. In columns of type v_0, you clearly always pick α. Otherwise, you need to assign $x_{\alpha,1}$ of the columns of type v_1 to α (and the rest, $x_{\beta,1} = n_1 - x_{\alpha,1}$, to β), $x_{\alpha,2}$ columns of type v_2 to α (and the rest, $x_{\beta,2} = n_2 - x_{\alpha,2}$ to β), $x_{\alpha,3}$ columns of type v_3 to α (and the rest, $x_{\beta,3} = n_3 - x_{\alpha,3}$ to β), and $x_{\alpha,4}, n_{\beta,4}, x_{\gamma,4}$ columns of type v_4 to α, β, γ respectively (with $x_{\alpha,4} + x_{\beta,4} + x_{\gamma,4} = n_4$). Clearly, given $x_{\alpha,1}$ through $x_{\gamma,4}$, the process of "normalize the columns, reorder the strings if necessary, make the selections, un-normalize the columns" takes time $O(m)$[Gramm et al., 2003]. The numbers $x_{\alpha,1}$ through $x_{\gamma,4}$ are the solution to the following IP:

$$
\begin{aligned}
\min \quad & d \\
s.t.: \quad & x_{\alpha,1} + x_{\beta,2} + x_{\beta,3} + x_{\beta,4} + x_{\gamma,4} \leq d \\
& x_{\beta,1} + x_{\alpha,2} + x_{\beta,3} + x_{\alpha,4} + x_{\gamma,4} \leq d \qquad (4.1) \\
& x_{\beta,1} + x_{\beta,2} + x_{\alpha,3} + x_{\alpha,4} + x_{\beta,4} \leq d \\
& \sum_{\sigma \in v_j} x_{\sigma,j} = n_j \quad j = 1, \ldots, 4 \qquad (4.2) \\
& x_{\sigma,j} \in \{0, 1, \ldots, n_j\} \qquad (4.3) \\
& d \in \mathbb{Z}_+. \qquad (4.4)
\end{aligned}
$$

In this formulation, for every column position type $v_j \in V(S)$ where $j = 1, \ldots, 4$ and $\sigma \in v_j$, the variable $x_{\sigma,j}$ is the number of occurrences that σ has in the closest string at locations that correspond to column type v_j. Let $s^i \in S$ where $i = 1, \ldots, 3$ be the input-instance, and let σ be the character of string s^i in column-position type v_j. The restrictions (4.1) calculate the Hamming distance of s^i from the center string, that is, for each column type v_j we sum the characters in the center string at locations that correspond to v_j that are different from the character s^i has in these locations. Constraints (4.2) impose the number of column types existed in each tuple, and (4.3) makes $x_{\sigma,j}$ have integer values. Finally, the objective is to minimize the value d.

4.1.2 CS3 Efficient linear-time solution for $k = 3$

The Gramm's algorithm for 3 strings with an arbitrary alphabet is a recursive algorithm, the input instance is transformed into a normalized one, then it is reordered the columns of the $k \times m$ matrix and considered consecutive columns in the reordered instance as a block. By sorting, the columns are already ordered in the sequence in which we will process them:

(0) Identity Case. All columns of type $[a, a, a]^t$,

(1) Diagonal Case. All blocks of type $[baa, aba, aab]^t$,

(2) 3/2 Letters Case. All blocks of type $[aa, ba, cb]^t$, $[aa, bb, ca]^t$, or $[ab, ba, ca]^t$ (the order of these three types among each other does not matter),

(3) 2/2 Letters Case. All blocks of type $[aa, ab, ba]^t$, $[ab, aa, ba]^t$, or $[ab, ba, aa]^t$ (we can find only one of these three possibilities, since, otherwise, we would have been able to build an additional block of type (1)),

(4) Remaining 2 Letters Case. All blocks of type $[a, a, b]^t$, $[a, b, a]^t$, or $[b, a, a]^t$ (as in case (3), we can find only one of these possibilities, since, otherwise, we would have been able to build an additional block in (3)),

(3') 3/3 Letters Case. All blocks of type $[aaa, bbb, ccc]^t$,

(4') Remaining 3 Letters Case. All blocks of type $[a, b, c]^t$.

Algorithm 4: CS3: Gramm's algorithm for $k = 3$

Input: $\mathcal{S} = \{s^1, s^2, s^3\}$: a normalized and reordered by blocks instance.

Output: `CS3`($\mathcal{S}$) : an optimal solution

1 **Function** `CS3`($\mathcal{S}$)

2 ($K0$) Given $[aaa]^t$ then **return** a.`CS3`(S')

3 ($K1$) Given $[baa, aba, aab]^t$ then **return** aaa.`CS3`(S')

4 ($K2$) Given $[aa, ba, cb]^t$ or $[aa, bb, ca]^t$ or $[ab, ba, ca]^t$ then **return** ca.`CS3`(S'), or ba.`CS3`(S'), or aa.`CS3`(S')

5 ($K3$) Given $[aa, ab, ba]^t$, or $[ab, aa, ba]^t$ or $[ab, ba, ca]^t$ then **return** aa.`CS3`(S')

6 ($k3'$) Given $[aaa, bbb, ccc]^t$ then **return** abc.`CS3`(S')

7 ($K4'$) Given $[aa, bb, ca]^t$ or $[a, b, c]^t$ then **return** ab.`CS3`(S'), or a.`CS3`(S').

The Algorithm 4 is a recursive algorithm, it creates a normalized one and reordering it, after that, recursively it fixes the characters in the current solution according to the different cases $(K0)(K1)(K2)(K3)(K4)$ or $(K0)(K1)(K2)(K3')(K4')$. Finally, to built the optimal solution to the original instance makes the reverse operation over the current solution.

4.1.3 MFA Efficient linear-time algorithm for 3-strings

In the Minimization First Algorithm (MFA), the identification of column-position types is needed, it breaks up into five tuples, finally, it decides a character for each column-position by simple evaluation through the different cases according to the number of tuples presented in the input instance. MFA can find the optimal solution by traversing all the positions of the input strings only once. With the algorithm we obtain, the optimal value $\lceil max_{i,j=1,2,3} d_H(s^i, s^j)/2 \rceil$ and when the number of "all mismatches" tuples is greater than the others, its optimal value is $\lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil$.

In this algorithm w.l.o.g. consider the pairwise distances of strings s^1, s^2 and s^3 satisfy $d_H(s^1, s^2) \geq d_H(s^1, s^3) \geq d_H(s^2, s^3)$, that is, string s^1 is farthest and s^3 is closest to the other two strings. In Algorithm 5, if the number of "all mismatches" tuples is zero, we have the binary case, then count is $\lfloor (d_H(s^{u_1}, s^{u_3}) - d_H(s^{u_2}, s^{u_3}))/2 \rfloor$. Otherwise, if $k_1 + k_2 > n_4$ and $k_2 > n_4$, then k_1 is 0, k_2 is n_4, and count is $\lfloor (k_1 - k_2 + n_4)/2 \rfloor$; if $k_1 + k_2 > n_4$ and $k_2 \leq n_4$, then k_1 is $n_4 - k_2$, and count is $\lfloor (k_1 + k_2 - n_4)/2 \rfloor$; finally, if $k_1 + k_2 < n_4$, then count is zero.

Algorithm 5 assigns the characters of s^1 to the solution x among the positions where (all mismatches but $k_1 > 0$) or (s^1 matches s^2 or s^1 mismatches s^3 but count> 0), and once count is reduced to zero, it fixes the characters of s^3 in the solution x. It assigns the characters of s^2 to the solution x among the positions where all mismatches but $k_2 > 0$, finally once k_1 and k_2 are reduced to zero, it fixes the characters of the set of strings in a round-robin order in the solution x among the positions where all mismatches.

Running time

The if-then conditional of lines 1-31 is executed when the input-instance has 3-sequences, it requires 1 step. Lines 2 and 3 sort the pairwise distances from s^1, s^2, and s^3, it requires $4m$ iterations to get the Hamming distances and c_1 steps to sort them in non-decreasing order by its Hamming distances which is constant time. Lines 4 and 5 make arithmetic operations over the Hamming distances, it takes a constant time c_2. In Line 7, the n_4 variable counts the number of times that the column-position type v_4 (all mismatches) are presented in the input-instance, it takes m steps. The if-then-else conditional of lines 8-16 makes arithmetic operations, it takes a constant time c_3. The for-loop of lines 17-31 iterates in m steps, so it requires $3m$ iterations. The if-then-else conditional of lines 18-31 makes arithmetic operations, it takes a constant time c_4. Therefore, the running time of MFA is proven to be $O(8m)$.

Algorithm 5: MFA pseudo-code.

Input: $\mathcal{S} = \{s^1, s^2, s^3\}$: a 3-CSP instance with m the length of strings.

Output: x: optimal solution such that $d_H(x, s) \leq d_{opt}\ \forall s \in \mathcal{S}$.

```
if |S| = 3 then
    Sort the pairwise distances of s^1, s^2 and s^3.
    Let u_1,u_2 and u_3 satisfy d_H(s^{u_1}, s^{u_2}) ≥ d_H(s^{u_1}, s^{u_3}) ≥ d_H(s^{u_2}, s^{u_3})
    k_1 ← d_H(s^{u_1}, s^{u_2}) − d_H(s^{u_2}, s^{u_3})
    k_2 ← d_H(s^{u_1}, s^{u_2}) − d_H(s^{u_1}, s^{u_3})
    k_3 ← 0 ;        // counter variable that iterates over the 3-strings
    Let n_4 be the number of column positions when all mismatches
    if n_4 = 0 then
        count ← ⌊(d_H(s^{u_1}, s^{u_3}) − d_H(s^{u_2}, s^{u_3}))/2⌋ ;    // Case 1, binary case
    else if k_1 + k_2 > n_4 then
        if k_2 > n_4 then
            count ← ⌊(k_1 − k_2 + n_4)/2⌋; k_1 ← 0; k_2 ← n_4 ;          // Case 3
        else
            count ← ⌊(k_1 + k_2 − n_4)/2⌋; k_1 ← n_4 − k_2 ;            // Case 4
    else
        count ← 0 ;                                                     // Case 2
    for i=1 to m do
        if (s_i^{u_1} ≠ s_i^{u_3} and s_i^{u_1} ≠ s_i^{u_2} and s_i^{u_2} ≠ s_i^{u_3}) then
            if (k_1 > 0 or k_2 > 0) then
                if (k_1 > 0) then
                    x_i ← s_i^{u_1}; k_1 ← k_1 − 1
                else if (k_2 > 0) then
                    x_i ← s_i^{u_2}; k_2 ← k_2 − 1
            else
                x_i ← s_i^{u_{k_3+1}}; k_3 ← (k_3 + 1)%3
        else if (s_i^{u_1} = s_i^{u_2} or (s_i^{u_1} ≠ s_i^{u_3} and count > 0)) then
            x_i ← s_i^{u_1}
            if (s_i^{u_1} ≠ s_i^{u_3} and s_i^{u_2} = s_i^{u_3}) then
                count ← count −1
        else
            x_i ← s_i^{u_3}
```

Theoretical analysis

We first introduce three lemmas, then use them to prove that MFA can find an optimal solution of 3-CSP with an arbitrary alphabet.

Lemma 1. *Let $\mathcal{S} = \{s^1, s^2, s^3\}$ be a 3-CSP instance with length m. If string x is an optimal solution of the CSP and d_{opt} is the corresponding distance, then $\lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil \leq d_{opt}$.*

Proof. Independently of whether the character appears in the column position j in the string solution x, it mismatches with minimum 1 character for v_1, v_2, v_3 and with minimum in 2 characters for v_4. The sum of these values is equal to the Hamming distance between x and $s^i \in \mathcal{S}$; dividing it by 3 we get the average Hamming distance. □

Lemma 2 ([Liu et al., 2011])**.** *Let $\mathcal{S}$ be an instance of the CSP. If string x is an optimal solution of the CSP and d_{opt} is the corresponding distance, then $d_{opt} \geq \lceil max_{i,j=1,\ldots,k} d_H(s^i, s^j)/2 \rceil$.*

Lemma 3. *Let $\mathcal{S} = \{s^1, s^2, s^3\}$ be a 3-CSP instance with length m. The number of columns satisfies the restriction $n_1 \geq n_2 \geq n_3$.*

Proof. From (4.5)-(4.9), we have $d_H(s^1, s^2) = n_1 + n_2 + n_4$, $d_H(s^1, s^3) = n_1 + n_3 + n_4$, and $d_H(s^2, s^3) = n_2 + n_3 + n_4$. Assume without loss of generality that $d_H(s^1, s^2) \geq d_H(s^1, s^3) \geq d_H(s^2, s^3)$. After making arithmetic operations, we have, $n_2 \geq n_3$ and $n_1 \geq n_2$. Finally, we get, $n_1 \geq n_2 \geq n_3$. □

Theorem 1. *Let 3-CSP be an instance of CSP with 3 sequences and an arbitrary alphabet. MFA always finds an optimal solution value to 3-CSP.*

Proof. The proof is composed by four cases, it is made by direct method.

$$v_0 \quad s_j^1 = s_j^2 = s_j^3 \qquad \text{all matches} \tag{4.5}$$

$$v_1 \quad s_j^1 \neq s_j^2 = s_j^3 \qquad s_j^1 \text{ is the minority} \tag{4.6}$$

$$v_2 \quad s_j^2 \neq s_j^1 = s_j^3 \qquad s_j^2 \text{ is the minority} \tag{4.7}$$

$$v_3 \quad s_j^3 \neq s_j^1 = s_j^2 \qquad s_j^3 \text{ is the minority} \tag{4.8}$$

$$v_4 \quad s_j^1 \neq s_j^2, s_j^1 \neq s_j^3, \text{ and } s_j^2 \neq s_j^3 \qquad \text{all mismatches} \tag{4.9}$$

From (4.5)-(4.9), we have the identification of column-position types, also called tuples. Observe that, some column positions presented in the input instance are repeated, therefore, the number of different tuples presented in any input-instance is equals to five.

Consider the alignment of the three strings s^1, s^2, and s^3. Note that, from (4.5)-(4.9), we have, in general, for any input-instance there are five different column-position types. According to Lemma 3, we get $n_1 \geq n_2 \geq n_3$. It follows that:

Case 1. If $n_4 = 0$, we have the binary case [Liu et al. proved it [Liu et al., 2011]]. Then in Algorithm 5, the initial value of variable **count** is equal to $\lceil (n_1 - n_2)/2 \rceil$. Note that $|\Gamma| = 2$ and so either $s_j^3 = s_j^1$ or $s_j^3 = s_j^2$ among the positions where s^1 mismatches s^2. Hence $d_H(s^1, s^2) = n_1 + n_2$. The solution x of MFA is decided by :

(1.1) Among the positions where s^1 matches s^2, $x_j = s_j^1 = s_j^2$

(1.2) Among the positions where s^1 mismatches s^2:

$$\begin{aligned} d_H(x, s^1) &\leq n_1 - \lceil (n_1 - n_2)/2 \rceil \\ d_H(x, s^2) &\leq n_2 + \lceil (n_1 - n_2)/2 \rceil \\ d_H(x, s^3) &\leq n_3 + \lceil (n_1 - n_2)/2 \rceil \leq n_3 + \lceil (d_H(s^1, s^2) - 2n_2)/2 \rceil \\ &\leq n_3 - n_2 + \lceil d_H(s^1, s^2)/2 \rceil \leq \lceil d_H(s^1, s^2)/2 \rceil. \end{aligned} \tag{4.10}$$

Taken together, sub-cases (1.1) and (1.2) give a lower bound, that is, $max_{i=1,2,3} d_H(x, s^i) \leq d_H(s^1, s^2)/2$ (See Figure 4.1(a)).

Case 2 If $n_1 - n_3 + n_2 - n_3 \leq n_4$. Then in Algorithm 5, the initial value of variable **count** is equal to 0, $k_1 = n_1 - n_3$, $k_2 = n_2 - n_3$. Note that $|\Gamma| = 3$. The solution x of MFA is decided as:

(2.1) Among the positions where there are at least two different symbols, we have $x_j = argmax_{\sigma \in \Gamma} \sum_{i=1}^{3} |s_j^i = \sigma|$

(2.2) Among the positions where all characters mismatches:

$$\begin{aligned} d_H(x, s^1) &\leq n_3 + k_1 + k_2 + \lceil 2(n_4 - k_1 - k_2)/3 \rceil \\ d_H(x, s^2) &\leq n_3 + k_1 + k_2 + \lceil 2(n_4 - k_1 - k_2)/3 \rceil \\ d_H(x, s^3) &\leq n_3 + k_1 + k_2 + \lceil 2(n_4 - k_1 - k_2)/3 \rceil \\ &\leq n_1 + n_2 - n_3 + \lceil 2(n_4 - n_1 - n_2 + 2n_3)/3 \rceil \\ &\leq \lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil. \end{aligned} \tag{4.11}$$

From (2.1) and (2.2), and by the Lemma 1, we obtain, $max_{i=1,2,3} d_H(x, s^i) \leq \lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil \leq d_{opt}$ (See Figure 4.1(b)).

Case 3 If $n_1 - n_3 + n_2 - n_3 > n_4$ and $n_2 - n_3 > n_4$. Then in Algorithm 5, the initial value of variable **count** is equal to $\lceil (n_1 - n_3 - (n_2 - n_3 - n_4))/2 \rceil$, $k_1 = 0$, $k_2 = n_4$. Note that $|\Gamma| = 3$ and so either $s_j^3 = s_j^1$ or $s_j^3 = s_j^2$ among the positions where s^1 mismatches s^2, $d_H(s^1, s^2) = n_1 + n_2 + n_4$. The solution x of MFA is decided as:

(3.1) Among the positions where s^1 matches s^2, $x_j = s_j^1 = s_j^2$

(3.2) Among the positions where s^1 mismatches s^2:

$$\begin{aligned} d_H(x,s^1) &\leq n_1 - \lceil (n_1 - n_3 - (n_2 - n_3 - n_4))/2 \rceil + n_4 \\ d_H(x,s^2) &\leq n_2 + \lceil (n_1 - n_3 - (n_2 - n_3 - n_4))/2 \rceil \\ d_H(x,s^3) &\leq n_3 + \lceil (n_1 - n_3 - (n_2 - n_3 - n_4))/2 \rceil + n_4 \\ &\leq n_3 + \lceil (n_1 - n_2 + n_4)/2 \rceil + n_4 \\ &\leq \lceil (n_1 + n_2 + n_4)/2 \rceil \\ \text{Since } k_2 &= n_2 - n_3 = n_4 \text{ we get } n_3 = n_2 - n_4. \end{aligned} \tag{4.12}$$

From (3.1) and (3.2), and by the Lemma 2, we get, $max_{i=1,2,3} d_H(x,s^i) \leq d_H(s^1,s^2)/2 \leq d_{opt}$ (See Figure 4.2(a)).

Case 4 If $n_1 - n_3 + n_2 - n_3 > n_4$ and $n_2 - n_3 \leq n_4$. Then in Algorithm 5, the initial value of variable **count** is equal to $\lceil (n_1 - n_3 - (n_4 - n_2 + n_3))/2 \rceil$, $k_1 = n_4 - n_2 + n_3$, $k_2 = n_2 - n_3$. Note that $|\Gamma| = 3$ and so either $s_p^3 = s_j^1$ or $s_j^3 = s_j^2$ among the positions where s^1 mismatches s^2, $d_H(s^1,s^2) = n_1 + n_2 + n_4$. The solution x is decided as:

(4.1) Among the positions where s^1 matches s^2, $x_j = s_j^1 = s_j^2$

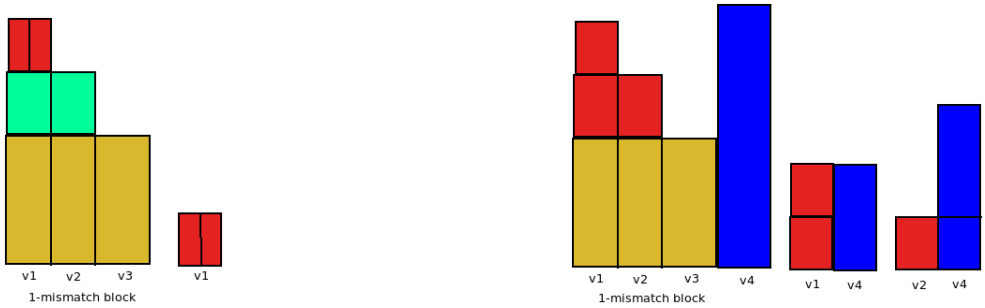
(4.2) Among the positions where s^1 mismatches s^2:

$$\begin{aligned} d_H(x,s^1) &\leq n_1 - \lceil (n_1 - n_3 - (n_4 - n_2 + n_3))/2 \rceil + n_2 - n_3 \\ d_H(x,s^2) &\leq n_2 + \lceil (n_1 - n_3 - (n_4 - n_2 + n_3))/2 \rceil + n_4 - n_2 + n_3 \\ d_H(x,s^3) &\leq n_3 + \lceil (n_1 - n_3 - (n_4 - n_2 + n_3))/2 \rceil + n_4 \\ &\leq \lceil (n_1 + n_2 + n_4)/2 \rceil. \end{aligned} \tag{4.13}$$

Both (4.1) and (4.2), and by the Lemma 2, we obtain, $max_{i=1,2,3} d_H(x,s^i) \leq d_H(s^1,s^2)/2 \leq d_{opt}$ (See Figure 4.2(b)).

Altogether Cases 1, 3, and 4 force that the solution x of MFA is an optimal one and the optimal solution is $\lceil max_{i,l=1,2,3} d_H(s^i,s^l)/2 \rceil$. Also, the optimal solution for Case 2 is $\lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil$. Thus the theorem holds. □

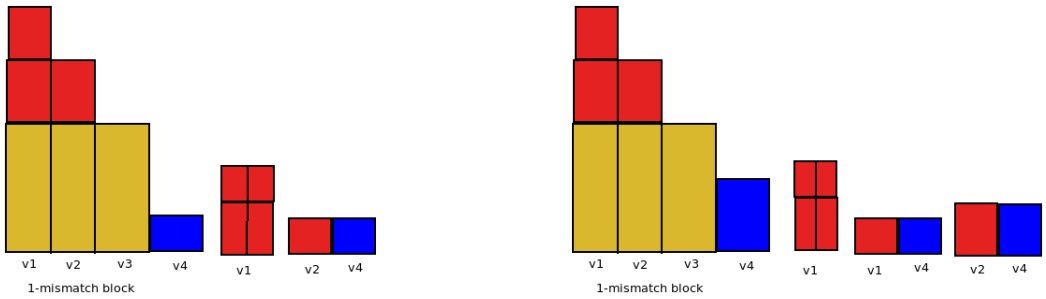
Illustration of application of Case 1, 3-CSP with a binary alphabet, there are three column types to be considered v_1, v_2, and v_3. The algorithm fixes 1-mismatch blocks, in goldenrod orange and green colors, by their majority consensus value, after that, it assigns column type v_1 (in red color), half of them by their majority and for other one by their minority consensus value. Illustration of application of Case 2, there are four column types to be considered v_1, v_2, v_3, and v_4.The algorithm fixes 1-mismatch blocks, in goldenrod orange color, by their majority consensus value, after that, it assigns v_1 and v_4 column



(a) Case 1 **(b)** Case 2

Figure 4.1: Illustration of application of Cases 1 and 2. Source: the authors.

types by their majority consensus value and the character of s^1; it fixes v_2 and v_4 column types by their majority consensus value and the character of s^2; finally it assigns the rest of column types v_4 in a round-robin order. Source: the authors.



(a) Case 3 **(b)** Case 4

Figure 4.2: Illustration of application of Cases 3 and 4. Source: the authors.

Illustration of application of Case 3, there are four column types to be considered v_1, v_2, v_3, and v_4.The algorithm fixes 1-mismatch blocks, in goldenrod orange color, by their majority consensus value, after that, it assigns v_2 and v_4 column types by their majority consensus value and the character of s^2; it fixes column type v_1 (in red color), half of them by their majority and for other one by their minority consensus value. Illustration of application of Case 4, there are four column types to be considered v_1, v_2, v_3, and v_4.The algorithm fixes 1-mismatch blocks, in goldenrod orange color, by their majority consensus value, after that, it assigns v_1 and v_4 column types by their majority consensus value and the character of s^1; it fixes v_2 and v_4 column types by their majority consensus value and the character of s^2; finally it fixes column type v_1 (in red color), half of them by their majority and for other one by their minority consensus value. Source: the authors.

Example 1. *Let $\mathcal{S}$ be a CSP instance with 3-strings and each string of length 10, $\mathcal{S}'$ is obtained from $\mathcal{S}$ where the set of strings was ordered in a pairwise manner by their*

Hamming distances, we get:

$$\mathcal{S} = \begin{cases} AGTATTGGTG \\ CCCTTTGAGA \\ TAGTGGGTCT \end{cases} \qquad \mathcal{S}' = \begin{cases} TAGTGGGTCT \\ AGTATTGGTG \\ CCCTTTGAGA \end{cases}$$

$$\mathcal{S}' = \left\{ \begin{array}{ccc|cccc|ccc} v_4 & v_4 & v_4 & v_2 & v_1 & v_1 & v_0 & v_4 & v_4 & v_4 \\ \hline \mathbf{T} & \mathbf{A} & G & \mathbf{T} & G & G & \mathbf{G} & \mathbf{T} & C & T \\ A & G & \mathbf{T} & A & \mathbf{T} & \mathbf{T} & \mathbf{G} & G & \mathbf{T} & G \\ C & C & C & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{G} & A & G & \mathbf{A} \\ \hline T & A & T & T & T & T & G & T & T & A \end{array} \right.$$

The number of column-position types which all mismatches is $n_4 = 6$, the three counters have the following values $k_1 = n_1 - n_3 = 2$, $k_2 = n_2 - n_3 = 1$, and count $= 0$, with these values, MFA identifies the Case 2, thus the optimal solution for S' or S is $x =$ TATTTTGTTA with Hamming distance $d_H(x, s) \leq 5, \quad \forall s \in \mathcal{S}$. Note that, the optimal solution value for the Case 2 is $\lceil (n_1 + n_2 + n_3 + 2n_4)/3 \rceil = \lceil (2 + 1 + 0 + 2(6))/3 \rceil = 5$.

4.1.4 Memory usage analysis in CS3 and MFA

The analysis of memory usage between CS3 algorithm (proposed by [Gramm et al., 2001]) and MFA (our proposal) as follows. CS3 is a recursive algorithm, it makes pre-processing over input-instance (a character matrix with dimensions $k \times m$), and also it reorders the normalized instance by column-positions to forming blocks, finally for each block, the algorithm fixes the characters in the current solution, after that, it makes a recursive call for the rest of unfixed column positions, the recursive calls generate a stack in internal memory, in the worst case, the total number of recursive calls is $log(m)$, it does the reverse process over the current solution to obtain an optimal solution for the original instance, so the total used internal memory is $O(k \times mlog(m))$, since for each call the CS3 creates a copy of the processed instance.

On the other hand, MFA is an iterative algorithm, MFA iterates the input instance through its column-positions, so the internal memory used is a matrix of dimensions $k \times m$ plus a character vector of length m, so the total used internal memory is $O(k \times m)$.

4.1.5 Remarks

In the literature there is an algorithm for 3-strings with an arbitrary alphabet, we use an important definition, column-position types known as tuples proposed by [Gramm et al., 2001].

Based on that we propose an efficient algorithm for the special case of CSP with 3-sequences and alphabet size $|\Gamma| \geq 2$ and its corresponding theoretical analysis. We explain important differences between those algorithms, the CS3 algorithm has memory usage $O(k \times mlog(m))$, meanwhile, the MFA method has memory usage $O(k \times m)$. Both algorithm always find the optimal soluton values, their running time is $O(m)$. Furthermore for any input instance there are 5 tuples, we extend that definition to 4-CSP, in the next section will be presented.

4.2 A linear-time algorithm for four binary strings

Boucher [Boucher et al., 2009] proposed an exact method for the 4-CSP with a binary alphabet, it is a combinatorial characterization of decoy sets, it embedded a linear-time algorithm that easy to implement, this algorithm demonstrates a gap in finding the optimal solution value at most 1 unit. To the contrary, this paper presents a formal proof for a linear-time algorithm to solve the 4-CSP for binary alphabet, our method is based on column types, and it always calculates the optimal solution value.

Notation. Throughout this section, we will be considering our input instance as a matrix. If the set $\mathcal{S}$ has n strings, $s^1, \ldots, s^n$, each of length m, we view $\mathcal{S}$ as a $n \times m$ matrix. We can thus refer to columns and rows. Thus, e.g., the element is the second row of column j, is the jth symbol of s^2. The distance of a string s' from $\mathcal{S}$ is $\max_{s \in \mathcal{S}} d_H(s', s)$. Given a string s we denote with s_j the character in the jth position of s. We refer to two identical columns as having the same column type. We denote with $V(\mathcal{S})$ the set of column types from $\mathcal{S}$. For a column type $v_i \in V(\mathcal{S})$ we denote with n_i the number of columns of type v_i in $\mathcal{S}$ with $i = 0, \ldots, |V(\mathcal{S})|$. We indicate with $v_0, \ldots, v_7$ the eight column types obtained from the 4-CSP normalized instance.

Our proposed algorithm is compared with an integer programming formulation for 4-CSP, and also we report the theoretical analysis of the Minimization First Algorithm (MFA)

4.2.1 An IP formulation for 4-CSP

Gramm *et al.* in 2001 presented an IP formulation that yields fixed-parameter tractability for Closest String with respect to parameter n (number of strings) [Gramm et al., 2001]. Based on that formulation, we propose an IP formulation restricted to 4-CSP with a binary alphabet, it is based on column-position types, it always provides optimal solution values, also it converges more quickly than other formulations since it has few variables and restrictions.

Define n_0 through n_7 as the number of normalized columns of type v_0 through v_7. In columns of type v_0, you clearly always pick α. Otherwise, you need to assign $x_{\alpha,1}$ of the columns of type v_1 to α (and the rest, $x_{\beta,1} = n_1 - x_{\alpha,1}$, to β), $x_{\alpha,2}$ columns to type v_2 to α (and the rest, $x_{\beta,2} = n_2 - x_{\alpha,2}$ to β), and so on, until $x_{\alpha,7}$ columns of type v_7 to α (and the rest, $x_{\beta,7} = n_7 - x_{\alpha,7}$ to β). Clearly, given $x_{\alpha,1}$ through $x_{\beta,7}$, the process of "normalize the columns, reorder the strings if necessary, make the selections, un-normalize the columns"

takes time $O(m)$. The numbers $x_{\alpha,1}$ through $x_{\beta,7}$ are the solution to the following IP:

$$
\begin{aligned}
P1: \min \quad & d && (4.14)\\
\text{s.t.:} \quad & x_{\alpha,1} + x_{\beta,2} + x_{\beta,3} + x_{\beta,4} + x_{\beta,5} + x_{\beta,6} + x_{\beta,7} \leq d \\
& x_{\beta,1} + x_{\alpha,2} + x_{\beta,3} + x_{\beta,4} + x_{\beta,5} + x_{\alpha,6} + x_{\alpha,7} \leq d \\
& x_{\beta,1} + x_{\beta,2} + x_{\alpha,3} + x_{\beta,4} + x_{\alpha,5} + x_{\alpha,6} + x_{\beta,7} \leq d && (4.15)\\
& x_{\beta,1} + x_{\beta,2} + x_{\beta,3} + x_{\alpha,4} + x_{\alpha,5} + x_{\beta,6} + x_{\alpha,7} \leq d \\
& \sum_{\sigma \in v_j} x_{\sigma,j} = n_j \quad j = 1, \ldots, 7 && (4.16)\\
& x_{\sigma,j} \in \{0, 1, \ldots, n_j\} && (4.17)\\
& d \in \mathbb{Z}_+. && (4.18)
\end{aligned}
$$

In this formulation, any column-position type $v_j \in V(S)$ where $j = 1, \ldots, 7$ and $\sigma \in v_j$, the variable $x_{\sigma,j}$ is the number of occurrences that σ has in the closest string at locations that correspond to column type v_j. Let $s^i \in S$ where $i = 1, \ldots, 4$ be the input-instance, and let σ be the character of string s^i in column-position type v_j. The restrictions (4.15) calculate the Hamming distance of s^i from the center string, that is, for each column type v_j we sum the characters in the center string at locations that correspond to v_j that are different from the character s^i has in these locations. Constraints (4.16) impose the number of column types existed in each tuple, and (4.17) makes $x_{\sigma,j}$ to have integer values. Finally, the objective is to minimize the value d.

We note that the number of variables in $P1$ is the sum of the number of characters for each tuple in $(v_1, \ldots, v_7)$ plus d (the maximum Hamming distance), that is, 15 variables; and the number of constrains is the number of input-strings plus the number of different column-position types, that is, 11 restrictions.

4.2.2 Exact algorithm for four binary strings

Although the CSP for a constant number of strings is solvable in polynomial time using integer programming, this algorithm increases to huge running time even for moderate number of variables [Gramm et al., 2001]. Hence, a combinatorial characterization of decoy sets for the special case of CSP with $n = 4$ strings and $|\Gamma| = 2$ was proposed in [Boucher et al., 2009]. In opposition, MSA is based on column types.

Algorithm 6: MSA pseudo-code

Input: $\mathcal{S}$: a 4-CSP normalized instance $v_0, \ldots, v_7$

Output: x: optimal solution such that $\max(d[.]) = d_{opt}$

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
  for $j \leftarrow 1, \ldots, m$ **do**
    $u[j] \leftarrow -\infty$;
    for $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
      $n_i \leftarrow n_i + 1$;
      $u[j] \leftarrow i$;
  $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ `twoFirstGreatestNumber`$(n_1, \ldots, n_4)$
  $count \leftarrow \lfloor(\eta_1 - \eta_2 - \lfloor(m - (n_0 + \cdots + n_4))/2\rfloor)/2\rfloor$;
  for $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
    $x[j] \leftarrow$ `majority`$(\mathcal{S}[.][j])$;
  for $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ **do**
    if $count > 0$ **then**
      $x[j] \leftarrow$ `minority`$(\mathcal{S}[.][j])$;
      $count \leftarrow count - 1$;
  $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
  for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
    $i \leftarrow$ `indexMaximum`(d);
    $k \leftarrow$ `indexMinimum`(d);
    iNumberMax $\leftarrow$ `numberOfMaximum`$(d[i], d)$
    switch *iNumberMax* **do**
      case *1 or 4* **do**
        $x[j] \leftarrow \mathcal{S}[i][j]$;
      case *2 or 3* **do**
        if $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
        else
          $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
          $x[j] \leftarrow \mathcal{S}[r][j]$;
    $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$;

Algorithm 7: MSA Functions

Function `twoFirstGreatestNumber`$(n_1, \ldots, n_4)$**:**

$\eta_1, \ldots, \eta_4 \leftarrow \text{sort}(n_1, \ldots, n_4)$;

iCount $\leftarrow 1$;

foreach $i \in \{n_1, \ldots, n_4\}$ **do**

if $i = \eta_1$ **then** $I_{\eta_1} \leftarrow$ iCount ;

iCount $\leftarrow$ iCount+1

return $\eta_1, I_{\eta_1}, \eta_2$;

Function `indexMaximum`(d)**:**

iMax $\leftarrow 0$;

for $i \leftarrow 1, \ldots, 4$ **do**

if $iMax < d[i]$ **then**

iMax $\leftarrow d[i]$;

iIndexMax $\leftarrow i$

return iIndexMax ;

Function `indexMinimum`(d)**:**

iMin $\leftarrow \infty$;

for $i \leftarrow 1, \ldots, 4$ **do**

if $iMin > d[i]$ **then**

iMin $\leftarrow d[i]$;

iIndexMin $\leftarrow i$

return iIndexMin ;

Function `numberOfMaximum`$(iMax, d)$**:**

iCount $\leftarrow 0$;

for $i \leftarrow 1, \ldots, 4$ **do**

if $d[i] = iMax$ **then**
iCount $\leftarrow$ iCount+1

return iCount;

Function `updateHammingDistance`$(\mathcal{S}[.][j], \sigma, d[.])$**:**

for $i \leftarrow 1, \ldots, 4$ **do**

if $\mathcal{S}[i][j] = \sigma$ **then**
$d[i] \leftarrow d[i] - 1$

return $d[.]$;

The Minimization Second Algorithm MSA

In the Minimization Second Algorithm (MSA), normalization process is required to transform the input-instance into a normalized one, therefore it breaks up into eight tuples, lastly, it decides a character for each column-position in 1-mismatch blocks by its majority consensus value and the others by simple evaluation through the different cases according to the number of tuples presented in the normalized instance.

Algorithm 6 assigns the characters that are with the majority consensus value to the solution x among the column-positions that belong 1-mismatch block, it fixes the characters that are with the minority consensus value to the solution x among the column-positions that is the greatest number of tuples in 1-mismatch blocks where (count>0), and once count is reduced to zero, for each iteration the Hamming distances between the partially filled candidate x and the set of strings are updated, hence, it assigns the characters of the string with index of greatest Hamming distance among the positions where the number of maximum Hamming distance is 1 or 4, otherwise it fixes the characters of the string with index that is not the minimum or maximum Hamming distance in the current Hamming distances vector.

Theoretical analysis of MSA

We first introduce one lemma, a lower bound for column-position types (v_5, v_6, v_7), to prove that MSA can find an optimal solution value of $|\mathcal{S}| = 4$, and $|\Gamma| = 2$ type of CSP.

Lemma 4. *Let $\mathcal{S} = \{s^1, \ldots, s^4\}$ be a CSP instance with length m, where each column position belongs to v_5, v_6, v_7 column types. Let n_5, n_6, n_7 denote their number of columns. If string t is an optimal solution of the CSP and d_{opt} is the corresponding distance, then : $\lceil (n_5 + n_6 + n_7)/2 \rceil \leq d_{opt}$.*

Proof. Independently of whether the character appears in the column position j in the string solution t, it mismatches with minimum 2 characters for v_5, v_6, v_7. The sum of these values is equals to the Hamming distance between t and $s^i \in \mathcal{S}$, dividing it by 4 we get the average Hamming distance, since the average distance from consensus to the 4-strings is a lower bound on the maximum distance, we get the lemma. □

Theorem 2. *Let 4-CSP be an instance of CSP with 4 sequences and $\Gamma = \{\alpha, \beta\}$. MSA always finds an optimal solution value to 4-CSP.*

Proof. The proof is composed in two main parts, one is made by direct method, and the other one is demonstrated by induction method.

At first we construct a normalized instance from the input instance as follows. Let Γ_j be the alphabet of c_j we ordered the symbols in Γ_j according to their frequency in c_j in non-decreasing order, we get, $f_{\sigma_1} \geq f_{\sigma_2} \cdots \geq f_{\sigma_q}$. Let Γ'_j be an alphabet such that $|\Gamma_j| = |\Gamma'_j|$, let $\phi : \Gamma \to \Gamma'$ be a bijective function such that,

$$\phi(c_j)_{j=1,\ldots,m} = \begin{cases} \alpha & \text{if} \quad c_j^i = \sigma_1 \quad i = 1, \ldots, 4 \\ \beta & \text{if} \quad c_j^i = \sigma_2 \quad \sigma_q \in c_j; \;\; q = 1, 2 \end{cases} \tag{4.19}$$

Note that, after we apply the $\phi(c_j)$ function for each column position, we get only eight column types $v_i,\ i = 0, \ldots, 7$, where each column type is a set of column positions.

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
α	β	α	α	α	α	α	α
α	α	β	α	α	α	β	β
α	α	α	β	α	β	β	α
α	α	α	α	β	β	α	β

Let $\mathcal{P}$ be a block of column types, that is, $\mathcal{P} = v_0 \ldots v_4$ and let $\mathcal{Q}$ be a block of the rest of column types that is, $\mathcal{Q} = v_5 v_6 v_7$. $\mathcal{P}$ and $\mathcal{Q}$ are called affixes of $\mathcal{S}$, we have $\mathcal{S} = \mathcal{P} \odot \mathcal{Q}$. Let A and B be two partially filled candidates with the same length and non-overlapping sets of indices (i.e. if $A[j] =\text{␣}$, then $B[j] \neq \text{␣}$ & vice versa), that is, $\{A, B\} \in (\Gamma \cup \text{␣})^m$, the output of the function $A \odot B$ is a string where each character belongs to the alphabet Γ, more formally we have:

$$A \odot B = \begin{cases} A[j] & \text{if } B[j] = \text{␣} \quad j = 1, \ldots, m \\ B[p] & \text{if } A[p] = \text{␣} \quad p = 1, \ldots, m \end{cases} \tag{4.20}$$

Let n_i be the frequency of the v_i column type, $i = 0, \ldots, 7$. Suppose w.l.o.g. $n_1 \geq n_2 \geq n_3 \geq n_4$. Let **count** be a number of column positions belong to v_1, we have **count**$= n_1 - n_2 - \lfloor (n_5 + n_6 + n_7)/2 \rfloor$. Thus, we break it into two cases:

Case 1: If $n_1 - n_2 > \lfloor (n_5 + n_6 + n_7)/2 \rfloor \geq 0$, then **count**$= n_1 - n_2 - \lfloor (n_5 + n_6 + n_7)/2 \rfloor$. In this case n_1 is the greatest column type frequency, that is $d_H(s^1, s^2) = n_1 + n_2 + (n_5 + n_6 + n_7)/2$, since in v_1 and v_2 the characters are different and the other cases are equals and by the Lemma 4. Also we know that there not exits Hamming distance less than $(n_1 + n_2 + (n_5 + n_6 + n_7)/2)/2$, MSA determines a consensus value $n_1 - \mathbf{count}/2$. As a result we have to demonstrate that those values are equals, that is, $n_1 - (n_1 - n_2 - (n_5 + n_6 + n_7)/2)/2 \to (n_1 + n_2)/2 + (n_5 + n_6 + n_7)/4$, so we get, $(n_1 + n_2 + (n_5 + n_6 + n_7)/2)/2$.

Case 2: If $n_1 - n_2 \leq \lfloor (n_5 + n_6 + n_7)/2 \rfloor$, then **count**$\leq 0$. In this case, MSA does not make any processing in the $\mathcal{P}$ block.

Suppose that in the $\mathcal{P}$ block we do not consider the $\lfloor$**count**$/2\rfloor$ column positions which belong to the v_1 column type. Let x be a string of length m with unfixed characters, that is $x_j \in \Gamma \cup \{\sqcup\}$ with $1 \leq j \leq m$ we have:

$$x_j = \begin{cases} \sigma & \text{argmax}_{\sigma_k} \sum_{i=1}^{4} |s_j^i = \sigma_k| \quad \sigma_k \in \mathcal{P}_j \quad k = 1, \ldots, |\mathcal{P}_j| \quad \forall s^i \in \mathcal{P} \\ \sqcup & \text{otherwise} \end{cases} \tag{4.21}$$

Suppose that there exists $\lfloor$**count**$/2\rfloor$ column positions which belong to the v_1 column type, consider those column positions in the $\mathcal{P}$ block, x is the string solution, such that x_j with $1 \leq j \leq m$ is unfixed, we have:

$$x_j = \begin{cases} \sigma & \text{argmin}_{\sigma_k} \sum_{i=1}^{4} |s_j^i = \sigma_k| \quad \sigma_k \in \mathcal{P}_j \quad k = 1, \ldots, |\mathcal{P}_j| \quad \forall s^i \in \mathcal{P} \\ \sqcup & \text{otherwise} \end{cases} \tag{4.22}$$

Note that at beginning x was a partially filled candidate, at this stage, if there are not unfixed characters, then x is an optimal solution with $d_H(x, s) \leq d_{opt}\ \forall s \in \mathcal{S}$.

The $\mathcal{Q}$ block is a 4-CSP instance composing by the prefixes of $\mathcal{S}$ where each column position belonging to v_5, v_6, v_7 column types. Let w be a vector of 4-integers, $w_i = m$ with $i = 1, \ldots, 4$. Let $\Phi(.)$ be a function, its input parameters are 4-integers which are $d_H(x, s^i)\ s^i \in \mathcal{S}$ when the j-th column position is being processed, it returns a character to be fixed in the j-th column position, we have:

$$x_j = \Phi(w - \sum_{p=1}^{j} \Delta_{\mathcal{Q}_p}^{\sigma}) \tag{4.23}$$

$$\Delta_{\mathcal{Q}_p}^{\sigma} = \begin{cases} 0 & \text{if } s[p] = \sigma \quad \forall s \in \mathcal{Q},\ \sigma \in \mathcal{Q}_p \\ 1 & \text{otherwise} \end{cases} \tag{4.24}$$

Let $\mathcal{I}$ be the set of indices of the maximum Hamming distance.

$$\mathcal{I}_j = \{\ \text{argmax}_i\ (w_i - \sum_{p=1}^{j} \Delta_{\mathcal{Q}_p}^{\sigma}),\ i = 1, \ldots, 4\} \tag{4.25}$$

Let $\mathcal{J}$ be the set of indices of the minimum Hamming distance.

$$\mathcal{J}_j = \{\ \text{argmin}_i\ (w_i - \sum_{p=1}^{j} \Delta_{\mathcal{Q}_p}^{\sigma}),\ i = 1, \ldots, 4\} \tag{4.26}$$

Note that, based on above definitions we define the $\Phi(.)$ function as follows,

$$\Phi(w - \sum_{p=1}^{j} \Delta^{\sigma}_{\mathcal{Q}_p})_{\forall s \in \mathcal{Q}} = \begin{cases} s_j^k & \text{if } |\mathcal{I}_j| = \{1,4\} \qquad\qquad k \in \mathcal{I}_j \\ s_j^k & \text{if } |\mathcal{I}_j| = \{2,3\} \ \& \ s_j^k \neq s_j^l \quad k \in \mathcal{I}_j,\ l \in \mathcal{J}_j \\ s_j^r & \text{if } |\mathcal{I}_j| = \{2,3\} \ \& \ s_j^k = s_j^l \quad r \neq k,\ r \in \mathcal{I}_j \end{cases} \tag{4.27}$$

We use induction for the rest of the proof on the number of unfixed positions in x the optimal solution. Our inductive hypothesis is that, there are k fixed positions in x, $\{j_1, \ldots, j_k\}$, with $k < m$, that is, $d_H(x,s) \leq d \ \forall s \in \mathcal{S}$, if x_j is unfixed, then $x_j = \Phi(w - \sum_{p=1}^{j} \Delta^{\sigma}_{\mathcal{Q}_p})$ with $j \in \{1, \ldots, m\} \backslash \{j_1, \ldots, j_k\}$, and $d_H(x,s) \leq d-1$ or $d_H(x,s) \leq d \ \forall s \in \mathcal{S}$.

Base case : if there are not unfixed positions in x, then MSA ends being x the optimal solution, with $d_H(x,s) \leq d_{opt} \ \forall s \in \mathcal{S}$.

Inductive Step: there are $k+1$ fixed positions in x, $\{j_1, \ldots, j_{k+1}\}$, by the inductive hypothesis we get, $d_H(x,s) \leq d-1$ or $d_H(x,s) \leq d \ \forall s \in \mathcal{S}$, consider that x_j is unfixed with $j \in \{1, \ldots, m\} \backslash \{j_1, \ldots, j_{k+1}\}$, then $x_j = \Phi(w - \sum_{p=1}^{j} \Delta^{\sigma}_{\mathcal{Q}_p})$, MSA minimizes the maximum Hamming distance in 1 or 0 units, so we have $d_H(x,s) \leq d-2$ or $d_H(x,s) \leq d-1 \ \forall s \in \mathcal{S}$, the inductive hypothesis is maintained. □

Running Time. The for-loop of lines 2-6 iterates in m steps the input instance, the inner for-loop iterates for each column type $v_0 \ldots, v_4$ in constant time, since the conditional executes when one of them is true, thus it requires m iterations. In the line 7, the **twoFirstGreatestNumber** function sorts in non-increasing order by the 4-integer values, it returns the first and second numbers and its index associated, thus we get $4\log(4)$ which is constant c_1. The line 8 makes arithmetic operations, we have a constant time c_2. The for-loop of lines 9-10 iterates in m steps the integer-vector u, for each column position the **majority** function returns the most repeated character over the column position $\mathcal{S}[.][j]$, it requires 4 steps, since there are only four strings, so we get $4m$. The for-loop of lines 11-14 iterates in m steps the integer-vector u, note that in the worst case the variable *count* is $m/2$, the **minority** function returns the least repeated character over the column position $\mathcal{S}[.][j]$, it requires 4 steps since the input instance consist only in 4 strings, so it requires $4(m/2) = 2m$. In line 15, d_H function calculates the Hamming distance between x and each input strings $s \in \mathcal{S}$, so we get $4m$. Note that, if all the positions in x are fixed, then MSA stops. To sum up, the total execution time at this point is $\max\{m, c_1, c_2, 4m, 2m, 4m\} = 4m$.

The for-loop of lines 16-28 iterates in m steps when there is $x[j]$ unfixed, so it requires m iterations. The lines 17 and 18 calculate the maximum and minimum indices over four integers which are the Hamming distances between x and s for each $s \in \mathcal{S}$, in 4

iterations, so it requires $4m$. In the line 19, the **numberOfMaximum** function calculates the number of times that $d[i]$ appears in d, it requires 4 iterations, so we have $4m$. The switch conditional of lines 20-27 evaluates according to the variable **iNumberMax**, The line 21 is executed when the variable **iNumberMax** is 1 or 4, then a character is fixed, so it takes m iterations. The switch-conditional branch of lines 23-27 is evaluated when the variable **iNumberMax** is 2 or 3, then the if-conditional block evaluates the difference of characters between the minimum and maximum indices on the j column position, the if-then conditional requires 1 iteration, and the if-else conditional have 4 steps, so we have $4m$. In the line 28, the **updateHammingDistance** function updates $d[i]$ with $i = 1 \ldots, 4$ subtracting one unit if the fixed character $x[j]$ is presented in $\mathcal{S}[i][j]$ the i-th row, zero in otherwise, it requires 4 steps in evaluating the integer vector d, so we have $4m$. Note that, at this step all the positions in x were fixed, the algorithm stops, the total execution time is $4m$. Therefore, the running time of MSA is proven to be $O(8m)$. □

Example 2. *Let $\mathcal{S}$ be a CSP instance with 4-strings and each string of length 5, that is, $\mathcal{S} = \{00110, 00101, 11110, 11011\}$, applying MSA, we get:*

$$\mathcal{S} = \begin{cases} 00110 \\ 00101 \\ 11110 \\ 11011 \end{cases}$$

v_5	v_5	v_4	v_2	v_7
0	0	1	1	0
0	0	1	0	1
1	1	1	1	0
1	1	0	1	1
$*$	$*$	1	1	$*$

v_5	v_5	v_4	v_2	v_7
0	0	1	1	0
0	0	1	0	1
1	1	1	1	0
1	1	0	1	1
1	0	1	1	1

*The column-position types are $\{v_2, v_4, v_5, v_7\}$, MSA makes pre-processing over the tuples $\{v_2, v_4\}$, thus we obtain the partially filled candidate $x = **01*$, where the character * indicates that it has not been decided yet, for the rest unfixed column-positions, MSA fixes them according to different cases, finally, the optimal solution is $x = 10111$. For more details see Appendix A.1, it is a step-by-step for this example using the MSA approach.*

4.2.3 Remarks

In the literature there is an efficient linear-time algorithm for four binary strings proposed by [Boucher et al., 2009], it sometimes did not get the optimal solution, on the other hand, our proposal, an efficient algorithm, always got the optimal solution. However, we are interested in the CSP (general case), thus, our results will be extended to column-position character, for each of them we calculate the Hamming distance, hence a greedy-choice will minimize the maximum Hamming distance.

4.3 A recursive exact algorithm for the general case

In this section, we propose a greedy heuristic algorithm for CSP that allows one to get a good approximation for an optimal solution. Also we pose an exact method, a recursive algorithm for the general case.

Notation. Throughout this section, we will be considering our input instance as a matrix. If the set $\mathcal{S}$ has k strings, $s^1, \ldots, s^k$, each of length m, we view $\mathcal{S}$ as a $k \times m$ matrix. We can thus refer to columns and rows, $\mathcal{S}[i][j]$ refers to string i, position j (i.e. the indexing is row-first). The distance of a string s' from $\mathcal{S}$ is $\max_{s \in \mathcal{S}} d_H(s', s)$. Given a string s we denote with $s[j]$ the character in the jth position of s. A set of column positions $\mathcal{I}$ is denoted as $\mathcal{I} = \{p_1, \ldots, p_m\} \subseteq \{1, \ldots, m\}$ where $p \in \mathcal{I}$ is a specific column position. For a given two strings s^1 and s^2, define $\mathcal{P}_{s^1,s^2}$ as a set of indices where $s^1[p] = s^2[p]$ for $p \in \mathcal{I}$, that is, $\mathcal{P}_{s^1,s^2} = \{p \in \mathcal{I} |\ s^1[p] = s^2[p]\} \subseteq \mathcal{I}$. Let $\mathcal{Q}_{s^1,s^2}$ be a set of indices complement of $\mathcal{P}$, that is, $\mathcal{Q}_{s^1,s^2} = \mathcal{I} \setminus \mathcal{P}_{s^1,s^2}$. Note that $|\mathcal{Q}_{s^1,s^2}| = d_H(s^1, s^2)$. Let X be a set of optimal solutions obtained from $\mathcal{S}$, that is $X = \{x |\ x$ is an optimal solution of $\mathcal{S} \setminus s^i$ with $s^i \in \mathcal{S},\ i = 1, \ldots, k\}$. Let $\mathcal{P}_X$ be a set of indices where all strings in X in those positions agree, that is, $\mathcal{P}_X = \{p \in \mathcal{I} |\ x^1[p] = x^i[p];\ \ i = 2, \ldots, k\} \subseteq \mathcal{I}$. Let $\mathcal{Q}_X$ be a set of complement indices, that is, $\mathcal{Q}_X = \mathcal{I} \setminus \mathcal{P}_X$. A sequence is a string of characters over an alphabet Γ (σ is a typical symbol in Γ). A substring of a sequence s is a subsequence of successive characters within s (not necessarily contiguous). Given a string s we denote with $s[\mathcal{P}_X]$ a substring of s restricted to a set of indices $\mathcal{P}_X$. A set of substrings $\mathcal{S}[\mathcal{P}_X]$ is called affixes of $\mathcal{S}$ restricted to a set of indices $\mathcal{P}_X$, that is, $\mathcal{S}[\mathcal{P}_X] = \{s[p]$ for $s \in \mathcal{S},\ p \in \mathcal{P}_X\}$. Let U_j be a set of characters used in the j-th position of the strings in $\mathcal{S}[j] = \{s^1[j], \ldots, s^k[j]\}\ j \in \mathcal{I}$.

The recursive algorithm (CSP-R) is formally established by its proof of correctness and its running time is described, at the same time GREEDY is posed by its theoretical analysis and its correspondingly execution time also is presented.

4.3.1 GREEDY heuristic algorithm

This algorithm is a polynomial heuristic for solving the general CSP, it is a greedy algorithm (GREEDY) because the choice at each position is global and guided, we choose a character according to a global evaluation of the Hamming distance; moreover, our method identifies a step on the solution space, since the choice of the character at position j affects the choice of the character at position $j + 1$.

Definition 5 (Partially filled candidate[Amir et al., 2016]). *We say that x^* is a partially*

filled candidate over Γ *if for every symbol* σ *in* x^*, $\sigma \in \Gamma \cup \{␣\}$, *where* $␣ \notin \Gamma$ *means that the value of this symbol has not been decided yet.*

Definition 6 (Consensus basis[Amir et al., 2016])**.** *We say that a partially filled candidate* x^* *is a consensus basis if there exists a consensus string* x^c *such that every symbol of* x^* *is either equal to the corresponding symbol of* x^c *or is* $␣$.

Description of the greedy algorithm

GREEDY is a simple but effective method that can be used to solve the CSP. Firstly, in this algorithm, the **construction phase** stores in M, the complement of the Hamming distance between $\sigma \in \mathcal{S}[.][j]$ and $\mathcal{S}[.][j]$. Also it stores, in M', the Hamming distance between the partially filled candidate x^* and s, $\forall s \in \mathcal{S}$. Then an **improve-solution** traverses all column-characters in M'; any specific column-character is selected such that it can be the best column character. The **greedy-choice**, at the same time, minimizes the maximum Hamming distance, chooses the column-character that has the minimum number of maximum Hamming distances, searches in three depth levels, and returns the maximum number of characters presented in the best column character. After that, all column characters associated to that column-position is removed in M', thus, GREEDY iterates over M' by the same improve-solution. The **stop criterion** is m iterations because at each iteration one column position is removed in M', where m is the size of the input-strings.

Construction phase

The construction phase creates two similar structure matrices M and M', each of them is an integer matrix of k rows and $|\Gamma| \times m$ columns. Their columns (j, σ), $\sigma \in s_j$, $\forall s \in \mathcal{S}$ refer to σ character in the j-th column in the given set of strings. Define $\mathcal{Q}$ as the set of indices of $\mathcal{S}$ where there are more than 1 different symbol at any column position. Let U_j be a set of characters used in the j-th position of the strings in $\mathcal{S}$. Define $\mathcal{Q} \times U_j$ as the Cartesian product of two sets $\mathcal{Q}$ and U_j, that is, $\mathcal{Q} \times U_j = \{(j, \sigma) : \ j \in \mathcal{Q} \text{ and } \sigma \in U_j\}$. For any element $(j, \sigma) \in \mathcal{Q} \times U_j$, $j\sigma$ refers to any column-character. In M, each element consists of only 1 or 0. In each iteration of GREEDY, for each $j\sigma$ column-character, the σ character is fixed in the j-th position of the current solution, thus in M', each column-character $j\sigma$ consists of Hamming distances between the current solution and each input-strings, that is, if $x[j] = \sigma$ then $M'[i, j\sigma] = d_H(x, s^i)$, $s^i \in \mathcal{S}$, $i = 1, \ldots, k$; $\forall \sigma \in U_j$. More formally

we have

$$\Delta^i_{j\sigma} = \begin{cases} 1 & \text{if } s^i[j] = \sigma \qquad s^i \in \mathcal{S} \qquad \sigma \in \Gamma \\ 0 & \text{otherwise} \end{cases} \tag{4.28}$$

Let w be the vector of Hamming distances, that is, $w_i = d_H(x, s^i)\ \ s^i \in \mathcal{S}\ \ i = 1, \ldots, k$. In M', each element is a positive integer value defined as follows:

$$M'[i, j\sigma] = w_i - M[i, j\sigma] \quad i = 1, \ldots, k \quad \forall j\sigma \in \mathcal{Q} \times U_j \tag{4.29}$$

The function in (4.28) calculates the complement of the exclusive OR (XOR) between each column-character and the set of input strings.

Improve-solution

The improve-solution, at each iteration, traverses the column-character in M', minimizes the maximum Hamming distance, chooses the column-character that has the minimum number of maximum Hamming distances, finally, searches depth in three levels and returns the maximum number of characters presented in any column of the input-instance.

Theorem 3. *Let $\mathcal{S} = \{s^1, \ldots, s^k\}$ be an instance for CSP each string of length m. There is x^c a good approximation for an optimal solution obtained from GREEDY, such that, $d_H(x^c, s^i) \leq d,\ i = 1, \ldots, k;\ s^i \in \mathcal{S}$.*

Proof. The proof is made by induction method. Define $\mathcal{Q}$ as the set of indices of $\mathcal{S}$ where there are more than 1 different symbol at any column position. Let $\mathcal{Q}_2, \mathcal{Q}_1$ be subsets of $\mathcal{Q}$, that is, $\mathcal{Q}_2 \subseteq \mathcal{Q}_1 \subseteq \mathcal{Q}$. Define $\mathcal{Q} \times U_j$ as the Cartesian product of two sets $\mathcal{Q}$ and U_j, that is, $\mathcal{Q} \times U_j = \{(j, \sigma) :\ j \in \mathcal{Q} \text{ and } \sigma \in U_j\}$. For any element $(j, \sigma) \in \mathcal{Q} \times U_j$, $j\sigma$ refers to any column-character. Define $D_{\mathcal{Q}_1 \times U_j}$ as the minimum of maximum Hamming distance. Let $D_{\mathcal{Q}_2 \times U_j}$ be the minimum of number of times that $D_{\mathcal{Q}_1 \times U_j}$ appears in any column-character.

$$D_{\mathcal{Q} \times U_j} = \max_i(w_i - \Delta^i_{j\sigma}) \quad \forall (j, \sigma) \in \mathcal{Q} \times U_j \quad i = 1, \ldots, k \tag{4.30}$$

$$D_{\mathcal{Q}_1 \times U_j} = \operatorname{argmin}_{j\sigma} D_{\mathcal{Q} \times U_j} \quad \forall (j, \sigma) \in \mathcal{Q} \times U_j \quad i = 1, \ldots, k \tag{4.31}$$

$$D_{\mathcal{Q}_2 \times U_j} = \operatorname{argmin}_{j\sigma} \sum_{i=1}^{k} |w_i - \Delta^i_{j\sigma} = D_{\mathcal{Q}_1 \times U_j}| \quad \forall (j, \sigma) \in \mathcal{Q}_1 \times U_j \tag{4.32}$$

$$\min(\mathcal{S}[\mathcal{Q}]) = \begin{cases} [\ \operatorname{argmax}_{j\sigma} \sum_{i=1}^{k} \Delta^i_{j\sigma}]^3 & \forall (j, \sigma) \in \mathcal{Q}_2 \times U_j \quad \text{If } |\mathcal{Q}| > 3 \\ \operatorname{argmax}_{j\sigma} \sum_{i=1}^{k} \Delta^i_{j\sigma} & \text{Otherwise} \end{cases} \tag{4.33}$$

The function in (4.30) determines the maximum Hamming distance for each column-character in $\mathcal{Q} \times U_j$. The function in (4.31) minimizes the maximum Hamming distance

between any column-character and the set of strings $\mathcal{S}[\mathcal{Q}]$. The function in (4.32) minimizes the number of maximum Hamming distances presented in $\mathcal{Q}_1 \times U_j$. The function in (4.33) reaches the best column-position-character in the third level when the number of column-positions presented in $\mathcal{Q}$ is greater than three, otherwise, it maximizes the number of characters presented in $\mathcal{Q}_2 \times U_j$. Finally the function in (4.33) is GREEDY applied to any column-position j ($j \in \mathcal{Q}$).

Base case: We start when the length of the strings is one, that is, $|s^i| = 1$, for $i = 1, \ldots, k$. Let x be a string of size 1 with one free position. Let $M[c_1]$ be a character matrix obtained from $\mathcal{S}$. Note that if there are two different symbols at this column position, we use the Function in (4.31). If there are three different symbols, we use the Function in (4.33), and if all characters are different, we use the Function in (4.32). Thus, we use the algorithm to fill in the blank to get a result $f(M[c_1]) = x^c$, such that $d_H(x^c, s) \leq d$ for all $s \in \mathcal{S}$.

Suppose there is a string x^c of size l, and the input strings $\mathcal{S} = \{s^1, \ldots, s^k\}$ of length l, such that, $d_H(x^c, s) \leq d$, $\forall s \in \mathcal{S}$.

Inductive Step: Now we start when the length of the strings is $l + 1$, that is, $|s^i| = l + 1$, for $i = 1, \ldots, k$. Let $M[c_1 \ldots c_{l+1}]$ be a character matrix obtained from the input instance, with $l + 1$ column positions. We apply the algorithm on that instance, that is, $f(M[c_1 \ldots c_{l+1}]) = y$ to get c_j ($j \in \{1, \ldots, l + 1\}$) a fixed column position at j-th position in y. Note that y is a string of size $l+1$ with one fixed column position at j-th position. By the inductive hypothesis there is a good approximation for a solution x of size l applying over $f(M[c_1 \ldots c_{l+1} \setminus c_j])$. Note that, the string x is a partially filled candidate, it has one blank (free position) at j-th position, $d_H(x, s) \leq d$ for all $s \in \mathcal{S}$, so we fix it with the (y_j) character, as a result, the maximum Hamming distance value may keep the same or reduce in one unit, thus, x^c is a good approximation for a solution with size $l + 1$, such that $d_H(x^c, s) \leq d$, $\forall s \in \mathcal{S}$; the inductive hypothesis is maintained. □

The running time

In the Algorithm 8, the **minimizeMaximumValue** function has as input parameters: ($\mathcal{S}$) a set of strings, (k) the number of strings, (m) the size of strings, and (x) a consensus basis. Line 2 calculates the Hamming distances between x and the set of strings, it requires $O(km)$. Line 3 initializes the 3-dimensional integer matrices. The for-loop of lines 4-9 stores the Hamming distance for each character presented in any column position of the set of strings, it takes $O(m|\Gamma|k)$. The while-loop of lines 10-17 evaluates for each unfixed column

position the matrix $\mathcal{T}$ (3-dimensional integer matrix with dimensions m, $|\Gamma|$, and k), so it takes $O(|\Gamma|km^2)$. Consequently, the total running time is $O(|\Gamma|km^2)$.

Example 3. *Let $\mathcal{S}$ be a CSP instance with 3-strings and each string of length 4, that is, $\mathcal{S} = \{GTCC, AGAG, CGAG\}$, applying greedy, we get:*

$$\mathcal{S} = \begin{cases} GTCC \\ AGAG \\ CGAG \end{cases}$$

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

1			2		3		4	
A	C	G	$\boldsymbol{G}$	T	$\boldsymbol{A}$	C	C	$\boldsymbol{G}$
4	4	3	***4***	3	***4***	3	3	***4***
3	4	4	***3***	4	***3***	4	4	***3***
4	3	4	***3***	4	***3***	4	4	***3***
4	4	4	***4***	4	***4***	4	4	***4***
2	2	2	***1***	2	***1***	2	2	***1***

1			2		3		4	
A	C	$\boldsymbol{G}$	G	T	A	$\boldsymbol{C}$	$\boldsymbol{C}$	G
4	4	***3***	0	0	4	***3***	***3***	4
2	3	***3***	0	0	2	***3***	***3***	2
3	2	***3***	0	0	2	***3***	***3***	2
4	4	***3***	0	0	4	***3***	***3***	4
1	1	***3***	0	0	1	***3***	***3***	1

1			2		3		4	
A	C	G	G	T	$\boldsymbol{A}$	C	C	$\boldsymbol{G}$
0	0	0	0	0	***3***	2	2	***3***
0	0	0	0	0	***2***	3	3	***2***
0	0	0	0	0	***2***	3	3	***2***
0	0	0	0	0	***3***	3	3	***3***
0	0	0	0	0	***1***	2	2	***1***

1			2		3		4	
A	C	G	G	T	A	C	$\boldsymbol{C}$	G
0	0	0	0	0	0	0	***2***	3
0	0	0	0	0	0	0	***2***	1
0	0	0	0	0	0	0	***2***	1
0	0	0	0	0	0	0	***2***	3
0	0	0	0	0	0	0	***3***	1

GREEDY initializes the current solution $x = ****$ with Hamming distance $d = [4, 4, 4]$, also it creates two integer matrices H filled by 0-1 digits and T is the difference between d and H (iteration #1) the column-character $2G$ is fixed as the best greedy-choice, now $x = *G**$ $d = [4, 3, 3]$, T is updated (iteration #2) the column-character $1G$ is fixed, we get, $x = GG**$ $d = [3, 3, 3]$, T is updated (iteration #3) the column-character $3A$ is fixed, we have, $x = GGA*$ $d = [3, 2, 2]$, T is updated (iteration #4) the column-character $4C$ is fixed, we get, $x = GGAC$ with Hamming distance $d = [2, 2, 2]$. For more details see Appendix A.2, it is a step-by-step for this example using the GREEDY algorithm.

Algorithm 8: GREEDY pseudo-code.

Input: $\mathcal{S}, k, m, x$: CSP instance, number, length of strings, and consensus basis

Output: x is a good approximation for an optimal solution $\mid \max_i d_H(x, s^i) \leq d$

```
1  Function minimizeMaximumValue(S, k, m, x):
2      d[.] ← d_H(x, S[.]);
       /* H is 3-dimensional int matrix m|Γ|(k+1)                    */
       /* T is a 3-dimensional int matrix m|Γ|(k+2)                  */
3      H ← ∅, T ← ∅
       /* Construction phase                                         */
4      for j ← 1, ..., m such that x[j] is unfixed do
5          foreach σ ∈ S[.][j] do
6              for i ← 1, ..., k do
7                  if σ = S[i][j] then H[j][σ][i] ← 1 ;
8                  else H[j][σ][i] ← 0;
9                  H[j][σ][k+1] ← H[j][σ][k+1] + H[j][σ][i]
       /* Improve-solution                                           */
10     while #unfixed > 0 do
11         T ← updateTable(H, d[.], x, T)
12         (T', d', x') ← (T, d, x)
13         if #unfixed > 3 then (j, σ) ← depthSearch3Levels(H, T', d', x');
14         else (j, σ) ← chooseBestValue(H, T, x);
15         x[j] ← σ
16         d[.] ← updateHammingDistance(S[.][j], x[j], d[.])
17         #unfixed ← #unfixed −1
18     return x[.]
```

4.3.2 Recurrence relation

The CSP-R algorithm is a recursive exact method to solve the general CSP. Let $\mathcal{S} = \{s^1, \ldots, s^k\}$ be the input-instance with k strings of length m. Our method solves recursively sub-instances of $k-1$ strings, that is, $\mathcal{S} \setminus \{s^i\}$ $(i = 1, \ldots, k)$; it removes one string for each time from the input-instance. Let X be the set of optimal solutions obtained from $\mathcal{S} \setminus \{s^i\}$, hence X is a set of k strings of length m. Let $\mathcal{P}_{\mathcal{X}}$ be the set of column-positions where all matches (characters agree), we can set x to be a consensus basis with fixed characters in

Algorithm 9: GREEDY functions.

Function `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - **foreach** $\sigma \in \mathcal{S}[.][j]$ **do**
    - iMax $\leftarrow 0$
    - **for** $i \leftarrow 1, \ldots, k$ **do**
      - $\mathcal{T}[j][\sigma][i] \leftarrow d[i] - \mathcal{H}[j][\sigma][i]$
      - **if** $iMax < \mathcal{T}[j][\sigma][i]$ **then** iMax $\leftarrow \mathcal{T}[j][\sigma][i]$;
    - $\mathcal{T}[j][\sigma][k+1] \leftarrow$ iMax
    - iCount $\leftarrow 0$
    - **for** $i \leftarrow 1, \ldots, k$ **do**
      - **if** $\mathcal{T}[j][\sigma][i] = iMax$ **then** iCount $\leftarrow$ iCount $+1$;
    - $\mathcal{T}[j][\sigma][k+2] \leftarrow$ iCount
- **return** $\mathcal{T}$

Function `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d'[.], x')$

- $\mathcal{Q} \leftarrow$ `chooseBestValues`$(\mathcal{H}, \mathcal{T}', x')$
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x''[j], d''[.])$
    - $\mathcal{T}' \leftarrow$ `updateTable`$(\mathcal{H}, d''[.], x'', \mathcal{T}')$
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}'', x')$
- **return** `chooseBestValue`$(\mathcal{H}, \mathcal{T}'', x')$

Function `updateHammingDistance`$(\mathcal{S}[.][j], \sigma, d[.])$

- **for** $i \leftarrow 1, \ldots, k$ **do**
  - **if** $\mathcal{S}[i][j] = \sigma$ **then** $d[i] \leftarrow d[i] - 1$;
- **return** $d[.]$;

Function `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$

- iMin1 $\leftarrow$ `chooseMinimumValue` $(\mathcal{T}[.][.][k+1])$
- iMin2 $\leftarrow$ `chooseMinimumValue` $(\mathcal{T}[.][.][k+2]$, iMin1$)$
- iMax $\leftarrow$ `chooseMaximumValue` $(\mathcal{H}[.][.][k+1]$, iMin1, iMin2$)$
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - **foreach** $\sigma \in \mathcal{S}[.][j]$ **do**
    - **if** $\mathcal{T}[j][\sigma][k+1], \mathcal{T}[j][\sigma][k+2], \mathcal{H}[j][\sigma][k+1] = iMin1, iMin2, iMax$ **then**
      - **return** (j, σ)

$\mathcal{P}_{\mathcal{X}}$ positions. Finally, our method uses GREEDY on $\mathcal{S}[\mathcal{Q}_{\mathcal{X}}]$ to fix the free positions in x, thus x is an optimal solution for the input-instance, that is, $d_H(x,s) \leq d_{opt} \ \ \forall s \in \mathcal{S}$.

A recursive algorithm

Intuitively, we can construct a recursive algorithm for the CSP as follows:

(1) Divide the input instance $\mathcal{S}$ into k subproblems, that is, $\mathcal{S}^i = \mathcal{S} \setminus \{s^i\}$ $(i = 1, \ldots, k)$. The subproblem $\mathcal{S}^i$ is built taking out the ith string of $\mathcal{S}$.

(2) Solve each subproblem $\mathcal{S}^i$ by solving them recursively. If the number of strings is equal to two we get the base case.

(3) Combine the solutions to the subproblems, fix the column positions by their majority consensus value when all characters in them are equals, as a result, we get a consensus basis. Finally fill the free column positions by GREEDY.

The proof of correctness

Define $\mathcal{S}^i$ as subset of $\mathcal{S}$, that is, $\mathcal{S}^i = \mathcal{S} \setminus s^i$ $(i = 1, \ldots, k)$ such that $|\mathcal{S}^i| = |\mathcal{S}| - 1$. The main idea for the recursive algorithm is to solve recursively $\mathcal{S}^i$ $(i = 1, \ldots, k)$; after that for each subset $\mathcal{S}^i$ we get an optimal solution x^i. X is the set of these partial solutions. Based on them the algorithm makes a consensus basis (i.e. fix the trivial column position by its majority consensus value), finally it fixes the unfixed column positions by GREEDY. More formally we get the following recursion:

$$\Phi(\mathcal{S}, k) \in \{\text{TRUE, FALSE}\} \tag{4.34}$$

The output of the function $\Phi(.)$ is TRUE if there exists a string x such that $d_H(x,s) \leq d_{opt} \ \ \forall s \in \mathcal{S}$, an optimal solution, in agreement with [Dalpasso and Lancia, 2018].

$$\Phi(\mathcal{S}, k) = \begin{cases} min\ \mathcal{S}[\mathcal{I} \setminus \mathcal{P}_{s^1,s^2}] & s^1, s^2 \in \mathcal{S} \quad \text{if } k = 2 \\ min\ \mathcal{S}[\mathcal{I} \setminus \mathcal{P}_{\Phi(\mathcal{S}\setminus s^i, k-1)}] & i = 1, \ldots, k \quad \text{if } k > 2 \end{cases} \tag{4.35}$$

To reconstruct the solution, we can set $\Phi(\mathcal{S}, k)$ to be any x which makes true the $\Phi(.)$ in (4.35). The output of $\Phi(\mathcal{S} \setminus s^i, k-1),\ i = 1, \ldots, k$ is a set of partial solutions X with k strings of length m. Moreover the affixes $\mathcal{S}[\mathcal{P}_X]$ is achieved by a pre-processing method, and the affixes $\mathcal{S}[\mathcal{Q}_X]$ is reduced by GREEDY .

We first introduce one lemma to prove that the CSP-R algorithm can find an optimal solution value for the general case.

Lemma 5. *Let $\mathcal{S} = \{s^1, s^2\}$ be an instance for CSP where each string is of length m, there is x an optimal solution, if there exists, obtained from $x = min\ \mathcal{S}[\mathcal{I} \setminus \mathcal{P}_{s^1,s^2}]$, where $d_H(x, s^i) \leq d^2_{opt},\ i = 1, 2;\ s^i \in \mathcal{S}$.*

base case. The proof is made by direct method. Let $\mathcal{P}_{s^1,s^2}$ be a set of column positions, that is, $\mathcal{P}_{s^1,s^2} = \{p \in \mathcal{I} |\ s^1[p] = s^2[p]\} \subseteq \mathcal{I}$. Let $\mathcal{Q}_{s^1,s^2} = \mathcal{I} \setminus \mathcal{P}_{s^1,s^2}$ then we have $min\ \mathcal{S}[\mathcal{Q}_{s^1,s^2}]$. Let x' be a partially filled candidate, where

$$x'[j] = \begin{cases} \sigma & \text{if } j \in \mathcal{P}_{s^1,s^2} \quad \sigma \in \mathcal{S}[j] \\ \text{␣} & \text{if } j \in \mathcal{Q}_{s^1,s^2} \end{cases} \tag{4.36}$$

Note that in the substring $x[\mathcal{Q}_{s^1,s^2}]$ all its characters are unfixed, so we apply GREEDY on the affixes $\mathcal{S}[\mathcal{Q}_{s^1,s^2}]$ for each column position $r \in \mathcal{Q}_{s^1,s^2}$. By the Theorem 3, $x[r] = min\ \mathcal{S}[r]\ \ \forall r \in \mathcal{Q}_{s^1,s^2}$. So we get $d_H(x, s^i) \leq d^2_{opt},\ i = 1, 2;\ s^i \in \mathcal{S}$, so we have the optimal solution and its corresponding optimal value. Therefore, the lemma holds. □

The Lemma 5 can be deduce, let $\mathcal{S} = \{s^1, s^2\}$ be a CSP instance with 2-strings, as follows:

	v_0	v_1	v_2
s^1	α	β	α
s^2	α	α	β

Note that, independently of whether the character appears in the column-position j in the string solution x, it mismatches with minimum 1 character for v_1, v_2. The sum of these values is equal to the Hamming distance between x and $s^i \in \mathcal{S}$, dividing it by 2 we get the average Hamming distance. The last result is also equal to the lower bound proposed by [Liu et al., 2011], that is, $d_{opt} \geq d_H(s^1, s^2)2$.

Finally we summarize our results in the following Theorem.

Theorem 4. *Given a set of strings $\mathcal{S} = \{s^1, \cdots, s^k\}$, the recursive algorithm finds a string x (an optimal solution, if there exists), such that $d_H(x, s^i) \leq d_{opt},\ i = 1, \ldots, k;\ s^i \in \mathcal{S}$.*

Proof. The proof is made by an induction method

Base case

$P(2)$: There is an optimal solution x^2 for two strings s^1 and s^2 getting from $x^2 = min\ \mathcal{S}[\mathcal{I} \setminus \mathcal{P}_{s^1,s^2}]$ with $s^1, s^2 \in \mathcal{S}$. By the Lemma 5, we get an optimal solution with length m such that $d_H(x^2, s^i) \leq d^2_{opt},\ i = 1, 2;\ s^i \in \mathcal{S}$.

Assume that $P(k-1)$ is true. Let $\bar{\mathcal{S}} = \{s^1, \ldots, s^{k-1}\}$ be a set of $(k-1)$-strings, then there is an optimal solution x^{k-1}, taking from $x^{k-1} = min\ \bar{\mathcal{S}}[\mathcal{I} \setminus \mathcal{P}_{\Phi(\bar{\mathcal{S}} \setminus s^i, k-2)}],\ i = 1, \ldots, k-1;\ s^i \in \mathcal{S}$, where x^{k-1} is a string of length m such that $d_H(x^{k-1}, s^i) \leq d^{k-1}_{opt}$.

Inductive step

Consider $P(k)$. Let $\mathcal{S} = \{s^1, \ldots, s^k\}$ be a set of k-strings. Then $\mathcal{S}$ can be decomposed into k subsets $\mathcal{S}^i \subseteq \mathcal{S}$, where $\mathcal{S}^i = \mathcal{S} \setminus s^i,\ i = 1, \ldots, k$, then the optimal solution x^k will be made up of the set of k optimal solutions $\bar{x}^i,\ i = 1, \ldots, k$ getting from each subset $\mathcal{S}^i$(subproblem), with $|\mathcal{S}^i| = k-1$. Then, $X = \cup_{i=1}^{k} \bar{x}^i,\ \bar{x}^i = min\ \mathcal{S}^i[\mathcal{I} \setminus \mathcal{P}_{\Phi(\mathcal{S}^i \setminus s^i, k-2)}],\ i = 1, \ldots, k-1;\ s^i \in \mathcal{S}^i$ with $d_H(\bar{x}^i, s^i) \leq \bar{d}^i_{opt}$. Obtained by induction hypothesis. Thus, $\mathcal{P}_X = \{p \in \mathcal{I} |\ \bar{x}^1[p] = \bar{x}^i[p],\ i = 2, \ldots, k\}$, and $\mathcal{Q}_X = \mathcal{I} \setminus \mathcal{P}_X$; then we have $min\ \mathcal{S}[\mathcal{Q}_X]$. Note that, x^k is a consensus basis, where

$$x^k[j] = \begin{cases} \sigma & \text{if } j \in \mathcal{P}_X \qquad \sigma \in \Gamma \\ \_ & \text{if } j \in \mathcal{Q}_X \end{cases} \tag{4.37}$$

Note that, in the substring $x^k[\mathcal{Q}_X]$ all its characters are free because it has not been decided yet. So we apply GREEDY on the affixes $\mathcal{S}[\mathcal{Q}_X]$ for each column position $r \in \mathcal{Q}_X$. Thus by the Theorem 3, there is x^k, an optimal solution obtained from $x^k[r] = min\ \mathcal{S}[r]\ \ \forall r \in \mathcal{Q}_X$. So we get $d_H(x^k, s^i) \leq d^k_{opt},\ i = 1, \ldots, k;\ s^i \in \mathcal{S}$, so we have the optimal solution and its corresponding optimal value. Therefore the theorem holds.

□

The running time complexity

The recursive combinatorial algorithm calls itself in k times, in order to determine the running time. We propose the following complexity-time function: for k the number of strings, we have

$$t(k) = \begin{cases} |\Gamma| k m^2 & \text{if } k = 2 \\ k.t(k-1) + |\Gamma| k m^2 & \text{if } k > 2 \end{cases} \tag{4.38}$$

To compute $t(k)$, we expand out the recurrence (4.38), obtaining

$$t(2) = 2|\Gamma| m^2 \tag{4.39}$$

$$t(3) = 3(2|\Gamma| m^2) + 3|\Gamma| m^2 \tag{4.40}$$

$$\ldots$$

$$t(k) = k!|\Gamma| m^2 + k!/2|\Gamma| m^2 + \cdots + k!/(k-1)!|\Gamma| m^2 \tag{4.41}$$

The recurrence in (4.41) is the function's value at the k-th iteration. Solving the sum, we get $t(k) = k!|\Gamma| m^2 \sum_{i=2}^{k} \frac{1}{(k-1)!}$. Because the sum of the reciprocals of factorials converge to e, we get $\sum_{i=2}^{k} \frac{1}{(i-1)!} \approx e$ which is a constant number. Finally, $k! \approx k^k$, and $|\Gamma| \approx k$ since a CSP instance with alphabet $|\Gamma| > k$ is isomorphic to a CSP instance with alphabet $|\Gamma'| = k$ [Gramm et al., 2001], thus $t(k) = \mathcal{O}(k^k.km^2) = \mathcal{O}(k^{k+1}.m^2)$.

Example 4. *Let $\mathcal{S}$ be a CSP instance with 3-strings and each string of length 4, that is, $\mathcal{S} = \{GTCC, AGAG, CGAG\}$, applying CSP-R (recursive exact method), we get:*

$$S'' = \begin{cases} \textit{CSP-R}(s^3, s^2) & x^1 & A\boldsymbol{G}A\boldsymbol{G} \\ \textit{CSP-R}(s^1, s^3) & x^2 & G\boldsymbol{G}C\boldsymbol{G} \\ \textit{CSP-R}(s^1, s^2) & x^3 & G\boldsymbol{G}C\boldsymbol{G} \end{cases}$$

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

1			2		3		4	
A	C	$\boldsymbol{G}$	G	T	A	$\boldsymbol{C}$	C	G
4	4	***3***	0	0	4	***3***	0	0
1	2	***2***	0	0	1	***2***	0	0
2	1	***2***	0	0	1	***2***	0	0
4	4	***3***	0	0	4	***3***	0	0
1	1	***1***	0	0	1	***1***	0	0

1			2		3		4	
A	C	G	G	T	A	$\boldsymbol{C}$	C	G
0	0	0	0	0	3	***2***	0	0
0	0	0	0	0	1	***2***	0	0
0	0	0	0	0	1	***2***	0	0
0	0	0	0	0	3	***2***	0	0
0	0	0	0	0	1	***3***	0	0

CSP-R makes 3-recursive calls with all string combinations, that is, $\{s^3, s^2\}$, $\{s^3, s^1\}$, $\{s^2, s^1\}$, each of them constitutes a partial solution, CSP-R creates a new instances with those partial solutions, after that it makes pre-processing over that instance, we get, $x = *G*G$ $d = [4, 2, 2]$, then it uses greedy to creates H a 0-1 matrix and $T = (d - H)$ (iteration #1) the column-character $1G$ is fixed as the best greedy-choice, now $x = GG * G$ $d = [3, 2, 2]$, T is updated (iteration #2) the column-character $4C$ is fixed, we have, $x = GGCG$ with Hamming distance $d = [2, 2, 2]$. For more details see Appendix A.3, it is a step-by-step for this example using the CSP-R exact recursive method.

4.3.3 Remarks

This chapter has provided the proposed methods to solve the Closest String Problem, firstly, for three strings with an arbitrary alphabet, secondly, four strings with a binary alphabet, finally, for the general case, a heuristic polynomial greedy algorithm and a recursive exact algorithm were summarized.

Algorithm 10: CSP-R, exact recursive pseudo-code.

Input: $\mathcal{S}, k, m$: CSP instance, number and length of strings

Output: x: optimal solution such that $\max_i d_H(x, s^i) \leq d_{opt}$

```
1  Function CSP-R(S, k, m, x):
2      if k = 2 then
           /* base case:  pre-processing method                    */
3          for j ← 1, ..., m do
4              if S[1][j] = S[2][j] then x[j] = S[1][j];
           /* greedy algorithm                                     */
5          x ← minimizeMaximumValue(S, k, m, x)
6      else
           /* pre-processing method                                */
7          for i ← 1, ..., k do
8              iCount ← 1
9              for j ← 1, ..., k do
10                 if i ≠ j then
                       /* S' is a set of k − 1 strings of size m   */
11                     copy(S'[iCount], S[j]);
12                     iCount ← iCount+1
               /* S'' is a set of k partial solutions of size m    */
13             S''[i] ← CSP-R(S', k − 1, m, S''[i])
14         for j ← 1, ..., m do
15             Q ← ∅
16             for i ← 1, ..., k do  Q ← Q ∪ S''[i][j] ;
17             if |Q| = 1 then  x[i] = Q[1] ;
           /* greedy algorithm                                     */
18         x ← minimizeMaximumValue(S, k, m, x)
19     return x[.]
```

Chapter 5

COMPUTATIONAL EXPERIMENTS

This chapter presents in comparison tables the results of computational experiments using the linear time algorithms for three and four strings, and also for the general case the heuristic greedy algorithm and the recursive exact algorithm.

5.1 Test environment

All the programs were developed in C++ and used the compiler Gnu C++ with optimizations. All tests were executed on a desktop computer with the following configuration: Dell processor Intel Core I7 with speed of 3.4 Ghz and 7.7 GB of RAM under Linux Ubuntu 18.04 LTS work on 64 bits. The solver IBM ILOG CPLEX version 12.7[CPLEX, 2014] was used to solve the IP-formulation.

The algorithm used for random-number generation is an implementation of the Well equidistributed long-period linear, a random number generator [Panneton et al., 2006]; this algorithm has been preferred because of its very long period of $2^{44497} - 1$.

Three classes of instances were tested. The first class is a binary alphabet (that is, $\{0, 1\}$). The second class is a DNA alphabet (that is, $\{A, C, G, T\}$), while the third class is a protein alphabet (that is, $\{A, B, \ldots, S, T\}$).

5.2 Results for linear-time algorithm with up to three strings

We now present the computational experiments carried out with the algorithms described in Sections 4.1.1 (An IP formulation for 3-CSP) and 4.1.3 (Efficient linear-time algorithm for three sequences).

MFA was executed over a set of 420 instances, with 14 instances for each of the alphabets. Ten instances were generated for each entry. In Tables 5.1 and 5.2 the results represent the average of the obtained values.

The headers on Tables 5.1 and 5.2 have the following meanings: the first column indicates the tested instance, indicating the parameter (m) size of string; the columns (2,4, and 20 Characters) indicates the (Min, Avg, Max) minimum, average, and maximum values of optimal solutions to 3-sequences for binary, DNA, and protein types. The columns labeled "IP-3" give the average running time (in milliseconds) for the IP-formulation with three strings. The columns labeled "MFA" give the average running time (in milliseconds) for the proposed algorithm.

In Table 5.3, we see that on average MFA yields solution values within 100% of the solution values obtained by IP-formulation for instances with alphabets of sizes two, four and twenty, respectively. The average CPU time for protein instances using MFA: 0.010 (milliseconds) which is less than that of IP-formulation: 1.047 (milliseconds).

Figure 5.1 presents in logarithmic scale the run-times obtained by IP-3 and MFA. It shows that the IP-3 has a monotonic increasing along the time for 2, 4 and 20 characters. Meanwhile, MFA has a less monotonic increasing. Because it traverses the column-positions once to get the optimal solution value.

5.3 Results for linear-time algorithm for four binary strings

We now present the computational experiments with three different methods, the first is an IP-formulation for 4-CSP with a binary alphabet presented in Section 4.2.1, the second is MSA described in Section 4.2.2, and the third was posed by Boucher *et al.* 2009 [Boucher et al., 2009].

Some computational experiments was done involving 140 instances, using the instance generator described in the literature [Meneses et al., 2004]. These instances consider bi-

Table 5.1: Summary of Results for 3-CSP with 2, and 4 characters.

Instance	2 Characters					4 Characters				
m	Min	Avg	Max	IP-3	MFA	Min	Avg	Max	IP-3	MFA
1000	250	258.9	264	0.009	0.001	429	441.1	451	0.065	0.003
2500	628	637.2	653	0.017	0.002	1080	1095.0	1113	0.048	0.002
5000	1252	1266.0	1285	0.031	0.003	2175	2190.3	2213	0.094	0.003
7500	1876	1900.7	1927	0.044	0.004	3252	3277.9	3308	0.143	0.004
10000	2456	2523.6	2575	0.059	0.005	4324	4382.9	4436	0.191	0.006
12500	3126	3150.8	3204	0.075	0.006	5429	5465.1	5509	0.252	0.007
15000	3740	3780.8	3818	0.089	0.007	6522	6562.9	6597	0.297	0.008
17500	4371	4406.3	4470	0.104	0.008	7615	7658.6	7719	0.347	0.009
20000	5006	5038.6	5086	0.119	0.009	8701	8749.1	8794	0.399	0.010
22500	5642	5660.8	5681	0.134	0.010	9820	9846.5	9893	0.450	0.012
25000	6239	6281.1	6329	0.149	0.011	10894	10943.9	10979	0.503	0.013
27500	6879	6917.9	6972	0.165	0.013	12007	12047.0	12100	0.551	0.014
30000	7504	7545.2	7586	0.180	0.013	13079	13143.7	13206	0.604	0.015
50000	12479	12526.1	12595	0.306	0.022	21803	21866.2	21928	1.129	0.024

Table 5.2: Summary of Results for 3-CSP with 20 characters.

Instance	20 Characters				
m	Min	Avg	Max	IP-3	MFA
1000	611	616.9	622	0.051	0.002
2500	1537	1545.1	1555	0.125	0.002
5000	3071	3088.1	3100	0.253	0.003
7500	4608	4629.1	4645	0.389	0.005
10000	6169	6183.1	6193	0.536	0.006
12500	7698	7716.5	7746	0.674	0.007
15000	9234	9261.5	9282	0.818	0.008
17500	10791	10808.9	10828	0.971	0.009
20000	12280	12347.3	12394	1.177	0.011
22500	13863	13888.0	13921	1.357	0.012
25000	15414	15440.8	15462	1.520	0.013
27500	16935	16981.4	17006	1.694	0.014
30000	18495	18535.9	18567	1.860	0.016
50000	30819	30859.3	30900	3.227	0.025

Table 5.3: Number of instances for which the optimal value is provided.

	Value	IP-3	MFA
Binary instances (140)	4421.00	0.106	0.008
DNA instances (140)	7690.73	0.362	0.009
Protein instances (140)	10850.14	1.047	0.010

nary alphabet that is, $\{0, 1\}$ with a set of four strings. The headers on Table 5.4 have the following meanings: The first block (Instance) is the tested instances indicating the

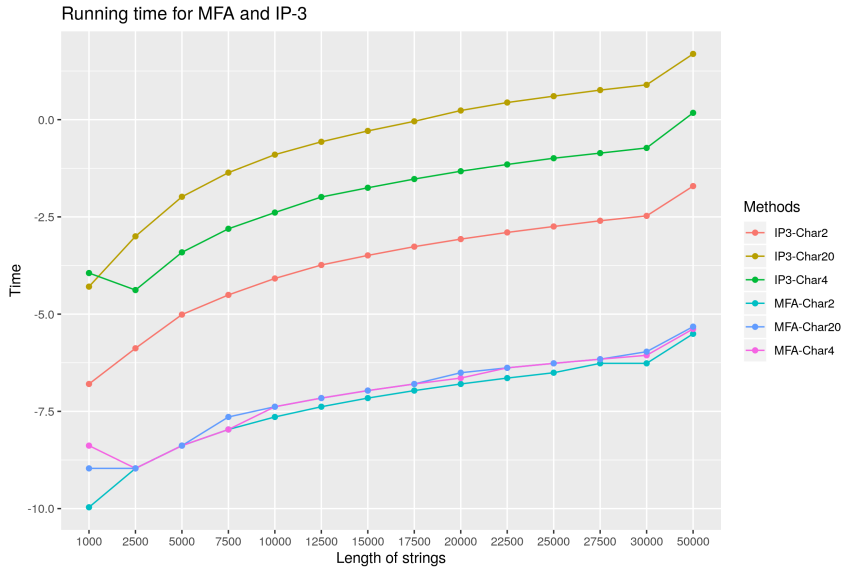


Figure 5.1: Run-time depending on different methods in logarithmic scale, Length of strings (x-axis), Time in logarithmic scale (y-axis), IP3-Char2 means IP-formulation for three strings applied to binary alphabet, MFA-Char4 means Minimization First algorithm applied to DNA alphabets. Source: the authors.

parameters (m) size of string. In the second block (IP) the columns (Min, Avg, and Max) present the minimum, average, and maximum optimal solution values obtained by the IP-formulation, the column (Time) indicates their execution time in milliseconds. The third block (MSA) the column (Time) presents its execution time in milliseconds generated by MSA. Finally, in the fourth block (Boucher) the columns (Min, Avg, and Max) show the minimum, average, and maximum values calculated by the Boucher's method, the column (Time) presents their execution time in milliseconds.

The Boucher's method posed a formal proof for the existence of a linear-time algorithm for the CSP with four binary strings, we implemented it, then we compare the optimal solution value in comparison tables, for example, the 4-CSP instances with $m = \{2500, \ldots, 30000\}$ have a gap of 1 unit compared to the IP-formulation, in these same instances our proposed algorithm got the optimal solution values. Consequently, our proposed algorithm demonstrates better results compared with Boucher's one.

In the Table 5.4, the Boucher's method posed a formal proof for the existence of a linear-time algorithm for CSP with four binary strings, we implemented it, then we compare the optimal solution value in comparison tables, for example, the 4-CSP instances with $m = \{2500, \ldots, 30000\}$ have a gap of 1 unit compared with the IP-formulation, in these same instances our proposed algorithm got the optimal solution value. Consequently, our

proposed algorithm demonstrates better results compared with Boucher's one.

In the Table 5.5, the algorithm embedded in the Boucher's method was easy to implement, so it not likely that we just have a buggy implementation of her algorithm, besides that for 80% of the tested instances got optimal solution value while for 20% always off by at most 1 unit.

Figure 5.2 shows the optimal solution values in logarithmic scale for the three algorithms. Table 5.4 presents the gap for optimal solution values between IP-4 and Boucher's methods, the worst gap value was for 5000 length of strings 0.0002% (error). On the other hand, MSA always got an exact solution value.

Finally, the IP-formulation presents less performance in its running time, also it leads to huge running time even for moderate number of variables. Thus, our proposed algorithm demonstrates to be a better algorithm in performance and effectiveness.

Table 5.4: Summary of results, for 4-sequences with 2 Characters.

Instance	IP-4				MSA	Boucher				
m	Min	Avg	Max	Time	Time	Min	Avg	Max	Time	Gap
1000	300	313.7	322	0.004	0.001	300	313.7	322	0.001	1.0000
2500	768	**778.3**	791	0.004	0.001	768	778.4	791	0.002	1.0001
5000	1547	**1565.7**	1579	0.004	0.002	1548	1565.8	1579	0.002	1.0002
7500	2324	**2343.2**	2364	0.004	0.002	2325	2343.7	2365	0.003	1.0001
10000	3108	**3130.0**	3154	0.004	0.003	3108	3130.2	3155	0.004	1.0001
12500	3892	**3914.2**	3944	0.004	0.003	3892	3914.3	3944	0.005	1.0000
15000	4637	**4670.3**	4695	0.004	0.004	4637	4670.5	4695	0.005	1.0000
17500	5453	**5471.9**	5496	0.005	0.004	5453	5472.1	5496	0.006	1.0000
20000	6226	**6249.6**	6292	0.005	0.005	6227	6250.2	6293	0.007	1.0001
22500	6978	**7031.2**	7052	0.005	0.005	6979	7031.6	7052	0.007	1.0001
25000	7773	**7808.3**	7832	0.005	0.005	7773	7808.4	7832	0.008	1.0000
27500	8552	**8597.8**	8636	0.005	0.006	8553	8597.9	8636	0.009	1.0000
30000	9313	**9363.4**	9390	0.006	0.006	9314	9363.5	9390	0.009	1.0000
50000	15582	15635.0	15698	0.008	0.010	15582	15635.0	15698	0.015	1.0000

Table 5.5: Number of instances for which the optimal value is provided.

Instances	IP-4	MSA	Boucher
Minimum (140)	5460.929	5460.929	5461.357
Average (140)	5490.900	5490.900	5491.093
Maximum (140)	5517.500	5517.500	5517.714
Time (140)	0.005 (140)	0.004 (140)	0.006 (113)

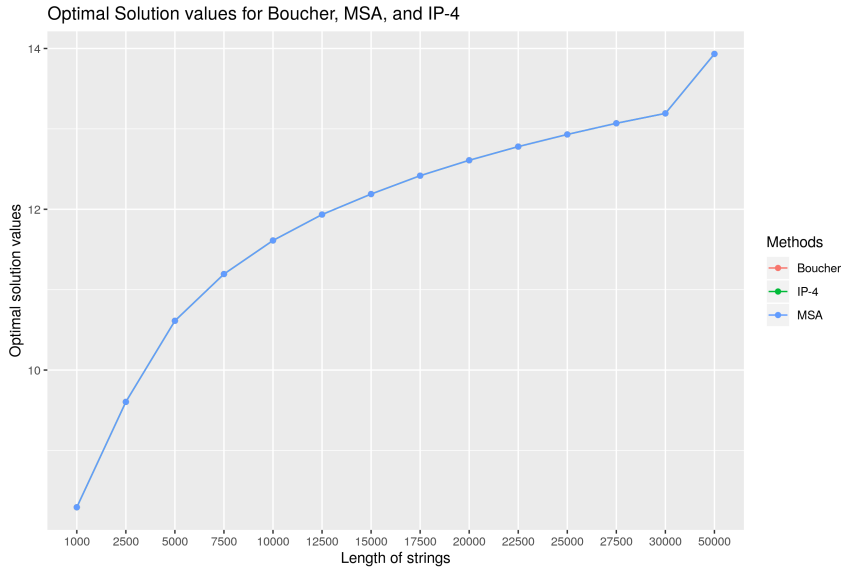


Figure 5.2: Optimal solution values depending on different methods in logarithmic scale, Length of strings (x-axis), Optimal Solution values in logarithmic scale (y-axis), Boucher means Boucher's method, IP-4 means IP-formulation for four binary strings, MSA means Minimization Second algorithm applied to binary alphabet. Source: the authors.

5.4 Results for recursive exact algorithm for the general case

We now present the computational experiments with three different methods. The first is an IP-formulation presented in Section 3.2. The second is the recursive algorithm (CSP-R) posed in Section 4.3.2. The third is GREEDY described in Section 4.3.1.

The computational experiments involving 900 instances with 30 instances for each of the alphabets. Ten instances were generated for each entry. The headers on Tables 5.6-5.8 have the following meanings: The first block (Instance) is the tested instance indicating the parameters: (k) number of strings, (m) size of string. The columns (Min, Avg, Max) present the minimum, average and maximum optimal solution values. The column (Time) indicates their execution time in milliseconds. The column (Gap) gives the relative gap between the heuristic solution and the IP solution, calculated as h/i, where h is the average heuristic value and i is the average IP solution value. The second block (IP) presents the optimal solution values generated by the IP-formulation. The third block (CSP-R) shows their execution time calculated by the recursive algorithm (CSP-R). Finally, in the fourth block (GREEDY) there is a good approximation for optimal solution values generated by GREEDY.

On Tables 5.6-5.8, the CSP-R algorithm got optimal solution values for 100% of the instances comparing to IP-formulation. Meanwhile, the CSP-R algorithm got a better performance for small instances, that is, in less than 6-strings its execution time got less than a one second. The boxplots in logarithmic scale for the three methods one for the lenght of string (Figure 5.3) and the other for the number of strings (Figure 5.4) show that the CSP-R has the worst running time.

On Tables 5.6-5.8, GREEDY got optimal solution values for 85% of the instances while for 15% (see Tables 5.6 and 5.7, the Greedy:average feasible solution values in bold) they were always off by at most 1 unit comparing to IP-formulation. Its CPU time got reasonable time, less than 38 milliseconds. Besides that, its running time increases slowly for large instances for the two, four, and twenty alphabets.

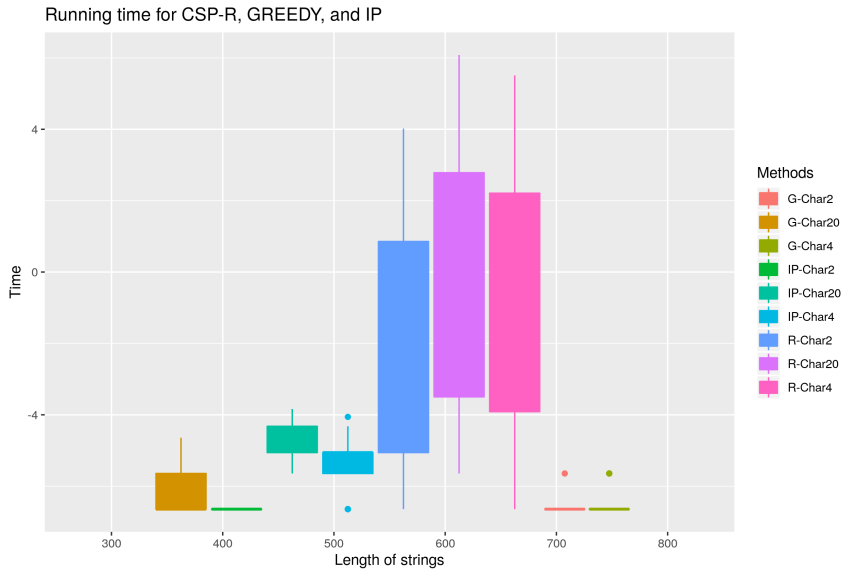


Figure 5.3: Boxplot in logarithmic scale of the running times for GREEDY and exact methods: The length of the strings (x-axis) and Time in logarithmic scale (y-axis), legend: G-Char2 means GREEDY method applied for binary characters, IP-Char4 means Integer Programming formulation applied for DNA alphabet, R-Char20 means exact recursive approach applied for protein alphabet. Source: the authors.

5.5 Results for GREEDY algorithm for the general case

Optimal solution values of IP against GREEDY algorithm, for the McClure Instances, over the Alphabet with 20 Characters. The experiments show that the greedy algorithm was

Table 5.6: Summary of Results for the Alphabet with Two Characters.

Instance		IP				CSP-R	GREEDY				
k	m	Min	Avg	Max	Time	Time	Min	Avg	Max	Time	Gap
3	300	73	79.9	86	0.01	0.00	73	79.9	86	0.00	1.000
	400	104	106.6	110	0.01	0.01	104	106.6	110	0.00	1.000
	500	125	128.9	136	0.01	0.01	125	128.9	136	0.00	1.000
	600	145	154.9	162	0.01	0.01	145	154.9	162	0.01	1.000
	700	172	182.7	190	0.01	0.02	172	182.7	190	0.01	1.000
	800	197	206.6	213	0.01	0.02	197	206.6	213	0.01	1.000
4	300	89	94.1	100	0.01	0.01	89	94.1	100	0.00	1.000
	400	118	124.1	130	0.01	0.02	118	124.1	130	0.00	1.000
	500	151	158.0	167	0.01	0.03	151	158.0	167	0.01	1.000
	600	177	184.6	191	0.01	0.04	177	184.6	191	0.01	1.000
	700	215	220.0	224	0.01	0.06	215	220.0	224	0.01	1.000
	800	239	248.0	259	0.01	0.08	239	248.0	259	0.01	1.000
5	300	94	97.6	100	0.01	0.06	94	97.6	100	0.00	1.000
	400	126	128.8	133	0.01	0.10	126	128.8	133	0.00	1.000
	500	155	161.1	169	0.01	0.15	155	161.1	169	0.01	1.000
	600	184	188.9	195	0.01	0.22	184	188.9	195	0.01	1.000
	700	218	224.8	230	0.01	0.30	218	224.8	230	0.01	1.000
	800	246	255.1	263	0.01	0.39	246	255.1	263	0.01	1.000
6	300	100	103.9	107	0.01	0.32	100	**104.0**	108	0.00	1.001
	400	135	137.6	141	0.01	0.58	135	137.6	141	0.00	1.000
	500	169	172.1	175	0.01	0.91	169	172.1	175	0.01	1.000
	600	203	208.1	211	0.01	1.33	203	208.1	211	0.01	1.000
	700	238	243.3	250	0.01	1.81	238	243.3	250	0.01	1.000
	800	269	275.9	284	0.01	2.33	269	**276.0**	284	0.01	1.000
7	300	104	106.7	110	0.01	2.24	105	**106.9**	110	0.00	1.002
	400	138	142.0	147	0.01	4.10	138	**142.2**	147	0.01	1.001
	500	171	174.6	178	0.01	6.29	171	**174.7**	178	0.01	1.001
	600	205	209.8	217	0.01	9.22	205	**210.2**	217	0.01	1.002
	700	241	244.4	249	0.01	12.64	242	**244.8**	250	0.01	1.002
	800	275	279.6	285	0.01	16.21	275	**279.7**	285	0.02	1.000

Table 5.7: Summary of Results for the Alphabet with Four Characters.

Instance		IP				CSP-R	GREEDY				
k	m	Min	Avg	Max	Time	Time	Min	Avg	Max	Time	Gap
3	300	124	130.3	134	0.01	0.01	124	130.3	134	0.00	1.000
	400	169	174.9	181	0.01	0.02	169	174.9	181	0.00	1.000
	500	213	219.0	228	0.01	0.02	213	219.0	228	0.01	1.000
	600	254	263.4	273	0.01	0.03	254	263.4	273	0.01	1.000
	700	292	304.6	315	0.02	0.04	292	304.6	315	0.01	1.000
	800	341	351.1	361	0.02	0.05	341	351.1	361	0.01	1.000
4	300	137	140.0	143	0.01	0.03	137	**140.2**	143	0.00	1.001
	400	183	186.9	191	0.02	0.06	183	186.9	191	0.00	1.000
	500	231	235.3	241	0.02	0.09	231	235.3	241	0.01	1.000
	600	275	280.7	285	0.02	0.12	275	280.7	285	0.01	1.000
	700	323	328.2	336	0.03	0.17	323	328.2	336	0.01	1.000
	800	367	372.5	378	0.03	0.21	367	372.5	378	0.01	1.000
5	300	149	152.3	157	0.02	0.16	150	152.5	157	0.00	1.001
	400	199	202.9	206	0.02	0.27	199	**203.4**	207	0.01	1.002
	500	250	252.9	256	0.03	0.43	250	**253.3**	256	0.01	1.002
	600	296	300.7	305	0.03	0.61	297	**300.9**	305	0.01	1.001
	700	353	355.3	360	0.03	0.84	354	**355.6**	360	0.01	1.001
	800	397	403.2	409	0.04	1.08	397	**403.3**	409	0.02	1.000
6	300	156	158.9	163	0.02	0.91	157	**159.3**	163	0.00	1.003
	400	207	212.2	217	0.03	1.63	208	**212.4**	218	0.01	1.001
	500	262	266.2	270	0.03	2.56	262	**266.3**	270	0.01	1.000
	600	314	317.8	320	0.04	3.66	314	**318.3**	321	0.01	1.002
	700	368	372.7	376	0.05	5.00	368	**372.8**	376	0.02	1.000
	800	421	424.8	427	0.05	6.51	421	**424.9**	428	0.02	1.000
7	300	163	164.5	166	0.03	6.45	164	**164.9**	166	0.00	1.002
	400	216	219.6	222	0.03	11.68	216	219.6	222	0.01	1.000
	500	268	272.6	278	0.03	17.90	268	**272.7**	278	0.01	1.000
	600	322	327.1	334	0.05	25.59	323	**327.4**	334	0.01	1.001
	700	375	382.8	388	0.06	34.93	376	**383.0**	388	0.02	1.001
	800	434	436.7	442	0.05	45.58	434	**436.8**	443	0.02	1.000

Table 5.8: Summary of Results for the Alphabet with Twenty Characters.

Instance		IP				CSP-R	GREEDY				
k	m	Min	Avg	Max	Time	Time	Min	Avg	Max	Time	Gap
3	300	183	185.5	188	0.02	0.02	183	185.5	188	0.00	1.000
	400	244	247.1	250	0.02	0.02	244	247.1	250	0.00	1.000
	500	305	308.9	315	0.03	0.03	305	308.9	315	0.01	1.000
	600	365	370.7	375	0.03	0.04	365	370.7	375	0.01	1.000
	700	426	431.6	435	0.04	0.06	426	431.6	435	0.01	1.000
	800	489	494.8	499	0.04	0.07	489	494.8	499	0.01	1.000
4	300	199	203.6	207	0.02	0.05	199	203.6	207	0.00	1.000
	400	268	271.5	276	0.03	0.08	268	271.5	276	0.01	1.000
	500	337	340.1	343	0.03	0.12	337	340.1	343	0.01	1.000
	600	401	407.0	412	0.04	0.18	401	407.0	412	0.01	1.000
	700	472	474.7	481	0.05	0.24	472	474.7	481	0.02	1.000
	800	539	542.8	547	0.05	0.32	539	542.8	547	0.02	1.000
5	300	211	213.4	218	0.02	0.23	211	213.4	218	0.01	1.000
	400	281	284.7	290	0.03	0.41	281	284.7	290	0.01	1.000
	500	353	356.4	359	0.04	0.63	353	356.4	359	0.01	1.000
	600	419	426.3	433	0.04	0.91	419	426.3	433	0.02	1.000
	700	493	498.9	503	0.05	1.23	493	498.9	503	0.02	1.000
	800	568	571.2	575	0.06	1.62	568	571.2	575	0.03	1.000
6	300	216	218.6	222	0.03	1.37	216	218.6	222	0.01	1.000
	400	289	293.5	297	0.03	2.44	289	293.5	297	0.01	1.000
	500	363	365.7	368	0.04	3.81	363	365.7	368	0.01	1.000
	600	435	438.5	442	0.05	5.43	435	438.5	442	0.02	1.000
	700	512	513.8	515	0.06	7.45	512	513.8	515	0.03	1.000
	800	582	585.8	589	0.07	9.69	582	585.8	589	0.03	1.000
7	300	223	224.7	227	0.03	9.65	223	224.7	227	0.01	1.000
	400	296	300.1	303	0.04	17.15	296	300.1	303	0.01	1.000
	500	372	373.9	376	0.05	26.60	372	373.9	376	0.02	1.000
	600	447	448.6	451	0.06	38.16	447	448.6	451	0.02	1.000
	700	520	522.7	526	0.07	51.91	520	522.7	526	0.03	1.000
	800	594	597.5	603	0.07	67.72	594	597.5	603	0.04	1.000

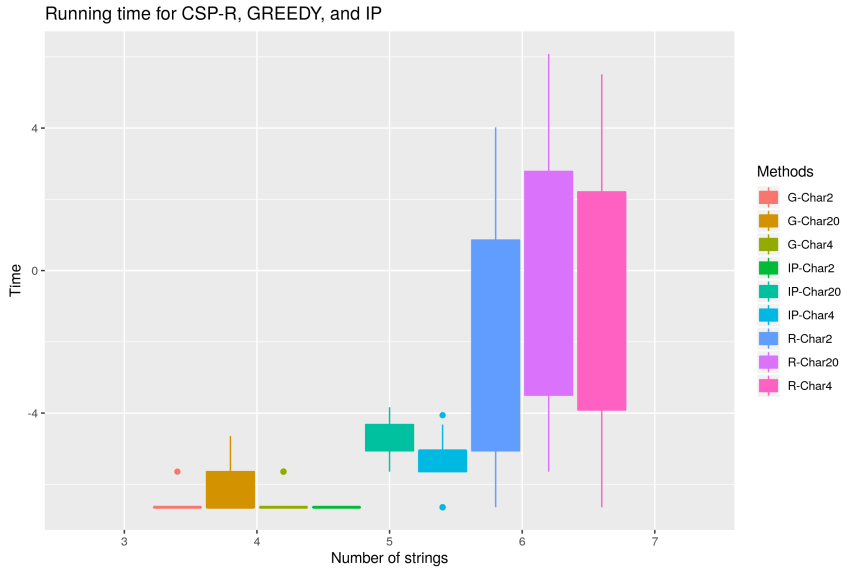


Figure 5.4: Boxplot in logarithmic scale of the running times for GREEDY and exact methods: The number of strings (x-axis) and Time in logarithmic scale (y-axis), legend: G-Char2 means GREEDY method applied for binary characters, IP-Char4 means Integer Programming formulation applied for DNA alphabet, R-Char20 means exact recursive approach applied for protein alphabet. Source: the authors.

able to find solutions within 50% of the optimal value in less than one second for instances with the protein alphabet.

In Table 5.9, GREEDY for 50% got the optimal solution value, that is, the worst Gap 1.013 was by the instances for 10-strings with 98 length of strings, and 12-strings with 98 length of strings, in those cases the algorithm gives a good approximation for an optimal solution value with 0.013% (error). Furthermore the running time for all real instances was less than one second for 20 characters. Because the greedy algorithm, for each iteration, traverses through the column-characters, after that, it makes a greedy-choice (minimizing the longest Hamming distance) among the best candidates, in this last step a bottleneck is created, the algorithm meets this problem by doing a depth search in three levels, thus increasing the potential of the proposed method.

Table 5.9: Summary of Results for the McClure Instances, over the Alphabet with 20 Characters.

Instance			IP		GREEDY		
Name	n	m	Val	Time	Val	Time	Gap
mc586.6.seq	6	100	72	0.011	**72**	0.187	1.000
mc582.6.seq	6	141	88	0.013	89	0.203	1.011
mc586.10.seq	10	98	75	0.023	76	0.345	1.013
mc582.10.seq	10	141	97	0.033	**97**	0.451	1.000
mc586.12.seq	12	98	77	0.016	78	0.987	1.013
mc582.12.seq	12	141	97	0.024	**97**	0.988	1.000

Chapter 6

CONCLUSION

The CSP is a problem widely explored in the literature, since 1997 when it was shown to be in NP-complete still for binary alphabet. Since that, different techniques have been used to solve the CSP. From its empirical side there are meta-heuristics and formulations in integer linear programming, to its theoretical side we have the approximation algorithms and the exact methods such as the fixed-parameter tractability.

In this work, we reviewed basic definitions for string selection problems related to Closest String Problem whose decision versions belong to NP-complete, furthermore, we showed a hierarchy of complexities with those problems. Thus the optimal solution for one of them will be a model to solve other, so find a method to solve can contribute to all those problems. Additionally we described the most important issues in this area, several methods, and algorithms derived from the literature review.

One of the first attempts to solve CSP was to use meta-heuristics in order to analyze its inherent parameters to the problem. In addition to the parameters (k) number of strings, (m) length of strings, (Γ) the alphabet, and (d) the Hamming distance; there is the structure of the character matrix associated to its complexity. Beside, other unknown properties. One of our limitations was to get the state-of-the-art instances to make proper comparisons with our proposed algorithm (GREEDY).

Our goal is to solve the problem accurately by using a direct combinatorial algorithmic technique parameterized by k (number of the input strings), with that objective, we designed and implemented exact methods for a few strings namely 3-CSP (for three strings) and 4-CSP (with four strings), linear-time algorithms, they are mainly a column-position type characterization. Additionally. For the general case, we proposed a greedy heuristic algorithm (GREEDY) and a recursive exact method.

The exact recursive algorithm repeatedly calls the greedy algorithm to build its partial

solutions from its base case (2-strings) until k (number of input strings). A possible improvement to this strategy would be to use a formulation in integer linear programming to build partial solutions for $k-1$ strings. We worked on this idea, unfortunately it didn't work out, so we couldn't report it.

Furthermore, our hypothesis was positive answered since there is a fixed parameter algorithm, our proposed method, called exact recursive algorithm for the general case with running time of $\mathcal{O}(k^{k+1}.m^2)$.

6.1 Future Works

In the present work we deal with the CSP specifically with the Hamming distance as a parameter, in the literature we can find the CSP with other metrics such as edit distance, Levenshtein distance, and ranking distance, the proposed methods could be used to solve such similar problems.

Since we have an exact combinatorial method and in the literature there are many formulations in integer programming. We can use a hybrid approach to merge the best of two techniques, so we can build a matheuristic using an integer linear programming formulation plus the exact recursive method for solving the CSP.

The exact recursive method builds its optimal solution value using partial solutions of smaller instances, these solutions are calculated on repeated occasions, so we could save those partial solutions to reuse them, thus avoiding the unnecessary construction of the same partial solutions over and over again. Based on this analysis, as a future work we intend to improve the performance of the exact recursive algorithm using a dynamic programming approach.

References

[Amir et al., 2009] Amir, A., Landau, G. M., Na, J. C., Park, H., Park, K., and Sim, J. S. (2009). *Consensus Optimizing Both Distance Sum and Radius*, pages 234–242. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Amir et al., 2016] Amir, A., Paryenty, H., and Roditty, L. (2016). Configurations and minority in the string consensus problem. *Algorithmica*, 74(4):1267–1292.

[Andoni et al., 2006] Andoni, A., Indyk, P., and Patrascu, M. (2006). On the optimality of the dimensionality reduction method. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 449–458. IEEE.

[Arbib et al., 2017] Arbib, C., Servilio, M., and Ventura, P. (2017). An improved integer linear programming formulation for the closest 0-1 string problem. *Computers & Operations Research*, 80:94 – 100.

[Babaie and Mousavi, 2010] Babaie, M. and Mousavi, S. R. (2010). A memetic algorithm for closest string problem and farthest string problem. In *18th Iranian Conference on Electrical Engineering (ICEE), 2010*, pages 570–575. IEEE.

[Basavaraju et al., 2014] Basavaraju, M., Panolan, F., Rai, A., Ramanujan, M., and Saurabh, S. (2014). On the kernelization complexity of string problems. In *International Computing and Combinatorics Conference*, pages 141–153. Springer.

[Ben-Dor et al., 1997] Ben-Dor, A., Lancia, G., Perone, J., and Ravi, R. (1997). Banishing bias from consensus sequences. In Apostolico, A. and Hein, J., editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, number 1264 in Lecture notes in computer science, pages 247–261, Aarhus, Denmark. Springer-Verlag.

[Berman et al., 1997] Berman, P., Gumucio, D., Hardison, R., Miller, W., and Stojanovic, N. (1997). A linear-time algorithm for the 1-mismatch problem. In *WADS'97*.

[Bixby, 1987] Bixby, R. E. (1987). Notes on combinatorial optimization. Technical report, Rice University, Department of Computational and Applied Mathematics.

[Boucher et al., 2009] Boucher, C., Brown, D. G., and Durocher, S. (2009). *On the Structure of Small Motif Recognition Instances*, pages 269–281. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Boucher et al., 2012] Boucher, C., Landau, G. M., Levy, A., Pritchard, D., and Weimann, O. (2012). On approximating string selection problems with outliers. *CoRR*, abs/1202.2820:427–438.

[Boucher et al., 2015] Boucher, C., Lo, C., and Lokshantov, D. (2015). Consensus patterns (probably) has no eptas. In *Algorithms-ESA 2015*, pages 239–250. Springer.

[Bulteau et al., 2014] Bulteau, L., Hüffner, F., Komusiewicz, C., and Niedermeier, R. (2014). Multivariate algorithmics for np-hard string problems. *Bulletin of EATCS*, 3(114).

[Chen, 2007] Chen, J.-C. (2007). Iterative rounding for the closest string problem. *arXiv preprint arXiv:0705.0561*.

[Chimani et al., 2011] Chimani, M., Woste, M., and Böcker, S. (2011). A closer look at the closest string and closest substring problem. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 13–24. Society for Industrial and Applied Mathematics.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. third edition.

[Cornell, 2016] Cornell, B. (2016).

[CPLEX, 2014] CPLEX (2014). *CPLEX 12.6.1 Reference manual*. IBM ILOG CPLEX.

[Croce and Garraffa, 2014] Croce, F. D. and Garraffa, M. (2014). The selective fixing algorithm for the closest string problem. *Computers & Operations Research*, 41:24 – 30.

[Dalpasso and Lancia, 2018] Dalpasso, M. and Lancia, G. (2018). New modeling ideas for the exact solution of the closest string problem. In Elloumi, M., Granitzer, M., Hameurlain, A., Seifert, C., Stein, B., Tjoa, A. M., and Wagner, R., editors, *Database and Expert Systems Applications*, pages 105–114, Cham. Springer International Publishing.

[Della Croce and Salassa, 2012] Della Croce, F. and Salassa, F. (2012). Improved lp-based algorithms for the closest string problem. *Computers & Operations Research*, 39(3):746–749.

[Deng et al., 2002] Deng, X., Li, G., and Wang, L. (2002). Center and distinguisher for strings with unbounded alphabet. *Journal of Combinatorial Optimization*, 6(4):383–400.

[Downey and Fellows, 1999] Downey, R. G. and Fellows, M. R. (1999). *Parameterized Complexity*. Springer-Verlag.

[Faro and Pappalardo, 2010] Faro, S. and Pappalardo, E. (2010). Ant-csp: An ant colony optimization algorithm for the closest string problem. In *SOFSEM*, volume 5901, pages 370–381. Springer.

[Fellows, 2002] Fellows, M. (2002). Parameterized complexity: the main ideas and connections to practical computing. In *Experimental algorithmics*, number 2547 in Lecture notes of computer science, pages 51–77. Springer-Verlag.

[Ferrer et al., 2010] Ferrer, M., Valveny, E., Serratosa, F., Riesen, K., and Bunke, H. (2010). Generalized median graph computation by means of graph embedding in vector spaces. *Pattern Recogn.*, 43(4):1642–1655.

[Frances and Litman, 1997] Frances, M. and Litman, A. (1997). On covering problems of codes. *Theor. Comput. Syst.*, 30:113–119.

[Gasieniec et al., 1999] Gasieniec, L., Jansson, J., and Lingas, A. (1999). Efficient approximation algorithms for the hamming center problem. pages 905–906.

[Gomes et al., 2008] Gomes, F. C., Meneses, C. N., Pardalos, P. M., and Viana, G. V. R. (2008). A parallel multistart algorithm for the closest string problem. *Computers & Operations Research*, 35(11):3636–3643.

[Gramm et al., 2001] Gramm, J., Niedermeier, R., and Rossmanith, P. (2001). Exact solutions for closest string and related problems. In *In Proceedings of the 12th Annual International Symposium on Algorithms and Computation (ISAAC 2001)*, volume 2223 of *Lecture Notes in Computer Science*, pages 441–452. Springer Verlag.

[Gramm et al., 2003] Gramm, J., Niedermeier, R., and Rossmanith, P. (2003). Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37:25–42.

[Hill, 1986] Hill, R. (1986). *A First Course in Coding Theory*. Oxford Applied Linguistics. Clarendon Press.

[Hufsky et al., 2011] Hufsky, F., Kuchenbecker, L., Jahn, K., Stoye, J., and Böcker, S. (2011). Swiftly computing center strings. *BMC Bioinformatics*, 12(1):106.

[Julstrom, 2009] Julstrom, B. A. (2009). A data-based coding of candidate strings in the closest string problem. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2053–2058. ACM.

[Keith et al., 2002] Keith, J. M., Adams, P., Bryant, D. E., Kroese, D. P., Mitchelson, K. R., Cochran, D. A. E., and Lala, G. H. (2002). A simulated annealing algorithm for finding consensus sequences. *Bioinformatics*, 18(11):1494âĂŞ1499.

[Lance and Williams, 1967] Lance, G. N. and Williams, W. T. (1967). A general theory of classificatory sorting strategies: 1. hierarchical systems. *Computer J.*, 9:373–380.

[Lanctot et al., 2003] Lanctot, K., Li, M., Ma, B., Wang, S., and Zhang, L. (2003). Distinguishing string selection problems. *Information and Computation*, 185(1):41–55.

[Latorre and de Freitas, 2016] Latorre, O. and de Freitas, R. (2016). An efficient algorithm for the closest string problem. *XXXVI Congresso da Sociedade Brasileira de Computação, I ETC*, pages 281–284.

[Latorre and de Freitas, 2018] Latorre, O. and de Freitas, R. (2018). An efficient combinatorial algorithm for closest string problem with four strings. *XIX Latin-Iberoamerican conference on operations research*, pages 171–178.

[Latorre and Salvatierra, 2019] Latorre, O. and Salvatierra, M. (2019). A recursive exact algorithm for the closest string problem. *Journal of Combinatorial Mathematics and Combinatorial Computing*, pages 171–185.

[Lenstra, 1983] Lenstra, H. W. (1983). Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548.

[Li et al., 1999] Li, M., Ma, B., and Wang, L. (1999). Finding similar regions in many strings. *Proceedings of the Thirty First Annual ACM Symposium on Theory of Computing, Atlanta*, pages 473–482.

[Li et al., 2002] Li, M., Ma, B., and Wang, L. (2002). On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171.

[Liu et al., 2005] Liu, X., He, H., and Sỳkora, O. (2005). Parallel genetic algorithm and parallel simulated annealing algorithm for the closest string problem. In *Advanced Data Mining and Applications*, pages 591–597. Springer.

[Liu et al., 2008] Liu, X., Holger, M., Hao, Z., and Wu, G. (2008). A compounded genetic and simulated annealing algorithm for the closest string problem. In *The 2nd International Conference on Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008.*, pages 702–705. IEEE.

[Liu et al., 2011] Liu, X., Liu, S., Hao, Z., and Mauch, H. (2011). Exact algorithm and heuristic for the closest string problem. *Computers & Operations Research*, 38(11):1513–1520.

[Ma and Sun, 2008] Ma, B. and Sun, X. (2008). *More Efficient Algorithms for Closest String and Substring Problems*, pages 396–409. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Mauch et al., 2003] Mauch, H., Melzer, M. J., and Hu, J. S. (2003). Genetic algorithm approach for the closest string problem. In *Proceedings of the 2003 IEEE Bioinformatics Conference, 2003. CSB 2003.*, pages 560–561. IEEE.

[McClure et al., 1994] McClure, M., Vasi, T., and Fitch, W. (1994). Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 11:571–592.

[Meneses et al., 2004] Meneses, C., Lu, Z., Oliveira, C., and Pardalos, P. (2004). Optimal solutions for the closest string problem via integer programming. *INFORMS Journal on Computing*, 16(4):419–429.

[Metropolis and Ulam, 1949] Metropolis, N. and Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, 44(247):335–341.

[Mousavi and Esfahani, 2012] Mousavi, S. R. and Esfahani, N. N. (2012). A grasp algorithm for the closest string problem using a probability-based heuristic. *Computers & Operations Research*, 39(2):238–248.

[Nienkötter and Jiang, 2016] Nienkötter, A. and Jiang, X. (2016). Distance-preserving vector space embedding for the closest string problem. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 1530–1535.

[Panneton et al., 2006] Panneton, F., L'Ecuyer, P., and Matsumoto, M. (2006). Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.*, 32(1):1–16.

[Pappalardo et al., 2014] Pappalardo, E., Cantone, D., and Pardalos, P. M. (2014). A combined greedy-walk heuristic and simulated annealing approach for the closest string problem. *Optimization Methods and Software*, 29(4):673–702.

[Roman, 1992] Roman, S. (1992). *Coding and Information Theory*, volume 134 of *Graduate Texts in Mathematics*. Springer-Verlag.

[Shaik et al., 2019] Shaik, N., Hakeem, K., Banaganapalli, B., and Elango, R. (2019). *Essentials of Bioinformatics, Volume I: Understanding Bioinformatics: Genes to Proteins*. Number v. 1 in Lecture notes in Bioinformatics. Springer International Publishing.

[Sze et al., 2004] Sze, S.-H., Lu, S., and Chen, J. (2004). Integrating sample-driven and pattern-driven approaches in motif finding. In *International Workshop on Algorithms in Bioinformatics*, pages 438–449. Springer.

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

[Vilca and Meneses, 2012] Vilca, O. L. and Meneses, C. N. (2012). Planos de corte e heurísticas para o closest string problem. *CLAIO & SBPO*.

[Wolsey, 1998] Wolsey, L. (1998). *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley.

Appendix A

Step-by-step examples

A.1 MSA step-by-step example

Minimization Second Algorithm (MSA) is an iterative algorithm to solve the 4-CSP, that is, four strings with a binary alphabet. In the following we explain step-by-step example, let $\mathcal{S}$ be a 4-CSP instance with a binary alphabet, that is, $\mathcal{S} = \{01011, 01000, 10011, 10110\}$.

Input instance

$$S_1 = \begin{cases} 01011 \\ 01000 \\ 10011 \\ 10110 \end{cases}$$

Normalized instance

$$S_2 = \left\{ \begin{array}{|ccccc|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} \right.$$

n_0	n_1	n_2	n_3	n_4
0	0	1	0	1

u :

3	4
4	2

Algorithm: MSA step-by-step iteration 1

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**

for $j \leftarrow 1, \ldots, m$ **do**
$u[j] \leftarrow -\infty$;
for $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
$n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;

$\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
$count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
for $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
$x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
for $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
$x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
$d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
$i \leftarrow$ indexMaximum(d);
$k \leftarrow$ indexMinimum(d);
switch *frequencyOfNumber*$(d[i], d)$ **do**
case *1 or 4* **do**
$x[j] \leftarrow \mathcal{S}[i][j]$;
case *2 or 3* **do**
if $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
else
$r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
$x[j] \leftarrow \mathcal{S}[r][j]$;
updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} \right.$$

n_0	n_1	n_2	n_3	n_4
0	0	1	0	1

u :

3	4
4	2

η_1	I_{η_1}	η_2	*count*
1	2	1	0

Algorithm: MSA step-by-step iteration 2

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- *count* $\leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and count* > 0 **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; *count* $\leftarrow$ *count* $- 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} \right.$$

x : ␣␣11␣

n_0	n_1	n_2	n_3	n_4
0	0	1	0	1

u :

3	4
4	2

η_1	I_{η_1}	η_2	*count*
1	2	1	0

Algorithm: MSA step-by-step iteration 3

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- *count* $\leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and count* > 0 **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; *count* $\leftarrow$ *count* $- 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{\begin{array}{|ccccc|}\hline 1&2&3&4&5\\\hline 0&0&1&1&0\\0&0&1&0&1\\1&1&1&1&0\\1&1&0&1&1\\\hline\end{array}\right.$$

$x : \text{␣␣11␣}$

n_0	n_1	n_2	n_3	n_4
0	0	1	0	1

$u :$

3	4
4	2

η_1	I_{η_1}	η_2	$count$
1	2	1	0

Algorithm: MSA step-by-step iteration 4

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor(\eta_1 - \eta_2 - \lfloor(m - (n_0 + \cdots + n_4))/2\rfloor)/2\rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{\begin{array}{|ccccc|}\hline 1&2&3&4&5\\\hline 0&0&1&1&0\\0&0&1&0&1\\1&1&1&1&0\\1&1&0&1&1\\\hline\end{array}\right.$$

$x : \text{␣␣11␣}$

$d : \begin{bmatrix} 3 & 4 & 3 & 4 \end{bmatrix}$

Algorithm: MSA step-by-step iteration 5

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor(\eta_1 - \eta_2 - \lfloor(m - (n_0 + \cdots + n_4))/2\rfloor)/2\rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$x : \text{␣␣11␣}$

$d : \begin{bmatrix} 3 & 4 & 3 & 4 \end{bmatrix}$

$j = 1,\ i = 2,\ k = 1$

Algorithm: MSA step-by-step iteration 6

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$x : \text{1␣11␣}$

$d : \begin{bmatrix} 3 & 4 & 3 & 4 \end{bmatrix}$

$j = 1,\ i = 2,\ k = 1,\ r = 3$

Algorithm: MSA step-by-step iteration 7

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$$x : 1\_11\_$$

$$d : \begin{bmatrix} 3 & 4 & 2 & 3 \end{bmatrix}$$

$$j = 1,\ i = 2,\ k = 1,\ r = 3$$

Algorithm: MSA step-by-step iteration 8

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**

- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$$x : 1\_11\_$$

$$d : \begin{bmatrix} 3 & 4 & 2 & 3 \end{bmatrix}$$

$$j = 2,\ i = 2,\ k = 3$$

Algorithm: MSA step-by-step iteration 9

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**

- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$$x : 1011\_$$

$$d : \begin{bmatrix} 3 & 4 & 2 & 3 \end{bmatrix}$$

$$j = 2,\ i = 2,\ k = 3$$

Algorithm: MSA step-by-step iteration 10

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$$x : 1011\_$$

$$d : \begin{bmatrix} 2 & 3 & 2 & 3 \end{bmatrix}$$

$$j = 2,\ i = 2,\ k = 3$$

Algorithm: MSA step-by-step iteration 11

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$x : 1011\_$

$d : \begin{bmatrix} 2 & 3 & 2 & 3 \end{bmatrix}$

$j = 5,\ i = 2,\ k = 1$

Algorithm: MSA step-by-step iteration 12

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$x : 10111$

$d : \begin{bmatrix} 2 & 3 & 2 & 3 \end{bmatrix}$

$j = 5,\ i = 2,\ k = 1$

Algorithm: MSA step-by-step iteration 13

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**
- **for** $j \leftarrow 1, \ldots, m$ **do**
  - $u[j] \leftarrow -\infty$;
  - **for** $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**
    - $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;
- $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;
- $count \leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**
  - $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and* $count > 0$ **do**
  - $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; $count \leftarrow count - 1$;
- $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;
- **for** $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  - $i \leftarrow$ indexMaximum(d);
  - $k \leftarrow$ indexMinimum(d);
  - **switch** *frequencyOfNumber*$(d[i], d)$ **do**
    - **case** *1 or 4* **do**
      - $x[j] \leftarrow \mathcal{S}[i][j]$;
    - **case** *2 or 3* **do**
      - **if** $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;
      - **else**
        - $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;
        - $x[j] \leftarrow \mathcal{S}[r][j]$;
  - updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

$$S_2 = \left\{ \begin{array}{|ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right.$$

$$x : 10111$$

$$d : \left[\begin{array}{cccc} 2 & 2 & 2 & 2 \end{array} \right]$$

$$j = 5,\ i = 2,\ k = 1$$

Algorithm: MSA step-by-step iteration 14

if $|S| = 4$ *and* $|\Gamma| = 2$ **then**

  for $j \leftarrow 1, \ldots, m$ **do**

    $u[j] \leftarrow -\infty$;

    for $i \leftarrow 0, \ldots, 4$ *such that* $(\mathcal{S}[.][j])^t = v_i$ **do**

      $n_i \leftarrow n_i + 1$; $u[j] \leftarrow i$;

  $\eta_1, I_{\eta_1}, \eta_2 \leftarrow$ twoFirstGreatestNumber$(n_1, \ldots, n_4)$;

  count $\leftarrow \lfloor (\eta_1 - \eta_2 - \lfloor (m - (n_0 + \cdots + n_4))/2 \rfloor)/2 \rfloor$;

  for $j \leftarrow 1, \ldots, m$ *such that* $u[j] \geq 0$ **do**

    $x[j] \leftarrow$ majority$(\mathcal{S}[.][j])$;

  for $j \leftarrow 1, \ldots, m$ *such that* $u[j] = I_{\eta_1}$ *and count* > 0 **do**

    $x[j] \leftarrow$ minority$(\mathcal{S}[.][j])$; *count* $\leftarrow$ *count* $- 1$;

  $d[.] \leftarrow d_H(x, \mathcal{S}[.])$;

  for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

    $i \leftarrow$ indexMaximum(d);

    $k \leftarrow$ indexMinimum(d);

    switch *frequencyOfNumber*$(d[i], d)$ **do**

      case *1 or 4* **do**

        $x[j] \leftarrow \mathcal{S}[i][j]$;

      case *2 or 3* **do**

        if $\mathcal{S}[i][j] \neq \mathcal{S}[k][j]$ **then** $x[j] \leftarrow \mathcal{S}[i][j]$;

        else

          $r \leftarrow$ index$(\mathcal{S}[.][j], i, k)$;

          $x[j] \leftarrow \mathcal{S}[r][j]$;

    updateHammingDistance$(\mathcal{S}[.][j], x[j], d[.])$;

A.2 GREEDY step-by-step example

Polynomial Heuristic Greedy algorithm (GREEDY) is an iterative algorithm to solve the CSP for the general case. In the following we explain a step-by-step example, let $\mathcal{S}$ be a 3-CSP instance with a DNA alphabet, that is, $\mathcal{S} = \{GTCC, AGAG, CGAG\}$.

Input instance

$$S_1 = \begin{cases} GTCC \\ AGAG \\ CGAG \end{cases}$$

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Algorithm: GREEDY step-by-step iter. 1

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while $\#unfixed > 0$ **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if $\#unfixed > 3$ **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

Input instance

$$S_1 = \begin{cases} GTCC \\ AGAG \\ CGAG \end{cases}$$

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	**1**	**0**
1	0	0	1	0	1	0	**0**	**1**
0	1	0	1	0	1	0	**0**	**1**
1	1	1	2	1	2	1	**1**	**2**

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Algorithm: GREEDY step-by-step iter. 2

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while $\#unfixed > 0$ **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if $\#unfixed > 3$ **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

Input instance

$$S_1 = \begin{cases} GTCC \\ AGAG \\ CGAG \end{cases}$$

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
4	4	3	4	3	4	3	3	4
3	4	4	3	4	3	4	4	3
4	3	4	3	4	3	4	4	3
4	4	4	4	4	4	4	4	4
2	2	2	1	2	1	2	2	1

Algorithm: GREEDY step-by-step iter. 3

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while $\#unfixed > 0$ **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if $\#unfixed > 3$ **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

Input instance

$$S_1 = \begin{cases} GTCC \\ AGAG \\ CGAG \end{cases}$$

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H and T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

1			2		3		4	
A	C	G	**G**	T	**A**	C	C	**G**
4	4	3	**4**	3	**4**	3	3	**4**
3	4	4	**3**	4	**3**	4	4	**3**
4	3	4	**3**	4	**3**	4	4	**3**
4	4	4	**4**	4	**4**	4	4	**4**
2	2	2	**1**	2	**1**	2	2	**1**

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	**G**	T	**A**	C	C	**G**
4	4	3	**4**	3	**4**	3	3	**4**
3	4	4	**3**	4	**3**	4	4	**3**
4	3	4	**3**	4	**3**	4	4	**3**
4	4	4	**4**	4	**4**	4	4	**4**
2	2	2	**1**	2	**1**	2	2	**1**

$\mathcal{Q} : \{(2, G), (3, A), (4, G)\}$

Algorithm: GREEDY step-by-step iter. 4

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

Algorithm: GREEDY step-by-step iter. 5

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : ␣G␣␣ d : 4 3 3

3-Dimensional matrix T'

1			2		3		4	
A	C	**G**	G	T	A	**C**	**C**	G
4	4	**3**	0	0	4	**3**	**3**	4
2	3	**3**	0	0	2	**3**	**3**	2
3	2	**3**	0	0	2	**3**	**3**	2
4	4	**3**	0	0	4	**3**	**3**	2
1	1	**3**	0	0	1	**3**	**3**	1

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GG␣␣ d : 3 3 3

3-Dimensional matrix T'

1			2		3		4	
A	C	G	G	T	**A**	C	C	**G**
0	0	0	0	0	**3**	2	2	**3**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**3**	3	3	**3**
0	0	0	0	0	**1**	2	2	**1**

Algorithm: GREEDY step-by-step iter. 6

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

Algorithm: GREEDY step-by-step iter. 7

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGA␣ d : 3 2 2

3-Dimensional matrix T'

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 8

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

3-Dimensional matrix T'

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 9

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : ␣␣A␣ d : 4 3 3

1			2		3		4	
A	C	**G**	G	**T**	A	C	**C**	G
4	4	**3**	4	**3**	0	0	**3**	4
2	3	**3**	2	**3**	0	0	**3**	2
3	2	**3**	2	**3**	0	0	**3**	2
4	4	**3**	4	**3**	0	0	**3**	4
1	1	**3**	1	**3**	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 10

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : G␣A␣ d : 3 3 3

1			2		3		4	
A	C	G	**G**	T	A	C	C	**G**
0	0	0	**3**	2	0	0	2	**3**
0	0	0	**2**	3	0	0	3	**2**
0	0	0	**2**	3	0	0	3	**2**
0	0	0	**3**	3	0	0	3	**3**
0	0	0	**1**	2	0	0	2	**1**

Algorithm: GREEDY step-by-step iter. 11

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : GGA␣ d : 3 2 2

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 12

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
- $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
    - $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
- **return** chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : GGAC d : 2 2 2

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 13

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
- $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
    - $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
- **return** chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : GGAC d : 2 2 2

$(4, G)$ x' : ␣␣␣G d : 4 3 3

1			2		3		4	
A	C	**G**	G	**T**	A	**C**	C	G
4	4	**3**	4	**3**	4	**3**	0	0
2	3	**3**	2	**3**	2	**3**	0	0
3	2	**3**	2	**3**	2	**3**	0	0
4	4	**3**	4	**3**	4	**3**	0	0
1	1	**3**	1	**3**	1	**3**	0	0

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:$\{(2, G), (3, A), (4, G)\}$

$(2, G)$ x' : GGAC d : 2 2 2

$(3, A)$ x' : GGAC d : 2 2 2

$(4, G)$ x' : G␣␣G d : 3 3 3

1			2		3		4	
A	C	G	**G**	T	**A**	C	C	G
0	0	0	**3**	2	**3**	2	0	0
0	0	0	**2**	3	**2**	3	0	0
0	0	0	**2**	3	**2**	3	0	0
0	0	0	**3**	3	**3**	3	0	0
0	0	0	**1**	2	**1**	2	0	0

Algorithm: GREEDY step-by-step iter. 14

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$
        updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

Algorithm: GREEDY step-by-step iter. 15

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
  $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
  foreach $(j, \sigma) \in \mathcal{Q}$ **do**
    $x'' \leftarrow x'$
    $d''[.] \leftarrow d'[.]$
    for $i \leftarrow 1, \ldots, 3$ **do**
      $x''[j] \leftarrow \sigma$
      $d''[.] \leftarrow$
        updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
      $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
    $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
  return chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:{(2, G), (3, A), (4, G)}

(2, G) x' : GGAC d : 2 2 2

(3, A) x' : GGAC d : 2 2 2

(4, G) x' : GG␣G d : 3 2 2

1			2		3		4	
A	C	G	G	T	A	**C**	C	G
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**3**	0	0

Algorithm: GREEDY step-by-step iter. 16

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
- $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
    - $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
- **return** chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:{(2, G), (3, A), (4, G)}

(2, G) x' : GGAC d : 2 2 2

(3, A) x' : GGAC d : 2 2 2

(4, G) x' : GGCG d : 2 2 2

1			2		3		4	
A	C	G	G	T	A	**C**	C	G
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**3**	0	0

Algorithm: GREEDY step-by-step iter. 17

Function depthSearch3Levels($\mathcal{H}, \mathcal{T}', d'[.], x'$)
- $\mathcal{Q} \leftarrow$ chooseBestValues($\mathcal{H}, \mathcal{T}', x'$)
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ updateHammingDistance($\mathcal{S}[.][j], x''[j], d''[.]$)
    - $\mathcal{T}' \leftarrow$ updateTable($\mathcal{H}, d''[.], x'', \mathcal{T}'$)
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)
- **return** chooseBestValue($\mathcal{H}, \mathcal{T}'', x'$)

x : ␣␣␣␣ d : 4 4 4

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

$\mathcal{Q}$:{(2, G), (3, A), (4, G)}

(2,G) x' : **GGAC** d : 2 2 2

(3, A) x' : GGAC d : 2 2 2

(4, G) x' : GGCG d : 2 2 2

1			2		3		4	
A	C	G	G	T	A	**C**	C	G
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**3**	0	0

x : ␣G␣␣ d : 4 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	**G**	T	**A**	C	C	**G**
4	4	3	**4**	3	**4**	3	3	**4**
3	4	4	**3**	4	**3**	4	4	**3**
4	3	4	**3**	4	**3**	4	4	**3**
4	4	4	**4**	4	**4**	4	4	**4**
2	2	2	**1**	2	**1**	2	2	**1**

Algorithm: GREEDY step-by-step iter. 18

Function `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d'[.], x')$
- $\mathcal{Q} \leftarrow$ `chooseBestValues`$(\mathcal{H}, \mathcal{T}', x')$
- **foreach** $(j, \sigma) \in \mathcal{Q}$ **do**
  - $x'' \leftarrow x'$
  - $d''[.] \leftarrow d'[.]$
  - **for** $i \leftarrow 1, \ldots, 3$ **do**
    - $x''[j] \leftarrow \sigma$
    - $d''[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x''[j], d''[.])$
    - $\mathcal{T}' \leftarrow$ `updateTable`$(\mathcal{H}, d''[.], x'', \mathcal{T}')$
  - $\mathcal{T}'' \leftarrow \mathcal{T}'' \cup$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}'', x')$
- **return** `chooseBestValue`$(\mathcal{H}, \mathcal{T}'', x')$

Algorithm: GREEDY step-by-step iter. 19

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
- **foreach** $\sigma \in \mathcal{S}[.][j]$ **do**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - **if** $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
    - **else** $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
    - $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**
- $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
- $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
- **if** *#unfixed* > 3 **then**
  - $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
- **else** $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
- $x[j] \leftarrow \sigma$
- $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
- #unfixed $\leftarrow$ #unfixed -1

x : ␣G␣␣ d : 4 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
4	4	3	0	0	4	3	3	4
2	3	3	0	0	2	3	3	2
3	2	3	0	0	2	3	3	2
4	4	3	0	0	4	3	3	4
1	1	3	0	0	1	3	3	1

Algorithm: GREEDY step-by-step iter. 20

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if *#unfixed* > 3 **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

x : ␣G␣␣ d : 4 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	**G**	G	T	A	**C**	**C**	G
4	4	**3**	0	0	4	**3**	**3**	4
2	3	**3**	0	0	2	**3**	**3**	2
3	2	**3**	0	0	2	**3**	**3**	2
4	4	**3**	0	0	4	**3**	**3**	4
1	1	**3**	0	0	1	**3**	**3**	1

Algorithm: GREEDY step-by-step iter. 21

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if *#unfixed* > 3 **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

x : GG␣␣ d : 3 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	**G**	G	T	A	**C**	**C**	G
4	4	**3**	0	0	4	**3**	**3**	4
2	3	**3**	0	0	2	**3**	**3**	2
3	2	**3**	0	0	2	**3**	**3**	2
4	4	**3**	0	0	4	**3**	**3**	4
1	1	**3**	0	0	1	**3**	**3**	1

Algorithm: GREEDY step-by-step iter. 22

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if *#unfixed* > 3 **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

x : GG␣␣ d : 3 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	3	2	2	3
0	0	0	0	0	2	3	3	2
0	0	0	0	0	2	3	3	2
0	0	0	0	0	3	3	3	3
0	0	0	0	0	1	2	2	1

Algorithm: GREEDY step-by-step iter. 23

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if *#unfixed* > 3 **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

x : GG␣␣ d : 3 3 3

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	**A**	C	C	**G**
0	0	0	0	0	**3**	2	2	**3**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**3**	3	3	**3**
0	0	0	0	0	**1**	2	2	**1**

Algorithm: GREEDY step-by-step iter. 24

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

x : GGA␣ d : 3 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	**A**	C	C	**G**
0	0	0	0	0	**3**	2	2	**3**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**2**	3	3	**2**
0	0	0	0	0	**3**	3	3	**3**
0	0	0	0	0	**1**	2	2	**1**

Algorithm: GREEDY step-by-step iter. 25

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

x : GGA␣ d : 3 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	0	0	2	3
0	0	0	0	0	0	0	2	1
0	0	0	0	0	0	0	2	1
0	0	0	0	0	0	0	2	3
0	0	0	0	0	0	0	3	1

Algorithm: GREEDY step-by-step iter. 26

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

x : GGA␣ d : 3 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 27

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
`/* Construction phase */`
for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**
  foreach $\sigma \in \mathcal{S}[.][j]$ **do**
    for $i \leftarrow 1, \ldots, k$ **do**
      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;
      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;
      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$
`/* Improve-solution */`
while *#unfixed* > 0 **do**
  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$
  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$
  if *#unfixed* > 3 **then**
    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;
  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;
  $x[j] \leftarrow \sigma$
  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$
  #unfixed $\leftarrow$ #unfixed -1

x : GGAC d : 2 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	**C**	G
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	1
0	0	0	0	0	0	0	**2**	3
0	0	0	0	0	0	0	**3**	1

Algorithm: GREEDY step-by-step iter. 28

$d[.] \leftarrow d_H(x, \mathcal{S}[.])$

$\mathcal{H} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$

`/* Construction phase */`

for $j \leftarrow 1, \ldots, m$ *such that* $x[j]$ *is unfixed* **do**

  foreach $\sigma \in \mathcal{S}[.][j]$ **do**

    for $i \leftarrow 1, \ldots, k$ **do**

      if $\sigma = \mathcal{S}[i][j]$ **then** $\mathcal{H}[j][\sigma][i] \leftarrow 1$;

      else $\mathcal{H}[j][\sigma][i] \leftarrow 0$;

      $\mathcal{H}[j][\sigma][k+1] \leftarrow \mathcal{H}[j][\sigma][k+1] + \mathcal{H}[j][\sigma][i]$

`/* Improve-solution */`

while *#unfixed* > 0 **do**

  $\mathcal{T} \leftarrow$ `updateTable`$(\mathcal{H}, d[.], x, \mathcal{T})$

  $(\mathcal{T}', d', x') \leftarrow (\mathcal{T}, d, x)$

  if *#unfixed* > 3 **then**

    $(j, \sigma) \leftarrow$ `depthSearch3Levels`$(\mathcal{H}, \mathcal{T}', d', x')$;

  else $(j, \sigma) \leftarrow$ `chooseBestValue`$(\mathcal{H}, \mathcal{T}, x)$;

  $x[j] \leftarrow \sigma$

  $d[.] \leftarrow$ `updateHammingDistance`$(\mathcal{S}[.][j], x[j], d[.])$

  #unfixed $\leftarrow$ #unfixed -1

A.3 CSP-R step-by-step example

Recursive exact method (CSP-R) is recursive algorithm to solve the CSP for the general case. In the following we explain a step-by-step example, let $\mathcal{S}$ be a 3-CSP instance with a DNA alphabet, that is, $\mathcal{S} = \{GTCC, AGAG, CGAG\}$.

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

Algorithm: CSP-R step-by-step iter. 1

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

Algorithm: CSP-R step-by-step iter. 2

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

Algorithm: CSP-R step-by-step iter. 3

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^2 : \mathbf{AGAG} \\ s^3 : C\mathbf{GAG} \end{cases}$$

$x : AGAG$

Algorithm: CSP-R step-by-step iter. 4

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^2 : \mathbf{AGAG} \\ s^3 : C\mathbf{GAG} \end{cases}$$

$$x : AGAG$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 5

Function CSP-R$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : GTCC \\ s^3 : CGAG \end{cases}$$

$$x :$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 6

Function CSP-R$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : \mathbf{G}T\mathbf{C}C \\ s^3 : C\mathbf{G}A\mathbf{G} \end{cases}$$

$$x : GGCG$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 7

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : \mathbf{G}T\mathbf{C}C \\ s^3 : C\mathbf{G}A\mathbf{G} \end{cases}$$

$$x : GGCG$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : GGCG \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 8

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \end{cases}$$

$$x :$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : GGCG \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 9

Function CSP-R$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : \mathbf{G}T\mathbf{C}C \\ s^2 : A\mathbf{G}A\mathbf{G} \end{cases}$$

$$x : GGCG$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : GGCG \\ x^3 : \end{cases}$$

Algorithm: CSP-R step-by-step iter. 10

Function CSP-R$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S' = \begin{cases} s^1 : \mathbf{G}T\mathbf{C}C \\ s^2 : A\mathbf{G}A\mathbf{G} \end{cases}$$

$$x : GGCG$$

$$S'' = \begin{cases} x^1 : AGAG \\ x^2 : GGCG \\ x^3 : GGCG \end{cases}$$

Algorithm: CSP-R step-by-step iter. 11

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$$S'' = \begin{cases} x^1 : A\mathbf{G}A\mathbf{G} \\ x^2 : G\mathbf{G}C\mathbf{G} \\ x^3 : G\mathbf{G}C\mathbf{G} \end{cases}$$

x : ␣G␣G d : 4 2 2

Algorithm: CSP-R step-by-step iter. 12

Function `CSP-R`$(\mathcal{S}, k, m, x)$

- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ `CSP-R`$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ `minimizeMaximumValue`$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$x:$ ␣G␣G $\quad d:$ 4 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Algorithm: CSP-R step-by-step iter. 13

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$x:$ ␣G␣G $\quad d:$ 4 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
4	4	3	0	0	4	3	0	0
1	2	2	0	0	1	2	0	0
2	1	2	0	0	1	2	0	0
4	4	3	0	0	4	3	0	0
1	1	1	0	0	1	1	0	0

Algorithm: CSP-R step-by-step iter. 14

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

x : ␣G␣G d : 4 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	**G**	G	T	A	**C**	C	G
4	4	**3**	0	0	4	**3**	0	0
1	2	**2**	0	0	1	**2**	0	0
2	1	**2**	0	0	1	**2**	0	0
4	4	**3**	0	0	4	**3**	0	0
1	1	**1**	0	0	1	**1**	0	0

Algorithm: CSP-R step-by-step iter. 15

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

x : GG␣G d : 3 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	**G**	G	T	A	**C**	C	G
4	4	**3**	0	0	4	**3**	0	0
1	2	**2**	0	0	1	**2**	0	0
2	1	**2**	0	0	1	**2**	0	0
4	4	**3**	0	0	4	**3**	0	0
1	1	**1**	0	0	1	**1**	0	0

Algorithm: CSP-R step-by-step iter. 16

Function CSP-R($\mathcal{S}, k, m, x$)
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R($\mathcal{S}', k-1, m, \mathcal{S}''[i]$)
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue($\mathcal{S}, k, m, x$)
- **return** x[.]

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$x:$ GG␣G $d:$ 3 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	0	0	0	3	2	0	0
0	0	0	0	0	1	2	0	0
0	0	0	0	0	1	2	0	0
0	0	0	0	0	3	2	0	0
0	0	0	0	0	1	3	0	0

Algorithm: CSP-R step-by-step iter. 17

Function CSP-R$(\mathcal{S}, k, m, x)$
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$

Input instance

$$S = \begin{cases} s^1 : GTCC \\ s^2 : AGAG \\ s^3 : CGAG \end{cases}$$

$x:$ GGCG $d:$ 2 2 2

3-Dimensional matrix H

1			2		3		4	
A	C	G	G	T	A	C	C	G
0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	1	2	1	2	1	1	2

3-Dimensional matrix T

1			2		3		4	
A	C	G	G	T	A	**C**	C	G
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	1	**2**	0	0
0	0	0	0	0	3	**2**	0	0
0	0	0	0	0	1	**3**	0	0

Algorithm: CSP-R step-by-step iter. 18

Function CSP-R$(\mathcal{S}, k, m, x)$
- **if** $k = 2$ **then**
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - **if** $\mathcal{S}[1][j] = \mathcal{S}[2][j]$ **then** $x[j] = \mathcal{S}[1][j]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **else**
  - **for** $i \leftarrow 1, \ldots, k$ **do**
    - iCount $\leftarrow 1$
    - **for** $j \leftarrow 1, \ldots, k$ **do**
      - **if** $i \neq j$ **then**
        - copy($\mathcal{S}'$[iCount], $\mathcal{S}[j]$);
        - iCount $\leftarrow$ iCount+1
    - $\mathcal{S}''[i] \leftarrow$ CSP-R$(\mathcal{S}', k-1, m, \mathcal{S}''[i])$
  - **for** $j \leftarrow 1, \ldots, m$ **do**
    - $Q \leftarrow \emptyset$
    - **for** $i \leftarrow 1, \ldots, k$ **do** $Q \leftarrow Q \cup \mathcal{S}''[i][j]$;
    - **if** $|Q| = 1$ **then** $x[i] = Q[1]$;
  - $x \leftarrow$ minimizeMaximumValue$(\mathcal{S}, k, m, x)$
- **return** $x[.]$