TU WIEN Informatics

Solving a Generalized Constrained Longest Common Subsequence Problem

Exact and Heuristic Methods

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christoph Berger
Matrikelnummer 01129111

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Marko Djukanovic, MSc

Wien, 21. Mai 2020

Christoph Berger

Günther Raidl

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

TU WIEN Informatics

Solving a Generalized Constrained Longest Common Subsequence Problem

Exact and Heuristic Methods

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christoph Berger
Registration Number 01129111

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Marko Djukanovic, MSc

Vienna, 21st May, 2020

Christoph Berger

Günther Raidl

Erklärung zur Verfassung der Arbeit

Christoph Berger

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

Wien, 21. Mai 2020

Christoph Berger

Acknowledgements

I would like to thank my supervisor Günther Raidl for giving me the opportunity to write this thesis. I am grateful for his valuable advice and feedback. Likewise, I want to express my sincere gratitude to Marko Djukanovic for his excellent professional guidance and support at all times. He was always available for questions and his input helped to significantly improve the quality of this work. Furthermore, I would like to thank Hannah Perkonigg for her help with proofreading.

Last but not least, I want to thank my friends and family who have given me unfailing support and encouragement throughout my years of study. Heartfelt thanks for keeping me going and making this accomplishment possible.

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Constrained Longest Common Subsequence (CLCS) Problem, welches eingeführt wurde, um die Ähnlichkeit verschiedener biologischer Sequenzen zu messen. Dabei wird für eine gegebene Menge von beliebigen Strings die längste gemeinsame Teilfolge an Zeichen gesucht, welche selbst wiederum einen bestimmten gegebenen String als Teilfolge enthält. Es handelt sich um eine Variante des gut untersuchten Longest Common Subsequence (LCS) Problems. Verschiedene Algorithmen sind bekannt um das CLCS Problem für genau zwei Strings (2–CLCS) zu lösen, das allgemeine m–CLCS Problem mit einer beliebigen Menge von Strings wurde bisher jedoch nur näherungsweise mit einem Approximationsverfahren gelöst. Das m–CLCS Problem kann in der Biologie der Identifikation von Molekülgruppen dienen, deren Moleküle eine Gemeinsamkeit in Form einer bestimmten vorhandenen Teilstruktur aufweisen.

In dieser Arbeit werden mehrere neue Methoden vorgestellt um das m–CLCS Problem effektiv zu lösen. Wir präsentieren eine heuristische Beam Search in der Form eines generellen Suchframeworks sowie einen exakten A*-Algorithmus. Außerdem wird eine Greedy Heuristic vorgestellt, die es ermöglicht, Lösungen von akzeptabler Qualität in kurzer Zeit zu finden.

Die Ergebnisse unserer Tests zeigen, dass unsere A*-Suche, geführt von bekannten Upper Bounds des LCS Problems, signifikant schneller im Lösen von 2–CLCS Instanzen als bisherige Algorithmen ist. Beim m–CLCS Problem mit mehreren Strings konnten von der A*-Suche kleine bis mittelgroße Instanzen gelöst werden. Für jene Instanzen, die von A* nicht gelöst werden können, schlagen wir den Einsatz von Beam Search vor. Zur Führung der Beam Search haben sich eine auf Wahrscheinlichkeitstheorie basierenden Heuristik, sowie eine Heuristik zur Berechnung der erwarteten Länge als besonders effektiv erwiesen. Diese Beam Search Konfigurationen konnten bei fast allen von der A*-Suche exakt gelösten Instanzen ebenfalls optimale Lösungen finden und waren dabei signifikant schneller als die A*-Suche. Das vorgestellte Suchframework kann fernerhin auf einfache Art für weitere Varianten des CLCS Problems adaptiert werden, beispielsweise für das (k, m)–CLCS Problem, bei dem eine beliebige Anzahl $k \in \mathbb{N}$ an Strings als notwendige Teilfolge der Lösung spezifiziert wird.

Abstract

In this thesis we are studying the constrained longest common subsequence (CLCS) problem that has been introduced as a specific measure of similarity of biological sequences. It extends the well-studied problem of finding a longest common subsequence (LCS) of a given set of strings by an additional pattern string that is required to be a subsequence of the LCS. There are several algorithms introduced in the literature dealing with the CLCS problem with exactly two input strings (2–CLCS), but the general m–CLCS problem with an arbitrary set of strings has not yet been approached except by one approximation algorithm. The m–CLCS problem may find its application in biology for discovering molecular clusters composed of molecules that all share a common structural pattern.

In this work we propose several new approaches to effectively solve the m–CLCS problem. We present a heuristic beam search in shape of a general search framework as well as an exact A* search algorithm. Moreover, a greedy heuristic to find CLCS solutions of reasonable quality within short time is proposed.

Our experimental evaluation has proven that our A* search guided by a tight upper bound calculation is significantly faster than current state-of-the-art algorithms in finding proven optimal solutions on various 2–CLCS problem instances. Moreover, for the general m–CLCS problem, our A* approach was able to solve small to medium instances to proven optimality within the allowed time and memory limit. For those instances where A* cannot prove optimality, we propose a heuristic beam search. Two beam search configurations, one guided by a probability based heuristic and another one guided by an expected-length calculation heuristic, specially adapted for the m–CLCS problem, have been shown as particularly efficient. They deliver solutions that almost all reach the quality of the optimal solutions proven by A* search within significantly less time. Moreover, the proposed search framework provides a solid basis for extensions towards more general variants of the CLCS problem like the (k, m)–CLCS problem, where instead of one, we are given an arbitrary number of $k \in \mathbb{N}$ pattern strings constraining the LCS.

Contents

CHAPTER 1

Introduction

Strings are objects commonly used for modeling `DNA` or `RNA` molecules. Finding similarities between molecular structures plays an important role in understanding biological processes that relate to the molecular structures. Such similarities can be properly expressed by the length of *subsequences* common for a given set of input strings. A subsequence of string s is any sequence of characters obtained by deleting zero or more characters from s. In particular, a widely used measure of similarity is provided by the *longest common subsequence* (LCS) problem [51] which is a well-known discrete optimization problem: given an arbitrary set of strings, we seek for a longest possible subsequence that is common for all input strings. The LCS problem has many applications not only in molecular biology [44] but also in data compression [60], pattern recognition, file plagiarism check, text editing [47] and others.

There are many well-studied variants of the LCS problem that have arisen from practice. Prominent examples include the *repetition-free longest common subsequence* (RFLCS) problem [2], the *longest arc-preserving common subsequence* (LAPCS) problem [45], and the *longest common palindromic subsequence* (LCPS) problem [18]. In this project we are interested in solving *the constrained longest common subsequence* (CLCS) problem [63, 3]: given two input strings s_1, s_2, and a pattern P, we seek for an LCS between the two input strings that has also string P as its subsequence. Figure 1.1 illustrates the problem with a small example. A possible application scenario of the CLCS problem concerns the identification of homology between two biological sequences which have a specific or putative structure in common [63]. Studying genomes of various species has shown that some segments are constrained in the lineage. It is estimated that roughly 8% of the human genome consists of sequences that are conserved in other eutherian mammals [58]. A higher proportion of sequences conserved reflects a lower divergence between species. In general, the length of a CLCS can be used as similarity measurement for molecules while taking a common specific segment that arises from some structural properties into account. A concrete example is described in [16]. It deals with the comparison of seven RNase sequences so that the three active-site residues, HKH, form part of the solution[1]. This pattern is responsible, in essence, for the main functionality of the RNase molecules such as catalyzing the degradation of RNA sequences. Furthermore, constrained sequences find application also in other areas, as for instance, in communication or magnetic recording [17].

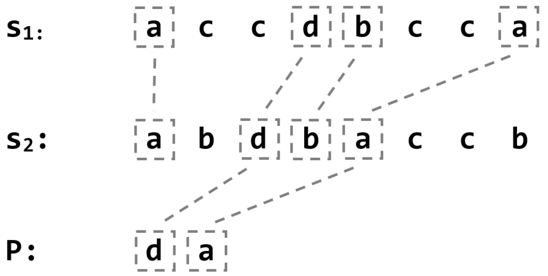


Figure 1.1: Example of a small CLCS problem instance with input strings $s_1 = \texttt{accdbcca}$, $s_2 = \texttt{abdbaccb}$, and pattern string $P = \texttt{da}$. The string adba is the longest common subsequence of s_1 and s_2 that has P as its subsequence. Thus, it is a CLCS for this instance.

The classical CLCS problem can be solved efficiently by applying a Dynamic Programming approach [22, 15]. It was shown that the problem is a special case of the constrained sequence alignment (CSA) problem [15]. Aligning multiple sequences simultaneously finds application in many areas, e.g. in studying gene regulation or inferring evolutionary relationship of genes or proteins [12]. Therefore, it is clearly important to consider also a general variant of the CLCS problem with more than two strings in input [33] (m–CLCS), which is $\mathcal{NP}$–hard. To the best of our knowledge, no work has been proposed in the literature for solving the m–CLCS problem except one existing linear time approximation algorithm developed by Gotthilf et al. [33].

1.1 Aim of this Work

Our primary goal with this research project is to effectively solve the m-CLCS problem. A literature review shows that for the related m-LCS problem, exact solving methods are quite restricted, only applicable for instances up to a certain size whereas middle-to-large instances are dealt via approximation or heuristic search. Beam search is considered as state-of-the-art for heuristic solving. Thus, to tackle the m-CLCS problem, an A* search is developed to provide an exact solving method and a beam search framework is created to approach large instances heuristically. Various heuristics and upper bounds are derived to evaluate nodes and guide the search process. The performances of the

[1] National Center of Biotechnology Information database, at http://www.ncbi.nlm.nih.gov.

developed algorithms are compared to the proposed (approximate) method from the literature.

Since the 2–CLCS problem is well–studied in the literature where many specialized exact approaches are proposed, we also compare our A* to some of them. As it turns out, no comprehensive comparison on instances of larger size has been made so far between current state-of-the-art approaches for the 2–CLCS problem. Thus, by conducting our exhaustive experimental studies on various artificial benchmarks as well as on a benchmark suite from the CLCS literature containing real biological sequences, we aim to provide more insights on that matter.

1.2 Structure of this Work

In this chapter we gave a problem description, an introduction to the problem's context and expressed the primary goals of this work. The remaining chapters are organized as follows. In Chapter 2 we start with an overview of the existing literature related to the CLCS–problem. Chapter 3 provides an overview of discrete optimization and describes the methodologies used in this thesis. In Chapter 4 we introduce preprocessing structures and propose a greedy heuristic procedure to quickly derive CLCS solutions of reasonable quality. In Chapter 5 the search framework for the m–CLCS problem is defined. The state graph is derived and various upper bound estimators and heuristics to evaluate CLCS subproblems are presented. In Chapter 6 a general beam search framework for the m–CLCS problem is derived and in Chapter 7 an A* search is proposed. Chapter 8 describes the main ideas of the algorithms from the literature used in our comparisons where also an ILP model for the 2–CLCS problem is proposed. Experimental studies for both, the classical 2–CLCS problem and the general m–CLCS problem, are presented by Chapter 9. In Chapter 10, conclusions and some directions for future work are outlined.

1.3 Preliminaries

Let us introduce essential terms and notations commonly used in this thesis. A *string* or *sequence* is composed of a finite amount of characters over an alphabet Σ. The length of a string s is denoted by $|s|$. The j-th letter of a string s is referred by $s[j]$, $j = 1, \dots, |s|$, and if $j > |s|$ then $s[j] = \varepsilon$, where ε denotes the empty string. Let $s_1 \cdot s_2$ denote the concatenation obtained by appending string s_2 to the end of string s_1. By $s[j, j']$, $j \leq j'$, we denote the continuous subsequence of s starting at position j and ending at position j'; if $j > j'$, then $s[j, j'] = \varepsilon$. Furthermore, we say, a *subsequence* of string s is any sequence of characters obtained by deleting zero or more characters from s. Finally, let $|s|_c$ be the number of occurrences of letter $c \in \Sigma$ in string s.

We formally define the m–CLCS problem as follows. We are given a set $S = \{s_1, \dots, s_m\}$ of $m \in \mathbb{N}$ non-empty input strings over Σ, and a so–called pattern string P over Σ. Henceforth, we refer to a CLCS problem instance by $I = (S, P, \Sigma)$ and denote the maximum length of the strings in S by n. The m–CLCS problem asks for a string s that is a subsequence of every input string and contains P as a subsequence. Such a string s is called a *valid solution* of I. Moreover, if s is of maximal length (w.r.t. all possible valid solutions of I), it is called an *optimal solution* or, simply, *CLCS*.

CHAPTER 2

Related Work

Most of the work related to the CLCS problem only considers two input strings. In order to efficiently solve the CLCS problem with more than two strings, it makes sense to explore state-of-the-art approaches that are applied in solving the well-studied, basic LCS problem. This is done in the following section. Afterwards, a literature overview for the CLCS problem is given.

2.1 LCS Problem: Literature Overview

The problem of finding a LCS of two input strings is polynomially solved by means of dynamic programming (DP) [34]. The DP-based algorithm proposed from Masek and Paterson [52] runs in $O(n^2/\log n)$ time where n is the length of the input strings. For an arbitrary number of input strings, the problem was shown to be $\mathcal{NP}$-hard [51, 1]. Various techniques are applied to solve the LCS problem to optimality, they include more advanced dynamic programming methods [37, 30], integer programming [8] and tree search [38, 28]. The first tree search method from Hsu and Du [38] systemically enumerates solutions for the LCS problem. Later, Easton and Singireddy [28] proposed a more efficient approach that applies new branching techniques and reduces the search space by eliminating branches that were proven to not contain an optimal solution. Finally, an algorithm based on A* search was proposed by Wang et al. [67]. Due to the complexity of the problem, exact algorithms become impractical for instances of large size. Thus, approximation or heuristic algorithms are applied to obtain good non-optimal solutions in reasonable time.

The first approximation proposed for the LCS problem was the Long Run (LR) algorithm [14, 43]. Its provided solutions are sequences composed of only one single letter and are within a factor of alphabet size $|\Sigma|$ of the optimal solution. Other approximation algorithms without the restriction to a single letter have been proposed later, including the Expansion algorithm by Bonizzoni et al. [10], the Best–Next for Maximal Available

Symbols (BNMAS) and Enhanced Long Run (ELR) by Huang et al. [39]. Just like the LR algorithm, they all guarantee $|\Sigma|$-Approximation, but they typically provide a better quality of solutions.

Heuristic approaches usually don't provide any performance guarantees but are often able to yield near-optimal solutions in affordable time. Easton and Singireddy [29] showed that their Time Horizon Specialized Branching Heuristics (THSB), which is based on a large neighborhood search, obtained better results than both, LR and Expansion algorithm. Shyu and Tsai [59] presented an Ant Colony Optimization (ACO) algorithm resulting in better solution qualities than BNMAS and Expansion algorithm. Blum et al. [7] then proposed a beam search (BS) approach on the LCS problem. The algorithm performs an incomplete tree search and was shown to dominate both THSB and ACO algorithm in terms of solution quality, establishing a new state-of-the-art for the LCS problem. Subsequently, many further approaches incorporating BS were introduced, e.g., [6, 49, 54, 61]. Tabataba and Mousavi presented a BS algorithm guided by a novel probabilistic heuristic [54] and a specific mechanism of reducing suboptimal solutions. The same authors extended their approach by proposing a hyper-heuristic algorithm (HH–BS) incorporated within a beam search [61]. The HH–BS was able to improve the results on all five benchmark sets used in [7]. Most recently, a heuristic estimating the expected length of an LCS was proposed by Djukanovic et al. [25]. In their experiments, BS guided by the novel heuristic steered the search better towards more promising regions. The results obtained were on many instance sets significantly better than previous approaches from literature.

2.2 CLCS Problem: Literature Overview

The classical CLCS problem with two input strings s_1 and s_2 and a pattern string P was initially introduced by Tsai [63] where the first algorithm to solve the problem by dynamic programming in $O(|s_1|^2 \cdot |s_2|^2 \cdot |P|)$ time was given. It was not of practical relevance due to its high complexity. Many efficient algorithms have been proposed since then, we give a short overview below and explain the most important ones with more details in Chapter 8.

An improvement over the algorithm from Tsai was first achieved by Chin et al. [15], creating a simple DP recursive relation to compute a CLCS, requiring only $O(|s_1|\cdot|s_2|\cdot|P|)$ time and space. Arslan and Eğecioğlu [3] proposed another DP-based algorithm to solve the CLCS problem within the same time complexity. While their approach requires more space than the algorithm of Chin et al., they also provided a procedure to solve a variant of the CLCS problem where the goal is to find an LCS for s_1, s_2 and a sequence whose edit distance from pattern P is less than a positive integer that is given in advance.

The first sparse approach was developed by Deorowicz [21]; it solves the CLCS problem in $O(|P| \cdot (|s_1| \cdot L + R) + |s_2|)$ time where L is the length of an LCS between s_1 and s_2 and R is the number of pairs of matching positions between s_1 and s_2. Later on, an improvement of the algorithm to further reduce computations that don't contribute to a final CLCS was introduced by Deorowicz and Obstoj [22]. Iliopoulos and Rahman [42] proposed another sparse algorithm to solve the CLCS problem in $O(|P| \cdot R \cdot \log\log|s_1| + |s_2|)$ time. To achieve such a time complexity, they utilize a BoundedHeap data structure [11] realized by means of Van Emde Boas (vEB) trees [65]. Moreover, an algorithm based on a finite automata designed to solve the CLCS problem for degenerate strings was proposed by Iliopoulos et al. [41]. Most recently, an algorithm especially suited for input strings that are highly similar was presented by Hung et al. [40]. It runs in $O(|P| \cdot L \cdot (|s_1| - L))$ time where L denotes the length of a CLCS. According to the authors, their approach was able to outperform other algorithms for the classical CLCS problem. Since they only performed experiments on a rather limited benchmark sets where $|\Sigma| = 256$ was fixed, it remains to check the algorithm's performance on wider classes of benchmark sets. For this purpose, we generate such benchmark sets in the course of our computational studies.

To the best of our knowledge, the approximation algorithm by Gotthilf et al. [33] is the only existing work towards solving the general m–CLCS problem with arbitrary number of input strings m. The algorithm is based on the idea that pattern string P obviously must be a part of any CLCS solution. For each letter of the pattern string, possible mappings in the strings $s_1, s_2, \ldots, s_m$ are assigned. The algorithm then examines the intermediate strings resulting from taking the part in between first and last mapping for each two adjacent letters in the pattern string. An LCS is approximated for these intermediate strings by checking which repetition of a letter yields the largest length. The pattern string with the approximated LCS inserted between the two corresponding letters clearly represents a feasible solution for the CLCS problem. The best found approximated CLCS solution is returned in the end. The algorithm provenly runs in linear time and yields an approximation factor of $\sqrt{l_{min} \cdot |\Sigma|}$, where l_{min} is the length of the shortest input string and $|\Sigma|$ is alphabet size.

CHAPTER 3

Methodology

In this chapter we provide theoretical foundations and basic concepts upon which our algorithms are based. The literature usually distinguishes between two distinct categories of optimization problems. In *continuous optimization problems*, the goal is find a set of real numbers or even functions, while in *discrete optimization problems* (also called *combinatorial problems*), the possibilities are limited to a finite set (discrete points) [57]. In this work, we only focus on the latter. We first formalize the idea of an optimization problem, provide basic terminology and then introduce concepts for solving such a problem.

We mainly follow the definition from Papadimitriou and Steiglitz [57] and introduce an *instance of an optimization problem* as a pair (F, c), where search space F is any set, the domain of feasible elements (*feasible solutions*), and c is the cost function defining a mapping $c : F \longrightarrow \mathbb{R}$. The goal is to find an element $f \in F$ for which

$$c(f) \leq c(f'), \text{ for all } f' \in F.$$

Such an element f presents a *globally optimal solution* to the given instance and is henceforth simply called *optimal solution.* Note that while an element $f \in F$ that minimizes $c(f)$ is searched, any maximization problem can trivially be converted to a minimization problem by multiplying its cost function by minus one. Hence, the concepts presented in this chapter can be applied to both minimization and maximization problems.

A distinction needs to be made between *problem* and *instance of a problem*: an optimization problem is defined as a set of instances [57]. In this way, a problem describes an issue in a general form while an instance is provided with specific input data and asks for a concrete solution.

A naive procedure for solving optimization problems can be constructed as follows. Enumerate all feasible solutions $f \in F$, check on each the value for $c(f)$ and remember the best solution. This works in principle, however, most interesting problems will have instances with such a large set of feasible solutions so that enumerating all of them will not be a viable option. In order to avoid full enumeration, we require a way of proving optimality without checking every single feasible solution. A basic method for doing so is known from integer programming (cf. [68]) and can be provided by the bounds of the instance. For an instance (F, c) of a minimization problem with optimal solution f^*, a value $p \in \mathbb{R}$ is called a *primal bound (upper bound)* iff $c(f^*) \leq p$. Straightforwardly, a value $d \in \mathbb{R}$ is called a *dual bound (lower bound)* iff $c(f^*) \geq d$. When we can find a primal bound p and dual bound d such that $p = d = c(f)$, $f \in F$ then f is provenly an optimal solution. Every feasible solution $f \in F$ provides a primal bound. Dual bounds on the other hand, can be obtained by heuristics. Note that in context of a maximization problem, the primal bound is a lower bound and the dual bound is an upper bound.

An optimization problem may be solved by an *exact method* or a *heuristic method.* Exact methods guarantee optimality on found solutions, but their application might not always be feasible. Heuristic methods (also called *approximate methods*) compute solutions in affordable time but are not guaranteed to find an optimal solution.

3.1 Branch-and-Bound

A common exact method for solving optimization problems is *branch-and-bound* (B&B), which is a technique of intelligently enumerating feasible solutions by making use of upper bounds / lower bounds. Its basic idea was already discussed about 60 years ago in the context of mathematical programming [48]. It applies the principle of the *divide and conquer* technique by breaking down the original problem into smaller subproblems. The B&B procedure builds a decision tree where each node represents a set of solutions. It goes through the search space by continuously performing the two main operations of (1) *branching* and (2) *bounding. Branching* splits a set of solutions, represented by a node, into multiple mutually exclusive subsets. Each of the subsets is then represented by a new child node. *Bounding* refers to the bounds that are used to prove optimality without the need to look at all feasible solutions. A global upper bound is maintained by always storing the so far best solution value. Before the branching of a node is performed, a lower bound is calculated for that node. If this lower bound is less than the global upper bound, then the node can't contain an optimal solution for the original (undivided) problem instance. Hence, it does not need to be considered anymore and its set of solutions can be discarded.

The B&B procedure will yield an optimal solution if it is allowed to run until all feasible solutions are either checked or discarded. In practice though, it makes sense to consider the case that the procedure might need to be terminated earlier and return the best found solution in such a case. A pseudocode of a B&B implementation for a minimization problem is provided in Algorithm 3.1.

A second way to eliminate nodes from the decision tree, apart from comparing its lower bound to the global upper bound, can be established by testing *dominance relations.* If

Algorithm 3.1 Branch-and-Bound (w.r.t. minimization)

```
Input: a problem instance (F,c)
Output: an optimal solution $v_{best}$
Initialize: $U \leftarrow \infty$, set of active nodes $A = \{F\}$
while $A \neq \emptyset$ do
    choose a node $v \in A$ for branching
    remove $v$ from $A$
    if lower bound of $v < U$ then
        split $v$ into subsets $v_1, \ldots, v_k$
        for each $v_i \in \{v_1, \ldots, v_k\}$ do
            if $v_i$ is a complete solution then
                if $c(v_i) < U$ then
                    $U \leftarrow c(v_i)$
                    $v_{best} \leftarrow v_i$
                end if
            else
                add $v_i$ to $A$
            end if
        end for
    end if
end while
return $v_{best}$
```

the best descendant of a node v_1 is at least as good as the best descendant of v_2, then we say v_1 *dominates* v_2, and v_2 can be discarded [57]. The existence of such relations as well as procedures for efficiently checking them depends on the individual problem.

Several decisions need to be made when implementing B&B for a specific optimization problem. First, there is the question on how the branching should be done, i.e., how the set of feasible solutions will be split into smaller sets. Also, the selection mechanism to decide which node is chosen for branching can have a big impact on the performance. Choices include first-in-first-out, last-in-first-out, lowest bound, or a priority system based on some problem-specific criteria. Furthermore, a lower bound calculation has to be devised and possibilities for dominance testing should be checked. Since there is no generally valid strategy, available options need to be evaluated for each problem individually.

In order to solve the LCS problem, Easton and Singireddy [28] applied the concept of B&B as follows. The root node of the search space corresponds to a partial solution with empty string ϵ. Then, branches are consecutively generated by creating a child node for each possible letter by appending the letter to the partial solution of its parent node. This way, each node represents the set of solutions that start with the node's partial solution. For each node, an upper bound is calculated from the length of its partial

Algorithm 3.2 Dynamic Programming for the LCS Problem

```
Input: strings $s_1$ and $s_2$
Output: matrix $M$ with results for all stages
Initialize: $M[i,0] \leftarrow 0, \forall i = 0, \ldots, |s_1|$ and $M[0,j] \leftarrow 0, \forall j = 0, \ldots, |s_2|$
for $i \leftarrow 1$ to $|s_1|$ do
    for $j \leftarrow 1$ to $|s_2|$ do
        if $s_1[i] = s_2[j]$ then
            $M[i,j] \leftarrow M[i-1, j-1] + 1$
        else
            $M[i,j] \leftarrow \max\{M[i,j-1], M[i-1,j]\}$
        end if
    end for
end for
return $M$
```

solution and the maximal number of letters that could potentially be further appended. If a node's upper bounds is smaller than the length of the so far best solution, it is discarded. We won't go into the details of the upper bound calculation here but refer to Section 5.2 where we present common upper bounds for the LCS problem.

3.2 Dynamic Programming

Similar to B&B, Dynamic Programming (DP) also breaks a problem into smaller parts and obtains a solution by solving these parts. For DP, this process is done in a specific manner. The main idea is to break a problem recursively into smaller stages in a way that the results of each stage can be computed from intermediate results of a previous stage. Intermediate results are stored in a table whereby performing the same computation more than once is avoided. Thus, DP can be particularly effective for problems where different solutions are often composed of identical partial solutions. More details and typical methods of DP can be found in [19].

To give a simple example of DP, we present in Algorithm 3.2 a procedure described by Wagner and Fischer [66] for calculating the length of an LCS of two strings s_1 and s_2. A two-dimensional matrix M is computed such that for any $0 \leq i \leq |s_1|$, $0 \leq j \leq |s_2|$, entry $M[i,j]$ corresponds to the length of an LCS of $s_1[1,i]$ and $s_2[1,j]$. Observe that after initialization of the border cases, the algorithm starts from index 1 and calculates each further value from a previous column/row. Finally, the length of an LCS of s_1 and s_2 can be found in $M[|s_1|, |s_2|]$.

Algorithm 3.3 General A* Search

```
Input: a weighted graph $G = (V, A)$, a start node $s$, a goal node $t$
Output: a smallest-cost path from $s$ to $t$
Initialize: open list $Q \leftarrow \{s\}$
while $Q \neq \emptyset$ do
    remove node $v$ with minimal $f(v) = g(v) + h(v)$ from $Q$
    if $v = t$ then
        return path derived by following pred(v)
    else
        for each successor $v'$ of $v$ do
            $cost_{new} \leftarrow g(v) + c(v, v')$
            if $v'$ reached for first time $\vee$ $cost_{new} < g(v')$ then
                $pred(v') \leftarrow v$
                $g(v') \leftarrow cost_{new}$
                put $v'$ in $Q$    // insert if $v'$ reached for first time, update otherwise
            end if
        end for
    end if
end while
return no path from $s$ to $t$ exists
```

3.3 A* Search

We make use of A* search in order to minimize the amount of solutions that we need to visit to find a proven optimal solution. In this section we describe the principles of the algorithm that was originally developed by Hart et al. [35] to find a smallest-cost path from a start node to a goal node in a weighted graph $G = (V, A)$. It works in a *best-first-search* manner, i.e., the most promising nodes are always considered first. In order to rank the nodes, A* search makes use of an evaluation function $f(v) = g(v) + h(v)$, for $v \in V(G)$, where $g(v)$ denotes the cost of a so-far best path from the start node to v, and $h(v)$ is the estimated cost of an optimal path from v to a goal node.

A* search maintains a list of open nodes, i.e., nodes whose successors have not been explored yet, and stores a set of all nodes encountered during the search. The search procedure begins with only the start node in the open list. Then, at each step, the node v that minimizes function $f(v)$ is taken from the open list. This node is then *expanded* by considering all possible successor nodes as follows. A successor node v' is updated if it has been seen before and a better path from start node to v' has been discovered. If it is the first time that successor node v' is reached, then it is added to the open list. Unless terminated early (e.g. due to limited time or memory resources), A* search stops once it selects a goal node for expansion. A pseudocode of an implementation of an A* search is provided in Algorithm 3.3.

In order to guarantee, that a path found from A* search is indeed a smallest-cost path, $h(v)$ is required to be *admissible*, meaning $h(v) \leq h^*(v)$, $\forall v \in V(G)$, where $h^*(v)$ denotes the cost of the real smallest-cost path from v to a goal node. Moreover, if $c(v, v') + h(v') \geq h(v)$, $\forall (v, v') \in A(G)$, where $c(v, v')$ denotes the cost from v to v', then $h(v)$ satisfies the consistency assumption and is called *monotonic*. A* search with monotonic $h(v)$ never needs to re-expand an already expanded node. It was shown that the number of node expansions required to find a proven optimal path by A* search with monotonic $h(v)$ is minimal among all (possible) search algorithms that use the same heuristic information and tie-breaking criterion [20].

3.4 Heuristic Methods

Many optimization problems from practice are of complex nature and too difficult to be solved by exact methods within an acceptable amount of time or memory. Not all applications require an optimal solution though, finding a "reasonably good" solution might often be sufficient. To this end, heuristic methods are applied. In this section, we present two basic heuristic methods, namely *constructive heuristics* and *local search*, and then give an introduction about *metaheuristics*. The following review of heuristic methods is based on Blum and Raidl [9] and Talbi [62].

Constructive Heuristics

A constructive heuristic starts from an empty (partial) solution and iteratively adds parts until the solution is complete. Parts that are once added to the solution are usually never replaced or removed, i.e., all decisions made in course of the procedure are final. A prominent variant of constructive heuristics are *greedy heuristics*. At each construction step, they evaluate all available options to extend the current partial solution and then choose the one that seems best from a local point of view. A well-known greedy heuristic for the LCS problem is the so-called *Best–Next* heuristic [39]. It starts with an empty solution string and adds at each step the most promising letter until no more letters can be added.

Local Search

Local search is a heuristic method that does not start from scratch but is given some initial solution with the goal to improve it. To this end, solutions from a so called *neighborhood* are explored. A neighborhood is established for an instance (F, c) of an optimization problem by assigning each solution $f \in F$ a set of neighbors $N(f) \subseteq F$. A neighbor $f' \in N(f)$ is typically generated by applying a specified set of changes – known as *move* – to the current solution f.

A local search replaces at each step its current solution with a better solution from a neighborhood until no improvement is possible to find. Different strategies exist for selecting a neighbor. The simplest way is called *first improvement* where the first neighbor that improves the current solution is selected in a deterministic way. In the *best* *improvement* strategy, all possible moves are tried and the best found solution is selected. In *random selection* strategy, the solution is chosen by random from all neighbors that improve the current solution.

Metaheuristics

Metaheuristics provide problem-independent approximate methods for optimization problems. Usually, they are applying at least one of the presented basic heuristic methods and incorporate other ideas in form of some higher-level framework. Many such heuristics exist, well-known algorithms include Variable Neighborhood Search [53], Tabu Search [31], Simulated Annealing [46], Ant Colony Optimization (see [5, 27]) and Evolutionary Algorithms (see [4]). Since our heuristic approach for the m–CLCS problem is based on Beam Search, we will introduce this method in detail and refer the interested reader for more information about other metaheuristics to the cited articles or to Blum and Raidl [9] and Talbi [62].

Beam search (BS) can be described as a heuristic derivative of B&B. It is known for its application in the context of scheduling problems (see, e.g., [56, 64]), but it is not restricted to any specific domain. The classic BS procedure performs a heuristic tree search where the search space is traversed in a limited *breadth-first-search* (BFS) manner. At each level of the search tree, only the most promising β nodes (w.r.t. some evaluation function) are kept for further expanding while all remaining ones are discarded; the so called *beam width* β is a parameter of the algorithm and needs to be chosen carefully. Searching with small β risks cutting off branches that might contain good solutions while searching with large β can require a lot of computation effort. Note that BS becomes a pure greedy construction procedure when $\beta = 1$ and BS becomes a full BFS when β is large enough to keep all nodes.

CHAPTER 4

A Fast Heuristic for the m–CLCS Problem

Before we establish our main search framework, we introduce essential preprocessing structures and propose a greedy construction heuristic to quickly derive CLCS solutions of reasonable quality.

4.1 Preprocessing Structures

In order to avoid performing identical computation steps over and over, commonly needed information is computed only once at the beginning and stored for the rest of the process. More specifically, our search framework makes use of the following two data structures established during preprocessing:

- $\mathit{Succ}[i,j,c]$ data structure stores for each string s_i, $1 \leq i \leq m$, for each position $1 \leq j \leq |s_i|$ and for each letter $c \in \Sigma$ the minimal (left-most) position x such that $x \geq j \wedge s_i[x] = c$. If no such position exists, we set $\mathit{Succ}[i,j,c] := |s_i| + 1$.

- $\mathit{Embed}[i,u]$ data structure stores for each string s_i, $1 \leq i \leq m$ and for each position $1 \leq u \leq |P|$ the maximal (right-most) position of s_i such that $P[u,|P|]$ is a subsequence of $s_i[x,|s_i|]$. If no such position exists, we set $\mathit{Embed}[i,u] := -1$.

The *Succ* data structure is derived in $O(m \cdot n \cdot |\Sigma|)$ time by a procedure presented in Algorithm 4.1. The *Embed* data structure is derived in $O(m \cdot |P|)$ time, see Algorithm 4.2.

Algorithm 4.1 Deriving *Succ* data structure

```
Input: alphabet $\Sigma$, $S = \{s_1, \cdots, s_m\}$
Output: $Succ$ data structure
Initialize: $Succ$ as empty structure
for $i \leftarrow 1$ to $m$ do // scan through the input strings
    $len \leftarrow |s_i|$
    for each $c \in \Sigma$ do
        $Succ[i, len, a] \leftarrow |s_i| + 1$ // initialize values for last position
    end for
    $Succ[i, len, s_i[len]] \leftarrow len$
    for $j \leftarrow (len - 1)$ to 1 do // scan string in reverse way
        for each $c \in \Sigma$ do
            if $s_i[j] = a$ then
                $Succ[i, j, a] \leftarrow j$
            else
                $Succ[i, j, a] \leftarrow Succ[i, j + 1, a]$
            end if
        end for
    end for
end for
return $Succ$
```

Algorithm 4.2 Deriving *Embed* data structure

```
Input: $S = \{s_1, \ldots, s_m\}$, pattern string $P$
Output: $Embed$ data structure
Initialize: $Embed[i, u] \leftarrow -1$ for all $i = 1, \ldots, m$, $u = 1, \ldots, |P|$
for $i \leftarrow 1$ to $m$ do // scan through the input strings
    $u \leftarrow |P|$
    for $j \leftarrow |s_i|$ to 1 do // scan string in reverse way
        if $s_i[j] = P[u]$ then
            $Embed[i, u] \leftarrow j$
            $u \leftarrow u - 1$
            if $u < 1$ then
                break
            end if
        end if
    end for
end for
return $Embed$
```

4.2 Greedy Heuristic for the m–CLCS Problem

With this section we aim to provide a very fast mechanism for constructing adequate CLCS solutions. To this end, we develop a greedy heuristic for the m–CLCS problem that incorporates the idea of the *Best–Next* heuristic presented in [39] in context of the m–LCS problem. The basic principle is straightforward: the algorithm starts with an empty string and builds a solution, appending at each construction step the letter that seems most promising at the moment. The procedure stops once no more letters can be added.

Our greedy heuristic for the m–CLCS problem is presented in Algorithm 4.3. The procedure starts by setting up the position vector $p^{\mathrm{L}} = (p^{\mathrm{L}}_1, \ldots, p^{\mathrm{L}}_m) \in \mathbb{N}^m$, referring to the respective pieces of input strings that are relevant to further extend the current greedy solution. We initialize $p^{\mathrm{L}} := (1, \ldots, 1)$ which means that the whole input strings are considered. Moreover, value $u := 0$ which keeps track of the progress of fulfilling the constraint, and solution $s := \varepsilon$ are initialized. Then, at each step, the set of letters Σ_{feas} that can feasibly extend current greedy solution s (by appending one character to the end of s) is determined, ensuring that a final outcome will contain pattern P as subsequence. In more detail, set Σ_{feas} is obtained efficiently by using *Succ* and *Embed* structures, a pseudocode is provided in Algorithm 4.4. Afterwards, every extension possibility, each given by one of the letters from Σ_{feas}, is evaluated by a greedy criterion. The letter that yields the lowest greedy value, denoted by c^*, is then appended to s. Moreover, when extending the current solution, a new subproblem relevant for further extensions of the current greedy solution is determined by updating the position vector p^{L} w.r.t. letter c^* (see line 11 in Algorithm 4.3). Further, value u, which is required to correctly compute Σ_{feas}, is increased by one if $c^* = P[u+1]$ holds. The steps of the procedure are repeated until eventually $\Sigma_{\text{feas}} = \emptyset$, returning a final greedy solution s.

A greedy criterion to evaluate feasible letters $c \in \Sigma_{\text{feas}}$ used to extend the current greedy solution is given by

$$g(p^{\mathrm{L}}, u; c) = pen(u,c) + \sum_{i=1}^{m} \frac{Succ[i, p^{\mathrm{L}}_i, c] - p^{\mathrm{L}}_i + 1}{|s_i| - p^{\mathrm{L}}_i + 1}, c \in \Sigma_{\text{feas}}, \text{ where}$$

$$pen(u,c) = \frac{1}{|P| - u + \mathbf{I}_{P[u+1]=c}}, \text{ and } c^* \longleftarrow \arg\min\nolimits_{c \in \Sigma_{\text{feas}}} g(p^{\mathrm{L}}, u; c). \tag{4.1}$$

As remarked above, those letters with smaller g–values are preferred. Note that the sum in (4.1) evaluates the amount of characters that are skipped (in relation to remaining lengths) from further search when current greedy solution s is extended by a letter $\mathtt{c}$. It is already used in the context of the m–LCS problem and its greedy criterion. In addition to it, a penalty value $pen(u,c)$ contributes to the sum (note that $\mathbf{I}_{P[u+1]=c}$ returns 1 if $P[u+1] = c$ is fulfilled, or 0 otherwise) giving a priority to the letter that matches the next position of pattern P that is not yet included into the current solution s. Hence, we intent more towards an increase of value u. The penalty value is constructed in a way that including pattern P into solution s as soon as possible is slightly preferred from the start and continually more priority is given to it when approaching to the end of the procedure where it remains to embed a shorter suffix of P into s.

Algorithm 4.3 Greedy Procedure for the m-CLCS Problem

```
Input: a problem instance $(S = \{s_1, \ldots, s_m\}, P, \Sigma)$
Output: a heuristic solution $s$
$p_i^{\mathrm{L}} \leftarrow 1$, $i = 1, \ldots, m$
$u \leftarrow 0$
$s \leftarrow \varepsilon$
$\Sigma_{\text{feas}} \leftarrow \textsc{Feasible}(p^{\mathrm{L}}, u)$
while $\Sigma_{\text{feas}} \neq \emptyset$ do
    $c^* \leftarrow \arg\min_{c \in \Sigma_{\text{feas}}} \{g(p^{\mathrm{L}}, u; c) \mid c \in \Sigma_{\text{feas}})\}$
    $s \leftarrow s \cdot c^*$
    for $i \leftarrow 1$ to $m$ do
        $p_i^{\mathrm{L}} \leftarrow Succ[i, p_i^{\mathrm{L}}, c^*] + 1$
    end for
    if $P[u+1] = c^*$ then
        $u \leftarrow u + 1$ // consider next letter in $P$
    end if
    $\Sigma_{\text{feas}} \leftarrow \textsc{Feasible}(p^{\mathrm{L}}, u)$
end while
return $s$
```

Algorithm 4.4 FEASIBLE Procedure

```
Input: a problem instance $(S = \{s_1, \ldots, s_m\}, P, \Sigma)$, position vector $p^{\mathrm{L}}$, length $u$
Output: set of feasible letters $\Sigma_{\text{feas}}$
Initialize: $\Sigma_{\text{feas}} \leftarrow \emptyset$
for each $c \in \Sigma$ do
    $p_u \leftarrow u$
    if $c = P[u]$ then // a strong match
        $p_u \leftarrow u + 1$
    end if
    $feasible \leftarrow true$;
    for $i \leftarrow 1$ to $m$ do // scan through the input strings
        if $Succ[i, p_i^{\mathrm{L}}, c] > |s_i| \vee (p_u \neq |P| \wedge Succ[i, p_i^{\mathrm{L}}, c] > Embed[i, p_u + 1])$ then
            $feasible \leftarrow false$;
            break
        end if
    end for
    if $feasible$ then
        add $c$ to $\Sigma_{\text{feas}}$
    end if
end for
return $\Sigma_{\text{feas}}$
```

CHAPTER 5

Search Space for the m–CLCS Problem

In order to set up a general search framework for the m–CLCS problem, we first construct the search space of the problem. We first derive the state graph for the m–CLCS problem in Section 5.1. This state graph builds the foundation for later presented advanced search techniques. Our beam search framework (Chapter 6) and our A* search (Chapter 7) both operate on the state graph defined in this chapter.

An important part of any informed search algorithm is an efficient heuristic that evaluates nodes and guides the search process. Thus, we derive several heuristic estimators to assess how promising is to deal with a certain m–CLCS subproblem represented by a node v from the state graph. In Section 5.2 we present upper bounds for the length of the CLCS. These bounds are admissible and monotonic, so that they can be utilized in the A* search. Afterwards, three novel heuristics, inspired by functions that work well in the context of the LCS problem, are proposed for the m–CLCS problem to guide our beam search. More specifically, we present a probability-based heuristic in Section 5.3, a heuristic that estimates the length of a CLCS in Section 5.4, and a heuristic that prefers nodes with larger remaining substrings in Section 5.5.

5.1 State Graph Definition

The state graph for the 2–CLCS problem has been proposed in our report [23]. In essence, we follow the descriptions in the paper and introduce the state graph for the general m–CLCS problem as follows.

Remember that $I = (S, P, \Sigma)$ denotes considered problem instance. Let s be a string over Σ that is a subsequence of all strings from S. Moreover, let p_i^{L} be the position in s_i such that $s_i[1, p_i^{\mathrm{L}} - 1]$ is the minimal string among all strings from $\{s_i[1, x] \mid x = 1, \ldots, p_i^{\mathrm{L}} - 1\}$

that contains s as a subsequence ($i = 1, \ldots, m$). We say that $p^{\mathrm{L}} = (p_1^{\mathrm{L}}, \ldots, p_m^{\mathrm{L}})$ is the *position vector* induced by s. Note that, in this way, s induces a subproblem $I[p^{\mathrm{L}}] = \{s_i[p_i^{\mathrm{L}}, |s_i|] \mid i = 1, \ldots, m\}$, because it can only be extended by potentially adding letters that appear in all the remaining parts of the input strings. In this context, let prefix $P[1, k']$ of pattern string P be the maximal string among all strings $P[1, x], x = 1, \ldots, |P|$ such that $P[1, k']$ is a subsequence of s. We then say that s is a *valid (partial) solution* iff $P[k' + 1, |P|]$ is a subsequence of all the strings in subproblem $I[p^{\mathrm{L}}]$.

The state graph $G = (V, A)$ of our beam search and A* search is a *directed acyclic graph*, which—at any moment—is created on the fly by our algorithms. Each node $v \in V(G)$ stores a triple $(p^{\mathrm{L},v}, l^v, u^v)$, where $p^{\mathrm{L},v}$ is a position vector that induces subproblem $I[p^{\mathrm{L},v}] = \{s_i[p_i^{\mathrm{L},v}, |s_i|] \mid i = 1, \ldots, m\}$, l^v is the length of the currently best known valid partial solution that induces $p^{\mathrm{L},v}$, and u^v is the length of the longest prefix string of pattern string P that is contained as a subsequence in the best known partial solution that induces node v. Moreover, there is an arc $a = (v, v') \in A$ with label $c(a) \in \Sigma$ between two nodes $v = (p^{\mathrm{L},v}, l^v, u^v)$ and $v' = (p^{\mathrm{L},v'}, l^{v'}, u^{v'})$ iff

- $l^v = l^{v'} + 1$ and
- Subproblem $I[p^{\mathrm{L},v'}]$ is induced by the partial solution that is obtained by appending letter $c(a)$ to the partial solution that induces v.

As remarked already above, we are only interested in meaningful partial solutions, and our search algorithms builds the state graph on the fly. In particular, for extending a node v, the outgoing arcs—that is, the letters that may be used to extend partial solutions that induce node v—are determined as follows. First of all, those letters must appear in all strings from $I[p^{\mathrm{L},v}]$; we call this subset of the alphabet *feasible letters*. In order to find the position of the first (left-most) appearance of each feasible letter in the strings from $I[p^{\mathrm{L},v}]$ we make use of a successor data structure determined during preprocessing that allows to retrieve each position in constant time. This position of first appearance of a feasible letter c in string $s_i[p_i^{\mathrm{L},v}, |s_i|]$ is retrieved from $Succ[i, p_i^{\mathrm{L},v}, c]$ for all $i = 1, \ldots, m$. Moreover, a feasible letter should not be taken for extending v in case it is dominated by another feasible letter: We say that a letter c is *dominated* by a letter $c' \neq c$ iff $Succ[i, p_i^{\mathrm{L},v}, c] \geq Succ[i, p_i^{\mathrm{L},v}, c']$ for all $i = 1, \ldots, m$. Note that a dominated letter cannot lead to a better solution than when taking the letter by which it is dominated instead.

Henceforth, we denote the set of feasible and non-dominated letters for extending a node v by $\Sigma_v^{\mathrm{nd}} \subseteq \Sigma$. However, in order to generate only extensions of v that correspond to feasible partial solutions, we additionally have to filter out those extensions that lead to subproblems whose strings do not contain the remaining part of P as a subsequence. These cases are encountered by utilizing *Embed* data structure that is built during preprocessing, see Section 4.1. More specifically, if $u^v \neq |P|$, we check for each letter $c \in \Sigma_v^{\mathrm{nd}}$ if $c \neq P[u^v + 1]$ and $Succ[i, p_i^{\mathrm{L},v}, c] > Embed[i, u^v + 1]$. If that is the case, then letter c cannot be used for extending a partial solution represented by v and consequently it is omitted from Σ_v^{nd}. An extension $v' = (p^{\mathrm{L},v'}, l^{v'}, u^{v'})$ is generated for each remaining letter $c \in \Sigma_v^{\mathrm{nd}}$, where $p_i^{v'} = Succ[i, p_i^v, c] + 1$ for $i = 1, \dots, m$, $l^{v'} = l^v + 1$ and $u^{v'} = u^v + 1$ in case $c = P[u^v + 1]$, respectively $u^{v'} = u^v$ otherwise.

The *root* node of the state graph is defined by $r = (p^{\mathrm{L},r} = \underbrace{(1, \dots, 1)}_{m \text{ times}}, l^r = 0, u^r = 0)$. Any node whose partial solution cannot be further extended, i.e., the node has no outgoing arcs, is called *complete* node. A longest path from the root node to a complete node of the state graph represents an optimal solution of the CLCS problem.

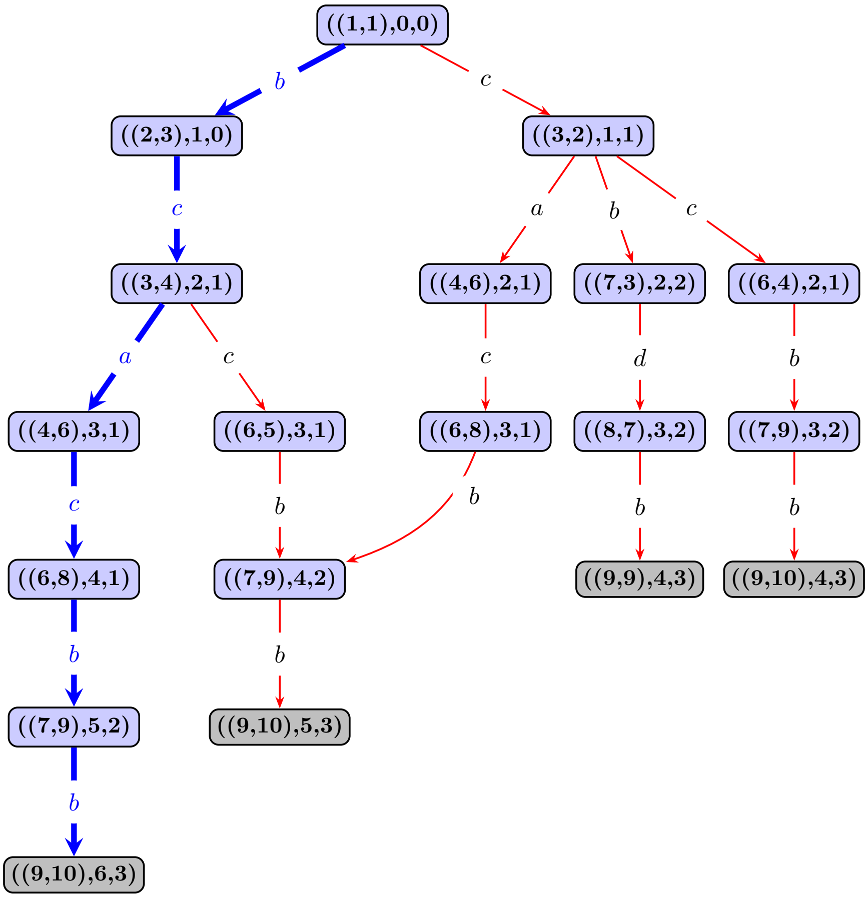


Figure 5.1: Example showing the full state graph for the problem instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). Marked by light-gray color, there are four complete nodes representing non-extensible solutions. The optimal solution is $s = \texttt{bcacbb}$ of length 6 that corresponds to the node $v = (p^{\mathrm{L},v} = (9, 10), l^v = 6, u^v = 3)$. The longest path that corresponds to the optimal solution is displayed in blue.

An example showing the full state graph for a given problem instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$) is shown in Figure 5.1.

The root node, for example, can only be extended by letters b and c, because letters a and d are dominated by the other two letters. Furthermore, note that node $((6,5),3,1)$ (induced by partial solution bcc) can only be extended by letter b. Even though letter d is not dominated by letter b, adding letter d can only lead to infeasible solutions, because any possible solution starting with bccd will have no pattern $P =$ cbb as a subsequence. Finally, the sequence of arc labels on the longest path is bcacbb, which is therefore the (unique) optimal solution to this example problem instance.

5.2 Upper Bounds

Observe that the solutions for a CLCS problem instance with an empty pattern string $P = \epsilon$ are equivalent to the solutions of an LCS problem instance with the same input strings. Moreover, pattern string P only restricts available solutions, i.e., by adding a pattern string P, the length of an solution can decrease but never increase. Thus, the upper bounds developed for LCS subproblems [25] are also valid upper bounds for CLCS subproblems. We present here two reasonably tight, efficiently calculable functions that are known in the literature for the LCS problem.

The upper bound proposed by Blum et al. [7] determines for each letter an upper limit on the number of its occurrences in any optimal solution. Summing up these values for all letters from Σ, a valid upper bound on the length of the optimal solution is obtained by

$$\mathrm{UB}_1(v) = \sum_{a\in\Sigma} \min_{i=1,\ldots,m} \{|s_i[p_i^{\mathrm{L},v}, |s_i|]\,|_a\} \tag{5.1}$$

The bound is efficiently calculated in $O(m \cdot |\Sigma|)$ time by making use of the intelligent data structures, see more details in [26].

A DP–based upper bound, was introduced by Wang et al. [67]. It makes use of the DP recursion for the LCS problem for all pairs of input strings $\{s_i, s_{i+1}\}, i = 1, \ldots, m-1$. More in detail, for $S_i = \{s_i, s_{i+1}\}$, a *scoring matrix* M_i $(i = 1, \ldots, m-1)$ is generated on the basis of a DP–recursion for the classical LCS problem whose entry $M[x, y]$, $x = 1, \ldots, |s_i|+1$, $y = 1, \ldots, |s_{i+1}| + 1$ corresponds to the length of $\mathrm{LCS}(\{s_i[x, |s_i|], s_{i+1}[y, |s_{i+1}|]\})$. Thus, this results in an upper bound

$$\mathrm{UB}_2(v) = \min_{i=1,\ldots,m-1} M_i[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}]. \tag{5.2}$$

Neglecting the preprocessing step for generating the scoring matrices, this bound can be efficiently calculated in $O(m)$ time.

The combination of both presented functions gives an even tighter upper bound for the LCS problem by

$$\mathrm{UB}(v) = \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}. \tag{5.3}$$

A tightening of upper bounds could be done for the CLCS–problem by utilizing a Dynamic Programming approach by Chin et al. [15] (we also refer to the approach by Deorowicz and Obstoj [22], where sparsity is exploited). For each two input strings $s_i, s_j \in S, i \neq j$, a scoring matrix $M_{ij}[x, y, z]$, $1 \leq x \leq |s_1|, 1 \leq y \leq |s_2|$, $1 \leq z \leq |P|$, can be generated; an entry $M_{ij}[x, y, z]$ stores the longest common subsequence of $s_i[x, |s_i|]$ and $s_j[y, |s_j|]$ containing $P[z, |P|]$ as its subsequence.

Preprocessing all the scoring matrices M_{ij} for $1 \leq i \leq j \leq m$, is a resource consuming step (the number of such different matrices is trivially bounded by $O(m^2)$), therefore we set $j := i + 1$ (each two consecutive strings here considered) and we preprocess only matrices $M_i = M_{i,i+1}$, $i = 1, \ldots, m-1$. Then, by following the derivation of UB_2 for the LCS problem [67], the scoring matrices M_i for the CLCS problem are derived and, in that way, an upper bound is generated by

$$\mathrm{UB}^{\mathrm{CLCS}}(v) = \min_{i=1,\ldots,m-1} M_i[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}, u^v]. \tag{5.4}$$

In details, the DP recursion which constructs a scoring matrix $M_i[x, y, z]$ of dimension $(|s_i| + 1) \times (|s_{i+1}| + 1) \times (|P| + 1)$, for each $i \in \{1, \ldots, m-1\}$ is determined by considering different cases that can occur when determining the value of $M_i[x, y, z]$:

- If $s_i[x] = s_{i+1}[y] = P[z]$ (a *strong* matching appears), then the value is equal to $1 + M_i[x + 1, y + 1, z + 1]$;
- If $s_i[x] = s_{i+1}[y]$, but $P[z] \neq s_i[x] \vee P[z] \neq s_{i+1}[y]$ (a *weak* matching appears), then it is equal to $1 + M_i[x + 1, y + 1, z]$;
- If $s_i[x] \neq s_{i+1}[y]$, then it is equal to $\max\{M_i[x + 1, y, z], M_i[x, y + 1, z]\}$.

The initialization is done by setting:

$$M_i(x, |s_{i+1}| + 1, |P| + 1) = M_i(|s_i| + 1, y, |P| + 1) = 0 \wedge \tag{5.5}$$
$$M_i(x, |s_{i+1}| + 1, z) = M_i(|s_i| + 1, y, z) = -\infty, \tag{5.6}$$

for $x = 1, \ldots, |s_i| + 1$, $y = 1, \ldots, |s_{i+1}| + 1$, and $z = 1, \ldots, |P|$. Unluckily, even though those structures establish straightforwardly a stronger upper bound than UB_2, our memory limit was not enough to keep them all in the memory for performing all our experiments by utilizing $\mathrm{UB}^{\mathrm{CLCS}}(\cdot)$ bound; simply, it was applicable only for small instances (with up to $n = 100$, see more about the practical use of this structures in Section 9. Therefore, their practical application is restricted and, thus, they are interesting more from a theoretical point of view.

5.3 Probability–Based Heuristic

Mousavi and Tabatabar [54] derived a DP recursion which determines the probability that any string s of length p is a subsequence of a *random* string s_1 of length q. In this context, a random string means that the characters in s_1 are uniformly distributed. Let us denote the relation *to be a subsequence* by $\prec_{subs.}$. The probability that $s \prec_{subs.} s_1$ can be expressed by a DP matrix $P^{\mathrm{subs}}(p,q)$, $0 \le p \le |s|, 0 \le q \le |s_1|$.

Under the given assumptions, the probability that random string s of length p is a common subsequence of all input strings from S is approximated by means of the $P^{\mathrm{subs}}(\cdot)$ DP table by

$$H(r) = Prob(s \prec_{subs.} S) = \prod_{i=1}^{m} Prob(s \prec_{subs.} s_i) = \prod_{i=1}^{m} P^{\mathrm{subs}}(p, |s_i|).$$

Choosing a proper value for p plays a crucial role in obtaining high-quality solutions. This value is heuristically determined once at each level of a BS run, i.e., once for each corresponding extension set (denoted by V_{ext}, see Chapter 6). In order to obtain a proper value for p in the context of the CLCS problem, we first determine

$$p^{\min} := \min_{v \in V_{\mathrm{ext}}} \left(|P| - u^v\right). \tag{5.7}$$

Note that the characters of $P[p^{\min}+1, |P|]$ are common extensions of any partial solution that relates to a subproblem of node $v \in V_{\mathrm{ext}}$. Thus, we set up the value of p by

$$p = p^{\min} + \min_{v \in V_{\mathrm{ext}}} \left\lfloor \frac{\min_{i=1,\ldots,m}\left\{|s_i| - p_i^{\mathrm{L},v} + 1\right\} - p^{\min}}{|\Sigma|} \right\rfloor. \tag{5.8}$$

In detail, at some moment, the characters $P[p^{\min}+1], \ldots, P[|P|]$ will be encountered as the labels of any path through node $v \in V_{\mathrm{ext}}$ towards a complete node. As these extensions are realized sooner or later, the probabilities that they occur as extensions of the partial solutions of node $v \in V_{\mathrm{ext}}$ are equal to 1. Therefore, the safe extensions will contribute by $p^{\min}$ units to the value of p. We justify the use of the second sum in equation (5.8) (heuristically) by the fact that the length of a CLCS gets smaller by increasing the alphabet size and gets larger as the minimal length among the lengths of the input string in $I[p^{\mathrm{L},v}]$ gets larger. We emphasize that there still might be room for improving the choice of p. If $p = 0$, we set $p = 1$ in order to break ties. Thus, each node $v \in V_{\mathrm{ext}}$ has been evaluated by

$$H(v) = \prod_{i=1}^{m} P^{\mathrm{subs}}(p, |s_i| - p_i^{\mathrm{L},v} + 1), \tag{5.9}$$

where p is determined by (5.8), supposing independence between input strings. The matrix P^{subs} of probabilities is computed in preprocessing. Note that when setting $|P| = u^v = 0$, the H–heuristic developed for the LCS problem [54] is covered by (5.9). Those nodes with a larger H–value are preferred among others.

5.4 Expected Length Calculation Heuristic on Random Strings

In this section we extend the state-of-the-art expected length calculation heuristic on random strings for the LCS problem [25] towards the CLCS problem. The model is established again on random input strings and random pattern P. In order to avoid unnecessary repetitions in the derivation of the heuristic, we build upon the findings of [25]. From there, it is known that

$$\mathbb{E}[Y] = \sum_{k=1}^{l_{\min}} \mathbb{E}[Y_k] \tag{5.10}$$

holds, where Y is a random variable denoting the length of an LCS, Y_k a binary random variable denoting that there is an LCS with a length of at least k and $\mathbb{E}(\cdot)$ is denoting the expectation. In the case of the CLCS problem, a similar formula can be derived by linking corresponding Y and Y_k variables, such that Y_k is a binary random variable denoting that there is an LCS with a length of at least k having pattern P as a subsequence. If we assume the existence of at least one feasible solution (i.e., pattern P is a subsequence of every input string), we get

$$\mathbb{E}[Y] = |P| + \sum_{k=|P|+1}^{l_{\min}} \mathbb{E}[Y_k].$$

For $k \geq |P|$, let T_k be the set of all subsequences of length k over alphabet Σ; there are $|\Sigma|^k$ such subsequences. For each $x \in T_k$, we assign an event Ev_x: x is a subsequence of S having P as its subsequence (obviously, it must be $|x| \geq |P|$). As it was assumed in the case of the LCS problem, independence among events Ev_x and Ev_y, for any $x \neq y, x, y \in T_k$ is assumed. Probability that string $s \in T_k$ is a subsequence of all input strings from S is equal to $\prod_{i=1}^{m} P^{\mathrm{subs}}(k = |s|, |s_i|)$. Further, the probability that random pattern P is a subsequence of string s is equal to $P^{\mathrm{subs}}(|P|, k)$, assuming

- that the distribution of the characters in s is uniform, and
- independence between the probabilities that events $s \prec_{subs.} s_i$ ($\forall i = 1, \ldots, m$) and $P \prec_{subs.} s$ occur.

From there, it follows that the probability that s is a common subsequence of all strings from S that has pattern P as a subsequence is equal to $P^{\mathrm{clcs}}(s, S) = P^{\mathrm{subs}}(|P|, k) \cdot \prod_{i=1}^{m} P^{\mathrm{subs}}(k, |s_i|)$.

Note that, under our assumptions, $Prob\{P \prec_{subs.} y\} = Prob\{P \prec_{subs.} x\} := P^{\text{subs}}(|P|, k)$, for any sample $x, y \in T_k$. Then, it follows that

$$\begin{aligned}\mathbb{E}[Y_k] &= 1 - \left(\prod_{s \in T_k} 1 - P^{\text{clcs}}(s, S) \right) \\ &= 1 - \left(1 - \left(\prod_{i=1}^{m} P^{\text{subs}}(k, |s_i|) \right) \cdot P^{\text{subs}}(|P|, k) \right)^{|\Sigma|^k}, \end{aligned} \tag{5.11}$$

applying the assumed independence between the particular events and the definition of the complement of an event. Summarizing the facts from formula 5.10 and formula 5.11, the expected length calculation heuristic for a CLCS is approximated for any $v \in V$ by

$$\begin{aligned}&\text{EX}^{\text{CLCS}}(v) = |P| - u^v + (l_{\min} - (|P| - u^v + 1) + 1) - \\ &\sum_{k=|P|-u^v+1}^{l_{\min}} \left(1 - \left(\prod_{i=1}^{m} P^{\text{subs}}(k, |s_i| - p_i^{\text{L},v} + 1) \right) \cdot P^{\text{subs}}(|P| - u^v, k) \right)^{|\Sigma|^k} = \\ &= l_{\min} - \sum_{k=|P|-u^v+1}^{l_{\min}} \left(1 - \left(\prod_{i=1}^{m} P^{\text{subs}}(k, |s_i| - p_i^{\text{L},v} + 1) \right) \cdot P^{\text{subs}}(|P| - u^v, k) \right)^{|\Sigma|^k} \end{aligned} \tag{5.12}$$

Note that formula 5.12 covers also the case for LCS problem, i.e, when $P = \varepsilon$, for which $P^{\text{subs}}(|P| - u^v, k) = 1$ holds.

Since $|\Sigma|^k$ might be a huge number, even non-storable within common double floating-point arithmetic, we make use of the power decomposition to deal with the issue. In details, each power can be decomposed as follows

$$(1 - x)^{|\Sigma|^k} = \left(\underbrace{\left(\ldots (1 - x)^{|\Sigma|^{k'}} \ldots \right)^{|\Sigma|^{k'}}}_{\lfloor \frac{k}{k'} \rfloor} \right)^{|\Sigma|^{(k \mod k')}}, \tag{5.13}$$

for each $x \in \mathbb{R}$. We set $k' = 25$ in our implementation.
To calculate (5.12), we need $O(m \cdot n)$ time, but this we speed up since it is not necessary to calculate each $E_k = \mathbb{E}[Y_k], k = |P|, \ldots, l_{\min}$, but just relevant ones that belong to the interval $(\epsilon, 1 - \epsilon)$. This is because most of the E_k values are nearly 0 or 1. We make use of the divide-and-conquer technique exploiting the fact that the sequence $\{E_k\}_k$ is decreasing and all the values of its items are in $[0, 1]$ to detect the relevant interval $(\epsilon, 1 - \epsilon)$ and calculate only those E_k which belong to the interval and in that way speed up the calculation of (5.12). We emphasize that the same scheme has been used in [25] for the LCS problem. Therefore, EX^{CLCS} is calculated in expected runtime $O(m \log n)$.

Additionally, an instability in calculation may occur when determining $(1-x)^\alpha$ for a minor value of x and huge α due to cancellation effects in the classical floating point arithmetic. We solved the issue by using following transformation. To ease the derivation, let $x = P^{\text{subs}}(k, |s_i|) \cdot \prod_{i=1}^{m} P^{\text{subs}}(k, |s_i|)$ and $\alpha = |\Sigma|^k$. When x is supposed to be small, i.e. in our implementation it means when $x \leq 10^{-10}$, we handle the case in the following different ways. Numerically, unstable cases occur when x is small and α is huge. We make use of the expression $\log(1-x)/x$ which is well approximated by the Taylor series as $-1 - \frac{x}{2} + o(x)$ when x close to 0, i.e.,

$$(1-x)^\alpha = \exp^{\alpha x \cdot \frac{\log(1-x)}{x}} \approx \exp^{\alpha x (-1-\frac{x}{2})}. \tag{5.14}$$

Calculation of $\alpha \cdot x$ can be still be numerically unstable to calculate (since one product is small while the other is large). Therefore, we may rewrite

$$\alpha \cdot x = \exp^{\log \alpha \cdot x} = \exp^{k \log|\Sigma| + \log x}, \tag{5.15}$$

If $k \log|\Sigma| + \log x$ large, the result of (5.14) will be negligible small. Our implementation checks if $k \log|\Sigma| + \log x > 300$, in which case $(1-x)^\alpha < \exp(-300)$, and therefore, zero is returned. Otherwise, $\tilde{\alpha} := \alpha x(-1 - \frac{x}{2})$ is determined. If $\tilde{\alpha}$ is close to zero, i.e. $\tilde{\alpha} < 10^{-15}$ in our implementation, we may again use the Taylor expansion of exponential function to get

$$(1-x)^\alpha = \exp^{\tilde{\alpha}} \approx 1 + \tilde{\alpha}. \tag{5.16}$$

For remaining cases, we consider $\tilde{\alpha}$ to be reasonably large such that $\exp^{\tilde{\alpha}}$ can be calculated in a numerically stable way and return this value as an approximate result of the $(1-x)^\alpha$. Thus, whenever $x \leq 10^{-10}$, we get that

$$(1-x)^\alpha = \begin{cases} 0, & k \cdot \log|\Sigma| + \log(x) > 300 \\ 1 + \tilde{\alpha}, & k \cdot \log|\Sigma| + \log(x) \leq 300 \ \wedge \ \tilde{\alpha} < 10^{-15} \\ \exp^{\tilde{\alpha}}, & \text{otherwise.} \end{cases}$$

5.5 Pattern Ratio Heuristic

For each node v, we may use the following heuristic

$$R(v) = \frac{\min_{i=1,\dots,m}(|s_i| - p_i^{\mathrm{L},v} + 1)}{|P| - u^v + 1}. \tag{5.17}$$

Note that a similar function was used in the context of the LCS problem in [69] and is here extended towards the m–CLCS problem. The use of the pattern ratio heuristic can be justified by the following two facts. First, the larger the length of the minimal (remaining) string in $I[p^{\mathrm{L},v}]$ is, the longer is the expected CLCS. Second, the larger the suffix string of P that remains to fulfill is, the shorter is the expected CLCS. Nodes with higher values are preferred over those with lower ones. Note that the use of (5.17) could introduce many ties among nodes. To break them, we introduce a well-known k-norm rule by

$$||v||_k^k = \sum_{i=1}^{m} \left(\frac{|s_i| - p_i^{\mathrm{L},v} + 1}{|P| - u^v + 1} \right)^k, \; k > 0,$$

and always preferring those nodes with higher $|| \cdot ||_k$ values. A value of $k \in \mathbb{R}^+$ is considered as a parameter of the algorithm. We set up $k = 2$ (the euclidean distance norm) in our experiments.

CHAPTER 6

Beam Search for the m–CLCS Problem

Beam search (BS) is a heuristic method to approach combinatorial optimization problems. It is well-known for its application in the context of scheduling problems (see, e.g., [56, 64]) but it is widely used to many other fields. Blum et al. [7] was the first who applied BS to solve the LCS problem. Later, due to the effectiveness which the BS heuristic showed, many others followed their approach utilizing BS in order to solve larger LCS instances heuristically, e.g., [54, 25].

6.1 General Beam Search Framework

In order to set up BS for the m–CLCS problem, we derive the state graph (node structure, complete nodes and arcs) as explained in Section 5.1. Furthermore, a function for evaluating the nodes is required. In the interest of being able to easily change the method of evaluation so that various configurations can be tested, we define the heuristic function h as a parameter of the algorithm. A general framework for using BS for the LCS problem is presented in [25], the pseudocode of an adapted version of the General BS framework in the context of the CLCS problem is shown by Algorithm 6.1.

The algorithm works on a set of nodes which is called *beam*, henceforth labelled by B. At the start, the beam is initialized with only the root node, representing an empty subsequence. At each iteration of the BS, procedure ExtendAndEvaluate(B, h, β, k_{ext}) expands all nodes from B, generating an extension set V_{ext}. At most $\lfloor \beta k_{\text{ext}} \rfloor$ new nodes are kept, which are determined by ranking the nodes according to a greedy criterion $g(v, a)$ (see equation (4.1)). If $k_{\text{ext}} = \infty$, this step is skipped. The heuristic h is consequently used to evaluate all nodes; the choices for h are discussed in Chapter 5. If any node from the extension set is a complete node with a larger length l^v than any found solution

Algorithm 6.1 Beam Search for m–CLCS Problem

```
Input: a problem instance ($S = \{s_1, \ldots, s_m\}, P, \Sigma$), $h$: heuristic function, $ub_{\text{prune}}$: upper bound function, $\beta$: beam width, $k_{\text{best}}$: parameter to check dominance, $k_{ext} \geq 1$: parameter to make pre-reduction of the beam size
Output: a feasible CLCS solution $s_{clcs}$
create root node $r \leftarrow ((1, ..., 1), 0, 0)$
$B \leftarrow \{r\}$
$s_{clcs} \leftarrow \varepsilon$
while $B \neq \emptyset$ do
    $V_{\text{ext}} \leftarrow$ ExtendAndEvaluate($B, h, \beta, k_{ext}$)
    update $s_{clcs}$ if a complete node $v$ with a new largest $l^v$ value reached
    $V_{\text{ext}} \leftarrow$ Prune($V_{\text{ext}}, ub_{\text{prune}}$)        // optional
    $V_{\text{ext}} \leftarrow$ Filter($V_{\text{ext}}, k_{\text{best}}$)          // optional
    $B \leftarrow$ Reduce($V_{\text{ext}}, \beta$)
end while
return $s_{clcs}$
```

so far, a new incumbent solution has been found, which is stored in s_{clcs}. Procedure Prune($V_{\text{ext}}, ub_{\text{prune}}$) removes all nodes whose upper bound value is lower or equal than the length of the best found solution so far; note that ub_{prune} must be a valid upper bound. To make pruning even at early iterations of BS possible, an initial solution is obtained as the outcome of the Greedy method, executed before the BS procedure. Procedure Filter($V_{\text{ext}}, k_{\text{best}}$) checks if there are nodes from V_{ext} that are dominated by other nodes. We say that a node v *dominates* another node v' of same length iff $p_i^{\text{L},v} \leq p_i^{\text{L},v'}$, for all $i = 1, \ldots, m \wedge u^v \geq u^{v'}$. This presents an extension of the *domination* relation introduced by Blum et al. [7] in the context of the LCS problem. Dominated nodes cannot lead to better solutions than the nodes by which they are dominated, hence, they are removed. As it can be a time-demanding task to examine all pairs of nodes, the domination is checked for each node $v \in V_{\text{ext}}$ only against the best k_{best} nodes (with respect to their heuristic values). The last step of each iteration of a BS run is Reduce(V_{ext}, β), where the best β nodes are selected w.r.t. heuristic h to generate the new beam for the next iteration. We repeat the same steps until beam B is empty.

6.2 A Working Example of Beam Search

To illustrate the workings of BS, we revisit the earlier introduced example instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$) for which the full state graph was already presented in Figure 5.1. The state graph generated by a run of BS with $\beta = 2$, $h(v) = l^v + \text{UB}(v)$, pruning disabled and filtering with $k_{\text{best}} = \infty$ on the above instance is given in Figure 6.1. In the first extension step, two child nodes of the root node are created. Since $\beta = 2$, both are kept and expanded again in the next iteration resulting in four nodes at level 2 of the search tree. Both the nodes $((4, 6), 2, 1)$

and $((6,4),2,1)$ are dominated by node $((3,4),2,1)$ and are therefore dropped out from further search. Subsequently, three nodes are created at iteration 3. To reduce the beam to a size of two, the node $((8,7),3,2)$ is dropped since its value from heuristic $h(v)$ is the lowest among the three nodes at this iteration. Then the BS continues expanding nodes from the beam. The first complete node is found at iteration 5, but the search is not yet finished. The best solution is finally found in iteration 6 when discovering node $((9,10),6,3)$. Since there is no more nodes left in the beam to expand, the BS stops.

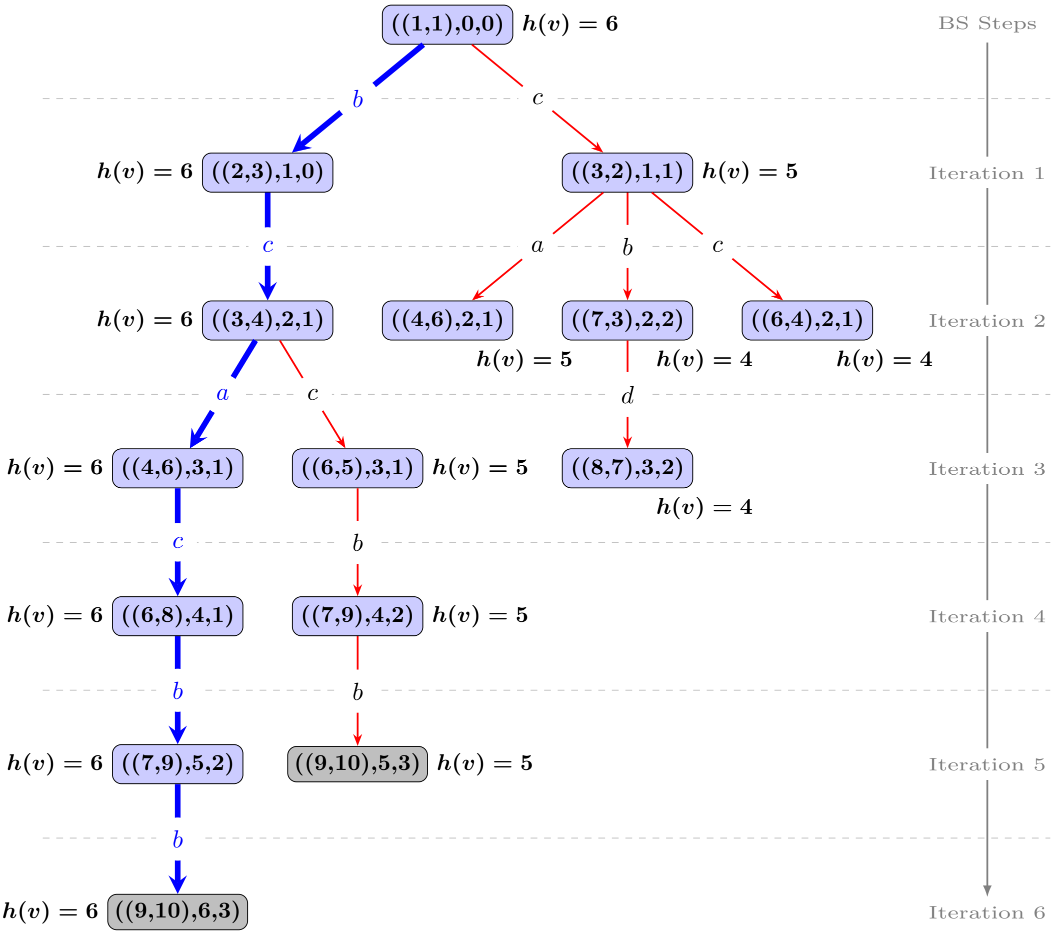


Figure 6.1: The example above shows the state graph generated by a run of BS ($\beta = 2$, $h(v) = l^v + \mathrm{UB}(v)$, $k_{\text{best}} = \infty$, no pruning) for the instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). The nodes filled by light-gray color are complete nodes of the state graph. Next to each node the value of the respective node from heuristic $h(v) = l^v + \mathrm{UB}(v)$ is displayed. The best found solution (which here is also an optimal solution) obtained by BS is $s = \texttt{bcacbb}$ of length 6, and it corresponds to the complete node $((9,10),6,3)$. The longest path that corresponds to that solution is displayed in blue.

CHAPTER 7

A* Search for the m–CLCS Problem

A* is a well-known informed search algorithm that originated from the field of pathfinding [35]. It is a graph traversal search, formulated for weighted graphs. It has been successfully applied to solve the LCS problem and its variants [24, 67] up to completeness. The basic principles of A* search were already described in Chapter 3, and we here set up an A* search to solve the m-CLCS problem.

7.1 A* Search Algorithm

Our A* search for the m–CLCS problem works on the state graph (node structure, complete nodes and arcs) described in Section 5.1. In order to set up the search, it remains to derive an evaluation function $f(v) = g(v) + h(v)$. We use $g(v) := l^v$ and an upper bound on the length of the CLCS for $h(v)$; possible candidates for this purpose were proposed in Section 5.2, for the majority of our experiments we use $h(v) := \mathrm{UB}(v)$, $v \in V(G)$.

For the remaining part of this section, we mainly follow the description of our A* search that has been provided in our report [23]. In order for the search process to be efficient, our implementation maintains two data structures: (1) a *hash-map* N storing all nodes that were encountered during the search, and (2) an *open list* $Q \subseteq N$ containing all not yet expanded/treated nodes. More specifically, N is implemented as a nested data structure of sorted (linked) lists within a hash map. The position vector $p^{\mathrm{L},v}$ of a node $v = (p^{\mathrm{L},v}, l^v, u^v)$ is mapped to a list storing pairs $(l^v,\ u^v)$. This structure allows for an efficient membership check, i.e., whether or not a node that represents subproblem a $I[p^{\mathrm{L},v}]$ was already encountered during the search, and a quick retrieval of the respective nodes. Note that storing multiple nodes presenting one subproblem $I[p^{\mathrm{L},v}]$

might occur, as the following short example demonstrates: Consider the problem instance with input strings s_1 = `bacxmnob`, s_2 = `abcxmbno`, s_3 = `acbxbmno`, and pattern string P = `b`. The A* search might, at some time, encounter node $v_1 = ((4,4,4),2,1)$ induced by partial solution `bx`, and—at some other time—it might encounter another node $v_2 = ((4,4,4),3,0)$ induced by partial solution `acx`. Even though the path from the root node to node v_1 is shorter than the path to node v_2, the former still leads to a better solution in the end (`bxmno` in comparison to `acxb`). As the information which of the nodes leads to an optimal solution is not known beforehand, both nodes are stored. Finally, the open list Q is realized by a priority queue with priority values $\pi(v) = l^v + \mathrm{UB}(v)$, for all $v \in V$. In case of ties, nodes with larger l^v-values are preferred. In the case of further ties, nodes with larger u^v-values are preferred.

The search starts by inserting the root node of the state graph into N and Q. Then, at each iteration, a node v with highest priority is retrieved from Q and expanded by considering all successor nodes for $c \in \Sigma_v^{\mathrm{nd}}$). If such an extensions leads to a new state, the corresponding node, denoted by v_{ext}, is added to N and Q. Otherwise, v_{ext} is compared to the nodes from set $N_{\mathrm{rel}} \subseteq N$ containing those nodes that represent the same subproblem $I[p^{\mathrm{L},v}]$. Dominated nodes are identified in this way and dropped from the search process, i.e., the dominated nodes are removed from N and Q. If node v_{ext} is dominated by one of the nodes from N_{rel}, it can simply be discarded. Otherwise, it is added to N and Q. In this context, given $v_1, v_2 \in N_{\mathrm{rel}}$ we say that v_1 *dominates* v_2 iff $l^{v_1} \geq l^{v_2} \,\wedge\, u^{v_1} \geq u^{v_2}$. We would like to emphasize that detecting the domination in N_{rel} was, on average, slightly faster when the elements of the lists were sorted in decreasing order of their u^v-values. Therefore, we used this ordering in our implementation.

As the upper bound function UB() is *admissible*—that is, it never underestimates the length of an optimal solution—A* search yields an optimal solution whenever the node selected for expansion is a complete node [35]. Moreover, note that UB() also is *monotonic*, which means that the upper bound of any child node never overestimates the upper bound of its parent node. This implies that no re-expansion of already expanded nodes become necessary [35]. In general, A* search is known to be optimal in terms of the number of node expansions required to prove optimality w.r.t. the upper bound and the tie–breaking criterion used. A pseudocode of our A* search implementation for the CLCS problem is provided in Algorithm 7.1.

7.2 A Working Example of A* Search

To illustrate an example of A* search, the state graph generated by performing an A* search on an instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a},\texttt{b},\texttt{c},\texttt{d}\}$) is given in Figure 7.1. In the first expansion step, two child nodes of the root node are created. The next node for expansion is node $((2,3),1,0)$ since it has a higher priority value than node $((3,2),1,1)$. The search then continues always expanding the non-expanded node with the highest priority value. Once node $((9,10),6,3)$ is selected for expansion and detected to be a complete node, the search stops and the final solution

Algorithm 7.1 A* Search for the m–CLCS Problem

```
Input: a problem instance $(S = \{s_1, \ldots, s_m\}, P, \Sigma)$
Output: an optimal CLCS solution
Initialize: an empty hash-map $N$ and priority queue $Q$
create root node $r \leftarrow ((1, ..., 1), 0, 0)$ and add it to $N$ and $Q$
while $Q \neq \emptyset$ do
    pop node $v$ with highest priority from $Q$
    determine $\Sigma_v^{\mathrm{nd}}$ for node $v$
    if $\Sigma_v^{\mathrm{nd}} = \emptyset$ then    // a complete node has been reached
        return solution derived through path from root $r$ to node $v$
    else
        for each $c \in \Sigma_v^{\mathrm{nd}}$ do
            generate node $v_{\mathrm{ext}}$ by appending $c$ to the partial solution of node $v$
            retrieve set $N_{\mathrm{rel}} \subseteq N$ presenting the subproblem $I[p^{\mathrm{L},v_{\mathrm{ext}}}]$
            $insert \leftarrow true$
            for each $v_{rel} \in N_{\mathrm{rel}}$ do
                if $l^{v_{\mathrm{rel}}} \geq l^{v_{\mathrm{ext}}} \ \wedge \ u^{v_{\mathrm{rel}}} \geq u^{v_{\mathrm{ext}}}$ then
                    $insert \leftarrow false$
                    break    // domination condition is fulfilled
                end if
                if $l^{v_{\mathrm{ext}}} \geq l^{v_{\mathrm{rel}}} \ \wedge \ u^{v_{\mathrm{ext}}} \geq u^{v_{rel}}$ then
                    remove $v_{\mathrm{rel}}$ from $N$ and $Q$
                end if
            end for
            if $insert$ then    // a new state generated
                add $v_{\mathrm{ext}}$ to $N$ and $Q$
            end if
        end for
    end if
end while
return no feasible solution exists
```

is derived through the arc labels on the path from root node to node $((9, 10), 6, 3)$. Note that the two nodes $((6, 5), 3, 1)$ and $((3, 2), 1, 1)$ remain in the priority queue until the end and get never expanded. Note that about half of the overall states of the full state graph are not even visited by A* which might give us an insight into the efficiency of the search.

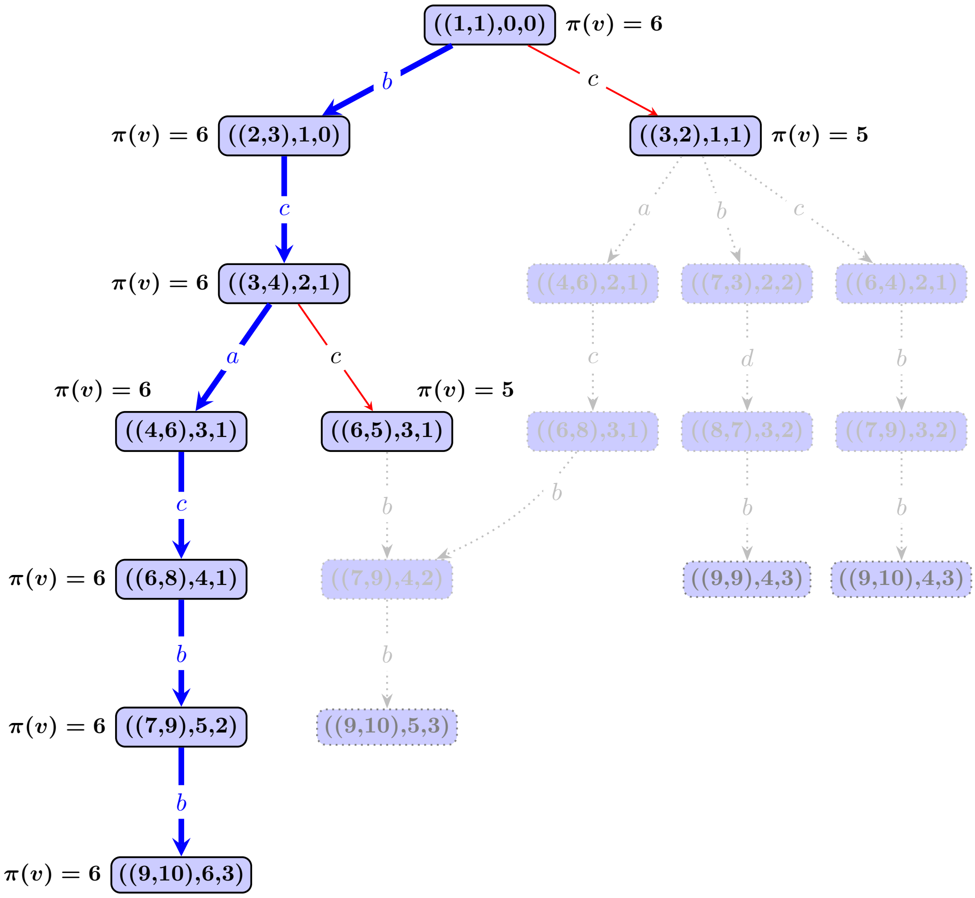


Figure 7.1: The above is an example given, showing the full state graph generated for the instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). The states created by the run of A* search are drawn solid, while the other (not reached) states are dashed and transparent. Displayed next to each node expanded by A* search is an priority value of the respective node, i.e., $\pi(v) = l^v + \mathrm{UB}(v)$. After 6 node expansions, a proven optimal solution $s = \texttt{bcacbb}$ is discovered by the A* algorithm, and it corresponds to the complete node $((9, 10), 6, 3)$. The longest path that corresponds to the optimal solution is displayed in blue.

CHAPTER 8

Algorithms for the Classical 2–CLCS Problem

In this chapter we describe the most relevant DP-based algorithms from literature for the classical 2–CLCS problem [63] with two input strings s_1 and s_2 and a pattern string P. Moreover, in Section 8.6 a model to solve the 2–CLCS Problem with *integer linear programming* (ILP) is formulated. We are not aware of any ILP model proposed for the 2–CLCS problem so far, so from computational point-of-view it is interesting to check efficiency of the general-purpose state-of-the-art solvers, such as Cplex solver, on this problem. Computational studies are then performed for all of the described algorithms and our A* approach in Chapter 9.

8.1 Algorithm by Chin et al.

The approach of Chin et al. [15] uses of a three-dimensional matrix M to calculate the CLCS. In detail, for any $0 \leq i \leq |s_1|$, $0 \leq j \leq |s_2|$, $0 \leq k \leq |P|$, the length of an optimal solution for the CLCS subproblem of $s_1[1,i]$ and $s_2[1,j]$ with respect to $P[1,k]$ is stored at $M(i,j,k)$. Matrix M is defined by the following DP recursive relation:

$$M(i,j,k) = \begin{cases} 1 + M(i-1, j-1, k-1), & \text{if } i,j,k > 0 \land \\ & \land\ s_1[i] = s_2[j] = P[k] \\ 1 + M(i-1, j-1, k), & \text{if } i,j > 0, s_1[i] = s_2[j] \land \\ & \land\ (k = 0 \lor s_1[i] \neq P[k]) \\ \max\{M(i-1,j,k), M(i,j-1,k)\}, & \text{if } i,j > 0\ \land\ s_1[i] \neq s_2[j] \end{cases} \tag{8.1}$$

with boundary conditions $M(i,0,0) = 0$, $M(0,j,0) = 0$, $M(i,0,k) = -\infty$, $M(0,j,k) = -\infty$ for $i = 0,...,|s_1|$, $j = 0,...,|s_2|$ and $k = 1,...,|P|$. The solution string for any subproblem can be constructed by backtracking the computation path from its corresponding matrix cell $M(i,j,k)$ to $M(0,0,0)$ and at each step prepending the letter $s_1[i]$ if the value was computed from $M(i-1,j-1,k)$ or $M(i-1,j-1,k-1)$. The time and space complexity of the algorithm of Chin et al. is $O(|s_1| \cdot |s_2| \cdot |P|)$.

8.2 Algorithm by Arslan and Eğecioğlu

Another algorithm to solve the classical 2–CLCS problem in $O(|s_1| \cdot |s_2| \cdot |P|)$ time was proposed by Arslan and Eğecioğlu [3]. Their method calculates the optimal length of CLCS subproblems in a similar way as the approach of Chin et al., but it is based on multiple three-dimensional matrices defined as follows:

$$M(i,j,k) = \max\{M'(i,j,k), M(i,j-1,k), M(i-1,j,k)\} \tag{8.2}$$

with

$$M'(i,j,k) = \max\{M''(i,j,k), M'''(i,j,k)\} \tag{8.3}$$

and

$$M''(i,j,k) = \begin{cases} 1 + M''(i-1,j-1,k-1), & \text{if } (s_1[i] = s_2[j] = P[k]) \wedge \\ & (k = 1 \vee (k > 1 \wedge \\ & \wedge\ M(i-1,j-1,k-1) > 0)) \\ 0, & \text{otherwise} \end{cases} \tag{8.4}$$

$$M'''(i,j,k) = \begin{cases} 1 + M'''(i-1,j-1,k), & \text{if } (s_1[i] = s_2[j]) \wedge \\ & (k = 0 \vee M(i-1,j-1,k) > 0) \\ 0, & \text{otherwise} \end{cases} \tag{8.5}$$

and boundary conditions $M(i,0,k) = 0$, $M(0,j,k) = 0$ for $i = 0,...,|s_1|$, $j = 0,...,|s_2|$ and $k = 0,...,|P|$. To obtain not only the length but also an actual CLCS, one needs keep track of the current solution string along with the calculations. A specialty of the proposed algorithm is that it can easily be adapted to solve a variant of the CLCS problem where the objective is to find an LCS for s_1, s_2 and a sequence, whose edit distance from pattern P is less than a positive integer that is given in advance.

8.3 Algorithm by Deorowicz

The algorithm developed by Deorowicz [21] was the first sparse approach, i.e., it is only calculating some and (usually) not all partial solutions. It solves the 2–CLCS problem in $O(|P| \cdot (|s_1| \cdot L + R) + |s_2|)$ time where L is the length of LCS between s_1 and s_2 and R is the number of pairs of matching positions between s_1 and s_2. In detail, the matrix of computation is processed for each level $k = 0, \ldots, |P|$ in a row-wise manner and an ordered list is maintained to store for each rank (representing the assumed length of a CLCS) the lowest possible column number. Furthermore, a two-dimensional matrix T is used to store computed values of the current and previous level. For each row i and column j, s.t. $s_1[i] = s_2[j]$, a partial solution is computed. If $s_1[i] = s_2[j] \neq P[k]$, then the value for the match at (i, j) is calculated from the highest rank in the list with a column number lower than j. Otherwise if $s_1[i] = s_2[j] = P[k]$, the value is calculated from matrix T. When for some rank a lower column number than previously known is found, the corresponding list entry is updated (or created if none is there) and the new information is cascaded to all following list entries. On completion, the highest rank in the list corresponds to the length of a CLCS. The algorithm is especially suited for a larger alphabet Σ.

Improvements of the algorithm were later proposed by Deorowicz and Obstoj [22]. They introduce so called *external–entry points* (EEP) [36], initially proposed for the pairwise sequence alignment problem, with the purpose of omitting those cells in computation that do not contribute to optimal CLCS solutions. We did our best to re-implement those improvements but were not successful due to lack of information and open questions concerning the integration of EEP into their approach. The both authors were contacted, but they have never replied to our queries.

8.4 Algorithm by Iliopoulos and Rahman

Iliopoulos and Rahman [42] proposed another sparse algorithm to solve the 2–CLCS problem in $O(|P| \cdot R \cdot \log\log|s_1| + |s_2|)$ time where R is the number of pairs of matching positions between s_1 and s_2. They modified the dynamic programming formulation of Arslan and Eğecioğlu and developed the following:

$$M(i,j,k) = \max\{M'(i,j,k), M''(i,j,k), M(i,j-1,k), M(i-1,j,k)\} \tag{8.6}$$

with

$$V_1 = \max_{1 \leq l_i < i, 1 \leq l_j < j, s_1[l_i] = s_2[l_j]} \{M(l_i, l_j, k-1)\} \tag{8.7}$$

$$M'(i,j,k) = \begin{cases} 1 + V_1, & \text{if } (s_1[i] = s_2[j] = P[k]) \land (k = 1 \lor (k > 1 \land V_1 > 0)) \\ 0, & \text{otherwise} \end{cases} \tag{8.8}$$

$$V_2 = \max_{1 \leq l_i < i, 1 \leq l_j < j, s_1[l_i] = s_2[l_j]} \{M(l_i, l_j, k)\} \tag{8.9}$$

$$M''(i,j,k) = \begin{cases} 1, & \text{if } (s_1[i] = s_2[j]) \ \land \ (i = 1 \lor j = 1) \\ 1 + V_2, & \text{if } (s_1[i] = s_2[j]) \land (k = 0 \lor V_2 > 0) \\ 0, & \text{otherwise.} \end{cases} \tag{8.10}$$

In the algorithm of Iliopoulos and Rahman, only matrix values where $s_1[i] = s_2[j]$ are calculated. Some extra variables are used to keep track of the current solution string. To retrieve the values of V_1 and V_2, a *BoundedHeap* data structure [11] realized by means of Van Emde Boas (vEB) trees [65] is used. In this way, updating and finding values in the heap can be done in $O(\log\log n)$ amortized time where n is the maximum number of keys supported by the *BoundedHeap* data structure.

8.5 Algorithm by Hung et al.

Very recently, Hung et al. [40] presented an algorithm that solves the 2–CLCS problem in $O(|P| \cdot L \cdot (|s_1| - L))$ time where L denotes the length of a CLCS and $|s_1| \leq |s_2|$. Hence, the approach is especially suited for input strings that are highly similar. It was developed on the basis of the diagonal concept for the LCS problem by Nakatsu et al. [55]. The algorithm builds a table D of dimension $|P| \times L$. Every cell $D_{i,l}$ of the table stores a set of triples. Hereby, each triple $(i', j, k) \in D_{i,l}$, $i' \leq i$ refers to a common subsequence of length l for $s_1[1, i']$ and $s_2[1, j]$ with respect to $P[1, |P| - k]$. The elements belonging to $D_{i,l}$ are generated by extending all the partial solutions from $D_{i-1,l-1}$ and further adding all triples of $D_{i-1,l}$. Subsequently, a domination check is performed to filter out those triples that are dominated by another triple within the same table cell. When a triple $(i', j, 0) \in D_{i,l}$ is encountered and there is no other $(i'', j'', 0) \in D_{i,l}$, $i' \neq i''$ and $j \neq j''$, it implies that a CLCS of $s_1[1, i']$, $s_2[1, j]$ with respect to P is of length l. Finally, the largest such found l over the course of the computations represents the length of a CLCS of the problem instance.

8.6 An ILP model for the 2–CLCS Problem

We are not aware of any previous approaches that uses integer programming techniques to solve the 2–CLCS problem. In order to check efficiency in course of our experimental studies, we formulate an ILP model as follows by extending the ILP model proposed for the LCS problem [8]. We say that (i, j) is a *weak* matching iff $s_1[i] = s_2[j], 1 \leq i \leq |s_1|$ and $1 \leq j \leq |s_2|$. Similarly, we say that (i, j, k) is a *strong* matching iff $s_1[i] = s_2[j] = P[k]$, $1 \leq k \leq |P|$. For each weak matching (i, j) we assign a binary variable z_{ij}; the set of such variables is denoted by Z. Similarly, for each strong match, a binary variable w_{ijk} is assigned; the set of all such variables is denoted by W. Two different variables z_{ij} and z_{kl} from Z are in *conflict* iff $(i \geq j$ and $k > l) \vee (i > k$ and $j \leq l)$ holds. Similarly, we say that two different variables w_{ijk}, $w_{pqr} \in W$ are in *conflict* iff (z_{ij} and z_{pq} are in conflict) $\vee (k = r) \vee$ (if $i > p$ then $k \leq r \vee$ if $i < p$ then $k \geq r$) holds. An ILP model for the 2–CLCS problem is then stated as follows:

$$\max \sum_{z \in Z} z \tag{8.11}$$

$$s.t.$$

$$z_{ij} + z_{kl} \leq 1 \qquad \text{for all } z_{ij}, z_{kl} \in Z, z_{ij} \neq z_{kl}, \text{ in conflict} \tag{8.12}$$

$$w_{ijk} + w_{pqr} \leq 1 \qquad \text{for all } w_{ijk}, w_{pqr} \in W, w_{ijk} \neq w_{pqr}, \text{ in conflict} \tag{8.13}$$

$$\sum_{w \in W} w = |P| \tag{8.14}$$

$$w_{ijk} \leq z_{ij} \qquad \text{for all } i = 1, \ldots, |s_1|, j = 1, \ldots, |s_2|, k = 1, \ldots, |P| \tag{8.15}$$

$$z_{ij}, w_{pqr} \in \{0, 1\} \qquad \text{for all } z_{ij} \in Z, w_{pqr} \in W. \tag{8.16}$$

Constraint (8.14) ensures that pattern string P is satisfied, i.e., exactly $|P|$ strong matchings are there in a feasible solution. By constraint (8.15) a relation between the strong matches and weak matches (i.e., the corresponding variables) is established. More precisely, we force that strong matches included into a solution are at the same time weak matches, i.e., $\forall i, j, k$ if $w_{ijk} = 1$ then $z_{ij} = 1$. Constants (8.12)–(8.13) ensure that neither of two variables from Z or W that are in conflict are turned on (i.e., $\neq 0$) at the same time (the relation of "being subsequence" is kept). Note that when $P = \varepsilon$, the above model yields the ILP model for the basic LCS problem [8]. However, this model does not scale well to the general m–CLCS problem since the number of constraints in the model grows exponentially in instance size and, therefore, this model will only fit to solve the 2–CLCS problem.

CHAPTER 9

Experimental Studies

In this chapter we describe the exhaustive experimental studies of the devised concepts. We start with the execution environment and implementation details in Section 9.1 and the used benchmark instances in Section 9.2. In Section 9.3, we study the performance of our A* search on the well-studied 2–CLCS problem and compare it to the performance of the existing state-of-the-art algorithms from literature specialized in solving this problem. Finally, in Section 9.4, we present computational results obtained from our A* search, greedy heuristic and a several beam search configurations for the m–CLCS problem.

9.1 Setup and Implementation Details

All algorithms were implemented in C++ and the experiments were conducted in single-threaded mode on a machine with an Intel Xeon E5–2640 processor with 2.40 GHz and a memory limit of 32 GB. The maximum computation time allowed for each run was limited to 15 minutes, i.e., 900 seconds. All the implementations were compiled using the C++ compiler GCC 7.4 with the highest available optimization level (-O4 flag has been turned on). The ILP model described in Section 8.6 was executed by Cplex 12.7 in single-threaded execution.

We aimed to re-implement all algorithms from literature the way they were described in their original articles as the respective code could not been obtained. In a few cases, due to a lack of sufficient details, we had to make our own specific implementation decisions. In particular, this was the case for the approach of Iliopoulos and Rahman [42]. The *bounded heap* data structure used in the algorithm has to be initialized for different indices, and it remains unclear how this has originally been done. In our implementation we create a new bounded heap for a new index by copying all the contents from the bounded heap of a previous index. This is the most time-demanding part of the algorithm, which is in particular noticed in the context of instances with large values of n.

We emphasize that in general, we did our best to achieve efficient re-implementations of the approaches from literature for the experimental comparison.

9.2 Benchmark Instances

For each combination of the number of input strings $m \in \{10, 50, 100\}$, the length of strings $n \in \{100, 500, 1000\}$, the alphabet size $|\Sigma| \in \{4, 20\}$ and the ratio between the length of the pattern string and the length of the largest input strings $p' = \frac{|P|}{n} \in \left\{\frac{1}{50}, \frac{1}{20}, \frac{1}{10}, \frac{1}{4}, \frac{1}{2}\right\}$, 10 instances are randomly generated, which gives 900 instances in total[1]. Each of instance sets is generated w.r.t. the following procedure. First, a pattern string P is generated uniformly-at-random (each character from Σ has equal probability to be chosen as i-th character of P). Further, we generate m input strings of equal length n by randomly generating different $|P|$ positions for each s_i and then setting up the characters of P at those positions. Remaining characters of each string s_i are filled in by choosing letters uniformly-at-random over alphabet Σ. This procedure ensures that at least one feasible CLCS solution exists to any of the generated instances.

We also apply the procedure as described above to generate the benchmark set for the 2–CLCS problem such that for each combination of the alphabet size $|\Sigma| \in \{4, 12, 20\}$ and the length of input strings $n \in \{100, 500, 1000, 2000\}$, we generate 10 instances, which gives us 600 instances in total. In addition to these artificially generated instances, we further use a benchmark suite from [22] that contains strings representing real-world biological sequences[2]. This benchmark set is henceforth called Real. Its data sets are taken from the experiments on the constrained multiple sequence alignment problem presented in [50] and [16]. Each possible pair of sequences within a data set together with some assumed constraint sequence build an instance for the 2–CLCS problem. Properties of the data sets are shown in Table 9.1. Overall, benchmark set Real consists of 121 problem instances.

Table 9.1: Data sets in the real data benchmark suite from [22].

data set	number of sequences	sequence length (min, med, max)	$\|\Sigma\|$	origin
$ds0$	7	(111, 124, 134)	20	[16]
$ds1$	6	(124, 149, 185)	20	[16]
$ds2$	6	(131, 142, 160)	20	[16]
$ds3$	5	(189, 277, 327)	20	[16]
$ds4$	6	(98, 114, 123)	20	[50]

[1] The instances for the m–CLCS problem are available at https://www.ac.tuwien.ac.at/files/resources/instances/m-clcs.zip

[2] The instances for the 2–CLCS problem are available at https://www.ac.tuwien.ac.at/files/resources/instances/clcs/2d-clcs.zip

9.3 Computational Results for the 2–CLCS Problem

In this section we analyze the performances of our exact algorithms and the approaches from literature. We compare

- A* search presented in Chapter 7,
- the ILP model proposed in Section 8.6, and
- several state-of-the-art algorithms from literature, specialized for solving the 2–CLCS problem, as listed below:
  - Algorithm by Chin et al. [15], labeled by Chin;
  - Algorithm by Deorowicz [21], labeled by Deo;
  - Algorithm by Arslan and Eğecioğlu [3], labeled by AE;
  - Algorithm by Iliopoulos and Rahman [42], labeled by IR;
  - Algorithm by Hung et al. [40], labeled by Hung.

A short description of these 2–CLCS algorithms has been given in Chapter 8.

We present results and observations from our report [23], as well as additional data and findings from experiments on the instances of large size ($n = 2000$) and experiments with our ILP approach. Tables 9.2–9.7 show the runtimes for each re-implemented algorithm from literature as well as our A* search and our ILP approach in seconds averaged over each group of instances. Results for the artificial instance sets are subdivided into five different subclasses w.r.t. the value of p', which determines the length of pattern string P. Concerning benchmark suite Real, the average running times refer to all those instances that belong to the respective data set in combination with a pattern P, cf. Table 9.7. For each instance group (line), the lowest runtimes among the competing algorithms are shown in bold font. The first two columns present the properties of the instance group, while the third column $\overline{|s|}$ lists the average length of the optimal solutions for the respective problem instances. The following columns are reserved to report the average running times of Chin, Deo, AE, IR, Hung, our A* algorithm and of our ILP approach, respectively.

The following observations can be drawn from these results:

- The small instances (where $n = 100$) are easy to solve and all competitors require only a fraction of a second for doing so; only for the ILP approach more computation time is needed and it fails to solve any of the instances with $n = 100$ and $|\Sigma| = 4$ or $n \geq 500$. The first of the other algorithms that starts losing efficiency with growing input string length is IR. Already starting with $n = 500$, the computation times start to grow significantly in comparison to the other approaches. This might be due to our design decision concerning the issue with the *BoundHeap* data structure, as mentioned before. However, this is most likely due to the complexity of the utilized data structure.

- Algorithm Chin clearly outperforms Deo for small alphabet size $|\Sigma|$. With growing $|\Sigma|$, as already noticed in earlier studies [21], Deo becomes more efficient. In fact, the two approaches perform similarly for $|\Sigma| = 20$. The advantages of Deo over Chin are noticed in particular for higher p'; see Table 9.5.
- Algorithm Hung generally performs better than Deo and Chin. This confirms the conclusions from the computational study in Hung et al. [40].
- With increasing p' and thus an increasing length of P, all approaches degrade in their performance, except for A* and Hung, which still remain highly efficient.
- Only A*, Chin and Hung are able to solve all of the largest instances (where $n = 2000$) within the given time and memory limit.
- A general conclusion for the artificial benchmark set is that A* search is in most cases about one to two orders of magnitude faster than Hung, which is overall the second-best approach.
- Concerning the results for benchmark set Real (see Table 9.7), we can conclude that all algorithms only require short times as the input strings are rather short. Nevertheless we can also see here that the A* search is almost consistently fastest.
- Figure 9.1 shows the influence of the instance length on the algorithms' runtimes for $|\Sigma| = 4$ and $|\Sigma| = 20$. Note that IR and ILP are not included here since they were obviously the slowest among the competitors. It can be noticed that the performance of A* is the only one that does not degrade much with increasing n.
- Figure 9.2 shows the influence of the length of P on the algorithms' runtimes for $n = 500$ and $n = 1000$ (in log-scale). It can be noticed again that A* does not suffer much from an increase of the length of P. This also holds for Hung but not the other competitors, whose performance degrade with increasing $|P|$.

Finally, we also compare the amount of work done by the algorithms in order to reach the optimal solutions. In the case of A*, this amount of work is measured by the number of generated nodes of the state graph. In the case of Deo, this refers to the number of different keys (i, j, k) generated during the algorithm execution. Finally, in the case of Hung, this is measured by the amount of newly generated nodes in each $D_{i,l}$ (which corresponds to the amount of non-dominated extensions of the nodes from $D_{i-1,l-1}$). Let us call this measure the *amount of created nodes* for all three algorithms. This measure is shown in log-scale in Figure 9.3 for the instances with $n = 500$. The x-axis of these graphics varies over different ratios $p' = \frac{|P|}{n}$. The curve denoted by Max (see legends) is the theoretical upper bound on the number of created nodes, which is $|s_1| \times |s_2| \times |P|$ for an instance $(\{s_1, s_2\}, P, \Sigma)$. The graphics clearly show that A* creates the fewest nodes in comparison to the other approaches. The difference becomes larger with an increasing length of P, which correlates with an increase in the similarity between the input strings. For those instances with strongly related input strings, the upper bound UB used in the A* search is usually tighter, which results in fewer node expansions. The amount of created nodes in A* decreases with an increasing length of P after some point, because the search space becomes more restricted; see Figure 9.3 and $|\Sigma| = 4$ from $p' \geq \frac{1}{4}$ onward and $|\Sigma| = 20$ from $p' \geq \frac{1}{20}$ onward.

Table 9.2: Instances with $p' = \frac{|P|}{n} = \frac{1}{50}$: Average runtimes in seconds.

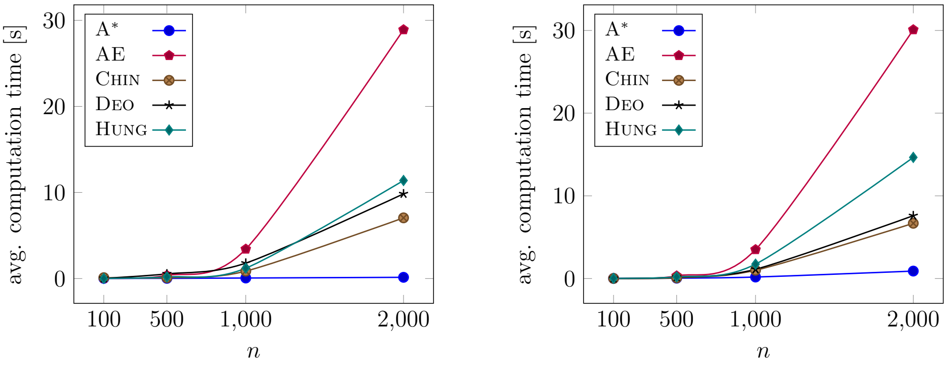
$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*	ILP
4	100	60.9	0.2	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	-
4	500	319.3	< **0.1**	0.1	0.2	6.5	0.1	< **0.1**	-
4	1000	646.3	0.2	1	1.3	86.4	0.5	< **0.1**	-
4	2000	1295.9	2.7	3.9	10.2	890.6	3.6	**0.2**	-
12	100	40.1	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**	42.0
12	500	216.0	< **0.1**	0.1	0.2	2.9	0.2	< **0.1**	-
12	1000	435.5	0.3	0.5	1.4	39.4	1	**0.1**	-
12	2000	876.4	2.6	3.3	10.2	453.8	7.6	**0.2**	-
20	100	33.5	< **0.1**	0.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**	6.4
20	500	175.7	< **0.1**	0.1	0.2	2.2	0.2	< **0.1**	-
20	1000	355.4	0.3	0.5	1.4	26.6	1.1	< **0.1**	-
20	2000	714.9	2.6	3	9.2	247.2	7.8	**0.1**	-

Table 9.3: Instances with $p' = \frac{|P|}{n} = \frac{1}{20}$: Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*	ILP
4	100	61.9	0.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	-
4	500	323.0	0.1	0.5	0.4	15.7	0.2	< **0.1**	-
4	1000	645.9	0.9	1.8	3.4	215.5	1.2	**0.1**	-
4	2000	1299.8	7	9.8	28.9	-	11.4	**0.1**	-
12	100	41.0	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**	88.7
12	500	215.3	0.1	0.2	0.4	5.3	0.3	< **0.1**	-
12	1000	437.0	0.9	1.1	3.4	69.2	2.2	**0.2**	-
12	2000	876.4	6.2	8.2	33.1	593.6	20.4	**0.2**	-
20	100	32.2	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	11.7
20	500	170.9	0.1	0.2	0.3	3.3	0.2	< **0.1**	-
20	1000	348.4	1	1.1	3.5	40.6	1.7	**0.2**	-
20	2000	696.1	6.7	7.6	30.1	387.1	14.6	**0.9**	-

Table 9.4: Instances with $p' = \frac{|P|}{n} = \frac{1}{10}$: Average runtimes in seconds.

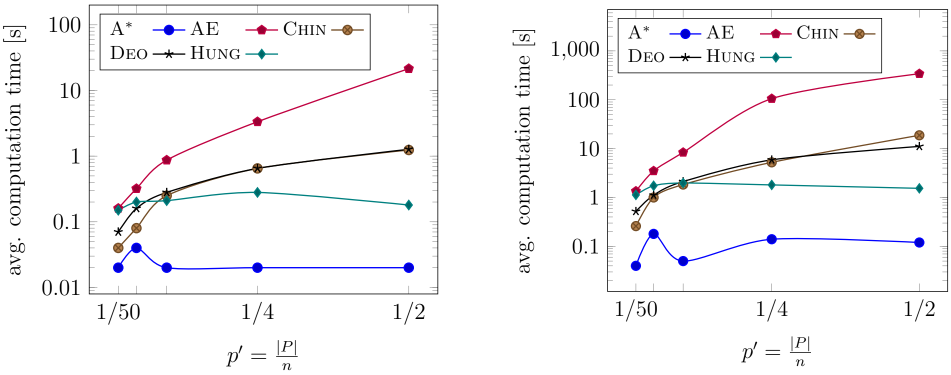
$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*	ILP
4	100	62.6	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**	-
4	500	320.9	0.3	0.6	0.9	26.8	0.4	< **0.1**	-
4	1000	646.4	1.8	3.5	9.2	331.2	3.3	< **0.1**	-
4	2000	1300.0	14.6	20.2	185.5	-	26	**0.1**	-
12	100	40.5	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**	172.1
12	500	207.1	0.2	0.3	0.9	7.3	0.3	< **0.1**	-
12	1000	419.0	2.1	2.2	8.3	91.1	2.7	**0.2**	-
12	2000	839.1	14.6	15.9	211	775.1	19.3	**0.4**	-
20	100	31.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	12.3
20	500	157.4	0.2	0.3	0.9	5.3	0.2	< **0.1**	-
20	1000	317.9	1.8	2.1	8.4	68.1	2	< **0.1**	-
20	2000	636.9	14.7	14	209.6	626	15.7	**0.2**	-



(a) $|\Sigma| = 4$

(b) $|\Sigma| = 20$

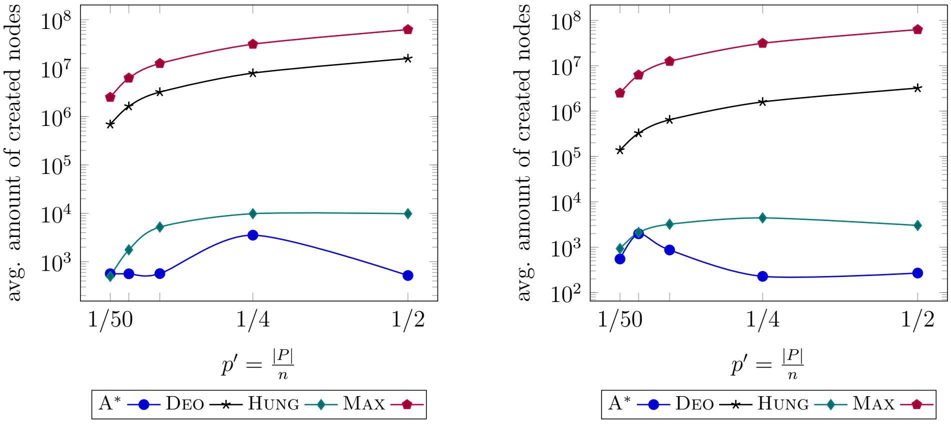
Figure 9.1: Computation times for 2–CLCS problem with $p' = \frac{1}{20}$.



(a) $n = 500$

(b) $n = 1000$

Figure 9.2: Computation times for 2–CLCS problem with $|\Sigma| = 20$.



(a) $|\Sigma| = 4$

(b) $|\Sigma| = 20$

Figure 9.3: Average amount of created nodes for 2–CLCS problem with $n = 500$.

Table 9.5: Instances with $p' = \frac{|P|}{n} = \frac{1}{4}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	Chin	Deo	AE	IR	Hung	A*	ILP
4	100	63.2	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**	-
4	500	320.1	0.6	1.4	2.7	34.8	0.5	< **0.1**	-
4	1000	642.5	5	6.6	113.6	436.6	4.5	**0.1**	-
4	2000	1281.6	90.3	-	-	-	28.4	**0.6**	-
12	100	39.9	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**	463.1[1]
12	500	203.0	0.6	0.7	3	18.7	0.3	< **0.1**	-
12	1000	413.2	5.3	5.7	112	213.2	3.2	< **0.1**	-
12	2000	818.7	100.8	-	-	-	23.1	**0.1**	-
20	100	35.7	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**	40.0
20	500	175.5	0.6	0.6	3.3	14.4	0.3	< **0.1**	-
20	1000	351.1	5.2	5.9	105.4	154.8	1.8	**0.1**	-
20	2000	704.1	81.3	-	-	-	17.3	**0.1**	-

[1] 9 out of 10 instances solved to optimality

Table 9.6: Instances with $p' = \frac{|P|}{n} = \frac{1}{2}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	Chin	Deo	AE	IR	Hung	A*	ILP
4	100	63.9	< **0.1**	< **0.1**	< **0.1**	0.2	< **0.1**	< **0.1**	-
4	500	325.5	1.4	1.5	22.5	60.6	0.4	< **0.1**	-
4	1000	652.5	19.1	12.6	336.5	739.4	3.6	< **0.1**	-
4	2000	1307.3	548.4	-	-	-	27.2	**0.1**	-
12	100	54.6	0.1	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**	884.7[1]
12	500	276.5	1.4	1.4	23.9	34.2	0.2	< **0.1**	-
12	1000	544.3	17.8	11.3	347.5	362.2	2.4	**0.1**	-
12	2000	1093.6	597.6	-	-	-	15.3	**0.1**	-
20	100	53.0	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**	152.8
20	500	264.9	1.2	1.3	21.5	30.6	0.2	< **0.1**	-
20	1000	524.5	18.8	11.1	341	278.8	1.5	**0.1**	-
20	2000	1055.5	558.2	-	-	-	12	**0.1**	-

[1] Only 1 out of 10 instances solved to optimality

Table 9.7: Instances from benchmark set Real: Average runtimes in seconds.

data set	P	$\overline{\lvert s\rvert}$	Chin	Deo	AE	IR	Hung	A*	ILP
$ds0$	HKH	60.62	0.012	0.015	0.012	0.026	0.017	**0.011**	33.94
$ds1$	HKH	64.00	**0.012**	0.017	0.013	0.032	0.019	0.015	332.05
$ds1$	HKSH	63.93	**0.011**	0.021	0.017	0.033	0.017	**0.011**	311.31
$ds1$	HKSTH	63.87	0.016	0.022	0.019	0.043	0.024	**0.012**	287.76
$ds2$	HKSH	79.60	0.015	0.020	0.016	0.030	0.052	**0.012**	93.60
$ds2$	HKSTH	77.87	**0.013**	0.018	0.016	0.030	0.051	**0.013**	157.26
$ds3$	HKH	103.90	0.018	0.026	0.019	0.138	0.188	**0.014**	834.19
$ds4$	DGGG	43.87	**0.012**	0.022	0.014	0.023	0.049	**0.012**	36.62

9.4 Computational Results for the m–CLCS Problem

In this section, we discuss the results of our computational studies for the general m–CLCS problem. All experiments were conducted with the benchmark instances presented in Section 9.2. Since BS takes several configuration parameters, which affect outcome and computation time, finding a proper setting is an important task. To this end, we first did preliminary experiments on BS with various parameter settings before we conducted our main studies.

9.4.1 Comparison between different Beam Search Configurations

In this series of experiments, we evaluate the performance of different beam search configurations and study the impact of beam width β and the filtering parameter k_{best} on solution quality and computation time. We compare four beam search configurations:

- BS guided by upper bound UB presented in formula 5.3 – henceforth labeled by BS-Ub,
- the configuration of BS guided by the probability heuristic H presented in Section 5.3 – henceforth labeled by BS-Prob,
- the configuration whose search is guided by an expected length calculation according to formula 5.12) – henceforth labeled by BS-Ex,
- the configuration of BS guided by the pattern ratio heuristic R presented in Section 5.5 – henceforth labeled by BS-Pat.

First, each of the configurations was executed with the following values of the beam size $\beta \in \{1, 10, 100, 1000, 2000, 5000\}$, fixed $k_{\text{best}} = 100$, $k_{\text{ext}} = \infty$ and Prune$(\cdot)$ procedure was active (set up in this way was the results of some preliminary runs) on all 900 instances. Figure 9.4 presents the solution quality and computation time averaged over all instances. To study the effects of filtering, each of the BS configurations was then executed for $k_{\text{best}} \in \{0, 10, 50, 100, 200\}$ ($k_{\text{best}} = 0$ means no filtering at all), fixed $\beta = 2000$, $k_{\text{ext}} = \infty$ and Prune$(\cdot)$ procedure was active. Figure 9.5 presents the solution quality and computation time averaged over all 900 instances. We made the following observations on these results:

- As we might expect, the larger beam width β, the higher is the average solution quality on cost of longer computation time. For $\beta \leq 100$, all four BS configurations are executed within a few seconds. For larger β, BS-Prob and BS-Ub are processed significantly faster than BS-Ex and BS-Pat. This is because these configurations utilize heuristics that are very quickly computed.
- Overall, BS-Ex and BS-Prob clearly perform significantly better than the other two compared BS configurations, regardless of which parameter setting for β and k_{best} is considered. We emphasize that BS-Ex with $\beta = 2000$ and $\beta = 5000$ deliver statistically equal solutions qualities; however, the latter needs much more time for its execution.

  The average solution quality from BS-Ex even with just $\beta = 100$ is higher than the average solution quality of BS-Ub with $\beta = 5000$. The ranking of the BS configurations w.r.t. their average solution quality also stays the same for any β and k_{best}: from best to worst performing configuration concerning solution qualities, the ranking is as follows: BS-Ex, BS-Prob, BS-Ub and BS-Pat.

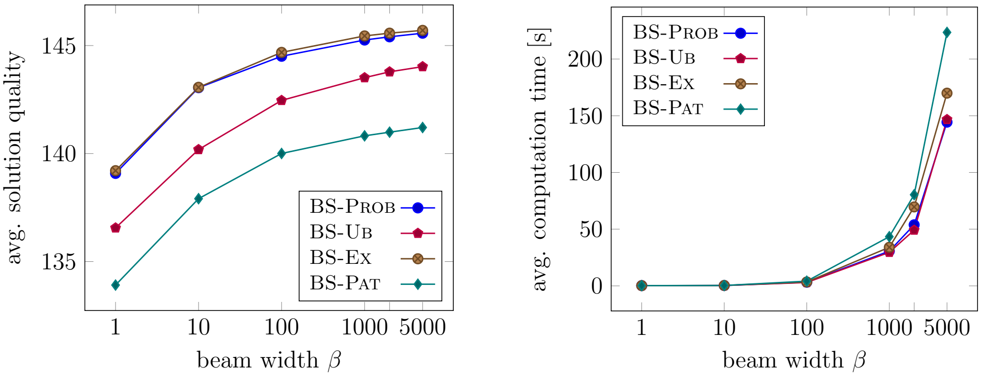


Figure 9.4: Results of BS with $k_{\text{best}} = 100$ and varying β.

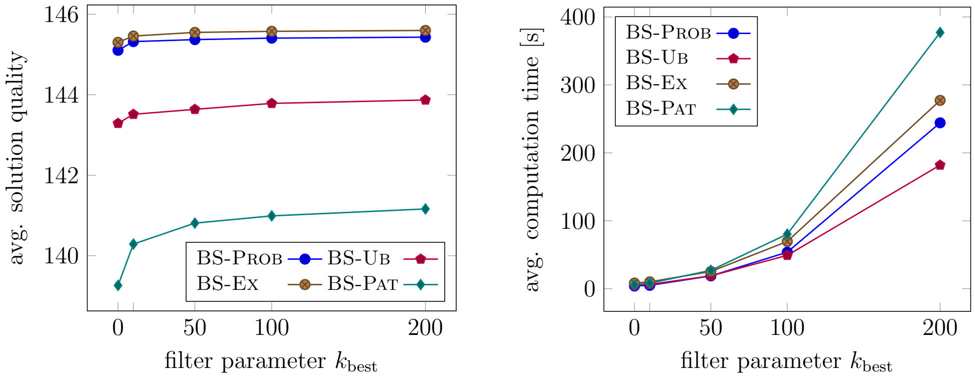


Figure 9.5: Results of BS with $\beta = 2000$ and varying k_{best}.

- An increase of parameter k_{best} leads to only slightly better results while computation times rise rather quickly. This is because the Filter() procedure takes for its execution almost quadratic time in terms of the number of nodes in V_{ext} when k_{best} is high. The largest difference in terms of solution qualities can be spotted between $k_{\text{best}} = 0$, i.e. no filtering, and $k_{\text{best}} = 10$. In time-sensitive applications, filtering with only small k_{best} can, therefore, present a good trade-off between quality of the results and computation time.

9.4.2 The Comprehensive Numerical Results

We compare the results of

- the Approximation algorithm from [33], henceforth labeled by Approx,
- the Greedy heuristic for the CLCS presented in Section 4.2, henceforth labeled by Greedy, and

- the four beam search configurations: BS-Ub, BS-Prob, BS-EX and BS-Pat.

From the last paragraph, where the results of several different configurations for β and k_{best} executed, we decided to use the following settings for each of the four configurations of beam search: $\beta = 2000$, $k_{best} = 100$, whereas Prune($\cdot$) procedure is always included (initial solution is obtained as the outcome of Greedy algorithm) and $k_{\text{ext}} = \infty$, i.e., no pre-reduction of the extension set V_{ext} is performed. Moreover, the results produced by A* from Section 7 are used here to show the solutions qualities of optimal values (if any) and the efficiency of the heuristic search.

The numerical results are presented in Tables 9.8– 9.12. These results are divided into different groups w.r.t. different values for ratio p'. For each of the algorithms we present the solution quality and times averaged over 10 instances from the corresponding group. The best results obtained among the competitor algorithms are shown in bold font. The first three columns present the properties of the instance set, the next three blocks are reserved to report the results of Approx, Greedy, and BS algorithm, respectively. The block which presents the BS results is additionally subdivided into four blocks, reporting the results of the four different BS configurations: BS-Ub, BS-Prob, BS-EX and BS-Pat, respectively. The last block consists of two columns reporting the number of the instances solved to optimality by means of the A* search and the corresponding average computation time required to reach the optimum, respectively.[3] A mark "–" was used for the avg. time for those cases where none of the instances was solved to optimality. If all of the 10 instances of a group were solved to optimality, an asterisk is used to mark that the result of an algorithm reaches the optimal value.

From the numerical results, we made the following observations:

- For $p' \leq \frac{1}{20}$, BS-Ex delivers in most cases significantly better results than the other BS configurations, Greedy and Approx algorithm. Involving a higher beam width β for each of the BS configurations pays off at the cost of longer computation time. BS-Ex loses its efficiency when the length of P becomes larger. That is mainly due to the fact that the larger the length of pattern P, the higher the similarity between input strings (supposing that P is a substring of each of the input strings), i.e., these strings become highly dependent. With a higher dependency, the guidance of EX becomes weaker. Note that this behaviour was already noticed in previous work [25].

- BS-Pat is inferior to the other three BS configurations in terms of solution quality on almost all instances. It seems that Pattern Ratio Heuristic suffers from a large amount of ties occurred in the search.

[3] For those instances with $n = 100$, we also incorporate an A* with $h(v)$ function compound as the minimum of UB_1 and UB^{CLCS}, but only one additional instance was solved to optimality and no significant change in the runtimes.

- The results of BS-Ub are comparable with the results of BS-Ex only when $|P|/n$ and n are both small. It is because UB is well–performing and especially tight for smaller instances.

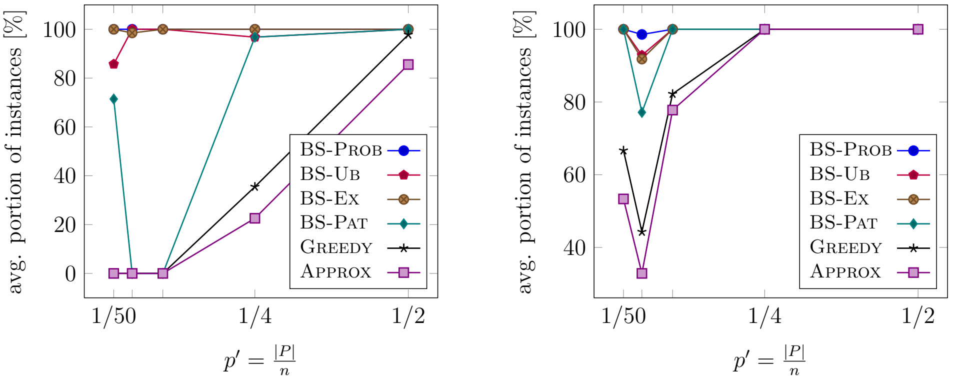
- BS-Prob gains terrain over BS-Ex when the length of P gets larger and $|\Sigma|$ gets smaller. For example, if $p' = \frac{1}{4}$ and $|\Sigma| = 4$, BS-Prob delivers better results than the other competitors.

- When p' is large ($p' \geq \frac{1}{4}$), the differences between solution quality obtained by BS and Greedy or Approx algorithms is small. That is mainly because a long pattern P yields a restricted search space. Thus, the instances become easier to solve.

- The number of instances solved to optimality increases as the ratio between $|P|$ and n increases. Also note that all the instances with $p' = \frac{|P|}{n} = \frac{1}{2}$ are solved to proven optimality. On this subset of instances, most of the competitor algorithms produce results in a quality that is nearly the optimum. Concerning the ease of solving, the next are the instances with $p' = \frac{1}{4}$ where A* could solve 131 out of 200 instances to proven optimum. Note that when $|\Sigma| = 20$ and p' is large $\left(p' \geq \frac{1}{4}\right)$, the search space is highly restricted so that the only feasible solution found was string P, which is, by that, the proven optimum.

- Concerning runtimes of the algorithms, both Approx and Greedy algorithms run in a fraction of second while BS takes more time due to additional procedures executed and significantly larger amount of node expansions involved. Interestingly, due to restriction in the search space, the runtimes of all the algorithms decrease when the ratio between $|P|$ and n increases.

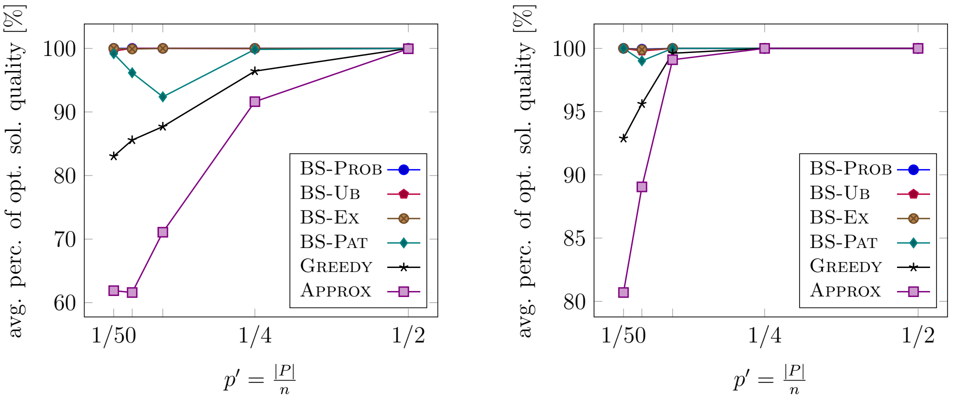
- On those instances where a proven optimal solution is found by the A* algorithm, we compare the qualities of the heuristic solutions obtained by our heuristic approaches with the quality of the proven optimum values. Figure 9.7 shows for each heuristic algorithm the avg. percentage of the obtained solution qualities in comparison to the optimal solution quality (reached by the A*), averaged over the set of all solved instances that are grouped by different value of p' (displayed by x-axis). Figure 9.6 shows for each algorithm the proportion of the instances solved to optimality by A* that could also reach the quality of the optimal solutions by the respective algorithm. It can be noticed that BS delivers strong results, BS-Prob, BS-Ub and BS-Ex configurations reach at least 98% of the optimum quality on those instances where optimality could be proven. In overall, BS-Prob and BS-Ex are performing best. The heuristic solutions obtained by BS-Prob coincide to the (quality of) optimal solution for all but one case (an instance out of the 10 from the set $m = 10$, $n = 500$, $|\Sigma| = 20$, $p' = \frac{1}{20}$) where A* successfully proves optimality. For $p' \in \left\{\frac{1}{50}, \frac{1}{20}, \frac{1}{10}\right\}$ and $|\Sigma| = 4$, Greedy and Approx do not deliver any optimal solution, the superiority of BS is clearly noticeable.



(a) $|\Sigma| = 4$

(b) $|\Sigma| = 20$

Figure 9.6: Percentage of instances solved to optimality.



(a) $|\Sigma| = 4$

(b) $|\Sigma| = 20$

Figure 9.7: Average fraction (in percent) of the length of heuristic solutions with respect to the length of the A* solutions.

Table 9.8: Instances with $p' = \frac{|P|}{n} = \frac{1}{50}$.

			Approx		Greedy		BS-Ub		BS-Prob		BS-Ex		BS-Pat		A*	
$\lvert\Sigma\rvert$	m	n	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	#	$\bar{t}$
4	10	100	20.9	< 0.1	28.4	< 0.1	34.2	22.9	**34.3**	20	**34.3**	20.8	33.8	26.2	7	290.4
4	10	500	117.8	< 0.1	159.6	< 0.1	180.4	149.1	183.6	157.3	**184.8**	143.2	177.7	174.7	0	-
4	10	1000	239.2	0.1	327.6	0.1	363.5	284.7	372.4	372.3	**376.3**	434.2	354.7	428.2	0	-
4	50	100	17.4	< 0.1	20.3	< 0.1	24.1	15.5	**24.2**	12.1	**24.2**	16.7	24	22	0	-
4	50	500	109.3	0.1	125.3	0.1	137.3	106	140.4	138.1	**141.8**	131.8	136.3	147.2	0	-
4	50	1000	228.9	0.5	263.6	0.5	279.8	257.9	288.7	231.1	**290.4**	340.0	277.2	251.7	0	-
4	100	100	17.0	< 0.1	18.9	< 0.1	**21.9**	16.1	**21.9**	16.3	**21.9**	14	21.6	19.4	0	-
4	100	500	108.1	0.2	118.5	0.2	128.4	135	131	118.2	**132.0**	115.2	127.6	160.2	0	-
4	100	1000	225.1	0.9	248.7	0.8	262.4	287.6	270.5	236.6	**272.1**	329.9	261.6	282	0	-
20	10	100	4.3	< 0.1	6.2	< 0.1	***7.9**	0.1	***7.9**	0.1	***7.9**	0.1	***7.9**	0.1	10	< 0.1
20	10	500	23.8	< 0.1	40.7	< 0.1	48.9	104.5	49.7	137	**50.4**	183.8	41.9	221.7	0	-
20	10	1000	48.9	0.1	82.5	0.1	97.7	246.8	102.0	280.7	**104.9**	344.3	85.6	551.4	0	-
20	50	100	2.8	< 0.1	***3.1**	< 0.1	***3.1**	< 0.1	***3.1**	< 0.1	***3.1**	< 0.1	***3.1**	< 0.1	10	< 0.1
20	50	500	20.0	0.1	24	0.1	28.3	49	**28.8**	46.8	**28.8**	100.3	26	135.5	0	-
20	50	1000	42.6	0.5	53.7	0.5	59.6	152.5	61.4	158.1	**62.3**	245.4	55.1	211.2	0	-
20	100	100	2.3	< 0.1	***2.4**	< 0.1	***2.4**	< 0.1	***2.4**	< 0.1	***2.4**	< 0.1	***2.4**	< 0.1	10	< 0.1
20	100	500	18.5	0.3	22	0.3	24.7	60.9	**25.2**	62.6	25.0	118.5	22.8	82.7	0	-
20	100	1000	41.1	1	49	0.8	52.8	166.2	54.7	188.6	**55.0**	334.8	50	342.7	0	-

Table 9.9: Instances with $p' = \frac{|P|}{n} = \frac{1}{20}$.

			Approx		Greedy		BS-Ub		BS-Prob		BS-Ex		BS-Pat		A*	
$\lvert\Sigma\rvert$	m	n	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	$\overline{\lvert s\rvert}$	$\bar{t}$	#	$\bar{t}$
4	10	100	21.4	< 0.1	30.4	< 0.1	**34.5**	19.2	**34.5**	16.8	**34.5**	21.7	33.4	25.6	3	332.8
4	10	500	119.7	< 0.1	159.6	< 0.1	181.7	130.1	184.2	163.7	**185.1**	179.8	173.3	192.1	0	-
4	10	1000	244.4	0.1	328.3	0.1	365.7	288.5	372.7	346.7	**374.1**	339.2	343.8	391	0	-
4	50	100	18.7	< 0.1	21.6	< 0.1	24.3	11.5	24.7	13.3	**24.9**	15.1	24	19.8	0	-
4	50	500	111.1	0.1	126.8	0.1	137.9	98.5	141.2	109.4	**142.2**	115.4	134.2	162.8	0	-
4	50	1000	232.7	0.5	265.9	0.4	281	226.4	290.1	267.6	**291.3**	289.4	273	366.4	0	-
4	100	100	17.6	< 0.1	18.4	< 0.1	22.3	11.6	22.4	9.6	**22.5**	13.60	21.9	19.7	0	-
4	100	500	109.4	0.2	117.6	0.2	128.9	101.2	131.9	86.2	**132.4**	119.3	126.6	156	0	-
4	100	1000	227.5	0.8	250.3	1	263.7	244.2	272.0	218.1	**273.0**	232.2	259.2	301.8	0	-
20	10	100	6	< 0.1	7.1	< 0.1	***7.3**	< 0.1	***7.3**	< 0.1	***7.3**	< 0.1	***7.3**	< 0.1	10	< 0.1
20	10	500	30.2	< 0.1	40.2	< 0.1	46.6	16.9	**47.0**	17.5	46.3	60.0	44.7	57	10	332.1
20	10	1000	56.6	0.1	81.2	0.1	95.7	37.9	**97.8**	45.5	95.4	185.4	87.9	146.3	0	-
20	50	100	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	10	< 0.1
20	50	500	26.9	0.1	28.2	0.2	***29.9**	1.8	***29.9**	1.7	***29.9**	1.3	***29.9**	1.5	10	1.2
20	50	1000	53.1	0.5	58.1	0.5	62.4	17.6	**62.7**	17	62.5	8.6	60.4	34.4	0	-
20	100	100	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	***5.0**	< 0.1	10	< 0.1
20	100	500	26.1	0.2	26.4	0.3	***27.3**	0.3	***27.3**	0.2	***27.3**	0.3	***27.3**	0.3	10	0.3
20	100	1000	52	1	54.8	0.9	57.2	14	***57.3**	13.6	***57.3**	9.4	56.4	17.7	10	86.0

Table 9.10: Instances with $p' = \frac{|P|}{n} = \frac{1}{10}$.

			Approx		Greedy		BS-Ub		BS-Prob		BS-Ex		BS-Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	#	$\bar{t}$
4	10	100	22.9	< 0.1	30.1	< 0.1	**34.6**	14.4	**34.6**	17.4	34.3	20.4	32.1	23	8	269.1
4	10	500	121.4	< 0.1	163.2	< 0.1	182.2	97.6	**185.0**	137	184.8	143.2	165.9	193.9	0	-
4	10	1000	245.5	0.1	328.7	0.1	365	212	375.8	240.5	**376.3**	434.8	330.4	391.7	0	-
4	50	100	19.8	< 0.1	22	< 0.1	24.9	10.1	**25.0**	11.2	24.3	19.6	23.5	19.9	0	-
4	50	500	114.2	0.1	129.5	0.1	138.7	102.4	**142.9**	99.6	141.8	131.8	131.2	145.9	0	-
4	50	1000	233.5	0.4	266.7	0.4	279.6	199	289.2	200.6	**290.4**	340.0	266	351.7	0	-
4	100	100	18.9	< 0.1	20.4	< 0.1	**23.0**	8.8	**23.0**	8.7	21.9	17.0	21.5	19.3	3	265.1
4	100	500	111.3	0.2	121.8	0.2	129.2	63.2	**133.3**	78.5	132.0	115.6	124.3	163.8	0	-
4	100	1000	230.3	0.9	253.1	0.8	262.3	122.7	270.9	183.3	**272.1**	329.9	255.2	316.3	0	-
20	10	100	***10.2**	< 0.1	10.1	< 0.1	***10.2**	< 0.1	***10.2**	< 0.1	***10.2**	< 0.1	***10.2**	< 0.1	10	< 0.1
20	10	500	51	< 0.1	52.5	< 0.1	***53.1**	< 0.1	***53.1**	< 0.1	***53.1**	< 0.1	***53.1**	< 0.1	10	< 0.1
20	10	1000	101	0.1	103.9	0.1	***105.4**	0.1	***105.4**	0.1	***105.4**	0.1	***105.4**	0.1	10	0.1
20	50	100	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	10	< 0.1
20	50	500	***50.0**	0.1	***50.0**	0.2	***50.0**	0.1	***50.0**	0.1	***50.0**	0.1	***50.0**	0.1	10	0.2
20	50	1000	***100.0**	0.5	***100.0**	0.4	***100.0**	0.5	***100.0**	0.5	***100.0**	0.5	***100.0**	0.4	10	0.5
20	100	100	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	***10.0**	< 0.1	10	< 0.1
20	100	500	***50.0**	0.3	***50.0**	0.3	***50.0**	0.3	***50.0**	0.3	***50.0**	0.3	***50.0**	0.2	10	0.3
20	100	1000	***100.0**	1	***100.0**	1.1	***100.0**	0.8	***100.0**	0.8	***100.0**	1.1	***100.0**	1	10	0.9

Table 9.11: Instances with $p' = \frac{|P|}{n} = \frac{1}{4}$.

			Approx		Greedy		BS-Ub		BS-Prob		BS-Ex		BS-Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	#	$\bar{t}$
4	10	100	28.6	< 0.1	32.3	< 0.1	***34.5**	1.1	***34.5**	0.9	***34.5**	1.0	***34.5**	1.5	10	0.2
4	10	500	134.3	< 0.1	159.8	< 0.1	179.3	45.6	**182.4**	48.8	181.1	98.0	168.6	97	1	660.8
4	10	1000	264.7	0.1	317.2	0.1	350.3	76.8	**361.7**	108	361.4	249.4	330.8	220.2	0	-
4	50	100	26.4	< 0.1	26.9	< 0.1	***27.5**	< 0.1	***27.5**	< 0.1	***27.5**	< 0.1	***27.5**	< 0.1	10	< 0.1
4	50	500	130.1	0.1	139.5	0.1	146.2	33.6	**148.3**	28	146.3	19.9	142.7	55.9	0	-
4	50	1000	257.4	0.5	277.3	0.5	291.9	73.6	**296.4**	63.6	289.5	41.1	284.2	107.6	0	-
4	100	100	25.9	< 0.1	26.2	< 0.1	***26.5**	< 0.1	***26.5**	< 0.1	***26.5**	< 0.1	***26.5**	< 0.1	10	< 0.1
4	100	500	128.9	0.2	135.8	0.2	140.4	24.6	**140.8**	34.8	140.3	17.4	137.3	45.9	0	-
4	100	1000	256.4	0.8	270.7	0.7	279.7	56.4	**282.5**	73.4	279.0	40.4	273.3	122	0	-
20	10	100	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	10	< 0.1
20	10	500	***125.0**	< 0.1	***125.0**	< 0.1	***125.0**	< 0.1	***125.0**	< 0.1	***125.0**	< 0.1	***125.0**	< 0.1	10	< 0.1
20	10	1000	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	10	0.1
20	50	100	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	10	< 0.1
20	50	500	***125.0**	0.1	***125.0**	0.1	***125.0**	0.2	***125.0**	0.2	***125.0**	0.1	***125.0**	0.1	10	0.1
20	50	1000	***250.0**	0.5	***250.0**	0.4	***250.0**	0.4	***250.0**	0.5	***250.0**	0.5	***250.0**	0.5	10	0.5
20	100	100	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	***25.0**	< 0.1	10	< 0.1
20	100	500	***125.0**	0.3	***125.0**	0.3	***125.0**	0.3	***125.0**	0.3	***125.0**	0.2	***125.0**	0.2	10	0.3
20	100	1000	***250.0**	1	***250.0**	1	***250.0**	1.1	***250.0**	1.1	***250.0**	0.8	***250.0**	1.1	10	1.0

Table 9.12: Instances with $p' = \frac{|P|}{n} = \frac{1}{2}$.

			Approx		Greedy		BS-Ub		BS-Prob		BS-Ex		BS-Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	$\overline{\|s\|}$	$\bar{t}$	#	$\bar{t}$
4	10	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
4	10	500	250.1	<0.1	***250.6**	<0.1	***250.6**	<0.1	***250.6**	<0.1	***250.6**	0.1	***250.6**	<0.1	10	<0.1
4	10	1000	500.1	0.1	501.5	0.1	***501.7**	0.1	***501.7**	0.1	***501.7**	0.1	***501.7**	0.1	10	0.1
4	50	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
4	50	500	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	10	0.1
4	50	1000	***500.0**	0.4	***500.0**	0.4	***500.0**	0.5	***500.0**	0.3	***500.0**	0.5	***500.0**	0.3	10	0.5
4	100	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
4	100	500	***250.0**	0.2	***250.0**	0.2	***250.0**	0.2	***250.0**	0.2	***250.0**	0.2	***250.0**	0.2	10	0.2
4	100	1000	***500.0**	1	***500.0**	0.9	***500.0**	1	***500.0**	0.8	***500.0**	1	***500.0**	0.8	10	0.8
20	10	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
20	10	500	***250.0**	<0.1	***250.0**	<0.1	***250.0**	<0.1	***250.0**	0.1	***250.0**	<0.1	***250.0**	<0.1	10	<0.1
20	10	1000	***500.0**	0.1	***500.0**	0.1	***500.0**	0.1	***500.0**	0.1	***500.0**	0.1	***500.0**	0.1	10	0.1
20	50	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
20	50	500	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	10	0.1
20	50	1000	***500.0**	0.5	***500.0**	0.5	***500.0**	0.4	***500.0**	0.4	***500.0**	0.5	***500.0**	0.4	10	0.5
20	100	100	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	***50.0**	<0.1	10	<0.1
20	100	500	***250.0**	0.2	***250.0**	0.3	***250.0**	0.3	***250.0**	0.2	***250.0**	0.2	***250.0**	0.2	10	0.3
20	100	1000	***500.0**	1	***500.0**	0.8	***500.0**	0.7	***500.0**	0.8	***500.0**	1	***500.0**	1.1	10	0.7

CHAPTER 10

Conclusions and Future Work

In this thesis we considered the variant of the CLCS problem with an arbitrary number of input strings, which is $\mathcal{NP}$–hard. A general search framework was presented to tackle the problem from where we derived various methods: a greedy heuristic to quickly find solutions of reasonable quality, a heuristic search presented by a general beam search framework, and an exact search established by means of an A* algorithm.

Concerning the exact solving, the A* search utilizing the known upper bounds for the LCS problem was able to solve more than a half of our instances generated for the m–CLCS problem to proven optimality. Moreover, all the 2–CLCS problem instances, the randomly generated ones and the instances from practice, were solved within a fraction of a second. A* search was compared to other algorithms from literature, specially developed for the classical 2–CLCS problem. The effectivity of the A* search was demonstrated by an exhaustive experimental evaluation, where we showed that the runtimes of our A* approach are about one to two orders of magnitude shorter than those of the best competitor algorithm. Moreover, it was shown that the A* search scales particularly well, its performance does not degrade much with an increase of the instance size, which is not the case for the other competitors.

Concerning heuristic solvers, the developed greedy heuristic was able to quickly construct a solution for each of our instances. In most cases, it took less than a second and yielded a significantly better solution quality than the outcome of the known approximation method from literature. In order to produce high-quality solutions, we developed the general BS framework. By extending promising heuristics for the LCS problem, three new heuristic estimators were proposed for the m–CLCS problem. The computational studies showed, that both our expected-length calculation heuristic ($\mathrm{EX}^{\mathrm{CLCS}}$) and the probability–based heuristic (H) provide effective guidance. Moreover, the beam search configuration guided by heuristic H showed its effectiveness by reaching optimal solutions for almost all those benchmarks where A* search was able to prove optimality.

Overall, we conclude that the developed search framework composed of greedy heuristic, beam search and A* search provides effective methods for solving the m–CLCS problem and could be used as set of tools to detect similarities and relations between arbitrary large molecular structures that appear in bioinformatics.

For future work, our search framework could be easily adapted to solve related variants of the LCS problem. For example, the *restricted* LCS (RLCS) problem [32, 13] which requires that a given pattern string is not part of the solution, could be dealt with algorithms similar to those proposed in this thesis. Also more general variants of the m–CLCS problem where an arbitrary number k of pattern strings are given in input – labeled by (k, m)–CLCS problem – could be solved by means of the derived m–CLCS search framework, with only a few minor adaptions required. Moreover, the presented A* search could be extended to an anytime algorithm so that while running an exact search, high-quality heuristic solutions are obtained along the way. As a result, A* search could provide solutions even when proving optimality is not feasible (e.g. due to limited time or memory resources). Concerning heuristics, we found a very effective search guidance for the m–CLCS problem. Still, it might be worth experimenting with other heuristics.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proceedings of FOCS 2015 – 56th Annual Symposium on Foundations of Computer Science*, pages 59–78. IEEE Press, 2015.

[2] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi. Repetition-free longest common subsequence. *Discrete Applied Mathematics*, 158(12):1315–1324, 2010.

[3] A. N. Arslan and Ö. Eğecioğlu. Algorithms for the constrained longest common subsequence problems. *International Journal of Foundations of Computer Science*, 16(06):1099–1109, 2005.

[4] T. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.

[5] C. Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2(4):353–373, 2005.

[6] C. Blum. Beam–ACO for the longest common subsequence problem. In *Proceedings of CEC 2010 – the IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE Press, 2010.

[7] C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009.

[8] C. Blum and P. Festa. *Metaheuristics for String Problems in Bio-informatics*. John Wiley & Sons, 2016.

[9] C. Blum and G. R. Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing, 2016.

[10] P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.

[11] G. S. Brodal, M. Kutz, K. Kaligosi, and I. Katriel. Faster algorithms for computing longest common increasing subsequences. *Journal of Discrete Algorithms*, 9(4):314–325, 2011.

[12] K.-M. Chao and L. Zhang. *Sequence Comparison – Theory and Methods.* Springer, London, UK, 2009.

[13] Y.-C. Chen and K.-M. Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, 21(3):383–392, 2011.

[14] F. Chin and C. K. Poon. Performance analysis of some simple heuristics for computing longest common subsequences. *Algorithmica*, 12(4-5):293–311, 1994.

[15] F. Y. Chin, A. De Santis, A. L. Ferrara, N. Ho, and S. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4):175–179, 2004.

[16] F. Y. Chin, N. Ho, T. Lam, P. W. Wong, and M. Chan. Efficient constrained multiple sequence alignment with performance guarantee. *Journal of Bioinformatics and Computational Biology*, 3(1):1–8, 2005.

[17] Y. Choi and W. Szpankowski. Pattern matching in constrained sequences. In *International Symposium on Information Theory, ISIT 2007*, pages 2606–2610. IEEE, 2007.

[18] S. R. Chowdhury, M. Hasan, S. Iqbal, and M. S. Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, 129(4):329–340, 2014.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, third edition, 2009.

[20] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[21] S. Deorowicz. Fast algorithm for constrained longest common subsequence problem. *Theoretical and Applied Informatics*, 19(2):91–102, 2007.

[22] S. Deorowicz and J. Obstój. Constrained longest common subsequence computing algorithms in practice. *Computing and Informatics*, 29(3):427–445, 2012.

[23] M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. An A* search algorithm for the constrained longest common subsequence problem. Technical Report AC-TR-20-004, Algorithms and Complexity Group, TU Wien, 2020. under review.

[24] M. Djukanovic, G. Raidl, and C. Blum. Exact and heuristic approaches for the longest common palindromic subsequence problem. In *Proceedings of LION 12 – the 12th International Conference on Learning and Intelligent Optimization*, volume 11353, pages 199–214. Springer, 2018.

[25] M. Djukanovic, G. Raidl, and C. Blum. A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In *Proceedings of LOD 2019 – the 5th International Conference on Machine Learning, Optimization, and Data Science*, volume 11943, pages 154–167. Springer, 2019.

[26] M. Djukanovic, G. Raidl, and C. Blum. Heuristic approaches for solving the longest common squared subsequence problem. In *Proceedings of EUROCAST 2019 - the 17th International Conference on Computer Aided Systems Theory, Part I*, volume 12013, pages 429–437. Springer, 2019.

[27] M. Dorigo, M. Birattari, and T. Stützle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[28] T. Easton and A. Singireddy. A specialized branching and fathoming technique for the longest common subsequence problem. *International Journal of Operations Research*, 4(2):98–104, 2006.

[29] T. Easton and A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 14(3):271–283, 2008.

[30] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming ii: Convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

[31] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

[32] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein. Restricted LCS. In *Proceedings of SPIRE 2010 – the 17th International Symposium on String Processing and Information Retrieval*, pages 250–257, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[33] Z. Gotthilf, D. Hermelin, and M. Lewenstein. Constrained LCS: hardness and approximation. In *Proceedings of CPM 2008 – the 19th Annual Symposium on Combinatorial Pattern Matching*, pages 255–262. Springer, 2008.

[34] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.

[35] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[36] D. He and A. N. Arslan. A space-efficient algorithm for the constrained pairwise sequence alignment problem. *Genome Informatics*, 16(2):237–246, 2005.

[37] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[38] W. J. Hsu and M. W. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59, 1984.

[39] K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of ICS 2004 – the 8th International Computer Symposium*, pages 1006–1011. IEEE Press, 2004.

[40] S.-H. Hung, C.-B. Yang, and K.-S. Huang. A diagonal-based algorithm for the constrained longest common subsequence problem. In *Proceedings of ICS 2018 – the 23rd International Computer Symposium*, pages 425–432. Springer Singapore, 2019.

[41] C. Iliopoulos, M. S. Rahman, M. Voráček, and L. Vagner. Finite automata based algorithms on subsequences and supersequences of degenerate strings. *Journal of Discrete Algorithms*, 8(2):117 – 130, 2010.

[42] C. S. Iliopoulos and M. S. Rahman. New efficient algorithms for the LCS and constrained LCS problems. *Information Processing Letters*, 106(1):13–18, 2008.

[43] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. In *Automata, Languages and Programming*, pages 191–202. Springer Berlin Heidelberg, 1994.

[44] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.

[45] T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In *Proceedings of CPM 2000 – 11th Annual Symposium on Combinatorial Pattern Matching*, pages 154–165. Springer, 2000.

[46] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[47] J. B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

[48] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[49] M. Lozano and C. Blum. A hybrid metaheuristic for the longest common subsequence problem. In *Proceedings of HM 2010 – the 7th International Workshop on Hybrid Metaheuristics*, volume 6373 of *LNCS*, pages 1–15. Springer, 2010.

[50] C. L. Lu and Y. P. Huang. A memory-efficient algorithm for multiple sequence alignment with constraints. *Bioinformatics*, 21(1):20–30, 2005.

[51] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[52] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980.

[53] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097 – 1100, 1997.

[54] S. R. Mousavi and F. S. Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012.

[55] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18(2):171–179, 1982.

[56] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:297–307, 1988.

[57] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., USA, 1982.

[58] C. M. Rands, S. Meader, C. P. Ponting, and G. Lunter. 8.2% of the human genome is constrained: Variation in rates of turnover across functional element classes in the human lineage. *PLoS Genetics*, 10(7):e1004525, 2014.

[59] S. J. Shyu and C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers & Operations Research*, 36(1):73–91, 2009.

[60] J. Storer. *Data Compression: Methods and Theory.* Computer Science Press, MD, USA, 1988.

[61] F. S. Tabataba and S. R. Mousavi. A hyper-heuristic for the longest common subsequence problem. *Computational Biology and Chemistry*, 36:42–54, 2012.

[62] E.-G. Talbi. *Metaheuristics: From Design to Implementation.* John Wiley & Sons, Hoboken, NJ, USA, 2009.

[63] Y. T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.

[64] J. M. Valente and R. A. Alves. Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. *Computers & Industrial Engineering*, 48(2):363–375, 2005.

[65] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[66] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[67] Q. Wang, M. Pan, Y. Shang, and D. Korkin. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *Proceedings of AAAI 2010 – the 24th AAAI Conference on Artificial Intelligence*, 2010.

[68] L. A. Wolsey. *Integer programming*, volume 52. John Wiley & Sons, 1998.

[69] J. Yang, Y. Xu, G. Sun, and Y. Shang. A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):862–870, 2012.