TU WIEN Informatics

Exact and Heuristic Approaches for Solving String Problems from Bioinformatics

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften

by

Marko Djukanovic, MSc
Registration Number 01652659

to the Faculty of Informatics

at the TU Wien

Advisor: Günther Raidl, Ao.Univ.Prof. Dipl.-Ing.Dr.techn.
Second advisor: Christian Blum, Ph.D., senior research scientists

The dissertation has been reviewed by:

Paola Festa	Vladimir Filipović

Vienna, 24th December, 2020

Marko Djukanovic

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

Erklärung zur Verfassung der Arbeit

Marko Djukanovic, MSc

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, 24. Dezember 2020

Marko Djukanovic

Acknowledgements

First and foremost I want primary to thank my advisor Günther Raidl who gave me the opportunity to do these doctoral studies. I am grateful for the fruitful collaboration, suggestions, his patience and enthusiasm to teach, and for his guidance towards this point. Moreover, I want to express my deep gratitude to my co-advisor Christian Blum who was always there to help me with long and serious discussions and revising my manuscripts. I was quite pleased to visit you in Barcelona and to stay there for a few months. This visit initiated studying new research topics that led to common publications. I want to emphasize that this thesis was funded by the Doctoral Program Vienna Graduate School on Computational Optimization (VGSCO), Austrian Science Foundation Project No. W1260-N35. The VGSCO organized many interesting courses from the wide area of optimization, which increase my level of expertise significantly. Also, I express gratitude to the many events organized by VGSCO, such as the workshops and excellent retreats. Many thanks go to my former and current colleagues at the Algorithms and Complexity Group for the friendly environment, the nice discussions from which I learned a lot and spread my viewpoints. Also, many thanks to the external reviewers of my thesis, Paola Festa and Vladimir Filipović, for their time and effort invested to evaluate this thesis.

Last but not least, the greatest thanks go to my family: my wife Jovanka, my parents, and my sister Marina for their ongoing support, encouragement that gave me the confidence and motivation to overcome all obstacles and made this work possible.

Abstract

This thesis provides several new algorithms for solving prominent string problems from the literature, most of these being variants of the well-known longest common subsequence (LCS) problem. Given a set of input strings, a longest common subsequence is a string of maximum length that can be obtained from each input string by removing letters, i.e., which is a common subsequence of all input strings. This is a combinatorial optimization problem, which can be solved efficiently for two strings but is $\mathcal{NP}$–hard and challenging to solve in practice for the general case of an arbitrary set of input strings. The LCS problem and related string problems make relationships among strings explicit and provide a range of important measures which serve for detecting similarities between molecules of various structures, for example. Finding similarities between molecules plays an important role in bioinformatics helping in understanding complex biological processes, identifying important motifs and clusters of molecules that possess similar behaviors. Other important applications lie, for example, in text processing. The Unix command *diff* and the Git version-control system are some examples where finding common text patterns quickly is important. String problems have attracted attention for more than 50 years due to their computational hardness, and various methods were derived and successfully applied.

The current work primarily focuses on $\mathcal{NP}$–hard variants of string problems, where previous exact approaches are of limited practical value, and therefore many heuristic methods had been suggested. On the one side, more effective exact approaches are proposed here, but on the other side also more effective heuristic methods that scale to large problem instances of practical interest. Moreover, we consider in particular also *anytime algorithms* that are in principle exact but can be terminated early and then yield promising approximate solutions.

Besides the basic LCS problem, we consider here the following important variants: the longest common palindromic subsequence problem, the arc-preserving LCS problem, the longest common square subsequence problem, the repetition-free LCS problem, and the constrained LCS problem.

Concerning heuristic approaches, we propose a general beam search framework in which also many previously described methods can be expressed. A rigorous experimental evaluation and comparison were done. In particular, new state-of-the-art results were obtained on various benchmark sets utilizing a novel heuristic guidance that approximates

the expected solution length of three different string problems. This guidance can more effectively lead the search towards promising search regions of the search space than formerly used functions. For solving the longest common square subsequence problem, in particular, we propose a hybrid of a Reduced Variable Neighborhood Search method and a Beam search technique.

Concerning exact techniques to solve the string problems, two kinds of methods are presented in this thesis: (*i*) pure exact methods based on A* search and (*ii*) anytime algorithms that build upon the A* search framework. An effective A* search is obtained primarily by utilizing an efficiently calculable and at the same time comparably tight upper bound function for the length of real optimal common subsequence. Besides exhibiting an excellent performance on the general LCS problem, experimental results indicate that this A* search is also able to outperform all previously published more specific exact algorithms for the special cases of the longest common palindromic subsequence problem and the constrained longest common subsequence problem with two input strings.

Concerning anytime algorithms, we first make use of the already derived A* search framework in the way that classical A* iterations are interleaved with beam search runs. The iterations are performed on the same state graph to avoid unnecessary expansions of redundant nodes in the search. This hybrid is able to meaningfully tackle large-sized problem instances that cannot be solved exactly with the A* search and yields, after early termination, also upper bounds in addition to the heuristic solutions. In order to improve the convergence of the hybrid, we further suggest another anytime algorithm variant in which the beam search part is replaced by a major iteration of the Anytime Column Search. The effectiveness of the two hybrids is experimentally compared on the basic LCS problem and the longest common palindromic subsequence problem. New state-of-the-art results are produced and better final optimality gaps were obtained by the latter hybrid, in comparison to a several state-of-the-art anytime algorithms from the literature. Optimality gaps obtained in practice for the considered string problems are, to the best of our knowledge, compared and reported for the first time in literature.

As an alternative exact approach, we further consider the transformation of LCS problem instances into Maximum Clique (MC) problem instance on the basis of so-called conflict graphs. In this way, state-of-the-art MC solving approaches can be utilized for solving the LCS problem instances. One of the best exact and one of the best heuristic MC solvers from the literature are used for this purpose. Due to the complexity of the transformation, the size of the conflict graph grows quickly in the LCS problem instance size. To address this issue, we present a conflict graph reduction technique based on suboptimality checks by making use of the best available lower and upper bounds of the problems. The high effectiveness of the reduction is demonstrated on the repetition-free LCS problem and the longest arc-preserving subsequence problem. In conjunction with the general-purpose mixed integer linear programming solver CPLEX, new state-of-the-art results are obtained on a wide range of benchmark instances. Some of the real-world longest common arc-preserving problem instances were solved exactly for the first time in the literature by applying this new technique.

Kurzfassung

In dieser Arbeit werden neue Algorithmen zur Lösung bedeutender Stringprobleme aus der Literatur präsentiert, die meisten davon Varianten des wohlbekannten *longest common subequence* (LCS) Problems. Bei einer Menge von Eingabestrings ist eine *longest common subequence* ein String von größter Länge, der durch das Löschen einzelner Zeichen aus jedem der Eingabestrings erzeugt werden kann. Einen solchen zu finden ist ein kombinatorisches Optimierungsproblem, welches zwar effizient für zwei Eingabestrings gelöst werden kann, im allgemeinen Fall einer beliebigen Anzahl von Strings aber ein NP-schweres Problem darstellt. Die Lösungen solcher LCS- und verwandter Stringprobleme zeigen Zusammenhänge zwischen Strings explizit auf und können beispielsweise als Maße zur Erkennung von Ähnlichkeiten von Molekülen unterschiedlicher Strukturen verwendet werden. Letzteres spielt eine wichtige Rolle in der Bioinformatik um komplexe biologische Prozesse zu verstehen, durch das Erkennen von Strukturmotiven und Clustern von Molekülen mit ähnlichem Verhalten. Eine andere Anwendung besteht in der Textverarbeitung. Das Unix Kommando *diff* und das Versionierungssystem Git sind Beispiele, bei denen das schnelle Auffinden gemeinsamen Textmuster wichtig ist. Stringprobleme erregen bedingt durch ihre rechnerische Schwere seit mehr als 50 Jahren Aufmerksamkeit, in denen eine Vielzahl von Methoden entwickelt und erfolgreich angewandt wurden.

Die derzeitige Forschung befasst sich primär mit NP-schweren Problemvarianten, bei denen bisherige exakte Verfahren begrenzten praktischen Wert haben und daher viele heuristische Methoden entwickelt wurden. In dieser Arbeit werden einerseits effektivere exakte Verfahren, andererseits effektivere heuristische Methoden vorgeschlagen, welche und zu großen Probleminstanzen von praktischem Interesse gut skalieren. Darüber hinaus betrachten wir *anytime* Algorithmen welche zwar prinzipiell exakte Verfahren sind aber frühzeitig abgebrochen werden können und dann eine näherungsweise optimale Lösung liefern.

Neben dem grundlegenden LCS Problem betrachen wir folgende wichtige Varienten davon: das *longest common palindromic subsequence* Problem, das *arc-preserving* LCS Problem, das *longest common square subsequence* Problem, das *repetition-free* LCS problem und das *constrained* LCS.

Auf der heuristischen Seite schlagen wir ein allgemeines *beam search* Gerüst vor, in welchem viele bisherige Ansätze ausgedrückt werden können. Eine umfassende experimentelle Evaluierung und Vergleiche wurden durchgeführt. Besonders hervorzuheben

sind neue führende Resultate auf verschiedenen Benchmark-Instanzen dreier unterschiedlicher Stringprobleme, welche mit einer neuartigen *guidance* Heuristik, die erwartete Lösungslänge approximiert, erzielt wurden. Diese *guidance* lenkt die *Suche* effektiver in vielversprechendere Bereiche des Suchraums als bisher verwendete Funktionen. Speziell für das *longest common square subsequence problem* schlagen wir einen hybriden Ansatz bestehend aus einer reduzierten variablen Nachbarschaftssuche und einer *beam search* vor.

Bezüglich exakter Methoden werden zwei Arten von Methoden in dieser Arbeit präsentiert: (*i*) rein exakte Methoden basierend auf der A*-Suche und (*ii*) *anytime* Algorithmen basierend auf der A*-Suche. Durch eine effizient berechenbare und zugleich enge obere Schranke für die Länge von optimalen *common subsequences* erhalten wir eine effektive A*-Suche. Abgesehen von einer exzellenten Performance auf dem allgemeinen LCS Problem zeigen experimentelle Resultate, dass diese A*-Suche auch bisher publizierte Resultate spezifischerer exakter Algorithmen für die Spezialfälle des *longest common palindromic subsequence* Problems und des *constrained longest common subsequence* Problems mit zwei Eingabestrings übertrifft.

Als *anytime* Algorithmus verwenden wir das bestehende Gerüst der A*-Suche und Verschränken diese mit *beam search* Läufen. Diese werden auf dem selben Zustandsgraphen wie die A*-Suche durchgeführt um redundante Expansionen von Knoten zu vermeiden. Dieser hybride Ansatz vermag große Probleminstanzen welche nicht exakt gelöst werden können in Angriff zu nehmen und liefert zusätzlich zu einer heuristische Lösung auch eine obere Schranke. Um die Konvergenz zu verbessern, schlagen wir zusätzlich einen *anytime* Algorithmus vor, bei dem der *beam search* Teil durch eine Hauptiteration einer *Anytime Column Search* ersetzt wird. Die Effektivität beider hybriden Ansätze wird experimentell auf dem klassischer LCS Problem und dem *longest common palindromic subsequence* Problem analysiert. Mit dem zweiten Ansatz finden wir neue führende Resultate und bessere Optimalitgarantien in Vergleich zu anderen *state-of-the-art anytime* Algorithmen aus der Literatur. Nach bestem Wissen wurden diese Optimalitgarantien für die betrachteten Stringprobleme zum ersten Mal verglichen und in der Literatur berichtet.

Als einen alternativen exakten Ansatz betrachten wir weiters die Transformation von LCS Probleminstanzen in *maximum clique* (MC) Probleminstanzen basierend auf dem sogenannten Konfliktgraphen. Dadurch können *state-of-the-art* MC Lösungsansätze verwendet werden um LCS Probleminstanzen zu lösen. Zu diesem Zwecke wird einer der besten exakten und einer der besten heuristischen MC Lösungsverfahren aus der Literatur verwendet. Die Größe des Konfliktgraphen wächst, der Komplexität der Transformation geschulde rapide an. Um dies abzumildern, präsentieren wir eine Reduktionstechnik für den Konfliktgraphen basierend auf Suboptimalitäts-Überprüfungen mittels der verfügbaren besten unteren und oberen Schranken einer Probleminstanz. Die hohe Effektivität dieses Ansatzes wird anhand des *repetition-free* LCS Problems und des *longest arc-preserving subsequence* Problems demonstriert. In Verbindungen mit dem CPLEX, einem Solver für ganzzahlige lineare Programme erlangen wir neue *state-of-the-art* Ergebnisse für einem

breiten Bereich von Benchmark-Instanzen. Einige praktische Benchmark-Instanzen des *longest common arc-preserving* Problems wurden mit Hilfe dieser Technik in der Literatur zum ersten Mal exakt gelöst.

Contents

CHAPTER 1

Introduction

In bioinformatics, strings are used for representing important biological molecules, such as DNA, RNA, and proteins. Finding similarities between molecular structures plays an important role in understanding biological processes. Such similarities can be expressed by considering *subsequences* that are common for all strings of 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 common measure of similarity can be obtained by considering the *longest common subsequence* (LCS) problem [128] which is a well-known (discrete) optimization problem stated as follows. Given a set of input strings, we aim at finding a longest possible subsequence that is common for all strings. For example, for the strings $\{\texttt{abbbcaab}, \texttt{abcccaa}\}$, an LCS is abcaa. At the first glance, this looks like a simple problem, but it is computationally challenging to solve in general since it belongs to the class of $\mathcal{NP}$–hard problems if an arbitrary set of input strings is given as an input. Only for a small number of input strings, the LCS can be solved efficiently in polynomial time, for example, by dynamic programming [128]. The LCS problem has many applications not only in molecular biology [96], but also in data compression [155], pattern recognition, file plagiarism check, text editing [112], among others.

In literature, there exists about a dozen of well-studied variants of the LCS problem that arise from practice, which have mostly been introduced within the last two decades. We will be considering some of the most important variants within this thesis. They are introduced in the next paragraphs. All these problem variants consist of the inclusion of additional constraints into the basic LCS problem. In that way, structurally different problems are obtained. In many cases, these problems are practically even more challenging to solve and ask for the development of algorithms that are different from those of the existing ones for the LCS problem literature.

The *longest common palindromic subsequence* (LCPS) problem [38] asks for a longest common subsequence that is at the same time a palindrome. A string is a palindrome if it is equal to its reverse string. For instance, if we are given two strings $s_1 =$

dabcbacbab, s_2 = abbcccbad, an LCPS is abcba. Palindromic subsequences are especially interesting in the biological context. Palindromic motifs are found in many genetic instructions such as DNA sequences. In the context of a research project on genome sequencing, it was discovered that many of the bases on the Y-chromosome are arranged as palindromes [117]. Biologists believe that identifying palindromic subsequences of DNA sequences may help to understand genomic instability [36, 159].

The *repetition-free longest common subsequence* (RFLCS) problem [2] asks for a longest common subsequence that has no character occurring more than once in the subsequence. For example, given two strings s_1 = dabcbacbab, s_2 = abbcccbad, an RFLCS is the sequence cba. The RFLCS problem appears for example when one deals with the molecules that come from different origins and the size of the alphabet is usually large (that is, much larger than it is the case of RNA, DNA molecules, and proteins). Another application is concerned with the gene duplication process which plays an important role in detecting similarity between molecules. The problem of gene duplication in the gene rearrangement domain is sparsely considered in literature, due to its difficulty. A similarity measure introduced for the RFLCS problem takes into account the concept of exemplar genes. Sankoff [150] proposed the exemplar model, which consists of identifying the exemplar representative in each genome for each family of duplicated genes. From the side of biology, it would mean that the exemplar may correspond to the original copy of the gene, which later created all other copies. In literature, it is proven that the RFLCS problem with two input strings is already $\mathcal{APX}$–hard, which implies $\mathcal{NP}$–hardness of the problem itself.

Given a pattern string P in addition to the input strings, the *constrained longest common subsequence* (CLCS) problem [162, 5, 49] asks for a longest common subsequence that also has a pattern string P as its subsequence. As an example, given two strings s_1 = dabcbacbab, s_2 = abbcccbad and pattern P=aca, the CLCS is the sequence abcba. An application scenario of the CLCS problem pertains to the identification of the homology between two biological sequences which have a specific or putative structure in common [162]. Studying genomes of various species has shown that some segments are constrained in the lineage. Roughly, around 8% of the human genome consists of sequences that are conserved in other eutherian mammals [145]. A higher proportion of sequences conserved reflects a lower divergence between species. Generally, the length of a CLCS can be used as a similarity measurement for molecules when one takes into account a common specific segment that arises from some structural properties of the compared molecules. A concrete example can be found in [35]. It deals with the comparison of seven RNase sequences so that the three active-site residues, HKH, form part of the solution. 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 applications in other areas, for instance, in communication or magnetic recording [37].

The *longest common square subsequence* (LCSqS) *problem*, described by Inoue et al. [91], asks for the longest common subsequence which is a square at the same time. String

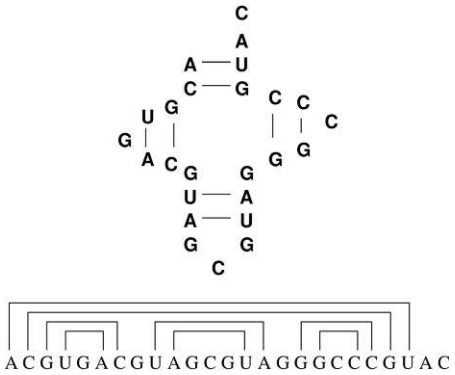


Figure 1.1: Example of an RNA molecule and its corresponding arc-annotation shape.

s is a square iff there exists string s' such that $s = s' \cdot s'$, where "$\cdot$" denotes the string concatenation. For example, if we are given $s_1 = \texttt{dabcdab}, s_2 = \texttt{adbdccdad}$, LCSqS is the sequence dada. The LCSqS is used as a measure of similarity between disjunctive parts of each of the molecules. It can give a better insight into the internal similarity of the compared molecules rather than just considering an LCS.

The *longest common arc-preserving subsequence* (LAPCS) *problem* [97, 62] differs structurally from the aforementioned problems. An arc-annotated string is a tuple (s, P_s), where s is a string over some alphabet Σ and P_s is a set of arcs linking pairs of positions of string s. More specifically, if $(x, y) \in P_s$, it means that there is an arc in sequence s between the character at position x and the character at position y. Now, for two arc-annotated strings (s_1, P_{s_1}) and (s_2, P_{s_2}), the LAPCS problem seeks for a longest common subsequence between sequences s_1 and s_2 such that all arcs are *preserved* in the common subsequence. We say that arcs are preserved in the common subsequence $\tilde{s}$ iff for each arc presented in $\tilde{s}$ there is an arc between the same characters in both arc-annotated input strings. If there is no restriction between the appearances of two different arcs in each of the input strings, the problem is provenly $\mathcal{NP}$–hard (even for just two input strings). However, if we add restrictions, such as no two arcs may share endpoints, and arcs in both arc-annotated input strings may not cross, the problem becomes much easier to solve—it is even polynomially solvable. For more about the other restricted variants of the LAPCS problem, see [97]. In this thesis, we are mostly interested in solving the general LAPCS problem where no restrictions between two arcs occur in the given arc-annotated input strings. The LAPCS problem has applications in comparing RNA and DNA molecules where there might exist arcs between *nucleogenus* bases, that is, arcs between *adenine* (A) and *thymine* (T) or arcs between *guanine* (G) and *uracil* (U) (*cytosine* (C) in DNA, respectively). An example of encoding an arc-annotated sequence from an RNA sequence is given in Figure 1.1. An example of a LAPCS problem instance can be found in Figure 1.2.

In the literature related to these string problems, there is a lack of exact approaches; a partial exception being the case of the classical LCS problem. This could be explained

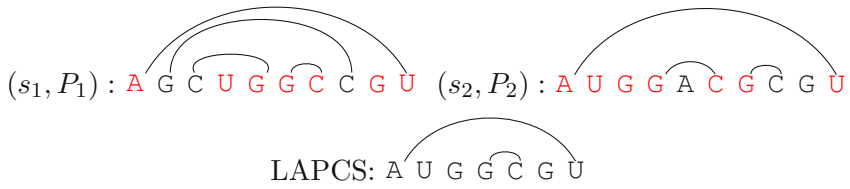


Figure 1.2: Example of an LAPCS instance. The characters of the solution are highlighted in each of the arc-annotated input strings.

by the two following facts. First, the theoretical worst-case complexity bound to solve these problems is high, which might indicate that chances are low to construct reasonably fast exact approaches. Secondly, obtaining a reasonably "good" heuristic solutions was already sufficient to detect similarity between molecules in practice. In this thesis, we aim to provide various techniques to tackle small-to-middle sized problem instances of the considered problems exactly and large-sized problem instances heuristically, in more effective ways. The next paragraphs and Table 1.1 give an overview of the algorithms developed for the considered problems.

For solving the classical LCS problem, we propose a general beam search (BS) framework (GBSF) by collecting the main ideas from the heuristic approaches that had been introduced in the last decades (like pruning and filtering) and incorporating them as components of the framework. Further, we derive a novel state-of-the-art guidance heuristic which approximates the expected length of an LCS under the assumption that all input strings are random. This heuristic guidance shows clear benefits as it can guide the search towards more promising regions even for real-word benchmark sets. It was able to deliver new best results on many considered instance classes. Moreover, the general search framework is introduced to overcome an issue with the earlier literature where a lack of a fair systematic comparison was noticed. The main conclusions derived from this research are: If the main goal is to produce high-quality solutions within a short time interval, then it is sufficient to apply the BS framework with a high beam width. For solving the LCS problem exactly, we further propose a novel A* search framework. The A* search is a well-known exact algorithm that is effective for various problems in the field of *Artificial Intelligence* which relate to pathfinding tasks on large weighted graphs [79]. The approach is based on efficiently calculable and reasonably tight upper bounds and carefully chosen data structures necessary to obtain an efficient search. Our A* search can solve more LCS instances to proven optimality than the best known exact techniques from the literature. In order to tackle large-sized problem instances, we develop two novel hybrids that belong to the class of anytime algorithms: (*i*) a combination of A* and GBSF (A*+BS), and (*ii*) a hybrid of A* and *Anytime Column Search* (A*+ACS) [165] utilizing the same filtering as GBSF. The idea of A*+BS is to interleave classical iterations of A* with GBSF that starts from the carefully selected not-yet-expanded node as its root node. In this way, A* iterations are engaged for improving current dual bounds of the problem while the task of GBFS executions is to possibly improve current primal bounds. If both

bounds match, optimality is proven. A*+ACS hybrid is constructed in a way that after several classical A* iterations, a major iteration of ACS is performed. The ACS search is guided by a novel search guidance which approximates the expected length of an LCS on random strings. A*+ACS hybrid can deliver high-quality solutions within a shorter time, it has better convergence and delivers smaller remaining gaps for the LCS problem in comparison to some other state-of-the-art anytime algorithms from earlier literature. Thus, it can be arguably considered to be a new state-of-the-art algorithm for the LCS problem. Rigorous experimental studies concerning the LCS problem are presented in Chapter 3.

The LCPS problem with an arbitrary set of input strings as input is addressed for the first time in [56]. We first propose a greedy heuristic to derive LCPS solutions of reasonable quality in a short runtime. Further, more sophisticated approaches are applied. First, the general search framework for the LCPS problem is proposed. Further, an extension of the previously mentioned beam search framework is presented towards the LCPS problem. Further, an efficient A* search is developed to tackle small-to-middle sized problem instances exactly. However, performing the A* search was mostly limited to instances of smaller sizes. In order to improve the performance of A*, a few *anytime algorithms* [185] have also been constructed. Again, a hybrid of A* search and BS (A*+BS) is considered as well as an improved hybrid one combining A* and *Anytime Column Search* (A*+ACS). These two hybrids differ from the same hybrids developed for the LCS problem in several details such as the node's structure and upper bounds utilized. A*+ACS is able to significantly outperform the results of A*+BS hybrid. The search guidance that approximates the expected length of an LCS on random strings is extended towards the LCPS and incorporated to guide the search of the ACS component in the A*+ACS hybrid. Moreover, a rigorous experimental study on the well-studied LCPS problem with two input strings (2–LCPS) is also conducted. They show the benefits of the proposed A* search also on this more specific case. These exhaustive computational studies show that A* needs significantly less time to prove optimality than the best known specialized approaches for the 2–LCPS problem in the literature. The studies concerning the LCSPS problem are presented in Chapter 4.

In order to solve the LCSqS problem, two heuristic algorithms are developed: (*i*) a *Randomized Local Search* (RLS) [148], and (*ii*) a hybrid of a *(Reduced) Variable Neighborhood Search* (VNS) [133] and a BS heuristic. The latter is able to produce better solutions than the RLS approach. VNS is responsible for ensuring better diversification whereas BS approach is responsible more on exploitation. The studies concerning the LCSqS problem are presented in Chapter 5.

Concerning the RFLCS and LAPCS problems, a new approach based on a transformation of a problem instance (of the considered string problems) into a Maximum Clique (MC) problem instance is proposed. More precisely, an instance is transformed by deriving conflict graph, and a solution of an MC on the complement of the conflict graph corresponds to a solution of the original RFLCS or LAPCS, respectively. State-of-the-art exact and heuristic solvers are utilized to solve the MC problem on the derived conflict

graphs, and, in that way, to solve the original string problem instances. As a consequence, many RFLCS and LAPCS problem instances are solved for the first time in the literature. However, the main limitation of the transformation was the size of the conflict graphs because their size grows exponentially in the instance size. To deal with the issue, we propose a reduction of the number of vertices of the conflict graphs based on suboptimality checks which make use of lower and upper bounds. The reduction is effective primarily when applying the MILP solver Cplex. The MILP–based approach is able to solve significantly more instances to proven optimality in case of both, RFLCS and LAPCS problem, than the two MC solvers included in the experimental evaluation. The studies concerning the RFLCS and LCSPS problem are presented in Chapter 6.

Concerning the CLCS problem, in order to obtain CLCS solutions of reasonable quality as quickly as possible, we developed a greedy method. For obtaining high-quality solutions, we extend the mentioned GBSF for the LCS problem towards the CLCS problem. This GBSF differs from the same framework developed for the LCS problem in several details such as the node's structure and additional data structures utilized to ensure an efficient search. Further, we consider an efficient A* search to solve the CLCS problem exactly. The efficiency of the A* search is proven by comparing it to the specialized algorithms for the 2–CLCS problem, that is well-studied in the literature. A* search is able to prove optimality for all considered (practical and random) benchmarks within one-to-two order of magnitude shorter runtimes than the best approaches from the literature. To guide the search of the GBSF, two effective heuristic search guidances are developed: (*i*) probability-based heuristic, and (*ii*) an extension of the approximate expected length calculation from the LCS towards the CLCS for random strings. Beam search guided by the two heuristics tends to be the most effective in solving the large-sized problem instances in comparison to some other search guidances used in our experimental evaluation. The details of the studies concerning the LCSPS problem are presented in Chapter 7.

1.1 Structure of the Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews basic techniques used to develop of our algorithms. In Chapter 3 the methods to solve the longest common subsequence problem are presented. Chapter 3 is based on the two published papers:

- M. Djukanovic, G. R. Raidl, and C. Blum. *A Beam Search for the Longest Common Subsequence Problem Guided by a Novel Approximate Expected Length Calculation.* Proceedings of LOD 2019 – *the 5th International Conference on Machine Learning, Optimization and Data Science* (Giuseppe Nicosia, Panos Pardalos, Giovanni Giuffrida, Renato Umeton, Vincenzo Sciacca, eds.), volume 11943 of LNCS, pages 154–167, Springer.

- M. Djukanovic, G. R. Raidl, and C. Blum. *Finding Longest Common Subsequences: New anytime* A* *search results.* Applied Soft Computing, 95:106499, 2020.

In Chapter 4 the methods to solve the longest common palindromic subsequence problem are described. This chapter is based on the two published papers:

- 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.
- M. Djukanovic, G. R. Raidl, Christian Blum, *Anytime algorithms for the longest common palindromic subsequence problem. Computers & Operations Research.* Volume 114:104827, 2020.

Chapter 5 presents two heuristic methods to solve the longest common square subsequence problem and is based on the paper:

- M. Djukanovic, G. R. Raidl, and C. Blum. *A Heuristic Approach for Solving the Longest Common Square Subsequence Problem.* In Proceedings of EUROCAST 2019 – *the 17th International Conference on Computer Aided Systems Theory* (Roberto Moreno-Díaz, Franz Pichler, Alexis Quesada-Arencibia, eds.), volume 12013 of LNCS, pages 429-437, 2019, Springer.

Chapter 6 describes a relation between the Maximum-Clique (MC) problem and variants of the longest common subsequence problems which are then solved via the most efficient (exact and heuristic) MC solvers from the literature. The content of this chapter mainly follows the published paper:

- C. Blum, M. Djukanovic, A. Santini, H. Jiang, C.-M. Li, F. Manyà, G. R. Raidl. *Solving longest common subsequence problems via a transformation to the maximum clique problem. Computers & Operations Research.* Volume 125:105089, 2021.

In Chapter 7 we study the generalized constrained longest common subsequence problem. This chapter is based on the following papers:

- M. Djukanovic, C. Berger, G. R. Raidl, C. Blum, *An A* search algorithm for the constrained longest common subsequence problem. Information Processing Letters.* 166:106041, 2020.
- M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. *On Solving a Generalized Constrained Longest Common Subsequence Problem.* In Proceedings of OPTIMA 20 – *the 11th International Conference Optimization and Applications*, volume 12422 of LNCS, pages 55–70, 2020, Springer.

Finally, Chapter 8 concludes this thesis with a summary of the key findings and an outlook on future research directions.

Further details on the re-implementations of some algorithms from the literature and extended computational results are provided in Appendices A–D.

1.2 Preliminaries

We give some preliminaries and notations that are commonly used within this thesis. By $S = \{s_1, \ldots, s_m\}$ we denote a set of m input strings over a (finite) alphabet Σ. The notation used throughout the thesis is listed:

- For strings s, we denote its length by $|s|$.
- The j-th letter of a string s is denoted by $s[j]$, $j = 1, \ldots, |s|$.
- By $s[j, j']$, $j \leq j'$, we denote the substring of s starting at the j-th position and ending at position j'; the notation refers to the empty string ε if $j > j'$.
- By $n = \max_{s_i \in S} |s_i|$ we denote the maximum input string length for set S.
- By $|s|_a$ we denote the number of occurrences of letter $a \in \Sigma$ in string s, and by $|s|_A = \sum_{a \in A} |s|_a$ we denote the total number of occurrences of letters from set $A \subseteq \Sigma$ in string s.
- The reverse string of string s is denoted by s^{rev}; note that if $s^{\text{rev}} = s$ then s is a palindrome; for example, `madam` is a palindrome.
- The concatenation of two strings s_1 and s_2 is denoted by $s_1 \cdot s_2$ and is obtained by appending string s_2 to string s_1.
- For $p^{\text{L}} \in \mathbb{N}^m$, by $S[p^{\text{L}}], p_i^{\text{L}} \in \{1, \ldots, |s_i|\}$ we denote the set $S[p^{\text{L}}] := \{s_i[p_i^{\text{L}}, |s_i|] \mid i = 1, \ldots, m\}$ generated from the initial set of strings S.

Table 1.1: An overview of the main contributions of the thesis.

Considered problems	**Heuristic approaches**	**Exact approaches**
LCS	• a novel GBSF • a novel expected length calculation heuristic for LCS	• an A* search • a novel anytime algorithms A*+BS and A*+ACS
LCPS	• a greedy approach • the GBSF for LCS extended towards LCPS problem • the expected length calculation for LCS extended towards LCPS	• an A* search • the anytime algorithms A*+BS and A*+ACS extended towards LCPS
LCSqS	• an ILS approach • a VNS & BS approach	–
RFLCS, LAPCS	• the instances transformed into MC instances • an efficient reduction of the MC instances proposed • the reduced MC instances solved via a state-of-the-art heuristic MC solver	• the instances transformed to MC instances • an efficient reduction of the MC instances proposed • the reduced MC instances solved via CPLEX and a state-of-the-art exact MC solver
CLCS	• a greedy approach • the GBSF extended towards CLCS problem • a probability–based heuristic • the expected length calculation for LCS extended towards CLCS	• an A* search

CHAPTER 2

Methodology

In this chapter, we give an overview of the concepts and techniques used throughout this thesis. This chapter starts by introducing combinatorial optimization problems in a general way. Section 2.1 describes fundamental exact methods to solve such problems, emphasizing branch-and-bound, dynamic programming, A* search, basic mixed integer linear programming techniques, a without checking every single feasible solution is an option. A basic method to deal with this is known as *branch-and-bound*. It utilizes bounds lower and upper bounds on achievable solution values, c.f. [132, 118]. If we are given an instance (Φ, F) of a maximization problem with optimal solution F^*, a value $p \in \mathbb{R}$ is called a *primal bound (upper bound)* iff $F(f^*) \geq p$. Straightforwardly, a value $d \in \mathbb{R}$ is called a *dual bound (lower bound)* iff $F(f^*) \leq d$. When one encounters a primal bound p and dual bound d for which $p = d = F(f)$, $f \in \Phi$ holds, f is then a proven optimal solution. Every feasible solution $f \in \Phi$ provides a primal bound. Dual bounds, on the other hand, are obtainable by utilizing upper bound heuristics. Note that in the context of a minimization problem, the primal bound is an upper bound and the dual bound is a lower bound.

As an example of a COP, consider the 0-1 knapsack problem (0-1 KP) [103]. An instance of the problem is represented by n items where $p_i \geq \mathbb{R}^+$ denotes the profit and $w_i \in \mathbb{R}^+$ the weight of i-th item, for $i = 1, \ldots, n$ and a real value $W > 0$ which denotes the knapsack capacity. We aim at finding a subset of items with a maximum possible profit such that the sum of their weights does not exceed the limit of W. Naturally, i-th item of an 0-1 KP instance is mapped to integer i. The set of feasible solutions can be represented as any subset of $\{1, \ldots, n\}$ so that the sum of the weights of the items considered in the solution does not exceed the capacity W. If a subset $\{1, 2, 4\}$ is a feasible solution, it means that the first, third, and fourth items are taken in the solution and that $w_1 + w_3 + w_4 \leq W$ holds. The objective function F of the problem w.r.t. a

feasible solution f is given by $F(f) = \sum_{i \in f} p_i$. Instances of the problem are parametrized w.r.t. the capacity W.

An optimization problem may be solved by *exact methods* or *heuristic methods*. Exact methods guarantee optimality on found solutions, but their application might not always be feasible in practice (e.g., due to memory or time limitations). Heuristic methods (also called *approximate methods*) compute solutions in affordable time but do not guarantee to find an optimal solution. Moreover, there are *approximation algorithms* which are polynomial–time algorithms that produce a solution which quality is within a factor of the quality of an optimal solution [98, 176]. An approximation algorithm is in principle a heuristic that provides quality guarantees.

2.1 Exact Methods

As mentioned above, the main property of exact methods is a guaranty to find an optimal solution. For COPs that belong to the class of polynomial problems $\mathcal{P}$, there exists an exact algorithm that is able to solve the problem in a polynomial time. In some cases, optimal solutions can be directly computed by exploiting problem–specific aspects. However, the main focus of this thesis are string problems that belong to the class of $\mathcal{NP}$–hard problems, which are in general computationally hard to solve (as long as $\mathcal{P} \neq \mathcal{NP}$). For solving such COPs, it is common to apply enumeration schemes and/or methods based on the divide-and-conquer principle. As a naive exhaustive enumeration is usually not applicable in practice, the key idea consists of omitting some solutions from explicit consideration while still guaranteeing to find at least one optimal solution.

The next sections are organized as follow. An overview of basic tree search algorithms is given in Section 2.1.1. Afterwards, two basic enumeration schemes are explained: branch-and-bound in Section 2.1.2 and dynamic programming in Section 2.1.3. Section 2.1.4 describes A* search, which is especially effective in path planning on huge weighted graphs and can also be applied in solving various (discrete) $\mathcal{NP}$–hard problems such as cutting stock problems, the demand control problems etc., see [169, 33]. In Section 2.1.5, basic integer linear programming techniques are discussed and the main aspects of constraint programming in Section 2.1.6, which declaratively express the constraints on the solutions for given set of decision variables.

2.1.1 Basic Tree Search Algorithms

In the field of combinatorial optimization, it is often effective to apply the principle of the *divide-and-conquer* (D&C) for solving problems. This paradigm is based on recursively partitioning the problem into a series of smaller subproblems until we get simple enough subproblems which can be trivially solved. These solutions are then combined to get a solution of the original problem. This mechanism can be represented as a search which generates a tree, or in the general case, a directed acyclic graph (DAG). Each search node of the tree represents a corresponding subproblem generated by D&C, or more

precisely, a partial solution of the main problem. In a general form, nodes are states which correspond to a (set of) partial solution(s) and remaining subproblems. There is a transition from node u to another node v iff the solution represented by u may be extended (by an action) to obtain a solution represented by node v. In essence, a transition represents a problem-specific transformation, that is, an operation. If there is a transition from node u to node v, v is called a *successor* of u. Initializing a root node of the tree is problem-specific. For example, in the case of the 0-1 knapsack problem, an initial solution may be given by an empty knapsack, that is, an empty solution $\{\}$, which corresponds to the root node. A transition to the subproblem may correspond to adding a not-yet-considered item in the current solution, that is, the *union* of the current partial solution and a not-yet-considered item. Goal nodes in the tree are those nodes which cannot produce further (feasible) transitions.

The core idea of tree search algorithms relies on processing nodes of the tree by exploring all transitions of node v which lead to possibly new nodes that are further used as candidates to process. The basic difference between each tree search algorithm is the criterion of choosing the next node to process, see Algorithm 1 (or [6]) and the parameter *strategy*.

Algorithm 1 A General Tree Search Algorithm

```
1: Input: A COP, strategy
2: Output: a solution or failure
3: Initialize: a search tree T with the initial state (root node) of the problem
4: loop
5:     if no a node in T whose successors are not-yet-visited then
6:         return failure
7:     end if
8:     choose a node v to process according to strategy
9:     if the node v is a goal state then
10:        return the corresponding solution associated with node v
11:    else
12:        encounter successors of v and add them into T
13:        mark v as visited
14:    end if
15: end loop
```

In general, tree search algorithms are classified into two groups:

- the search algorithms which use a fixed strategy to methodically traverse the search tree. *Breadth-first* and *depth-first* search algorithms are the well–known such algorithms. They are in general not suitable for solving complex problems as the large search space makes them impractical for the given time and memory limits.

- the search algorithms which use the information from a heuristic function to determine the order in which the nodes are traversed, giving preference to those nodes that are more likely to reach the required goal (node). A cleaver heuristic search strategy can ensure that longer paths need not be considered in the succeeding search iterations. Examples of this kind of search include Beam search as a heuristic approach and A* search as an exact approach. These algorithms are detailed in Sections 2.2.2 and 2.1.4, respectively.

As an example, we consider the 0-1 knapsack problem and the following instance:

Item	1	2	3
p_i	2	3	3
w_i	2	2	3

where the capacity of knapsack is $W = 5$.

Concerning the state graph of the problem, the root corresponds to an empty solution (which the objective value is equal to 0) is initialized. As transitions from nodes at level i of the tree, we make a decision either to include i-th item in the respective solutions at level i or not. At level 1, the following states are included: one which includes item 1 (the decision $x_1 = 1$) into a solution, and the other which does not (the decision $x_1 = 0$); this means, the two nodes correspond to the solutions $\{1\}$ and $\{\}$, respectively, are generated at level 1. For level 2, we have the transitions that consider item 2 to be included or not, for the two solutions at level 1. Note that at each level of the tree, we check for each transition if it leads to an unfeasible solution, that is, it the new solution exceeds the capacity of the knapsack. Only feasible transitions have been generated in the tree. The complete search tree of the given 0-1 KP instance is given in Figure 2.1.

Breadth-First Search (BFS) *algorithm.* BFS starts at the root of the state graph and examines all nodes at one level before examining nodes at the next level. It is guaranteed to find the goal node (if one exists) with the longest path in terms of the number of edges from the initial state, so it provides a complete search. A disadvantage of BFS is its memory consumption since, as always, all nodes of the current level have to be kept in memory. The search is typically realized respecting FIFO order, that is, maintaining not-yet-examined nodes in a data structure called *queue*. BFS finds many applications such as: finding the shortest path between two nodes, checking the connectivity of a graph, testing bipartiteness of a graph, computing the maximum flow in a flow network, etc., see for example [27]. A BFS traversal on the graph in Figure 2.1 is given by $\{\}, \{1\}, \{\}, \{1\}, \{1, 2\}, \{2\}, \{\}, \{1, 3\}, \{1\}, \{2, 3\}, \{2\}, \{3\}, \{\}$.

Depth-First Search (DFS) *algorithm.* DFS traverses a graph in a depth-ward manner and uses a *stack* structure to remember the next vertex to expand. It means that, at each iteration, a deepest not-yet-processed node is always expanded first. Concerning the memory requirements of DFS, only a single path from the root node to the current node,

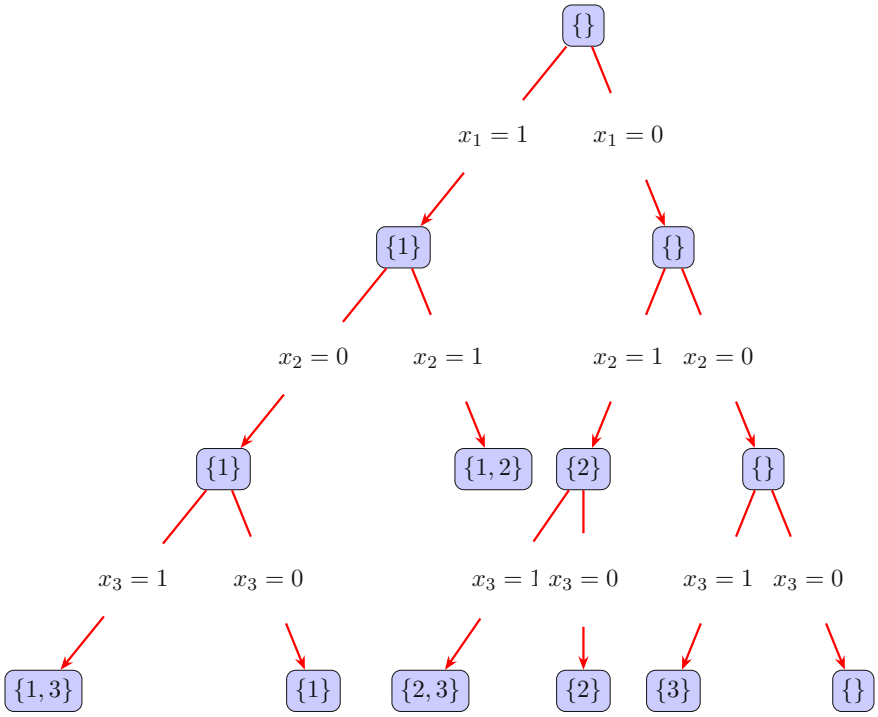


Figure 2.1: Example of a tree.

and any not-yet-examined nodes on the path are stored in a LIFO queue. Applications of the DFS algorithm include finding connected components, topological sorting, finding 2-edge/vertex-connected components, finding 3-edge/vertex-connected components, finding strongly connected components of a graph, solving games such as the Eight Queens Puzzle problem, etc., see for example [160]. A DFS traversal on the graph in Figure 2.1 is given by $\{\}, \{1\}, \{1\}, \{1, 3\}, \{1\}, \{1, 2\}, \{2, 3\}, \{2\}, \{2\}, \{3\}, \{\}, \{\}$.

Iterative Deepening Depth-first Search (IDDFS) [109]. It operates like DFS apart from that the algorithm imposes a limit on how deep the search traverses. The search is repeated with an increased depth limit until a goal state is found. IDDFS keeps advantages of both BFS and DFS. If we continuously increment the depth limit by one until a solution is found, IDDFS search has the same property as BFS since it always finds the longest path to a solution. On the other hand, if a DFS approach on every iteration is used, IDDFS will avoid the memory cost of BFS. For the graph on Figure 2.1, IDDFS traversal with the initial limit of 1 is given in the following order: $\{\}, \{1\}, \{\}, \{1\}, \{1, 3\}, \{1\}, \{1, 2\}, \{2, 3\}, \{2\}, \{3\}, \{\}$.

2.1.2 Branch–and–Bound Algorithm

Note that the descriptions of the algorithms are given w.r.t. maximization.

A general exact technique for solving COPs is *branch-and-bound* (B&B), which enumerates

feasible solutions by making use of upper and lower bound calculation. Its basic idea was invented about 60 years ago in the context of mathematical programming [115, 134]. It applies the principle of the *divide-and-conquer* programming paradigm. The B&B procedure implicitly builds a decision tree such that each node represents a set of solutions and the root node presents the whole problem itself. The B&B performs through the search space by continuously executing the operations of *branching* and *bounding*. *Branching* splits a set of solutions, represented by a node, into multiple mutually disjoint subsets. Each of the subsets represents a new child node in the decision tree. *Bounding* determines a dual bound on the real optimal cost of each subproblem. A global primal bound is maintained by storing so-far best solution value. Before the branching of a node is performed, an upper bound is determined for that node. If its upper bound value is lower than the global primal bound, the node cannot contain an optimal solution for the original problem instance. Hence, considering such a node anymore in the search is not a meaningful option, i.e., its set of solutions can be discarded. The B&B procedure yields an optimal solution if it runs until all feasible solutions are either checked or discarded. The pseudocode of the B&B is presented in Algorithm 2.

Another way to eliminate nodes from B&B tree, apart from comparing upper bound and global primal bound, can be established by checking the *dominance relation*. If the best descendant of a node v_1 is at least as good as the best descendant of node v_2, we say that v_1 *dominates* v_2, which results in discarding the node v_2 [141] from the search. The existence of such relations as well as coming up with an efficient procedure to check the dominance is, in essence, a problem-specific task and the dominance checking is usually time-consuming. B&B, besides lower and upper bounds, has to define also the rules that determine which node is considered next in the search. A common strategy is always picking a node with the largest upper value.

As an exemplary application of the B&B algorithm, we consider the 0-1 knapsack problem [108]. For a branching rule, we could use partitioning a set of feasible solutions into a subset where an item is part of all solutions and another subset consisting of the remaining solutions. For a bounding information, we could first pack all items that are already fixed by the branching and then the remaining items in decreasing order w.r.t. their profit–per–weight values. For another applications of B&B strategy, see the one which solves the LCS problem, introduced in [84].

2.1.3 Dynamic Programming

Dynamic programming (DP) is a powerful mathematical optimization method developed by Richard Bellman [9] in the late 1950s. It is an exact approach that solves a recursive formulation of the problem such that it simplifies a COP by breaking it down into simpler subproblems in a recursive manner. Unlike the divide and conquer paradigm which combines solutions to obtain an optimal solution, these subproblems are not solved independently – results of these smaller subproblems are remembered in the memoization process and reused for the overlapping subproblems. This means that each subproblem is solved only once and then the result is memorized. Wherever there is a recursive solution

Algorithm 2 Branch-and-Bound (maximization)

Input: a problem instance (Φ,F)
Output: an optimal solution
Initialize: $F^* \leftarrow 0$, $A_N \leftarrow \{\Phi\}$ // A_N is set of active nodes
while $A_N \neq \emptyset$ **do**
    Select a node $v \in A_N$ for branching
    if upper bound of $v > F^*$ **then** // may not be pruned
        Split a set of solutions repres. v into subsets of sols. $v_1, \ldots, v_k$ // child. nodes
        for each such v_i **do**
            if v_i is a complete solution **then** // complete: v_i no further partitioning
                if $F(v_i) > F^*$ **then**
                    $F^* \leftarrow F(v_i)$ // a new incumbent found
                    $v_{\text{best}} \leftarrow v_i$
                end if
            else
                Add v_i to A_N
            end if
        end for
    end if
    Remove v from A_N
end while
return the solution represented by node v_{best}

formulation that has repeated calls for the same inputs, it can be optimized by DP with the help of memoization.

In order to solve a COP by means of DP, it is common to detect an *optimal substructure* of the problem. We say that a COP has optimal structure iff it can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems. Another key property is that there should be only a polynomial number of different subproblems that have to be solved. More details and typical methods of DP can be found in [42]. For the theoretical background of the DP approach, we refer to [11, 106].

As an example, let us consider the LCS problem with two input strings s_1 and s_2 solved by DP [170]. To define an *optimal substructure*, by $M(i, j)$ we denote the length of an LCS for two prefix strings $s_1[1, i]$ and $s_2[1, j]$, $1 \leq i \leq |s_1|$, and $1 \leq j \leq |s_2|$. Let us find out which subproblems needs to be determined beforehand in order to be able to calculate $M(i, j)$. If $s_1[i] = s_2[j], i, j \geq 1$, $M(i, j)$ can be calculated by using the information for the LCS between $s_1[1, i-1]$ and $s_2[1, j-1]$, that is $M(i, j) = M(i-1, j-1) + 1$ holds. Otherwise, two subproblems must be calculated, i.e. two LCSs $\{s_1[1, i],\ s_2[1, j-1]\}$ and $\{s_1[1, i-1],\ s_2[1, j]\}$, that are stored in $M(i-1, j)$ and $M(i, j-1)$; in that way $M(i, j) = \max\{M(i-1, j), M(i, j-1)\}$. As a trivial case, we initialize $M(i, 0) = 0$

and $M(0,j)=0$, for $i=0,\ldots,|s_1|$ and $j=0,\ldots,|s_2|$. The optimal value of the initial problem $\{s_1,s_2\}$ is then stored in $M(|s_1|,|s_2|)$. Memoization is done by initializing a $((|s_1|+1)\times(|s_2|+1))$–dimensional matrix M. Summarizing, the full DP recurrence of the problem is given as follows:

$$M(i,j)=\begin{cases}1+M(i-1,j-1), & \text{for } i,j\geq 1\wedge s_1[i]=s_2[j]\\ \max\{M(i-1,j),M(i,j-1)\}, & \text{else,}\end{cases}$$

where $M(i,0)=0$ and $M(0,j)=0$, $i=0,\ldots,|s_1|$ and $j=0,\ldots,|s_2|$. Note that the memory and time complexity for this DP approach is $O(|s_1|\cdot|s_2|)$.

This recursion can be extended to solve the constrained LCS problem with two input strings (2–CLCS) [162]. The optimal substructure is described as follows. By $M(i,j,k)$ we denote the length of an LCS between $s_1[1,i]$ and $s_2[1,j]$ w.r.t. $P[1,k]$. If $a[i]=b[j]=P[k]$, for computing $M(i,j,k)$ we previously compute $M(i-1,j-1,k-1)$, and in that case we conclude $M(i,j,k)=1+M(i-1,j-1,k-1)$. Otherwise, two cases are possible: (i) if $s_1[i]=s_2[j]$, then a subproblem that needs to be considered is $s_1[1,i-1],s_2[1,j-1]$ with $P[1,k]$, yielding the recursion $M(i,j,k)=1+M(i-1,j-1,k)$, and (ii) if $s_1[i]\neq s_2[j]$, two subproblems have to be computed, yielding the recursion $M(i,j,k)=\max\{M(i-1,j,k),M(1,j-1,k)\}$. Note that for the memoization we use a three-dimensional matrix, yielding the complexity to solve 2–CLCS problem to $O(|s_1|\cdot|s_2|\cdot|P|)$.

2.1.4 A* Search

The algorithm was originally developed by Hart et al. [79] to find a smallest-cost path from a start node to a goal node in large weighted graphs $G=(V,A)$, where V denotes set of nodes and A denotes a set of arcs. It is a widely used algorithm in the field of *Artificial intelligence* that belongs to the class of *informed search* methods (uses the idea of heuristic search) for finding shortest or longest paths. A* search makes use of a specific mechanism to minimize the number of nodes that are required to visit in order to encounter a proven optimal solution. It works in a *best-first-search* manner, meaning that the most promising nodes are always expanded first. For ranking 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 an estimated cost of an optimal path from v to a goal node (dual bound).

In order to establish an efficient search, A* search maintains a list of open nodes storing those nodes whose successors have not yet been explored. It also maintains a list of all so far reached nodes. The search procedure keeps the root node r in the open list. At each iteration, a node v with the maximal $f(v)$-value is taken from the open list. This node is *expanded* by visiting all its successor nodes, which are treated in the following way.

Algorithm 3 A* Search (maximization)

Input: A weighted graph $G = (V, A)$, root node r, a goal node t
Output: A longest-cost path from root r to t
Initialize: open list $Q \leftarrow \{r\}$
while $Q \neq \emptyset$ **do**
    Remove node v with maximal $f(v) = g(v) + h(v)$ from Q
    if $v = t$ **then** // complete path found
        return derive the $r - t$ path
    else
        for each successor v' of v **do**
            $f' \leftarrow g(v) + w(v, v')$ // $w(v, v')$: weight of edge vv'
            if v' visited for the first time $\vee$ $f' > g(v')$ **then**
                $g(v') \leftarrow f'$
                $Q \leftarrow Q \cup \{v'\}$ // v' inserted in Q if reached for the first time, otherwise update node v'
            end if
        end for
    end if
end while
return no path from r to t exists

A successor node v' of a node is updated only if: (i) it has been visited (or expanded) before and a better path from root node r to v' has been encountered, or (ii) it is visited for the first time in the search. If any of these two conditions is fulfilled, node v' has been added in the open list. Unless terminated early (e.g. due to time or memory limitation), A* search stops once a goal node is selected for expansion. Pseudocode of an A* search is provided in Algorithm 3. Since at each step, a node with the highest (if maximizing) $f()$–value is extracted, the set of open nodes is realized by means of *priority queue* while the data structure to maintain the set of all visited nodes is a problem–specific (a hash-map is mostly used in applications).

Some theoretical background on A* search. For guarantying that a path found from A* search is indeed a longest-cost path, $h(v)$ must be *admissible* which means $h(v) \geq h^*(v)$, $\forall v \in V(G)$, where $h^*(v)$ denotes real optimal cost of the path from v to a goal node. Moreover, if $w(v, v') + h(v') \leq h(v)$, $\forall (v, v') \in A(G)$, where $w(v, v')$ is the cost from v to v', $h(v)$ is called *monotonic*. A* search with monotonic $h(v)$ will never re-expand already expanded nodes. It is proved that the number of node expansions required to find a proven optimal path by A* search with a monotonic heuristic $h(v)$ is minimal among all search algorithms that utilize the same heuristic guidance and the same tie-breaking mechanism [47].

Dijkstra's shortest path algorithm can be considered to correspond to the special case of A* where $h(v) = 0, \forall v \in V(G)$ [45]. On the other hand, Dijkstra and A* search can be

seen as a special case of DP [63] with merging DP cells. A* search also present a special case of a generalization of B&B [137].

2.1.5 Integer Linear Programming

An *integer linear program* (ILP) is a mathematical problem formulation in which all of the variables are integers while the objective function as well as all constraints are linear in the variables. Solving an ILP is in general $\mathcal{NP}$–complete [28]. A special case of an ILP is the 0-1 integer linear program, where all variables are binary. This problem belongs to the famous 21 (Karp's) $\mathcal{NP}$–complete problems [64]. A proof of $\mathcal{NP}$–hardness can be performed by reducing the minimum vertex cover problem to an ILP.

The canonical form of an ILP is given as follows:

$$\begin{aligned} & z = F(x) = \max \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ & \text{s.t.} \\ & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq 0, \\ & \mathbf{x} \in \mathbb{Z}^n, \end{aligned} \tag{2.1}$$

where A is a matrix whose entries are all integer values, $\mathbf{b} \in (\mathbb{R}^m)^T$ and $\mathbf{c}^T \in \mathbb{R}^n$ are vectors. Model (2.1) can be transformed into a standard form by introducing slack variables whenever "$\leq$" is presented within constraints. Many problems are modeled by means of an ILP like scheduling problems [111], graph problems [30], telecommunications networks problems [102], etc. If we replace $\mathbf{x} \in \mathbb{Z}^n$ by $\mathbf{x} \in \mathbb{R}^n$ in the model (2.1) , we get a linear program (LP) also called linear programming relaxation. LPs are in practice efficiently solved by means of the simplex algorithm [105]. In general, LPs can be solved in polynomial time by the interior-point method [100].

To solve an ILP, a simple idea would be to solve an LP relaxation of the ILP and round the values that are obtained. However, it might happen that the rounded solution is not optimal or it may not even be feasible (violating some constraints). If we assume that matrix A is unimodular, $\mathbf{b}$ has all integer entries, and $A\mathbf{x} = \mathbf{b}$, every feasible corner solution will be an integral one which means that a solution returned by the simplex algorithm will be integral. It is quite rare in practice that matrix A is unimodular, in which case, there exist many exact techniques to solve such ILP.

A valid inequality for an ILP is any constraint that does not eliminate any feasible integer solution. If the addition of such a valid inequality cuts off the current solution of the LP relaxation, it is called a *cutting* plane. Cutting planes serve to eliminate part of the LP feasible region as well as fractional solutions possibly. The *cutting plane method* is an exact algorithm that solves a series of LP relaxations of the original ILP, iteratively adding linear constraints that navigate the search towards an integer value. The use of the cutting planes to solve (Mixed) ILP was introduced by Gomory [71] in the 1950s. The basic strategy of the Gomory cutting plane method consists of the following steps:

1. use simplex algorithm to solve the LP relaxation of the ILP problem, let x^{relax} be an optimum of the relaxation; if the solution is unbounded or unfeasible, the status is reported;
2. if the values of x^{relax} are all integer, an optimal solution is found;
3. otherwise, iteratively determine and add Gomory cutting planes to the relaxation.

A gomory cut is generated at each iteration as follows:

- when the simplex method solves the relaxation, we obtain

$$\sum_{i,j} {a^*}_{ij} x_j = b_j^*, \tag{2.2}$$

  where formula (2.2) refers to the (matrix) tableau obtained in the last iteration of the dual simplex method utilized to solve the relaxation.

  Select any constraint with non-integer b_i^*, for some $i \in \{1, \ldots, j\}$;

- the selected constraint in 2.2 is rewritten by using fractional part $f_{ij}^* = a_{ij}^* - \lfloor a_{ij}^* \rfloor$ and $f_i = b_i^* - \lfloor b_i^* \rfloor$, i.e

$$\sum_{ij} f_{ij}^* x_j - f_j^* = \lfloor b_j^* \rfloor - \sum_{i,j} \lfloor a_{ij} \rfloor x_j$$

- add a new constraint $\sum_{ij} f_{ij} x_j - f_j \geq 0$ to the current simplex tabelau (as a row);
- repeat above steps (using dual simplex) until all b_j^*'s are integers.

Choosing a constraint in (2.2) may be a heuristic decision; for example one can choose the common strategy is to select non-integer b_i^* with the largest $f_i^* = b_i^* - \lfloor b_i^* \rfloor$ (fraction) value.

Another algorithm, which is frequently effective in solving ILP, is the LP-based B&B method which performs branching over a variable with a fractional value obtained as optimal solutions of the relaxation. The idea of the B&B is already discussed in Section 2.1.2 and consists of dividing the original problem into a series of sub-problems. Many of these subproblems will not be solved in the case when an upper bound of the optimal solution of the considered subproblem is smaller than the current best solution (obtained directly by a heuristic search). For the ILP problem, the relaxation is an upper bound on the optimal solution. The B&B to solve the ILP uses the same procedure as in Section 2.1.2 where the specific aspects are resolved in the following way (see [177]).

1. Each node relates to an LP relaxation and its solution; let for each node v, the related LP relaxation be denoted by LP_v and its solution by $x_v^* = (x_{v,1}^*, \ldots, x_{v,n}^*)$; if x_v^* is an integer vector, B&B checks if a new best lower bound has obtained, and if an optimal solution has found.

2. At the root node we solve the LP relaxation of the original ILP;

3. The branching of each node v is performed in the following way:

   - select $i \in \{1, \ldots, n\}$ with the largest fractional part $f_i^* = x_{v,i}^* - \lfloor x_{v,i}^* \rfloor$ for branching;
   - generate two new successor nodes v^L and v^R of node v. Node v^L corresponds to an LP relaxation obtained by adding the constraint

     $$x_i \geq \lceil x_i^* \rceil$$

     into the relaxation LP_v of node v, and node v^R corresponds to an LP relaxation obtained by adding constraint

     $$x_i \leq \lceil x_i^* \rceil - 1$$

     into the relaxation LP_v of nde v;

4. the common strategy to select the next node to be expanded is choose a node with the largest UB value among not-yet-expanded nodes.

For the following ILP problem

$$\begin{aligned} &\max z = F(z) = 100x_1 + 150x_2 \\ &\text{s.t.} \\ &15x_1 + 30x_2 \leq 200, \\ &8000x_1 + 4000x_2 \leq 40000, \\ &x_1, x_2 \geq 0, x_1, x_2 \in \mathbb{Z}^n, \end{aligned} \tag{2.3}$$

Figure 2.2 shows a B&B to solve it.

The combination of B&B and the cutting plane method in practice works much better in many applications than the cutting algorithms alone. This hybrid method is known as *branch and cut* (BnC) [131]. The BnC method solves the LP relaxations of the original problem using the regular simplex algorithm [138]. These LPs are generated by splitting the problem into multiple subproblems using B&B that are then solved, recursively. During the search, cutting planes that are violated by the obtained solution of the relaxation are detected and then added to the original LP. Nowadays, all commercial ILP solvers, like Cplex, use cutting methods in one way or another (as BnC). For more advanced techniques to solve ILP such as Benders decomposition, column generation, and Dantzig-Wolfe decomposition, we refer to [129].Currently, the best known commercial ILP solvers used in the literature are Cplex [43] and Gurobi [125].

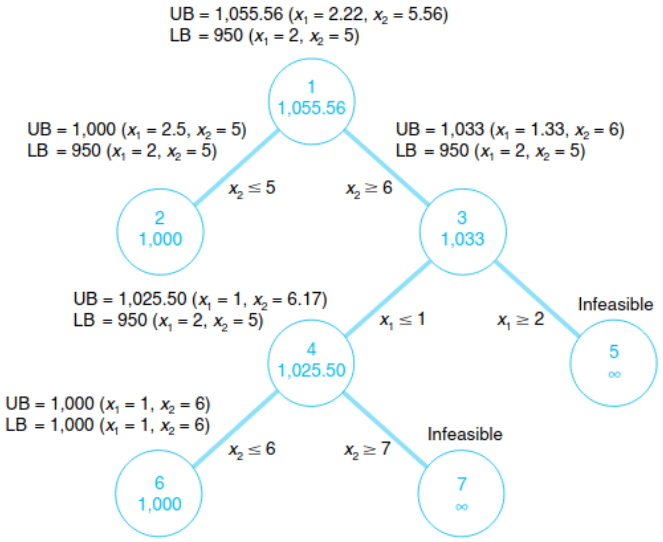


Figure 2.2: A working example of the B&B for an ILP, downloaded from here.

As an example for an ILP model, we consider the well-known *maximum independent set* (MIS) problem on graph $G = (V, A)$, which is modeled by the following ILP model:

$$\max \sum_{v \in V} \tag{2.4}$$

s.t.

$$x_u + x_v \le 1, \qquad (u, v) \in V \times V, u \ne v, \tag{2.5}$$

$$x_u \in \{0, 1\}, \qquad u \in V. \tag{2.6}$$

If x^* denotes an optimal solution of the above ILP, the set $V' = \{u \mid x^*_u = 1\} \subseteq V$ denotes a MIS on graph G.

We now define an ILP model for the RFLCS problem which plays an important role in the methodology of solving the problem. For each match between input strings s_1 and s_2, one decision variable is generated. That is, for all $s_1[i] = s_2[j]$, we assign one decision variable $z_{ij} \in \{0, 1\}$. By $\ell(z_{ij})$ we denote the letter assigned by the variable z_{ij}. Further, we say that there exists a *conflict* between two different decision variables z_{ij} and z_{kl} iff

$$((i \le k) \wedge (j \ge l)) \vee ((i \ge k) \wedge (j \le l)) \vee \ell(z_{ij}) \ne \ell(z_{kl}).$$

With $z_{ij} \prec z_{kl}$, we denote that the two variables are in conflict. Let Z denotes the set of all decision variables of an instance, and $Z_a := \{z \in Z \mid \ell(z) = a\}$, $a \in \Sigma$. Now, an ILP

model for the RFLCS problem can be presented by the following model:

$$\max \sum_{z \in Z} z \tag{2.7}$$

s.t.

$$\sum_{z \in Z_a} z \leq 1, \qquad \forall a \in \Sigma, \tag{2.8}$$

$$z' + z'' \leq 1, \qquad z' \prec z'', z', z'' \in Z, \tag{2.9}$$

$$z \in \{0, 1\}, \qquad z \in Z. \tag{2.10}$$

Equation (2.8) ensures that *repetition-free* constraint is fulfilled and Equation (2.9) ensures that the feasible solution refers to a common subsequence (CS). The objective function maximizes the length of CS which does not have letters that appears more than once in the subsequence.

2.1.6 Constraint Programming

We mainly follow [167] to give a short overview on this paradigm.

Constraint Programming (CP) is a general framework to model and solve *Constraint Satisfaction Problems* (CSPs). A CSP is a triple (X, D, C) where $X = \{x_1, \ldots, x_p\}$ represents a set of variables, $D = D_1 \times \ldots \times D_p$ respective (finite) domains of variables and $C = \{C_1, \ldots, C_q\}$ set of constraints. A constraint $C_i = (S, R)$ is a tuple where $S \subseteq X$ (scope) are variables induced into constraint C_i and $R \subseteq D$ (relation) are tuples satisfying C_i.

Given a CSP X, a solution of X is an assigning $x_1 \mapsto a_1, \ldots, x_p \mapsto a_p$ such that $a_i \in X_i$ and all constraints C_i are satisfied $(i = 1, \ldots, p)$. Related problems ask for finding all solutions or finding a best solution w.r.t. an objective function of a (combinatorial) constraint optimization problem. CP is a solving paradigm which provides a very flexible modelling language, close to natural language. This paradigm can be described as a combination of constraint propagation and tree search. Constraint propagation is a reduction of variable domains by logic deductions based on the constraints. Constraint propagation makes the domains consistent with each constraint. At each search state, specific constraint propagation algorithms are applied. The solution process of a CP includes three basic steps (Algorithm 4):

- *domain filtering* in which *inconsistent* values from the domains of the variables are removed based on individual constraints.
- *constraint propagation* which propagates the (restricted) domains through the constraints, by re-evaluating them until there are no more changes in the domains. A basic propagation procedure for a constraint C is given in Algorithm 5.
- *search* which implicitly enumerates all possible variable-value combinations; the search tree is kept to a smallest possible size due to domain filtering and constraint

Algorithm 4 A General Constraint Programming Scheme (for CSP)

```
Create model of a COP
while ¬ (solved ∨ infeasible) do
    Remove-inconsistent-values    //constraint propagation
    Select-decision-variable    // values distribution
    Select-value-for-variable
end while
```

Algorithm 5 Constraint Propagation

```
Input: C: constraint, X_c: variables, D_c: domain
for all x ∈ X_c do
    for all a ∈ D_c do
        find solution to C with x = a
        if no such solution exists then
            D_c ← D_c \ {a}
        end if
        if X_c = ∅ then
            return false
        end if
    end for
end for
return true
```

propagation. Some of the most used search strategies include back and forward checking, variable and value ordering, B&B, etc.

Concerning mixed ILP, we are used to formulating problems as a set of linear inequalities. In CP we describe substructures (so-called *global constraints*), combining them in various combinators. For example, the sequencing constraints are rather messy in MIP, but straightforward and intuitive in CP. So, CP models are usually much more intuitive than MIP models. CP offers the user many kinds of global constraints, which makes modeling simple, easy and natural to interpret. One of the most used global constraints for modeling is $alldifferent(x) = \wedge_{(i,j)|\, i \neq j} x_i \neq x_j$. A global constraint is a combinatorial structure that is made as a combination of elementary constraints. It serves as an expressive building block for modeling a problem. A global constraint utilizes powerful algorithms from Artificial Intelligence, Operational Research, Graph theory etc., to ensure effective search. As a good receipt for modeling, it is always a good strategy to identify possibly global constraints in the model. For more about global constraints, see [147, 7].

Well known Constraint Programming Solvers are IBM ILOG CP *Optimizer* [114], *MiniZinc* [139], *GeCode* [151], *Prolog* [41], and *Choco* [143]. *MiniZinc* is a more general CP modeling language that can be applied in conjunction with different solvers.

As an example, we generate a CP model for the *graph coloring problem* which is given as follows. Given is a graph $G = (V, A)$ and $K \in \mathbb{N}$. In order to do coloring of graph G, to each vertex $v \in V$ we need to assign a number (color), denoted by $col(v)$ such that the coloring is valid. A coloring of graph is *valid* iff for each edge $e = (v_i v_j) \in A$, $col(v_i) \neq col(v_j)$. The problem aims at finding a valid coloring of graph G with uses a minimal number of colors (bounded by K). A CP model is given by:

$$\begin{aligned}
&\text{vars } X: x_i \text{ for each } v_i \in V; \\
&\text{domains } D: D_i = \{1, \ldots, K\}; \\
&\text{constraints } C_{ij}: x_i \neq x_j, \forall e = (v_i v_j) \in A; \\
&\qquad\qquad x_0 = \max\{x_i \mid i \in V\} \qquad (2.11) \\
&\text{objective: } \min x_0 \qquad (2.12)
\end{aligned}$$

Note that the above model can be additionally improved (by breaking symmetries).

2.2 Heuristic Methods

The main purpose of heuristic methods relies on providing good solutions in short time. In contrast to exact methods, the produced solutions might not be optimal and usually, no performance guarantee is provided by heuristics. Heuristics are applied either to generate a solution that could serve as promising starting solutions of an exact method (e.g., to prune suboptimal components) or they are the only practically applicable methods due to the difficulty of the considered problem instances.

We start by describing constructive heuristics in Section 2.2.1 which return feasible solutions in polynomially many steps w.r.t. the instance size. An extension of the idea of the constructive heuristics is given by the Beam Search heuristic, which is discussed in Section 2.2.2. Section 2.2.3 describes local search that provides a mechanism of finding possibly better solutions by performing a series of small changes. The concept of locally optimal solutions, which are solutions that cannot be improved anymore by local search, is described here. In order to also escape from local optimal solutions and cover the whole search pace better, the development of *metaheuristics* is motivated.

Metaheuristics present problem independent frameworks that deal with difficult problem instances for which an application of exact algorithms is hardly possible. Metaheuristics cover a wide range of ideas on how to efficiently search through a set of feasible solutions to reach a (near-) optimal solution. Each of the metaheuristics defines their own mechanisms for *intensification* and *diversification*. Intensification refers to the ability to obtain high-quality solutions within each (explored) region of the search. Intensification denotes the process of exploration of similar solutions to a considered solution to achieve improved solutions. It is typically realized using local search or extensions thereof. Diversification generally refers to the ability to visit distant regions of the search space w.r.t. current position in the search. It is applied when the possibilities for improving solutions are (almost) exhausted by moving the focus of the search to new (possibly not-yet-explored)

regions.

In the last few decades, a large variety of different metaheuristics have been proposed; for an overview, see [158, 69]). A few metaheuristics used in the course of our work are discussed here. The iterated greedy algorithm is described in Section 2.2.4. Section 2.2.5 explains the idea of the general variant of variable neighborhood search which is primarily based on local search utilizing multiple neighborhood structures and shaking steps that are responsible for the intensification and diversification of the algorithm, respectively. For other prominent metaheuristics, we refer to simulated annealing (SA) [168], tabu search (TS) [67], genetic algorithm (GA) [154], particle swarm optimization (PSO) [101], ant colony optimization (ACO) [59], etc. SA models the process of heating a material while slowly decreasing the temperature to decrease the defects, that is, to minimize the system energy. TS makes use of the search history (tabu list) for diversification by remembering already visited solutions discovered by local search and, in that way, avoiding to visit the explored regions of the search space over again within a limited time. SA and TS are both based on local search. GA algorithm is inspired by the process of natural selection. It maintains a set of solutions that is called a population. At each iteration, the algorithm chooses some solutions (individuals) of the population. Each individual is then modified (that is, recombined and possibly randomly mutated to hopefully obtain better solutions) to generate a new population for the next iteration. PSO is inspired by reproducing observed behaviors of animals in their natural habitats, such as birds flocking or fish schooling. PSO maintains a population of candidate solutions (particles) and moving the particles around in the search space according to some rules based on the particles' positions and velocities. Each movement of a particle is influenced by its local best-known position but is also guided toward the best-known positions in the search-space w.r.t. all particles. This pushed the swarm towards the best solution. The inspiration of ACO metaheuristic comes from the behavior of ants for finding shortest paths. GA, PSO, and ACO belong to the population-based metaheuristics. Our descriptions of the methods mainly follow the textbook [19].

2.2.1 Constructive Heuristics

A *Constructive Heuristic* (CH) aims at producing a solution in typically short runtimes. It starts with an empty solution which is iteratively extended until no further extensions is possible. CH is called a *greedy heuristic* (GH) iff at each iteration, the decision of which extension to perform among all possible extensions is based on some greedy criterion. The choice of the greedy criterion has the main impact on the performance of the method. A pseudocode of the CH is given in Algorithm 6.

As an example of CH, we consider the well–known *traveling salesman* (TSP) problem. In the beginning, we initialize s^P as an empty path. We extend the path by repeatedly adding an edge of minimal cost determined from those edges that link not-yet-visited cities (those which are not yet on the path that presents s^P) with those that are already visited, that is, included in s^P.

Algorithm 6 Constructive Heuristic

1: **Input:** an instance of a COP
2: **Output:** A (feasible) non-expandable solution (or reporting that no feasible solution)
3: $s^P \leftarrow ()$ // partial solution set to empty solution
4: **while** Extend$(s^P) \neq \emptyset$ **do**
5: Select component $e \in$ Extend(s^P) // w.r.t. some greedy criterion g
6: Extend s^P by e
7: **end while**

Concerning our string problems, the LCS problem with m input strings $S = \{s_1, \ldots, s_m\}$ is solved by the Best–Next heuristic (BNH), which is a greedy heuristic, proposed in [65]. First, note that the solution components of the problem are letters from the alphabet Σ. Extending a partial solution s^P by some letter a refers to the operation of concatenation, that is, appending a character to partial solution s^P. Letter a has to be a feasible extension, that is, $s^P \cdot a$ must be also a common subsequence for all input strings. The BNH initializes an empty string $s^P := \varepsilon$ as a starting solution. It also initializes the left pointers $p_i^{\mathrm{L}} = 1$, for all $1, \ldots, m$, assigning those pieces of input strings for which feasible extensions of current s^P are determined, that is, feasible letters are checked from the set of strings $S[p^{\mathrm{L}}] = \{s_i[p_i^{\mathrm{L}}, |s_i|] \mid i = 1, \ldots, m\}$. For the components that can feasibly extend current solution s^P we choose those characters that appear in all strings from $S[p^{\mathrm{L}}]$ are. We denote such set of letters by Extend(s^P). For each feasible letter $a \in$ Extend(s^P), it makes sense to consider those letters at the positions of the first occurrences of letter a in strings $s_i[p_i^{\mathrm{L}}, |s_i|]$, $i = 1, \ldots, m$, denoted by $p_{i,a}^{\mathrm{L}}$, $i = 1, \ldots, m$. At each iteration of BNH, we choose a letter $a^* \in$ Extend(s^P) which minimizes the (greedy) criterion

$$g(a, p^{\mathrm{L}}) = \sum_{i=1}^{m} \frac{p_{i,a}^{\mathrm{L}} - p_i^{\mathrm{L}} + 1}{|s_i| - p_i^{\mathrm{L}} + 1}, a \in \text{Extend}(s^P),$$

and then we do the extension $s^P := s^P \cdot a^*$ and update the current left pointers $p_i^{\mathrm{L}} := p_{i,a^*}^{\mathrm{L}} + 1, i = 1, \ldots, m$. The above steps are repeated until Extend$(s^P) = \emptyset$, returning s^P as a final solution. Note that some letters from Extend(s^P) might dominate other ones, which can then be filtered out from the set. For the details, see [65].

2.2.2 Beam Search

In essence, Beam Search (BS) is an incomplete version of BFS, in which at each level a subset of most β nodes is selected and further processed. The pseudocode of BS is presented in Algorithm 7. It includes a (weighted) graph G, beam width β, starting node r and heuristic h as an input; heuristic h serves for valuating the nodes of G. BS is a heuristic search algorithm that maintains a collection of nodes, called *beam* B, at each level of the search. At each iteration of a BS, the successor nodes of any node from B

Algorithm 7 Beam Search

1: **Input:** G: a (weighted) search graph, r: starting node, β: beam width, h: search heuristic
2: **Output:** a path from r node with approximate (possible optimal) to a goal node
3: $B \leftarrow \{r\}$
4: $P_v \leftarrow \emptyset$;
5: $cost_{\text{best}} \leftarrow 0$
6: **while** $B \neq \emptyset$ **do**
7: $\quad V_{\text{ext}} \leftarrow \emptyset$
8: $\quad$ **for each** $v \in B$ **do**
9: $\quad\quad$ **if** v is a *goal* node **then**
10: $\quad\quad\quad P_v \leftarrow$ derive the $r - v$ path
11: $\quad\quad\quad$ **if** cost of path $P_v > cost_{\text{best}}$ **then** // new incumbent has reached?
12: $\quad\quad\quad\quad P_{\text{best}} \leftarrow P_v$
13: $\quad\quad\quad\quad cost_{\text{best}} \leftarrow$ cost of path P_v
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **else**
16: $\quad\quad\quad V_{\text{ext}} \leftarrow V_{\text{ext}} \cup Successors(v)$ // add children of v
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad B \leftarrow$ `Reduce`$(V_{\text{ext}}, h, \beta)$ // $\beta > 0$ best nodes for the new beam
20: **end while**

are visited and among them (up to) $\beta > 0$ most promising nodes w.r.t. their h–value are selected to generate a beam of the next level (`Reduce()` procedure in Algorithm 7). The above steps are repeated until beam B is empty. The longest path from the root node to a goal node, encountered during the search, is stored as the incumbent solution. The longest path is, afterwards, returned as an output of BS.

As we said, BS is performed in a limited breadth-first-search manner. It can be considered as optimization of BFS which reduces its memory requirements. For the performance of BS, two facts play a major role

- the value of beam width β, and
- the choice of heuristic h.

The choice of h is usually a problem specific task, whereas the right choice of β involves advances techniques for tuning parameters and is usually dependent on the instance size. Note that BS with $\beta = 1$ corresponds to a constructive heuristic with $h = g$.

The worst-case time scenario of BS occurs when the heuristic h leads the search towards the maximum depth of the graph G. The worst-case time, in that case, is $O(|B| \cdot d \cdot \tilde{h})$, where d is the maximum depth of any path in G which starts from root r and $\tilde{h}$ is the

time complexity of h calculation. The worst-case memory complexity is also $O(|B| \cdot d)$. The linear memory consumption allows BS to search deeply into huge search spaces and eventually find solutions that other search algorithms can hardly reach. BS has found success in a wide range of optimization problems proving its effectiveness; such examples include problems from the domain of job scheduling problems, speech recognition, vision, planning, machine learning etc., see, for example [107, 66, 140, 113].

As an example of performing a BS, we consider the TSP and the following instance:

City	1	2	3	4
1	-	5	12	8
2	5	-	8	2
3	12	8	-	8
4	8	2	8	-

In order to apply a BS, we first define a node's structure and the neighborhood relation. Due to simplicity, we do not think about memory optimization of the BS or breaking symmetries in the search, but just give the main focus on illustrating a working example of a BS.

Each node v refers to a set $T_v \subseteq \{1, \ldots, c\}$ of nodes which are included in the tour, where c is the number of cities in the respective instance. The order of nodes in the set is preserved. For example, if $v = \{1, 2, 3\}$, it means that we link city 1 with 2, and city 2 with 3 and obtain the respective path. Note that in this way the number of such nodes in the search is bounded by $O(n!)$. A goal node is any node for which $|T_v| = c$. There is an edge between two nodes v_1 and v_2 iff $T_{v_2} = T_{v_1}.push\_back(\{c_1\})$ and $T_{v_2} = T_{v_1} + 1$, for some city c_1. It means that visiting the successor nodes of node v correspond to appending the path that indicates node v by a new city (which is not already on the path).

It remains to define an upper bound on the length of an optimal tour. We use here the simple Nearest-Neighbor (NN) heuristic due to ease of demonstration and computing the bounds in our working example. The heuristic works as follow for specific node v. We start at the latest added city of T_v and travel to the nearest not-yet-scheduled vertex. We keep continuing the process until all vertices that are not in T_v are included, returning to the leading city (in this case the city labeled by 1). This procedure produces a valid Hamiltonian cycle, which is at least as long as the minimum cycle. For the other more complex upper bounds on the TSP, we refer the reader to [4]. In our example, $h(v)$ will indicate the sum of the path assigned by the node and the value obtained by the NN heuristic.

2.2.3 Local Search

Local search is a technique that aims at producing a series of improvements of solution quality starting from a given solution. In this section, we first introduce the concepts of

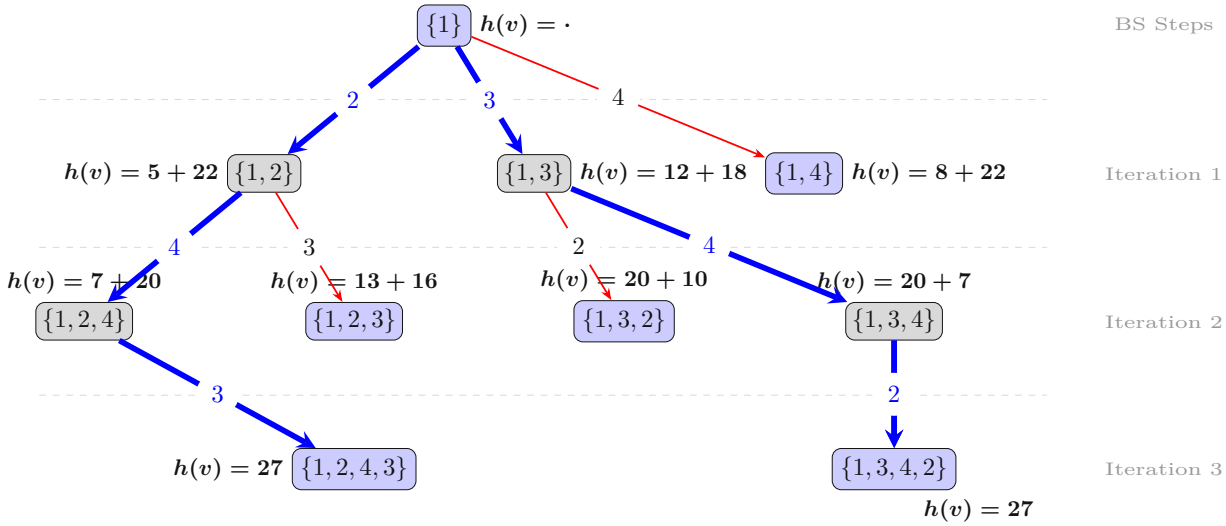


Figure 2.3: The example above shows the BS state graph generated by a working BS on the above instance set. The blue lines denote best paths (solutions) obtained by BS. The gray nodes are nodes kept in the beams of respective level.

neighborhoods and local optimal solutions which are the basic terms of local search. We mainly follow the book by Papadimitriou and Steiglitz [141].

Given a solution x of a considered COP instance (Φ, F), a *neighborhood* $\mathcal{N}(x)$ defines a set of feasible solutions (points) that are close to x w.r.t. some measure. More formally, a neighborhood presents a map $\mathcal{N} : \Phi \mapsto 2^{\Phi}$, where each feasible solution $x \in \Phi$ is mapped to a subset of feasible solutions, called neighbors of x. A neighborhood function can be defined by means of a collection of operator functions $\Delta : \Phi \mapsto \Phi$ such that

$$x' \in \mathcal{N}(x) \iff \exists \delta \in \Delta, \delta(x) = x'.$$

A natural choice in many applications is the *k-exchange* neighborhood: x and x' are neighbors iff they differ in at most k solution components. As an example, for TSP problem a 2-exchange neighborhood is a natural choice where solution components are edges in the given graph.

A solution x of a problem instance (Φ, F) is *locally optimal* w.r.t. a neighborhood $\mathcal{N}$ iff $F(x) \leq F(x')$ $(F(x) \geq F(x'))$, for all $x' \in \mathcal{N}(x)$, in the case we are minimizing (maximizing). A *globally optimal* solution is that solution which is locally optimal w.r.t. any neighborhood. Note that not every local optimum is globally optimal.

Pseudocode of local search w.r.t. minimization is given in Algorithm 8. The algorithm assumes an instance and a neighborhood as an input. At each iteration, local search improves current solution if a solution in the neighborhood of x is found with a better cost. The algorithm terminates if the current solution is locally optimal w.r.t. neighborhood $\mathcal{N}$. Local search implements a walk through the neighborhood graph. There are two basic

Algorithm 8 Local Search (maximization)

1: **Input:** an instance of a COP, neighborhood $\mathcal{N}$
2: **while** $\{x' \ni \mathcal{N}(x) \mid F(x') > F(x)\} \neq \emptyset$ **do**
3: $\quad x \leftarrow$ choose $x' \in \mathcal{N}(x)$ if $F(x') > F(x)$ holds
4: **end while**

strategies of choosing solutions for replacing incumbent solution under an assumption that there is more than one solution with better cost value. These strategies are: (*i*) *the first improvement* and (*ii*) *the best improvement* strategy. The first improvement rule accepts a first (chosen) solution from the neighborhood with a better cost as a new incumbent whereas the later strategy checks all solutions from the neighborhood and a locally optimum solution is taken as a new incumbent. However, finding the best solution in a neighborhood requires enumerating all solutions in the neighborhood. Hence, performing the best improvement strategy might be too expensive. However, in practice, it frequently happens that applying the best improvement strategy yields a local optimum in a few iterations of the local search.

2.2.4 Iterated Greedy

One straightforward way to improve over the generation of a single greedy solution is to apply the repeated calls of a greedy heuristic in order to generate a variety of different candidate solutions from which we choose the best one. This makes sense only if the greedy heuristic is not fully deterministic, that is, when additional randomization of the construction process is utilized. However, repeated construction of solutions may have disadvantages. Constructing a full solution can be time-consuming and the initial construction steps may require a significant amount of time when compared to later construction phases. One more disadvantage of such approach is that no information is pursued from one solution construction to another one, that is, the repeated construction never uses the knowledge gained from previously constructed solutions. A method that deals with the issues and allows arbitrary time to generate different solutions by constructive heuristics is called *iterated greedy* (IG) [156].

The pseudocode of IG is presented in Algorithm 9. IG assumes an instance of the considered problem and an initial solution (usually obtained from the same greedy heuristic applied in the reparation phase of IG) as the input. The algorithm starts by (randomly) destructing some components of the incumbent solution. That solution, denoted by s', is repaired using a greedy constructive heuristic up to the completion. Note that the destruction phase controls the diversification of the algorithm. The obtained complete solution is (optionally) improved by local search (see more about this technique in Section 2.2.3) which further boosts the performance of IG (to control the intensification). If the acceptance criterion is fulfilled, the obtained solution is being accepted as a new incumbent solution. The simplest criterion would be accepting those solutions with a better cost. If an early convergence is required, the diversification of IG

Algorithm 9 Iterated Greedy (IG)

```
1: Input: an instance of a COP, initial solution s
2: Output: (an improved) solution s
3: while ¬ some termination criterion has met do
4:     s' ← Destroy solution s
5:     s' ← Construct complete solution from s'
6:     s' ← Apply local search at s'    // optionally
7:     if s' accepted then
8:         s ← s'
9:     end if
10: end while
11: return s;
```

could be reinforced by using the acceptance criterion from SA [168], where accepting a worse solution is also a possibility. The above steps are repeated until some termination criterion has reached (e.g. time limit or the maximum number of iterations allowed between two improvements).

2.2.5 Variable Neighborhood Search

One way of extending the local search technique is given with *Variable Neighborhood Descent* (VND) presented by Algorithm 10, where more than one neighborhood is considered. The method relies on the observation that a locally optimal solution w.r.t. some neighborhood might be improved by considering different neighborhoods. VND algorithm consists of systematically changing the neighborhoods. VND assumes a sequence of neighborhood $\mathcal{N}_1^{\text{VND}}, \ldots, \mathcal{N}_{p_{\max}}^{\text{VND}}$ as an input. It starts by performing the local search w.r.t. neighborhood $\mathcal{N}_1^{\text{VND}}$ (as in Algorithm 8). When a locally optimal solution is reached for neighborhood $\mathcal{N}_1^{\text{VND}}$, the successor neighborhood ($\mathcal{N}_2^{\text{VND}}$) is considered next. We perform a local search for the neighborhood $\mathcal{N}_2^{\text{VND}}$ and if a locally optimal solution yields a new incumbent, we move down the search to neighborhood $\mathcal{N}_1^{\text{VND}}$, otherwise, we go to the next neighborhood until all neighborhood structures are examined.

The *General Variable Neighborhood Search* (GVNS) [78] is a metaheuristic that uses neighborhoods for both, intensification and diversification strategy. It performs the procedure of systematically changing neighborhoods as already discussed for the VND in Section 2.2.3. Its pseudocode is given in Algorithm 11. The GVNS is a generalization of the basic VNS [133]. (G)VNS is built upon the following observations: (*i*) a locally optimal solution w.r.t. one neighborhood does not need to be a locally optimal for another neighborhood, (*ii*) a global optimum is a local optimum w.r.t. all possible neighborhoods, and (*iii*) frequently local optimum w.r.t. one or more neighborhoods are close to each other.

The input of the GVNS assumes: (*i*) an initial solution s that is often obtained as

Algorithm 10 Variable Neighborhood Descent (VND)

Input: an instance of a COP, neighborhoods $\mathcal{N}_1^{\text{VND}}, \ldots, \mathcal{N}_{p_{\max}}^{\text{VND}}, p_{\max} \geq 1$
$p \leftarrow 1$
while $p \leq p_{\max}$ **do**
  if $\left\{x' \ni \mathcal{N}_p^{\text{VND}}(x) \mid F(x') > F(x)\right\} \neq \emptyset$ **then**
    $x \leftarrow$ choose $x' \in N_p^{\text{VND}}(x)$ if $F(x') > F(x)$ holds
    $p \leftarrow 1$
  else
    $p \leftarrow p + 1$
  end if
end while

an outcome of a greedy constructive heuristic, (*ii*) a set of shaking neighborhoods $\{\mathcal{N}_1, \ldots, \mathcal{N}_{p_{\max}}\}$, and (*iii*) a set of local search neighborhoods $\{\mathcal{N}_1^{\text{VND}}, \ldots, \mathcal{N}_{q_{\max}}^{\text{VND}}\}$. The set of shaking neighborhoods serves to control diversification and the set of local search neighborhoods serves to control intensification of the method. At each major iteration a solution s' is randomly selected from the first shaking neighborhood w.r.t. incumbent solution s. Then, a VND method has been performed w.r.t. solution s' and the set of local search neighborhoods, returning a solution denoted (again) by s'. If a new solution s' is better than s, it has been replaced by s' and the above steps are repeated by the shaking neighborhood $\mathcal{N}_1$, otherwise the iteration proceeds by using the next (shaking) neighborhood. The major iteration ends when a random solution s' picked from $\mathcal{N}_{p_{\max}}$ after performing a VND step, was worse than current incumbent s. If a termination criterion of the GVNS has not yet been reached, the next major iteration has been performed with the starting (shaking) neighborhood $\mathcal{N}_1$ by repeating the above steps. Otherwise, the algorithm terminates by returning so-far best solution.

The performance of the GVNS is sensitive w.r.t. (*i*) the choice of the neighborhood structures, which is problem-specific, and (*ii*) the order of neighborhoods. Usually, the order among neighborhoods is established w.r.t. the cardinality of used neighborhoods. It is common in practice that the shaking neighborhoods are much larger than the local search neighborhoods. The motivation behind this fact is that randomly sampled solutions should have a strong ability to jump over unexplored regions of Φ.

Note that a *basic* VNS metaheuristic is derived from the GVNS if instead of VND step, a single local search w.r.t. a single neighborhood is performed. The *reduced* VNS (RVNS) is extracted from the GVNS if only the step of selecting random points from the shaking neighborhoods is performed without any local search. RVNS is useful in large problem instances, where applying local search technique is too costly.

Algorithm 11 GVNS metaheuristic

Input: initial solution s of a COP's instance, neighborhoods $\mathcal{N}_1, \ldots, \mathcal{N}_{p_{\max}}$ and $\mathcal{N}_1^{\text{VND}}, \ldots, \mathcal{N}_{q_{\max}}^{\text{VND}}$
Output: (improved) solution s
while $\neg$ (*termination criterion has met*) **do**
    $p \leftarrow 1$
    while $p \leq p_{\max}$ **do**
        $s' \leftarrow$ pick random point from $\mathcal{N}_p(s)$ // shaking phase
        $s' \leftarrow \text{VND}\left(s', \mathcal{N}_1^{\text{VND}}, \ldots, \mathcal{N}_{q_{\max}}^{\text{VND}}\right)$
        if $F(s') > F(s)$ **then**
            $s \leftarrow s'$
            $p \leftarrow 1$
        else
            $p \leftarrow p + 1$ // use next (VNS) neighborhood
        end if
    end while
end while
return s

2.3 Anytime Algorithms

In this section, we describe the class of algorithms called *anytime algorithms* and give the details over two important algorithms that belong to this class: *Anytime Pack Search* described in Section 2.3.1 and *Anytime Column Search* described in Section 2.3.2. They are important pieces in our experimental evaluations.

The next paragraph mainly follows the text from [185] and [186, 187].

In the field of *intelligent systems* it becomes undesirable, and sometimes infeasible, to find the optimal action in each situation due to the complexity of reasoning. The problem is then facilitated to find intelligent systems which make rational decisions after performing the right amount of thinking. It is widely accepted that successful systems must provide a trade-off between decision quality and computational requirements of decision-making. The term *ayntime algorithms* was initially introduced by Dean and Boddy [46], Horvitz [83] and others in late 1980s in the context of the work on time-dependent planning. Nowadays, the following definition of the anytime algorithms is widely accepted: an algorithm belongs to the class of anytime algorithms iff it fulfills the following properties: (*i*) it is able to return high-quality solutions at almost any time when terminated, (*ii*) it gradually improves solution quality as computation time increases, and (*iii*) if enough resources is ensured, it is able to prove optimality. Hence, anytime algorithms offer a trade-off between resource consumption and output quality. An anytime algorithm may also be named an "interruptible algorithm". Anytime algorithms play also important role in the field of *Artificial Intelligence* where usually to solve the

problems the algorithms take a longer time to complete results. Quality of anytime algorithms can be measured in several ways: certainty, accuracy and specificity, see [185].

Many existing programming techniques produce useful anytime algorithms. Examples can be found in iterative deepening search, variable precision logic, randomized techniques, etc. [185]. In the literature, various anytime algorithms are proposed. Based on the construction, anytime algorithms are divided into two groups, A*–based anytime algorithms and BS–based anytime algorithms.

Concerning A*–based approaches, Hansen et al. [77] and Hansen and Zhou [76] proposed *Anytime Weighted* A* which makes use of the heuristic function weighted by a constant factor $w > 0$, i.e, $f(v) := g(v) + w \times h(v)$ (g and h keep the same meaning as in A* search), in order to achieve a quick convergence to a heuristic and usually sub-optimal solution. The authors showed that an obtained solution is an w-approximation if heuristic h is admissible. A generalization of this idea, called *Anytime Restricted* A* (ARA*), was presented in [124]. The main idea is to exchange the constant weight w of Anytime Weighted A* with a linearly decreasing sequence of weights, one for each algorithm iteration. The value of the initial weight has—in general—a significant impact on the convergence of ARA*. Since choosing appropriate weights in ARA* is a problem specific task, Berg et al. [166] proposed *Anytime Non-Parametric* A* (ANA*), eliminating the ad-hoc parameters involved in ARA* by adapting the greediness of the search as path quality improves. Aine et al. [3] proposed *Anytime Window* A* (AWA*), in which the nodes from the open list within one of the levels of depth from a range defined by the window size are expanded, converging to a sub-optimal solution at each iteration. The window size is adapted at each iteration to produce improved solutions. A memory-bounded version of AWA* was proposed by Vadlamudi et al. [163].

Beside the A*–based anytime approaches, the literature offers BS–based algorithms, extended to be anytime algorithms. Most of these algorithms work on the principle of initially performing a single beam search to get reasonably good, suboptimal (heuristic) solutions, and then initializing the beam of subsequent BS runs with nodes which were pruned in previous iterations (see, for example, [182, 183]). However, the literature does not provide a work offering a comprehensive comparison of these algorithms. Recently, Vadlamudi [164] presented *Anytime Pack Search* (APS), showing that it outperforms anytime algorithms such as ANA* and AWA*. This study considers problems from various domains. APS maintains a global priority queue Q. At each iteration, the β most promising nodes from Q are picked and used as an initial beam for the current run of beam search. Note that the nodes for the initial beam may be from different levels of the search tree. When performing the beam search at each iteration, the pruned nodes are being added to Q. In the same paper, the authors proposed a version of APS, called *Anytime Progressive Pack Search* (APPS), which aims at improving the anytime behaviour of APS. This is done by increasing the size of the initial beam (β) dynamically during the search process using a step size parameter each time when no better solution has been found. Otherwise, the beam size is reset to the initial value of β. Experimental results show that APPS can achieve better anytime behaviour than APS.

2.3.1 Anytime Pack Search

This section mainly follows the paper [164].

Anytime Pack Search (APS) quickly produces solutions of reasonable quality and improves them over time by focusing the exploration on a limited set of most promising nodes in each iteration. The main iteration of the algorithm consists of exploring a *pack* (of size K) of most promising not-yet expanded nodes, their K most promising children, and the K most promising children of these children, and so on, until the bottom level is reached. The process is repeated in each iteration with the most promising nodes that are chosen from the set of not-yet-expanded nodes. The effort of APS in a given iteration can be controlled/estimated by the parameter K. APS tends to improve the existing solution or improve the bound on the optimal solution (or both), in each iteration when admissible heuristics are utilized. Note that APS is a complete algorithm.

APS details. The pseudocode of APS is given in Algorithm 12. It takes a search graph G, a starting state node r and the size of *pack* K (parameter of the APS) as input. This parameter can be controlled by the user and it serves for determining the frequency at which solutions are attempted to be produced by APS. The set of not-yet-expanded nodes Q might be realized utilizing a priority queue whose nodes are prioritized according to some (heuristic) evaluator h; at the beginning we have $Q := \{r\}$. At each major iteration, (up to) best K nodes have been popped up from Q and stored into beam B. All successor nodes of each node from B are generated and stored in the set of extensions V_{ext}. If the goal nodes are encountered, complete paths are generated which costs are then compared to the cost of the current best (complete) path. In the case when a better path is found, a new incumbent has reached. At each level of the major iteration, up to $K > 0$ best nodes from V_{ext} are popped up for a new beam B. Concerning the remaining nodes from $V_{\text{rest}} = V_{\text{ext}} \setminus B$, we check for each such node v if (i) v has never been visited before, or (ii) a new best r–v path encountered. If any of these two conditions is fulfilled, v is stored in Q, otherwise, it has been omitted from further search. Afterwards, nodes from B are expanded repeating the steps until beam B is empty. The major iteration of APS is repeated until no unexpanded nodes are left in the search, that is, $Q = \emptyset$.

Computational experiments from the literature showed that the APS significantly outperforms other anytime approaches concerning both, solution quality and gaps quality on a wide range of problems such as the sliding-tile puzzle problem, travelling salesman problem, and single-machine scheduling problem. An interesting variant of the APS algorithm is called *Anytime progressive pack search* (APPS). It may further improve the convergence of the APS algorithms in a time-bounded manner. The value of pack size K is set to `INIT` and is increased at each major iteration of APS by `STEP` units while it is less than the value of `BOUND` (`INIT`, `STEP` and `BOUND` are three parameters of the APPS).

2.3.2 Anytime Column Search

This section mainly follows the paper [165].

Algorithm 12 APS Algorithm

Input: G: a (weighted) search graph, $K > 0$: pack size, r: start node, h: heuristic to prioritize nodes in Q
Output: a best path from r to a goal node in graph G
Initialize: open list $Q \leftarrow \{r\}$, $cost_{\text{best}} \leftarrow 0$ // not-yet-expanded nodes
$P_{\text{best}} \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
    $B \leftarrow$ Pop up to most promising K nodes from Q
    $V_{\text{ext}} \leftarrow \emptyset$
    for each $v \in B$ **do**
        if v is a goal node **then**
            if the cost of $r - v$ path $> cost_{\text{best}}$ **then**
                $P_{\text{best}} \leftarrow$ derive the new best path $r - v$
                $cost_{\text{best}} \leftarrow$ the cost of path P_{best}
            end if
        else
            $V_{\text{ext}} \leftarrow V_{\text{ext}} \cup Successors(v)$ // $Successors(v)$: returns all succ. of v
        end if
    end for
    $B \leftarrow$ pop (up to) K most promising nodes from V_{ext}
    $V_{\text{rest}} \leftarrow V_{\text{ext}} \setminus B$ // remaining nodes
    for each $v \in V_{\text{rest}}$ **do**
        if v not yet visited **then**
            $Q \leftarrow Q \cup \{v\}$
        end if
        if a $r - v$ with better cost found **then**
            $Q \leftarrow Q \cup \{v\}$ // node v is re-opened
            Update the (best) cost of $r - v$ path
        end if
    end for
end while
return P_{best}

Anytime Column Search (ACS) is a complete algorithm which guarantees to produce an optimal solution when terminates. It takes the column-width $\beta > 0$ as a parameter. The basic idea is to explore up to β (column-width) number of most promising not-yet-expanded nodes at *each level* of the state graph and repeat this procedure until the open lists of all levels become empty. The concept of the algorithm fits well with anytime objectives such as finding an initial solution quickly and improving it over time. Visually, the set of nodes expanded in each iteration form a column (see Figure 2.4), and the whole algorithm can be interpreted as a sliding window moving from left to right. When an admissible heuristic is utilized, the best solution obtained in the anytime manner can be

Algorithm 13 ACS Algorithm

Input: G: a (weighted) search graph, $\beta > 0$: column-width, open lists $Q_i, i = 0, \ldots, max\_depth$, r: start node, h: heuristic to guide search
Output: a best path from r to a goal node in graph G
Initialize: open list $Q_0 \leftarrow \{r\}$, $Q_i \leftarrow \emptyset, i = 1, \ldots, max\_depth$, $cost_{\text{best}} \leftarrow 0$
$P_{\text{best}} \leftarrow \emptyset$
while $\exists Q_i \neq \emptyset$ **do**
    for $i \leftarrow 0$ to max_depth **do**
        $B \leftarrow$ pop up β nodes from Q_i
        for each $v \in B$ **do**
            if v is a *goal* node **then**
                if the cost of $r - v$ path $> cost_{\text{best}}$ **then**
                    $P_{\text{best}} \leftarrow$ derive the path $r - v$
                    $cost_{\text{best}} \leftarrow$ the cost of path P_{best}
                end if
            else
                $V_{\text{ext}} \leftarrow V_{\text{ext}} \cup Successors(v)$
            end if
        end for
        for each $v \in V_{\text{ext}}$ **do**
            if $v \notin Q_{i+1}$ **then**
                Add v to Q_{i+1}
            else if a new best cost of $r - v$ path found
                Move node v to Q_{i+1}
                Update new (best) cost of $r - v$ path
            end if
        end for
    end for
end while
return P_{best}

used to cut off nodes which yield suboptimal solutions (admissible pruning).

ACS details. Pseudocode 13 presents the algorithmic steps of the ACS algorithm. At each level i, $1 \leq i \leq max\_depth$, a single open list Q_i (as priority queue) is maintained to keep not-yet-expanded nodes at different levels. These nodes are sorted w.r.t. heuristic h. At each major iteration, (up to) β most promising nodes at each level have been expanded starting with depth 0 (root node r) till the maximum allowed depth (max_depth). Each next major iteration start from the first level where still not-yet-expanded nodes exists, i.e., its priority queue Q_i is non-empty. The algorithm terminates when the priority queues of all levels are empty, i.e., no not-yet-expanded nodes in the search.

Note that in the above description of the algorithm's details, we do not assume that the

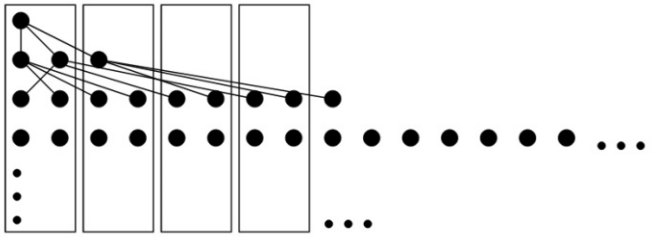


Figure 2.4: An illustration for a working ACS ($\beta = 2$).[1]

heuristic used in the search is admissible. Also, as the goal nodes we usually consider complete nodes. Saying that node v is *complete* means that v has no successor nodes. From the experimental studies on the sliding-puzzle problem, traveling salesman problem, and robotic arm trajectory planning problem [165], the efficacy of the ACS is checked and proven in comparison to some other state-of-the art anytime algorithms from literature.

[1]The figure borrowed from [165].

CHAPTER 3

The Longest Common Subsequence Problem

This chapter addresses the prominent classical *longest common subsequence problem* which has many applications in bioinformatics since it provides a basic measure of similarity between molecular structures. Moreover, specific algorithms developed for this problem can also be used in the Unix command DIFF as well as the version control system Git. In the course of this work, we have developed a general search framework, a generalized BS framework with a novel heuristic guidance which approximates the expected length of an LCS problem. Furthermore, we developed an exact A* search and two anytime algorithms to tackle large-sized LCS problem instances. The algorithms are rigorously tested on a wide range of different random and practical benchmark sets that are commonly used in the literature within the last 20 years.

This chapter is based on the following two papers.

- The conference paper published in *the Proceedings of the 5th International Conference on Machine Learning, Optimization, and Data Science* (LOD 2019) [55]. This paper describes a generalized beam search framework to solve the LCS problem. A novel heuristic guidance based on the expected length calculation of an LCS has been derived. Numerous state-of-the-art results were obtained in relatively short runtimes utilizing the new guiding function into the proposed BS. We emphasize that this paper has been nominated for the best paper award of the conference.
- The extended version of this work has been published in the *Applied Soft Computing* journal (IF=5.472) [58]. In this publication we have extended our studies for the LCS problem towards considering A*–based anytime algorithms. A novel hybrid A*+ACS has been proposed to solve the LCS problem which was able to further boost the quality of the solutions on a wide range of benchmark sets. Moreover,

for those instances where optimality could not be proven, we reported optimality gaps for the first time in the literature for this kind of problems. More precisely, the best gaps were mainly derived by our hybrid A*+ACS in comparison to a few state-of-the-art anytime algorithms from the literature.

3.1 Introduction

Given a set of m input strings $S = \{s_1, \ldots, s_m\}$, the *longest common subsequence* (LCS) problem [128] aims at finding a subsequence of maximal length which is common for all the strings in S. LCS problem provides a popular similarity measure in computational biology. Well-known similarity measures in the context of computational biology include, besides the LCS length, the *Levenshtein* distance which calculates the minimum number of single-character edits (insertions, deletions or substitutions) required to change one sequence into the other. Another example is the *Damerau-Levenshtein* distance [120] which adds transpositions to the three edit operations that are already considered in the Levenshtein distance. Finally, it is also worth to mention the *Canberra* distance (used, for example, to analyze the gut microbiome in different disease states), and the *Google* distance [184]. Well-known similarity measures for sentences and/or texts include metrics such as the *Euclidean*, the *Manhattan* and the *Minkovski* distance [146]. The *soft cosine measure* [153] considers similarities between pairs of features, and the *Jaccard* similarity [110] is defined as the size of the intersection divided by size of the union of two sets. Finally, well-known measures of similarity for time series include *Dynamic Time Warping* (DTW) [144], the *matrix-based Euclidean* distance (GMED), and *matrix-based dynamic time warping* (GMDTW) [181], among others. Recently, many approaches from the field of deep learning and machine learning have been developed to derive measures of similarities that take the semantic meaning of the compared sentences into account. These include *deep architecture Match–SRNN* [171] that utilizes a spatial recurrent neural network to generate the global interaction between two sentences, the *Word Order Similarity* [92] which is defined as the normalized difference of word order between two sentences and the *Latent Semantic Analysis* (LSA) [116]. However, we focus here on the efficient calculation of the LCS measure. Apart from applications in computational biology [96], this problem appears, for example, also in data compression [155, 8], text editing [112], and the production of circuits in field-programmable gate arrays [24].

Concerning exact approaches for the LCS problem, an integer linear programming model has been proposed in [18]. It is, however, not competitive as it cannot be applied to any of the commonly used benchmark instances due to too many variables and constraints in the model. Dynamic programming approaches are reasonable for small m and small n, but they also quickly run out of memory for larger instances and then typically return only weak solutions, if at all. Chen et al. [31] proposed the parallel FAST_LCS search algorithm, which is based on producing a special successors table to obtain all the identical pairs and their levels. Successor nodes are derived in parallel. Pruning operations are utilized to reduce the computational effort. While the algorithm is effective for a small number of input strings, it also struggles for larger m. Wang et al. [173] proposed another

parallel algorithm called QUICK-DP, which is based on the dominant point approach and employs a fast divide-and-conquer technique to compute the dominant points. More recently, Li et al. [123] suggested the Top_MLCS algorithm, which is based on a directed acyclic layered-graph model (called irredundant common subsequence graph) and parallel topological sorting strategies used to filter out paths representing suboptimal solutions. Moreover, the authors showed that the earlier dominant-point-based algorithms do not scale well to larger LCS instances, and Top_MLCS significantly outperforms them. In addition to the sequential Top_MLCS, also a parallel variant was proposed. Another parallel space efficient algorithm based on a graph model, called the Leveled-DAG, was described by Peng and Wang [142]. It eliminates all the nodes in the layered graph that do not contribute to the construction of the LCS, and thus keeps only the nodes from the current level and some previously generated ones. In the experimental comparison, Leveled-DAG and Top_MLCS solved the same number of benchmark instances to proven optimality, but Leveled-DAG consumed less memory. Despite these recent advances, solving practically relevant instances to proven optimality remains a substantial challenge in terms of memory and computation time, even when utilizing many parallel threads. The existing exact methods are therefore frequently not applicable in practice. Concerning the anytime approaches, two anytime algorithms have been proposed in the literature so far to solve the LCS problem: Pro-MLCS [180] and SA-MLCS [179]. Both algorithms are based on the dominant point method [172], which features a special distance measure `dist` for heuristic guidance and a specific multi-dimensional data structure for checking the dominance relation of already explored nodes during the search. Algorithm Pro-MLCS iteratively extends a fixed number of nodes at each level in a level-by-level manner and is similar to anytime column search, see Section 2.3.2. On the other side, SA-MLCS applies an iterative beam widening strategy in successive iterations to reduce space requirements. It differs from Pro-MLCS in the data structures utilized to maintain open nodes. A specific priority queue is realized for SA-MLCS which stores those nodes whose children have not all been expanded, further exploited in the algorithm to make use of the search information from previous iterations to improve efficiency of the SA-MLCS. Last but not least, [179] describes another memory bounded variant of SA-MLCS, called SLA-MLCS. A weakness of all these approaches is that they are not able to provide an upper bound on the solution quality and therefore no quality guarantee in case of early termination. Moreover, neither in [180] nor in [179] enough details are provided concerning the multi-dimensional data structure for checking dominance. This made it, unfortunately, impossible to re-implement the algorithms with all their details, and source code is not provided by the authors. However, in the experimental section of this work we consider the distance measure `dist` as an alternative heuristic guidance and we also build upon anytime column search.

Another branch of work concerns of approximation algorithms. In [95], a simple Long Run (LR) algorithm was proposed that finds an LCS consisting of a single letter, with a $|\Sigma|$-approximation ratio. Bonizzoni et al. [23] developed the so-called Expansion Algorithm (EA), which is also a $|\Sigma|$-approximation algorithm. EA generally outperforms the LR algorithm. Tsai and Tsu [161] introduced an improvement of the EA algorithm. Finally,

two additional approximation algorithms—Enhanced Long Run (ELR) and Best-Next for Maximal Available Symbols (BNMAS)—were proposed in [85].

Concerning heuristic approaches to solve the LCS problem, a break-through in terms of both, computation time and solution quality was achieved by the *Beam Search* (BS) of Blum et al. [16]. This algorithm is an incomplete tree search which relies on a solution construction mechanism based on the Best–Next heuristic and exploits bounding information using a simple upper bound function to prune non promising solutions. The algorithm was able to outperform all existing algorithms at the time of its presentation. Later, Wang et al. [174] proposed a fast A*-based heuristic utilizing a new DP-based upper bound function. Mousavi and Tabataba [135] proposed a variant of the BS which uses a probability-based heuristic and a different pruning mechanism. Moreover, Tabataba et al. [157] suggested a hyper-heuristic approach which makes use of two different heuristics and applies a beam search with a low beam width first to make the decision about which heuristic to use in a successive BS with a higher beam width. This approach was, at that moment, state-of-the-art for the LCS problem. Recently, a *chemical reaction optimization* [93] was also proposed for the LCS and the authors claimed to achieve new best results for some of the benchmark instances. We gave our best to re-implement their approach but were not successful due to many ambiguities and mistakes found within the algorithm's description and open questions that could not be resolved from the paper. The authors of the paper also were not able to provide us the original implementation of their approach or enough clarification. Therefore, we exclude this algorithm from further consideration in our experimental comparison. In conclusion, the currently best performing heuristic approaches to solve also large LCS problem instances are thus based on BS guided by different heuristics and incorporate different pruning mechanisms to omit nodes that likely lead to weaker suboptimal solutions. More detailed conclusions are, however, difficult as the experimental studies in the literature are limited and partly questionable: On the one hand, in [157, 16] mistakes in earlier works have been reported, whose impact on the final solution quality are not known. On the other hand, the algorithms have partly been tested on different LCS instance sets and/or the methods were implemented in different programming languages and the experiments were performed on different machines.

Our contributions to the heuristic solving are as follows. To resolve these computational issues, we proposed a general beam search framework from where we extract all known BS-based methods from the literature in order to compare rigorously and more fairly the methods on all sets of existing benchmark instances. Furthermore, we derive a novel heuristic function for guiding BS that computes an approximate expected length of an LCS.

Our contributions to the exact solving of the LCS problem are as follows. We first propose an exact A* search for the LCS problem. This A* search is shown to be effective for solving small-sized problem instances, but as one may expect it has serious scalability issues similar to other exact methods in terms of time and memory requirements when considering larger instances. We therefore extend this A* search by applying two

alternative hybrid search strategies, turning the original A* search into effective anytime algorithms for finding an LCS. Both follow the idea of interleaving traditional A* search iterations with heuristic search—either BS or anytime column search [165]—and they are labeled A*+BS and A*+ACS, respectively. The A* framework ensures completeness and provides upper bounds at any time, while the embedded heuristic search iterations rely on the expected length calculation heuristic and are responsible for producing a first approximate solution quickly and improving it over time. Most importantly, the heuristic search iterations also operate on the shared list of open nodes of A* search in order to avoid redundant node expansions.

3.1.1 Some Basic Concepts

A string s is called a (valid) *partial solution* to S, if and only if s is a subsequence of each string in S, that is, a *common subsequence* of S.

Any subproblem of S is defined on the basis of a so-called *left position vector* $\mathbf{p} \in \mathbb{N}^m$, with $1 \leq p_i \leq |s_i|$ for $i = 1, \ldots, m$. In particular, for a given $\mathbf{p} = (p_1, \ldots, p_m)$, subproblem $S[\mathbf{p}]$ concerns the substrings $s_i[\mathbf{p}, |s_i|]$ for all $i = 1, \ldots, m$. In other words, $S[\mathbf{p}]$ contains the right part of each string from S starting from the position indicated in the left position vector $\mathbf{p}$. Note that the original problem S can be denoted by $S[\mathbf{p} = (1, \ldots, 1)]$. Given a (partial) solution s to S—that is, a string s that is a common subsequence of S—a subproblem $S[\mathbf{p}]$ is induced by defining $\mathbf{p}$ in the following way. For each $i = 1, \ldots, m$, p_i is determined such that $s_i[1, p_i - 1]$ is the minimal-length string among all substrings $s_i[1, p]$, $p = 1, \ldots, |s_i|$, that contain s as a subsequence. For example, given $S = \{\texttt{abcaac}, \texttt{acbaba}\}$ and the partial solution $s = \texttt{aca}$, the induced subproblem $S[\mathbf{p}]$ is defined by left position vector $\mathbf{p} = (5, 5)$. Note that there is potentially more than one partial solution inducing the same subproblem, respectively, the same left position vector. In the example above, partial solution $s' = \texttt{aba}$, for example, induces the same subproblem and the same left position vector as partial solution $s = \texttt{aca}$. Moreover, partial solutions inducing the same subproblem and the same left position vector may have different lengths. Considering again the example from above, substring $s'' = \texttt{aa}$ induces the same subproblem and the same left position vector as s and s'.

The rest of the chapter is organized as follow. Section 3.2 a state graph of the problem is described. In Section 3.3 a generalized beam search framework has been derived and also a novel search guidance. Also, existing heuristic approaches are expressed by means of this framework. Section 3.4 describes an A* framework to solve the LCS problem. In Section 3.5 two anytime algorithm are proposed. Section 3.6 gives a detailed experimental comparisons between both, exact and heuristic approaches. Finally, Section 3.7 sketches some directions for promising future work.

3.2 State Graph for the LCS Problem

The state graph that is used by all BS variants known in the literature so far, and which will also be used by the A* search proposed in this section, is a *directed acyclic* graph $G = (V, A)$, where a node $v = (\mathbf{p}^{\mathrm{L},v}, l^v) \in V$ represents the set of partial solutions that (i) have the same length l^v and that (ii) induce the same subproblem denoted by $S[\mathbf{p}^{\mathrm{L},v}]$ and left partition vector $\mathbf{p}^{\mathrm{L},v}$. An arc $a = (v_1, v_2) \in A$ between two nodes $v_1 \neq v_2 \in V$—carrying label $\ell(a) \in \Sigma$—exists, if and only if the following two conditions are fulfilled:

1. $l^{v_2} = l^{v_1} + 1$
2. The partial solution inducing v_2 is produced by appending $\ell(a)$ to the partial solution inducing v_1.

The root node r of G corresponds to the original problem S, which is induced by the empty partial solution denoted by ε. In technical terms, $r = ((1, \ldots, 1), 0)$. In order to derive the successor nodes of a node $v \in V$, we first determine the subset $\Sigma_v \subseteq \Sigma$ of letters that can be used to feasibly extend the partial solutions represented by v. Obviously, Σ_v consists of all letters $a \in \Sigma$ that appear at least once in each string of $S[\mathbf{p}^{\mathrm{L},v}]$. For each letter $a \in \Sigma_v$, the position of the first occurrence of a in $s_i[p_i^{\mathrm{L},v}, |s_i|]$ is denoted by $\mathbf{p}^{\mathrm{L},v}{}_{i,a}$, $i = 1, \ldots, m$. Set Σ_v may frequently be reduced by identifying *dominated* letters: We say that letter $a \in \Sigma_v$ *dominates* letter $b \in \Sigma_v$ if and only if $p_{i,a}^{\mathrm{L},v} \leq p_{i,b}^{\mathrm{L},v}$ for all $i = 1, \ldots, m$. Dominated letters can safely be ignored since they always lead to suboptimal solutions. Let $\Sigma_v^{\mathrm{nd}} \subseteq \Sigma_v$ be the subset of those letters that are non-dominated. Graph G contains for each letter $a \in \Sigma_v^{\mathrm{nd}}$ a successor node $v' = (\mathbf{p}^{\mathrm{L},v'}, l^v + 1)$ of v, where $p_i^{\mathrm{L},v'} = p_{i,a}^{\mathrm{L},v'} + 1$, $i = 1, \ldots, m$. A node v that has no successor node—that is, when $\Sigma_v^{\mathrm{nd}} = \emptyset$—is called a *non-extensible* node. Now, note that any path from the root node r to any node in $v \in V$ represents the feasible partial solution obtained by collecting and concatenating the labels of the traversed arcs[1]. Any path from r to a non-extensible node represents a common subsequence of S that cannot be further extended, and any longest path from r to a non-extensible node represents an optimal solution to problem instance S.

3.3 A General Beam Search Framework for the LCS Problem

In the literature for the LCS problem, the so-far leading algorithms to approach larger instances heuristically are all based on Beam Search (BS). Recall that it is an incomplete tree search which expands nodes in a breadth-first search manner. A collection of nodes,

[1] We emphasize that it is not necessary to store actual partial solutions s in the nodes. A longest path to any node in the graph starting from the root node and the respective partial solution can be efficiently derived in a backward manner by iteratively identifying a predecessor in which the l^v-value always decreases by one.

Algorithm 14 A Generalized BS framework (GBSF) for the LCS problem

Input: an instance (S, Σ), heuristic function h to evaluate nodes; upper bound function ub_{prune} to prune nodes; parameter k_{best} to filter nodes (non-dominance relation check); β: beam size (and others depending on the specific algorithm)
Output: a feasible LCS solution
$B \leftarrow \{r\}$
$s_{\text{lcs}} \leftarrow \varepsilon$
while $B \neq \emptyset$ **do**
    $V_{\text{ext}} \leftarrow \texttt{ExtendAndEvaluate}(B, h)$
    Update s_{lcs} if a complete node v with a new largest l^v value reached
    $V_{\text{ext}} \leftarrow \texttt{Prune}(V_{\text{ext}}, ub_{\text{prune}})$ // optional
    $V_{\text{ext}} \leftarrow \texttt{Filter}(V_{\text{ext}}, k_{\text{best}})$ // optional
    $B \leftarrow \texttt{Reduce}(V_{\text{ext}}, \beta)$
end while
return s_{lcs}

called the *beam*, is maintained. Initially, the beam contains just the root node r. In each major iteration, BS expands all nodes of the beam in order to obtain the respective successor nodes at the next level. From those, the $\beta > 0$ most promising nodes are selected to become the beam for the next iteration, where β is a strategy parameter called beam width. This expansion and selection steps are repeated level by level until the beam becomes empty. We will consider several ways to determine the most promising nodes to be kept at each step of BS in Section 3.3.1. The BS returns the partial solution of a complete node with the largest l^v value discovered during the search. The main difference among BS approaches from the literature is the heuristic functions used to evaluate LCS nodes for the selection of the beam and pruning mechanisms to recognize and discard dominated nodes. A general BS framework for the LCS is shown in Algorithm 14.

Procedure $\texttt{ExtendAndEvaluate}(B, h)$ derives and collects the successor nodes of all $v \in B$ and evaluates them by heuristic h, generating the set of extension nodes V_{ext} ordered according to non-increasing h-values. $\texttt{Prune}(V_{\text{ext}}, ub_{\text{prune}})$ optionally removes any dominated node v for which $l^v + ub_{\text{prune}}(v) \leq |s_{\text{lcs}}|$, where $ub_{\text{prune}}(\cdot)$ is an upper bound function for the number of letters that may possibly still be appended, or in other words an upper bound for the LCS of the corresponding remaining subproblem, and $|s_{\text{lcs}}|$ is the length of the so far best solution. $\texttt{Filter}(V_{\text{ext}}, k_{\text{best}})$ is another optional step. It removes nodes corresponding to dominated letters as defined in Section 3.2, but in a possibly restricted way controlled by the parameter k_{best} in order to limit the spent computing effort. More concretely, the dominance relationship is checked for each node $v \in V_{\text{ext}}$ against the k_{best} most promising nodes from V_{ext}. Last but not least, $\texttt{Reduce}(V_{\text{ext}}, \beta)$ returns the new beam consisting of the β best ranked nodes in V_{ext}.

3.3.1 Estimators for Evaluating Nodes

In the literature, several different functions are used for evaluating and pruning nodes, i.e., for h and ub_{prune}. In the following we summarize them.

Fraser [65] used as upper bound on the number of letters that might be further added to a partial solution leading to a node v—or in other words the length of an LCS of the induced remaining subproblem $S[\mathbf{p}^{\mathrm{L},v}]$–by $\mathrm{UB}_{\min}(v) = \mathrm{UB}_{\min}(S[\mathbf{p}^{\mathrm{L},v}]) = \min_{i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1)$.

Blum et al. [16] suggested the upper bound $\mathrm{UB}_1(v) = \mathrm{UB}_1(S[\mathbf{p}^{\mathrm{L},v}]) = \sum_{a\in\Sigma} c_a$, with $c_a = \min_{i=1,\ldots,m} |s_i[p_i^{\mathrm{L},v}, |s_i|]|_a$, which dominates $\mathrm{UB}_{\min}$. The upper bound UB_1 guarantees $|\Sigma|$ performance ratio, that is, $\frac{\mathrm{UB}_1(S)}{\mathrm{LCPS}} \leq |\Sigma|$, for any instance (S, Σ) [84]. While UB_1 is efficiently calculated using smart preprocessing in $O(m \cdot |\Sigma|)$, it is still a rather weak bound. The same authors proposed the following ranking function $\mathrm{Rank}(v)$ to use for heuristic function h. When expanding a node v, all the successors v' of v are ranked either by $\mathrm{UB}_{\min}(v')$ or by $g(v,v') = \left(\sum_{i=1}^{m} \frac{p_i^{\mathrm{L},v'} - p_i^{\mathrm{L},v} - 1}{|s_i| - p_i^{\mathrm{L},v}}\right)^{-1}$. If v' has the largest $\mathrm{UB}_{\min}(v')$ (or $g(v,v')$) value among all the successors of v, it receives rank 1, the successor with the second largest value among the successors receives rank 2, etc. The overall value $\mathrm{Rank}(v)$ is obtained by summarizing all the ranks along the path from the root node to the node corresponding to the partial solution. Finally, the nodes in V_{ext} are sorted according to non-increasing values $\mathrm{Rank}(v)$ (i.e., smaller values preferable here).

$\mathrm{UB}_2^{\text{comp}}$ bound for the LCS problem is based on the standard DP procedure for calculating the LCS of two input strings; see, for example, [174]. More specifically, this algorithm for determining the LCS of two strings s_i and s_j consists in filling a $(|s_i|+1)\times(|s_j|+1)$ matrix M_{ij}, whose entries $\mathrm{M}_{ij}[x,y]$ finally correspond to the lengths of the longest common subsequence for $s_i[x,\ldots,|s_i|]$ and $s_j[y,\ldots,|s_j|]$ with $x = 1,\ldots|s_i|+1,\ y = 1,\ldots,|s_j|+1$. Hereby, all entries with $x = |s_i|+1$ or $y = |s_j|+1$ are set to zero. The content of all other entries is determined with the following recursive formula:

$$M_{ij}[x-1,y-1] = \begin{cases} M_{ij}[x,y] + 1, & \text{if } s_i[x] = s_j[y] \\ \max\{M_{ij}[x,y-1], M_{ij}[x-1,y]\}, & \text{otherwise.} \end{cases}$$

The so-called complete upper bound $\mathrm{UB}_2^{\text{comp}}(v)$ for a node $v \in V$—that is, for the subproblem $S[\mathbf{p}^{\mathrm{L},v}]$ of still relevant substrings—can now be computed as

$$\mathrm{UB}_2^{\text{comp}}(v) := \min_{1\leq i<j\leq m} \left(M_{i,i+1}[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}]\right). \tag{3.1}$$

$\mathrm{UB}_2^{\text{comp}}$ can be calculated efficiently in time $O(m)$ by creating appropriate data structures in preprocessing. By combining the two upper bounds we obtain the so far tightest known bound $\mathrm{UB}(v) = \min(\mathrm{UB}_1(v), \mathrm{UB}_2^{\text{comp}}(v))$ that can still efficiently be calculated in $O(m \cdot |\Sigma|)$ time. This bound will serve in `Prune()` of our BS framework, since it can filter more non-promising nodes than when UB_1 or UB_2 are just individually applied.

Mousavi and Tabataba [135, 157] proposed two heuristic guidances. The first estimation is derived by assuming that all input strings are uniformly at random generated and that they are mutually independent. The authors derived a recursion which determines the probability $\mathcal{P}(p,q)$ that a uniform random string of length p is a subsequence of a string of length q. These probabilities can be calculated during preprocessing and are stored in a matrix. For some fixed k, using the assumption that the input strings are independent, each node is evaluated by $\mathrm{H}(v) = \mathrm{H}(S[\mathbf{p}^{\mathrm{L},v}]) = \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)$. This corresponds to the probability that a partial solution represented by v can be extended by k letters. The value of k is heuristically chosen as $k := \max\left(1, \left\lfloor \frac{1}{|\Sigma|} \cdot \min_{v \in V_{\mathrm{ext}},\, i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1) \right\rfloor\right)$. The second heuristic estimation, the so called *power* heuristic, is proposed as follows:

$$\mathrm{Pow}(v) = \mathrm{Pow}(S[\mathbf{p}^{\mathrm{L},v}]) = \left(\prod_{i=1}^{m} (|s_i| - p_i^{\mathrm{L},v} + 1) \right)^{q} \cdot \mathrm{UB}_{\min}(v),\ q \in [0,1).$$

It can be seen as a generalized form of $\mathrm{UB}_{\min}$. The authors argue to use smaller values for q in case of larger m and specifically set $q = a \times \exp(-b \cdot m) + c$, where $a, b, c \geq 0$ are then instance-independent parameters of the algorithm.

3.3.2 Approximate Expected Length Calculation of an LCS Problem

Some of the BS approaches make use of a heuristic guidance function instead of an upper bound for the selection of the nodes that form the beam of the next iteration. In the following we briefly describe the one that we introduced in [55]. This heuristic guidance function is based on a DP recursion by Mousavi and Tabataba [135], which calculates the probability that any string of length p is a subsequence of a *uniform* random string of length q, for $0 \leq p, q \leq n$, as

$$\mathrm{P}(k,q) = \begin{cases} 0 & \text{if } k > q \\ 1 & \text{if } k = 0 \\ \frac{1}{|\Sigma|} \cdot \mathrm{P}(k-1, q-1) + \frac{|\Sigma|-1}{|\Sigma|} \cdot \mathrm{P}(k, q-1) & \text{else.} \end{cases} \tag{3.2}$$

Let us assume that these probabilities are stored in a matrix P with elements $\mathrm{P}[p,q] \in [0,1],\ 0 \leq p, q \leq n$.

Concerning related work on the expected length of the LCS, Chvátal and Sankoff [39] considered the expected length of two random sequences of length n over an alphabet Σ. The authors derived explicit formulas for small n, and lower and upper bounds for the so-called Chvátal–Sankoff constants $\gamma_{|\Sigma|}$, for $|\Sigma| > 1$, defined as the limits of the ratios between the expected length and n, as n increases towards infinity. These constants are still not known so far, but Dančik and Paterson [44] improved the upper bounds for γ_2 based on the theory of Markov chains. Dixon [51] considered the case for two binary strings of different lengths. He conjectured an approximate upper bound for the expected length under certain additional conditions. Znamenskij [188] came up with the hypothesis of an accurate formula for the expected length for the case of two random strings of

different length and an arbitrary alphabet. An empirical indication for the correctness of this hypothesis is given, and computational experiments showed the precision of the formula with a high accuracy. A proof for Sankoff and Mainville's conjecture about the convergence of $\gamma_{|\Sigma|}$ as $|\Sigma|$ tends toward infinity can be found in [104]. We are not aware of any previous work on the expected length of a LCS for more than two random strings or of an LCPS for random strings.

Let X be the random variable corresponding to the length of an LCS for a set S of randomly generated input strings. Clearly, X can never be larger than the length of the shortest string in S, denoted by $l_{\max} = \min_{i=1,\ldots,m} |s_i|$. The expected length of an LCPS can be expressed as $\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} l \cdot \Pr[X = l]$ with $\Pr[X = l]$ denoting the probability that this length is l. Furthermore, let $Y_l \in \{0, 1\}$ be the random variable indicating if the strings from S have a common subsequence of length l, $l \geq 0$. Observe that the existence of a subsequence of size $l > 1$ always implies the existence of subsequences of size $l' = 0, \ldots, l-1$. Therefore, it holds that $\Pr[X = l] = \mathbb{E}[Y_l] - \mathbb{E}[Y_{l+1}]$ for $l = 0, \ldots, l_{\max}$, i.e., the probability that there exists a subsequence of size l but no longer one. This implies that

$$\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} l \cdot (\mathbb{E}[Y_l] - \mathbb{E}[Y_{l+1}]) = \sum_{l=1}^{l_{\max}} \mathbb{E}[Y_l]. \tag{3.3}$$

In order to approximate $\mathbb{E}[Y_l]$, it can be first observed that—for an alphabet of size $|\Sigma|$—there are $|\Sigma|^l$ different sequences of length l. Following equation (3.2), the probability that a specific sequence s of length l is a subsequence of all strings in S is equal to $\Pr[s \prec S] = \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)$. In the following let us make the simplifying assumption that for each sequence of length l the event of appearing as common subsequence of S is independent of the events of the other sequences. Clearly, this does not entirely hold in reality and an error has been introduced, but it simplifies our considerations to a level with which we can deal further. The probability that S has any common subsequence of length $l \in \mathbb{N}$ can then be approximately expressed as

$$\tilde{\mathbb{E}}[Y_l] = 1 - (1 - \Pr[s \prec S])^{|\Sigma|^l} = 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^l}, \tag{3.4}$$

i.e., the inverse probability of the case that none of the $|\Sigma|^l$ sequences of length l is a common subsequence of S. Ultimately, the approximate expected length of the LCPS can be expressed as

$$\tilde{\mathbb{E}}[X] = l_{\max} - \sum_{l=1}^{l_{\max}} \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^l}. \tag{3.5}$$

Calculating $\tilde{\mathbb{E}}[X]$ directly according to equation (3.5) is in practice hardly possible due to the extremely large power values one obtains for not so small string lengths l. Classical double precision floating point arithmetic is insufficient for strings with already more

than about 40 letters. However, the term from the sum on the right-hand side of (3.5) can be decomposed to

$$\left(1-\prod_{i=1}^{m}\mathrm{P}(l,|s_i|)\right)^{|\Sigma|^{\lceil l/2\rceil}} = \left(\underbrace{\left(\ldots\left(1-\prod_{i=1}^{m}\mathrm{P}(l,|s_i|)\right)^{|\Sigma|^p}\cdots\right)^{|\Sigma|^p}}_{\lfloor l/p\rfloor \text{ times}}\right)^{|\Sigma|^{l \bmod p}} \tag{3.6}$$

for $p \in \mathbb{N}_{>0}$. One could use $p = 25$, for example, which will yield small enough values for all intermediate results when using classical double precision floating point arithmetic.

While this decomposition avoids overflows there are other issues when $\prod_{i=1}^{m} P(l,|s_i|)$ becomes small due to cancellation effects in the limited precision of classical floating point arithmetic. In our implementation, it is specifically checked if $\prod_{i=1}^{m} P(l,|s_i|) < 10^{-10}$ and this case is handled in the following different way.

To ease the further considerations, let us define $x := \prod_{i=1}^{m} P(l,|s_i|)$ and $\alpha := |\Sigma|^l$; we now have to calculate $(1-x)^\alpha$. The numerically problematic situation occurs when x is close to zero and α is large. To resolve this issue, we make use of the fact that $\ln(1-x)/x$ can be well approximated for small x by taking the first two terms of the Taylor series expansion at $x = 0$, which is $-1 - x/2 - o(x)$. This yields

$$(1-x)^\alpha = e^{\alpha\cdot\ln(1-x)} = e^{\alpha x\cdot\frac{\ln(1-x)}{x}} \approx e^{\alpha x\cdot(-1-\frac{x}{2})}. \tag{3.7}$$

Here, however, the product αx may still be numerically problematic to calculate, in fact already the calculation of $\alpha = |\Sigma|^l$ alone may already exceed the limits of a classical double precision floating point arithmetic. We therefore rewrite

$$\alpha x = e^{\ln(\alpha x)} = e^{l\cdot\ln|\Sigma|+\ln(x)} \tag{3.8}$$

and check if already $l\cdot\ln|\Sigma|+\ln(x)$ is so large that the overall result will be negligibly small. More specifically in our implementation, we return zero as result if $l\cdot\ln|\Sigma|+\ln(x) > 300$ as then $(1-x)^\alpha < e^{-e^{300}}$.

Otherwise, $\tilde{\alpha} := \alpha x\cdot(-1-\frac{x}{2})$ is determined. If $\tilde{\alpha}$ is close to zero (i.e., $|\tilde{\alpha}| < 10^{-15}\}$ in our implementation), $1-e^{\tilde{\alpha}} \approx -\tilde{\alpha}$ holds, and consequently (3.7) is approximated well by returning $1+\tilde{\alpha}$.

Last but not least, in the remaining case we consider $\tilde{\alpha}$ to be in a reasonable range so that $e^{\tilde{\alpha}}$ can be calculated in a numerically stable way and this value is returned as approximate result of $(1-x)^\alpha$. Summarizing the above, whenever $x \le 10^{-10}$, we calculate

$$(1-x)^\alpha \approx \begin{cases} 0 & \text{if } l\cdot\ln|\Sigma|+\ln(x) > 300 \\ 1+\tilde{\alpha} & \text{if } l\cdot\ln|\Sigma|+\ln(x) \le 300 \wedge |\tilde{\alpha}| < 10^{-15} \\ e^{\tilde{\alpha}} & \text{else,} \end{cases} \tag{3.9}$$

and for $x > 10^{-10}$, we determine $(1-x)^\alpha$ safely by applying the decomposition rule (3.6).

In order to determine the approximate expected LCS length $\tilde{\mathbb{E}}[X]$ according to (3.5), the terms $(1-\tilde{\mathbb{E}}[Y_l])$ must be calculated for $l = 1,\ldots,l_{\max}$, which requires $O(mn)$ time. In the case of larger n, this would be inefficient and be a bottleneck of our whole approach to solving the LCS problem. To reduce this complexity, the values for most l are interpolated using a divide-and-conquer scheme. This approach exploits the fact that the sequence of values $\{\tilde{\mathbb{E}}[Y_l]\}_{l=1,\ldots,l_{\max}}$ is monotonically decreasing with values in the interval $[0,1]$. The approach starts by defining the artificial border values $\tilde{\mathbb{E}}[Y_0] := 1$ and $\tilde{\mathbb{E}}[Y_{l_{\max}+1}] := 0$ and setting $l = 0$ and $l' = l_{\max}+1$. Then it applies the following recursive principle: If $l+1 < l'$, the values for $\tilde{\mathbb{E}}[Y_l]$ and $\tilde{\mathbb{E}}[Y_{l'}]$ are known but not yet some lying inbetween. In this case, if $\tilde{\mathbb{E}}[Y_l] - \tilde{\mathbb{E}}[Y_{l'}] \leq \varepsilon$ for some sufficiently small ε ($\varepsilon = 10^{-6}$ in our implementation), $\tilde{\mathbb{E}}[Y_{l''}]$ is determined for $l'' = l+1,\ldots,l'-1$ by linear interpolation between $\tilde{\mathbb{E}}[Y_l]$ and $\tilde{\mathbb{E}}[Y_{l'}]$. Otherwise, we calculate the middle value $\tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}]$ according to our approximation above and recursively call the procedure for $\{\tilde{\mathbb{E}}[Y_l],\ldots,\tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}]\}$ and $\{\tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}],\ldots,\tilde{\mathbb{E}}[Y_{l'}]\}$.

Finally, recall that each node $v \in N$ of our state graph represents a subproblem $S[\mathbf{p}^{\mathrm{L},v}]$, and we can determine corresponding approximate expected LCS lengths according to (3.5) and the above described stable and efficient calculation method for these:

$$\mathrm{EX}(v) = \sum_{l=1}^{l^v_{\max}} 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^l}, \tag{3.10}$$

where $l^v_{\max} = \max_{i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1)$. Note that $\mathrm{EX}(v)$, in contrast to the upper bound functions from the previous section, does not possess the property of being admissible in the context of A* search.

3.3.3 Expressing Existing Approaches in Terms of the GBSF

All BS-related approaches from the literature can be defined as follows within our GBSF framework from Algorithm 14.

BS by Blum et al. [16]. The heuristic function h is set to $h = \mathrm{Rank}^{-1}$. Function $\texttt{Prune}(V_{\mathrm{ext}}, ub_{\mathrm{prune}})$ uses $ub_{\mathrm{prune}} = \mathrm{UB}_1$. Moreover, all nodes that are not among the $\mu \cdot \beta$ most promising nodes from V_{ext} are pruned. Hereby, $\mu \geq 1$ is an algorithm-specific parameter. Finally, with a setting of $k_{\mathrm{best}} \geq \beta \cdot |V_{\mathrm{ext}}|$ in function $\texttt{Filter}(V_{\mathrm{ext}}, k_{\mathrm{best}})$, the original algorithm is obtained. Instead of testing this original algorithm, we study here an improved version that uses $ub_{\mathrm{prune}} = \mathrm{UB}$. Moreover, during tuning (see below) we will consider also values for k_{best} such that $k_{\mathrm{best}} < \beta \cdot |V_{\mathrm{ext}}|$. The resulting method is henceforth denoted by Bs-Blum.

Heuristic by Wang [173]. $h = \mathrm{UB}_2$ is used as heuristic function. Moreover, a priority queue to store extensions is used instead of the standard vector structure used in the implementation of other algorithms. Function $\texttt{Prune}(V_{\text{ext}}, ub_{\text{prune}})$ removes all those nodes from V_{ext} whose h-values deviate more than $W \geq 0$ units from the priority value of the most promising node of V_{ext}. Filtering is not used. Instead of $h = \mathrm{UB}_2$ (as in the original algorithm) we use here $h = \mathrm{UB}$, which significantly improves the algorithm henceforth denoted by Bs-Wang.

BS approaches by Mousavi and Tabataba [135, 157]. The first approach, denoted by Bs-H, uses $h = \mathrm{H}$, whereas in the second one, denoted by Bs-Pow, $h = \mathrm{Pow}$ is used. No pruning is done. Finally, a restricted filtering ($k_{\text{best}} > 0$) is applied.

The Hyper-heuristic approach by Mousavi and Tabataba [157]. This approach, henceforth labeled Hh, combines heuristic functions H and Pow as follows. First, Bs-H and Bs-Pow are both executed using a low beam width $\beta_h > 0$. Based on the outcome of these two executions, either Bs-H or Bs-Pow will be selected as the final method executed with a beam width $\beta \gg \beta_h$. The result is the best solution found in both phases.

3.4 A* Search Framework

In this section, we set up an efficient A* to solve the LCS problem. Our A* search for the LCS problem operates on the state graph G as defined in Section 3.2. In the context of the LCS, the cost of a path refers to its length, and a path is better the longer it is. Furthermore, any non-extensible node of the state graph represents a goal node. Good candidates for $h(\cdot)$ in the context of the LCS are upper bound functions, such as the ones discussed in Section 3.3.1. They never underestimate the length of the best/longest path to a goal node and are called *admissible* in the terminology of A* search which guarantees that an optimal solution is found when a goal node is finally selected for expansion and the search terminates. Moreover, the proposed upper bounds are *monotonic*, see Section 2.1.4. To efficiently retrieve the node with the highest priority in each iteration, A* search maintains all open nodes in a priority queue Q. Additionally, our A* search maintains a hash map N with left position vectors $\mathbf{p}^{\mathrm{L},v}$ as keys mapping to the respective l^v-values. By this data structure, we can efficiently recognize already reached left position vectors.

A* search starts with the initialization of Q, that is, $Q = \{r\}$. At each iteration, it expands the top node of Q by generating the respective successor nodes. If its left position vector is not already present in N, a successor node is added to N and Q. If on the other side, the successor's left position vector is already in N, it is checked if the new path to v is longer than the already known one. If this is the case, the l^v-value of v is updated correspondingly, and the priority of v (used for its ranking in Q) is adapted accordingly. The algorithm keeps expanding the top nodes of Q until optimality is reached by selecting

Table 3.1: Overview on the anytime algorithms considered for comparison.

Algorithm	Main idea
A*+BS	embeds a generalized BS in A* search [**?**]: the search strategy switches—every δ regular A* iterations—to BS starting from the highest priority node from A*'s priority queue
ACS	anytime column search [165]: ACS repeatedly iterates over all levels of the state graph, expanding at each level up to β (column width) most promising nodes
APS	anytime pack search [164]: APS maintains a priority queue just like A* search; but instead of expanding only one node in each major iteration, it performs a BS initialized with a certain number (pack size) of the top-ranked nodes from the priority queue
APPS	anytime progressive pack search [164]: this is a variant of APS in which the pack size is adapted based on the observed performance of the BS
A*+ACS	our new approach: interleaves δ regular A* iterations with single ACS iterations of column width β

a goal node or either the memory limit or a time limit is exceeded. One potential problem is that Q typically contains many nodes with the same priority value. These ties are broken by prioritizing those nodes which are farther away from the root node, i.e., the ones with higher l^v values. Remaining ties are broken with the help of a k-norm of the remaining string lengths, i.e., each node is evaluated by $\kappa(v) = \left(\sum_{i=1}^{m}(|s_i| - p_i^{\mathrm{L},v} + 1)^k\right)^{\frac{1}{k}}$. These $\kappa(v)$-values can be seen as a rough heuristic indicator for the cost-to-go, nodes with larger values are expected to be more promising. We used $k = 0.5$ in our implementation. A pseudo-code of our A* search for the LCS problem is given in Algorithm 15. Note that an alternative A* algorithm was proposed in [173]. However, this simpler algorithm uses just the weaker upper bound function UB_2 to guide the search, does not consider tie breaking, and has a larger memory footprint.

3.5 Anytime Algorithms to Solve the LCS Problem

The two new anytime algorithms that we present in this work for the LCS problem are based on the A* framework from Section 3.4. Our main idea is to embed efficient heuristic approaches into the A* framework which is repeatedly executed inbetween regular A* iterations. Our A* anytime variants—apart from providing excellent solutions— can return proven gaps at almost any time when terminated prematurely.

Before outlining our anytime approaches, Table 3.1 summarizes the main ideas of the anytime algorithms that are covered in our experimental evaluation.

3.5.1 A*+BS Approach

Since BS approaches are the state-of-the-art heuristic techniques for the LCS problem but A* search is more promising when it comes to solving smaller instances to proven optimality, it seems sensible to combine A* search with BS into an anytime search method,

Algorithm 15 A* for the LCS problem.

```
1: N: hash map for all reached left position vectors with the lengths of the longest
   paths; Q: priority queue with all open nodes
2: p^{L,r} ← (1, ..., 1)
3: r ← (p^{L,r}, 0)
4: N[p^{L,r}] ← 0
5: Q ← {r}
6: while time and memory limit not exceeded and Q is not empty do
7:     v ← Pop the top node from Q
8:     Σ_v^nd ← non-dominated feasible letters concerning subproblem S[p^{L,v}]
9:     if Σ_v^nd = ∅ then      // v is a goal node
10:        return optimal solution s_lcs retrieved from v
11:    else
12:        for all a ∈ Σ_v^nd do      // expand v
13:            p_i^{L,v'} ← p_{i,a}^{L,v'} + 1, i = 1, ..., m
14:            l^{v'} ← l^v + 1
15:            if p^{L,v'} ∈ N then
16:                if N[p^{L,v'}] < l^{v'} then      // a better path to the node was found
17:                    N[p^{L,v}] ← l^{v'}
18:                    Update priority value of node v in Q
19:                end if
20:            else      // a new node
21:                f_{v'} ← l^{v'} + UB(v')
22:                Add v' of the priority f_{v'} to Q
23:                Add v' to N
24:            end if
25:        end for
26:    end if
27: end while
28: return empty solution ε
```

denoted by A*+BS. At the start of A*+BS, a run of BS with small width is performed for which the beam is initialized with the root node r. This initial BS run takes place to obtain a first reasonable approximate solution (and thus a primal bounds) rather quickly. Then the algorithm proceeds by iteratively applying the following scheme. First, δ traditional iterations of A* search are performed, with $\delta > 0$ being a strategy parameter. Second, a BS run is applied in which the first beam is initialized with the top node of Q. The algorithm stops once optimality is proven or a memory limit, respectively time limit, is exceeded. To avoid redundant recalculations, all the embedded BS calls and the A* search act on the same search tree. All non-expanded nodes encountered during a BS run are used to update the hash map N and are inserted into the priority queue Q (if not already there). Moreover, if a new best path to some node is encountered within any

BS iteration, an update of the corresponding node in N is performed by changing the key to the new l^v-value, and the node is then added into the corresponding beam, that is, the nodes which were already encountered during the search are allowed to be added into V_{ext}.

A pseudo-code for A*+BS is provided in Algorithm 16. Parameters $\beta > 0$ (beam width of BS) and $\delta > 0$ (frequency of BS applications) control the balance between BS and classical A* search iterations and thus the emphasis on improving the primal bound versus the dual bound, respectively. Beam search makes use of a function `Filter`(V_{ext}, k_{filter}) for filtering dominated successor nodes at each step. This procedure works as follows. Up to k_{filter} of the most promising nodes are selected from V_{ext} as a reference set. Then, all other nodes from V_{ext} that are dominated by at least one of these reference solutions are removed from V_{ext}. If $k_{\text{filter}} = 0$, no filtering is applied. Moreover, the employed BS uses the upper bound UB from Section 3.3.1 in order to choose up to β nodes for the beam of the next step. Finally, note that the A* search framework ensures completeness of the A*+BS algorithm and provides proven gaps at any time.

The procedure `ExpandNode` for the expansion of a node and updating the respective data structures is provided in Algorithm 17 (for now, disregard the lines marked to be relevant only for A*+ACS). If the node to be expanded is a goal node, it is checked if it yields a new best solution. If this is the case, s_{best} is updated accordingly. Moreover, if the length of the so-far best solution s_{best} is greater or equal to the f–value of the top node in Q, the flag *opt* is set to *true*, meaning that the search terminates with proven optimality of s_{best}.

3.5.2 A*+ACS Approach

Nevertheless, after an intensive study of the A*+BS algorithm, the following shortcomings of A*+BS were detected:

1. Even though our new upper bound UB is tighter than the UB_1 bound from [54], it is still far from being a tight bound. Therefore, in case of larger instances the nodes with the highest priority values in Q—that is, those nodes that are used to initialize the BS runs—are generally close to the root node of the search tree and the chance that they are promising starting nodes for BS is rather low.

2. The embedded BS does not ensure that the most promising nodes from each level of the state graph are included in the beam corresponding to this level, as only extensions of the starting node of each BS application are considered.

It was observed that these problems lead to the following behaviour. The solution quality of A*+BS at a certain time can often be significantly exceeded by a single BS run whose beam width is chosen such that its computation time is comparable. While the pure BS is no anytime algorithm and does not provide any lower bound, this observation nevertheless indicates room for improvement. Moreover, applying a rather large beam

Algorithm 16 A*+BS for the LCS problem.

1: N: hash map for all reached left position vectors with the lengths of the longest paths; Q: priority queue of not yet expanded nodes; $\beta > 0$: beam width; $\delta > 0$: number of consecutive A* iterations; $k_{\text{filter}} \geq 0$: extent of filtering
2: $s_{\text{best}} \leftarrow \varepsilon$
3: $\mathbf{p}^{\text{L},r} \leftarrow (1, \ldots, 1)$
4: $r \leftarrow (\mathbf{p}^{\text{L},r}, 0)$
5: $N[\mathbf{p}^{\text{L},r}] \leftarrow 0$
6: $Q \leftarrow \{r\}$
7: $opt \leftarrow false$
8: **while** not *opt* **and** neither *memory limit* nor *time limit* exceeded **do**
9: $B \leftarrow$ Pop the β top nodes from Q
10: **while** $B \neq \emptyset$ **do**
11: // perform BS:
12: **for** all $v \in B$ **do**
13: `ExpandNode`(v) // see Alg. 17
14: Store respective children of v in V_{ext}
15: **end for**
16: `Filter`($V_{\text{ext}}, k_{\text{filter}}$) // filter dominated nodes from V_{ext}
17: $B \leftarrow$ `Reduce`(V_{ext}, β)
18: **end while**
19: $iter \leftarrow 0$
20: **while** $iter < \delta$ **and** neither *memory limit* nor *time limit* exceeded **do**
21: // perform A* iteration:
22: $v \leftarrow$ get top node from Q
23: Remove v from Q
24: `ExpandNode`(v) // see Alg. 17
25: $iter \leftarrow iter + 1$
26: **end while**
27: **end while**
28: return s_{best}

width in A*+BS leads to finding good heuristic solutions early, but afterwards, these solutions are hardly improved. On the other side, applying a rather small beam width leads to initial heuristic solutions of lower quality which are improved over time, without, however, reaching the final solution quality of A*+BS when using a rather large beam width.

Therefore, the following potential improvements of A*+BS are proposed here. First, the standard BS component is exchanged with a BS version known as *Anytime Column Search* (ACS), proposed by Vadlamudi et al. [165]. The most interesting feature of ACS is that it expands the most promising open nodes at each level of the state graph. Moreover, the use of the upper bound for guiding ACS is exchanged with the approximation of the

Algorithm 17 `ExpandNode`(v).

Input: a node v to be expanded; a flag parameter
Uses resp. updates: s_{lcs}, N, Q and if called from A*+ACS, Q_j, $j = 0, \ldots, j_{\max}$;
if $\Sigma_v^{\text{nd}} = \emptyset$ **then** // v is a *complete* node
  $s \leftarrow$ derive the non-extensible solution corresponding to v
  if $|s_{\text{lcs}}| < |s|$ **then** // update best sol.
    $s_{\text{lcs}} \leftarrow s$
  end if
else
  for all $a \in \Sigma_v^{\text{nd}}$ **do** // expand v
    $\mathbf{p}^{\text{L},v'}{}_i \leftarrow p_{i,a}^{\text{L},v'} + 1,\ i = 1, \ldots, m$
    $l^{v'} \leftarrow l^v + 1$
    if $\mathbf{p}^{\text{L},v'} \in N$ **then**
      if $N[\mathbf{p}^{\text{L},v'}] < l^{v'}$ **then** // a better path to the node encountered
        $N[\mathbf{p}^{\text{L},v'}] \leftarrow l^{v'}$
        Update priority of the corresponding node in Q;
        if called from A*+ACS **then**
          Move node v' from Q_{l^v} to $Q_{l^{v'}}$
        end if
      end if
    else // create new node
      Add v' to N
      $f_{v'} \leftarrow l^{v'} + \text{UB}(v')$
      Add v' with priority $f_{v'}$ to Q
      if called from A*+ACS **then**
        $e_{v'} \leftarrow \text{EX}(v')$
        Add v' with priority $e_{v'}$ to $Q_{l^{v'}}$
      end if
    end if
  end for
end if
if $|s_{\text{lcs}}| \geq \max_{v \in Q} f(v)$ **then**
  $opt \leftarrow true$
end if

expected length of an LCPS as derived in Section 4.4.

As mentioned above, each BS run in A*+BS starts from the current top node of Q. This means that each BS run only deals with extensions of this single node, and consequently the search space is rather restricted. In particular, many other highly promising nodes at different levels of the state graph may have already been identified, but they are ignored. In order to deal with this potential short-coming, we developed an alternative approach in the line of Section 4.5.3 in which BS runs are exchanged by major iterations of the

above already mentioned *Anytime Column Search* (ACS) [165]; this hybrid approach is henceforth labeled A*+ACS.

Anytime column search is an iterative algorithm which maintains for each level j of the state graph a priority queue Q_j that stores—in the context of the LCS problem—all open nodes v with $l^v = j$, $j = 0, \ldots, j_{\max}$, $j_{\max} = \mathrm{UB}(r)$. Initially, Q_0 contains the root node r and the other priority queues are empty. Each *major iteration* of ACS considers all levels $j = 0, \ldots, j_{\max}$ with non-empty queues Q_j in turn, and expands β nodes (or less if Q_j is shorter. The procedure terminates with an optimal solution once all priority queues are empty. Note that ACS in general finds heuristic solutions very quickly since each major iteration identifies usually at least one non-extensible heuristic solution.

The main idea for combining A* with ACS consists again in interleaving classical A* iterations with major ACS iterations. Hereby, A* keeps working on the basis of priority list Q and the priority function that utilizes the upper bound function $\mathrm{UB}(v)$. In this way, the whole approach will maintain the completeness of classical A* search and $\max_{v \in Q} f(v)$ always is a true upper bound for the optimal solution value. In contrast to Q, the heuristic guidance function EX from Section 3.3.2 is used as sorting criterion for the nodes in the level-specific ACS-queues Q_j. Remember that EX is usually a more promising guidance to find good heuristic solutions, but as it is no valid upper bound, it cannot be used for proving optimality. Moreover, note that changes made in priority queue Q must be accompanied by corresponding changes in priority queues Q_j and vice versa. To enable a direct lookup of priority queue entries for a given node, we make use of the corresponding hash map N.

The pseudo-code of the A*+ACS is presented in Algorithm 18. Note that at each entry of the main while loop (lines 8–32), the algorithm first executes one major iteration of ACS (lines 10–31) and afterwards δ classical A* iterations (lines 22–28). Note that, just like A*+BS, the algorithm potentially makes use of filtering when case $k_{\text{filter}} > 0$ during the major iterations of ACS (line 21). The only difference is that nodes removed from V_{ext} due to filtering are not only removed from N and Q but also from the corresponding queue Q_j. Parameters β and δ play the same role as in A*+BS, namely, controlling the balance between finding good heuristic solutions and improving the dual bound over time. Finally, A*+ACS terminates either with a proven optimal solution, or once the memory limit or the time limit is exceeded, returning the best non-extensible solution found up to this point.

3.6 Computational Studies

The presented BS framework was implemented in C++ and all experiments were performed in single-threaded mode on an Intel Xeon E5-2640 with 2.40GHz and 16 GB of memory. A* search and other competitor anytime variants are run with 32 GB of memory and the maximum computation time for each run was limited to 900 seconds.

Benchmark sets. The related literature offers six different benchmark sets for the LCS

Algorithm 18 A*+ACS for the LCS problem.

```
1: N: hash map for all reached left position vectors with the lengths of the longest paths;
   Q: priority queue of not yet expanded nodes; Q_j: priority queues maintained for
   each level j of the state graph; β > 0: a beam width; δ > 0: amount of consecutive
   A* iterations; k_filter ≥ 0: extent of filtering
2: s_best ← ε
3: p^{L,r} ← (1, ..., 1)
4: r ← (p^{L,r}, 0)
5: N[p^{L,r}] ← 0
6: Q ← {r}; Q_0 ← {r}
7: opt ← false
8: while not opt and neither memory limit nor time limit exceeded do
9:     lev ← 0
10:    while lev < j_max do
11:        // perform ACS iteration:
12:        b ← 0
13:        V_ext ← ∅
14:        while Q_lev ≠ ∅ and b < β do
15:            v ← get the top node from Q_lev
16:            Remove v from Q_lev and Q
17:            ExpandNode(v) // see Alg. 17
18:            Store respective children of v in V_ext    // keep track of nodes for filtering
19:            b ← b + 1
20:        end while
21:        Filter(V_ext, k_filter)    // filter dominated nodes from V_ext
22:        lev ← lev + 1
23:    end while
24:    iter ← 0
25:    while iter < δ and neither memory limit nor time limit exceeded do
26:        // perform A* iteration:
27:        v ← top node from Q
28:        Remove v from Q and Q_{l^v}
29:        ExpandNode(v) // see Alg. 17
30:        iter ← iter + 1
31:    end while
32: end while
33: return s_best
```

problem. The ES benchmark, introduced by Easton and Singireddy [60], consists of 600 instances of different sizes in terms of the number and the length of the input strings, and in terms of the alphabet size. A second benchmark consists of three groups of 20 instances each: Random, Rat and Virus. It was introduced by Shyu and Tsai [152]

for testing their ant colony optimization algorithm. Hereby, Rat and Virus consist of sequences from rat and virus genomes. The BB benchmark of 80 instances was generated by Blum and Blesa [13] in a way such that a large similarity between the input strings exists. Finally, the BL instance set [18] consists of 450 problem instances that were generated uniformly at random.

3.6.1 Computational Experiments: Heuristic Approaches

The five approaches from the literature as detailed in Section 3.3.3 are compare to our own approach, labeled Bs-Ex, which uses $h = \text{EX}$, no pruning, and involving restricted filtering.

The final solution quality produced by any of the considered BS methods is largely determined by the beam size parameter β. Based on the conducted preliminary experiments, we decided to test all algorithms with a setting aiming for a low computation time ($\beta = 50$) and with a second setting aiming for a high solution quality ($\beta = 600$). The first setting is henceforth called the *low-time* setting, and the second one the *high-quality* setting. Note that when using the same value of β, the considered algorithms expand a comparable number of nodes. The remaining parameters of the algorithms are tuned by *irace* [127] for the *high-quality* setting. Separate tuning runs with a budget of 5000 algorithm applications are performed for benchmark instances in which the input strings have a random character (ES, Random, Rat, Virus, BL),[2] and for the structured instances from set BB. 30 training instances are used for the first tuning run and 20 for the second one.

The outcome reported by *irace* for random instances is as follows. Bs-Blum makes use of function $g(.,.)$ within $h = \text{Rank}^{-1}$. Moreover, $\mu = 4.0$ and $k_{\text{best}} = 5$ are used. Bs-Wang uses $W = 10$. For Bs-H we obtain $k_{\text{best}} = 50$, for Bs-Pow we get $k_{\text{best}} = 100$, $a = 1.677$, $b = 0.054$, and $c = 0.074$. Finally, Hh uses $\beta_h = 50$, and for Bs-Ex we get $k_{\text{best}} = 100$.

For the structured instances from set BB *irace* reports the following. Bs-Blum makes use of UB_{min} within $h = \text{Rank}^{-1}$. Moreover, it uses $\mu = 4.0$ and $k_{\text{best}} = 1000$. Bs-Wang uses $W = 10$, Bs-H needs $k_{\text{best}} = 100$, and Bs-Pow requires $k_{\text{best}} = 100$, $a = 1.823$, $b = 0.112$, and $c = 0.014$. Finally, Hh uses $\beta_h = 50$ and for Bs-Ex $k_{\text{best}} = 100$. At this point we want to emphasize that we made sure that the five re-implemented competitor algorithms obtain equivalent (and often even better) results on all benchmark sets than those reported in the original papers.

We now proceed to study the numerical results presented in Tables 3.2–3.7. In each table, the first three columns describe the respective instances in terms of the alphabet size ($|\Sigma|$), the number of input strings (n), and the maximum string length (m). Columns 4–8 report the results obtained with the *low-time* setting, while columns 9–13 report on the results of the *high-quality* setting. The first three columns of both blocks provide the results of the best performing algorithm among the five competitors from the literature. Listed are for

[2] Note that even instance sets Rat and Virus contain sequences that are close to random strings.

Table 3.2: Results on benchmark set Rat.

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
4	600	10	**201**	0.09	Bs-Pow	198	0.22	204	1.18	Bs-Pow	***205**	3.09
4	600	15	**182**	0.10	Bs-Pow	**182**	0.18	184	0.62	Bs-H	***185**	2.65
4	600	20	**169**	0.05	Bs-Pow	168	0.15	170	0.94	Bs-Pow	***172**	2.25
4	600	25	166	0.12	Bs-Pow	**167**	0.18	168	1.01	Bs-Pow	***170**	2.71
4	600	40	**151**	0.04	Bs-H	146	0.15	150	1.02	Bs-Pow	**152**	1.81
4	600	60	149	0.10	Bs-Pow	**150**	0.17	151	1.16	Bs-Pow	***152**	2.27
4	600	80	**137**	0.05	Bs-H	**137**	0.17	139	0.67	Bs-H	***142**	2.47
4	600	100	**133**	0.07	Bs-Pow	131	0.14	135	0.47	Bs-H	***137**	2.50
4	600	150	125	0.06	Bs-H	**127**	0.13	126	0.91	Bs-Pow	***129**	1.97
4	600	200	**121**	0.09	Bs-Pow	**121**	0.17	***123**	0.70	Bs-Pow	***123**	2.65
20	600	10	**70**	0.09	Bs-H	**70**	0.37	***71**	1.86	Bs-H	***71**	3.44
20	600	15	61	0.15	Bs-Pow	**62**	0.28	62	1.40	Bs-H	***63**	2.55
20	600	20	**53**	0.12	Bs-Pow	**53**	0.20	**54**	1.15	Bs-H	**54**	2.45
20	600	25	**50**	0.22	Bs-Wang	**50**	0.21	51	1.09	Bs-H	***52**	2.94
20	600	40	**48**	0.09	Bs-H	47	0.19	**49**	1.15	Bs-Blum	**49**	2.97
20	600	60	**46**	0.09	Bs-H	**46**	0.20	**47**	1.61	Bs-Pow	46	2.42
20	600	80	**43**	0.18	Bs-Blum	41	0.21	***44**	1.14	Bs-H	43	2.64
20	600	100	**38**	0.11	Bs-Pow	**38**	0.23	39	0.96	Bs-H	***40**	2.54
20	600	150	**36**	0.32	Bs-Blum	**36**	0.14	**37**	5.11	Bs-Wang	**37**	2.03
20	600	200	**34**	0.10	Bs-Pow	**34**	0.18	**34**	2.62	Bs-Blum	**34**	2.74

each instance (or instance group) the (average) solution length, the respective (average) computation time, and the algorithm that achieved this result. The last two columns of both blocks present the (average) solution length and the (average) computation time of our new Bs-Ex. The overall best result of each comparison is indicated in bold font, and an asterisk indicates that this result is better than the so-far best known one from the literature. These results allow to make the following observations.

- Concerning the *low-time* setting of the algorithms, the approaches from the literature compare as follows. Bs-H and Bs-Pow seem to outperform the other approaches in the context of benchmarks Rat and Virus, with Bs-Blum and Bs-Wang gaining some terrain when moving towards the alphabet size of $|\Sigma| = 20$. Furthermore, Hh and—to some extent—Bs-H dominate the remaining approaches in the context of benchmarks ES and BL. Concerning the structured instances from set BB the picture is not so clear. Here, the oldest BS approach (Bs-Blum) is able to win over the other approaches in three out of seven cases.

- The results obtained by Bs-Ex with the *low-time* setting are comparable to the best results obtained by the methods from the literature. More specifically, Bs-Ex produces comparable results for Virus and Rat and is able to outperform the other approaches in the context of ES and BL. As could be expected, for the BB instance set, in which the input strings have a strong relation to each other, the EX guiding function cannot successfully guide the search. This is because EX assumes the input strings to be random strings, that is, to be independent of each other.

Table 3.3: Results on benchmark set `Virus`.

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
4	600	10	**225**	0.04	Bs-H	223	0.21	226	0.68	Bs-H	*227	2.88
4	600	15	200	0.04	Bs-H	**201**	0.23	204	0.71	Bs-H	*205	2.24
4	600	20	186	0.05	Bs-H	**188**	0.18	190	0.69	Bs-H	*192	2.69
4	600	25	**191**	0.06	Bs-H	**191**	0.20	*194	0.68	Bs-H	*194	2.20
4	600	40	165	0.04	Bs-H	**167**	0.17	*170	1.21	Bs-Pow	*170	2.24
4	600	60	**163**	0.04	Bs-H	162	0.27	*166	0.69	Bs-H	*166	2.38
4	600	80	157	0.04	Bs-H	**158**	0.19	159	0.72	Bs-H	*163	2.70
4	600	100	153	0.07	Bs-H	**156**	0.19	**158**	0.90	Bs-H	**158**	2.31
4	600	150	**154**	0.06	Bs-H	**154**	0.22	**156**	0.66	Bs-H	**156**	2.37
4	600	200	**153**	0.09	Bs-H	152	0.39	*155	1.22	Bs-H	154	2.63
20	600	10	**75**	0.15	Bs-Pow	74	0.28	*77	2.38	Bs-Pow	76	2.86
20	600	15	**63**	0.16	Bs-Pow	**63**	0.24	*64	1.57	Bs-H	*64	2.91
20	600	20	**59**	0.13	Bs-H	**59**	0.29	**60**	1.58	Bs-H	**60**	2.68
20	600	25	**55**	0.11	Bs-Pow	54	0.20	**55**	1.10	Bs-H	**55**	2.65
20	600	40	**49**	0.08	Bs-H	**49**	0.20	*50	0.85	Bs-H	*50	2.85
20	600	60	**47**	0.16	Bs-Pow	46	0.19	47	1.43	Bs-Blum	*48	3.34
20	600	80	44	0.18	Bs-Blum	**46**	0.30	**46**	1.39	Bs-H	**46**	2.60
20	600	100	**44**	0.14	Bs-H	**44**	0.27	44	2.04	Bs-Blum	*45	2.33
20	600	150	**45**	0.11	Bs-H	**45**	0.24	**45**	2.94	Bs-Blum	**45**	2.75
20	600	200	**43**	0.17	Bs-H	**43**	0.28	**44**	1.69	Bs-H	43	3.17

Table 3.4: Benchmark `Random`. Results are averages over 10 instances per row.

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
4	600	10	217	0.05	Bs-H	**221**	0.20	220	0.84	Bs-H	**221**	2.77
4	600	15	**201**	0.04	Bs-H	**201**	0.20	203	0.78	Bs-H	*204	2.04
4	600	20	**191**	0.10	Bs-Pow	**191**	0.19	192	0.71	Bs-H	*193	2.96
4	600	25	184	0.06	Bs-Pow	**186**	0.18	*187	0.63	Bs-H	*187	3.00
4	600	40	172	0.05	Bs-H	**173**	0.18	173	0.79	Bs-H	*175	2.48
4	600	60	165	0.12	Bs-Pow	**166**	0.17	166	0.77	Bs-H	*168	2.34
4	600	80	159	0.04	Bs-H	**161**	0.20	161	0.83	Bs-H	*163	2.32
4	600	100	**158**	0.05	Bs-H	**158**	0.19	158	0.68	Bs-H	*159	2.18
4	600	150	151	0.06	Bs-H	**152**	0.21	152	1.05	Bs-H	*153	3.00
4	600	200	**150**	0.07	Bs-H	**150**	0.24	*151	1.15	Bs-H	*151	3.11
20	600	10	61	0.09	Bs-H	**62**	0.30	62	1.96	Bs-H	*63	4.12
20	600	15	**52**	0.11	Bs-Pow	51	0.24	*53	2.26	Bs-Pow	*53	3.71
20	600	20	46	0.08	Bs-H	**47**	0.26	*48	1.48	Bs-H	*48	2.61
20	600	25	**44**	0.10	Bs-H	43	0.21	**44**	1.38	Bs-H	**44**	2.54
20	600	40	**38**	0.10	Bs-H	**38**	0.20	38	1.17	Bs-Blum	*39	2.58
20	600	60	**35**	0.08	Bs-H	34	0.19	**35**	1.32	Bs-H	**35**	2.54
20	600	80	32	0.15	Bs-Blum	**33**	0.18	**33**	1.72	Bs-Blum	**33**	2.31
20	600	100	**31**	0.08	Bs-H	**31**	0.17	*32	1.09	Bs-Pow	*32	2.03
20	600	150	**29**	0.24	Bs-Blum	**29**	0.23	**29**	2.27	Bs-Blum	**29**	2.95
20	600	200	**28**	0.11	Bs-H	**28**	0.22	**28**	1.43	Bs-H	**28**	3.31

Table 3.5: Results on benchmark set ES (averaged over 50 instances per row).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
2	1000	10	608.52	0.31	Hh	**609.80**	0.39	614.2	1.42	Bs-Pow	***615.06**	4.43
2	1000	50	533.16	0.33	Hh	**535.02**	0.42	536.46	1.05	Bs-H	***538.24**	4.43
2	1000	100	515.94	0.11	Bs-H	**517.38**	0.46	518.56	1.33	Bs-H	***519.84**	4.82
10	1000	10	199.10	0.53	Hh	**199.38**	0.47	202.72	2.52	Bs-Pow	***203.10**	5.64
10	1000	50	133.86	0.46	Hh	**134.74**	0.35	135.52	2.12	Bs-Pow	***136.32**	3.94
10	1000	100	121.28	0.50	Hh	**122.10**	0.40	122.40	1.50	Bs-H	***123.32**	4.32
25	2500	10	**230.28**	2.33	Hh	223.00	1.57	***235.22**	10.45	Bs-Pow	231.12	19.10
25	2500	50	136.6	1.69	Hh	**137.90**	1.24	138.56	7.23	Bs-Pow	***139.50**	14.51
25	2500	100	120.3	1.74	Hh	**121.74**	1.32	121.62	7.29	Bs-Pow	***122.88**	15.97
100	5000	10	**141.86**	16.12	Hh	139.82	6.98	***144.90**	75.88	Bs-Pow	144.18	91.87
100	5000	50	70.28	9.16	Hh	**71.08**	4.79	71.32	39.11	Bs-Pow	***71.94**	53.54
100	5000	100	59.2	8.71	Hh	**60.04**	4.75	60.06	36.03	Bs-Pow	***60.66**	53.67

Table 3.6: Results on benchmark BL (averaged over 10 instances per row, $|\Sigma| = 4$).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
4	100	10	**34.0**	0.01	Bs-Pow	**34.0**	0.02	***34.1**	0.14	Bs-Pow	***34.1**	0.39
4	100	50	23.7	0.03	Hh	**23.8**	0.02	***24.2**	0.08	Bs-H	***24.2**	0.30
4	100	100	**21.5**	0.03	Hh	**21.5**	0.02	***22.0**	0.23	Bs-Wang	***22.0**	0.27
4	100	150	**20.2**	0.03	Hh	**20.2**	0.02	***20.5**	0.12	Bs-Pow	***20.5**	0.31
4	100	200	**19.8**	0.01	Bs-H	19.5	0.02	***19.9**	0.14	Bs-H	***19.9**	0.31
4	500	10	**182.0**	0.24	Hh	181.2	0.25	***184.1**	1.03	Bs-Pow	184.0	2.41
4	500	50	138.6	0.21	Hh	**139.1**	0.19	140.1	0.90	Bs-Pow	***141.0**	2.13
4	500	100	129.2	0.06	Bs-H	**129.7**	0.18	130.2	1.01	Bs-Pow	***130.8**	2.10
4	500	150	124.7	0.07	Bs-H	**125.5**	0.19	125.9	0.79	Bs-H	***126.4**	2.38
4	500	200	122.6	0.07	Bs-H	**123.0**	0.22	123.2	0.83	Bs-H	***123.7**	2.61
4	1000	10	368.3	0.35	Hh	**368.5**	0.42	373.2	1.80	Bs-Pow	***374.6**	5.22
4	1000	50	284.2	0.36	Hh	**286.2**	0.35	287.0	1.69	Bs-Pow	***288.6**	4.43
4	1000	100	267.5	0.11	Bs-H	**268.8**	0.41	269.5	1.36	Bs-H	***270.6**	4.56
4	1000	150	259.5	0.14	Bs-H	**261.2**	0.47	261.5	1.38	Bs-H	***262.8**	5.30
4	1000	200	254.9	0.17	Bs-H	**256.0**	0.52	256.5	1.81	Bs-H	***257.6**	6.31

Table 3.7: Results on benchmark set BB (averaged over 10 instances per row).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\bar{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\bar{s}_{\text{best}}\|$	$\bar{t}[s]$
2	1000	10	**662.9**	0.33	Hh	635.1	0.44	***676.5**	1.16	Bs-H	673.5	5.49
2	1000	100	**551.0**	0.54	Hh	525.1	0.50	***560.7**	2.10	Bs-Pow	536.6	6.05
4	1000	10	**537.8**	0.43	Hh	453.0	0.48	***545.4**	1.73	Bs-H	545.2	6.24
4	1000	100	**371.2**	0.24	Bs-Pow	318.6	0.53	***388.8**	2.86	Bs-Pow	329.5	5.85
8	1000	10	**462.6**	0.27	Bs-Blum	338.8	0.53	***462.7**	7.93	Bs-Blum	***462.7**	7.90
8	1000	100	**260.9**	0.87	Bs-Blum	198.0	0.67	***272.1**	18.43	Bs-Blum	210.6	8.00
24	1000	10	**385.6**	0.67	Bs-Blum	**385.6**	1.04	**385.6**	13.14	Bs-Blum	**385.6**	16.24
24	1000	100	**147.0**	0.66	Bs-Pow	95.8	0.98	***149.5**	8.01	Bs-Pow	113.3	12.45

- Concerning the results obtained with the *high-quality* setting, the comparison of the algorithms from the literature can be summarized as follows. For all benchmark sets (apart from BB) the best performance is shown by Bs-H and/or Bs-Pow. For benchmark set BB the picture is, again, not so clear, with Bs-Blum gaining some terrain.

- Bs-Ex with the *high-quality* setting outperforms the other approaches from the literature on all benchmark sets except for BB, the latter again due to the strong correlation of the strings. In fact, in 48 out of 67 cases (concerning benchmarks Rat, Virus, ES and BL) Bs-Ex is able to obtain a new best-known result. Moreover, in most of the remaining cases, the obtained result is equal to the so-far best known one.

- Concerning the run times of the approaches, the calculation of EX is done in $O(m \log n)$ time and, therefore, is a bit more expensive when compared to the simpler UB, H, or Pow. However, this is not a significant issue since almost all runs completed within rather short times of usually a few seconds up to less than two minutes.

- We performed Wilcoxon signed-rank tests with an error level of 5% to check the significance of differences in the results of the approaches. These indicate that the solutions of the *high-quality* Bs-Ex are in the expected case indeed significantly better than those obtained from the *high-quality* state-of-the-art approaches from the literature for all except for the Virus benchmark, where no conclusion can be drawn, and the BB benchmark, where the Bs-Ex results are significantly worse due to the strong relationship among the sequences.

Overall, the numerical results clearly show that EX is a better guidance for BS than the heuristics and upper bounds used in former work, as long as (near-) independence among the input strings is given. Finally, note that the result for Random and for the instances with $|\Sigma| > 4$ of set BL are not provided due to page limitations. Full results can be found at https://www.ac.tuwien.ac.at/files/resources/instances/LCS/LCS-report.zip.

3.6.2 Computational Experiments: Exact Approaches

In the following we first provide a summary of the algorithms that are considered for the experimental evaluation. These are our two anytime algorithms (*i*) A*+BS and (*ii*) A*+ACS, (*iii*) the APS algorithm from [164], which is one of the state-of-the-art anytime variants from literature that we implemented for comparison purposes, and (*iv*) A*+ACS-dist which is the variation of A*+ACS in which the heuristic guidance function EX is replaced by the dist(·) estimation from Pro-MLCS [180] and SA-MLCS [179]. Unfortunately, we were not able to do a full comparison to Pro-MLCS and SA-MLCS as the codes could not be obtained from the authors and the description

of the special multi-dimensional tree data structure for determining dominated solutions is insufficient for a re-implementation. However, A*+ACS-`dist` without the classical A* iterations in (i.e., when setting $\delta = 0$) almost corresponds to Pro-MLCS except that instead of the multi-dimensional data structure from [180], Filter$(\cdot, \cdot)$ is used for filtering dominated solutions.

The considered algorithms were evaluated by the quality of the best solutions they provided and by the *percentage gaps*, which are calculated at time $t > 0$ as $gap(t) := \frac{\text{ub}(t) - |s_{\text{best}}(t)|}{\text{ub}(t)} \cdot 100\%$, where $s_{\text{best}}(t)$ denotes the best solution found up to time t and $\text{ub}(t)$ denotes the upper bound on the length of an optimal solution obtained from the f-value of the top node in Q at time t (or the optimal solution value when already available).

Tuning of the Algorithms' Parameters

In order to ensure a fair comparison, the parameters of all considered algorithms were tuned by `irace` [127]. This tuning took place under the same conditions (computation time limit: 900 seconds; memory limit: 32 GB) as later the final experimental evaluation. After conducting some preliminary experiments, we decided to use the following domains for the values of the parameters for the tuning:

- $\delta \in \{0, 1, 10, 50, 100, 500, 1000, 5000, 10000, 20000, 50000\}$,
- $k_{\text{filter}} \in \{0, 1, 10, 50, 100, 500, 1000, 5000, 10000, +\infty\}$,
- $\beta \in \{1, 50, 100, 500, 1000, 5000, 10000, 20000\}$,

Since the parameter *pack* of APS refers to the beam width of that algorithm, we chose the same domain for *pack* and β. As we expected potentially stronger differences in suitable settings for the dependent instances `BB` and the quasi-independent other instances, we decided to apply tuning for these two instance categories separately. We used 40 additional randomly generated instances for the tuning process aimed for quasi-independent instances. The budget of `irace` was set to 5000 optimization runs in this case. On the other side, we generated 20 additional dependent instances for tuning purposes, in the same way as reported in [13]. The budget of `irace` was set to 1000 optimization runs in this second case. In addition to the separation concerning the instance type—quasi-independent versus dependent—we applied for each instance type two tuning runs with different aims. One of these tuning runs aimed for final solution quality, and the other one for small dual gaps. The results of these four tuning runs are reported in the four sub-tables of Table 3.8.

Concerning the tuning results for the quasi-independent instances, note that a higher beam size β is necessary when aiming for solution quality. On the other hand, when we focus on small dual gaps, the amount of A* iterations (δ) has to be significantly increased for all algorithms in comparison. This result appears conclusive when considering that classical A* iterations are primarily important for improving the dual bound. Concerning the tuning results for the dependent instances, we can also notice the requirement of

Table 3.8: Tuning results.

(a) Tuning for solution quality on quasi-independent instances.

Parameter \ Algorithm	A*+BS	A*+ACS	A*+ACS-dist	APS
δ	50	1	100	–
β	500	500	100	–
k_{filter}	1	1	0	0
pack	–	–	–	500

(b) Tuning for small gaps on quasi-independent instances.

Parameter \ Algorithm	A*+BS	A*+ACS	A*+ACS-dist	APS
δ	10000	1000	500	–
β	500	1	1	–
k_{filter}	100	0	0	100
pack	–	–	–	500

(c) Tuning for solution quality on dependent instances.

Parameter \ Algorithm	A*+BS	A*+ACS	A*+ACS-dist	APS
δ	500	500	100	–
β	1000	1	1	–
k_{filter}	1000	1	0	1000
pack	–	–	–	1000

(d) Tuning for small gaps on non-indepenent instances.

Parameter \ Algorithm	A*+BS	A*+ACS	A*+ACS-dist	APS
δ	5000	20000	20000	–
β	1000	50	100	–
k_{filter}	1000	100	100	5000
pack	–	–	–	1000

a higher value for δ when aiming for small gaps. Rather interesting is the large value required for β in the case of A*+BS and APS when aiming for solution quality. In contrast, the tuning procedure has yielded a lower beam size for algorithms A*+ACS and A*+ACS-dist.

Exact Solving with Classical A* Search

Initial tests indicated that our classical A* search is only meaningfully applicable to the smallest instances of the benchmark sets, that is, the instances with string length $n = 100$ from set BL. The corresponding results can be found in Table 3.9, in which we compare the proposed A* approach to the exact solver Top_MLCS [123].

		A*			Top_MLCS		
m	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}[s]$	#opt	$\overline{\|s\|}$	$\bar{t}[s]$	#opt
	4	20.5	428.33	6	0.0	–	0
10	12	12.7	1.73	10	12.7	5.2	10
	20	7.9	0.08	10	7.9	0.28	10
	4	0.0	–	0	0.0	–	0
50	12	6.9	0.17	10	6.9	0.46	10
	20	3.0	0.06	10	3.0	0.08	10
	4	0.0	–	0	0.0	–	0
100	12	5.2	0.08	10	5.2	0.23	10
	20	2.1	0.07	10	2.1	0.08	10
	4	0.0	–	0	0.0	–	0
150	12	4.7	0.07	10	4.7	0.16	10
	20	1.9	0.08	10	1.9	0.08	10
	4	0.0	–	0	0.0	–	0
200	12	4.1	0.07	10	4.1	0.18	10
	20	1.1	0.06	10	1.1	0.11	10

Table 3.9: Classical A* search: average results for benchmark BL, $n = 100$.

In our comparison we made use of the original implementation of Top_MLCS provided by the authors[3]. We remark that Top_MLCS can effectively exploit a parallel hardware architecture, but we performed it in single threaded mode in order to ensure a fair comparison with our A* approach. Besides the instance characteristics, Table 3.9 lists average solution lengths $\overline{|s|}$, average times $\bar{t}$ in seconds until proven optimality has been reached, and the number of instances that could be solved to optimality #opt (out of ten per line) for both approaches.

From Table 3.9 it can be observed that all problem instances with $|\Sigma| \geq 12$ are solved—by both algorithms—to proven optimality, and runtimes are typically only a fraction of a second. However, A* needs significantly less time especially for the instances with $|\Sigma| = 12$. Additionally, our A* approach solved six (out of ten) instances with $|\Sigma| = 4$ and $m = 10$ to proven optimality[4], while Top_MLCS was not able to do so due to running out of memory. None of the instances with $|\Sigma| = 4$ and $m \geq 50$ could be solved to optimality by the two algorithms due to the memory limit.

In summary, A* is able to solve 106 instances from the literature to proven optimality. At this point we would like to stress that the mixed integer linear programming solver Cplex in version 12.9 applied to the LCS model from [18] could not solve any of these instances due to a huge number of variables and constraints.

Anytime Algorithms: comparisons. In contrast to the classical A* search, the anytime algorithms are able to yield meaningful results on all problem instances. Re-

[3]The source code of Top_MLCS can be found at https://github.com/dxslin/mlcs.

[4]All ten instances with $m = 10$, $n = 100$, $|\Sigma| = 4$, could be solved by A* when increasing the memory limit to 40GB.

member that we aim to compare A*+BS and A*+ACS with APS and A*+ACS-dist. Additional reason why the Pro-MLCS [180] and SA-MLCS [179] are not considered in this comparison is because they are not designed for providing gaps upon premature termination. Moreover, we would like to emphasize that—as observed in [55]—the main factor for obtaining high quality solutions is the heuristic guidance function. For this reason we study algorithm A*+ACS-dist which makes use of the heuristic guidance function $\texttt{dist}(v) = \sum_{i=1}^{m} p_i^{\mathrm{L},v}$ from Pro-MLCS and SA-MLCS. As already mentioned, when setting $\delta = 0$ in A*+ACS-dist, we get reasonably close to the original Pro-MLCS algorithm. In the following we report on results both concerning the obtained (average) solution quality and (average) gaps. For improving the readability result tables for benchmark sets Rat, Virus, ES, and BB are given in the main text, whereas the tables for Random and BL are provided in Appendix A.1. More specifically, the results concerning solution quality of the first four data sets can be found in Tables 3.10–3.13, while the corresponding results concerning the gaps are presented in Tables 3.14–3.17. The first three table columns indicate the characteristics of the considered sub-groups of the benchmark sets in terms of $|\Sigma|$, m, and n. Subsequently, the results of the four algorithms are presented. Each of these four blocks consists of four columns listing the following information: the average solution quality ($\overline{|s|}$), the average gaps ($\overline{gap}\,[\%]$), the average time at which the best solution was found ($\overline{t_{\mathrm{best}}}\,[s]$), and the average total runtime ($\bar{t}\,[s]$). The tables showing the results with the parameter settings aiming for solution quality have an additional column labeled *lit. best* that reports the best-known result from the literature for the respective instance, or instance group (without considering the results from the current work). Asterisks in the solution quality column indicate that the best-known result from the literature was beaten. The best result concerning the comparison of the four algorithms considered in this work is always indicated in boldface. Note that in tables presenting results obtained with parameter settings aiming for solution quality, this concerns the columns on the average solution quality. While in tables presenting results obtained with parameter settings aiming for small gaps, this concerns the columns listing the average gaps.

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$
4	10	600	197	41.7	2.15	685.6	*206	39.4	130.6	900.0	197	41.0	866.9	900.0	204	39.5	98.2	731.2	205
4	15	600	180	47.8	11.6	735.3	*189	45.5	740.1	900.0	181	47.5	130.3	900.0	186	45.6	75.4	603.1	185
4	20	600	166	43.3	29.5	900.0	*174	41.2	12.3	900.0	167	42.2	420.9	900.0	171	41.0	71.7	776.0	172
4	25	600	166	51.3	74.4	684.2	*173	49.4	38.3	900.0	166	50.4	212.3	900.0	170	49.4	57.6	642.5	170
4	40	600	152	50.0	570.7	900.0	*154	49.7	32.8	900.0	151	49.8	183.0	900.0	150	50.3	6.5	755.4	153
4	60	600	148	55.6	186.4	900.0	*154	54.0	510.3	900.0	148	55.3	129.1	900.0	151	54.4	384.7	893.9	152
4	80	600	136	52.3	190.8	900.0	*144	49.8	427.9	900.0	137	50.9	308.0	900.0	126	55.0	0.6	754.5	142
4	100	600	134	52.0	180.9	900.0	*139	50.7	458.7	900.0	134	51.6	31.9	900.0	132	52.5	421.7	809.7	137
4	150	600	123	44.3	29.9	900.0	*131	41.0	39.2	900.0	124	43.6	89.8	900.0	110	50.2	848.4	900.0	129
4	200	600	121	46.9	20.8	900.0	*126	45.0	288.0	900.0	121	46.8	23.8	900.0	105	53.7	821.4	900.0	123
20	10	600	69	63.1	5.4	900.0	*72	61.3	136.7	900.0	69	61.9	5.0	900.0	*72	59.8	172.7	900.0	71
20	15	600	61	66.8	5.8	900.0	**63**	65.9	3.8	900.0	61	65.5	44.3	900.1	**63**	64.2	536.0	900.0	**63**
20	20	600	52	68.9	6.4	900.0	*55	68.2	7.1	900.0	53	67.5	66.8	900.0	52	68.3	11.2	900.0	54
20	25	600	50	71.4	7.5	900.0	**52**	70.6	3.4	900.0	51	70.0	34.1	900.0	**52**	69.4	53.9	900.0	**52**
20	40	600	49	72.6	185.3	900.1	*50	72.1	138.6	900.0	49	71.7	11.8	900.0	47	72.5	685.8	900.0	49
20	60	600	46	73.7	138.6	900.0	**47**	73.0	11.5	900.0	46	72.8	15.6	900.0	46	72.3	690.3	900.2	**47**
20	80	600	**44**	71.6	367.0	900.1	**44**	70.5	132.5	900.0	**44**	70.1	145.4	900.3	42	71.4	638.4	900.0	**44**
20	100	600	39	75.8	280.6	900.2	**40**	75.3	6.5	900.0	39	74.5	254.7	900.2	38	74.8	141.9	905.8	**40**
20	150	600	37	76.0	30.3	900.3	*38	75.5	21.4	900.0	37	74.8	30.8	900.0	37	74.4	844.5	900.0	37
20	200	600	33	75.7	137.1	900.2	*35	74.6	144.7	900.0	34	73.0	499.2	900.2	33	73.6	104.0	900.0	34

Table 3.10: `Rat` benchmark. Results when aiming for solution quality.

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$
4	10	600	223	39.9	879.4	900.4	*228	38.2	80.8	900.0	222	39.3	19.2	826.6	221	39.6	394.2	900.0	227
4	15	600	200	45.2	3.7	900.0	*206	43.7	92.5	900.0	200	44.6	527.2	900.0	201	44.5	578.8	629.3	205
4	20	600	185	45.4	276.2	900.0	*194	42.9	327.6	900.0	185	45.1	61.0	900.0	183	45.9	190.8	609.0	192
4	25	600	190	46.8	185.8	900.0	*196	45.3	128.2	900.0	190	46.3	13.3	856.9	190	46.3	341.5	697.2	194
4	40	600	167	51.3	265.6	900.2	*174	49.6	264.0	900.0	167	50.7	191.4	900.0	152	55.2	246.4	678.0	170
4	60	600	162	52.9	185.0	900.0	*168	51.3	49.8	900.0	162	52.4	74.5	900.0	152	55.3	342.0	729.2	166
4	80	600	156	54.1	9.9	900.1	**163**	52.3	61.2	900.0	157	53.4	39.6	900.0	137	59.2	407.3	793.4	**163**
4	100	600	153	55.0	74.7	900.0	*160	53.1	71.5	900.0	153	54.5	79.7	900.0	136	59.5	636.2	872.6	158
4	150	600	152	54.9	19.7	900.0	*157	53.7	40.3	900.0	152	54.6	20.9	900.0	137	59.0	238.9	790.7	156
4	200	600	149	55.5	26.4	900.1	*156	53.6	582.5	900.0	150	54.8	602.6	900.0	133	59.9	310.3	897.3	154
20	10	600	74	60.8	132.2	900.0	**77**	59.3	14.6	900.0	75	59.0	189.1	900.0	76	58.2	26.7	900.0	77
20	15	600	62	66.7	7.4	900.0	**64**	65.8	4.0	900.0	63	65.0	32.4	900.0	**64**	64.2	127.7	900.0	64
20	20	600	58	69.1	7.7	900.1	*61	67.6	28.9	900.0	59	67.8	258.7	900.0	*61	66.5	852.6	900.0	60
20	25	600	53	70.4	7.4	900.1	*56	68.9	82.8	900.0	54	68.8	119.9	900.0	55	68.0	37.0	900.0	55
20	40	600	49	72.9	40.0	900.0	*51	71.8	110.4	902.3	49	71.7	5.1	900.0	49	71.8	118.1	900.0	50
20	60	600	47	73.4	312.9	900.0	**48**	73.0	6.1	900.0	47	72.2	7.0	900.0	47	72.4	837.4	900.0	**48**
20	80	600	45	74.6	744.7	900.2	**46**	74.0	7.1	900.0	45	73.4	8.8	900.1	45	73.4	683.6	900.0	**46**
20	100	600	44	75.0	97.2	900.1	**45**	74.6	8.9	900.0	44	74.3	134.1	900.1	44	74.0	880.7	900.0	**45**
20	150	600	45	75.1	42.6	900.6	*46	74.6	27.7	900.0	45	74.4	48.8	900.3	44	74.7	257.4	900.1	45
20	200	600	43	76.0	60.3	900.2	**44**	75.1	44.8	900.0	43	75.1	65.0	900.5	43	74.7	110.7	900.1	**44**

Table 3.11: `Virus` benchmark. Results when aiming for solution quality.

			A* + BS				A*+ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
4	10	600	198	40.5	354.2	741.0	206	**38.0**	468.2	900.0	198	40.7	454.1	900.0	204	38.6	315.1	683.5
4	15	600	180	46.9	3.3	900.0	187	**44.5**	211.8	900.0	181	46.9	7.3	908.2	186	44.6	419.7	713.7
4	20	600	166	42.2	13.9	900.0	173	**39.5**	384.1	900.0	167	42.2	123.9	902.4	170	40.6	176.4	716.3
4	25	600	165	50.5	116.8	900.0	173	**47.4**	430.3	900.0	166	50.4	336.5	900.0	170	48.3	393.9	769.8
4	40	600	150	50.0	121.3	900.0	154	**48.1**	258.3	900.0	151	49.8	18.5	900.0	151	49.2	11.8	771.6
4	60	600	149	54.6	8.1	900.0	153	**53.1**	215.1	900.0	149	55.0	7.5	900.1	149	54.3	217.8	748.4
4	80	600	136	50.7	205.8	900.0	143	**47.6**	33.8	900.0	137	50.9	359.9	900.0	127	53.5	11.9	755.2
4	100	600	133	51.6	144.6	900.0	138	**49.6**	11.8	900.0	134	51.6	37.5	900.0	127	53.6	332.5	900.0
4	150	600	123	44.1	250.9	900.1	131	**40.2**	519.7	900.0	124	43.6	104.4	900.0	105	52.1	826.2	900.0
4	200	600	121	46.5	22.8	900.0	124	**44.9**	17.2	900.1	121	46.7	24.4	900.2	102	54.7	124.8	900.0
20	10	600	70	59.8	7.0	900.0	71	59.0	20.9	900.1	70	60.5	7.2	900.0	71	**58.7**	234.4	900.0
20	15	600	60	65.1	7.8	900.0	63	**62.9**	5.9	900.1	61	65.3	23.1	900.1	62	63.3	84.6	884.8
20	20	600	53	66.9	358.5	900.1	55	**65.2**	196.4	900.1	54	67.1	48.4	900.1	52	66.9	114.8	900.0
20	25	600	50	69.7	8.2	900.0	52	68.3	15.0	911.1	51	70.0	42.9	900.2	52	**68.1**	583.1	900.1
20	40	600	48	71.6	12.6	900.3	49	**70.3**	137.1	900.0	49	71.8	567.4	900.2	45	72.6	75.4	900.0
20	60	600	45	72.7	18.9	900.4	47	**70.3**	346.6	900.4	47	72.2	652.4	900.2	45	71.5	509.9	900.1
20	80	600	44	69.4	642.3	900.4	43	**69.1**	63.4	900.3	44	70.5	65.4	900.2	40	71.2	243.8	900.1
20	100	600	38	74.3	24.1	900.0	40	**71.8**	175.4	900.3	39	74.5	216.8	900.6	37	73.8	431.1	900.0
20	150	600	37	73.2	31.1	900.7	37	**71.5**	89.5	900.3	37	74.8	31.3	900.5	34	74.0	612.1	900.2
20	200	600	32	77.1	37.6	900.0	34	**70.2**	28.1	900.6	35	72.0	433.3	900.1	31	72.8	152.6	900.2

Table 3.14: `Rat` benchmark. Results when aiming for small gaps.

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$
10	1000	2	604.7	23.5	350.8	866.7	***618.9**	21.7	323.2	900.4	603.6	23.3	254.7	873.3	615.4	21.8	234.7	760.8	615.1
10	1000	10	195.7	57.9	285.9	891.0	***205.0**	55.9	251.3	900.7	195.5	57.5	294.8	888.8	204.1	55.6	230.7	771.9	203.1
50	1000	2	526.6	33.0	287.3	897.5	***540.9**	31.2	302.2	900.0	526.9	32.6	264.1	887.4	532.8	31.9	301.2	696.2	538.2
50	1000	10	131.0	71.3	219.88	900.2	***137.5**	69.9	158.1	900.0	131.2	71.0	137.5	900.2	134.5	70.2	321.1	867.3	136.3
100	1000	2	509.1	35.1	250.8	900.1	***522.1**	33.4	324.6	900.0	509.4	34.8	283.8	900.0	512.5	34.4	274.5	781.6	519.8
100	1000	10	118.8	73.9	217.7	900.2	***124.1**	72.7	121.0	900.0	118.9	73.6	175.9	900.1	120.5	73.1	356.0	900.0	123.3
10	2500	25	224.5	72.1	276.4	900.0	235.0	70.7	419.5	900.4	223.9	72.0	263.5	900.0	***236.6**	70.4	374.8	897.3	235.2
50	2500	25	132.1	83.3	217.1	900.1	***140.4**	82.3	239.8	900.0	132.6	83.1	212.8	900.3	136.5	82.6	368.1	900.0	139.5
100	2500	25	116.8	85.2	268.2	900.6	***123.4**	88.1	223.6	900.0	117.0	85.1	350.8	900.7	118.6	84.8	352.7	900.1	122.9
10	5000	100	137.5	84.0	338.2s	900.4	***145.7**	84.7	434.3	900.2	136.8	84.1	392.8	901.1	144.6	83.1	340.6	900.1	144.9
50	5000	100	67.4	92.0	355.8	902.8	***72.0**	97.6	286.1	900.1	67.6	91.9	432.1	902.7	69.6	91.9	330.6	900.7	71.9
100	5000	100	57.1	93.1	584.4	906.6	***60.8**	97.4	515.7	900.1	57.1	93.1	601.9	905.8	57.9	93.6	382.3	901.5	60.7

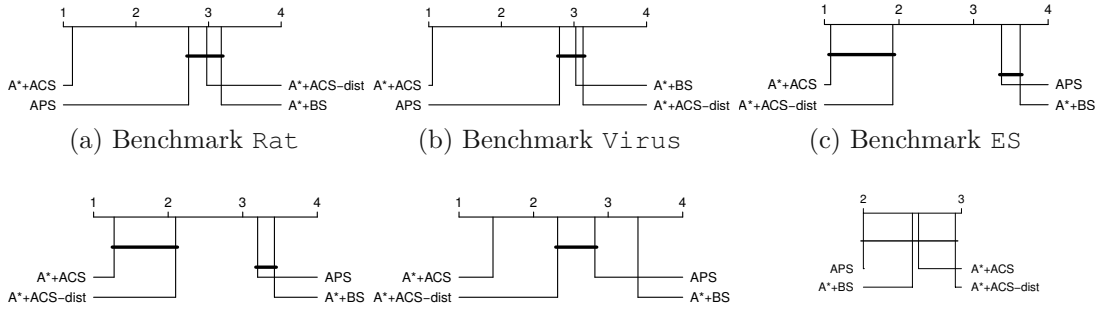
Table 3.12: ES benchmark. Results when aiming for solution quality (averaged over 50 instances per row).

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$
2	10	1000	675.1	16.4	64.9	900.0	676.6	16.5	347.1	900.0	675.7	15.6	57.1	900.0	***676.7**	16.4	152.0	885.4	676.5
2	100	1000	561.3	31.2	260.3	900.0	547.1	32.6	497.6	900.0	***563.6**	30.6	264.3	900.0	486.7	40.0	464.9	870.0	560.7
4	10	1000	545.2	30.3	13.0	900.0	***545.5**	30.7	204.9	900.0	545.2	29.4	13.4	900.0	***545.5**	30.5	85.6	798.4	545.4
4	100	1000	389.4	51.1	209.8	900.4	344.3	56.4	503.4	900.0	***390.2**	50.9	362.7	900.0	273.6	65.4	291.2	881.0	388.8
8	10	1000	**462.7**	39.0	17.0	900.0	**462.7**	39.9	68.4	900.0	**462.7**	38.0	19.2	900.0	**462.7**	39.7	16.9	827.2	**462.7**
8	100	1000	273.1	65.1	143.7	900.1	223.7	71.1	631.7	900.1	***273.4**	65.0	179.8	900.1	164.7	78.7	408.7	900.0	272.1
24	10	1000	**385.6**	42.0	43.8	900.0	**385.6**	47.0	33.8	900.0	**385.6**	40.5	35.5	900.0	**385.6**	46.8	8.5	900.0	**385.6**
24	100	1000	149.4	79.5	138.0	900.4	117.0	83.5	636.9	900.3	149.4	79.5	145.2	900.4	83.8	88.2	550.2	900.0	**149.5**

Table 3.13: BB benchmark. Results when aiming for solution quality (averages over ten instances per row).

			A* + BS				A*+ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
4	10	600	222	38.7	3.0	910.7	228	**36.8**	637.5	900.0	224	38.1	20.7	826.2	221	38.4	550.5	649.1
4	15	600	200	44.4	48.3	900.0	206	**42.5**	21.9	900.0	201	44.2	702.0	746.8	200	44.1	333.0	579.2
4	20	600	185	44.9	406.2	900.0	192	**42.7**	246.2	900.0	186	44.8	139.6	900.0	183	45.4	583.8	592.5
4	25	600	189	46.3	4.8	903.8	196	**44.2**	850.5	900.0	190	46.3	21.5	900.0	188	46.3	263.9	575.7
4	40	600	166	50.7	13.5	900.0	173	**48.4**	866.7	900.0	167	50.9	846.3	900.0	152	54.6	338.5	609.3
4	60	600	162	51.9	42.7	900.0	168	**49.9**	645.2	900.0	162	52.4	85.5	900.0	150	55.2	472.9	757.9
4	80	600	157	53.1	12.3	900.0	163	**50.8**	114.6	900.0	157	53.4	11.1	900.0	134	59.5	327.0	861.3
4	100	600	153	54.2	14.0	900.0	160	**51.5**	806.0	900.0	153	54.5	14.5	900.1	133	59.7	715.9	900.0
4	150	600	152	54.4	143.6	900.0	157	**52.6**	415.1	900.1	152	54.6	802.9	900.1	136	58.9	725.0	900.1
4	200	600	150	54.7	645.8	900.0	155	**52.7**	415.1	900.2	150	54.8	729.9	900.0	132	59.8	151.7	900.0
20	10	600	74	58.4	27.7	900.0	77	**56.5**	320.2	900.0	75	58.8	111.8	900.1	76	56.8	255.7	900.0
20	15	600	62	64.8	8.2	900.1	64	**62.6**	3.1	900.0	63	65.0	30.6	900.1	64	**62.6**	851.2	900.0
20	20	600	58	67.6	9.0	900.0	60	65.9	33.8	900.0	59	67.8	19.0	900.1	60	**65.7**	57.5	900.0
20	25	600	53	68.5	9.7	900.0	55	66.7	26.1	901.4	53	69.4	9.6	900.0	55	**66.5**	690.9	900.1
20	40	600	49	71.3	579.5	900.0	50	**70.1**	52.2	900.3	49	72.0	25.7	900.2	48	71.1	162.7	900.1
20	60	600	47	72.0	18.5	900.0	48	**70.4**	107.1	900.2	47	72.5	18.1	900.4	45	72.2	316.6	900.1
20	80	600	45	73.1	283.9	900.0	46	**71.4**	22.2	900.3	45	73.7	67.1	900.2	44	72.7	736.1	900.0
20	100	600	43	74.1	27.3	900.0	45	**72.0**	190.9	900.4	44	74.3	113.1	900.2	43	73.3	653.6	900.2
20	150	600	44	76.0	49.4	900.0	45	**72.7**	48.8	900.0	45	74.6	185.4	900.4	43	73.9	229.3	900.3
20	200	600	43	75.8	65.0	900.0	43	**73.3**	105.9	900.4	43	75.0	64.9	900.3	42	73.8	236.9	900.1

Table 3.15: Virus benchmark. Results when aiming for small gaps.



(a) Benchmark `Rat` (b) Benchmark `Virus` (c) Benchmark `ES`

(d) Benchmark `Random` (e) Benchmark `BL` (f) Benchmark `BB`

Figure 3.1: Critical difference plots concerning solution quality.

			A* + BS				A*+ACS				APS				A* + ACS-dist			
n	m	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
1000	10	2	606.4	22.8	177.3	881.0	618.1	**21.2**	427.4	900.0	607.5	22.7	165.9	881.7	614.9	21.6	269.3	818.2
1000	10	10	196.9	56.8	205.9	900.0	204.2	**54.9**	283.2	900.1	198.0	56.9	205.9	900.0	203.5	55.0	284.6	707.5
1000	50	2	526.4	32.6	300.6	892.9	540.3	**30.6**	377.0	900.0	526.6	32.7	267.5	900.0	532.6	31.6	349.2	799.5
1000	50	10	130.6	70.9	160.1	900.0	137.1	**69.1**	294.6	900.3	131.3	71.0	206.1	900.1	133.7	69.8	293.7	836.6
1000	100	2	508.9	34.8	265.1	900.0	521.6	**32.9**	336.9	900.1	509.4	34.8	297.5	900.0	512.0	34.1	440.7	875.2
1000	100	10	118.6	73.4	112.4	900.1	123.7	**71.9**	287.4	900.2	119.0	73.6	191.8	900.1	119.6	72.8	378.4	900.0
2500	10	25	226.6	71.5	179.5	900.6	231.5	70.6	576.4	900.1	227.5	71.5	244.3	900.1	235.2	**70.1**	443.9	900.0
2500	50	25	131.9	83.3	181.1	900.4	139.5	**81.9**	388.9	900.3	132.4	83.2	261.9	900.4	135.3	82.5	424.6	900.3
2500	100	25	116.5	85.2	221.0	900.6	122.7	**84.0**	360.3	900.5	117.0	85.1	362.6	900.7	117.6	84.7	405.0	900.2
5000	10	100	138.9	83.8	327.7	901.0	143.4	**82.9**	643.8	900.8	139.3	83.8	414.2	900.9	143.0	83.0	466.8	900.3
5000	50	100	67.3	92.0	337.6	902.4	71.0	**91.3**	470.8	903.5	67.5	92.0	411.2	902.6	68.3	91.6	536.0	902.0
5000	100	100	57.0	93.1	575.9	900.0	59.6	**92.6**	488.6	907.3	57.0	93.1	626.2	905.7	56.2	93.0	518.6	903.4

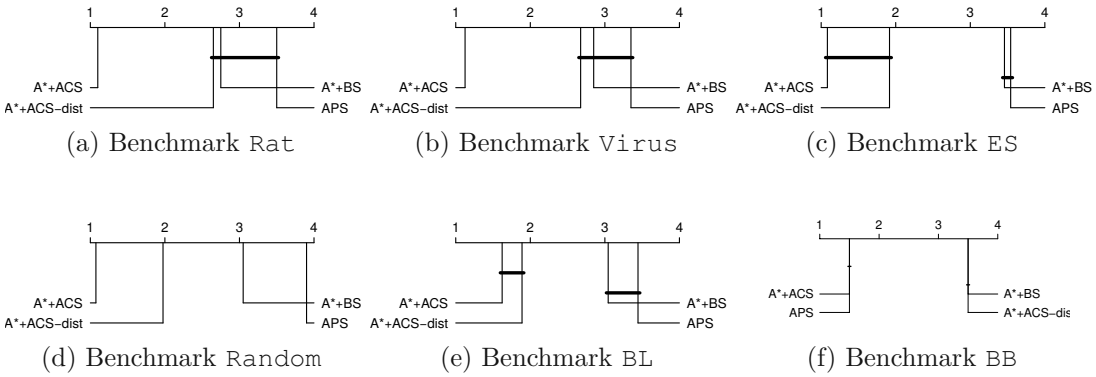
Table 3.16: `ES` benchmark. Results when aiming for small gaps (averages over 50 instances per row).

			A* + BS				A*+ACS				APS				A* + ACS-dist			
$\lvert\Sigma\rvert$	m	n	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
2	10	1000	676.7	16.6	155.8	900.3	675.4	**16.2**	18.1	900.0	674.6	**16.2**	115.0	900.0	676.7	16.5	63.9	884.6
2	100	1000	547.4	32.6	428.3	900.5	561.8	31.0	357.3	900.0	563.2	**30.9**	557.2	900.0	486.5	40.1	398.6	864.0
4	10	1000	545.5	30.7	98.4	901.0	545.2	**30.0**	14.4	900.0	545.2	**30.0**	60.4	900.0	545.5	30.6	42.1	853.6
4	100	1000	346.5	56.1	507.0	901.0	389.2	**50.9**	186.7	900.0	389.4	51.1	400.2	900.0	270.2	65.8	487.3	859.6
8	10	1000	462.7	39.8	15.6	900.5	462.7	38.7	18.8	900.0	462.7	**38.6**	89.6	900.0	462.7	39.6	7.5	900.2
8	100	1000	224.1	71.0	524.3	901.9	273.0	**65.1**	106.8	900.1	272.9	**65.1**	388.1	900.1	160.9	79.1	575.0	901.4
24	10	1000	385.6	46.3	1.5	901.1	385.6	41.6	34.1	900.0	385.6	**41.2**	122.8	900.0	385.6	46.0	1.7	900.8
24	100	1000	120.9	82.9	657.5	908.4	149.4	**79.5**	138.6	900.4	149.3	**79.5**	580.9	900.6	80.6	88.6	606.8	910.1

Table 3.17: `BB` benchmark. Results when aiming for small gaps (averaged over ten instances per row).

A study of the numerical results allows to draw the following conclusions:

- A*+ACS generally outperforms the three competitors in terms of solution quality in the context of instances with quasi-independent input strings (that is, benchmark sets `Rat`, `Virus`, `Es`, `Random` and `BL`). The only exception is benchmark set `BB`, in which instances consist of dependent input strings. The reason for this behavior



(a) Benchmark Rat (b) Benchmark Virus (c) Benchmark ES

(d) Benchmark Random (e) Benchmark BL (f) Benchmark BB

Figure 3.2: Critical difference plots concerning the obtained gaps.

is clearly that the heuristic guidance function EX() works in general very well for instances with quasi-independent input strings, while it tends to mislead for instances with related input strings. Observe in Table 3.13 that the performance of A*+ACS and of A*+ACS-dist strongly decreases, especially when the instances consist of many input strings ($m = 100$). A more visual presentation of the results is provided in Figures A.1–A.4 in Appendix A.2, where the improvement of A*+ACS over the three competitors is shows in percent.

- In order to check the statistical significance of differences, Friedman's test was performed simultaneously for all four anytime approaches. Given that in all cases the test rejected the hypothesis that the algorithms perform equally, pairwise comparisons were performed using the Nemenyi post-hoc test [48]. The outcome is shown in Figure 3.1 by means of so-called critical difference plots, one for each benchmark set. In short, each algorithm is positioned in the horizontal segment according to its average ranking concerning the considered set of instances. Then, the critical difference (CD) is computed for a significance level of 0.05 and the performance of those algorithms that have a difference lower than CD are considered as equal—that is, no difference of statistical significance can be detected. This is indicated in the graphics by horizontal bars joining the respective algorithms. Figure 3.1 shows that A*+ACS produces significantly better results concerning solution quality for benchmark sets Rat, Virus, and BL. The differences observed for benchmark sets Random and ES are statistically not significant (despite the fact that A*+ACS produces new state-of-the-art results in 13 out of 20, and in 10 out of 12 cases, respectively).

- Just like classical A*, both A*+ACS-dist and A*+ACS are able to prove optimality for the instances of benchmark set BL with $n = 100$ and $|\Sigma| \geq 12$. This is indicated by entries with value 0.0 in columns with heading $\overline{gap}$ [%] (see Table A.2). However, as expected, more computation time is needed than by A*.

- A*+ACS does not only beat the competitors we considered here. It performs also

very favorably in comparison to purely heuristic state-of-the-art approaches from the literature. This can be observed by comparing the performance of A*+ACS with the last columns in Tables 3.10–3.13 and Tables A.1–A.2 which contain the so far best known results from the literature[5]. Overall A*+ACS was able to obtain new best-known results in 82 out of 117 cases (table rows).

- For what concerns the performance of the four algorithms with respect to the produced gaps—see Tables 3.14–3.17 and Tables A.3–A.4 (from Appendix A.1)—it can be observed that—also in this case—A*+ACS generally outperforms the competing algorithms. This is with the exception of benchmark set `BB`, where no clear tendency can be identified. The statistical significance of this conclusion is tested in the same way as done for the case of aiming for solution quality. The corresponding critical difference plots are shown in Figure 3.2. Moreover, the improvements of A*+ACS over the competitors are graphically shown in Figures A.5–A.8 (Appendix A.2).
- Nevertheless, observe that A*+ACS-`dist` often produces better final gaps than A*+ACS for instances with a low number of input string. This is the case, for example, for instances with $n = 10$ from benchmark set `BL`; see Table A.4 and Figure A.5. A possible explanation for this behavior is as follows. On the one hand, the heuristic guidance function `dist`$(\cdot)$ performs rather well for instances with a low number of input strings (that is, a low m-value), which means that A*+ACS-`dist` will not have a major disadvantage with respect to A*+ACS in those cases. On the other hand, `dist`$(\cdot)$ requires less computation time than the EX function used by A*+ACS. This implies that A*+ACS-`dist` is able to perform more node expansions than A*+ACS within the allowed computation time, which leads to better upper bounds.

Comparison of the Algorithms' Anytime Behavior

.

So far we have only studied the final results of the algorithms for what concerns solution quality and gaps. However, in the context of anytime algorithms, another important aspect to take into account is their anytime behavior. In order to visualize the anytime behavior of the algorithms, we plot the evolution of the solution quality, respectively the gaps, over time (either averaged over all problem instances of the same specifications, or averaged over multiple runs for single problem instances, depending on the benchmark set). The plots concerning solution quality are shown, for seven representative cases, in Figure 3.3, while the ones concerning the gaps are shown, for the same seven cases, in Figure 3.4. In addition to the curves showing the average behavior, these graphics also contain boxplots—shown every 200 seconds—indicating the variability of the algorithm performance.

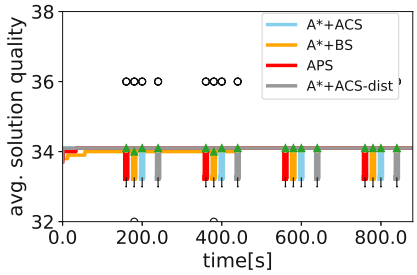
[5] As the best result for a specific group of instances from the literature we took the maximum average solution quality among the reported averages (if any) from [16, 135, 55, 157, 18]. Most of best results so far are from BS-EX [55].

The following observations can be made concerning the anytime plots on solution quality:

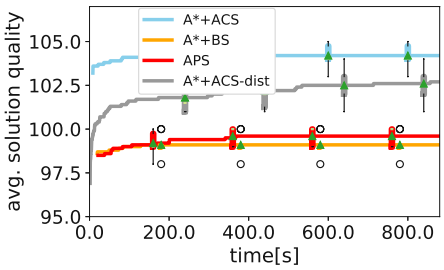
- A*+ACS generally finds solutions of higher quality than the other algorithms in early stages of the search process. The main reason for this is clearly the heuristic guidance function EX() which is utilized in A*+ACS.
- Notice also that A*+ACS is able to find improving solutions more frequently than A*+BS or APS. For these latter algorithms it seems much harder to find improving solutions at later stages of the search process. Even though A*+ACS-`dist` can be said to generally outperform APS and A*+BS, it can not match the performance of A*+ACS. It can also be observed that the compared algorithms find improving solutions in general more frequently when the alphabet size is rather small.
- APS and A*+BS, which make both use of an embedded BS to find heuristic solutions, show a similar evolution of solution quality over time. It is noticeable that a rather large beam size β is required to achieve the best possible anytime performance within the given computation time.

Concerning the anytime performance of the algorithms with respect to the gaps we can make the following observations:

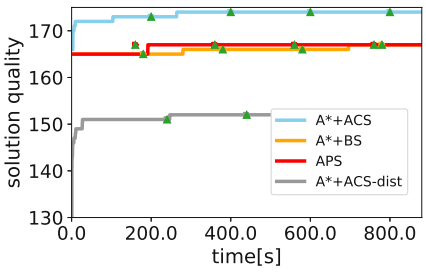
- For the smallest ones of the considered instances—that is, the instances from benchmark set `BL` with $n = 100$—A*+BS shows the best evolution of the obtained gaps (see Figure 3.4a). This is for the following two reasons: (*i*) the parameter values identified by our tuning process allow a significant amount of A* iterations, which is crucial for obtaining a favorable evolution of the gaps, and (*ii*) near-optimal solutions are easily obtained for these instances by any of the algorithms.
- Concerning the remaining medium-size and large–size instances, A*+ACS shows a better anytime performance concerning the gaps for the `Virus`, `Rat`, `Random`, and `ES` benchmarks; see Figures 3.4b–3.4f. This is because the ACS-iterations, even with a rather low value of β, are still able to find rather high-quality primal solutions, while a significantly increased number of A*-iterations (in comparison to the parameter setting used when aiming for solution quality) provides improved upper bounds. In this sense, A*+ACS is an algorithm that is much better balanced than the competitors.
- In the case of small alphabet sizes, A*+ACS-`dist` is not able to keep up with the performances of the other algorithms (see Figures 3.4c and 3.4d). Mainly responsible for this is the heuristic function `dist`$(\cdot)$, which provides a weaker guidance than in particular EX for finding good primal bounds, especially in the case of small alphabet sizes.
- APS and A*+BS show a similar behavior concerning the evolution of the average gaps over time. The necessity of working with a large beam width (β) hinders



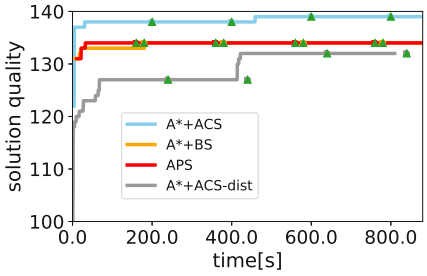
(a) Benchmark set BL: 10 instances with $|\Sigma| = 4, m = 10, n = 100$.



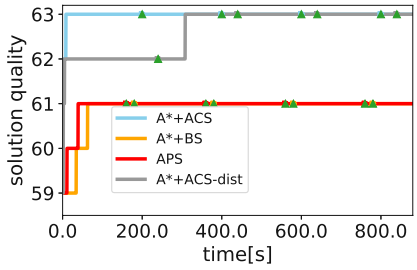
(b) Benchmark set BL: 10 instances with $|\Sigma| = 12, m = 50, n = 1000$.



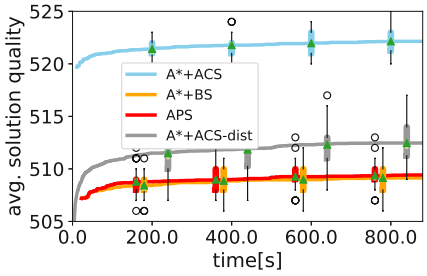
(c) Benchmark set Virus: instance $|\Sigma| = 4$, $m = 40, n = 600$.



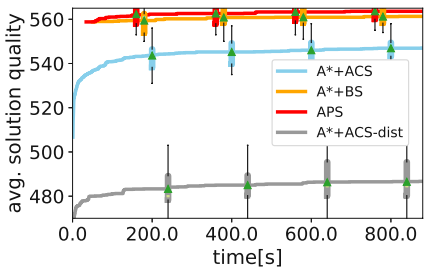
(d) Benchmark set Rat: instance $|\Sigma| = 4$, $m = 100, n = 600$.



(e) Benchmark set Random: instance $|\Sigma| = 20$, $m = 10, n = 600$.

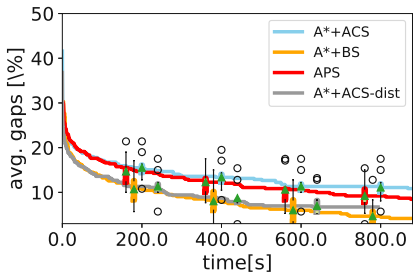


(f) Benchmark set ES: 50 instances with $|\Sigma| = 2, m = 100, n = 1000$.

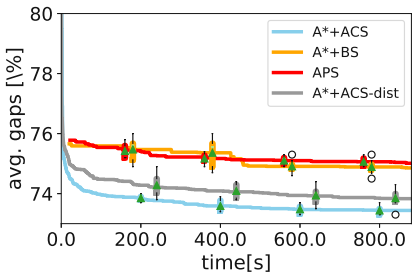


(g) Benchmark set BB: 10 instances with $|\Sigma| = 2, m = 100, n = 1000$.

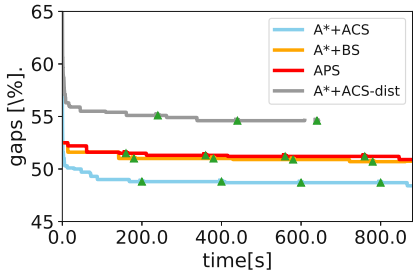
Figure 3.3: Comparison of the algorithms' anytime behavior concerning solution quality.



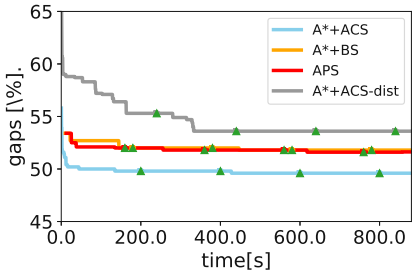
(a) Benchmark set BL: 10 instances with $|\Sigma| = 4, m = 10, n = 100$.



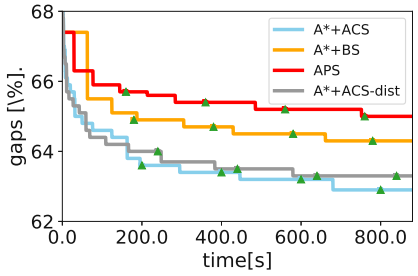
(b) Benchmark set BL: 10 instances with $|\Sigma| = 12, m = 50, n = 1000$.



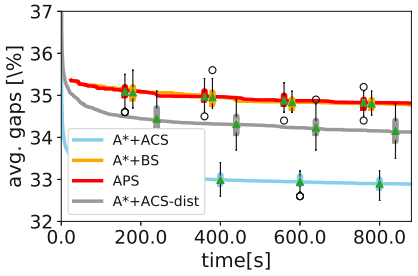
(c) Benchmark set Virus: instance $|\Sigma| = 4$, $m = 40, n = 600$.



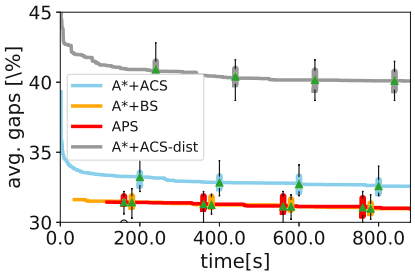
(d) Benchmark set Rat: instance $|\Sigma| = 4$, $m = 100, n = 600$.



(e) Benchmark set Random: instance $|\Sigma| = 20$, $m = 10, n = 600$.



(f) Benchmark set ES: 50 instances with $|\Sigma| = 2, m = 100, n = 1000$.



(g) Benchmark set BB: 10 instances with $|\Sigma| = 2, m = 100, n = 1000$.

Figure 3.4: Comparison of the algorithms' anytime behavior concerning gaps.

the evolution of the gap since the search is mainly focused on improving solution quality and less on improving the upper bound.

3.7 Conclusions

In this chapter we presented heuristic and exact approaches to solve the LCS problem. Concerning heuristic search, generalized beam search framework has been proposed. A novel expected length calculation heuristic EX was developed for random strings. Based on the beam search framework and heuristic EX, we were able to deliver a new state-of-the-art BS-configuration that outperforms the other heuristic approaches from the literature. Concerning the exact solving, A* search was proposed. This A* search makes use of the combination of the efficient upper bound calculation for the length of the LCS and is able to solve instances of up to $n = 100$ and $|\Sigma| \geq 12$ to proven optimality (106 instances from the literature are solved to optimality), most of them in a fraction of a second. For larger or more complex instances, however, the exact A* search soon either runs out of memory or requires substantially more time. Therefore, we combined A* search with either BS or ACS by interleaving traditional A* iterations with BS runs of small width or single iterations of ACS, respectively. Note that we did this combination in a way to avoid redundant expansions of the same nodes, i.e., the methods act on a shared list of open nodes. These anytime algorithms, denoted by A*+ACS and A*+BS either run until optimality is proven or they are terminated prematurely, in which case a solution of promising quality is returned in combination with an upper bound. To the best of our knowledge, we report proven optimality gaps for larger LCS instances for the first time ever in the literature. Our two anytime algorithms were compared to the well known *Anytime Pack Search* (APS) and a variant of A*+ACS employing the `dist` heuristic as guidance. All the parameters of the algorithms were tuned w.r.t. both, solution quality and small gaps by using `irace`. Our computational study showed that A*+ACS performs in most cases significantly better than the other algorithms concerning solution quality. New best solutions were found by A*+ACS for 82 different LCS instance groups from the literature ($\approx 70\%$ of all instance groups from the literature), and for the remaining groups, the so far best known results were matched by A*+ACS in most cases. Also concerning optimality gaps, A*+ACS outperforms the other approaches in most cases or is on par with them. Last but not least, A*+ACS usually provides a better anytime behavior in the sense that it earlier produces better results, and more frequently improves on them over time. Responsible for the success of A*+ACS is the careful selection and combination of strategies and components that proved already successful or promising in earlier works such as pure heuristic beam searches. The most important aspect is that we use, on the one hand, the upper bound UB for steering the classical A* search iterations and, on the other hand, the separate heuristic function EX for guiding the ACS iterations. While UB is required to obtain upper bounds and finally prove optimality, EX approximates the expected LCS length for unrelated random strings and is, for most of the considered benchmark instances, very well suited to lead ACS to promising heuristic solutions. The benefits of EX diminish,

however, when instances with strongly related strings are considered, as for example in benchmark set BB. There, EX tends to become a disadvantage and a classical upper bound UB for the LCS problem becomes advantageous since it obviously becomes much tighter than when the input strings are similar.

CHAPTER 4

The Longest Common Palindromic Subsequence Problem

In this chapter we address the longest common palindromic subsequence problem which finds applications in bioinformatics and providing a specific measure of similarity between molecular structures. In the course of this work, on the one hand, we have developed two heuristic algorithms: (*i*) a greedy heuristic and (*ii*) a beam search heuristic guided by a novel efficient heuristic guidance. On the other hand, the exact approaches were developed: (*i*) an A* search to tackle the small to middle-sized instances, an (*ii*) two anytime algorithms to tackle the large-sized problem instances.

In the course of this work, we have published two scientific papers:

- The conference paper is published in the Proceedings of *the 12th Learning and Intelligent Optimization* (LION 12) conference [54]. In this work we have presented a greedy heuristic, a BS, an A* search, and anytime algorithm, called A*+BS to solve the LCPS problem. A*+BS is hybrid of A* search and BS. Moreover, the classical LCPS problem for two input string is studied in this work. Our computational studies prove that our A* search is the new state-of-the-art algorithm for this problem.

- The extended version of the above conference paper has been published in *Computers & Operations Research* journal (IF=3.002) [57]. In this paper we extended our studies for the LCPS problem published at the conference by (*i*) providing a new heuristic guidance that approximates the expected length of an LCPS, and (*ii*) developing a hybrid A*+ACS for the LCPS which outperformed anytime

performance of the aforementioned A*+BS hybrid w.r.t. the solution as well as the final gap quality.

4.1 Introduction

Palindromic subsequences are especially interesting in the biological context. In many genetic instructions, such as for example DNA sequences, palindromic motifs are found. In the context of the research project on genome sequencing, it was discovered that many of the bases on the Y-chromosome are arranged as palindromes [117]. A palindrome structure allows the Y-chromosome to repair itself by bending over at the middle if one side is damaged. Moreover, it is believed that palindromes are also frequently found in proteins [70], but their role in the protein function is less understood. Biologists believe that identifying palindromic subsequences of DNA sequences may help to understand genomic instability [36, 159]. Palindromic subsequences seem to be important for the regulation, for example, of gene activity, because they are often found close to promoters, introns and untranslated regions.

Recall that an important way for the comparison of two or more strings is to find long *common subsequences*. In the course of this work, we are interested in a *common palindromic subsequence* of a set of (arbitrary) m strings $S = \{s_1, \ldots, s_m\}$ of maximum length. For biologists, it is not only of interest to identify the palindromic subsequences of an individual DNA string, for example, but it is also important to find longest common palindromic subsequences of multiple input strings in order to identify relationships among them.

4.1.1 Related Work

The longest common palindromic subsequence (LCPS) problem has so far only been studied for the case of $m = 2$ input strings (2–LCPS). Chowdhury et al. [38] propose two different algorithms: a conventional dynamic programming with time and space complexity $O(n^4)$, and a sparse dynamic programming algorithm with time complexity $O(R^2 \log^2 n \log\log n + n)$ and space complexity $O(R^2)$, where R is the number of matching position pairs between the two input strings. Furthermore, Hasan et al. [80] solved the 2–LCPS by making use of a so-called palindromic subsequence automaton (PSA). This algorithm has a time complexity of $O(n + R_1|\Sigma| + R_2|\Sigma| + n + R_1R_2|\Sigma|)$, where R_1 and R_2 denote the numbers of states of the two automata constructed for the two input strings and are bounded by $O(n^2)$. Finally, Inenaga and Hyyrö [89] present an algorithm that runs in $O(\sigma R^2 + n)$ time and uses $O(R^2 + n)$ space, where σ denotes the number of distinct characters occurring in both of the input strings.

By reducing the general LCS problem to the LCPS problem in polynomial time, it can be shown that the LCPS problem with an arbitrary number of input strings is $\mathcal{NP}$-hard. To the best of our knowledge, no algorithm has been published yet for solving this general m–LCPS problem, which is henceforth simply called LCPS problem.

In this paragraph we point out the main differences between methodologies we used to solve the LCS problem from one side, and the LCPS problem from another side. Although we employ, from a conceptual point of view, the same hybrid search strategies as in Chapter 3 to also solve the LCPS problem, we want to emphasize the significant differences between the application of the algorithmic concepts to the LCS problem presented in the last chapter and their application to the LCPS problem described in this chapter. Note, for example, that the best exact algorithms for the LCS problem for two input strings ($m = 2$) require $O(n^2)$ time, while the best exact algorithm for the LCPS problem requires $O(n^4)$ time. This already hints that both problems are structurally quite different from each other. These differences lead to the following differences in the adaptation of the algorithmic concepts to both problems:

- The search spaces of the two problems (in terms of the definition of the A* nodes) differ. This is due to the fact that in the LCS problem solutions are generated from left to right, while in the LCPS problem a solution construction starts from the left and from the right at the same time.
- The upper bound UB_1 utilized for the two problems is different.
- The expected length calculation heuristics (EX) for guiding the tree search techniques differ, even though similar ideas are used for their derivation.
- Last but not least, the implementations differ in additional details. For example, to solve the LCS problem, we make use of an efficient way of filtering the dominated nodes in BS and ACS iterations (i.e., *the restricted filtering*) and as well as pruning of the suboptimal nodes. In terms of time of development of the LCPS project, this came first and therefore these two aspects have not been considered in this research.

The rest of the chapter is organized as follow. Section 4.2 a Greedy heuristic to quickly derive feasible LCPS solutions is given. In Section 4.3 a general search framework and an A* framework to solve the LCPS problem are described. In Section 4.4 a novel expected length calculation heuristic is derived. Section 4.5 provides two anytime algorithms to tackle large-sized problem instances. Section 4.6 provides a detailed experimental comparison between described methods including also studies for the polynomial 2–LCPS problem. Finally, Section 4.7 sketches some directions for promising future work.

4.2 A Greedy Heuristic for the LCPS Problem

Inspired by the well-known Best–Next heuristic for the LCS problem [65], we present in the following a constructive greedy heuristic for the LCPS problem. Henceforth, a string s is called a *valid partial solution* concerning input strings $S = \{s_1, \ldots, s_m\}$, if $s \cdot s^{\text{rev}}$ or $s \cdot s[1, |s| - 1]^{\text{rev}}$ is a common palindromic subsequence of the strings in S. The greedy heuristic starts with an empty partial solution $s = \varepsilon$ and extends, at each construction

step, the current partial solution by appending exactly one letter (if possible). During the whole process, the algorithm makes use of pointers $p_i^{\mathrm{L}} \leq p_i^{\mathrm{R}}$ that indicate for each input string s_i, $i = 1, \ldots, m$, the still *relevant substring* $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]$ from which the letter for extending s can be chosen. The choice of a letter with respect to a greedy criterion is explained below. At the start of the heuristic, i.e., when $s = \varepsilon$, the pointers are initialized to $p_i^{\mathrm{L}} := 1$ and $p_i^{\mathrm{R}} := |s_i|$, referring to the first, respectively, last letter of each string s_i, $i = 1, \ldots, m$. In other words, at each iteration the set of relevant substrings denoted by $S[\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}}] = \{s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}] \mid i = 1, \ldots, m\}$ forms an LCPS subproblem, and the current partial solution s is ultimately extended by appending the solution to this subproblem.

One of the questions that remain is how to determine the subset of letters from Σ that can be used to extend a current partial solution s. For this purpose, let $c_a := \min_{i=1,\ldots,m} |s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]|_a$ be the minimum number of occurrences of letter $a \in \Sigma$ in the relevant substrings with respect to s, and let $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})} := \{a \in \Sigma \mid c_a \geq 1\}$ be the set of letters appearing at least once in each relevant substring. In principle, any letter from $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ might be used to extend s. However, there might be dominated letters in this set. In order to introduce the domination relation between two letters, we use the first and last positions at which each letter $a \in \Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ appears in each relevant substring $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]$:

$$q_{i,a}^{\mathrm{L}} := \min \{j = p_i^{\mathrm{L}}, \ldots, p_i^{\mathrm{R}} \mid s_i[j] = a\}$$
$$q_{i,a}^{\mathrm{R}} := \max \{j = p_i^{\mathrm{L}}, \ldots, p_i^{\mathrm{R}} \mid s_i[j] = a\}$$

A letter $a \in \Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ is called *dominated* if there exists a letter $b \in \Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$, $b \neq a$, such that $q_{i,b}^{\mathrm{L}} < q_{i,a}^{\mathrm{L}} \wedge q_{i,b}^{\mathrm{R}} > q_{i,a}^{\mathrm{R}}$ for $i = 1, \ldots, m$. Clearly, it is better to delay the consideration of dominated letters and select a non-dominated letter for the extension of s. Furthermore, letters $a \in \Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ with $c_a = 1$, called *singletons*, should only be considered when no other letters remain in $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$, since only one such letter can be chosen as single middle letter in the final solution. Accordingly, let the set of all non-dominated non-singleton letters from $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ with respect to s be denoted by $\Sigma^{\mathrm{nd}}_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$. Given a partial solution s, the selection of the letter to be appended to s and the adaption of the pointers work as follows:

1. If $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ is empty, the algorithm terminates with $s \cdot s^{\mathrm{rev}}$ as resulting common palindromic subsequence, since no further extension is possible.

2. Otherwise, if $\Sigma^{\mathrm{nd}}_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ is empty, only singletons remain in $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$. The algorithm terminates with the common palindromic subsequence $s \cdot a \cdot s^{\mathrm{rev}}$, where a is the first singleton from $\Sigma_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ in alphabetic order.

3. Otherwise, select a letter $a \in \Sigma^{\mathrm{nd}}_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ that minimizes the *greedy function* $g(a, \mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})$, which will be discussed in Section 4.2.1. Ties are broken randomly. Extend the

current partial solution s and adapt the pointers as follows:

$$s := s \cdot a \tag{4.1}$$

$$p_i^{\mathrm{L}} := q_{i,a}^{\mathrm{L}} + 1 \qquad i = 1, \ldots, m \tag{4.2}$$

$$p_i^{\mathrm{R}} := q_{i,a}^{\mathrm{R}} - 1 \qquad i = 1, \ldots, m \tag{4.3}$$

4.2.1 Greedy Function

The greedy function that is used to evaluate any possible extension $a \in \Sigma^{\mathrm{nd}}_{(\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}})}$ for a given partial solution extends the one used in [18] in a straight-forward manner. It calculates the sum of those fractions of the relevant substrings $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]$ that will be discarded from further consideration when appending character a as next letter to the partial solution:

$$g(a, \mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}}) := \sum_{i=1}^{m} \frac{q_{i,a}^{\mathrm{L}} - p_i^{\mathrm{L}} + p_i^{\mathrm{R}} - q_{i,a}^{\mathrm{R}}}{p_i^{\mathrm{R}} - p_i^{\mathrm{L}} + 1}. \tag{4.4}$$

The major advantage of this function is its simplicity, as it can be calculated in time $O(m)$. This function also has some weaknesses: (*i*) it does not take into account that, when choosing a specific letter, as a result, more or fewer letters might be excluded from further consideration, even in cases in which the chosen letter has a good (low) greedy function value; (*ii*) it does not take into account that of all singletons at most one can finally be selected. Instead of improving the above greedy function along these lines, and thereby increasing its time complexity, we consider it more promising—especially with the type of algorithm in mind that will be presented in the next section—to develop upper bound functions for estimating the length of an LCPS. As we will see, these bounds can also be used as alternative greedy functions to evaluate possible extensions of partial solutions.

4.3 A* Search for the LCPS Problem

Problem-specific aspects in order to realize an A* search for a specific problem are primarily to define (*i*) the state graph including the root and goal nodes and (*ii*) the heuristic function h. This will be done in the following for the LCPS problem.

4.3.1 State Graph

Let $\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}} \in \mathbb{N}^m$ be m-dimensional integer valued vectors such that $1 \leq p_i^{\mathrm{L}} \leq p_i^{\mathrm{R}} \leq |s_i|$ for all $i = 1, \ldots, m$. Given such vectors $\mathbf{p}^{\mathrm{L}}$ and $\mathbf{p}^{\mathrm{R}}$, set $S[\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}}] := \{s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}] \mid i = 1, \ldots, m\}$ consists of a continuous substring $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]$ for each input string s_i, $i = 1, \ldots, m$. Hereby, $\mathbf{p}^{\mathrm{L}}$ is called the *left position vector* and $\mathbf{p}^{\mathrm{R}}$ is called the *right position vector*. Moreover, $S[\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}}]$ is henceforth called a *subproblem* of the original LCPS problem. For the definition of the state graph of the A* approach only those subproblems that are induced by valid partial solutions are considered. More specifically, a valid partial

solution s *induces* a subproblem $S[\mathbf{p}^{\mathrm{L}}, \mathbf{p}^{\mathrm{R}}]$ if and only if the following two conditions hold:

1. $s_i[1, p_i^{\mathrm{L}} - 1]$ is a minimal string among all strings $s_i[1, x]$ with $1 \leq x \leq p_i^{\mathrm{L}} - 1$ containing s as a subsequence for all $i = 1, \ldots, m$.
2. $s_i[p_i^{\mathrm{R}} + 1, |s_i|]$ is a minimal string among all strings $s_i[x, |s_i|]$ with $p_i^{\mathrm{R}} + 1 \leq x \leq |s_i|$ containing s^{rev} as a subsequence for all $i = 1, \ldots, m$.

In this context, note that the same subproblem may be induced by more than one valid partial solution. As an example consider $S = \{\texttt{abccdccba}, \texttt{baccdccab}\}$, and partial solutions $s = \texttt{ac}$ and $s' = \texttt{bc}$. It holds that $\mathbf{p}^{\mathrm{L}} = (4, 4)$ and $\mathbf{p}^{\mathrm{R}} = (6, 6)$ in both cases, and thus, both partial solutions will be represented by a common node in the state graph. Here, s and s' have the same length, but this need not be the case in general.

The state graph of our A* search is a directed *acyclic graph* $G = (V, A)$ in which each node corresponds to a unique state $v = (\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}) \in V$ and thus also to a unique LCPS subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$. For the reason outlined above, a node v may potentially be induced by multiple valid partial solutions. The special root node $r = ((1, \ldots, 1), (|s_1|, \ldots, |s_m|)) \in V$ represents the original LCPS problem, which can be said to be induced by the empty partial solution ε.

An arc $(v, v') \in A$ leading from a node $v \in V$ to a node $v' \in V$ corresponds to the extension of a partial solution s inducing v to a partial solution s' inducing v' by one specific letter $a \in \Sigma$, i.e., $s' = s \cdot a$. Any path from the root node r to any node in V represents a valid partial solution, which is directly given by the sequence of letters associated with the arcs.

A letter $a \in \Sigma$ is called *feasible* for a node $v \in V$ if it appears at least twice in each substring of subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$. Let $a, b \in \Sigma$ be two feasible letters with respect to node v. Moreover, let v' be the node corresponding to the subproblem induced by appending a to some feasible partial solution inducing v, and let v'' be the node induced by appending b to some feasible partial solution inducing v. We say that letter b is *dominated* by letter a (or a is dominating b) if and only if $p_i^{\mathrm{L},v'} \leq p_i^{\mathrm{L},v''} \wedge p_i^{\mathrm{R},v''} \leq p_i^{\mathrm{R},v'}$ $\forall i = 1, \ldots, m$. Obviously, arcs that correspond to appending dominated letters do not need to be considered. Therefore, an arc (v, v') exists in our state graph G if and only if the letter that is used for obtaining v' from v is (i) feasible and (ii) non-dominated. This subset of letters with respect to a node v is henceforth denoted by $\Sigma_v^{\mathrm{nd}} \subseteq \Sigma$. In other words, each node $v \in V$ has an outgoing arc $(v, v') \in A$ for each letter $a \in \Sigma_v^{\mathrm{nd}}$. Letters that do not appear in at least one of the substrings of $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ are called *infeasible letters*. Moreover, letters that appear at least once in each substring of $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$, and exactly once for at least one $i \in \{1, \ldots, m\}$, are called *singleton letters*. Nodes $v \in V$ without any outgoing arcs are *goal nodes*.

In order to solve the LCPS problem, a longest path in the state graph leading from the root node r to some goal node has to be identified. In respect to the number of arcs, such

a longest path represents a longest partial solution s. The corresponding palindromic subsequence is $s \cdot s^{rev}$, but note that it may still be possible to insert, as the last step, a singleton letter $a \in \Sigma$ in the middle, yielding the longer palindrome $s \cdot a \cdot s^{rev}$. Thus, we have to either find a longest partial solution that allows such an extension by a singleton letter or prove that no other equally long path with a singleton-extension exists, in which case $s \cdot s^{rev}$ is returned as LCPS. To account for this possible insertion of a final middle element and to represent by the path lengths in our state graph the actual lengths of resulting palindromes, each arc $(u, v) \in A$ gets assigned length

$$w(u,v) = \begin{cases} 3 & \text{if } v \text{ is a goal node and } S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}] \text{ contains a singleton letter} \\ 2 & \text{else.} \end{cases} \tag{4.5}$$

When referring to the length of a path from now on, we therefore mean the sum of these lengths of all the arcs forming the path.

During our search—as described in detail below—nodes $v \in V$ store as additional information the length l^v of the currently longest path from r to v.[1]

4.3.2 Upper Bounds for the Length of an LCPS

Remember that A* depends on a heuristic function $h(v)$ for estimating the still possibly length of a longest path from node $v \in V$ to some goal node, i.e., for the length of the LCPS of the subproblem represented by v. This function is generally implemented—in the context of maximization problems such as the LCPS problem—in terms of an upper bound in order to ensure *admissibility*, i.e., the completeness of the A* search. In the following a combined upper bound UB is presented, composed of two individual upper bounds UB_1 and UB_2, that is $\mathrm{UB}(v) = \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}$ for all $v \in V$. Hereby, UB_1 is based on the UB_1 bound from [54], while UB_2 is newly developed. In contrast to [54], an appropriate preprocessing action for speeding up the calculation of UB_1 in a significant way is used.

Let us denote by c_a the minimum number of occurrences of a letter $a \in \Sigma$ in the subproblem represented by a current node $v \in V$, that is, $c_a := \min_{i=1,\ldots,m} |s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]|_a$. A simple upper bound is given by

$$\mathrm{UB}_1(v) = \mathrm{UB}_1(S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]) := \left(2 \cdot \sum_{a \in \Sigma} \left\lfloor \frac{c_a}{2} \right\rfloor \right) + \mathbb{1}_{\exists a \in \Sigma | c_a \bmod 2 = 1}. \tag{4.6}$$

The last term considers the fact that at most one singleton letter can finally be added at the end of a solution construction, with $\mathbb{1}$ denoting the unit step function that yields one if and only if the condition in the subscript is fulfilled, i.e., there exists a letter in Σ

[1] In this context we emphasize that it is not necessary to store actual partial solutions s with the nodes. For any node in the graph, the longest path to it and the respective partial solution can finally be efficiently derived. This is done in a backward manner by iteratively identifying predecessors in which the l^v-values always decrease by two when l^v is even or by three if l^v is odd.

with an odd value of c_a. In a naive fashion, $\mathrm{UB}_1(v), v \in V$ is calculated in $O(mn)$ time. However, the repeated calculation of UB_1 can be sped up by a preprocessing step that determines the number of occurrences of each letter in all postfixes of all input strings in advance. More precisely, the values

$$\omega_{i,j,a} = \left|s_i[j, |s_i|]\right|_a \qquad \forall i = 1, \ldots, m,\ j = 1, \ldots, n+1,\ a \in \Sigma. \tag{4.7}$$

are predetermined. During the actual A* search, c_a can then be efficiently determined for a current state v as $c_a := \min_{i=1,\ldots,m}(\omega_{i,p_i^{\mathrm{L},v},a} - \omega_{i,p_i^{\mathrm{R},v}+1,a})$, and UB_1 can be calculated in $O(m|\Sigma|)$ time.

Although the second upper bound UB_2 proposed in [54] is comparably tight, it was judged to be impractical due to being computationally too expensive. Therefore, in this section an alternative UB_2 bound is proposed, which is based on the standard DP procedure for calculating the LCS of two input strings; see, for example, [174]. The so-called complete upper bound $\mathrm{UB}_2^{\mathrm{comp}}(v)$ for a node $v \in V$—that is, for the subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ of still relevant substrings can be computed similarly as in the case of the LCS problem and Equation (3.1), as follows:

$$\mathrm{UB}_2^{\mathrm{comp}}(v) := \min_{1 \le i < j \le m} \left(M_{ij}[p_i^{\mathrm{L},v}, p_j^{\mathrm{L},v}] - M_{ij}[p_i^{\mathrm{R},v}+1, p_j^{\mathrm{R},v}+1] \right). \tag{4.8}$$

$\mathrm{UB}_2^{\mathrm{comp}}$ is, for the following reasons, indeed an upper bound for the length of an LCPS of $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$. Let $S' = \{s'_1, \ldots, s'_m\}$ and $S'' = \{s''_1, \ldots, s''_m\}$ be two sets, each one containing m strings. Moreover, let $C = \{s'_1 \cdot s''_1, \ldots, s'_m \cdot s''_m\}$. Obviously it holds that $\mathrm{LCS}(S') + \mathrm{LCS}(S'') \le \mathrm{LCS}(C)$, where $\mathrm{LCS}(S')$ denotes the length of the LCS of the strings in S', etc. This immediately implies that $\mathrm{LCS}(S') \le \mathrm{LCS}(C) - \mathrm{LCS}(S'')$. Moreover, it holds that the length of the LCPS for S' cannot exceed $\mathrm{LCS}(S')$. However, since the number of strings in S can be large, we replace the complete upper bound $\mathrm{UB}_2^{\mathrm{comp}}$ with the following final definition of UB_2, which is a faster approximation of $\mathrm{UB}_2^{\mathrm{comp}}$:

$$\mathrm{UB}_2(v) := \min_{i=1,\ldots,m-1} \left(M_{i,i+1}[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}] - M_{i,i+1}[p_i^{\mathrm{R},v}+1, p_{i+1}^{\mathrm{R},v}+1] \right). \tag{4.9}$$

Note that for the efficient evaluation of (4.9), the matrices $M_{i,i+1}$, $i = 1, \ldots, m-1$ have to be determined in a preprocessing step, which can be achieved in $O(mn^2)$ time. The calculation of $\mathrm{UB}_2(v)$ for any node $v \in V$ then runs in $O(m)$ time.

4.3.3 Details of the A* Search for LCPS Problem

A* maintains two sets of nodes: N stores all so far reached nodes, while Q, the set of *open nodes*, is the subset of nodes in N that have not yet been *expanded*, i.e., whose outgoing arcs and respective successors have not yet been considered. Node set N is realized by means of a hash map in order to be able to efficiently find an already existing node for a state $(\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v})$, or to determine that no respective node exists yet. The set of open nodes Q is realized by means of a heap in which the nodes are partially sorted

Algorithm 19 A* Search for the LCPS Problem

Input: an instance (S, Σ)
Output: an optimal LCPS solution
Create root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$ with $l^r = 0$
Add r to the initially empty node set N and the set of open nodes Q
loop
  Pop a node v with largest priority $f(v)$ from open nodes Q
  Determine Σ_v^{nd} from $\mathbf{p}^{\mathrm{L},v}$ and $p^{\mathrm{R},v}$
  if $\Sigma_v^{\mathrm{nd}} = \emptyset$ **then** // goal node reached
    $s \leftarrow$ partial solution corresponding to a longest path from r to v
    if $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ contains a singleton letter a **then**
      return palindrome $s \cdot a \cdot s^{\mathrm{rev}}$
    else
      return palindrome $s \cdot s^{\mathrm{rev}}$
    end if
  else
    for $a \in \Sigma_v^{\mathrm{nd}}$ **do** // consider successors
      Compute node v' that results from appending a at node v
      if $S[\mathbf{p}^{\mathrm{L},v'}, \mathbf{p}^{\mathrm{R},v'}]$ contains only singleton letters **then**
        $w(v, v') \leftarrow 3$
      else
        $w(v, v') \leftarrow 2$
      end if
      if $v' \notin N$ **then**
        Add new node v' with $l^{v'} = l^v + w(v, v')$ to N and Q
      else if $l^v + w(v, v') > l^{v'}$ **then** // a better path to v'
        $l^{v'} \leftarrow l^v + w(v, v')$
        Update entry for v' in Q with new priority value $f(v') = l^{v'} + \mathrm{UB}(v')$
      end if
    end for
  end if
end loop

according to the priority function $f(v) = g(v) + h(v) := l^v + \mathrm{UB}(S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}])$. In case of ties, nodes with larger l^v are preferred. In case of further ties, they are broken by considering the distance between the positions $\mathbf{p}^{\mathrm{L},v}$ and $\mathbf{p}^{\mathrm{R},v}$ as measured by means of the k-norm, for some $k > 0$ being a parameter of the algorithm.

The pseudo-code of our A* search is shown in Algorithm 19. It starts with the root node as a unique node in N and Q. At each step, the first node v from Q—that is, the highest priority node—is chosen and removed from Q. If this node is non-extensible it is a goal node. In this case, the algorithm derives the actual partial solution corresponding to the longest path to v and returns with the resulting palindrome. Note that if a singleton

letter remains in $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$, it is added as middle letter. Since our priority function is *admissible*, cf. [79], it guarantees that an optimal solution has been reached. Otherwise, node v is extended by considering each possible extension $a \in \Sigma_v^{\mathrm{nd}}$. Corresponding arc costs $w(v, v')$ are normally two to account for letter a being added twice in the final palindrome, but three in case only singleton letters remain in $S[\mathbf{p}^{\mathrm{L},v'}, \mathbf{p}^{\mathrm{R},v'}]$ to additionally account for a respective final middle letter. For each obtained new state it is checked if a respective node exists already in N. If this is not the case, a corresponding new node is added to N and Q. Otherwise, the existing node's length-value $l^{v'}$ is updated in case the new path via v represents a new longest partial solution.

Finally, note that all proposed upper bound functions presented in Section 4.3.2 have the property of being *monotonic* (also called *consistent*), because the upper bound values of child nodes are always at most as high as the upper bound values of their parents. Due to monotonicity, no re-expansions of already expanded nodes will be necessary [79].

Deriving the left and the right position vectors and thus the state for each successor v' of a node v is the computationally most expensive step during the process of *expanding a node* v. More specifically, for each string s_i, $i = 1, \ldots, m$, $p_i^{\mathrm{L},v'}$ (respectively $p_i^{\mathrm{R},v'}$) of a child node v' of v, given by expanding a valid partial solution represented by v by means of a letter $a \in \Sigma$, is determined as the position of the first occurrence of a in string $s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]$ (respectively, the last occurrence of a in string $s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]$). Finding these positions can be done efficiently by establishing during preprocessing a successor (predecessor) data structure as follows. The successor structure contains a value $\mathrm{Succ}[i, j, a]$ for each $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $a \in \Sigma$ corresponding to the minimal position $p > j$ such that $s_i[p] = a$. If there is no such position, the special value $n + 1$ is used. The predecessor structure stores a value $\mathrm{Pred}[i, j, a]$ for all $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $a \in \Sigma$ corresponding to the maximal position $p < j$ such that $s_i[p] = a$. In case of no such position, zero is used here. Both structures can be built in $O(m \cdot n \cdot |\Sigma|)$ time, and by using it all successors of a node can be derived in $O(m \cdot |\Sigma|)$ time.

Remember that in Section 4.3.1 a dominance relation between two letters a and b for extending a node v was introduced. Dominated letters are clearly sub-optimal choices and therefore their further consideration is avoided. This pruning according to dominance may be generalized: When a new state $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ is obtained, it can be checked if there is some other already considered node $v' \in N$ for which $p_i^{\mathrm{L},v'} \leq p_i^{\mathrm{L},v} \wedge p_i^{\mathrm{R},v} \leq p_i^{\mathrm{R},v'}$ $\forall i = 1, \ldots, m$ holds and for which $l^{v'} \geq l^v$. Such a node v' would dominate v in the sense that v cannot lead to any better solution, and consequently, node v and the arc leading to it can be omitted from further consideration. Unfortunately, this generalized dominance check requires $O(|N| \cdot m)$ time. In practical experiments with our A* search, it turned out that the introduced overhead is substantial and can be dramatic especially for longer runs when $|N|$ becomes large. Usually the gained reductions in the number of avoided nodes cannot outweigh this disadvantage. Therefore, we stay here with the simple dominance checks among the successors of a node.

In the rest of this chapter, we first propose novel heuristic guidance for the LCPS problem and two variants of search algorithms that build upon the presented A* search are

investigated. In particular, our goal is to find an algorithm with an improved anytime behaviour, i.e., which provides a first heuristic solution soon and continuously improving upon it over time.

4.4 Approximating the Expected Length of an LCPS for Random Strings

In prior work on beam search algorithms for the LCS problem, Mousavi and Tabataba [135] noticed that the LCS problem instances generally used in the related literature have properties close to those of random instances. That is, the probability for a letter $a \in \Sigma$ to appear at the position i of any of the input strings is (nearly) equal for all letters from Σ. Based on this observation they derived a heuristic function for guiding their beam search approach, which led to a new state-of-the-art performance at that time. In other words, they discovered that their heuristic function guides beam search much better than the available upper bound functions. However, since their heuristic function is not a proper upper bound, it cannot be used to prove optimality. In the following, the heuristic function from [135] is first revisited in the context of the LCPS problem and then build upon it by deriving an approximation for the expected length of an LCPS for random strings. This function will later be used in combination with the previous upper bound to be able to find good heuristic solutions quickly (due to using the heuristic function) but to possibly prove optimality as well.

Mousavi and Tabataba came up with the recursion which calculates the probability that a specific string s of length k is a subsequence of a string t of length q, where t is generated uniformly at random, see Equation (3.2). All probabilities $\mathrm{P}(k, q)$ for $k, p = 0, \ldots, n$ can be calculated and stored in $O(n^2)$ time during preprocessing. Let us now consider a node $v \in V$ from our state graph. Given P, one can calculate the probability that the remaining subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ contains a specific palindrome of length k by

$$\Pr(k, S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]) = \prod_{i=1}^{m} \mathrm{P}(k, |s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]|) = \prod_{i=1}^{m} \mathrm{P}(k, p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1). \tag{4.10}$$

In fact, Mousavi and Tabataba directly used these probabilities as heuristic function $h(v)$ to rank in their beam search all successor nodes at a current level for selecting the most promising ones and filtering out the rest. Higher values of $h(v)$ are preferred in this ranking. For parameter k, they used at each level of the beam search the same value, which they determined from the set of all nodes to be compared (V_{ext}) simply by

$$k := \min_{v \in V_{\mathrm{ext}},\ i=1,\ldots,m} \left(\left\lfloor \frac{p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1}{|\Sigma|} \right\rfloor \right).$$

While this approach can be meaningful in the context of a standard beam search, it cannot be easily adopted in a more general search like A*, where at each iteration a node has to be evaluated efficiently in relation to the potentially huge set of all open

nodes with different distances to the root. Most importantly, there will not be a single meaningful value for k, and it would not make sense to repeatedly re-evaluate all open nodes for changing values of k.

Instead, we strive to approximate the real expected length of an LCPS for a set of strings $S = \{s_1, \ldots, s_m\}$ under the assumption that the input strings are mutually independent uniform random strings. Note that this approximation will be used to evaluate a subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$ represented by a node $v \in V$ in an alternative way.

Let X be the random variable corresponding to the length of an LCPS for a set S of randomly generated input strings. By following the same derivation of the expected length calculation heuristic in the context of the LCS problem, the expected length of an LCPS can be expressed as $\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} l \cdot \Pr[X = l]$ with $\Pr[X = l]$ denoting the probability that this length is l. Furthermore, let $Y_l \in \{0, 1\}$ be the random variable indicating if the strings from S have a common palindromic subsequence of length l, $l \geq 0$. Applying the same reasoning as in Section 3.3.2, it holds that

$$\Pr[X = l] = \mathbb{E}[Y_l] - \mathbb{E}[Y_{l+1}] \text{ for } l = 0, \ldots, l_{\max},$$

i.e., the probability that there exists a palindromic subsequence of size l but no longer one. It further implies that $\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} = \sum_{l=1}^{l_{\max}} \mathbb{E}[Y_l]$. In order to approximate $\mathbb{E}[Y_l]$, it can be first observed that—for an alphabet of size $|\Sigma|$—there are $|\Sigma|^{\lceil \frac{l}{2} \rceil}$ different palindromes of length l. This is because the first half and the possible middle element can be assigned any letters from Σ and the second half must be equal to the reverted first half. In the following let us make the simplifying assumption that for each palindrome of length l the event of appearing as common subsequence of S is independent of the events of the other palindromes. Clearly, this does not entirely hold in reality and we introduce an error. The probability that S has any common palindromic subsequence of length $l \in \mathbb{N}$ can then be approximately expressed as

$$\tilde{\mathbb{E}}[Y_l] = 1 - (1 - \Pr[s \prec S])^{|\Sigma|^{\lceil l/2 \rceil}} = 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^{\lceil l/2 \rceil}}, \tag{4.11}$$

i.e., the inverse probability of the case that none of the $|\Sigma|^{\lceil l/2 \rceil}$ palindromes of length l is a common subsequence of S. Ultimately, the approximate expected length of the LCPS can be expressed as

$$\tilde{\mathbb{E}}[X] = l_{\max} - \sum_{l=1}^{l_{\max}} \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^{\lceil l/2 \rceil}}. \tag{4.12}$$

To illustrate the error introduced by the assumed independence, the following special cases are considered.

- Let $S = \{s_1\}$ and $l = |s_1|$. At most one of the $|\Sigma|^{\lceil l/2 \rceil}$ different palindromes of length l can be a subsequence of s_1 since s_1 has to correspond to it. Our calculation yields

$$\tilde{\mathbb{E}}[Y_l] = 1 - (1 - 1/|\Sigma|^l)^{|\Sigma|^{\lceil l/2 \rceil}},$$

while the correct value corresponds to the probability that s_1 is palindromic, which is

$$\tilde{\mathbb{E}}[Y_l] = \frac{|\Sigma|^{\lceil l/2 \rceil}}{|\Sigma|^l} = 1/|\Sigma|^{\lfloor l/2 \rfloor}.$$

- Let $S = \{s_1\}$ with $|s_1| \geq 1$ and $l = 1$. It follows that $\tilde{\mathbb{E}}[Y_l] = 1 - (1-1)^{|\Sigma|} = 1$, which corresponds to the correct probability for $\mathrm{E}[Y_l]$.
- Let $S = \{s_1, \ldots, s_m\}$ with $l = |s_1| = \ldots = |s_m|$.

$$\tilde{\mathbb{E}}[Y_l] = 1 - \left(1 - \frac{1}{|\Sigma|^{lm}}\right)^{|\Sigma|^{\lceil l/2 \rceil}}$$

while

$$\mathbb{E}[Y_l] = \frac{1}{|\Sigma|^{l(m-1)+\lfloor l/2 \rfloor}} = \frac{1}{|\Sigma|^{\lfloor l \cdot (2m-1)/2 \rfloor}}.$$

Finally, recall that each node $v \in N$ of our state graph represents a subproblem $S[\mathbf{p}^{\mathrm{L},v}, \mathbf{p}^{\mathrm{R},v}]$, and we can determine corresponding approximate expected LCPS lengths according to (3.5) and the above described stable and efficient calculation method for these:

$$\mathrm{EX}(v) = \sum_{l=1}^{l^v_{\max}} 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^{\left\lceil \frac{l}{2} \right\rceil}}, \tag{4.13}$$

where $l^v_{\max} = \max_{i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1)$. The efficient calculation and the issue with numeral instabilities of Equation (4.13) are resolved in the same way as in Section 3.3.2. Note that $\mathrm{EX}(v)$, in contrast to the upper bound functions from Section 4.3.2, does not possess the property of being admissible in the context of A* search.

4.4.1 Beam Search for the LCPS Problem

For a pure heuristic way of solving the LCPS problem we apply a Beam Search technique, where its search is applied on the state graph constructed in Section 4.3.1.

The pseudo-code of our BS is an extension of Algorithm 14 with the following differences: (*i*) the LCS nodes are replaced by the LCPS nodes (*ii*) dominance relation is extended in the context of for LCPS nodes: given LCPS nodes $u, v \in V$, u *dominates* v iff $u \neq v$ and $p_i^{\mathrm{L},u} \leq p_i^{\mathrm{L},v} \wedge p_i^{\mathrm{R},u} \geq p_i^{\mathrm{R},v}$ for all $i = 1, \ldots, m$, (*iii*) different upper bounds and heuristics applied to gude the search, and (*iv*) the full filtering has been applied, that is, parameter k_{best} is set to $+\infty$.

4.5 Anytime Algorithms to Solve the LCPS Problem

Classical A* search is targeted towards finding a proven optimal solution in the least number of expanded nodes, but in general, it yields no meaningful or particularly promising heuristic solution before it terminates when the target node is selected for expansion.

When consulting the literature, several attempts can be found at improving the anytime performance of A* search as well as several attempts at using beam search related algorithms in an anytime fashion. A short overview of these approaches is provided in Section 2.3.

Before outlining our anytime approaches, Table 3.1 summarizes the main ideas of the anytime algorithms that are covered in our experimental evaluation in Section 4.6. In the next two sections, we give the descriptions of the two novel anytime algorithms that are specifically applied to the LCPS problem.

4.5.1 A*+BS Hybrid

We already presented the basic idea of this hybrid in Section 3.5.1 in the context of the LCS problem. The hybrid A*+BS algorithm embeds a standard beam search into the A* search framework such that, after performing a number of regular A* iterations, the search strategy repeatedly is switched to a BS starting from the node with the highest priority value from Q. Note that the new nodes discovered by the BS applications are also incorporated into set N and the priority queue Q, and expanded nodes are removed from Q. Therefore, the embedded BS might end without delivering any complete solution. Moreover, for all considered extensions of nodes, the same steps regarding the update of the nodes in N and Q are performed as in A*, cf. Algorithm 19. Finally, note that with beam width $\beta = 1$ the embedded BS corresponds to the technique called *simple diving*.

4.5.2 Tie Breaking

While executing preliminary experiments for A*, we realized that many ties occur when ordering the nodes in the priority queue Q with respect to their priorities in $\pi(v)$. To guide the search in better ways, we decided to use the length of a represented longest partial solution as a secondary decision criterion in such cases. This improved the performance significantly but still suffered from a significant number of ties. In order to also break these, it turned out to be beneficial to additionally consider the p-norm, which is for a node v defined as

$$||v||_p = \left(\sum_{i=1}^{m} \left| p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} \right|^p \right)^{1/p}. \tag{4.14}$$

Given two nodes $u \neq v$ with the same priority value and the same maximum length concerning the represented partial solutions, a node with a lower p-norm is finally preferred. The inspiration for making use of this norm is that the smallest still relevant

substrings potentially have a higher impact on the final length of complete solutions than the larger ones. However, considering only the shortest one of the still relevant substrings—that is, applying the min norm—could be highly misleading. Therefore, a p value from $(0, 1)$ appears meaningful. Following further preliminary experiments, we finally chose $p = 0.5$ for all experiments.

4.5.3 A*+ACS Hybrid

As indicated before, ACS is an iterative algorithm that, at each major iteration, expands nodes with the highest priority values at each level of the state graph [165]. In order to do so, the algorithm interprets the so far investigated parts of the state graph in a layered fashion, where level $j \geq 0$ contains any node $v \in N$ having depth j, i.e., can be reached from the root node r via j so far known arcs but not more. In our context of the LCPS, level j thus contains the nodes for which corresponding partial solutions with up to j letters are known. If a node is updated during the search process because a longer partial solution—represented by a longer path to this node—is found, the node will change to the respective higher level. In contrast to the classical A* search, ACS maintains an individual priority list Q_l of open nodes for each level $j = 0, \ldots, j_{\max}$, where $j_{\max}$ is an upper bound for the depth of nodes; in our implementation, $j_{\max} = \lfloor \mathrm{UB}(r)/2 \rfloor$ is used. Initially, Q_0 contains the root node and all other priority queues are empty. Each iteration of ACS considers all the levels $j = 0, \ldots, j_{\max}$ with non-empty queues Q_i in turn and expands from each β nodes (or less if Q_j becomes empty). ACS terminates with an optimal solution only when all priority lists become empty. However, ACS finds at least one complete solution at each major iteration, which favours our goal of producing heuristic solutions as soon as possible. An algorithm similar to the ACS was proposed in [99] but presented in a more general form.

The idea is to embed this ACS in our A* by interleaving classical A* iterations with ACS iterations. A pseudo-code of this A*+ACS is presented in Algorithms 20 and 21 and it has similarity to Algorithm 18, but since there are a lot of minor details on which they differ, we consider that it would be confusing to explain them with referring to the latterly mentioned pseudocode. So, we decide to present the pseudocode of A*+ACS for the LCPS problem with all detail for ease of reading.

The main algorithmic framework is that of A*. However, initially and after every batch of $\delta > 0$ iterations of A*, the algorithm executes one iteration of ACS. Algorithm 20 maintains in s_{best} the so far best found complete solution. Each A* iteration still expands a node v from the global open list Q having the largest priority value $f(v)$. In this way, the whole approach maintains the completeness property of the classical A* search and $\max_{v \in Q} f(v)$ provided by the top element of Q always is an upper bound for the optimum solution value. In contrast, the level-wise priority queues Q_j, $j = 0, \ldots, j_{\max}$, of ACS make use of the approximate expected value function EX, cf. (4.13), for prioritizing the nodes. As it is expected to lead the construction of heuristic solutions in substantially better ways. Note that changes in Q (removals and additions) must be reflected by corresponding changes in the priority queues Q_j, and vice versa. To do this efficiently,

Algorithm 20 A*+ACS for the LCPS Problem

Input: an instance (S, Σ)
Output: best found LCPS solution s_{best}
Parameters: ACS column width β, number of A* iterations inbetween ACS δ
Create root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$ with $l^r = 0$
Add r to the initially empty node set N and the global set of open nodes Q
Initialize per-level priority queues $Q_0 = \{r\}$ and $Q_i = \emptyset$, $j = 1, \ldots, j_{\max}$
optimal $\leftarrow$ **false**
$s_{\text{best}} \leftarrow \varepsilon$
loop // perform an ACS iteration of width β:
  for $j \leftarrow 0, \ldots, j_{\max}$ **do**
    $b \leftarrow 0$
    while $Q_j \neq \emptyset \wedge b < \beta$ **do** // select and expand next node at level j
      Pop a node v with the largest EX(v)-value from Q_j
      Remove v also from Q
      `ExpandNode`(v)
      if *optimal* $\vee$ *time or memory limit reached* **then**
        return so far best solution s_{best}
      end if
      $b \leftarrow b + 1$
    end while
  end for
  loop δ times // perform δ normal A* iterations:
    Pop a node v with largest priority $f(v)$ from Q
    Remove v also from $Q_{\lfloor l^v/2 \rfloor}$
    `ExpandNode`(v)
    if *optimal* $\vee$ *time or memory limit reached* **then**
      return so far best solution s_{best}
    end if
  end loop
end loop

we augment in our implementation the heap data structures for the priority queues by corresponding hash tables, which enable a direct lookup of the priority queue entries for given nodes. The actual expansion of a node, which is identical for the ACS as well as the A* iterations, is separately shown in Algorithm 21. It follows the principles already known from Algorithm 19. When a goal node is reached it is checked if it yields a new best solution and s_{best} is updated in this case. At its end, Algorithm 21 always checks if the length of the so-far best solution is larger than or equal to the current maximum f-value of Q, in which case the flag *optimal* is set to true and the main algorithm terminates with the proven optimal solution s_{best}. Moreover, A*+ACS also terminates when reaching a specified time or memory limit, in which case it returns the best complete solution found

Algorithm 21 `ExpandNode`(v)

Input: Node v to be expanded
Uses/updates: V, N, Q, $Q_1, \ldots, Q_{l_{\max}}$, s_{best}, *optimal*
Determine Σ_v^{nd} from $\mathbf{p}^{\text{L},v}$ and $\mathbf{p}^{\text{R},v}$
if $\Sigma_v^{\text{nd}} = \emptyset$ **then**
    if $|s_{\text{best}}| < f(v)$ **then** // goal node reached
        $s \leftarrow$ partial solution corresponding to a longest path from r to v
        if $S[\mathbf{p}^{\text{L},v}, \mathbf{p}^{\text{R},v}]$ contains a singleton letter a **then**
            $s_{\text{best}} \leftarrow s \cdot a \cdot s^{\text{rev}}$
        else
            $s_{\text{best}} \leftarrow s \cdot s^{\text{rev}}$
        end if
    end if
else
    for $a \in \Sigma_v^{\text{nd}}$ **do** // consider successors
        Compute node v' that results from appending a at node v
        if $S[\mathbf{p}^{\text{L},v'}, \mathbf{p}^{\text{R},v'}]$ contains only singleton letters **then**
            $w(v, v') \leftarrow 3$
        else
            $w(v, v') \leftarrow 2$
        end if
        if $v' \notin N$ **then**
            Calculate $\text{EX}(v')$ and $f(v')$
            Add new node v' with $l^{v'} = l^v + w(v, v')$ to N, Q, and $Q_{\lfloor l^{v'}/2 \rfloor}$
        else if $l^{v'} < l^v + w(v, v')$ **then** // a better path to v'
            Remove v' from $Q_{\lfloor l^{v'}/2 \rfloor}$
            $l^{v'} \leftarrow l^v + w(v, v')$
            Update entry for v' in Q with new $f(v')$
            Add v' in $Q_{\lfloor l^{v'}/2 \rfloor}$ with $E(v')$
        end if
    end for
end if
if $|s_{\text{best}}| \geq$ maximum f-value of nodes in Q **then**
    optimal $\leftarrow$ **true**
end if

up to this point.

In summary, the ACS iterations augment the classical A* iterations in order to find promising heuristic solutions soon and possibly improve them continuously over time. This counter-balances the pure best-first strategy of A*. The number of A* iterations δ between the executions of the ACS iterations as well as ACS's width parameter β control

the balance between providing good heuristic solutions and improving the upper bound over time.

4.6 Experimental Results

All proposed algorithms as well as algorithms considered in the following for comparison were implemented in C++ using GCC 4.7.3. All experiments were performed on a cluster of machines with Intel Xeon E5649 CPUs with 2.53 GHz and a memory limit of 15GB in single-threaded mode. The maximum computation time allowed for each run was limited to 15 minutes, i.e., 900 seconds.

The following algorithms are considered in this section: (*i*) A*+BS is the A*/beam search hybrid (*ii*) the A*+ACS algorithm, (*iii*) the anytime-A* variants APS and APPS from [164] which were implemented for comparison reasons, and (*iv*) a stand-alone ACS algorithm—henceforth labelled ACS-UB—using the upper bound UB for prioritizing the nodes. This last algorithm, whose primary target is producing good heuristic solutions—is studied for comparison purposes to get an impression about the impact of the novel heuristic guidance function EX() from Equation (4.13) in our A*+ACS. The results of Greedy heuristic 4.2 were inferior w.r.t. the other approaches and their results are shown by means of an aggregated table 4.4.

We also would like to point out that, during experimentation, it was noticed that the original APPS performed significantly worse with respect to the obtained solution quality when the beam width was set back to the initial value each time a new incumbent was found. Therefore, our implementation applies a purely progressive increase of the beam width after each BS run.

All considered algorithms will be evaluated by the obtained solution quality and by the *percentage gap*, which is calculated at time $t > 0$ as $gap(t) := \frac{\mathrm{ub}(t)-|s_{\mathrm{best}}(t)|}{\mathrm{ub}(t)} \cdot 100\%$, where $s_{\mathrm{best}}(t)$ denotes the best found solution at time t and $\mathrm{ub}(t)$ the upper bound obtained from the f-value of the top node of Q at time t (or the optimal solution value when already available). In case of ACS-UB, the upper bound is calculated by $\mathrm{ub}(t) := \max_{i=0,\ldots,j_{\max}}\{f(u_i) \mid Q_i \neq \emptyset \ \wedge u_i \text{ is the top node of } Q_i \text{ at time } t\}$.

4.6.1 Benchmark Instances

We use a set of benchmark instances originally provided in [18] for the LCS problem. This instance set consists of ten randomly generated instances for each combination of the number of input strings $m \in \{10, 50, 100, 150, 200\}$, the length of input strings $n \in \{100, 500, 1000\}$, and the alphabet size $|\Sigma| \in \{4, 12, 20\}$. This makes a total of 450 problem instances. In general, the results of our algorithms will be provided as averages over the ten instances of each combination. In order to compare the algorithms concerning the 2–LCPS problem, a new set of larger instances was generated with the instance generator from [18]. More specifically, for each combination of $|\Sigma| \in \{4, 12, 20\}$ and $n \in \{100, 200, 300, 400, 500\}$, ten instances were created yielding a total of 150 2–LCPS

instances. These benchmark instances are provided at `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS`.

4.6.2 Tuning of the Algorithms' Parameters

In order to ensure a fair comparison, the tuning tool irace [127] is employed to derive well-working parameter settings for all five considered algorithms. A*+BS has the following parameters: (δ) the number of A* iterations performed after each BS run, (β) the beam width, and (k) the parameter for the k-norm used in tie-breaking. APS has the following parameters: (*pack*) the number of nodes taken from the top of the queue in order to form the initial beam, and (k). APPS has the same parameters as APS and in addition (*step*) the amount of increase applied to *pack* after each BS run. Next, ACS-UB involves parameter (β), which is the number of expansions at each level of the state graph, and (k). Finally, A*+ACS has the parameters: (δ) the number of A* iterations performed after applying an iteration of ACS, (β) the number of expansion allowed at the same level of ACS, and (k).

The irace tool was applied separately for each algorithm and for each alphabet size. Analyzing preliminary experiments, it was found that the size of the alphabet has more influence on the behavior of the algorithms than the lengths of the input strings and their number. In order to obtain tuning instances, for each $|\Sigma|$ one random instance for each combination of m and n was generated. This makes a total of 15 tuning instances for each alphabet size, and 45 tuning instances in total. The tuning process for each alphabet size was given a budget of 1000 runs and each run was limited by a run time limit of 900 seconds and a memory limit of 15 GB.

Tuning for Solution Quality

The first set of tuning experiments was aimed at tuning the algorithm performance with respect to solution quality, that is, for obtaining the best possible solution quality at the end of a run. In the following we present the parameter value domains used during tuning as well as the best configurations for each algorithm as determined by irace. Note that meaningful ranges for the domains were determined by preliminary experiments.

For parameter k the values {0.1, 0.2, 0.5, 1.0, 2.0} were considered for all algorithms. The domain of parameter β in A*+BS and parameter *pack* in APS and APPS was {1, 50, 100, 500, 1000, 2000, 5000, 10000, 20000}. In contrast, in the context of ACS-UB and A*+ ACS parameter β was given domain $\{1, 5, 10, 20, 50, 100\}$. Finally, $\delta \in \{1, 5, 10, 20, 50, 100, 1000\}$ was considered for both A*+BS and A*+ ACS, and $step \in \{1, 5, 10, 50, 100, 200, 500\}$ for APPS. The best configurations as determined by irace are provided in Table 4.1.

Tuning for Small Gaps

The tuning experiments from the previous section were repeated with the aim of obtaining small gaps, thus, considering in addition to the final solution quality also the respective

Table 4.1: Tuning results concerning solution quality.

(a) A*+BS

$\|\Sigma\|$	δ	β	k
4	1	10000	0.5
12	10	2000	0.2
20	20	1000	0.1

(b) APS

$\|\Sigma\|$	*pack*	k
4	10000	0.1
12	10000	0.2
20	5000	0.1

(c) APPS

$\|\Sigma\|$	*pack*	*step*	k
4	10000	10	0.2
12	10000	10	0.2
20	5000	5	0.1

(d) ACS-ub

$\|\Sigma\|$	β	k
4	20	0.2
12	20	0.5
20	100	1.0

(e) A*+ACS

$\|\Sigma\|$	δ	β	k
4	100	10	1.0
12	50	10	1.0
20	100	20	0.1

Table 4.2: Tuning results concerning small gaps.

(a) A*+BS

$\|\Sigma\|$	δ	β	k
4	20000	500	1.0
12	10000	1000	1.0
20	10000	500	0.5

(b) APS

$\|\Sigma\|$	*pack*	k
4	20000	1.0
12	20000	1.0
20	10000	1.0

(c) APPS

$\|\Sigma\|$	*pack*	*step*	k
4	20000	500	1.0
12	10000	1000	1.0
20	20000	500	1.0

(d) ACS-ub

$\|\Sigma\|$	β	k
4	10	0.2
12	1	0.1
20	1	0.5

(e) A*+ACS

$\|\Sigma\|$	δ	β	k
4	5000	20	1.0
12	10000	10	1.0
20	5000	10	1.0

upper bounds. Naturally one may expect for this case other parameter settings to be ideal, in particular those putting more emphasize on classical A* search iterations. The parameter domains were chosen as in the previous subsection, with the exception of the δ parameter in the case of A*+BS and A*+ACS and the *step* parameter in APPS. These were chosen as $\delta \in \{1, 100, 500, 1000, 5000, 10000, 20000, 50000\}$ and *step* $\in \{1, 10, 50, 100, 500, 1000, 5000\}$. The best configurations as determined by irace are provided in Table 4.2. Indeed, it can be observed that the resulting values in particular for parameter δ, the number of classical A* iterations, increase significantly when tuning for small gaps.

4.6.3 Computational Results and Comparison

Table 4.3 shows average results of the algorithms over all instance groups with the parameter settings obtained by tuning for solution quality, while Table 4.5 shows the results with the settings obtained when targeting small gaps. Note that the results of APPS are excluded here as it turned out that they are very similar to those of APS.

Each of these tables consists of three sub-tables, one per alphabet size, and they have the following format. The first two columns indicate the type of problem instances considered in terms of n and m. Remember that the considered benchmark set consists of ten problem instances per combination of $|\Sigma|$, n and m. Consequently, each table row provides the results of the four considered algorithms (A*+BS, APS, ACS-UB, and A*+ACS) as averages over the ten respective instances. For each algorithm, column $\overline{|s|}$ lists the average final solution quality, column $\overline{\mathrm{t_{best}}}$[s] the average time (in seconds) at which the best solution of a run was found, column $\overline{\mathrm{t}}$[s] the overall average runtime, and, finally, column $\overline{\mathrm{gap}}$[%] the average gap in percent. Note that the overall run time of an algorithm can only be smaller than 900 seconds (the run time limit) when a proven optimal solution is found, or in case the memory limit of 15 GB is reached before the run time limit. The former happens for all algorithms in the context of all instances with $n = 100$, thus all considered algorithms are able to prove optimality for all these instances. The latter happens, for example, in the case of the ten instances with $|\Sigma| = 4$, $m = 10$ and $n = 1000$ (see Table 4.3a). Note that the best result in each table row is marked bold.

The following main observations can be made concerning these results.

- As mentioned already above, all algorithms are able to solve the problem instances with $n = 100$ to optimality, this within a fraction of a second. Therefore, in what follows, we will focus on the instances with $n \in \{500, 1000\}$.

- A*+ACS outperforms all other algorithms in terms of solution quality, both when tuned for solution quality and when tuned for minimizing the gap. In order to confirm this statistically—at least for the case of tuning for solution quality—Friedman's tests was performed simultaneously considering all four algorithms for the subsets of the benchmark set with different alphabet sizes.[2] Given that in all cases the test rejected the hypothesis that the algorithms perform equally, pairwise comparisons were performed using the Nemenyi post-hoc test [68]. Obtained results are shown in Figure 4.1 by means of so-called critical difference plots. Each algorithm is positioned in the segment according to its average ranking concerning the considered subset of instances. Then, the critical difference (CD) is computed for a significance level of 0.05 and the performance of those algorithms that have a difference lower than CD are regarded as equal—that is, no difference of statistical significance can be detected. This is indicated in the figures by horizontal bars

[2] All these tests and the resulting plots were generated using R's **scmamp** package [29].

Table 4.3: Average results of the algorithms when tuned for solution quality.

(a) $|\Sigma| = 4$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	**28.9**	1.2	5.3	0.0	**28.9**	1.0	1.8	0.0	**28.9**	< 0.1	2.4	0.0	**28.9**	< 0.1	1.9	0.0
	500	159.9	13.7	368.7	45.4	159.9	9.5	251.0	44.4	161.2	130.1	415.5	43.5	**162.3**	61.2	573.6	42.6
	1000	323.1	41.0	278.8	47.63	323.3	56.3	269.1	47.3	326.2	146.3	301.3	46.6	**330.7**	256.1	532.1	45.7
50	100	**21.8**	1.3	1.3	0.0	**21.8**	1.0	1.0	0.0	**21.8**	< 0.1	0.9	0.0	**21.8**	< 0.1	0.6	0.0
	500	130.5	25.2	468.2	54.1	130.5	18.2	346.4	53.1	131.3	111.9	490.2	52.5	**132.7**	135.0	555.4	51.4
	1000	267.9	177.8	576.6	56.0	267.8	83.0	507.1	55.7	268.9	314.7	683.1	55.5	**273.0**	92.7	722.1	54.5
100	100	**20.1**	1.7	1.7	0.0	**20.1**	1.2	1.2	0.0	**20.1**	< 0.1	1.2	0.0	**20.1**	< 0.1	0.8	0.0
	500	123.5	53.5	616.2	56.3	123.5	64.3	459.4	55.4	124.1	107.3	651.4	54.9	**125.1**	86.0	688.4	54.0
	1000	254.8	123.0	686.2	58.0	254.8	127.9	563.0	57.7	255.4	324.0	744.1	57.6	**259.8**	199.8	765.8	56.6
150	100	**19.0**	2.1	2.2	0.0	**19.0**	1.4	1.4	0.0	**19.0**	< 0.1	0.9	0.0	**19.0**	< 0.1	0.5	0.0
	500	120.3	128.1	792.7	57.2	120.2	51.8	528.6	56.4	120.7	118.1	736.7	56.1	**121.7**	67.2	723.8	55.1
	1000	249.0	170.3	750.3	58.8	249.0	155.8	580.1	58.6	249.8	337.8	780.8	58.4	**253.3**	192.7	752.6	57.6
200	100	**18.5**	1.8	1.8	0.0	**18.5**	1.7	1.7	0.0	**18.5**	0.1	1.1	0.0	**18.5**	< 0.1	0.7	0.0
	500	118.0	99.1	843.3	57.9	118.0	74.5	601.5	57.2	118.4	216.6	852.4	56.9	**119.5**	25.6	778.4	55.9
	1000	245.0	287.2	859.9	59.4	244.8	196.2	671.8	59.3	245.4	275.0	884.5	59.1	**249.4**	238.2	840.9	58.2

(b) $|\Sigma| = 12$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0
	500	61.5	3.5	453.0	65.1	61.8	15.5	371.4	62.0	62.3	77.1	400.9	61.3	**62.4**	26.4	649.7	60.6
	1000	126.7	6.3	291.3	68.2	127.2	28.2	281.9	67.1	128.7	89.1	309.3	66.6	**130.5**	115.5	531.3	65.9
50	100	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0
	500	43.3	5.7	555.7	73.8	43.6	25.2	548.0	70.6	43.8	41.4	580.8	70.4	**44.3**	96.0	617.5	69.3
	1000	91.1	21.5	568.8	76.4	91.7	92.7	714.8	75.3	92.4	246.2	608.9	75.0	**93.7**	70.2	649.8	74.4
100	100	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0
	500	39.0	19.2	798.0	75.8	39.0	46.9	829.4	72.8	39.1	35.8	856.4	72.4	**39.6**	88.0	798.4	71.5
	1000	83.9	66.2	879.1	78.0	84.1	147.3	812.9	77.1	84.7	291.2	891.6	76.9	**85.9**	78.6	871.2	76.3
150	100	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0
	500	37.0	22.6	900.0	76.8	37.1	68.8	900.0	73.9	37.2	52.8	900.0	73.7	**37.6**	22.6	881.7	72.8
	1000	80.3	44.4	900.0	78.8	80.6	209.3	900.0	78.0	81.0	212.7	900.0	77.9	**82.2**	51.9	900.0	77.2
200	100	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0
	500	35.8	60.6	900.0	77.3	**36.0**	90.8	900.0	74.4	**36.0**	31.8	900.0	75.0	**36.0**	0.6	900.0	73.7
	1000	78.2	152.1	900.0	79.2	78.4	297.7	900.0	78.5	78.7	273.0	900.0	78.4	**80.0**	126.8	900.0	77.8

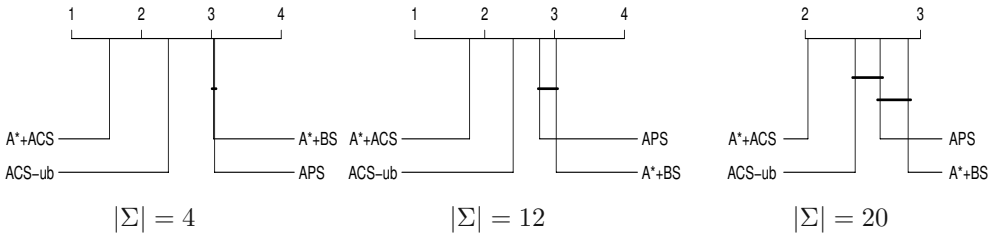
(c) $|\Sigma| = 20$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{s}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0
	500	38.7	9.7	351.4	69.5	38.6	7.7	523.2	65.2	**38.9**	5.8	319.1	64.6	**38.9**	1.4	801.9	63.8
	1000	79.7	60.3	353.8	74.2	79.9	20.3	500.2	72.9	80.9	75.6	386.2	72.5	**81.8**	133.4	688.0	71.8
50	100	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0
	500	25.0	4.3	750.3	76.8	25.0	15.0	900.0	71.2	**25.1**	23.2	740.1	72.2	**25.1**	74.9	858.7	71.0
	1000	54.4	44.9	768.7	81.5	54.6	46.8	888.0	80.2	55.0	101.8	728.0	80.0	**55.6**	42.6	881.8	79.5
100	100	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0
	500	21.8	8.1	900.0	78.6	21.9	26.0	900.0	73.2	**22.1**	56.3	893.6	73.5	**22.1**	5.9	900.0	72.7
	1000	48.9	62.8	900.0	83.2	48.9	89.9	900.0	82.1	49.1	71.8	899.3	81.9	**50.1**	110.1	900.0	81.2
150	100	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0
	500	20.6	8.3	900.0	79.4	20.9	46.5	900.0	74.2	**21.0**	43.0	900.0	73.7	**21.0**	8.7	900.0	73.3
	1000	46.3	99.6	900.0	84.0	46.6	202.8	900.0	82.8	46.8	139.4	900.0	82.7	**47.2**	124.9	900.0	82.2
200	100	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0
	500	19.5	74.0	900.0	80.1	19.5	49.0	900.0	74.8	19.9	65.6	900.0	73.2	**20.0**	91.3	900.0	73.5
	1000	44.8	56.2	900.0	84.4	45.0	190.6	900.0	83.4	45.0	42.7	900.0	83.3	**45.7**	175.4	900.0	82.7

joining the respective algorithms. The figures show that, for each alphabet size, A*+ACS outperforms the other three algorithms with statistical significance.

Table 4.4: Average improvements (in percent) in final solution quality/length of A*+ACS over other algorithms, aggregated over all values for m.

$\|\Sigma\|$	n	A*+BS[%]	APS[%]	ACS-UB[%]	GREEDY[%]
4	100	0.00	0.00	0.00	13.13
	500	1.38	1.40	0.86	12.24
	1000	1.95	1.96	1.53	9.58
12	100	0.00	0.00	0.00	12.00
	500	1.50	1.09	0.73	14.30
	1000	2.58	2.19	1.47	12.47
20	100	0.00	0.00	0.00	3.34
	500	1.36	1.03	0.10	18.08
	1000	2.25	1.90	1.33	15.35



$|\Sigma| = 4$ $|\Sigma| = 12$ $|\Sigma| = 20$

Figure 4.1: Critical difference plots concerning the results of the algorithms tuned for solution quality. The benchmark instances are split according to alphabet size.

- Furthermore, it can be observed that A*+ACS outperforms the other three algorithms also concerning the gap. Again, this holds both when tuned for solution quality and when tuned for minimizing the gap. The corresponding critical difference plots—concerning the results obtained after tuning for minimizing the gap—are shown in Figure 4.2. They confirm that A*+ACS outperforms the other algorithms with statistical significance. As all algorithms make use of the same upper bound function, the difference in gaps must be attributed to the fact that A*+ACS produces significantly better primal solutions than the other algorithms.
- Concerning the remaining three algorithms, it can be observed that A*+BS is generally the weakest algorithm with respect to solution quality. However, this algorithm usually provides better gaps. This is with the exception of instances with $|\Sigma| = 20$ where A*+BS also exhibits the weakest performance regarding the gaps. The best algorithm among A*+BS, APS and ACS-UB concerning solution quality is ACS-UB.

Table 4.4 further summarizes the results concerning the final solution quality by showing the average improvements (in percent) of A*+ACS over the other algorithms, aggregated over all values for m. This table also includes the improvements over the simple greedy

Table 4.5: Average results of the algorithms when tuned for small gaps.

(a) $|\Sigma| = 4$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	28.9	0.1	1.3	**0.0**	28.9	1.6	1.9	**0.0**	28.9	0.0	2.6	**0.0**	28.9	0.1	2.2	**0.0**
	500	158.6	28.4	287.7	43.5	160.3	27.4	335.9	44.2	161.1	144.7	486.8	43.4	162.0	67.7	614.2	**42.0**
	1000	320.2	132.6	351.0	47.2	324.2	36.2	200.6	47.1	326.2	233.4	526.9	46.6	330.0	143.3	542.3	**45.4**
50	100	21.8	0.2	0.7	**0.0**	21.8	1.2	1.2	**0.0**	21.8	0.0	0.8	**0.0**	21.8	0.1	0.6	**0.0**
	500	129.4	48.6	476.0	52.2	130.9	37.7	341.7	52.9	131.3	147.9	676.8	52.3	132.3	93.7	551.2	**50.9**
	1000	266.2	156.0	605.5	55.4	268.2	123.9	492.4	55.6	268.9	327.3	747.1	55.4	273.0	217.4	560.3	**54.1**
100	100	20.1	0.3	0.9	**0.0**	20.1	1.4	1.4	**0.0**	20.1	0.0	1.1	**0.0**	20.1	0.1	0.8	**0.0**
	500	122.6	55.0	606.9	54.4	123.7	69.2	438.4	55.2	124.1	139.1	852.3	54.8	124.9	55.3	607.2	**53.3**
	1000	253.8	141.7	675.9	57.4	255.2	195.5	527.0	57.6	255.6	469.6	889.5	57.4	259.5	226.7	672.4	**56.2**
150	100	19.0	0.3	0.7	**0.0**	19.0	1.6	1.6	**0.0**	19.0	0.0	0.7	**0.0**	19.0	0.1	0.7	**0.0**
	500	119.3	24.8	719.1	55.6	120.5	104.7	500.9	56.3	120.7	161.8	894.1	55.9	121.4	25.3	848.4	**54.7**
	1000	247.9	28.4	743.7	58.3	249.4	481.3	830.0	58.5	249.3	214.9	900.0	58.4	253.0	142.8	720.1	**57.2**
200	100	18.5	0.4	0.8	**0.0**	18.5	1.6	1.6	**0.0**	18.5	0.1	1.0	**0.0**	18.5	0.1	0.7	**0.0**
	500	117.3	47.7	810.6	56.2	118.0	199.4	672.2	57.1	118.4	267.5	900.0	56.8	119.5	140.8	768.1	**55.2**
	1000	244.3	93.1	831.8	58.8	245.0	377.7	614.5	59.2	245.4	365.6	900.0	59.1	249.4	349.7	802.3	**57.8**

(b) $|\Sigma| = 12$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**
	500	61.4	1.2	270.2	60.8	61.8	32.2	402.4	61.9	62.3	68.1	332.5	61.3	62.2	20.0	601.2	**59.5**
	1000	125.7	3.3	338.6	67.0	128.1	70.6	353.7	66.8	128.7	98.2	332.3	66.6	130.0	151.0	592.1	**65.3**
50	100	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**
	500	43.1	2.5	426.9	69.7	43.7	38.2	418.7	70.5	43.8	44.1	574.6	70.4	44.1	80.1	621.3	**68.5**
	1000	91.0	76.2	658.4	75.0	91.9	153.1	622.3	75.2	92.3	249.5	736.4	75.1	93.2	74.0	685.5	**74.0**
100	100	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**
	500	38.9	4.3	656.2	71.6	39.0	72.4	640.3	72.7	39.1	35.4	804.6	72.6	39.2	35.8	840.1	**71.0**
	1000	83.6	17.5	759.3	76.8	84.3	248.6	696.7	77.0	84.7	287.7	854.4	76.9	85.4	143.9	896.2	**75.9**
150	100	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**
	500	37.0	6.5	784.6	72.7	37.1	107.3	762.7	73.7	37.2	49.3	892.7	73.7	37.3	146.8	891.5	**72.1**
	1000	80.1	18.4	839.8	77.7	80.9	329.1	756.2	77.8	81.0	202.6	900.0	77.9	81.7	171.5	900.0	**76.9**
200	100	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**
	500	35.6	8.4	867.2	73.5	36.0	140.2	850.0	74.2	36.0	18.1	900.0	74.5	36.0	20.3	900.0	**72.9**
	1000	77.9	26.3	900.0	78.2	78.6	469.9	867.9	78.4	78.8	343.4	900.0	78.3	79.4	117.8	900.0	**77.5**

(c) $|\Sigma| = 20$.

m	n	A*+BS				APS				ACS-UB				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
10	100	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**
	500	34.4	39.4	457.4	66.8	38.7	15.2	493.4	64.9	38.9	8.2	532.6	64.1	38.9	17.8	581.5	**62.4**
	1000	70.3	96.9	403.1	75.1	80.2	35.3	374.9	72.8	80.9	157.0	580.5	72.2	81.3	115.6	700.3	**71.2**
50	100	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**
	500	23.2	44.8	749.8	72.2	25.0	22.3	873.8	70.7	25.1	38.6	900.0	71.8	25.0	6.7	735.4	**70.2**
	1000	50.2	155.7	786.3	80.9	54.8	87.8	654.5	80.2	55.0	115.4	900.0	79.9	55.5	124.3	825.6	**79.0**
100	100	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**
	500	20.4	151.5	900.0	73.7	21.9	46.9	900.0	72.9	22.1	68.5	900.0	73.6	21.9	24.1	900.0	**71.6**
	1000	45.6	197.9	900.0	82.5	49.0	139.7	900.0	81.9	49.0	32.0	900.0	81.8	49.7	73.6	900.0	**80.9**
150	100	1.1	4.2	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**
	500	19.0	10.0	900.0	74.8	20.9	60.9	900.0	73.4	21.0	79.8	900.0	72.8	20.9	41.7	900.0	**72.1**
	1000	43.2	114.9	900.0	83.3	46.7	244.5	900.0	82.7	46.7	100.4	900.0	82.7	47.0	47.2	900.0	**81.8**
200	100	1.1	3.9	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**
	500	18.1	52.1	900.0	76.2	19.7	104.0	900.0	74.7	19.9	112.9	900.0	73.9	19.8	288.6	900.0	**73.4**
	1000	42.2	341.8	900.0	83.6	45.0	280.5	900.0	83.3	45.0	48.9	900.0	83.3	45.1	36.6	900.0	**82.6**

heuristic from [54] (see last table column). Except for the smallest instances with $n = 100$, where all anytime variants were able to find proven optimal solutions, the consistent

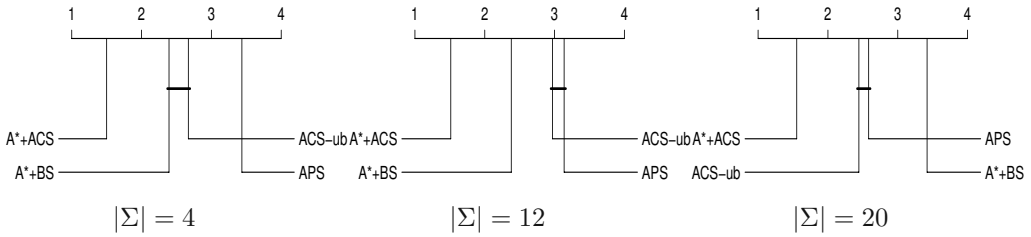
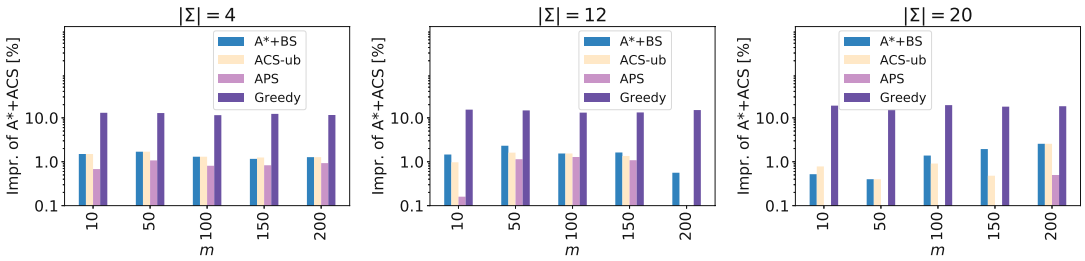
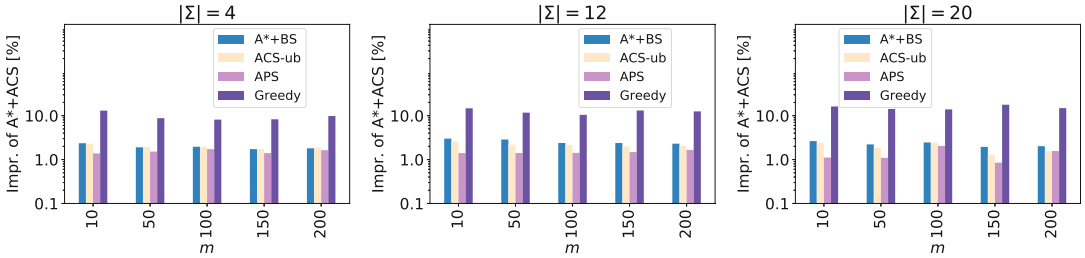


Figure 4.2: Critical difference plots concerning the results of the algorithms tuned for small gaps. The benchmark instances are split according to alphabet size.



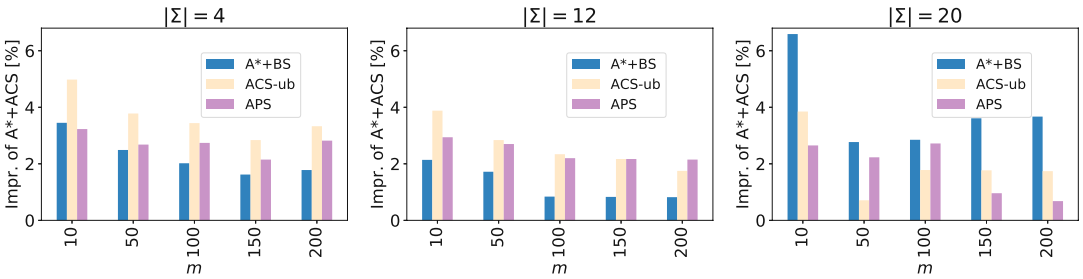
(a) Instances with $n = 500$.



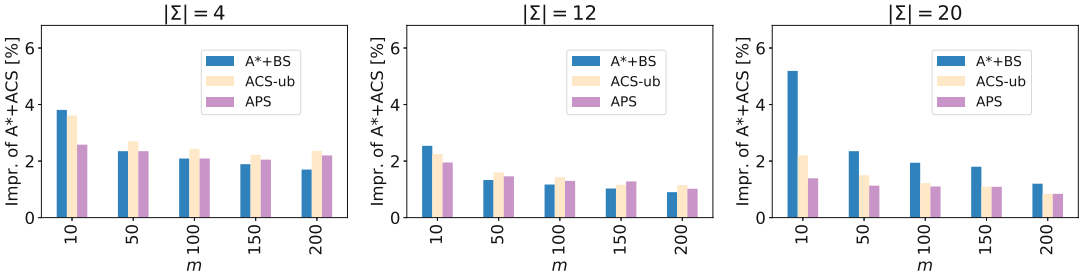
(b) Instances with $n = 1000$.

Figure 4.3: Average improvements (in percent) in solution quality/length of A*+ACS over the other algorithms.

benefits of A*+ACS are again clearly observable. We can see that the largest differences occur, in general, for $n = 1000$ and $|\Sigma| = 20$, and the average improvements of A*+ACS are up to $\approx 18\%$ over the greedy heuristic and up to 2.58% over the other anytime algorithms. In addition, the improvements of A*+ACS are graphically shown for $n = 500$ and $n = 1000$ in Figure 4.3; note the logarithmic scaling of the y axes. Figure 4.4 further presents the improvements of A*+ACS over the other anytime algorithms concerning the final gaps.



(a) Instances with $n = 500$.



(b) Instances with $n = 1000$.

Figure 4.4: Average improvements (in percent) concerning the final gaps of A*+ACS over the other algorithms.

4.6.4 The Anytime Performance of the Algorithms

As stated above, apart from the solution quality and the optimality gap finally obtained by the algorithms when the time limit is reached, another important aspect of their behaviour concerns the anytime performance. In order to visualize the anytime performance, we plot the evolution of the solution quality over time (averaged over ten problem instances of the same type). Figure 4.5 shows these plots for the representative case with $m = 50$ and $n = 1000$ considering all three alphabet sizes. In addition to the line plots for the average behaviour, boxplots indicating the variance are shown every 200 seconds.[3] The evolution of the obtained average gaps over time are shown in the same way in Figure 4.6. Note that information is only plotted concerning complete—in the sense of non-expandable—solutions. This is the reason why, for instance, the anytime line plot of APS in the three graphics of Figure 4.5 does not start at time zero. The list of all anytime plots w.r.t. solution quality and gaps quality are given in Appendix B.2 and Appendix B.3.

The following observations can be made with respect to the anytime plots on solution quality.

[3]We provide the complete set of graphics, concerning all combinations of n and m, as supplementary material under `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS`.

- A*+ACS outperforms all other approaches during all stages of the search process. That is, A*+ACS finds better solutions than the other algorithms already very early during the search process. Moreover, A*+ACS does not seem to suffer as much from early stagnation as A*+BS. This boost of solution quality can primarily be attributed to the incorporation of the new approximate expected length calculation (4.13) as heuristic function, which turns out to be much better guidance than classical upper bounds. A direct indication for this is obtained when comparing the anytime plots of A*+ACS and ACS-UB (which does not make use of the approximate expected length function).

- APS and A*+BS, which both make use of embedded BS runs in order to find good heuristic solutions, show a similar anytime behaviour considering solution quality. However, a rather large beam size (β) is required in order to obtain the best possible solution quality at the end of a run. This fact is obviously negative for the anytime performance of the algorithms, as they perform very few major iterations. The role of the A* iterations is almost irrelevant for A*+BS.

- ACS-UB not only outperforms A*+BS and APS concerning the final solution quality, but it also shows an improved anytime performance when comparing with the ones of APS and A*+BS.

When considering the anytime plots concerning the evolution of the gaps, the following can be observed:

- A*+ACS produces significantly better gaps when compared to those of the three other algorithms, over the whole run-time of the algorithms. This means that the significantly increased number of A* iterations (when compared to the parameter setting aimed for solution quality; see Tables 4.1e and 4.2e) pays off for A*+ACS. However, it is also interesting to remark that, even with the parameter setting aimed for minimizing the gaps, the algorithm still provides a performance concerning solution quality that outperforms all other approaches.

- Even though A*+BS uses a number of A* iterations that is one order of magnitude larger than the one used by A*+ACS, this does not pay off for the former algorithm. In the case of the instances with $|\Sigma| = 20$, for example, A*+BS shows by far the worst anytime performance in the comparison.

- ACS-UB and APS show a similar anytime performance concerning the evolution of the gaps.

- Generally, A*+ACS shows a very good balance between ensuring good gaps and providing high-quality heuristic solutions. This can be explained by the use of both, an improved upper bound function and a strong guidance by our approximate heuristic, see (4.13).

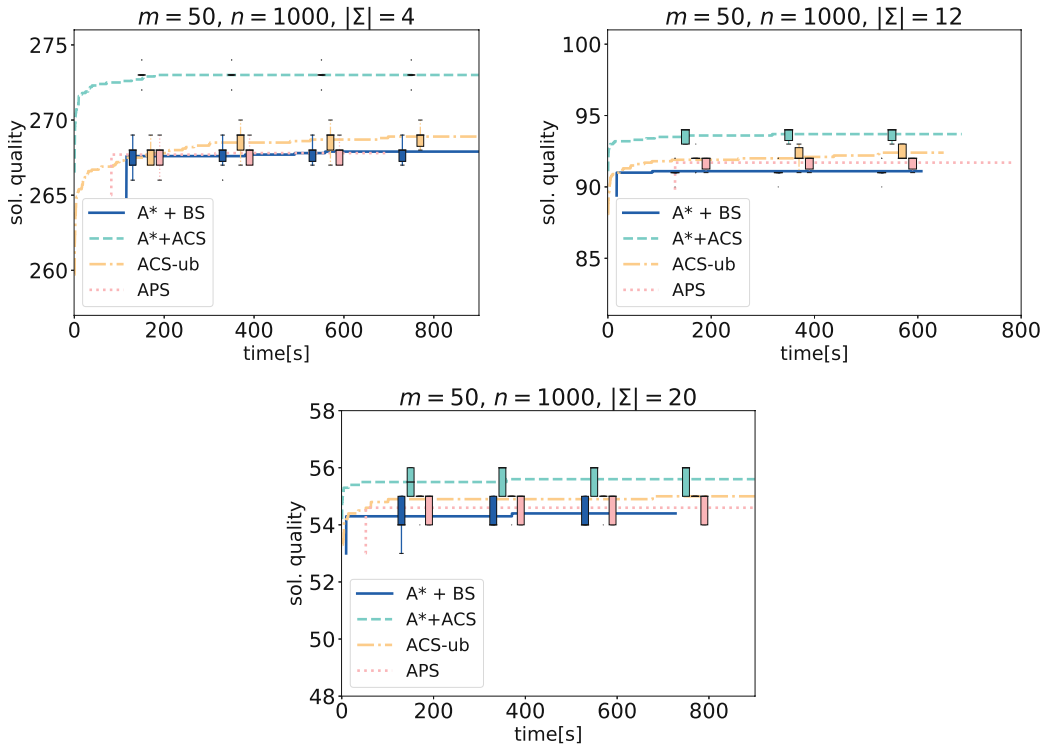


Figure 4.5: Evolution of solution quality over time for the exemplary case with $m = 50$ and $n = 1000$. The algorithms were tuned for solution quality.

Table 4.6: Overview on the 2–LCPS algorithms from the literature.

Algorithm	**Main idea**
CPSA	common palindromic subsequence automata [80]: an automaton accepting a language consisting of all common palindromic subsequences is constructed for each input string and its reversal; the longest path in the intersection of the two automata for both input strings, found by topological sorting, corresponds to an optimal solution
DP	dynamic programming [90]: a subproblem for the 2–LCPS problem is specified by a 4-D vector (i, j, k, l), $i, j, k, l \in \{1, \ldots, n\}$, representing the substrings $s_1[i, j]$ and $s_2[k, l]$
MNDRS	minimum depth nested rectangular structures [38]: a sparse DP approach based on a geometric problem interpretation; a rectangle is associated with each 4D–matching (hereby, (i, k) is the bottom-left corner and (j, l) is the top-right corner); finding the longest sequence of nested rectangles yields an optimal solution

4.6.5 Computational Study for the 2–LCPS Problem

As already pointed out in Section 4.1.1, the existing works from the literature on the LCPS problem primarily consider exact algorithms for the problem variant with only two input strings ($m = 2$), that is, the 2–LCPS problem. We implemented all these approaches—that is, the DP and the MNDRS approaches from [38] and the CPSA

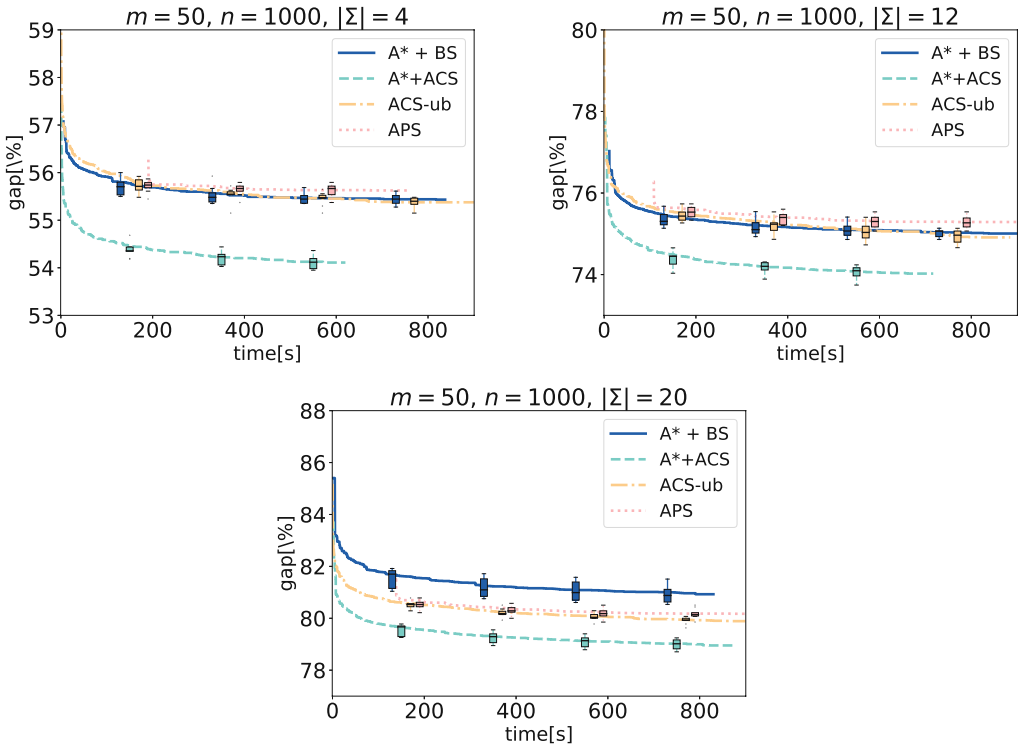


Figure 4.6: Evolution of the optimality gaps over time for the exemplary case with $m = 50$ and $n = 1000$. The algorithms were tuned for minimizing the gap.

approach from [80]—for the experimental evaluation. A summary of the main ideas of the competing approaches is provided in Table 4.6; see also Section 4.1.1 for more details. Note, however, as some of these approaches are described in the original articles only from a theoretical point of view, we sometimes had to make our own design decisions for what concerns, for example, suitable data structures. A detailed description of our implementations can be found in Appendix B.4. In addition to these three approaches we tested a basic *Constraint Programming* (CP) model, detailed in Appendix B.1 in conjunction with MiniZinc 2.1.5 and its Gecode backbone solver.

Finally, our pure A* search as described in Section 4.3 is also considered in the comparison. Compared to the hybrid A*–based approaches, pure A* search can be expected to require fewer node expansions to prove optimality.

The five considered approaches were applied once with a computation time limit of 900 seconds and a memory limit of 15 GB to the 150 2–LCPS instances described in Section 4.6.1. Results are shown in Table 4.7, which lists for each instance group (n, $|\Sigma|$) consisting of ten instances and for each approach the number of instances the method was able to solve to proven optimality (*#opt*), the number of instances for which the method was terminated either due to exceeding the time limit (*#te*) or the memory limit

Table 4.7: Results for the 2–LCPS instances.

n	$\|\Sigma\|$	A*				MNDRS				CPSA				DP				CP			
		$\#opt$	$\#te$	$\#me$	$\bar{t}[s]$	$\#opt$	$\#te$	$\#me$	$\bar{t}[s]$	$\#opt$	$\#te$	$\#me$	$\bar{t}[s]$	$\#opt$	$\#te$	$\#me$	$\bar{t}[s]$	$\#opt$	$\#te$	$\#me$	$\bar{t}[s]$
100	4	10	0	0	0.2	10	0	0	0.4	10	0	0	0.4	10	0	0	1.1	10	0	0	15.8
	12	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.2	10	0	0	1.1	10	0	0	4.9
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.2	10	0	0	1.1	10	0	0	1.3
200	4	10	0	0	0.6	10	0	0	8.5	10	0	0	27.4	10	0	0	25.6	0	10	0	–
	12	10	0	0	0.3	10	0	0	0.2	10	0	0	2.7	10	0	0	20.7	0	10	0	–
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.8	10	0	0	13.7	0	10	0	–
300	4	10	0	0	5.2	10	0	0	45.7	10	0	0	431.3	10	0	0	61.5	0	10	0	–
	12	10	0	0	0.3	10	0	0	1.4	10	0	0	22.1	10	0	0	60.9	0	10	0	–
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	5.9	10	0	0	54.0	0	10	0	–
400	4	10	0	0	26.6	9	0	1	158.9	0	10	0	–	0	0	10	–	0	10	0	–
	12	10	0	0	7.9	10	0	0	7.6	10	0	0	154.0	0	0	10	–	0	10	0	–
	20	10	0	0	2.9	10	0	0	1.6	10	0	0	31.1	0	0	10	–	0	10	0	–
500	4	10	0	0	64.9	0	0	10	–	0	10	0	–	0	0	10	–	0	10	0	–
	12	10	0	0	24.3	10	0	0	17.8	10	0	0	745.0	0	0	10	–	0	10	0	–
	20	10	0	0	9.8	10	0	0	4.9	10	0	0	108.2	0	0	10	–	0	10	0	–

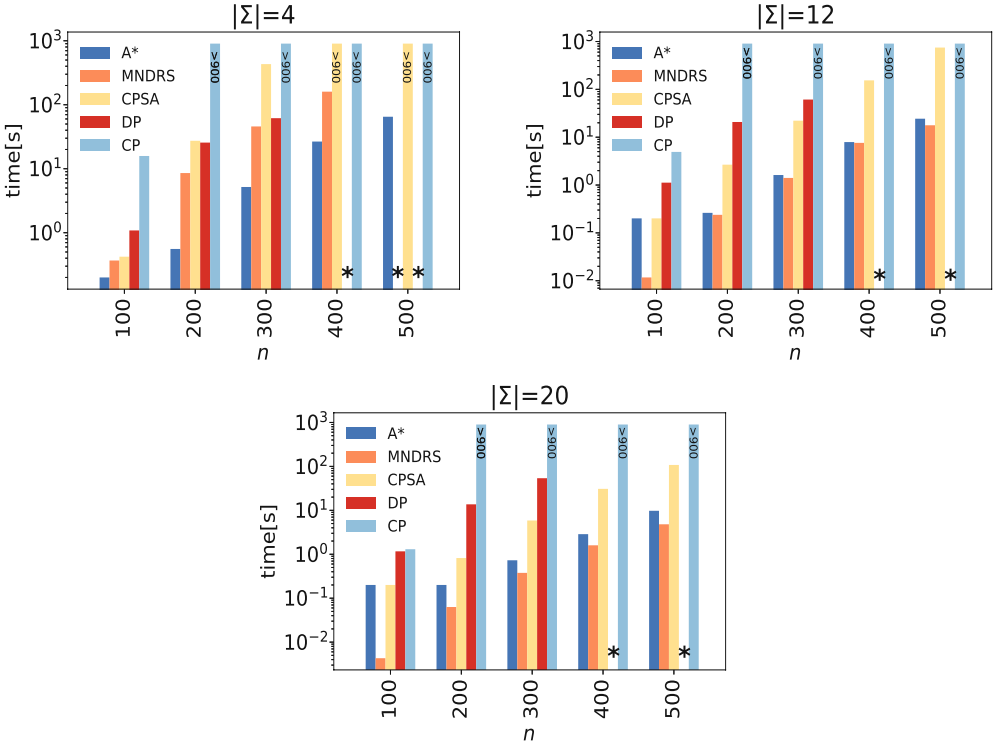


Figure 4.7: Average computation times of the algorithms for 2–LCPS.

($\#me$), and the average computation times of all successful runs (or "–" if no run could prove optimality). The average computation times are also provided in graphical form in Figure 4.7. Note that the y-axis of these plots uses a logarithmic scaling. Moreover, cases in which the memory limit was exceeded are marked with a black asterisk, while cases in which the maximum allowed computation time was exceeded are marked with ">900".

The obtained results allow to draw the following conclusions:

- Our A* approach is the only algorithm that can find all optimal solutions and prove their optimality both within the time limit and respecting the imposed memory constraint.
- MNDRS is the second-best algorithm, only starting to fail for instances with $|\Sigma| = 4$ and $n \in \{400, 500\}$. In particular, MNDRS fails due to exceeding the memory limit due to the data structures it requires.
- CPSA fails for the same problem instance types as MNDRS. However, in contrast to MNDRS it fails due to exceeding the computation time limit.
- DP is only able to solve problem instances up to $n = 300$. Starting from $n = 400$, the algorithm fails due to the memory limit.
- The CP approach is clearly the weakest one in the comparison. This approach is only able to solve the problem instances with $n = 100$. In all other cases, CP fails due to reaching the computation time limit. In fact, starting from $n = 400$, CP is not able to provide any solution within the allowed computation time.
- Concerning the computation time requirements, it can be observed that A* and MNDRS—when able to solve an instance—are the fastest approaches. A* has advantages in the context of instances with $|\Sigma| = 4$. For instances with $|\Sigma| = 12$ both algorithms require comparable times, and for $|\Sigma| = 12$ MDRS is on average slightly faster. On the other side, CP is by far the most time consuming approach.

Finally, we would like to stress again that A* is the only algorithm which was able to solve all instances to optimality, respecting the time and memory limits. This confirms that the way of merging the nodes in the state graph of the A* search has a crucial impact on reducing the algorithms' memory consumption.

4.7 Conclusions

In this chapter we considered the LCPS problem and studied a variety of algorithms for it. For the exact solving of the problem, we proposed an A* search guided by a more effective upper bound calculation. This algorithm is able to reliably solve LCPS instances with two input strings to proven optimality in short time, even when the strings have lengths up to 500 letters. None of the other solution approaches considered here performed equally consistent.

When it comes to solve the LCPS for a larger number of strings, however, the classical A* search clearly also has its limits due to the complexity of the problem. In this case, anytime algorithms are particularly interesting for practice, as they deliver promising heuristic solutions almost immediately, can be expected to continuously improve on them, and still retain the chance of finishing with proven optimality when the time allows. The first algorithm of this kind for the LCPS problem was the hybrid A*+BS, which embeds

beam search in the A* framework. As a weakness it has been recognized that calling the beam search more frequently with lower beam width is usually substantially less effective than calling it only fewer times with larger beam width. Unfortunately, this property stays in contrast to the goal of an anytime approach to obtain improved solutions more continuously.

With A*+ACS, a clearly superior approach is provided. It replaces the beam search with iterations of anytime column search, which expands nodes at all levels more uniformly and therefore leads to a more continuous improvement. Most importantly, we also introduced a novel heuristic function that represents an approximation for the expected length of the LCPS. Using this function as guidance within the ACS iterations instead of the classical upper bound calculation leads to substantially better heuristic solutions. In order to still obtain quality guarantees and optimality proofs when time allows, classical A* iterations still rely on the (improved) upper bound calculation.

Different parameter settings are suitable for A*+ACS whether one aims just on pure heuristic performance or when considering also upper bounds and wanting to minimize optimality gaps. Detailed parameter tuning for both cases was performed using irace, and the obtained settings provide a solid basis for reasonable choices when confronted with new instances. Our computational evaluation and comparison of different A*-based anytime approaches for the LCPS clearly show the benefits and superiority of the new A*+ACS concerning final solution quality, final remaining optimality gap, as well as the overall anytime-behavior.

CHAPTER 5

The Longest Common Square Subsequence Problem

The longest common square subsequence (LCSqS) problem, a variant of the longest common subsequence (LCS) problem, aims at finding a subsequence common to all input strings that is, at the same time, a square subsequence. A string s is a *square* iff $s = s' \cdot s'$ for some string s'. The content of this chapter is based on

- an article published in the Proceedings of *the 17th International Conference on Computer Aided Systems Theory* (EUROCAST 2019) conference [56]. So far the LCSqS was considered only for two input strings. In order to tackle the LCSqS with an arbitrary set of input strings, this paper proposes two heuristic approaches: (i) a randomized local search, and (ii) a hybrid of variable neighborhood search and beam search.

5.1 Introduction

The longest common square subsequence (LCSqS) problem was proposed by Inoue et al. [91]. The length of the LCSqS can be seen as a measure of similarity between disjunctive parts of each of the molecules. Therefore, it can give more insight into the internal similarity of molecules, comparing to the basic variants of the LCS problem. The information about the internal similarity between parts of the molecules can be obtained by detecting the length of an LCSqS. Inoue et al. [91] proved that the LCSqS problem is $\mathcal{NP}$-hard for an arbitrary set of input strings and proposed two approaches for solving the LCSqS with two input strings: (i) a *Dynamic Programming* (DP) approach which runs in $O(n^6)$ time, and (ii) a sparse DP-based approach, which makes use of a special geometric data structure known as multidimensional balanced 3D range trees. It can be proven that, if m is fixed, the LCSqS is polynomially solvable by DP in $O(n^{3m})$ time

which is not practical even for small m. To the best of our knowledge, no algorithm has been so far proposed for solving the LCSqS problem with an arbitrary number $m \geq 2$ of input strings. The highlights of this chapter are as follow:

- A transformation of the LCSqS problem to a series of the standard LCS problems is described.
- An approach based on a randomized local search (RLS) and a hybrid of *a Reduced Variable Neighborhood Search* (RVNS) [133] and *a Beam Search* (BS) for solving the LCSqS problem with an arbitrary number of input strings are proposed.
- An approximation of the expected length of an LCS problem is incorporated into the BS framework to guide its search towards more promising regions where high-quality LCSqS solutions could be produced.

The rest of the chapter is organized as follows. Section 5.2 gives a basic reduction from the LCSqS to the series of the LCS problem and two approaches to solve this problem. Section 5.3 provides computational results, and Section 5.4 gives an overview over this study and some research questions which might be promising for further consideration.

5.2 Algorithms for Solving the LCSqS Problem

Let us denote by $\mathbb{P} := \{(q_1, \ldots, q_m) : 1 \leq q_i \leq |s_i|\} \subset \mathbb{N}^m$ all possibilities for partitioning the strings from S each one into two consecutive substrings. For each $\mathbf{q} \in \mathbb{P}$, we define the left and right partitions of S by $S^{\mathrm{L},\mathbf{q}} = \{s_1[1, q_1], \ldots, s_m[1, q_m]\}$ and $S^{\mathrm{R},\mathbf{q}} = \{s_1[q_1 + 1, |s_1|], \ldots, s_m[q_m + 1, |s_m|]\}$, respectively. Let $S^{\mathbf{q}} := S^{\mathrm{L},\mathbf{q}} \cup S^{\mathrm{R},\mathbf{q}}$ be the joint set of these partitions. Finding an optimal solution s^*_{lcsqs} to the LCSqS problem can then be done as follows. First, an optimal LCS $s^*_{\mathrm{lcs},\mathbf{q}}$ must be derived for all $S^{\mathbf{q}}$, $\mathbf{q} \in \mathbb{P}$. Let $s^*_{\mathrm{lcs}} = \arg\max\{|s^*_{\mathrm{lcs},\mathbf{q}}| : \mathbf{q} \in \mathbb{P}\}$ Then, $s^*_{\mathrm{lcsqs}} = s^*_{\mathrm{lcs}} \cdot s^*_{\mathrm{lcs}}$. Unfortunately, the LCS problem is already $\mathcal{NP}$–hard [128], and the size of $\mathbb{P}$ grows exponentially with the instance size. This approach is, therefore, not practical. However, we will make use of this decomposition approach in a heuristic way as shown in the following two sections.

5.2.1 Randomized Local Search Approach

In this section we adapt and iterate Best–Next constructive heuristic (BNH) [65] (see Chapter 2.2.1) for the LCS problem in order to derive approximate LCSqS solutions in the sense of a randomized local search (RLS). Pseudocode of the RLS if given in Algorithm 22. We start with the $\mathbf{q} = \left(\lfloor \frac{|s_1|}{2} \rfloor, \ldots, \lfloor \frac{|s_m|}{2} \rfloor\right)$ and by executing a BNH on the corresponding set $S^{\mathbf{q}}$ to produce an initial approximate LCSqS solution $s_{\mathrm{lcsqs}} = \mathrm{BNH}(S^{\mathbf{q}})^2$. At each iteration, q is perturbed by adding to each q_i, $i = 1, \ldots, m$, a random offset sampled from the discretized normal distribution $\lceil \mathcal{N}(0, \sigma^2) \rceil$ with a probability $destr \in (0, 1)$, where the standard deviation is a parameter of the algorithm. BNH is applied to

Algorithm 22 Randomized Local Search

Input: an instance (S, Σ), destruction parameter $destr \in (0, 1)$, standard deviation σ
Output: a feasible LCSqS solution
$\mathbf{q} \leftarrow q_i = \left\lfloor \frac{|s_i|}{2} \right\rfloor$, $i = 1, ..., m$
$s_{\text{lcsqs}} \leftarrow \varepsilon$
while $t_{\max}$ exceeded **do**
    $\mathbf{q'} \leftarrow$ perturb $\mathbf{q}$ acc. to *destr* probability defining offset w.r.t. standard distr. σ
    $s \leftarrow \text{BNH}(S^{\mathbf{q'}})$
    if $2 \cdot |s| > |s_{\text{best}}|$ **then**
        $s_{\text{lcsqs}} \leftarrow s \cdot s$
        $\mathbf{q} \leftarrow \mathbf{q'}$
    end if
end while
return s_{best}

the resulting string set $S^{\mathbf{q}}$ for producing a new solution. A better solution is always accepted as new incumbent solution s_{lcsqs}. The whole process is iterated until a time limit $t_{\max} > 0$ is exceeded. Note that if s_{lcsqs} is the current incumbent, only values in $\{|s_{\text{lcsqs}}|/2 + 1, \ldots, |s_i| - |s_{\text{lcsqs}}|/2 - 1\}$ for q_i can lead to better solutions. We therefore iterate the random sampling of each q_i until a value in this range is obtained.

5.2.2 RVNS&BS Approach

As an alternative to the RLS described above we consider a variable neighborhood search approach [133]. More precisely, we use a version of the VNS with no local search method included, known as *Reduced* VNS (RVNS).

For a current vector $q \in \mathbb{P}$, we define a move in the k-th neighborhood, $k = 1, \ldots, m$, by perturbing exactly k randomly chosen positions as above by adding a discretized normally distributed sampled random offset. Again, we take care not to choose meaningless small or larges values. We then evaluate q by the following the 3-step process. We first calculate $ub_q = 2 \cdot \text{UB}(S^{\mathbf{q}})$, and if $ub_{\mathbf{q}} \leq |s_{\text{lcsqs}}|$, q cannot yield an improved incumbent solution and q is discarded. Otherwise, we perform a fast evaluation of q by applying BNH which yields a solution $s = (\text{BNH}(S^{\mathbf{q}'}))^2$. If $|s| > \alpha \cdot |s_{\text{lcsqs}}|$, where $\alpha \in (0, 1)$ is a threshold parameter of the algorithm, we consider q promising and further execute BS on $S^{\mathbf{q}}$, yielding solution $s^{\text{bs}} = (\text{BS}(S^{\mathbf{q}'}, \beta))^2$. Again, the incumbent solution s_{lcsqs} is updated by any obtained better solution. If an improvement has been achieved, the RVNS&BS always continues with the first neighborhood, i.e. $k := 1$, otherwise with the next neighborhood, i.e. $k := k + 1$ until $k = m + 1$ in which case k is reset to 1. In order to improve the performance, we store all partitionings evaluated by BS, together with their evaluations, in a hash map and retrieve these values in case the corresponding partitionings are re-encountered.

Algorithm 23 RVNS&BS algorithm for the LCSqS

Input: an LCSqS instance, β: beam width, σ: standard deviation, α: a trash-hold of performing BS
Output: a feasible LCSqS solution
$\mathbf{q} \leftarrow \left(\lfloor \frac{|s_1|}{2} \rfloor, \ldots, \lfloor \frac{|s_m|}{2} \rfloor\right)$, k-exchange neighborhood structures $\{N_k\}_{k=1}^{m}$
$s \leftarrow \text{BS}(\mathbf{q}, \beta)$ //store partitionings for which BS has been performed
$C[\mathbf{q}] \leftarrow 2 \cdot |s|$
$s_{\text{lcsqs}} \leftarrow s \cdot s$
while $t_{\max}$ is not exceeded **do**
    $k \leftarrow 1$
    while $k \leq m$ **do**
        $\mathbf{q}' \leftarrow$ perturb k positions randomly from $\mathbf{q}$ by a random offset σ //shake
        $ub_{\mathbf{q}'} \leftarrow \text{UB}(\mathbf{q}')$
        if $ub_{\mathbf{q}'} > |s_{\text{lcsqs}}|$ **then**
            $s \leftarrow \text{BNH}(\mathbf{q}')$
            if $2|s| > \alpha \cdot |s_{\text{lcsqs}}|$ **then**
                if $\mathbf{q} \notin C$ **then** //evaluate node
                    $s \leftarrow \text{BS}(\mathbf{q}', \beta)$
                    $C[\mathbf{q}'] \leftarrow 2 \cdot |s|$
                else
                    $f_{\mathbf{q}'} \leftarrow C[\mathbf{q}']$ // BS already performed for q'
                end if
            end if
            if $2|s| > |s_{\text{lcsqs}}|$ **then**
                $s_{\text{lcsqs}} \leftarrow s \cdot s$
                $\mathbf{q} \leftarrow \mathbf{q}'$
                $k \leftarrow 1$
            else
                $k \leftarrow k + 1$
            end if
        end if
    end while
end while
return s_{lcsqs}

5.3 Computational Experiments

The algorithms are implemented in C++ and all experiments are performed on a single core of an Intel Xeon E5-2640 with 2.40GHz and 8 GB of memory.

We used the set of benchmark instances provided in [18] for the LCS problem. This instance set consists of ten randomly generated instances for each combination of the number of input strings $m \in \{10, 50, 100, 150, 200\}$, the length of the input strings

Table 5.1: Selected results for $n = 100$.

m	$\|\Sigma\|$	**RVNS & BS**		**RLS & BS**		**RVNS+Dive**		**RLS**	
		$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$
10	4	**27.08**	67.71	26.54	44.94	26.96	51.20	26.42	34.40
10	12	8.24	17.73	8.04	13.59	**8.28**	19.27	7.70	21.92
10	20	3.84	0.02	**4.00**	1.66	3.96	0.05	**4.00**	4.44
50	4	**18.54**	10.53	18.16	24.12	**18.54**	45.81	18.04	19.43
50	12	**3.90**	15.34	3.82	11.01	3.88	5.00	3.80	28.98
50	20	0.20	0.01	**0.46**	4.77	0.20	0.00	0.40	0.01
100	4	**16.14**	16.72	16.02	17.95	**16.14**	8.44	16.00	28.95
100	12	1.60	0.02	**2.00**	0.10	1.64	6.19	**2.00**	0.39
100	20	**0.00**	0.02	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
150	4	14.34	43.40	14.40	38.22	**15.06**	85.49	14.28	38.18
150	12	0.40	0.03	**2.00**	10.47	0.40	0.00	1.94	25.31
150	20	**0.00**	0.03	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
200	4	**14.00**	4.88	**14.00**	8.68	**14.00**	1.36	13.94	24.12
200	12	0.00	0.03	**1.58**	33.89	0.00	0.00	1.34	60.30
200	20	**0.00**	0.05	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00

Table 5.2: Results for $n = 500$.

m	$\|\Sigma\|$	**RVNS & BS**		**RLS & BS**		**RVNS+Dive**		**RLS**	
		$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$
10	4	**156.58**	140.99	156.14	146.08	149.78	160.69	149.24	110.09
10	12	**58.64**	116.44	58.16	126.15	56.16	112.91	56.00	72.37
10	20	**35.78**	85.44	35.12	48.07	34.54	50.42	34.56	71.16
50	4	**124.30**	52.66	124.12	160.39	120.32	86.33	120.12	109.37
50	12	**38.66**	60.86	38.38	60.35	38.04	92.53	38.02	44.35
50	20	**21.14**	78.62	20.52	34.12	20.64	66.00	20.68	61.19
100	4	**115.94**	68.28	115.64	116.64	112.34	84.31	111.86	98.60
100	12	**34.00**	56.69	33.82	124.41	32.54	46.40	33.14	97.40
100	20	**18.00**	25.46	17.90	86.93	17.38	105.82	17.52	44.17
150	4	**112.00**	40.02	111.84	105.86	108.78	119.46	107.60	92.53
150	12	**31.98**	104.70	31.54	105.95	30.42	52.22	30.62	21.20
150	20	**16.00**	5.24	**16.00**	8.46	**16.00**	31.58	**16.00**	16.52
200	4	**109.86**	152.67	108.78	102.03	106.22	66.72	104.94	90.66
200	12	**30.00**	24.29	**30.00**	47.88	29.26	146.88	29.66	93.29
200	20	**14.48**	54.76	14.26	35.50	14.04	3.51	14.10	12.82

$n \in \{100, 500, 1000\}$, and the alphabet size $|\Sigma| \in \{4, 12, 20\}$. This makes a total of 450 problem instances. We apply each algorithm ten times to each instance, with a time limit of 600 CPU seconds.

From preliminary experiments we noticed that the behavior of our algorithms mostly depends on the length n of the input strings. Therefore, we tuned the algorithms

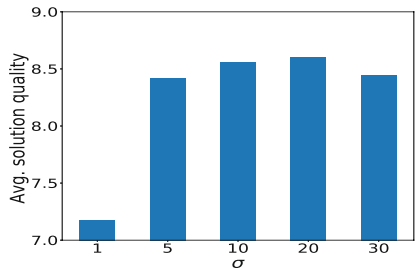
Table 5.3: Results for $n = 1000$.

m	$\|\Sigma\|$	**RVNS & BS**		**RLS & BS**		**RVNS+Dive**		**RLS**	
		$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$	$\overline{\|s\|}$	$\overline{t_{\text{best}}}[s]$
10	4	**321.14**	206.16	320.94	193.50	304.48	186.65	304.34	161.08
10	12	**123.88**	118.87	123.68	155.13	117.86	151.51	117.88	134.88
10	20	**76.84**	126.40	76.66	141.40	73.80	118.86	73.72	76.98
50	4	**261.52**	127.81	260.82	135.14	252.94	131.88	249.84	153.18
50	12	**86.02**	113.80	85.92	146.46	83.34	132.11	83.98	100.37
50	20	**49.78**	116.89	49.76	188.74	48.12	54.48	48.70	74.04
100	4	**247.20**	197.87	246.24	147.54	240.24	109.11	234.36	145.40
100	12	77.38	211.06	**77.50**	209.02	75.44	137.65	75.28	118.57
100	20	43.34	161.50	**43.40**	201.43	42.02	17.65	42.28	30.80
150	4	**240.00**	167.73	239.12	178.24	234.02	127.31	227.02	127.81
150	12	**73.76**	181.25	73.42	251.89	71.76	121.40	71.36	120.95
150	20	**40.02**	114.79	40.00	140.61	39.88	121.36	39.96	59.40
200	4	**235.50**	213.72	234.44	202.34	230.10	135.37	222.66	145.99
200	12	**70.76**	211.03	70.28	243.41	69.10	144.78	68.30	44.32
200	20	38.04	132.86	**38.12**	165.15	38.00	59.74	38.02	24.07

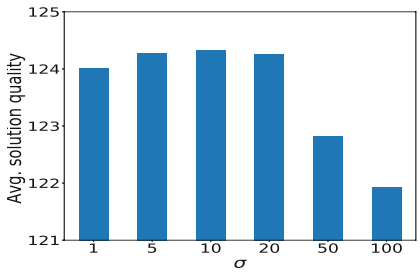
separately for instances with string length 100, 500, and 1000. The *irace* tool [127] was used for this purpose. For RLS, we obtained $destr = 0.2$ and $\sigma = 5$ (for $n = 100$), $destr = 0.3$ and $\sigma = 10$ (for $n = 500$), and $destr = 0.3$ and $\sigma = 20$ (for $n = 1000$). For RVNS&BS, we obtained $\alpha = 0.9$ and $\beta = 100$ (for $n = 100$), $\alpha = 0.9$ and $\beta = 200$ (for $n = 500$), and $\alpha = 0.9$ and $\beta = 200$ (for $n = 1000$). For σ of the RVNS&BS, *irace* yielded the same values as for the RLS. Moreover, EX was preferred over UB as a guidance for BS.

We additionally include here results for RVNS&Dive, which is RVNS&BS with $\beta = 1$. In this case, BS reduces to a simple greedy heuristic (or dive). This was done for checking the impact of a higher beam size. Moreover, RLS&BS refers to a version of RLS in which BNH is replaced by BS with the same beam size as in RVNS&BS. Selected results are shown in Tables 5.1–5.3. For each of the algorithms, we present the avg. solution quality and the avg. median time when the best solution was found. From the results we conclude the following:

- RVNS&BS produces solutions of significantly better quality than the other algorithms on harder instances.
- The rather high beam size is useful for finding approximate solutions of higher quality.
- Concerning the computation time for harder instances, the times of the RVNS&BS are usually higher than those of the RLS. It seems harder for BNH to help to improve solution quality in later stages of the RLS than for the BS in RVNS&BS.

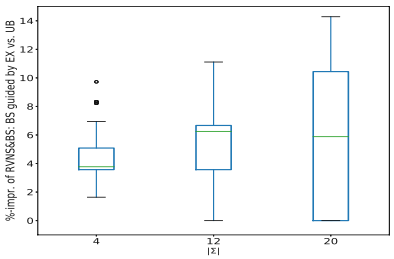


Instance: $m = 10, n = 100, |\Sigma| = 12$.

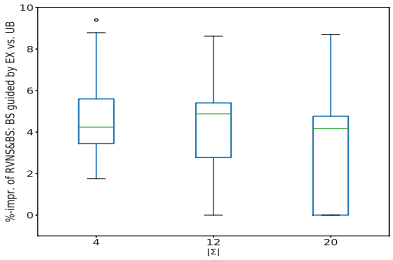


Instance: $m = 50, n = 500, |\Sigma| = 4$.

Figure 5.1: The impact of parameter σ on the solution quality of RVNS&BS.



$n = 500$.



$n = 1000$.

Figure 5.2: Improvements of solution quality when using EX instead of UB for guiding BS in RVNS&BS

- From Figure 5.1 we can see that, for smaller instances with larger alphabet sizes, stronger jumps in the search space are in essence preferred. This is because a small number of feasible solutions is distributed over the search space, and to find them it is convenient to allow large, random jumps in the search. When n is larger, choosing to do larger jumps in the space is not a good option (see the bar plot on the right). This can be explained by the fact that already the vector q that is defined by the middle of all input strings (which are generated uniformly at random) yields a promising solution, and many promising partitions are clustered around this vector. By allowing larger jumps, we move further away from this middle vector quickly, which yields usually in weaker solutions.

Figure 5.2 provides box plots showing the relative differences between the quality of the solutions obtained by RVNS&BS using EX and RVNS&BS using UB ($\beta = 200$). The figure shows a clear advantage of several percent when using EX over the classical upper bound UB as search guidance.

5.4 Conclusions

This article provides the first heuristic approaches to solve the LCSqS problem for an arbitrary set of input strings. We applied a reduction of the LCSqS problem to a series of standard LCS problems by introducing a partitioning of the input strings as a first-level decision. Our RVNS framework explores the space of partitionings, which are then tackled by BNH and, if promising, by BS. Hereby, BS is guided by a heuristic which approximates the expected length of an LCS. Overall, RVNS&BS yields significantly better solutions that the also proposed, simpler RLS.

CHAPTER 6

Application of Maximum Clique Solvers to Solve LCS Problems

This chapter describes a relation between the well-known Maximal Clique (MC) problem and the longest common subsequence problem and the variants thereof.

In the course of this work we have published

- an article in the *Computers & Operations Research* journal (IF=3.002) [17].

In this chapter, we present an unified approach for solving string problems by transforming an instance of the LCS problem (and three variants thereof) into an MC problem instance, called a conflict graph. Solving the maximum independent set problem in this conflict graph corresponds to solving the longest common subsequence on the original LCS instance. We actually solve MC problem on the complement of the conflict graphs by means of the best known exact and heuristic MC solvers from literature. A way of reducing the conflict graphs is also presented and its effectiveness is demonstrated on various benchmark sets from the literature. We emphasize that the core ides of work are introduced by C. Blum. The main contributions of the author of this thesis were related to finding an efficient and effective reduction of the size of conflict graphs, the implementation of the reduction as well as testing and demonstrating the benefits of the proposed reduction technique.

6.1 Introduction

It has already been mentioned that the classicl LCS has a variety of applications. Some real-life applications require additional constraints, motivating the studies of the problems which are variations of the basic LCS problem. These include the repetition-free longest

common subsequence (RFLCS) problem [2], the constrained longest common subsequence (CLCS) problem [162], and the generalized constrained longest common subsequence (GCLCS) problem [32]. Others are mentioned in survey papers such as [22] in Chapter 4 the LCPS problem has been considered and in Chapter 5 the LCSqS problem has been studied. Henceforth, in this chapter we refer to the variants of the classical LCS problem, in general, as LCS-type problems.

Although LCS-type problems are presented in the literature for almost fifty years, their computational difficulties cause the research is still active on these topics. In this chapter we propose the new approach of transforming instances of the classical LCS-type problems into the instances of the *maximal clique* (MC) problem [21]. The core idea of the transformation is to construct, for each LCS instance, a corresponding *conflict graph* [119]. Hereby, an independent set in the conflict graph corresponds to a common subsequence concerning the original LCS instance. Moreover, a maximum independent set (MIS) in the conflict graphs corresponds to a longest common subsequence of the LCS instance. Note that finding a MIS in the conflict graph is equivalent to finding a largest clique on the complement graph of the conflict graph. Therefore, an LCS problem instance can be solved by finding a largest clique in the complement of the conflict graph.

The advantages of this approach are twofold. First, because of a steady improvement of the solvers for the MC problem, we have high-performing algorithms at our disposal that may make solving an MC problem on the complement of the conflict graph faster than solving the original LCS problem with known exact algorithms. Second, we will show that our transformation–in addition to the classical LCS problem–can be used to tackle other LCS-type problems from the literature, thus providing a unified approach for different LCS-type problems.

In the rest of this chapter, Section 6.2 provides the details of the transformations required to build the conflict graphs of the LCS-type problems considered in this work. In Section 6.3, a way to reduce the size of conflict graphs is proposed while in Section 6.4 experimental studies on this approach are reported. Section 6.5 highlights the conclusions on these studies and outlines some directions for future work.

6.2 Considered problems and transformations

This section start by defining a way to transform an instance of the classical LCS problem into a conflict graph in which a maximum independent set corresponds to a longest common subsequence of the original problem instance. Henceforth, an LCS problem instance is described by a pair (S, Σ) in which $S = \{s_1, \ldots, s_m\}$ is a set of input strings over the finite alphabet Σ. Given such an instance, we construct an undirected multi-layered graph $G = (V, E)$ whose vertex set V is partitioned into sets $\{V_1, \ldots, V_m\}$. Each V_i is called a *layer* and consists of $|s_i|$ vertices. Note that each layer represents exactly one input string and each vertex of the layer represents a position in the string. More specifically, $V_i = \{v_{i,1}, \ldots, v_{i,|s_i|}\}$, where vertex $v_{i,j}$ represents the j-th position of input string s_i.

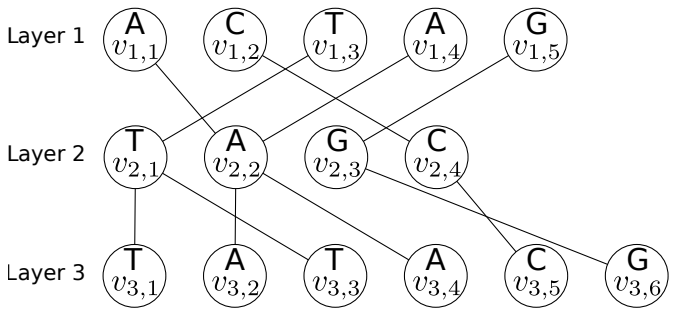


Figure 6.1: The undirected multi-layered graph G obtained from the LCS instance $(S = \{s_1 = \texttt{ACTAG}, s_2 = \texttt{TAGC}, s_3 = \texttt{ATACG}\}, \Sigma = \{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\})$.

We also partition the edge set E of the multi-layered graph G into sets $\{E_1, \ldots, E_{m-1}\}$, where E_i is the set of edges between layers V_i and V_{i+1}. Set E_i contains an edge connecting vertices $v_{i,j}$ and $v_{i+1,k}$ if and only if $s_i[j] = s_{i+1}[k]$, i.e., if the letter at position j of input string s_i is equal to the letter at position k of input string s_{i+1}. Figure 6.1 shows an example of this graph construction for three strings over an alphabet of size four.

Any sequence $p = (v_{1,j_1}, v_{2,j_2}, \ldots, v_{m,j_m})$ of m vertices with the i-th vertex of p being from the i-th layer of G is called a *complete path* in G iff fulfills the condition that the letters at the positions of the input strings corresponding to the $m-1$ vertices are all the same, that is, $s_1[j_1] = s_2[j_2] = \ldots = s_n[j_m]$. Note that if p fulfills this condition, there is—by definition—an edge between each pair of consecutive vertices of p. Given a complete path $p = (v_{1,j_1}, v_{2,j_2}, \ldots, v_{m,j_m})$, the common letter at positions $j_1, \ldots, j_m$ of the m input strings is also called the *letter of* p. We denote it by $\ell(p)$.

Two complete paths p and q, with $p = (v_{1,j_1}, v_{2,j_2}, \ldots, v_{m,j_m})$ and $q = (v_{1,k_1}, v_{2,k_2}, \ldots, v_{m,k_m})$, are said to *cross* if and only if there is at least one index $l \in \{1, \ldots, m\}$ such that $j_l \leq k_l$ and at least one index $r \in \{1, \ldots m\}$, $r \neq l$, such that $j_r \geq k_r$. To make the concept of crossing paths clearer, refer to Figure 6.2 which shows two examples based on the instance depicted in Figure 6.1. In Figure 6.2 (left), the solid and dashed paths are crossing because they contain crossing edges between layers 1 and 2. In Figure 6.2 (right), they cross because they contain a common vertex in layer 2.

Given these notations, the classical LCS problem can be transformed into the maximum independent set (MIS) problem as follows. First, note that solving the classical LCS problem relates to finding the largest set of non-crossing complete paths in the respective multi-layered graph G. Based on graph G we create the conflict graph $G^c = (V^c, E^c)$ where each vertex corresponds to the complete path of the graph G and an edge between two vertices exists iff the corresponding path cross. Then, solving the LCS problem is equivalent to solving the MIS problem in G^c which, in turn, is equivalent to solving the MC problem in the complement $\overline{G^c}$ of graph G^c.

In the rest of this section we consider three LCS-type problems and show how analogous transformations allow us to reduce each problem to an MC problem on the complement

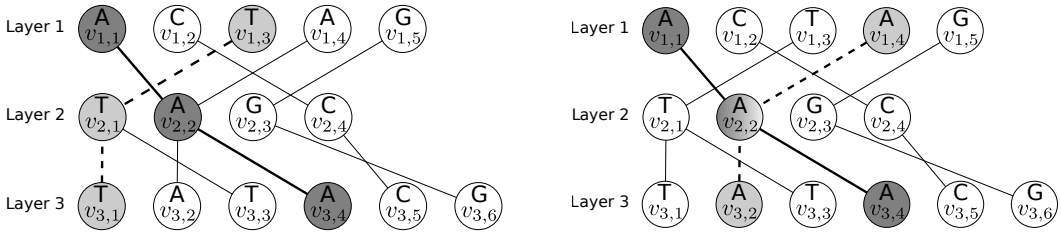


Figure 6.2: Two examples of complete paths that cross, based on the LCS instance from 6.1. (a) Paths $p = (v_{1,1}, v_{2,2}, v_{3,4})$ and $q = (v_{1,3}, v_{2,1}, v_{3,1})$ cross because their corresponding edges between layers 1 and 2 cross. (b) Paths $p = (v_{1,1}, v_{2,2}, v_{3,4})$ and $q = (v_{1,4}, v_{2,2}, v_{3,2})$ cross because they both include vertex $v_{2,2}$ from the second layer.

of conflict graphs.

6.2.1 Repetition-Free Longest Common Subsequence Problem

The repetition-free longest common subsequence (RFLCS) problem [2] is an LCS variant in which valid solutions are further constrained to contain each possible letter at most once. It was introduced as a comparison measure for sequences of different biological origin. In the related literature, this problem is generally considered for the case $m = 2$, that is, for two input strings. Note that even for $m = 2$ the problem is $\mathcal{APX}$–hard (which implies it is $\mathcal{NP}$-hard), as shown by [2]. It is still an open question whether the RFLCS admits a constant factor approximation. A fixed parameter tractable (FTP) algorithm was presented in [12]. Blum and Blesa [14] proposed the current best specialized algorithm for this problem: a construct, merge, solve and adapt (CMSA) approach in which the authors initialise the reduced sub-instance by beam search. In [14], the authors show how their algorithm outperforms other metaheuristics and the application of Cplex to an ILP model of the problem.

To generate the conflict graph for the RFLCS problem, we first build the multi-layered graph G concerning the two input strings, just like in the case of the classical LCS problem. Note that, due to the two input strings, G will have two layers. Two complete paths p and q of G are in conflict if they fulfill at least one of the following two conditions:

1. p and q cross each other.
2. p and q have the same letter: $\ell(p) = \ell(q)$.

Note that the second condition ensures that no letter appears more than once in a solution.

6.2.2 Longest Arc-Preserving Common Subsequence Problem

The second considered LCS variant is known as the longest arc-preserving common subsequence (LAPCS) problem [62]. As in the case of the RFLCS problem, the LAPCS problem is studied for two input strings/sequences in the literature. Note that, in the case of the LAPCS problem, the input strings are arc-annotated. An *arc annotation* of a string s is a pair of positions in s, say (i_1, i_2) with $i_1, i_2 \in \{1, \ldots, |s|\}$ and $i_1 < i_2$. An arc-annotated sequence is a pair (s, P_s) where s is a string over some finite alphabet Σ and P_s is the set of arc annotations of s. Given two arc-annotated sequences (s_1, P_1) and (s_2, P_2), the two-layered multi-graph G is constructed for s_1 and s_2 in the same way as shown before. Any set S of non-crossing complete paths in P is a feasible LAPCS solution if the following additional condition is fulfilled. For any pair $p = (v_{1,j_1}, v_{2,j_2}) \neq q = (v_{1,k_1}, v_{2,k_2})$ of non-crossing complete paths from S with $j_1 < k_1$ it must hold that if (j_1, k_1) is an arc annotation of s_1—that is, if $(j_1, k_1) \in P_1$—then (j_2, k_2) must be an arc annotation of s_2, and vice versa. The optimization goal is to find a largest feasible solution S.

Arc-annotated sequences are useful for the structural comparison of RNA sequences. Figure 6.3 shows an example of an arc-annotated RNA sequence in which the arc annotations are indicated as solid lines linking the nucleobases ACGU. [61, 62] introduced the LACPS problem and showed that it is $\mathcal{NP}$-hard already for two strings. Researchers have also focused on special cases of the problem and developed polynomial time algorithms, approximation algorithms and fixed parameter tractability results for some of these cases [62, 97, 75, 72].

Blum and Blesa [15] proposed the best specialized algorithms for the LAPCS. Depending on the problem instance characteristics, the state-of-the-art algorithm is either a heuristic based on problem reduction, or an iterative probabilistic algorithm, both of which solve reduced ILP models. The authors compared these algorithms with the application of Cplex to solve the MIS problem in the corresponding conflict graphs.

To generate the conflict graph for a LAPCS problem instance consisting of (s_1, P_1) and (s_2, P_2), we first construct the two-layered multi-graph G based on s_1 and s_2, as done in the classical LCS problem case. Two complete paths $p = (v_{1,j_1}, v_{2,j_2}) \neq q = (v_{1,k_1}, v_{2,k_2})$ with $j_1 < k_1$ are in conflict if and only if they fulfill at least one of the following two conditions:

1. p and q cross each other.
2. p and q violate the arc preservation constraints. This happens when either $(j_1, k_1) \in P_1$ and $(j_2, k_2) \notin P_2$, or $(j_2, k_2) \in P_2$ and $(j_1, k_1) \notin P_1$.

Figure 6.4 shows an example LAPCS instance. The solution depicted with dashed lines is infeasible because it matches $v_{1,2}$ and $v_{1,4}$ in s_1 with, respectively, $v_{2,4}$ and $v_{2,5}$ in s_2. An arc annotation links the positions in s_1 but not in s_2, thus violating condition 2 above. The solution depicted with straight, solid lines, instead, is feasible.

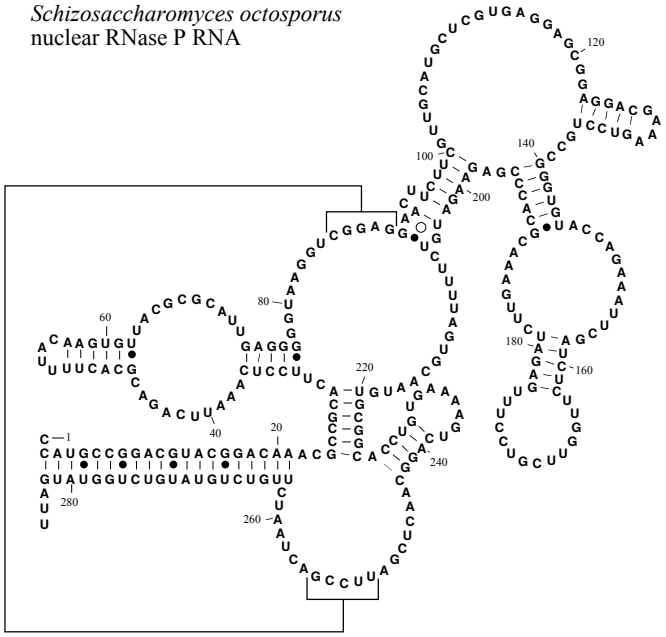


Figure 6.3: Example of an arc-annotated sequence (RNA of *Schizosaccharomyces octosporus*). The connections between different positions of the RNA sequence, indicated by short lines, are the members of the arc annotation set. Note that this graphic was obtained from the RNase P Database [26].

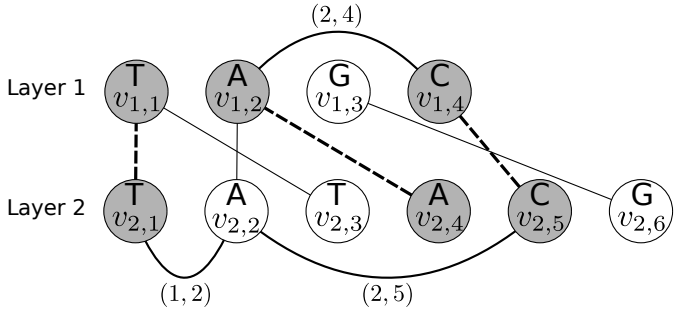


Figure 6.4: This example shows the undirected multi-layered graph G obtained from the LAPCS instance consisting of $(s_1 = \texttt{TAGC}, P_1 = \{(2,4)\})$ and $(s_2 = \texttt{TATACG}, P_2 = \{(1,2),(2,5)\})$. The solution in dashed lines $\{p = (v_{1,1}, v_{2,1}), q = (v_{1,2}, v_{2,4}), r = (v_{1,4}, v_{2,5})\}$ is not valid because arc $(2,4) \in P_1$ connects two chosen positions in s_1, while the corresponding chosen positions in s_2 — that is, positions 4 and 5 — are not connected by an arc from P_2. The solution in straight, solid lines is, in contrast, feasible.

6.2.3 Longest Common Palindromic Subsequence Problem

Finally, we also show how to transform the so-called longest common palindromic subsequence (LCPS) problem [38] into instances of the MC problem instances. Even though we will show later that this transformation is not practical for benchmark instances from the literature, because as the resulting conflict graphs are too large, we present the transformation, since it might help in order to derive transformations for other LCS-type problems in subsequent works.

The LCPS problem is an LCS variant in which we look for a longest common subsequence s^* of m input strings such that s^* is also a palindrome, see Chapter 4. Remember that a string is a *palindrome* if it coincides with its reverse. For example, KAYAK is a palindrome. In many studies [38, 80, 89] specialized exact algorithms for the LCPS problem on two input strings (2–LCPS) are presented. Moreover, [89] proved—based on results from [1]—that the theoretical lower bound on solving the 2–LCPS is $O(n^4)$. [54, 57] (see Section 4.5) presented the first works on the instances with $m > 2$, introducing two A*-based hybrid anytime algorithms, that is, exact algorithms that return a feasible solution of reasonable quality whenever they are terminated [165].

After generating the layered multi-graph G for the m input strings, in the same way as in the cases outlined before, the conflict graph is built as follows. The set of vertices V^c of the conflict graph G^c consists of two disjoint subsets of vertices: V_{single} and V_{pairs}. More specifically, V_{single} contains a vertex v_p for each complete path $p \in P$, and V_{pairs} contains a vertex $v_{p,q}$ for each pair of complete paths $p \neq q$ with $\ell(p) = \ell(q)$ that do not cross each other. Notice that in the previous cases—that is, the classical LCS problem, the RFLCS problem, and the LAPCS problem—the number of vertices in the conflict graph was equal to the number of complete paths in the multi-layered graph G, say z. In contrast, the number of vertices in the conflict graph of the LCPS problem is of the order $\mathcal{O}(z + z^2)$. Finally, we define the edges of the conflict graph by the following conflict relations:

1. Conflicts between vertices from V_{single}: these vertices are all in conflict with each other. This is because the vertices from V_{single} model the possibility to have a singleton letter in the middle of a solution. For example, KAYAK has Y as a singleton letter in the middle. In contrast, KAAK for example, has no singleton letter in the middle. As a solution can have at most one singleton letter in the middle, all vertices from V_{single} are in conflict with each other. As a consequence, all other vertices that form part of a solution are from V_{pairs}. In the case of KAYAK, for example, there would be two such vertices: one representing the two K's and one for the two A's.

2. Conflicts between vertices from V_{pairs}: to describe a conflict between two such vertices, it is actually easier to state when they are *not* in conflict with each other.

Consider two vertices $v_{p,q}, v_{p',q'} \in V_{\text{pairs}}$, with

$$p = (v_{1,j_1}, \ldots, v_{m,j_m})$$
$$q = (v_{1,k_1}, \ldots, v_{m,k_m})$$
$$p' = (v_{1,j'_1}, \ldots, v_{m,j'_m})$$
$$q' = (v_{1,k'_1}, \ldots, v_{m,k'_m})$$

and assume wlog that $j_1 < k_1$ and that $j'_1 < k'_1$. Then $v_{p,q}$ and $v_{p',q'}$ are *not* in conflict if either $j_i < j'_i < k'_i < k_i$ for all $i = 1, \ldots, m$, or $j'_i < j_i < k_i < k'_i$ for all $i = 1, \ldots, m$.

3. Conflicts between vertices from V_{single} and vertices from V_{pairs}: again, we state when there is *no* conflict between two such vertices. Consider vertex $v_{p'} \in V_{\text{single}}$ and vertex $v_{p,q} \in V_{\text{pairs}}$, with

$$p = (v_{1,j_1}, \ldots, v_{m,j_m})$$
$$q = (v_{1,k_1}, \ldots, v_{m,k_m})$$
$$p' = (v_{1,j'_1}, \ldots, v_{m,j'_m})$$

and assume w.l.o.g. that $j_1 < k_1$. Then $v_{p''}$ and $v_{p,p'}$ are not in conflict if $j_i < j'_i < k_i$ for all $i = 1, \ldots, m$.

Notice that all vertices from V_{pairs} have weight 2 and, if chosen in the final clique, they will contribute for two letters in the respective solution.

Figure 6.5 shows the multi-layerd graph for input strings `TAGCAT` and `TATACG`. Complete paths are shown by lines and, in particular, we use dashed and dotted lines to highlight relevant paths concerning letters `T` and `A`. Note how the rightmost highlighted paths for `T` and `A` are crossing. Therefore, the potential solution `TAAT` cannot be constructed. This string is only a substring of the first input string, but not of the second one. The optimal solution in this example is, in fact, `TAT`.

6.3 Conflict graph reduction

The size of the conflict graphs (in terms of the number of vertices) mainly depends on the length and the number of input strings. The sizes of the conflict graphs can be expressed as follows: $\mathcal{O}(n^m)$ in the case of the classical LCS problem, $\mathcal{O}(n^2)$ in the case of the RFLCS and LAPCS problems, and $\mathcal{O}(n^m + n^{2m})$ in the case of the LCPS problem. In fact, during preliminary experiments, we realized that the conflict graphs are too large, even for rather small problem instances from the literature, in the cases of the classical LCS problem and the LCPS problem. Therefore, we henceforth focus exclusively on RFLCS and LAPCS problems. However, even for these two problems, the conflict graphs are very large when large-scale problem instances are concerned. Therefore, we decided

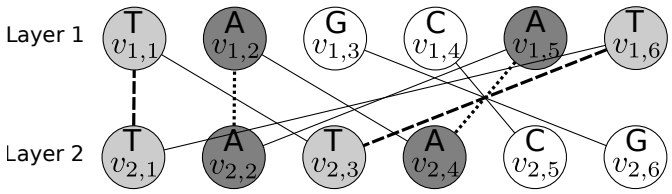


Figure 6.5: The multi-layered graph G obtained from the LCPS instance on the two input strings $s_1 = \texttt{TAGCAT}$ and $s_2 = \texttt{TATACG}$. This graph contains 10 complete paths, corresponding to the 10 vertices of the conflict graph (V_{single}). Two pairs of non-crossing complete paths have the same letters: the first pair (with letter T) is indicated in light gray and dashed lines, the second one (with letter A) is indicated in dark gray and dotted lines.

to investigate techniques for reducing the size of the conflict graphs. Note that there are potentially two strategies for reducing the size of a given conflict graph G^c: (i) making use of problem-specific information relative to the respective LCS-type problems, and (ii) analyzing and reducing G^c from the MC problem. However, the latter strategy has proven ineffective in preliminary computational experiments. This is because solver LMC (the state-of-the-art exact MC problem solver that we used [94, 121]) already implements powerful graph reduction procedures which were not able to reduce G^c. Therefore we make use of LCS-specific information to reduce the conflict graph in a novel way.

The main idea for our reduction of the conflict graphs is based on making use of a high-quality primal (lower) bound value lb for the tackled problem, that is, the value of a high-quality solution. The value of the best-known solution from the literature can be taken for this purpose, for example. Before we proceed, the following notation is required: given a string t and two indices $l, r \in \{1, \ldots, |t|\}$ with $l \leq r$, $t[l, r]$ denotes the substring of t starting at position l and ending at position r. Now, based on the primal bound lb, it can be decided for every complete path $p = v_{1,j_1}, \ldots, v_{m,j_m}$ of the multi-layered graph, if the corresponding vertex v_p can be removed from the conflict graph G^c without losing an optimal solution.[1] This is done as follows. First, note that the complete path under consideration splits each input string s_i into two parts: $s_i[1, j_i - 1]$ (the left-hand side) and $s_i[j_i + 1, |s_i|]$ (the right-hand side). Henceforth we denote the set of left-hand sides corresponding to a complete path p by S_p^{L}, and the set of right-hand sides by S_p^{R}. More formally:

$$S_p^{\text{L}} = \{s_i[1, j_i - 1] \mid i = 1, \ldots, m\}$$
$$S_p^{\text{R}} = \{s_i[j_i + 1, |s_i|] \mid i = 1, \ldots, m\}$$

[1] Note that the conflict graph reduction will be described for a general case of n input strings, even though we only have two input strings in the cases of the RFLCS and LAPCS problems.

Note that both S_p^{L} and S_p^{R} are subinstances of the original problem instance. Therefore, any upper bound function UB() known for the problem (RFLCS, respectively LAPCS) can be used for (over)-estimating the quality of the length of an optimal solution in S_p^{L} and S_p^{R}. Given such an upper bound function UB(), vertex v_p and all corresponding edges can be deleted from the conflict graph G^{c} iff

$$\mathrm{UB}(S_p^{\mathrm{L}}) + 1 + \mathrm{UB}(S_p^{\mathrm{R}}) < lb\,. \tag{6.1}$$

For the following discussion, bear in mind that any upper bound for the classical LCS problem is also an upper bound for the RFLCS and LAPCS problems. This is because these two problems correspond to classical LCS problems with additional constraints. In other words, the set of valid solutions of an RFLCS problem instance, respectively a LAPCS problem instance, is a subset of the set of valid solutions of the instance if solved as a classical LCS problem. Therefore, upper bound functions developed for the classical LCS problem are candidates to be used for UB() in 6.1.

Blum et al. [16], for example, introduced an upper bound function henceforth labelled $\mathrm{UB}_1^{\mathrm{LCS}}()$ for the classical LCS problem (see Section 3.3.1). Let $\delta(a, S)$ for $a \in \Sigma$ evaluate to one, if letter a appears at least once in each input string from S, and otherwise to zero. As each letter from Σ can mostly appear once in a valid RFLCS solution, $\mathrm{UB}_1^{\mathrm{LCS}}()$ from above reduces to the following upper bound function in the context of the RFLCS problem:

$$\mathrm{UB}_1^{\mathrm{RFLCS}}(S) = \sum_{a \in \Sigma} \delta(a, S)$$

Finally, when used for our purposes—that is, for obtaining an upper bound for (sub-)instances S_p^{L} and S_p^{R} in Section 6.3 in the context of an RFLCS instance—we can even exclude letter $l(p)$ (the letter of path p) from the sum. This results in:

$$\mathrm{UB}_1^{\mathrm{RFLCS}}(S, p) = \sum_{a \in \Sigma \setminus \{l(p)\}} \delta(a, S)\,.$$

For the classical LCS problem we are aware of another upper bound, labelled $\mathrm{UB}_2()$, which is based on dynamic programming (DP). It is obtained in $\mathcal{O}(m)$ incorporating appropriate preprocessing steps, see Section 3.3.1. In particular, note that in the context of the RFLCS and LAPCS problems, the preprocessing is done in $\mathcal{O}(n^2)$ time.

In summary, for the conflict graph reduction in the context of the RFLCS problem, UB() is defined as $\min\{\mathrm{UB}_1^{\mathrm{RFLCS}}(), \mathrm{UB}_2()\}$; and UB_2 in the context of the LAPCS problem, because $\mathrm{UB}_2() < \mathrm{UB}_1^{\mathrm{LCS}}()$ in all cases.

6.4 Experimental evaluation

The computational experiments aim to compare two strategies to solve LCS problems: (*i*) their direct solution using a specialized state-of-the-art algorithm, and (*ii*) their

transformation to the MIS, respectively the MC, problems and the subsequent solution by CPLEX[2] (in case of the MIS problem) or by the following MC solvers:

- LMC. This exact MC solver was introduced by [94, 121]. It is currently one of the best exact solvers available for the MC problem. It combines an aggressive preprocessing of the graph with a MaxSAT solver [122] in a branch-and-bound scheme.
- LSCC-BMS. This is one of the best-performing heuristic algorithms for the MC problem. [175] introduced this local-search-based algorithm, whose main strengths are a configuration checking procedure that reduces the probability of cycling during local search, and a low-complexity vertex swap neighborhood which is fast even on massive graphs[3].

Note that both CPLEX and LSCC-BMS were executed on a cluster of 12-core Intel Xeon 5670 CPUs at 2.9GHz and at least 40GB of RAM. LMC was executed on a cluster with 8-core Intel Xeon E5-2680 CPUs at 2.4GHz and with 128 GB of memory. In both cases, the memory consumption of each process was limited to 16 GB.

RFLCS benchmark instances Two sets of problem instances can be found in the related literature. The first set, henceforth denoted by RFLCS-SET1, consists of 30 randomly generated problem instances for each combination of the input sequence length $n \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$ and the alphabet size $|\Sigma| \in \{\frac{n}{8}, \frac{n}{4}, \frac{3n}{8}, \frac{n}{2}, \frac{5n}{8}, \frac{3n}{4}, \frac{7n}{8}\}$. This results in a total of 1680 instances. The second set, henceforth by RFLCS-SET2, consists of 30 randomly generated instances for each combination of the alphabet size $|\Sigma| \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and the maximal repetition of each letter, reps $\in \{3, 4, 5, 6, 7, 8\}$. In total, set RFLCS-SET2 contains 1440 instances.

LAPCS benchmark instances The recent literature on the LAPCS problem considers both artificial instances (benchmark set LAPCS-ARTI) and real RNA instances (benchmark set LAPCS-REAL). Each artificial instance consists of two randomly generated RNA strings of length $n \in \{100, 200, \ldots, 900, 1000\}$. Moreover, each input string has $n_{\text{arcs}} \in \{\frac{n}{10}, \frac{n}{5}, \frac{n}{2}\}$ randomly generated unique arc annotations. Set LAPCS-ARTI consists of 30 instances for each combination of n and n_{arcs}, which makes a total of 900 problem instances. Set LAPCS-REAL consists of 10 problem instances that are composed of arc-annotated RNA sequences downloaded from the RNase P Database [26]. Note that the alphabet size in all cases is equal to four. Table 6.1 summaries the characteristics of these instances.

[2]IBM ILOG CPLEX is an optimization software package that includes state-of-the-art exact techniques for solving integer linear programming models, among others. It is available for free for academic purposes. For more information, we refer the interested reader to `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`. In this work, we made use of version 12.7.

[3]We downloaded the code of LSCC-BMS from `http://ai.nenu.edu.cn/wangyy/Yiyuandata/LocalSearchforMWCP.htm` on April 29, 2019.

Instance	First String			Second string		
	RNA	n	n_{arcs}	RNA	n	n_{arcs}
Real_1	*Allochromatium vinosum*	369	119	*Haemophilus influenza*	377	124
Real_2	*Bacteroides thetaiotaomicron*	361	121	*Porphyromonas gingivalis*	398	131
Real_3	*Halococcus morrhuae*	475	154	*Haloferax volcanii*	433	142
Real_4	*Klebsiella pneumoniae*	383	127	*Escherichia coli*	377	124
Real_5	*Methanococcus jannaschii*	252	75	*Archaeoglobus fulgidus*	229	67
Real_6	*Methanosarcina barkeri*	371	115	*Pyrococcus abyssi*	330	100
Real_7	*Mycoplasma genitalium*	384	119	*Mycoplasma pneumoniae*	369	112
Real_8	*Saccharomyces kluveri*	336	90	*Schizosaccharomyces octosporus*	281	71
Real_9	*Serratia marcescens*	378	125	*Shewanella putrefaciens*	354	115
Real_10	*Streptomyces bikiniensis*	398	135	*Streptomyces lividans*	405	138

Table 6.1: Characteristics of real instances from set LAPCS-REAL. All 20 arc-annotated RNA sequences were taken from the RNase P Database [26].

Due to the fact that the amount of reduction of the conflict graphs from 6.3 depends on the quality of the used primal bound per instance, we used the currently best-known solution values from the literature for all considered instances. In the case of the RFLCS problem, these values were taken from [14], and in the case of the LAPCS problem from [15].

6.4.1 Results without conflict graph reduction

All three methods—CPLEX, LMC, and LSCC-BMS—were applied with a computation time limit of 3600 seconds (1 hour) and a memory limit of 16GB per run to all RFLCS and LAPCS problem instances. The results are presented in numerical form in Tables 6.2 and 6.3 concerning the RFLCS problem, and in Tables 6.4 and 6.5 concerning the LAPCS problem. The first two columns in Tables 6.2–6.4 indicate the problem instance characteristics, while the third column provides the currently best-known results from the literature. In the case of the RFLCS problem, the best-known results were obtained by the current state-of-the-art method—a hybrid CMSA algorithm—from [14]. The best-known results for the LAPCS problem were obtained by two state-of-the-art algorithms—a hybrid evolutionary algorithm (HYB-EA) and an ILP-based heuristic—from [15]. Note that, in this way, the results of our transformation-based approaches are compared to the currently leading methods. Each table row provides results averaged over 30 problem instances of the same type. 6.5 is slightly different. The first column provides the instance name, while the second column indicates the best-known results from the literature. Moreover, each table row only covers one single problem instance. In the case of the LAPCS problem, the best-known results from the literature are additionally marked either by an a, indicating that an ILP-based heuristic has produced this result, or by a b, which indicates that the HYB-EA algorithm has generated this result. In Tables 6.2–6.4, the results of CPLEX and LSM are each provided in four columns. The first one (with heading **r**esult) contains the average solution quality obtained for the 30 problem instances. The second column (with heading $\bar{t}$) indicates the average computation time at which the best solution of a run

Table 6.2: Experimental results for RFLCS instances RFLCS-SET1.

$\|\Sigma\|$	n	**Spec.**	CPLEX				LMC				LSCC+BMS	
		Tech.	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$
$n/8$	32	4.00	**4.00**	0.09	0.09	30	**4.00**	0.00	0.01	30	**4.0**	0.01
	64	8.00	**8.00**	0.81	0.81	30	**8.00**	0.00	0.07	30	**8.0**	0.00
	128	16.00	**16.00**	8.12	8.12	30	**16.00**	0.00	49.61	30	**16.0**	0.01
	256	31.97	**31.97**	188.31	188.31	30	31.90	20.54	– –	0	**31.97**	0.09
	512	63.27	5.17	625.34	– –	0	62.50	485.59	– –	0	**63.90** *	68.84
	1024	111.57	0.03	1461.74	– –	0	112.53	818.57	– –	0	**116.10** *	1297.00
	2048	182.67	– –	– –	– –	0	182.40	1331.53	– –	0	181.67	1394.27
	4096	283.33	– –	– –	– –	0	281.37	1037.61	– –	0	261.37	1510.89
$n/4$	32	7.83	**7.83**	0.03	0.03	30	**7.83**	0.00	0.00	30	**7.83**	0.00
	64	14.67	**14.67**	0.29	0.29	30	**14.67**	0.00	0.01	30	**14.67**	0.00
	128	25.77	**25.93** *	2.02	2.50	30	**25.93** *	0.01	0.09	30	**25.93** *	0.02
	256	43.70	**43.97** *	30.92	51.17	30	**43.97** *	0.12	0.80	30	**43.97** *	0.22
	512	67.90	68.50	582.53	1622.77	27	**68.57** *	75.61	185.15	30	**68.57** *	7.57
	1024	103.00	0.00	240.97	– –	0	103.77	386.81	– –	0	**104.87** *	877.29
	2048	154.33	0.00	1398.78	– –	0	152.87	438.52	– –	0	151.33	1485.85
	4096	226.67	– –	– –	– –	0	223.57	780.50	– –	0	207.03	1984.69
$3n/8$	32	8.77	**8.77**	0.02	0.02	30	**8.77**	0.00	0.00	30	**8.77**	0.00
	64	15.53	**15.53**	0.10	0.10	30	**15.53**	0.00	0.00	30	**15.53**	0.00
	128	24.90	**24.90**	1.75	1.79	30	**24.90**	0.00	0.03	30	**24.90**	0.01
	256	39.97	**39.97**	5.25	5.90	30	**39.97**	0.02	0.20	30	**39.97**	0.13
	512	59.77	**59.97** *	106.42	133.02	30	**59.97** *	0.46	1.83	30	**59.97** *	1.99
	1024	90.50	90.67	2204.06	2263.32	23	**90.73** *	5.71	30.67	30	**90.73** *	145.24
	2048	130.57	0.00	547.50	– –	0	129.67	233.36	105.92	1	129.13	1578.88
	4096	191.37	– –	– –	– –	0	188.30	311.61	– –	0	179.73	1670.85
$n/2$	32	8.87	**8.87**	0.01	0.01	30	**8.87**	0.00	0.00	30	**8.87**	0.00
	64	14.80	**14.80**	0.06	0.06	30	**14.80**	0.00	0.00	30	**14.80**	0.00
	128	22.93	**22.93**	0.76	0.78	30	**22.93**	0.00	0.01	30	**22.93**	0.00
	256	35.10	**35.20** *	2.18	2.27	30	**35.20** *	0.02	0.09	30	**35.20** *	0.09
	512	53.10	**53.13** *	31.82	34.03	30	**53.13** *	0.08	0.66	30	**53.13** *	0.71
	1024	79.03	**79.13** *	627.90	701.13	30	**79.13** *	6.04	11.56	30	**79.13** *	30.80
	2048	115.30	0.00	248.56	– –	0	115.07	432.97	598.59	19	114.87	1517.02
	4096	167.47	0.00	1295.77	– –	0	165.87	390.18	– –	0	159.37	1490.48
$5n/8$	32	8.60	**8.60**	0.01	0.01	30	**8.60**	0.00	0.00	30	**8.60**	0.00
	64	13.30	**13.30**	0.03	0.03	30	**13.30**	0.00	0.00	30	**13.30**	0.00
	128	21.20	**21.20**	0.36	0.37	30	**21.20**	0.00	0.01	30	**21.20**	0.00
	256	32.53	**32.53**	4.21	4.36	30	**32.53**	0.01	0.05	30	**32.53**	0.04
	512	47.83	**47.83**	13.06	13.15	30	**47.83**	0.04	0.33	30	**47.83**	0.28
	1024	70.03	**70.20** *	208.55	215.63	30	**70.20** *	1.43	4.12	30	**70.20** *	8.70
	2048	103.80	48.33	2306.93	3328.76	1	**103.97** *	63.19	158.21	30	103.87	936.80
	4096	150.00	0.00	878.84	– –	0	148.53	302.72	1607.66	2	145.77	1423.49
$3n/4$	32	8.17	**8.17**	0.00	0.00	30	**8.17**	0.00	0.00	30	**8.17**	0.00
	64	12.53	**12.53**	0.02	0.02	30	**12.53**	0.00	0.00	30	**12.53**	0.00
	128	19.70	**19.70**	0.17	0.18	30	**19.70**	0.00	0.00	30	**19.70**	0.00
	256	29.97	**29.97**	2.25	2.32	30	**29.97**	0.00	0.03	30	**29.97**	0.02
	512	44.53	**44.57** *	4.90	4.94	30	**44.57** *	0.03	0.19	30	**44.57** *	0.29
	1024	65.07	**65.20** *	96.77	97.46	30	**65.20** *	0.75	2.11	30	**65.20** *	3.39
	2048	94.53	**94.67** *	1829.86	1862.21	30	**94.67** *	4.57	18.69	30	94.63	638.75
	4096	136.57	0.00	500.41	– –	0	135.73	355.77	682.50	13	133.53	1617.99
$7n/8$	32	7.67	**7.67**	0.00	0.00	30	**7.67**	0.00	0.00	30	**7.67**	0.00
	64	11.57	**11.57**	0.01	0.01	30	**11.57**	0.00	0.00	30	**11.57**	0.00
	128	18.40	**18.40**	0.12	0.12	30	**18.40**	0.00	0.00	30	**18.40**	0.00
	256	27.80	**27.80**	1.21	1.22	30	**27.80**	0.00	0.02	30	**27.80**	0.01
	512	40.57	**40.60** *	2.93	3.01	30	**40.60** *	0.02	0.12	30	**40.60** *	0.10
	1024	60.50	**60.57** *	79.74	79.76	30	**60.57** *	0.28	1.19	30	**60.57** *	3.55
	2048	88.00	**88.00**	831.15	896.78	30	**88.00**	4.13	18.68	30	**88.00**	114.45
	4096	127.20	0.00	361.39	– –	0	126.50	212.34	478.99	17	125.47	1608.56

Table 6.3: Experimental results RFLCS instances Rflcs-Set2.

$\|\Sigma\|$	reps	Spec.	Cplex				Lmc				Lscc-Bmc	
		Tech.	result	$\bar{t}$	$\bar{t}_{\text{opt}}$	#opt	result	$\bar{t}$	$\bar{t}_{\text{opt}}$	#opt	result	$\bar{t}$
4	3	3.47	**3.47**	0.00	0.00	30	**3.47**	0.00	0.00	30	**3.47**	0.00
	4	3.77	**3.77**	0.00	0.00	30	**3.77**	0.00	0.00	30	**3.77**	0.00
	5	3.83	**3.83**	0.00	0.00	30	**3.83**	0.00	0.00	30	**3.83**	0.00
	6	3.90	**3.90**	0.00	0.00	30	**3.90**	0.00	0.00	30	**3.90**	0.00
	7	3.97	**3.97**	0.01	0.01	30	**3.97**	0.00	0.00	30	**3.97**	0.00
	8	3.97	**3.97**	0.01	0.01	30	**3.97**	0.00	0.00	30	**3.97**	0.00
8	3	6.23	**6.23**	0.00	0.00	30	**6.23**	0.00	0.00	30	**6.23**	0.00
	4	6.87	**6.87**	0.00	0.00	30	**6.87**	0.00	0.00	30	**6.87**	0.00
	5	7.40	**7.40**	0.02	0.02	30	**7.40**	0.00	0.00	30	**7.40**	0.00
	6	7.53	**7.53**	0.02	0.02	30	**7.53**	0.00	0.00	30	**7.53**	0.00
	7	7.70	**7.70**	0.06	0.06	30	**7.70**	0.00	0.00	30	**7.70**	0.00
	8	7.77	**7.77**	0.05	0.05	30	**7.77**	0.00	0.00	30	**7.77**	0.00
16	3	9.70	**9.70**	0.01	0.01	30	**9.70**	0.00	0.00	30	**9.70**	0.00
	4	11.57	**11.57**	0.03	0.03	30	**11.57**	0.00	0.00	30	**11.57**	0.00
	5	12.93	**12.93**	0.06	0.06	30	**12.93**	0.00	0.00	30	**12.93**	0.00
	6	14.00	**14.00**	0.15	0.16	30	**14.00**	0.00	0.01	30	**14.00**	0.00
	7	14.93	**14.93**	0.30	0.30	30	**14.93**	0.00	0.02	30	**14.93**	0.02
	8	14.80	**14.80**	0.37	0.38	30	**14.80**	0.00	0.02	30	**14.80**	0.00
32	3	16.13	**16.13**	0.08	0.08	30	**16.13**	0.00	0.00	30	**16.13**	0.00
	4	19.00	**19.00**	0.27	0.27	30	**19.00**	0.00	0.01	30	**19.00**	0.00
	5	21.63	**21.63**	0.83	0.85	30	**21.63**	0.00	0.02	30	**21.63**	0.01
	6	23.73	**23.73**	1.57	1.65	30	**23.73**	0.00	0.04	30	**23.73**	0.01
	7	25.53	**25.57** *	2.23	2.34	30	**25.57** *	0.02	0.10	30	**25.57** *	0.03
	8	27.40	**27.50** *	4.59	4.71	30	**27.50** *	0.06	0.23	30	**27.50** *	0.07
64	3	25.43	**25.43**	0.88	0.91	30	**25.43**	0.00	0.01	30	**25.43**	0.00
	4	30.37	**30.37**	2.65	2.80	30	**30.37**	0.01	0.05	30	**30.37**	0.02
	5	34.87	**34.93** *	3.57	4.66	30	**34.93** *	0.02	0.13	30	**34.93** *	0.07
	6	39.07	**39.13** *	13.36	17.37	30	**39.13** *	0.05	0.34	30	**39.13** *	0.18
	7	43.50	**43.63** *	28.44	55.76	30	**43.63** *	0.16	0.92	30	**43.63** *	0.40
	8	45.17	**45.53** *	58.39	116.58	30	**45.53** *	1.38	5.41	30	**45.53** *	0.75
128	3	36.70	**36.77** *	2.39	2.44	30	**36.77** *	0.01	0.09	30	**36.77** *	0.14
	4	44.90	**45.03** *	12.95	15.22	30	**45.03** *	0.06	0.37	30	**45.03** *	0.39
	5	53.23	**53.43** *	48.50	64.03	30	**53.43** *	0.15	1.08	30	**53.43** *	1.12
	6	61.07	**61.53** *	183.29	300.56	30	**61.53** *	4.55	7.76	30	**61.53** *	4.42
	7	67.90	68.40	749.39	1377.40	25	**68.47** *	8.25	54.19	30	**68.47** *	5.13
	8	73.57	74.37	1288.16	1932.99	11	74.30	524.20	474.27	13	**74.60** *	22.68
256	3	54.97	**55.03** *	46.81	48.61	30	**55.03** *	0.08	0.69	30	**55.03** *	1.06
	4	68.70	**68.93** *	247.83	268.90	30	**68.93** *	0.31	2.90	30	**68.93** *	8.53
	5	81.00	**81.43** *	917.97	1182.86	30	**81.43** *	9.65	21.74	30	**81.43** *	45.01
	6	93.10	73.83	2951.48	3090.98	2	93.17	239.22	418.66	17	**93.53** *	162.94
	7	103.50	0.00	308.34	– –	0	103.13	132.52	499.03	3	**104.40** *	734.99
	8	113.70	0.00	501.06	– –	0	113.10	298.94	– –	0	**114.70** *	1300.54
512	3	81.57	**81.63** *	524.51	536.33	30	**81.63** *	0.72	5.38	30	**81.63** *	41.71
	4	100.83	78.63	2899.04	3142.25	3	101.10	157.68	230.29	29	**101.13** *	602.19
	5	120.43	0.00	404.86	– –	0	118.70	539.36	851.21	5	119.60	1147.39
	6	137.03	0.00	681.76	– –	0	135.50	483.24	– –	0	136.00	1894.44
	7	154.57	0.00	1218.70	– –	0	152.33	784.72	– –	0	150.63	1784.08
	8	172.10	– –	– –	– –	0	169.90	698.89	– –	0	166.47	1428.87

Table 6.4: Experimental results for LAPCS instances LAPCS-ARTI.

was found, while the third column (with heading $\bar{t}_{\mathbf{opt}}$) provides the average computation time at which optimality was proven. Finally, the fourth table column contains the number of instances that could be solved to optimality. This fourth table column is not provided in Table 6.5, as it only deals with one instance per table row. Furthermore, the results of LSCC-BMS are given in two columns in all cases, providing the (average) result and the (average) computation time. Note that a value in the columns with heading **result** is indicated in bold font if the value is at least as good as the best-known one from the literature. Moreover, a value is marked by an asterisk in case it corresponds to a new best-known result. Finally, the results of CPLEX and LMC are marked by a grey background if they correspond to provenly optimal results.

The following observations can be made in the case of the RFLCS problem:

- While both LMC and LSCC-BMS are able to provide feasible solutions for all problem instances from both sets (RFLCS-SET1 and RFLCS-SET2), CPLEX suffers from a sharp phase transition when the conflict graphs become too large. Observe, for example, the case ($|\Sigma| = n/8, n = 256$) in Table 6.2 in comparison to the next larger case ($|\Sigma| = n/8, n = 512$). While CPLEX is able to solve all instances of the first case to optimality, it only provides very short solutions in the second case.

- Concerning the comparison of the two exact solvers, we can state that LMC (the MC solver) clearly outperforms CPLEX. LMC is able to solve 1282 RFLCS-SET1 instances and 1237 RFLCS-SET2 instances to optimality, while CPLEX can only solve 1221 RFLCS-SET1 instance and 1181 RFLCS-SET2 instances to optimality. Moreover, LMC does not suffer from the above-mentioned phase transition for the remaining instances, and it requires generally less computation time. More specifically, while LSM requires—on average—41.7 seconds for proving optimality (if possible) of RFLCS-SET1 instances, CPLEX requires 187.2 seconds; respectively 34.07 and 127.14 seconds in the case of the RFLCS-SET2 instances.

- The heuristic MC solver LSCC-BMS is especially successful in those cases in which the exact techniques start to fail. See, for example, cases ($|\Sigma| = n/8, n \in \{512, 1024\}$) in Table 6.2 and cases ($|\Sigma| = 256, \text{reps} \in \{6, 7, 8\}$) in Table 6.3. LSCC-BMS can be seen as the most successful one among the techniques, providing new best-known results in 35 cases (considering both instance sets together), while LMC provides new best-known results in 30 cases and CPLEX in 24 cases.

All in all we can state that the idea of solving the RFLCS problem by means of the transformation to the MC problem is very successful, even before trying to reduce the size of the conflict graphs.

Table 6.5: Experimental results for LAPCS instances Lapcs-Real.

Inst.	Spec.	Cplex			Lmc			Lscc-Bms	
Name	**Tech.**	**result**	t	t_{opt}	**result**	t	t_{opt}	**result**	t
Real_1	268^b	– –	– –	– –	259	2691.58	– –	231	3504.69
Real_2	291^b	– –	– –	– –	283	637.94	– –	216	1088.45
Real_3	294^b	– –	– –	– –	284	104.13	– –	234	1580.28
Real_4	374^b	– –	– –	– –	**374**	34.59	– –	366	2148.66
Real_5	178^b	– –	– –	– –	**179***	6.04	– –	170	2336.97
Real_6	209^b	– –	– –	– –	206	30.64	– –	197	2181.59
Real_7	330^b	– –	– –	– –	**330**	43.61	– –	251	1461.38
Real_8	177^b	– –	– –	– –	175	3309.91	– –	173	448.26
Real_9	302^b	– –	– –	– –	**304***	44.36	– –	226	49.66
Real_10	361^a	– –	– –	– –	**361**	71.14	– –	272	496.70

Let us now turn towards the LAPCS problem. In some aspects, the observations that can be made in the context of the artificial instances (Lapcs-Arti; Table 6.4) are similar to the ones made for the RFLCS problem. Cplex suffers from a sharp phase transition. In fact, it is only able to provide solutions for the case of the smallest problem instances ($n = 100$). Lmc does not suffer from this phase transition and is able to provide feasible solutions of reasonable quality until instances with input strings of length $n = 500$. Both Lmc are Cplex are able to solve 80 problem instances to optimality. And finally, the heuristic MC solver Lscc-Bms is again very successful in those cases in which Lmc and Cplex start to fail proving optimality (see the instances with $n = 200$). Concerning the results obtained for the real instances (Lapcs-Real; Table 6.5), we can state that Lsm is, by far, the most successful algorithm. While Cplex is not able to derive any feasible solutions and Lscc-Bms never matches the best results from the literature, Lsm matches the best results from the literature in three cases and obtains new best-known solutions in two additional cases. Nevertheless, we can state that the results—obtained before trying to reduce the size of the conflict graphs—are rather unsatisfactory in the context of the LAPCS problem. The main reason for this is the increased size of the conflict graphs in comparison to the RFLCS problem, which is due to the small alphabet size of four.

6.4.2 Results after conflict graph reduction

After reducing all the conflict graphs with the method described in 6.3, we first measured the amount of reduction that was achieved. This reduction is displayed for all RFLCS and LAPCS problem instances by means of boxplots in Figures 6.6–6.9. More specifically, the boxplots show the percentage reduction concerning the number of vertices of the original conflict graphs. If the reduction for an instance is at 60%, for example, this means that the reduction technique was able to remove 60% of the vertices of the original conflict graph. In the context of the RFLCS instances, we can state that the percentage reduction tends to grow with a growing string length and a growing alphabet size. Note

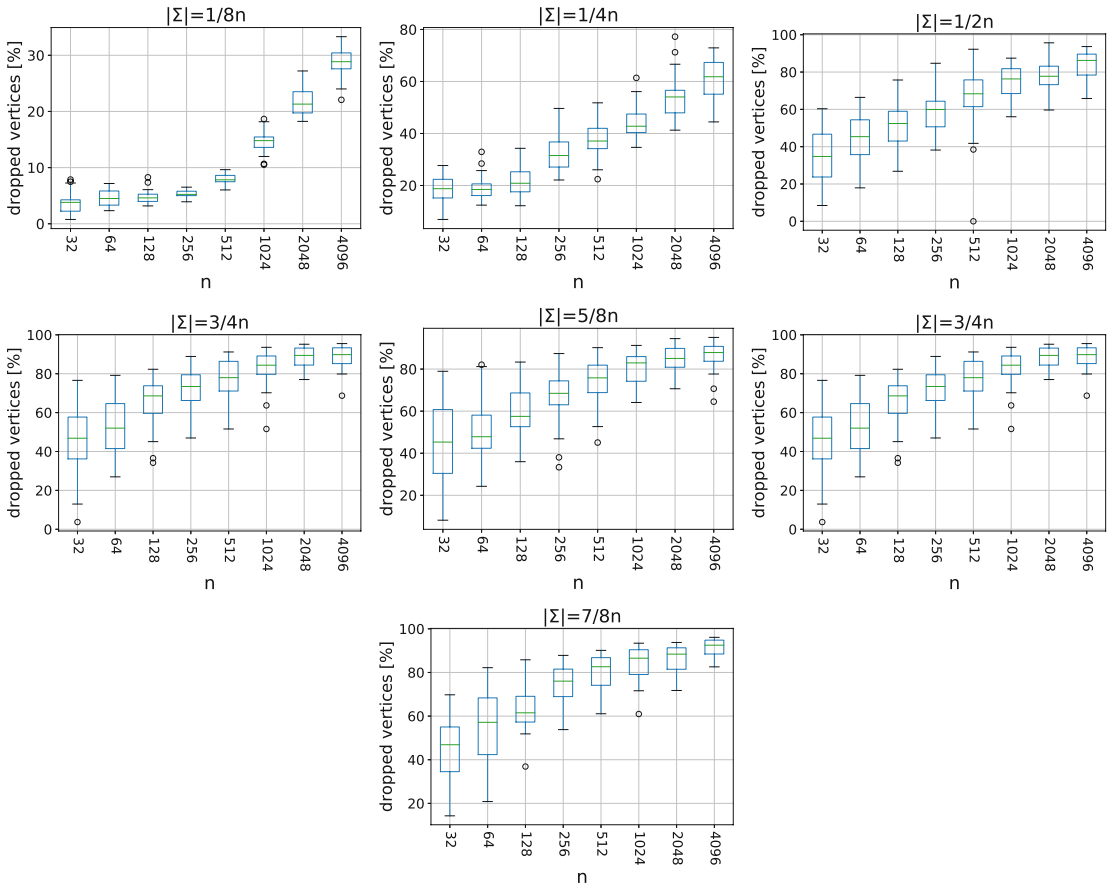


Figure 6.6: Graph reduction (in %) for RFLCS instances from set RFLCS-SET1.

that for long strings on large alphabets we were able to achieve reduction percentages of more than 90%. Concerning the LAPCS problem, it can be observed that the reduction percentages grow with an increasing number of arc annotations. However, they slightly increase with a growing input string length. This is due to the small alphabet size of four. Finally, it is worth mentioning that in the case of the real problem instances (set LAPCS-ARTI; 6.9) we were able to achieve very high reduction percentages, sometimes well over 90%. This indicates the difference in structure between artificial and real problem instances.

The numerical results obtained by the three considered techniques after conflict graph reduction are provided in Tables C.1–C.4 that can be found in Appendix C. The structure of these tables is very similat to the one of Tables 6.2–6.5 which was described at the beginning of 6.4.1. The only difference is that the part on the state-of-the-art results—see the columns with heading "Spec. Tech."–is now extended by the corresponding computation times, that is, the times at which these results were obtained by the respective

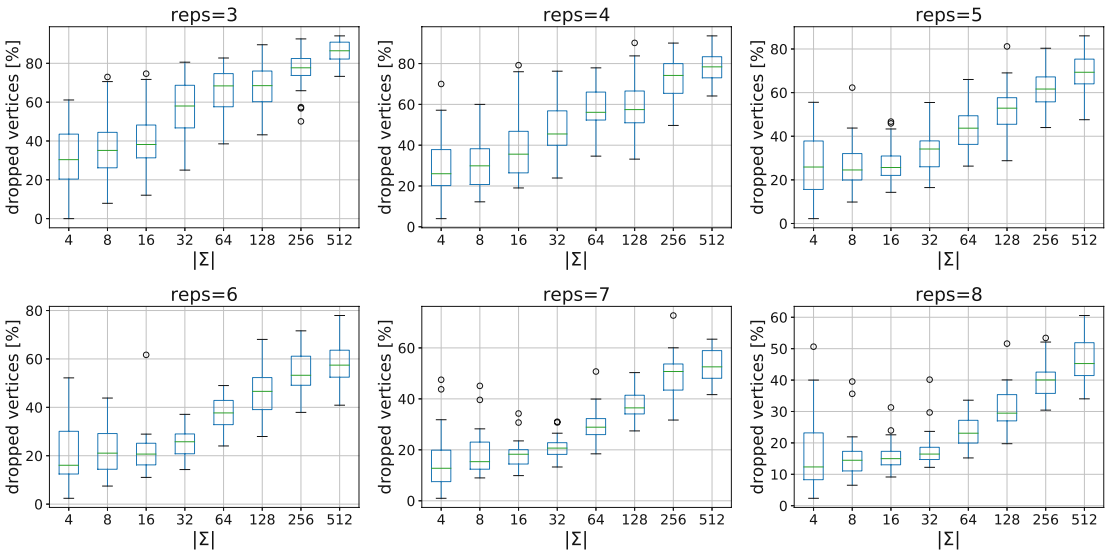


Figure 6.7: Graph reduction (in %) for RFLCS instances from set Rflcs-Set2.

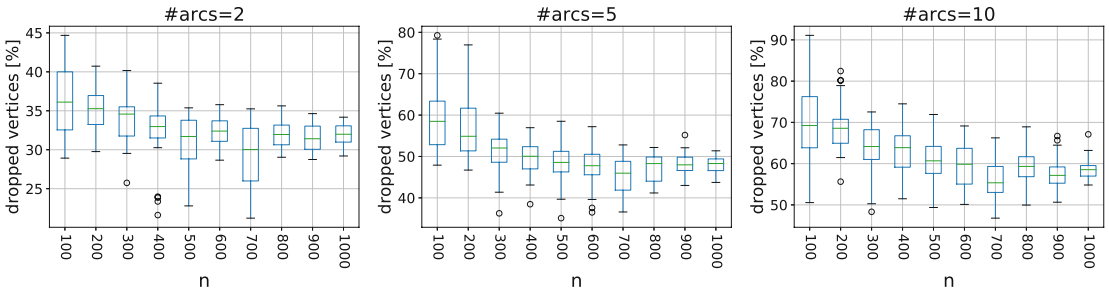


Figure 6.8: Graph reduction (in %) for LAPCS instances from set Lapcs-Arti.

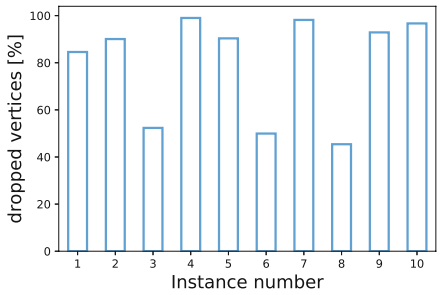


Figure 6.9: Graph reduction (in %) for LAPCS instances from set Lapcs-Real.

Table 6.6: Differences in performance of the exact methods (Cplex and Lmc) summarized for the four different data sets. The five measures (E-M1–E-M5) are described in the text.

Data set	Cplex					Lmc				
	E-M1	**E-M2**	**E-M3**	**E-M4**	**E-M5**	**E-M1**	**E-M2**	**E-M3**	**E-M4**	**E-M5**
Rflcs-Set1	144.29	172.77	257	61.29	60	9.33	32.99	84	0.24	0
Rflcs-Set2	78.90	105.19	133	82.34	30	5.58	14.90	10	0.20	0
Lapcs-Arti	330.89	526.45	30	-0.39	120	29.70	20.02	0	-0.0004	150
Lapcs-Real	– –	– –	7	– –	9	– –	– –	5	0.0	0

techniques.[4] The computation time of the fastest technique in each table row is underlined. Moreover, in order to relate the two sets of results, the values in the columns with heading **result** are marked differently. More specifically, values marked by a preceding =-symbol are equal to the values obtained by the same technique before graph reduction. Furthermore, values marked in italic font and by a preceding −-symbol are worse than the values obtained by the same technique before graph reduction, and values marked in bold font and by a preceding +-symbol are better than the corresponding values before conflict graph reduction.

In order to relate the performance of a technique before graph reduction with its performance after graph reduction, we also computed a set of measures that are provided in Table 6.6 for Cplex and Lmc, and in Table 6.7 for Lscc-Bms. The measures regarding the exact techniques (see Table 6.6) are as follows.

1. **Measure E-M1** refers to those instances that were solved to optimality, both concerning the original conflict graph and the reduced conflict graph. In particular, it provides the average time saved for finding the best solution of a run (in seconds) after reducing the respective graph.
2. **Measure E-M2** is very similar, just that it refers to the average time saving for proving optimality.
3. **Measure E-M3** indicates the number of instances additionally solved to optimality after graph reduction.
4. **Measure E-M4** indicates the average improvement in solution quality (in percent) for all those instances for which feasible solutions can be found both before and after graph reduction, but for which optimality cannot be proven.
5. Finally, **measure E-M5** reports on the number of instances for which a feasible (and possibly optimal) solution can be found after graph reduction, and for which no feasible solution could be found before graph reduction.

[4] Note that the state-of-the-art techniques for both the RFLCS and the LAPCS problem were executed on the same computers as Cplex and Lscc-Bms.

Table 6.7: Differences in performance of the heuristic method (Lscc-Bms) summarized for the four different data sets. The four measures (H-M1–H-M4) are described in the text.

Data set	Lscc-Bms			
	H-M1	**H-M2**	**H-M3**	**H-M4**
Rflcs-Set1	77.34	296	27	0
Rflcs-Set2	39.41	114	9	0
Lapcs-Arti	176.21	280	33	30
Lapcs-Real	– –	9	1	0

In the context of the heuristic MC solver Lscc-Bms (see Table 6.7), measures H-M1–H-M4 can be described as follows.

1. In all those cases in which the same result is obtained by Lscc-Bms before and after conflict graph reduction, **measure H-M1** refers to the average time saving per instance (in seconds) for achieving this result.
2. **Measure H-M2** indicates the number of instances for which the result of Lscc-Bms improves after graph reduction.
3. **Measure H-M3** refers to the number of cases in which the result gets worse.
4. Finally, **measure H-M4** counts the number of instances for which Lscc-Bms can find a feasible solution after graph reduction, while before graph reduction Lscc-Bms was not able to find any feasible solution.

Remarks concerning the results for the RFLCS problem:

- The great beneficiary of the applied conflict graph reduction is Cplex. Cplex is now able to solve 1478 Rflcs-Set1 instances (out of 1680) and 1314 Rflcs-Set2 instances (out of 1440) to optimality, while Lmc now solves 1366 Rflcs-Set1 instances and 1247 Rflcs-Set2 instances to optimality. Nevertheless, Cplex still suffers from a sharp phase transition which, due to the graph reduction, has been moved to larger problem instances. Also the time savings achieved for finding the best solutions of a run and for proving optimality are much higher in the case of Cplex when compared to those of Lmc (see Table 6.6).
- The heuristic MC solver Lscc-Bms is also able to profit from the graph reduction. It provides an improved result for 296 Rflcs-Set1 instances and for 114 Rflcs-Set2 instances, while worse results are only produced in 27, respectively 9, cases. Moreover, in those cases in which Lscc-Bms obtains the same result before and after graph reduction, the average time saving per instance is approx. 77 seconds for the Rflcs-Set1 instances, and approx. 39 seconds for the Rflcs-Set2 instances.

After studying the results obtained for the LAPCS instances, the following observations can be made:

- Concerning the set of artificial problem instances (LAPCS-ARTI), it can be observed that all three techniques are now able to provide solutions for some of the larger instances. CPLEX, for example, can now provide solutions for the instances with $n = 200$ and for the case $(n = 300, n_{\text{arcs}} = 30)$, for which no result was obtained before conflict graph reduction. However, while LSCC-BMS is able to improve its results for many instances (see cases $n \in \{200, \ldots, 500\}$), LSM is again not able to take much profit from the graph reduction. In fact, the results of LSM after graph reduction are sometimes even worse than before; see case $n = 200$ and $n_{\text{arcs}} \in \{20, 40\}$, for example. On average, LSM is not able to improve its results for those instances for which a feasible solution was obtained before and after conflict graph reduction, but for which optimality could not be proven; see measure E-H4 in Table 6.6. CPLEX is now able to solve 110 problem instances to optimality, while LSM can solve 80 problem instances, the same ones that it was able to solve before conflict graph reduction. Again, LSCC-BMS performs best when the performance of CPLEX and LSM starts to decline (see the cases with $n = 200$).
- Finally, the results—in particular those of CPLEX—for the real-life instances of set LAPCS-REAL are quite pleasing. CPLEX is able to solve seven out of 10 instances to optimality. In three of these cases, the best-known result from the literature is improved. LSM, on the other side, obtains exactly the same results as before conflict graph reduction, with the difference that optimality can be proven now for five out of the 10 problem instances. LSCC-BMS is again able to take profit from the graph reduction, improving its results in 9 out of 10 cases.

Finally, a summary of the obtained results in comparison to the current state of the art is provided in Table 6.8. Concerning the instances of set RFLCS-SET1, for example, our algorithm approaches were able to improve the current state-of-the-art algorithm from [14] in 23 out of 56 cases, the results were matched in 28 cases, and in only five cases our results were inferior to the state of the art. In general, Table 6.8 shows that our transformation-based approaches are very successful in the context of the RFLCS problem, while they only succeeded for the smaller instances of set LAPCS-ARTI and the real-life instances from set LAPCS-REAL. Finally, the following observations can be made concerning the comparison of the computation times of our approaches with those of the state of the art. In the context of the RFLCS problem, our approaches are generally faster than the current state-of-the-art approaches, especially for instances of RFLCS-SET1 with alphabets of medium and large size. The state-of-the-art approaches are only faster for RFLCS-SET1 instances with small alphabet sizes and rather long input strings. The same happens for RFLCS-SET2 instances with large alphabet sizes and many repetitions. The computation time comparison concerning the LAPCS problem reflects the analysis from above concerning solution quality. In particular, our approaches–especially LSCC-BMS–are only faster than the current state of the art for the smallest

Table 6.8: Number of instances for which better, equally good, and worse solutions were obtained.

	RFLCS-Set1	RFLCS-Set2	LAPCS-Arti	LAPCS-Real
Better	23	22	6	3
Equal	28	24	0	4
Worse	5	2	24	3

LAPCS-Arti instances with input string length $n = 100$. Starting from LAPCS-Arti instances with $n = 200$, our approaches require considerably more computation time than state-of-the-art techniques. On the other side, in the context of the real life instances of set LAPCS-Real our approaches are faster in eight out of 10 cases, reflecting the good results obtained for this instance set.

6.5 Conclusions

In this chapter, we proposed a new way to transform longest common subsequence problem instances into instances of the maximal clique problem. Moreover, we defined a technique for the reduction of the resulting graphs, based on high-quality primal bounds. The benefits of this approach were experimentally studied in the context of two longest common subsequence variants: (*i*) the repetition-free longest common subsequence (RFLCS) problem and (*ii*) the longest arc-preserving common subsequence (LAPCS) problem. Both problem variants are $\mathcal{NP}$-hard even for two input strings. We compared the application of CPLEX for solving the maximum independent set problem, which is the complementary problem of the maximal clique problem, with the application of recent heuristic and exact maximal clique solvers. The three approaches were applied both before and after graph reduction. The best results were obtained after graph reduction, even though the impact of graph reduction was very different for the three solvers. Summarizing, we were able to solve 2613 of the 3120 RFLCS instances to optimality. Moreover, 110 out of 900 artificially created LAPCS problem instances were solved to optimality. In the context of the LAPCS problem, it was especially pleasing to see seven out of 10 real-life instances solved to optimality for the first time.

CHAPTER 7

The Constrained Longest Common Subsequence Problem

In this chapter, we study a generalized constrained longest common subsequence problem which is given as follow. Given an arbitrary set of input strings S and a pattern string P, the problem aims at finding a subsequence common for all strings from S having P as its subsequence.

The content of this chapter includes two published papers.

- A journal paper [52] published in the *Information Processing Letters* journal (IF=0.677) [52]. In this paper, we have presented an A* search algorithm to solve the classical CLCS problem (with two input strings). The A* search is compared to a few state-of-the-art approaches from the literature. The approach delivers an order of magnitude lower runtimes to prove optimality in comparison to a few other state-of-the-art approaches from the literature.

- The conference paper published in the Proceedings of the *11th International Conference Optimization and Applications* (OPTIMA-20) conference [53]. In the course of this work, we consider the $\mathcal{NP}$–hard variant of the CLCS problem where an arbitrary set of the input strings and a single pattern string are given in the input. First, we adopt an existing A* search from the classical CLCS problem with two input strings to an arbitrary number of input strings. To tackle large problem instances approximately, we additionally propose a greedy heuristic and a beam search approach. Various search guidances have been proposed to guide the search towards promising regions. It is important to mention a probability-based heuristic and the expected length calculation heuristic for the CLCS problem which represent extensions of the respective heuristics for the LCS problem. Beam search

turns out to be the best heuristic approach, returning almost all optimal solutions obtained by A* search.

Moreover, in the course of this project, the master thesis of Christoph Berger [10] was successfully done. In his thesis, extended computational study on the m–CLCS problem are presented. Christoph Berger took an active role in the research such as implementing the A* search for the m–CLCS problem as well as testing and analyzing the results for the both of our papers. The duties of the author of the thesis concerned of the thesis' supervision, actively constructing and implementing the approaches, verifying and validating the implementation, writing the drafts of both papers, and responding to the referees' comments.

7.1 Introduction

The *generalized constrained longest common subsequence* (m–CLCS) problem [74] considered in this chapter is stated as follows. Given m input strings and a pattern string P, we seek for a longest common subsequence for the input strings that includes P as its subsequence. This problem gives a useful measure of similarity when additional information which concerns of the common structure of all the input strings is known beforehand. The most studied CLCS variant in the literature is the one where only two input strings (2–CLCS) are given as an input; see, for example, [162, 5, 34].

The m–CLCS problem is $\mathcal{NP}$–hard [1]. An application of this general variant is motivated from computational biology when it is necessary to find the commonality for not just two but an arbitrary number of DNA molecules. When computing the commonality of more than two biological sequences, it may be important to include a common specific structure. To the best of our knowledge, the approximation algorithm by Gotthilf et al. [74] is the only existing algorithm for solving the general m–CLCS problem so far.

We first propose the general search framework for the m–CLCS problem and develop an A* search framework. Since A* will sooner or later comes to the issues with scalability, to solve the large-scale instances we propose two heuristic techniques: (i) a greedy heuristic that is efficient for producing reasonably good solutions within a short runtime, and (ii) a beam search (BS) technique which is able to produce high-quality solutions but takes more time. In order to efficiently run the BS framework, we developed two effective search guidances: a probability-based guidance and the guidance that approximates the expected length calculation of an CLCS on random strings.

This chapter is organized as follows. Section 7.2 describes the greedy heuristic for the m–CLCS problem. In Section 7.3 the general search framework for the m–CLCS problem is presented. Section 7.4 describes the A* search, and in Section 7.5 the beam search is proposed. In Section 7.6, the detailed computational experiments are presented. Section 7.7 concludes this work and outlines directions for future research.

7.2 A Fast Heuristic for the m–CLCS Problem

We make use of two data structures created during preprocessing to set up an efficient search:

- For each $i = 1, \ldots, m$, $j = 1, \ldots, |s_i|$, and $c \in \Sigma$, $Succ[i, j, c]$ stores the minimal position index x such that $x \geq j \wedge s_i[x] = c$ or -1 if c does not occur in s_i from position j onward. This structure is built in $O(m \cdot n \cdot |\Sigma|)$ time.
- For each $i = 1, \ldots, m$, $u = 1, \ldots, |P|$, $Embed[i, u]$ stores the right-most position x of s_i such that $P[u, |P|]$ is a subsequence of $s_i[x, |s_i|]$. If no such position exists, $Embed[i, u] := -1$. This table is built in $O(|P| \cdot m)$ time.

In the following we present Greedy, a heuristic for the m–CLCS problem inspired by the well-known Best–Next heuristic [85] for the LCS problem. Greedy is pseudo-coded in Algorithm 24. The basic principle is straight-forward. The algorithm starts with an empty solution string $s := \epsilon$ and proceeds by appending, at each construction step, exactly one letter to s. The choice of the letter to append is done by means of a greedy function. The procedure stops once no more letters can be added. The basic data structure of the algorithm is a *position vector* $\mathbf{p}^s = (p_1^s, \ldots, p_m^s) \in \mathbb{N}^m$ which is initialized to $\mathbf{p}^s := (1, \ldots, 1)$ at the beginning. The superscript indicates that this position vector depends on the current (partial) solution s. Given $\mathbf{p}^s$, $s_i[p_i^s, |s_i|]$ for $i = 1, \ldots, m$ refer to the substrings from which letters can still be chosen for extending the current partial solution s. Moreover, the algorithm starts with a pattern position index $u := 1$. The meaning of u is that $P[u, |P|]$ is the substring of P that remains to be included as a subsequence in s. At each construction step, first, a subset $\Sigma_{\text{feas}} \subseteq \Sigma$ of letters is determined that can feasibly extend the current partial solution s, ensuring that the final outcome contains pattern P as a subsequence. More specifically, Σ_{feas} contains a letter $c \in \Sigma$ iff (i) c appears in all strings $s_i[p_i^s, |s_i|]$ and (ii) $s \cdot c$ can be extended towards a solution that includes pattern P. Condition (ii) is fulfilled if $u = |P| + 1$, $P[u] = c$, or $Succ[i, p_i^s, c] < Embed[i, u]$ for all $i = 1, \ldots, m$ (assuming that there is at least one feasible solution). These three cases are checked in the given order, and with the first case that evaluates to true, condition (ii) evaluates to true; otherwise, condition (ii) evaluates to false. Next, dominated letters are removed from Σ_{feas}. For two letters $c, c' \in \Sigma_{\text{feas}}$, we say that c *dominates* c' iff $Succ[i, p_i^s, c] \leq Succ[i, p_i^s, c']$ for all $i = 1, \ldots, m$. Afterwards, the remaining letters in Σ_{feas} are evaluated by the greedy function explained below, and a letter c^* that has the best greedy value is chosen and appended to s. Further, the position vector $\mathbf{p}^s$ is updated w.r.t. letter c^* by $p_i^s := Succ[i, p_i^s, c^*] + 1$, $i = 1, \ldots, m$. Moreover, u is increased by one if $c^* = P[u]$. These steps are repeated until $\Sigma_{\text{feas}} = \emptyset$, and the greedy solution s is returned.

The greedy function used to evaluate each letter $c \in \Sigma_{\text{feas}}$ is

$$g(\mathbf{p}^s, u, c) = \frac{1}{l_{\min}(\mathbf{p}^s, c) + \mathbb{1}_{P[u]=c}} + \sum_{i=1}^{m} \frac{Succ[i, p_i^s, c] - p_i^s + 1}{|s_i| - p_i^s + 1}, \tag{7.1}$$

where $l_{\min}(\mathbf{p}^s, c)$ is the length of the shortest remaining part of any of the input strings when considering letter c appended to the solution string and thus consumed, i.e., $l_{\min} := \min\{|s_i| - Succ[i, p_i^s, c] \mid i = 1, \ldots, m\}$, and $1_{P[u]=c}$ evaluates to one if $P[u] = c$ and to zero otherwise. Greedy chooses at each construction step a letter that minimizes $g()$. The first term of $g()$ penalizes letters for which the $l_{\min}$ is decreased more and which are not the next letter from $P[u]$. The second term in Eq. (4.4) represents the sum of the ratios of characters that are skipped (in relation to the remaining part of each input string) when extending the current solution s with letter c.

Algorithm 24 Greedy heuristic for the m–CLCS problem

```
1: Input: problem instance I = (S, P, Σ)
2: Output: heuristic solution s
3: s ← ε
4: p_i^s ← 1, i = 1, ..., m
5: u ← 1
6: Σ_feas ← set of feasible and non-dominated letters for extending s
7: while Σ_feas ≠ ∅ do
8:     c* ← arg min{g(p^s, u, c) | c ∈ Σ_feas)}
9:     s ← s · c*
10:    for i ← 1 to m do
11:        p_i^s ← Succ[i, p_i^s, c*] + 1
12:    end for
13:    if P[u] = c* then
14:        u ← u + 1 // consider next letter in P
15:    end if
16:    Σ_feas ← set of feasible and non-dominated letters for extending s
17: end while
18: return s
```

7.3 State Graph for the m–CLCS Problem

This section describes the state graph for the m–CLCS problem, in which paths from a dedicated root node to inner nodes correspond to (meaningful) partial solutions, paths from the root to sink nodes correspond to complete solutions, and directed arcs represent (meaningful) extensions of partial solutions.

Given an m–CLCS problem instance $I = (S, P, \Sigma)$, let s be any string over Σ that is a common subsequence of all input strings S. Such a (partial) solution s induces a position vector $\mathbf{p}^s$ in a well-defined way by assigning a value to each p_i^s, $i = 1, \ldots, m$, such that $s_i[1, p_i^s - 1]$ is the smallest string among all strings in $\{s_i[1, k] \mid k = 1, \ldots, p_i^s - 1\}$ that contains s as a subsequence. Note that these position vector are the same ones as already defined in the context of Greedy. In other words, s induces a subproblem $S[\mathbf{p}^s] := \{s_1[p_1^s, |s_1|], \ldots, s_m[p_m^s, |s_m|]\}$ of the original problem instance. This is because s

can only be extended by adding letters that appear in all strings of $s_i[p_i^s, |s_i|]$, $i = 1, \ldots, m$. In this context, let substring $P[1, k']$ of pattern string P be the maximal string among all strings of $P[1, k]$, $k = 1, \ldots, |P|$, 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 the strings in subproblem $S[\mathbf{p}^s]$, that is, a subsequence of $s_i[p_i^s, |s_i|]$ for all $i = 1, \ldots, m$.

The state graph $G = (V, A)$ of our A* search is a *directed acyclic graph* where each node $v \in V$ stores a triple $(\mathbf{p}^v, l^v, u^v)$, with $\mathbf{p}^v$ being a position vector that induces subproblem $S[\mathbf{p}^v]$, l^v is the length of (any) valid partial solution (i.e., path from the root to node v) that induces $\mathbf{p}^v$, and u^v is the length of the longest prefix string of pattern P that is contained as a subsequence in any of the partial solutions that induce node v. Moreover, there is an arc $a = (v, v') \in A$ labeled with letter $c(a) \in \Sigma$ between two nodes $v = (\mathbf{p}^v, l^v, u^v)$ and $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ iff (i) $l^{v'} = l^v + 1$ and (ii) subproblem $S[\mathbf{p}^{v'}]$ is induced by the partial solution that is obtained by appending letter $c(a)$ to the end of a partial solution that induces v. As mentioned above, we are only interested in meaningful partial solutions, and thus, for feasibly extending a node v, only the letters from Σ_{feas} can be chosen (see Section 7.2 for the definition of Σ_{feas}). An extension $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ is therefore generated for each $c \in \Sigma_{\text{feas}}$ in the following way: $p_i^{v'} = Succ[i, p_i^v, c] + 1$ for $i = 1, \ldots, m$, $l^{v'} = l^v + 1$, and $u^{v'} = u^v + 1$ in case $c = P[u^v]$, respectively $u^{v'} = u^v$ otherwise.

The *root* node of the state graph is defined by $r = (\mathbf{p}^r = (1, \ldots, 1), l^r = 0, u^r = 1)$ and it thus represents the original problem instance. Sink nodes correspond to non-extensible states. A longest path from the root node to some sink node represents an optimal solution to the m–CLCS problem. Figure 7.1 shows as example the full state graph for the problem instance $(\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}, s_3 = \texttt{bbccabcdbba}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\})$. 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. Moreover, note that node $((6, 5, 5), 3, 2)$ (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` cannot lead to any feasible solution, because any solution starting with `bccd` does not have $P =$ `cbb` as a subsequence.

7.3.1 Upper Bounds for the m–CLCS Problem

As any upper bound for the general LCS problem (see Section 3.3.1) is also valid for the m–CLCS problem [55], we adopt $\text{UB}(v) = \min\{\text{UB}_1(v), \text{UB}_2(v)\}$ from existing work on the LCS problem.

We again emphasize that this upper bound has desirable theoretical properties like *admissibility* for the A* search, which means that its values never underestimate the optimal value of the subproblem that corresponds to a node v, and *monotonicity*, that is, the estimated upper bound of any child node is never smaller than the upper bound of the parent node.

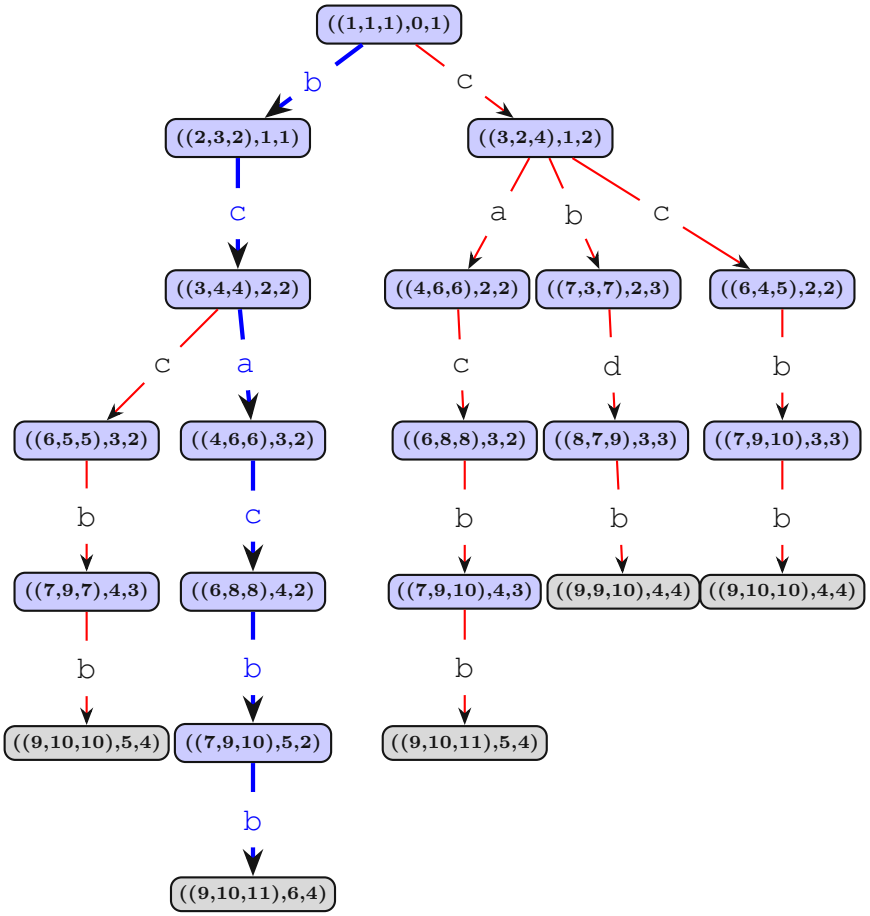


Figure 7.1: State graph for the instance ($\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}, s_3 = \texttt{bbccabcdbba}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). There are five non-extensible sink nodes (shown in gray). The longest path corresponds to the optimal solution $s = \texttt{bcacbb}$ with length six and leads to node $v = (\mathbf{p}^v = (9, 10, 11), l^v = 6, u^v = 4)$ (shown in blue).

7.4 A* Search for the m–CLCS Problem

In this section we establish an efficient A* search to solve CLCS problem. So, in our context the graph to be searched is the acyclic state graph $G = (V, A)$ introduced in the previous section.

The search maintains a list of *open nodes*, which is initialized with the root node, and works in a *best-first-search* manner by *expanding* in each iteration a most promising open node. In order to rank open nodes, A* search makes use of a priority function $f(v) = g(v) + h(v)$, for $v \in V(G)$, where we set $g(v) := l^v$ and $h(v) := \mathrm{UB}(v)$ from the previous section.

In order for the search process to be efficient, our implementation maintains two data

structures: (i) a hash-map N storing all nodes that were encountered during the search, and (ii) the open list $Q \subseteq N$ containing all not yet expanded/treated nodes. More specifically, N is implemented as a nested data structure of sorted lists within a hash map. The position vector $\mathbf{p}^v$ of a node $v = (\mathbf{p}^v, l^v, u^v)$ is mapped to a (linked) 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 $S[\mathbf{p}^v]$ was already encountered during the search, and a quick retrieval of the respective nodes.

Note that it might occur that several nodes representing the same subproblem $S[\mathbf{p}^v]$ are stored, as the following example demonstrates: Consider the problem instance with input strings $s_1 =$ `bacxmnob`, $s_2 =$ `abcxmbno`, and pattern string $P =$ `b`. The A* search might, at some time, encounter node $v_1 = ((4,4),2,1)$ induced by partial solution `bx`, and—at some other time—it might encounter another node $v_2 = ((4,4),3,0)$ 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 $f(v) = l^v + \mathrm{UB}(v)$, 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 $a \in \Sigma_v^{\mathrm{nd}}$). 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 $S[\mathbf{p}^v]$. 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 emphasize that detecting the domination in N_{rel} was, on average, slightly faster when the elements of the lists were sorted in decreasing order w.r.t. their u^v-values. Therefore, we used this ordering in our implementation.

As the upper bound function UB() is *admissible* and *monotonic*, it means that no re-expansion of already expanded nodes become necessary [79] in the A* for the CLCS problem and that the algorithm is 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 25.

7.4.1 Time and Memory Complexity of A* search

In general, an upper bound for the worst-case performance of A* search is $O(b^d)$, where b is the branching factor—which, in our case, is the number of letters—and d is the

Algorithm 25 A* search for the m–CLCS Problem

Input: a problem instance $I = (S, P, \Sigma)$
Output: an optimal CLCS solution
Initialize hash-map N and priority queue Q
Create root node $r = ((1, \ldots, 1), 0, 1)$ and add it to N and Q
while $Q \neq \emptyset$ **do**
    $v \leftarrow Q$.pop()
    Determine Σ_v^{nd} for node v
    if $\Sigma_v^{\text{nd}} = \emptyset$ **then** // a complete node is reached
        Return the solution that corresponds to node v
    else
        for each $c \in \Sigma_v^{\text{nd}}$ **do**
            Generate node v_{ext} by appending c to the part. solution of v
            Retrieve $N_{\text{rel}} \subseteq N$: nodes representing subproblem $S[\mathbf{p}^{v_{\text{ext}}}]$
            insert $\leftarrow$ *true*
            for each $v_{\text{rel}} \in N_{\text{rel}}$ **do**
                if $l^{v_{\text{rel}}} \geq l^{v_{\text{ext}}} \wedge u^{v_{\text{rel}}} \geq u^{v_{\text{ext}}}$ **then**
                    insert $\leftarrow$ *false*
                    break // domination condition is fulfilled
                end if
                if $l^{v_{\text{ext}}} \geq l^{v_{\text{rel}}} \wedge u^{v_{\text{ext}}} \geq u^{v_{rel}}$ **then**
                    Remove v_{rel} from N and Q
                end if
            end for
            if *insert* **then** // new state is non-dominated
                Add v_{ext} to N and Q
            end if
        end for
    end if
end while
return no feasible solution exists

length of an optimal solution. In other words, the runtime of A* search is, in general, exponential. Providing a tighter bound is often hardly possible, as the practical runtime strongly depends on the used guidance heuristic [149]. In practice, however, it frequently happens that A* search, when using a meaningful heuristic, is quite fast, even in those cases in which nothing better than the exponential worst-case run time can be proven. Therefore, respective publications typically focus more on empirically observed run times or indicate the number of expanded/visited nodes, for example, [174].

Nevertheless, it is possible to derive polynomial worst-case time and space complexity bounds for our A* search from Algorithm 25, if n and m are fixed, as follows. The number of visited nodes is bounded by $O(n^m \cdot |P|)$. Since the used upper bound function

is monotonic, we can be sure that no re-expansion of already expanded nodes is necessary, which further implies that the outer while-loop of Algorithm 25 is executed at most $O(n^m \cdot |P|)$ times. The pop() function in Line 6 needs a constant time to retrieve the top node of Q. Afterwards, reorganizing the nodes in the priority queue Q is done in $O(\log |Q|) = O(m \log(n \cdot |P|) = O(m \log n)$ time. Determining the set of non-dominated nodes of a node v is achieved in $O(|\Sigma|^2 \cdot m)$ time by pairwise comparisons. For generating all child nodes of a node v and then checking the domination among the nodes which refer to the same subproblem (Lines 15–23), $O(|\Sigma| \cdot mn \cdot \log n)$ time is required in total. Note that the factor $\log(n)$ reflects the time required to check the domination of a single node, which can be done via binary search. The code in Lines 24–27 executes in $O(\log(n \cdot |P|)) = O(n)$ time. Overall, to execute a single iteration of the main while-loop, we need

Nevertheless, it is possible to derive polynomial worst-case time and space complexity bounds for our A* search from Algorithm 25 as follows. The number of visited nodes is bounded by $O(n^m \cdot |P|)$. Since the used upper bound function is monotonic, we can be sure that no re-expansion of already expanded nodes is necessary, which further implies that the outer while-loop of Algorithm 25 is executed at most $O(n^m \cdot |P|)$ times. The pop() function in Line 6 needs a constant time to retrieve the top node of Q. Afterwards, reorganizing the nodes in the priority queue Q is done in $O(\log |Q|) = O(m \log(n \cdot |P|) = O(m \log n)$ time. Determining the set of non-dominated nodes of a node v is achieved in $O(|\Sigma|^2 \cdot m)$ time by pairwise comparisons. For generating all child nodes of a node v and then checking the domination among the nodes which refer to the same subproblem (Lines 15–23), $O(|\Sigma| \cdot mn \cdot \log n)$ time is required in total. Note that the factor $\log(n)$ reflects the time required to check the domination of a single node, which can be done via binary search. The code in Lines 24–27 executes in $O(\log(n \cdot |P|)) = O(n)$ time. Overall, to execute a single iteration of the main while-loop, we need

$$O(m \log n + |\Sigma| \cdot mn \cdot \log n + |\Sigma|^2 \cdot m + \log(n \cdot |P|)) = \tag{7.2}$$
$$O(|\Sigma| \cdot nm \cdot \log n + |\Sigma|^2 \cdot n) = O(n \cdot |\Sigma| \cdot (m \log n + |\Sigma|)) \tag{7.3}$$

time. For executing the whole algorithm, the time is in

$$O(n \cdot |\Sigma| \cdot (m \log n + |\Sigma|) \cdot O(n^m \cdot |P|) = \tag{7.4}$$
$$O(n^{m+1} \cdot |P| \cdot |\Sigma| \cdot (m \log n + |\Sigma|)). \tag{7.5}$$

Since $|\Sigma|$, in practice, represents a small constant number, the time to execute our A* search is in

$$O(n^{m+1} \cdot |P| \cdot m \log n). \tag{7.6}$$

Concerning the space complexity of the proposed A* algorithm, the worst case corresponds to storing all nodes of the state graph, and is thus in $O(n^m \cdot |P|)$.

7.5 Beam Search for the m–CLCS Problem

It is well known from research on other LCS variants that *beam search* (BS) is often able to produce high-quality approximate solutions in the domain of string problems [55]. For those cases in which our A* approach is not able to deliver an optimal solution in a reasonable computation time, it is quite natural to propose the BS approach which follows from the generalized beam search framework for the LCS problem (see Section 3.3) with slight modifications.

Before we run the BS procedure, Greedy is executed returning an initial solution s_{bsf}. This solution can be served in BS for pruning partial solutions (nodes) that provenly cannot be extended towards a solution better than s_{bsf}. Beam search for the CLCS differs from the GBSF for the LCS problem 14 in the following aspects:

- BS is applied on the search space described in Section 7.3.
- the domination relation is defined as follows. Given $v, v' \in V_{\text{ext}}$, we say in this context that v *dominates* v' iff $p_i^v \leq p_i^{v'}$, for all $i = 1, \ldots, m \wedge u^v \geq u^{v'}$. Note that this is a generalization of the domination relation introduced in [16] for the LCS problem.

Several options for heuristic $h(v)$ in the context of the CLCS problem are detailed in the next section.

7.5.1 Options for the Heuristic Guidance of BS

Different functions can be used as heuristic guidance of the BS, that is, for the function h that evaluates the heuristic goodness of any node $v = (\mathbf{p}^{\text{L},v}, l^v, u^v) \in V$. An obvious choice is, of course, the upper bound UB from Section 3.3.1. Additionally, we consider the following three options.

Probability Based Heuristic. For a probability based heuristic guidance, we make use of a DP recursion from [135] for calculating the probability $\Pr(p, q)$ that any string of length p is a subsequence of a *random string* of length q. These probabilities are computed in a preprocessing step for $p, q = 0, \ldots, n$. This is already used in the context of the LCS problem. Assuming that same assumptions hols as for the LCS problem, we get $\Pr(s \prec S) = \prod_{i=1}^{m} \Pr(p, |s_i|)$.

Given V_{ext} in some construction step of BS, the question is now how to choose the value p common for all nodes $v \in V_{\text{ext}}$ in order to take profit from the above formula in a sensible heuristic manner. For this purpose, we first calculate

$$p^{\min} = \min_{v \in V_{\text{ext}}} \left(|P| - u^v + 1 \right), \tag{7.7}$$

where P is the pattern string of the tackled m–CLCS instance. Note that the string $P[p^{\min}, |P|]$ must appear as a subsequence in all possible completions of all nodes from

$v \in V_{\text{ext}}$, because pattern P must be a subsequence of any feasible solution. Based on $p^{\min}$, the value of p for all $v \in V_{\text{ext}}$ is then heuristically chosen as

$$p = p^{\min} + \min_{v \in V_{\text{ext}}} \left\lfloor \frac{\min_{i=1,\dots,m} \{|s_i| - p_i^v + 1\} - p^{\min}}{|\Sigma|} \right\rfloor . \tag{7.8}$$

The intention here is, first, to let the characters from $P[p^{\min}, |P|]$ fully count, because they will—as mentioned above—appear for sure in any possible extension. This explains the first term ($p^{\min}$) in Eq. (7.8). The second term is justified by the fact that an optimal m–CLCS solution becomes shorter if the alphabet size becomes larger. Moreover, the solution tends to be longer for nodes v whose length of the shortest remaining string from $S[\mathbf{p}^v]$ is longer than the one of other nodes. We emphasize that this is a heuristic choice which might be improvable. If p would be zero, we set it to one in order to break ties. The final probability-based heuristic for evaluating a node $v \in V_{\text{ext}}$ is then

$$H(v) = \prod_{i=1}^{m} \Pr(p, |s_i| - p_i^v + 1), \tag{7.9}$$

and those nodes with a larger H–value are preferred.

Expected Length Based Heuristic. In Section 3.3.2 we derived an approximate formula for the expected length of a longest common subsequence of a set of uniform random strings. Before we extend this result to the m–CLCS problem, we state those aspects of the results from this section that are needed for this purpose.

In particular, for the LCS problem we know that

$$\mathbb{E}[Y] = \sum_{k=1}^{l_{\min}} \mathbb{E}[Y_k], \tag{7.10}$$

where $l_{\min} := \min\{|s_i| \mid i = 1, \dots, m\}$, Y is a random variable for the length of an LCS, and Y_k is, for any $k = 1, \dots, l_{\min}$, a binary random variable indicating whether or not there is an LCS with a length of at least k.

In the context of the m–CLCS problem, a similar formula with the following re-definition of the binary variables is used. Y is now a random variable for the length of an LCS that has pattern string P as a subsequence, and the Y_k are binary random variables indicating whether or not there is an LCS with a length of at least k having P as a subsequence. If we assume the existence of at least one feasible solution, instead of formula (7.10) we get $\mathbb{E}[Y] = |P| + \sum_{k=|P|+1}^{l_{\min}} \mathbb{E}[Y_k]$.

For $k = |P|, \dots, l_{\min}$, let T_k be the set of all possible strings of length k over alphabet Σ. Clearly, there are $|\Sigma|^k$ such strings. For each $s \in T_k$ we define the event Ev_s that s is a subsequence of all input strings from S having P as a subsequence. For simplicity, we assume the independence among events Ev_s and $\text{Ev}_{s'}$, for any $s, s' \in T_k$, $s \neq s'$. With this assumption, the probability that string $s \in T_k$ is a subsequence of all input

strings from S is equal to $\prod_{i=1}^{m} \Pr(|s|, |s_i|)$. Further, under the assumption that (i) s is a uniform random string and (ii) the probabilities that s is a subsequence of s_i (denoted by $\Pr(s \prec s_i)$) for $i = 1, \ldots, m$, and the probability that P is a subsequence of s (denoted by $(\Pr(P \prec s))$ are independent, it follows that the probability $P^{\text{CLCS}}(s, S, P)$ that s is a common subsequence of all strings from S having pattern P as a subsequence is equal to $\Pr(|P|, k) \cdot \prod_{i=1}^{m} \Pr(k, |s_i|)$. Moreover, note that, under our assumptions, it holds that $\Pr(P \prec s') = \Pr(P \prec s'') = \Pr(|P|, k)$, for any pair of sampled strings $s', s'' \in T_k$. Therefore, it follows that

$$\begin{aligned} \mathbb{E}[Y_k] &= 1 - \prod_{s \in T_k} \Big(1 - P^{\text{CLCS}}(s, S, P)\Big) \\ &= 1 - \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i|)\right) \cdot \Pr(|P|, k)\right)^{|\Sigma|^k} . \end{aligned} \tag{7.11}$$

Using this result, the expected length of a final m–CLCS solution that includes a string inducing node $v \in V$ as a prefix can be approximated by the following (heuristic) expression:

$$\begin{aligned} &\text{EX}^{\text{CLCS}}(v) \overset{7.10,7.11}{=} |P| - u^v + (l_{\min} - (|P| - u^v + 1) + 1) - \\ &\sum_{k=|P|-u^v+1}^{l_{\min}} \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i| - p_i^{\text{L},v} + 1)\right) \cdot \Pr(|P| - u^v, k)\right)^{|\Sigma|^k} = \\ &= l^v_{\min} - \sum_{k=|P|-u^v+1}^{l^v_{\min}} \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i| - p_i^{\text{L},v} + 1)\right) \cdot \Pr(|P| - u^v, k)\right)^{|\Sigma|^k} , \end{aligned} \tag{7.12}$$

where $l^v_{\min} = \min\{|s_i| - p_i^{\text{L},v} + 1 \mid i = 1, \ldots, m\}$. To calculate this value in practice, one has to take care of numerical issues, in particular the large power value $|\Sigma|^k$. We resolve it in the same way as in Section 4.4 by applying a Taylor series expansion.

Pattern Ratio Heuristic. So far we have introduced three options for the heuristic function in beam search: the upper bound (Section 7.3.1), the probability based heuristic and the expected length based heuristic (Section 7.5.1). With the intention to test, in comparison, a much simpler measure we introduce in the following the pattern ratio heuristic that only depends on the length of the shortest string in $S[\mathbf{p}^v]$ and the length of the remaining part of the pattern string to be covered ($|P| - u^v + 1$). In fact, we might directly use the following function for estimating the goodness of any $v \in V$:

$$R(v) := \frac{\min_{i=1,\ldots,m}(|s_i| - p_i^v + 1)}{|P| - u^v + 1}. \tag{7.13}$$

In general, the larger $R(v)$, the more preferable should be v. However, note that the direct use of (7.13) generates numerous ties. In order to avoid a large number of ties,

instead of $R(v)$ we use the well-known k-norm

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

with some $k > 0$. Again, nodes $v \in V$ with a larger $||\cdot||_k$-values are preferable. In our experiments, we set $k = 2$ (Euclidean norm).

7.6 Experimental Evaluation

All algorithms were implemented in C++ with g++ 7.4 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 one hour. The source code of this project is accessible at `https://www.ac.tuwien.ac.at/files/resources/software/clcs.zip`.

7.6.1 Benchmark Instances

Concerning the general m–CLCS problem, for each combination of the number of input strings $m \in \{10, 50, 100\}$, the length of input strings $n \in \{100, 500, 1000\}$, the alphabet size $|\Sigma| \in \{4, 20\}$ and the ratio $p' = \frac{|P|}{n} \in \left\{ \frac{1}{50}, \frac{1}{20}, \frac{1}{10}, \frac{1}{4}, \frac{1}{2} \right\}$, ten instances were created, each one as follows. The following procedure was used for generating each instances. First, a pattern string P was created uniformly at random, that is, each character from Σ has an equal chance to be chosen for each position of P. Second, two input strings of equal length n were generated as follows. First, $|P|$ different positions were randomly chosen in each input string. Then, characters $P[1], \ldots, P[|P|]$ are placed (in this order) from left to right at these positions. Finally, the remaining characters of each input string were set to letters chosen uniformly at random from the alphabet Σ. This procedure ensures that at least one feasible CLCS solution exists for each benchmark instances. The benchmarks are available at `https://www.ac.tuwien.ac.at/files/resources/instances/m-clcs.zip`. Overall, we thus created and use 900 benchmark instances.

Concerning the specific 2–CLCS problem, for each combination of $n \in \{100, 500, 1000\}$ (length of the input strings), $|\Sigma| \in \{4, 12, 20\}$ (alphabet size), $p' = \frac{|P|}{n} \in \left\{ \frac{1}{50}, \frac{1}{20}, \frac{1}{10}, \frac{1}{4}, \frac{1}{2} \right\}$ (length of the pattern string), ten problem instances were randomly generated. This results in a total of 450 instances. Unfortunately, none of the artificial benchmarks from [50] and [86] were provided to us, although the respective authors were contacted with this concern. These benchmark instances for the 2–CLS problem are available at `https://www.ac.tuwien.ac.at/files/resources/instances/clcs/2d-clcs.zip`

In addition to these artificially generated instances, we use a benchmark suite from [50] based on strings representing real biological sequences[1]. This benchmark set is henceforth

[1] Available at `http://sun.aei.polsl.pl/~sdeor/pub/do09-ds.zip`.

Table 7.1: Benchmark suite `Real` from [50].

data set	number of sequences	sequence length (min, med, max)	$\|\Sigma\|$	origin
$ds0$	7	(111, 124, 134)	20	[35]
$ds1$	6	(124, 149, 185)	20	[35]
$ds2$	6	(131, 142, 160)	20	[35]
$ds3$	5	(189, 277, 327)	20	[35]
$ds4$	6	(98, 114, 123)	20	[126]

called `Real`. It has its origins in experimental studies on the constrained multiple sequence alignment (CMSA) problem considered in [35, 126]. Each possible pair of sequences from this data set, together with a pattern string, was used in [50] to define a problem instance for the CLCS problem. Properties of the input strings, together with their origins, are provided in Table 7.1. In particular, Chin et al. [35] provided four sets of strings containing RNase sequences with lengths from 111 to 327. In contrast, set $ds4$—containing aspartic acid protease family sequences—was provided by Lu and Huang [126], also in the context of the CMSA problem. Overall, benchmark set `Real` consists of 121 problem instances.

7.6.2 Computational Studies for the general m–CLCS Problem

We include the following six algorithms (resp. algorithm variants) in our comparison: (*i*) the approximation algorithm from [74] (Approx), (*ii*) Greedy from Section 7.2, and (*iii*) the four beam search configurations differing only in the heuristic guidance function. These BS versions are denoted as follows. Bs-Ub refers to BS using the upper bound, Bs-Prob refers to the use of the probability based heuristic, Bs-Ex to the use of expected length based heuristic, and BS-Pat to the use of the pattern ratio heuristic. Moreover, we include the information of how many instances of each type were solved to optimality by the exact A* search. Concerning the beam search, parameters β (the maximum number of nodes kept for the next iteration) and k_{best} (the extent of filtering) are crucial for obtaining good results. Tuning of the parameters of different BS configuration was presented in Appendix D.2. The results of tuning indicated that selecting $\beta = 2000$ and $k_{\text{best}} = 100$ for all of our algorithms seems like a reasonable choice. Moreover, the tuning report indicated that function upper bound based pruning is indeed beneficial.

Table 7.2 reports results for the instances with $p' = \frac{1}{20}$ and Table 7.3 those for the instances with $p' = \frac{1}{4}$. The remaining numerical results are reported in Appendix D. The first three columns of each table indicate the instance characteristics. Then, for the six competitors we provide in each table row the obtained solution quality and computation time averaged over the 10 instances with the respective characteristics. The best result of each table row is shown in bold. Finally, for A* search we provide in each table row the number of instances solved to optimality (out of 10) and the average runtime required to

Table 7.2: Results for instances with $p' = \frac{|P|}{n} = \frac{1}{20}$.

			Approx		Greedy		Bs-Ub		Bs-Prob		Bs-Ex		Bs-Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	#	$\bar{t}[s]$
4	10	100	21.4	< 0.1	30.8	< 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	162.3	< 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	330.9	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	21.3	< 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	127.1	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	0.3	281	226.4	290.1	267.6	**291.3**	289.4	273	366.4	0	-
4	100	100	17.6	< 0.1	18.5	< 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	119.5	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	248	0.9	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	< 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	0.1	***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	58.2	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	0.2	***27.3**	0.3	***27.3**	0.2	***27.3**	0.3	***27.3**	0.3	10	0.3
20	100	1000	52	1	54.7	0.8	57.2	14	***57.3**	13.6	***57.3**	9.4	56.4	17.7	10	86.0

Table 7.3: Results for instances with $p' = \frac{|P|}{n} = \frac{1}{4}$.

			Approx		Greedy		Bs-Ub		Bs-Prob		Bs-Ex		Bs-Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}[s][s]$	$\overline{\|s\|}$	$\bar{t}[s][s]$	$\overline{\|s\|}$	$\bar{t}[s][s]$	$\overline{\|s\|}$	$\bar{t}[s][s]$	$\overline{\|s\|}$	$\bar{t}[s][s]$	$\overline{\|s\|}$	$\bar{t}[s][s]$	#	$\bar{t}[s][s]$
4	10	100	28.6	< 0.1	32.2	< 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	160.4	< 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	317.4	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	0.3	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	0.9	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**	0.5	***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**	0.2	***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**	0.8	***250.0**	1.1	***250.0**	1.1	***250.0**	0.8	***250.0**	1.1	10	1.0

do so. A preceding asterisk indicates that the respective result is provenly optimal. The results allow to make the following observations.

- The m–CLCS problem tends to be most difficult to solve for short pattern strings—that is, low values of $|P|$—and for small alphabet sizes. With growing $|P|$ and $|\Sigma|$, the problem becomes easier. On the one side, this is indicated by the results of the A* search. When $\frac{|P|}{n} = 1/20$ and $|\Sigma| = 4$, A* can only solve three problem instances to optimality. When moving to instances with $|\Sigma| = 20$, A* search can already solve 70 instances to optimality. The corresponding numbers for instances with $\frac{|P|}{n} = 1/4$ are 31 (for $|\Sigma| = 4$) and 90 (for $|\Sigma| = 20$). In fact, in this last case 90 corresponds to all problem instances of this type. On the other side, the decreasing problem difficulty for growing $|P|$ and $|\Sigma|$ is also indicated by the

differences between the results of the heuristic algorithms. In fact, for $\frac{|P|}{n} = 1/20$ and $|\Sigma| = 20$ all algorithms are able to solve all 90 problem instances to optimality.

- The reason for the problem difficulty to decrease with growing $|P|$ can be explained as follows. With growing $|P|$, the similarity between the input strings also grows. This results in a decrease of the search space size. Moreover, from [55] we know that EX-type guidance for BS becomes worse with a growing similarity of the input strings. And, in fact, this observation holds also in the case of the m–CLCS problem. For $\frac{|P|}{n} = 1/20$, Bs-Ex delivers in most cases better results than the other BS configurations. However, Bs-Ex seems to lose efficiency for $\frac{|P|}{n} = 1/4$ where Bs-Prob is generally the better choice.
- All our heuristic algorithms significantly improve over Approx, which is the only existing technique from the literature. This also holds for Greedy, which requires approximately the same computation time as Approx. Only when instances are easy to solve—that is, when n and m are small, and $|\Sigma|$ and $|P|$ are rather large—Approx is competitive with our algorithms.
- All versions of BS improve over Greedy. However, this comes at the price of significantly elevated computation times.
- Bs-Pat, which uses the most simplistic guidance heuristic, is clearly inferior to the other three BS variants in terms of solution quality for almost all instance types.
- The results of Bs-Ub are comparable with those of Bs-Ex only when $|P|$ and n are both small. This is because the upper bound is especially tight for smaller instances.

Finally, we want to shed some more light on the comparison of the heuristic techniques with A* search. For this purpose, from now on we only consider those instances that can be solved to optimality by A* search. The two plots in Fig. 7.2 show for each heuristic algorithm and each value of $\frac{|P|}{n}$ (x-axis) the average fraction (in percent) of the length of heuristic solutions in respect to the length of the A* search solutions. A dot at 100% means that the length of the heuristic solution matches the length of the optimal solution from A* search. The plots show, in particular, that Bs-Prob, Bs-Ub and Bs-Ex always reach at least a value of 98%. Complementary, the two plots in Fig. 7.3 show for each heuristic algorithm and for each value of $\frac{|P|}{n}$ (x-axis) the percentage of instances that were solved to optimality. Bs-Prob fails to deliver an optimal solution for only one instance of type $m = 10$, $n = 500$, $|\Sigma| = 20$, and $\frac{|P|}{n} = 1/20$.

7.6.3 Computational Results for 2-CLCS Problem

We aimed to re-implement all algorithms from the literature in the way in which they are described in the original articles as the respective code could not be obtained. In a few cases, due to a lack of sufficient details, we had to make our own specific implementation decisions. This was, in particular, the case for the algorithm of Iliopoulos and Rahman

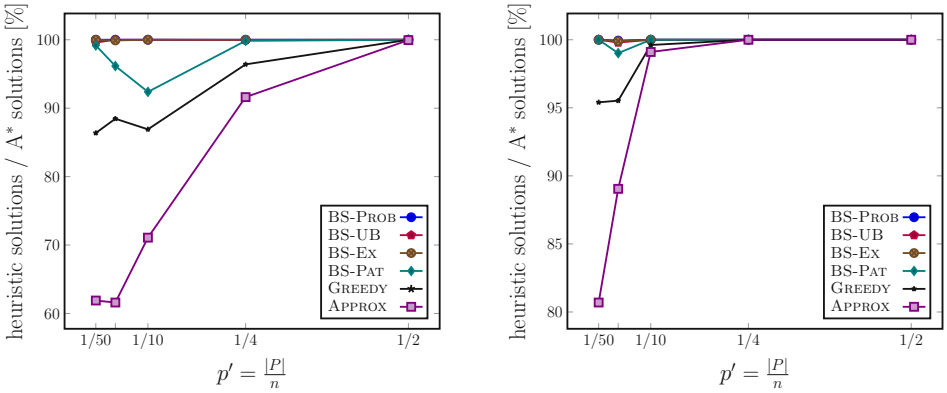


Figure 7.2: Average fraction (in percent) of the length of heuristic solutions with respect to the length of the A* solutions.

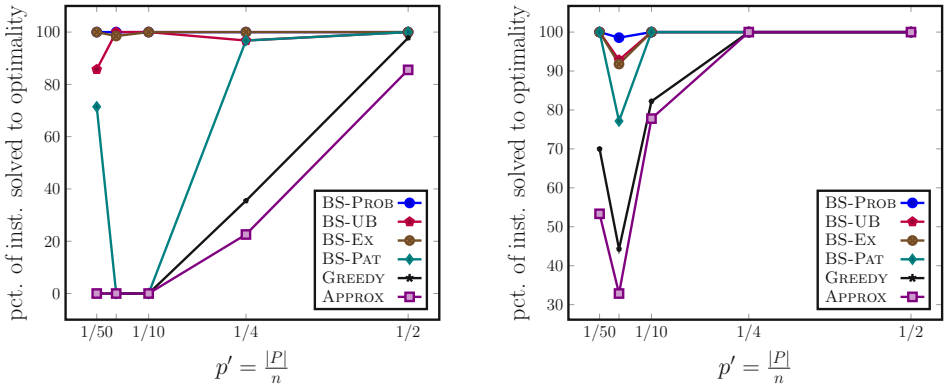


Figure 7.3: Percentage of instances solved to optimality.

[88]: The *bounded heap* data structure has to be initialized for different indices, and it remains unclear how this can be done efficiently. The authors were contacted with this issue but we did not receive a response. Our implementation creates a new *bounded heap* for a new index by copying the content from the *bounded heap* of the 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. Unfortunately, the original article does not contain any computational study that could serve as a comparison but just focuses on asymptotic runtimes from a theoretical point-of-view. A short overview over the algorithms from the literature can be found in Appendix D.

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

We compare our A* search from Section 7.4 with our re-implementations of the following state-of-the-art algorithms from the literature.

- `Chin`: Algorithm by Chin et al. [34];

Table 7.4: Instances with $p' = \frac{|P|}{n} = \frac{1}{50}$: Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin[s]	Deo[s]	AE[s]	IR[s]	Hung[s]	A*[s]
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**
12	100	40.1	**< 0.1**	0.1	**< 0.1**	0.1	**< 0.1**	**< 0.1**
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**
20	100	33.5	**< 0.1**	0.1	**< 0.1**	**< 0.1**	**< 0.1**	**< 0.1**
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**

- `Deo`: Algorithm by Deorowicz [49];
- `AE`: Algorithm by Arslan and Eğecioğlu [5];
- `IR`: Algorithm by Iliopoulos and Rahman [88];
- `Hung`: Algorithm by Hung et al. [86].

In general, all algorithms could find optimal solutions and prove their optimality for all instances. However, the required runtimes differ sometimes substantially. Tables 7.4–7.9 show these runtimes for each re-implemented algorithm as well as our A* search 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 7.9. For each instance group (line), the lowest runtimes among the competing algorithms are displayed 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 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.
- The first algorithm that starts losing efficiency with growing input string length is `IR`. Already starting with $n = 500$, the computation times start to grow substantially in comparison to the other approaches, which is most likely due to the complexity of the utilized data structures. We remark that our specific implementation decision concerning the initialization of the *bounded heap* may have a significant impact, as mentioned already in Section D.1.

Table 7.5: Instances with $p' = \frac{|P|}{n} = \frac{1}{20}$: Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*
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**
12	100	41.0	**<0.1**	0.1	**<0.1**	0.1	**<0.1**	**<0.1**
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**
20	100	32.2	**<0.1**	**<0.1**	**<0.1**	**<0.1**	**<0.1**	**<0.1**
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**

Table 7.6: Instances with $p' = \frac{|P|}{n} = \frac{1}{10}$: : Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*
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**
12	100	40.5	**<0.1**	0.1	**<0.1**	0.1	**<0.1**	**<0.1**
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**
20	100	31.1	**<0.1**	**<0.1**	**<0.1**	**<0.1**	**<0.1**	**<0.1**
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**

Table 7.7: Instances with $p' = \frac{|P|}{n} = \frac{1}{4}$: Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*
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**
12	100	39.9	**<0.1**	0.1	**<0.1**	0.1	**<0.1**	**<0.1**
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**
20	100	35.7	**<0.1**	**<0.1**	**<0.1**	0.1	**<0.1**	**<0.1**
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**

- Algorithm Chin clearly outperforms Deo when $|\Sigma|$ is small. With growing $|\Sigma|$, as already noticed in earlier studies [49], Deo becomes more efficient. In fact, the two approaches perform similarly for $|\Sigma| = 20$. The advantages of Deo over Chin are

Table 7.8: Instances with $p' = \frac{|P|}{n} = \frac{1}{2}$: Average runtimes in seconds.

$\|\Sigma\|$	n	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*
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**
12	100	54.6	0.1	**<0.1**	**<0.1**	0.1	**<0.1**	**<0.1**
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**
20	100	53.0	**<0.1**	0.1	**<0.1**	0.1	**<0.1**	**<0.1**
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**

Table 7.9: Benchmark set Real: Average runtimes in seconds.

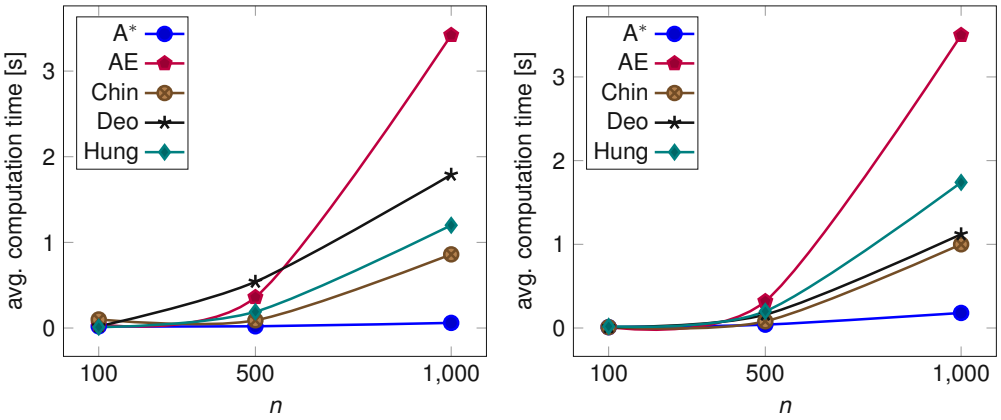
data set	P	$\overline{\|s\|}$	Chin	Deo	AE	IR	Hung	A*
*ds*0	HKH	60.62	0.012	0.015	0.012	0.026	0.017	**0.011**
*ds*1	HKH	64.00	**0.012**	0.017	0.013	0.032	0.019	0.015
*ds*1	HKSH	63.93	**0.011**	0.021	0.017	0.033	0.017	**0.011**
*ds*1	HKSTH	63.87	0.016	0.022	0.019	0.043	0.024	**0.012**
*ds*2	HKSH	79.60	0.015	0.020	0.016	0.030	0.052	**0.012**
*ds*2	HKSTH	77.87	**0.013**	0.018	0.016	0.030	0.051	**0.013**
*ds*3	HKH	103.90	0.018	0.026	0.019	0.138	0.188	**0.014**
*ds*4	DGGG	43.87	**0.012**	0.022	0.014	0.023	0.049	**0.012**

noticed in particular for higher p'; see Table 7.7.

- Algorithm Hung generally performs better than Deo and Chin. This confirms the conclusions from the computational study in Hung et al. [86].
- 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.
- A general conclusion for the artificial benchmark set is that A* search is in most cases about one order of magnitude faster than Hung, which is overall the second-best approach.
- Concerning the results for benchmark set Real (see Table 7.9), 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 7.4 shows the influence of the instance length on the algorithms' runtimes for $|\Sigma| = 4$ and $|\Sigma| = 20$. Note that IR is not included here since it was 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 7.5 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 degrades with increasing $|P|$.
- We emphasize that our A* search is also executed on the harder instances with $n \in \{2000, 5000\}$. The results are presented and discussed in the master thesis [10].

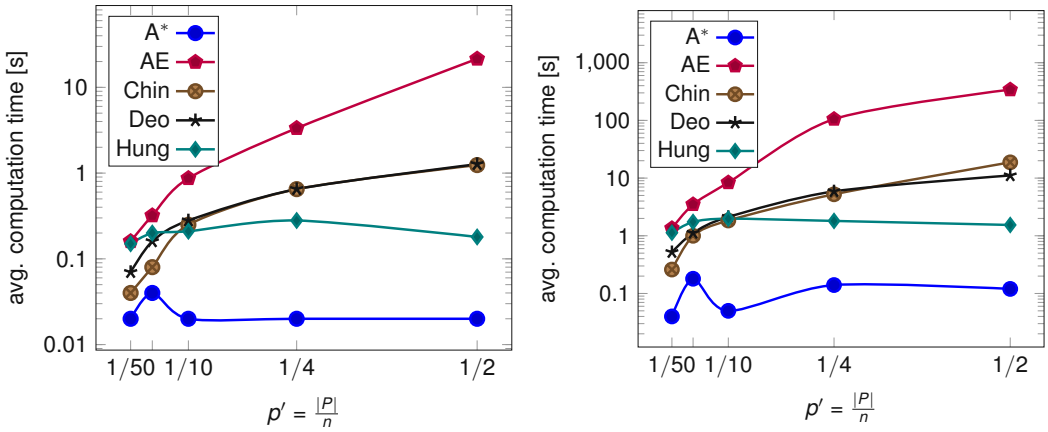
Finally, we also compare the amount of work done by the algorithms in order to reach optimal solutions. In the case of A*, this amount of work is measured by the number of generated nodes of the state graph. In the case of `Deo`, this refers to the number of different keys (i, j, k) generated during 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 Fig. 7.6 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 $(S = \{s_1, s_2\}, P, \Sigma)$. 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 Fig. 7.6 and $|\Sigma| = 4$ from $p' \geq \frac{1}{4}$ onward and $|\Sigma| = 20$ from $p' \geq \frac{1}{20}$ onward.



(a) $|\Sigma| = 4$ (b) $|\Sigma| = 20$

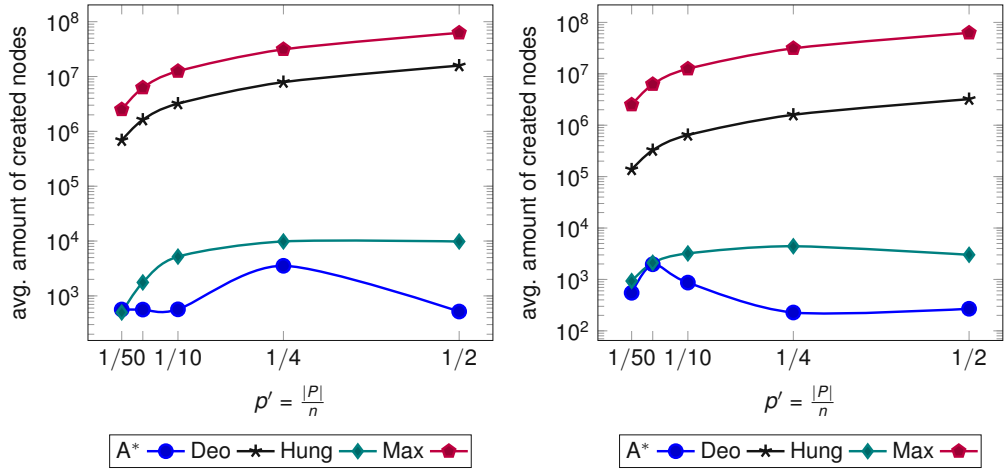
Figure 7.4: Average computation times of the algorithms for $p' = \frac{1}{20}$.

From a more practical point of view, our results give the impression that the more misleading the heuristic function used by our A* for a specific problem instance is,



(a) $n = 500$ (b) $n = 500$

Figure 7.5: Average computation times of the algorithms for $|\Sigma| = 20$.



(a) $|\Sigma| = 4$ (b) $|\Sigma| = 20$

Figure 7.6: Average amount of created nodes by the algorithms for $n = 500$.

the higher will be its running time. More specifically, the heuristic employed in our A* search seems more misleading when the input strings are rather similar. In order to verify this impression, we conducted an additional set of experiments. First, we generated an additional set of problem instances with different degrees of similarity in the input strings. For example, the similarity of $\theta = 0.3$ means that, on average, about 30% of the positions in the two input strings have the same character. We generated 10 problem instances with input string length $n = 100$ for each similarity degree $\theta \in \{0.1, 0.2, 0.5, 0.8, 0.9\}$ and an alphabet size of $|\Sigma| = 12$. Moreover, the same random pattern string $P =$ `abbbcbcbdb` was used for all instances. The running times of our A* algorithms are shown in comparison to algorithm `Chin` in Table 7.10. Results

Table 7.10: Results for instances with different degrees of similarity (θ) of the input strings. The similarity of the input strings grows with an increasing value of θ.

θ	$\overline{\lvert s \rvert}$	Chin	A*
0.1	41.3	0.060	**0.050**
0.2	43.8	0.070	**0.050**
0.5	55.0	0.061	**0.052**
0.8	73.2	**0.050**	0.055
0.9	82.5	**0.050**	0.075

indeed confirm our observation from above. That is, when the degree of similarity is rather low, our A* algorithm is faster (see the results for $\theta \in \{0.1, 0.2, 0.5\}$. On the other side, when the degree of similarity is rather high ($\theta \in \{0.8, 0.9\}$, Chin is faster. This is because, in the case of instances with a rather high θ-value, a significant amount of time of the overall running time of A* is spent to calculate the upper bound values of the generated nodes.

However, as shown in our experimental evaluation, A* can be expected to outperform the competitor algorithms in most cases, especially harder ones.

7.7 Conclusions

In this chapter, we studied the generalized constrained longest common subsequence problem which is $\mathcal{NP}$–hard. Apart from a simple greedy heuristic, we also introduced four different variants of beam search that differ in the heuristic guidance used for selecting the partial solutions to be further expanded in the subsequent iteration. More specifically, we considered an upper bound, a probability-based heuristic, an expected length based heuristic, and a simple greedy criterion. Our approaches are compared to the approximation algorithm from the literature, the only one so far available for the problem. In general, the BS variant using the expected length calculation heuristic performs best when the pattern string is rather short, while the BS variant with the probability-based heuristic is leading when the pattern string is longer. Moreover, instances become more easy to solve the longer the pattern is. For the exact solving, we presented an exact A* search algorithm.

Concerning the 2–CLCS problem, the literature offers algorithms based on dynamic programming as well as sparse approaches. The effectivity of A* to solve the 2–CLCS problem was demonstrated by comparing it to several other so far leading algorithms from the literature. The A* search was able to solve all artificially generated benchmarks as well as the real benchmark instances in a fraction of a second. More specifically, the running times required by A* are about an order of magnitude smaller than those of the second-best algorithm. Interestingly, the performance of A* does not degrade much with an increase in the instance size, which is not the case for the other algorithms from the

literature. We conclude that A* search is a tool that has a great potential to be used to study similarities between sequences. In fact, our A* search is the new state-of-the-art method for the 2–CLCS problem.

CHAPTER 8

Conclusions and Future Work

In this thesis, we provide various exact and heuristic methods to solve the longest common subsequence (LCS) problem and several variants thereof which arise in practice. The basic LCS problem seeks a maximal subsequence which is common for all strings from a set of input strings. It is of high practical relevance in particular since it provides a basic measure of similarity between molecular structures. Finding relations between molecules plays an important role in understanding complex biological processes that relate to the structure of the molecules. Specific relations and specific structures of molecules ask for setting up different measures of similarity such as the longest common palindromic subsequence problem, the longest common square subsequence problem, the arc-preserving longest common subsequence problem, the constrained longest common subsequence problem and the repetition-free longest common subsequence problem, among others.

First, for each of the considered problems, general search frameworks have been introduced as a basis to develop more advanced search techniques. We developed (*i*) greedy heuristics to obtaining solutions of reasonable quality within a short time, and different (*ii*) exact, and (*iii*) heuristic search procedures. Concerning heuristic solving, a generalized BS framework (GBSF) has been described. Concerning exact solving, two kinds of methods are developed (*i*) a pure exact A* search, and (*ii*) two novel A*–based anytime algorithms: A*+BS and A*+ACS.

GBSF employs special pruning and filtering procedures for omitting suboptimal and dominated nodes, respectively. In order to guide the search towards promising regions, a new state-of-the-art heuristic guidance has been introduced. The heuristic is based on an approximation of the expected length of an LCS supposing randomness of the strings in the input. Based on this formula, the expected length calculation heuristic is further extended towards the LCPS and CLCS problem, also guiding the search towards more promising regions than other heuristics known from the literature. Moreover, for the

CLCS a new probability-based heuristic is developed which is an extension of such a heuristic for the LCS problem.

Concerning the heuristic results on the three considered problems (LCS, LCPS, CLCS), the following main conclusions can be drawn from our experimental evaluations:

- For the LCS problem, GBSF guided by the expected length calculation heuristic is currently the leading method on the class of quasi-independent instances. When there is high similarity between input strings, a new tight upper bound is proposed as guidance and GBSF performs best in this case.
- For the LCPS problem, essentially the same conclusion is drawn as for the LCS problem.
- For the CLCS problem, the GBSF guided by the expected length calculation heuristic for the CLCS problem performs best when pattern strings are short w.r.t. the length of longest input strings, whereas the probability-based heuristic provides state-of-the-art results when the pattern string is longer.

Concerning the heuristic approaches to solve the longest common square subsequence (LCSqS) problem, we transform an instance into a series of LCS problem instances, which are then solved. Two heuristic approaches are developed in this thesis: (*i*) a randomized local search, and (*ii*) a hybrid of a (reduced) variable neighborhood search and a beam search for the LCS problem. The beam search component of the latter is equipped to find a high-quality LCSqS solution while the (reduced) variable neighborhood search component ensures a reasonable diversification of the search. Our experiments show that the latter hybrid is able to deliver best LCSqS solutions for middle–to–large benchmark instances when the BS is guided by the expected length calculation heuristic for LCS problem.

Concerning exact solving, we proposed an A* search framework and applied it efficiently on the LCS, LCPS and CLCS problems. The tightest known upper bounds for the LCS from the literature are utilized for the LCS and CLCS problem, whereas a new upper bound is proposed and utilized in the A* search for the LCPS problem. The following conclusions can be drawn from our experimental evaluations:

- For the LCS problem, our A* search could solve small-sized instances (up to the strings' lengths of 100) in a fraction of a second. It outperforms the state-of-the-art exact approach from the literature in time, memory, and the number of instances solved to optimality.
- For the LCPS problem, our A* search is the first approach in the literature proposed for the general problem variant with arbitrary many input strings. The method shows its efficiency in particular on small-sized instances up to string lengths of 100. Moreover, A* is applied on the well–studied variant of the LCPS problem with two

input strings and compared to other state-of-the-art approaches and a CP approach. The experiments show that our A* search is the only approach able to solve all the instances within the given time and memory. Running times of A* search are an order of magnitude shorter than the times of the second-best approach.

- For the CLCS problem, A* search is also the first exact approach proposed for the general problem variant with more than two strings. The performance of A* is highly dependent on the length of the pattern string. The longer it is, the easier the problem is to solve. In general, when the ratio between the pattern's length and the longest input string is about 0.2, A* needs a fraction of a second to solve any instances considered in our experimental evaluation. If the ratio is less than 0.1, small-sized instances (up to the strings' length of 100) are solved by A* search. Moreover, A* search is applied on the well-studied CLCS problem with two input strings and has been shown that it can outperform each of five former state-of-the-art approaches from the literature.

The A* search framework was able to tackle small-sized problem instances exactly. In order to further improve the performance of our A* search, we turned the search into an anytime search such that powerful heuristic search techniques are interleaved with classical A* search iterations. In this thesis, two such algorithms are investigated: (*i*) A*+BS where A* iterations are combined with a GBSF, and (*ii*) A*+ACS where A* iterations are interlined within a major iteration of the anytime column search. Iterations of ACS are guided by the expected length calculation heuristic of the considered problem. The performance of the two algorithms is studied on the LCS and LCPS problems. The following conclusions can be drawn from our experimental these evaluations:

- For the LCS problem, A*+ACS performs better than A*+BS and a few other state-of-the-art anytime algorithms. The convergence towards excellent solutions of A*+ACS is much faster than the convergence of the other anytime algorithms. Obtained heuristic results are new state-of-the-art results for the LCS problem on the quasi-independent benchmark sets. Also, the obtained final gaps of A*+ACS are in general smaller than those of the other state-of-the-art anytime approaches from the literature as well as A*+BS.

- For the LCPS problem, similar conclusions can be drawn as in the case of the LCS problem.

In addition to the above techniques, an alternative exact approach was proposed for solving the LCS problems based on the transformation of a problem instance to an instance of the maximum clique problem (also called conflict graphs). It turned out that this transformation and successive solving of the maximum clique problem is highly beneficial for, in particular, the repetition-free longest common subsequence (RFLCS) problem and the longest arc-preserving common subsequence (LAPCS) problem. However,

for large-sized problem instances, the corresponding conflict graphs become huge and, therefore, we proposed a reduction technique based on the best available lower and upper bounds from the literature. This reduction is highly effective for the RFLCS problem. More than 90% of the considered instances were solved to proven optimality by applying the general-purpose solver Cplex, outperforming the best exact and heuristic MC solvers from the literature. For the LAPCS problem, seven out of ten real-world benchmark instances could be solved for the first time in literature by using the reduction.

Concerning future work on the considered problem:

- In the case of the classical LCS problem, it would be interesting to compare A* search to the state-of-the-art approaches from the literature which are specially designed for two strings only.
- In the case of the CLCS problem, anytime algorithms would also be interesting to study in order to obtain optimality gaps on the large-sized problem instances. Moreover, the case of the CLCS problem with arbitrary many pattern strings seems interesting to consider since in biology it makes sense to consider more than just one putative pattern string in RNA molecules, see [130].
- In the case of the LCSqS problem, we are not aware of any exact algorithm in the literature. A possible extension of our general A* search framework seems to be a promising option;
- In the case of the RFLCS problem, recently, multi-valued decision diagrams (MDDs) are shown to be a strong approach to solve the RFLCS problem [82], especially to further reduce the size of the conflict graphs. In the literature, embedding the (relaxed) MDDs into B&B was shown to be a promising technique [40]. Therefore, B&B combined with the MDDs to solve remaining unsolved instances of the problem would be a promising direction. Considering anytime algorithms to solve large–sized instances is another promising option.
- In the case of the LAPCS problem, it gets much harder to come up with some reasonable general search framework due to the specificity of the problem. Constructing a reasonable state graph structure is a crucial starting point.
- In the case of the LCPS, CLCS and LCSqS problems, the instances considered in the experiments were all randomly generated. One should study the performance of the constructed algorithms on real-world benchmark sets.
- Last but not least, studying other variants of the LCS problem such as the doubly-constrained LCS [22] problem, the restricted LCS [73] problem, and the longest common increasing subsequence problem [178] is also an important research direction.

APPENDIX A

LCS Problem: Supplementary Material

In Appendix A we provide an additional material to the studies described in Chapter 3.

A.1 The Full Anytime Results

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$
4	10	600	214	41.8	2.52	900.0	***223**	39.2	548.2	900.0	214	41.0	2.2	900.0	***223**	38.6	173.6	681.1	221
4	15	600	198	46.5	35.8	624.91	***206**	44.5	16.0	900.0	199	45.6	892.9	900.0	205	44.1	140.4	596.2	204
4	20	600	189	48.4	687.7	900.0	***195**	46.7	130.0	900.0	188	48.1	245.0	900.0	194	46.6	158.7	686.9	193
4	25	600	183	49.9	357.7	900.0	***189**	48.2	26.7	900.0	182	49.4	7.7	900.0	187	48.2	110.4	627.1	187
4	40	600	170	53.4	234.6	796.0	***177**	51.5	457.5	900.0	170	52.9	110.3	900.0	174	51.7	585.8	718.9	175
4	60	600	162	55.2	8.6	900.0	***169**	53.3	275.9	900.0	162	54.7	7.9	900.0	166	53.6	448.3	705.0	168
4	80	600	158	56.5	112.4	900.0	***164**	54.8	337.6	900.0	158	56.1	67.3	900.0	159	55.7	9.1	765.3	163
4	100	600	155	57.4	170.6	900.0	***161**	55.6	735.4	900.0	155	56.8	70.9	900.0	157	56.3	24.1	832.1	159
4	150	600	149	58.9	19.7	900.0	***155**	57.2	487.5	900.0	150	58.0	71.8	900.0	149	58.4	20.1	900.1	153
4	200	600	147	59.1	135.3	900.0	***152**	57.8	130.5	900.0	147	58.8	54.0	900.0	147	58.7	430.0	900.0	151
20	10	600	61	66.9	62.6	900.0	**63**	65.4	8.1	900.0	61	65.7	38.7	900.0	**63**	64.2	315.5	900.0	**63**
20	15	600	51	72.3	347.2	900.0	**53**	71.0	4.4	900.0	51	71.3	5.8	900.0	**53**	70.1	201.5	900.0	**53**
20	20	600	46	74.6	57.0	900.0	**48**	73.5	3.6	900.0	47	73.1	804.5	900.0	**48**	72.4	160.4	900.0	**48**
20	25	600	43	76.0	7.8	900.1	***45**	74.7	7.1	900.0	44	74.6	187.0	900.1	***45**	73.7	342.2	900.0	44
20	40	600	38	78.5	9.8	900.1	**39**	77.8	4.3	900.0	38	77.8	11.4	900.1	**39**	76.9	778.8	900.6	**39**
20	60	600	34	80.6	23.0	900.1	***36**	79.2	14.5	900.0	35	79.2	283.5	900.0	35	78.9	10.2	900.0	35
20	80	600	**33**	80.8	15.7	900.3	**33**	80.7	5.9	900.0	**33**	80.1	20.0	900.1	**33**	79.9	495.3	900.0	**33**
20	100	600	31	82.0	23.8	900.0	**32**	81.4	7.6	900.0	31	81.3	21.3	900.1	**32**	80.4	237.4	900.3	**32**
20	150	600	29	83.0	413.5	900.6	***30**	82.4	733.9	900.0	29	82.4	817.7	900.2	29	82.1	817.7	900.1	29
20	200	600	27	84.0	37.7	901.2	**28**	83.3	14.1	900.0	27	83.1	19.7	900.1	27	83.1	19.7	900.1	**28**

Table A.1: `Random` benchmark. Results when aiming for solution quality.

			A* + BS				A*+ACS				APS				A* + ACS-dist				*lit. best*
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$
10	100	4	**34.1**	51.4	20.1	882.1	**34.1**	15.1	0.3	900.2	**34.1**	10.9	4.0	900.0	**34.1**	9.9	0.1	800.7	**34.1**
10	100	12	**12.7**	0.0	0.2	5.2	**12.7**	0.0	0.2	5.2	**12.7**	0.0	0.7	7.1	**12.7**	0.0	0.0	4.2	**12.7**
10	100	20	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.0	0.1	**7.9**
10	500	4	179.9	40.9	312.0	900.0	***186.0**	38.8	109.5	900.0	179.6	40.2	221.6	792.7	185.5	38.2	52.8	736.9	184.1
10	500	12	76.4	60.2	134.8	900.0	***79.3**	58.5	19.4	900.0	76.4	59.1	123.5	900.0	79.2	57.3	57.2	795.6	78.7
10	500	20	49.7	66.7	87.4	900.0	***51.3**	65.2	48.2	900.0	49.8	65.2	104.8	900.0	***51.3**	63.7	4.2	900.0	51.1
10	1000	4	362.6	42.4	209.7	900.0	***378.0**	40.0	369.7	901.1	362.1	42.1	200.9	900.0	376.1	40.0	288.7	686.6	374.6
10	1000	12	156.2	62.2	214.4	900.4	***163.7**	60.4	143.4	900.0	156.2	61.7	229.8	885.6	163.2	60.0	226.5	818.5	162.0
10	1000	20	102.4	68.9	120.8	900.0	***107.4**	67.3	134.1	900.0	102.6	68.3	238.0	900.2	***107.4**	66.7	294.8	891.2	106.5
50	100	4	**24.2**	32.9	18.7	884.1	**24.2**	29.2	0.3	893.3	**24.2**	24.3	2.3	900.0	**24.2**	23.6	61.3	754.5	**24.2**
50	100	12	**6.9**	0.0	0.1	0.3	**6.9**	0.0	0.1	0.2	**6.9**	0.0	0.2	0.2	**6.9**	0.0	0.1	0.2	**6.9**
50	100	20	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**
50	500	4	136.9	54.1	63.6	892.8	***142.0**	52.4	238.3	897.3	137.2	53.3	137.0	877.7	139.7	52.5	301.7	703.0	141.0
50	500	12	47.8	74.0	190.0	900.0	***49.7**	72.8	152.6	900.0	48.1	73.0	92.5	900.1	49.1	72.2	172.2	893.6	49.2
50	500	20	28.2	79.3	54.3	900.1	**29.3**	78.3	3.6	900.0	28.3	78.3	31.0	900.1	29.1	77.2	133.7	901.0	**29.3**
50	1000	4	278.6	55.3	162.5	900.0	***291.0**	53.4	348.6	900.0	279.0	55.0	190.2	900.0	285.0	54.0	271.7	713.3	288.6
50	1000	12	99.1	75.4	79.0	900.3	***104.2**	74.2	72.8	900.0	99.6	75.0	157.8	900.1	102.7	74.1	358.2	896.1	103.5
50	1000	20	60.5	81.0	114.5	900.1	***63.2**	80.1	52.6	900.0	60.8	80.4	190.2	900.2	62.2	79.9	307.1	901.7	62.5
100	100	4	21.9	37.3	226.2	900.0	***22.1**	32.5	1.1	895.5	***22.1**	28.1	12.2	900.0	***22.1**	24.9	63.1	815.0	22.0
100	100	12	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.0	0.1	**5.2**
100	100	20	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**
100	500	4	127.6	57.9	135.2	900.0	***131.9**	55.6	74.6	900.0	127.7	56.5	86.0	900.0	128.7	56.1	358.6	868.4	130.8
100	500	12	41.8	76.8	41.0	900.1	***43.4**	75.9	52.0	900.2	42.0	75.8	17.8	900.2	42.4	75.4	136.2	900.1	43.1
100	500	20	24.2	81.6	16.1	900.4	***25.0**	80.9	7.1	900.0	24.2	80.8	17.3	900.4	24.5	80.1	107.9	900.0	24.9
100	1000	4	261.8	57.9	135.3	900.0	***272.4**	56.2	291.9	900.0	262.6	57.5	317.2	900.1	265.1	57.1	257.0	817.3	270.6
100	1000	12	89.2	77.8	106.3	900.2	***93.1**	76.8	67.3	900.0	89.0	77.6	93.3	900.2	90.5	77.0	295.2	900.0	92.4
100	1000	20	52.8	83.2	50.4	900.2	***55.1**	82.5	61.1	900.6	53.0	82.9	124.1	900.5	53.6	82.5	216.9	901.6	54.7
150	100	4	20.3	37.4	23.4	900.0	***20.8**	30.6	2.9	899.1	20.7	26.3	117.9	900.0	20.7	22.4	38.5	826.7	20.5
150	100	12	**4.7**	0.0	0.0	0.1	**4.7**	0.0	0.0	0.0	**4.7**	0.0	0.1	0.1	**4.7**	0.0	0.0	0.0	**4.7**
150	100	20	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**
150	500	4	123.5	58.4	180.3	900.0	***127.5**	56.9	150.9	899.7	124.0	57.7	313.6	900.0	123.9	57.6	314.1	884.5	126.4
150	500	12	39.5	77.9	113.9	900.2	***40.9**	77.1	24.1	900.0	39.8	77.1	184.7	900.3	39.7	76.9	141.0	900.0	40.4
150	500	20	22.5	82.8	31.4	900.5	**23.0**	82.3	7.4	900.0	22.5	81.7	27.9	900.8	22.4	81.5	73.3	900.1	**23.0**
150	1000	4	254.6	59.0	365.0	900.1	***264.0**	57.5	245.6	900.0	254.5	58.8	311.1	900.1	255.9	58.6	295.5	864.8	262.8
150	1000	12	84.5	79.0	105.5	900.3	***88.1**	78.0	44.4	900.0	84.6	78.6	101.7	900.3	85.2	78.3	296.4	900.0	87.7
150	1000	20	49.7	84.2	98.6	900.4	***51.6**	83.5	86.8	900.0	49.8	83.8	114.8	900.9	49.9	83.6	280.8	900.1	51.2
200	100	4	19.9	37.9	5.7	900.0	***20.1**	32.8	9.0	898.2	***20.1**	28.2	58.8	900.1	19.9	24.6	17.0	861.6	19.9
200	100	12	**4.1**	0.0	0.0	0.0	**4.1**	0.0	0.0	0.0	**4.1**	0.0	0.1	0.1	**4.1**	0.0	0.0	0.0	**4.1**
200	100	20	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**
200	500	4	121.0	59.3	93.0	900.0	***124.8**	57.9	188.1	900.0	121.3	58.6	126.0	900.1	121.0	58.6	258.6	880.1	123.7
200	500	12	38.0	78.7	41.0	900.4	***39.1**	77.9	57.1	900.0	38.0	78.0	31.2	900.3	38.0	77.7	82.9	900.0	39.0
200	500	20	21.0	83.7	28.0	900.5	***22.0**	82.9	41.4	900.2	21.1	82.9	107.5	901.0	21.2	82.3	116.4	900.1	21.8
200	1000	4	249.6	59.8	283.5	900.1	***258.8**	58.3	170.0	900.0	249.8	59.6	235.8	900.1	250.2	59.4	448.5	893.6	257.6
200	1000	12	81.8	79.5	244.9	900.2	***85.2**	78.6	59.5	900.0	81.9	79.2	205.1	900.4	81.9	79.1	205.4	900.0	84.8
200	1000	20	47.8	84.6	305.7	900.4	***49.4**	84.1	93.4	900.0	47.9	84.5	190.7	900.6	47.9	84.2	309.5	900.1	49.1

Table A.2: BL benchmark. Results when aiming for solution quality (averages over ten instances per row).

			A* + BS				A*+ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
4	10	600	215	40.4	3.4	900.0	222	38.3	100.0	900.0	219	39.5	568.6	900.0	223	**37.9**	236.1	633.2
4	15	600	200	45.2	146.8	900.0	206	**43.3**	167.7	900.0	200	45.4	74.6	900.0	204	43.8	64.4	586.4
4	20	600	188	47.6	209.3	749.4	194	**45.8**	158.7	900.0	189	47.8	731.3	917.0	194	**45.8**	75.0	887.5
4	25	600	182	49.3	397.2	900.0	189	**46.9**	235.2	900.0	183	49.3	189.7	900.0	186	47.8	7.3	578.1
4	40	600	171	52.1	57.8	812.6	176	**50.4**	42.8	900.0	169	53.2	34.9	900.0	173	51.3	627.6	640.6
4	60	600	162	54.5	167.6	900.0	168	**52.4**	471.3	900.1	163	54.6	462.0	900.0	164	53.5	56.2	628.7
4	80	600	157	56.0	108.8	900.1	163	**54.0**	115.6	900.1	159	55.8	115.6	900.0	159	55.1	65.0	725.5
4	100	600	155	56.6	170.2	900.0	160	**54.7**	168.1	900.0	155	56.9	84.4	900.0	157	55.5	118.0	900.0
4	150	600	149	58.1	20.5	900.1	154	**56.2**	301.2	900.0	149	58.4	20.1	900.0	150	57.4	173.2	790.8
4	200	600	147	58.5	115.3	900.1	151	**56.7**	37.8	900.0	147	58.8	54.8	900.0	146	58.2	221.0	900.0
20	10	600	61	64.3	2.8	900.0	63	**62.9**	162.8	900.1	62	65.0	29.2	900.1	62	63.1	59.3	890.1
20	15	600	51	70.5	7.7	900.1	53	**68.8**	86.5	900.0	52	70.8	63.9	900.0	52	69.2	47.9	814.7
20	20	600	46	73.1	7.3	900.1	48	**71.3**	79.9	900.0	46	73.9	8.5	900.1	47	71.9	8.0	855.2
20	25	600	43	74.6	9.1	900.0	45	**72.7**	47.0	900.0	44	74.6	440.7	900.2	44	73.2	7.4	868.9
20	40	600	38	77.2	9.9	900.0	39	**75.9**	29.4	900.0	38	77.8	12.4	900.3	38	76.5	142.2	900.0
20	60	600	34	79.4	434.1	900.2	36	**77.4**	592.2	900.2	35	79.2	293.9	900.3	35	78.0	179.8	900.0
20	80	600	32	80.2	15.8	900.4	33	**78.8**	27.9	900.2	32	80.7	21.4	900.2	32	79.5	88.7	900.2
20	100	600	31	80.7	24.9	900.5	32	**79.5**	56.2	900.2	31	81.0	10.2	900.2	31	80.1	37.5	900.1
20	150	600	28	83.9	36.3	900.0	29	**81.2**	41.1	900.0	29	82.4	74.3	900.9	28	81.8	26.2	900.2
20	200	600	27	84.3	43.2	900.5	28	**81.6**	20.9	900.0	28	82.9	856.6	900.7	27	82.2	387.0	900.3

Table A.3: `Random` benchmark. Results when aiming for small gaps.

A.2 Improvements of A*+ACS Over Other Approaches

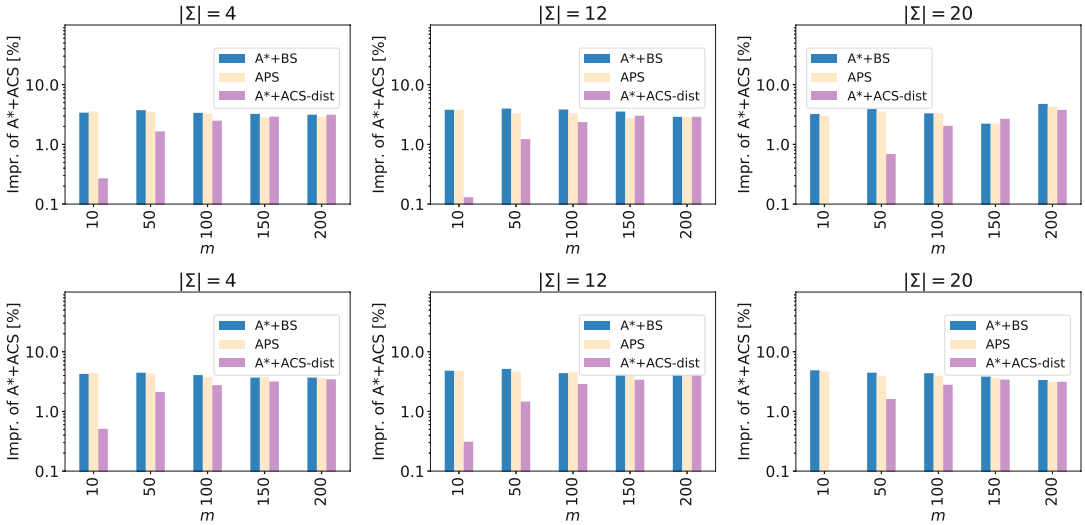


Figure A.1: Improvement of A*+ACS over the competitors in terms of solution quality (in %) for benchmark set `BL`. First row: instances with $n = 500$. Second row: instances with $n = 1000$.

			A* + BS				A*+ACS				APS				A* + ACS-dist			
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
10	100	4	34.0	**4.2**	0.3	797.6	34.1	10.8	1.8	900.0	34.1	8.4	0.6	898.9	34.1	6.7	1.4	645.0
10	100	12	12.7	**0.0**	0.3	2.7	12.7	**0.0**	1.6	8.2	12.7	**0.0**	0.5	4.1	12.7	**0.0**	0.3	2.4
10	100	20	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.0	0.0
10	500	4	180.7	39.3	113.1	901.3	185.4	37.7	337.6	900.0	181.2	39.5	28.2	900.0	185.3	**37.3**	160.8	656.6
10	500	12	76.7	57.9	24.1	900.0	79.0	56.6	216.1	900.1	77.2	58.3	180.3	900.0	79.1	**55.8**	163.0	795.5
10	500	20	49.6	64.1	5.9	900.0	51.2	62.7	205.8	900.1	50.1	64.8	16.1	900.1	51.3	**61.7**	72.4	900.0
10	1000	4	365.5	41.4	168.4	900.0	376.2	39.6	385.8	900.0	366.5	41.4	77.4	900.0	375.6	**39.5**	229.3	700.6
10	1000	12	157.1	61.1	113.5	900.0	162.7	59.6	273.3	900.1	158.2	61.2	135.2	900.0	162.1	**59.5**	111.2	742.0
10	1000	20	103.4	67.5	38.4	900.0	106.6	66.3	265.9	900.2	104.1	67.7	83.7	900.1	106.4	**66.0**	240.6	890.9
50	100	4	23.9	21.1	0.8	900.0	24.2	18.7	9.6	900.0	24.2	25.3	2.6	900.0	24.1	**18.0**	106.5	748.2
50	100	12	6.9	**0.0**	0.2	0.3	6.9	**0.0**	0.3	0.5	6.9	**0.0**	0.3	0.4	6.9	**0.0**	0.1	0.1
50	100	20	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0
50	500	4	136.5	53.3	110.9	900.0	141.3	**51.3**	142.3	901.0	137.2	53.4	101.3	900.0	138.8	52.0	256.1	712.6
50	500	12	47.4	72.8	8.3	900.1	49.2	**71.3**	152.7	900.2	48.1	73.0	137.3	900.0	48.4	71.5	130.8	808.5
50	500	20	28.1	77.7	20.2	900.1	29.3	**76.2**	121.1	900.3	28.4	78.2	33.8	900.3	28.7	**76.2**	94.7	900.5
50	1000	4	278.3	55.0	157.3	900.0	289.8	**52.9**	410.9	900.0	278.9	55.0	96.1	900.0	284.2	53.7	332.7	756.7
50	1000	12	99.0	74.8	104.1	900.1	103.7	**73.4**	288.9	900.2	99.6	75.0	205.8	900.1	101.7	73.8	378.8	874.5
50	1000	20	60.0	80.4	98.4	900.2	62.6	**79.3**	134.3	900.4	60.7	80.6	132.0	900.3	61.3	79.6	320.7	901.0
100	100	4	21.6	24.6	1.3	900.0	22.0	20.4	16.1	900.1	22.1	28.1	12.9	900.0	22.0	**17.4**	80.6	813.9
100	100	12	5.2	**0.0**	0.1	0.1	5.2	**0.0**	0.0	0.1	5.2	**0.0**	0.2	0.2	5.2	**0.0**	0.0	0.0
100	100	20	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0
100	500	4	127.3	56.3	96.3	900.0	131.4	**54.4**	250.1	900.0	127.7	56.5	91.2	900.0	128.0	55.4	242.5	857.0
100	500	12	41.8	75.4	88.1	900.1	43.2	**74.2**	185.7	900.2	42.0	76.0	30.5	900.2	42.0	74.6	131.8	900.0
100	500	20	24.2	80.1	17.0	900.2	24.8	**79.0**	156.8	900.6	24.2	80.7	14.8	900.4	24.0	79.3	152.9	900.1
100	1000	4	261.9	57.5	206.9	900.0	271.4	**55.7**	312.8	900.1	262.6	57.5	354.9	900.0	264.7	56.7	521.1	890.2
100	1000	12	88.9	77.3	53.2	900.1	92.7	**76.1**	363.4	900.3	89.1	77.5	186.5	900.2	89.7	76.7	368.2	900.0
100	1000	20	52.8	83.0	103.0	900.2	54.8	**81.6**	310.1	900.6	53.0	82.9	126.8	900.2	52.9	82.1	309.9	900.1
150	100	4	20.0	22.0	1.6	900.1	20.6	18.1	17.9	900.0	20.7	26.8	123.5	900.1	20.6	**15.1**	182.6	834.9
150	100	12	4.7	**0.0**	0.1	0.1	4.7	**0.0**	0.0	0.0	4.7	**0.0**	0.1	0.1	4.7	**0.0**	0.0	0.0
150	100	20	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0
150	500	4	123.4	57.5	254.8	900.0	127.0	**55.8**	229.7	900.1	123.9	57.7	251.4	900.0	123.2	57.0	364.8	848.4
150	500	12	39.3	76.8	21.3	900.2	40.5	**75.6**	95.5	900.4	39.8	77.1	205.9	900.3	39.3	76.1	124.2	900.1
150	500	20	22.4	82.5	25.8	900.2	22.9	**80.3**	324.6	901.0	22.5	82.0	36.4	900.9	22.1	80.6	142.3	900.2
150	1000	4	254.1	58.7	141.1	900.0	263.3	**57.0**	340.2	900.1	254.6	58.8	293.6	900.1	255.0	58.2	284.5	872.9
150	1000	12	84.4	78.8	159.2	900.7	87.7	**77.2**	294.5	900.4	84.6	78.6	106.5	900.3	84.5	78.0	204.9	900.1
150	1000	20	49.7	84.1	101.7	900.6	51.0	**82.8**	155.6	901.0	49.8	83.8	117.6	900.6	49.0	83.3	309.8	900.3
200	100	4	19.7	23.9	2.3	900.0	20.0	20.2	47.5	900.1	20.1	26.8	56.4	900.1	19.8	**17.8**	134.5	878.7
200	100	12	4.1	**0.0**	0.1	0.1	4.1	**0.0**	0.0	0.1	4.1	**0.0**	0.1	0.1	4.1	**0.0**	0.0	0.0
200	100	20	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0
200	500	4	121.0	58.4	190.1	900.1	124.0	**56.9**	187.4	900.1	121.3	58.6	133.5	900.1	120.3	58.0	186.4	880.4
200	500	12	38.0	77.4	50.6	900.2	38.9	**76.3**	84.2	900.4	38.0	78.0	31.9	900.5	37.5	77.0	214.4	900.1
200	500	20	21.0	83.5	28.0	900.1	21.7	**81.1**	256.0	901.3	21.1	83.0	111.8	901.1	20.9	81.4	288.6	900.2
200	1000	4	249.3	59.6	124.0	900.0	258.1	**57.7**	431.9	900.2	249.8	59.6	242.5	900.1	249.0	59.2	436.3	899.3
200	1000	12	81.7	79.5	130.9	900.2	84.6	**78.0**	311.7	900.7	81.9	79.2	187.4	900.4	81.1	78.8	332.9	900.1
200	1000	20	47.5	84.7	106.7	900.5	49.0	**83.3**	221.1	900.8	47.9	84.5	203.9	900.7	46.9	83.9	298.5	900.3

Table A.4: `BL` benchmark. Results when aiming for small gaps (averages over ten instances per row).

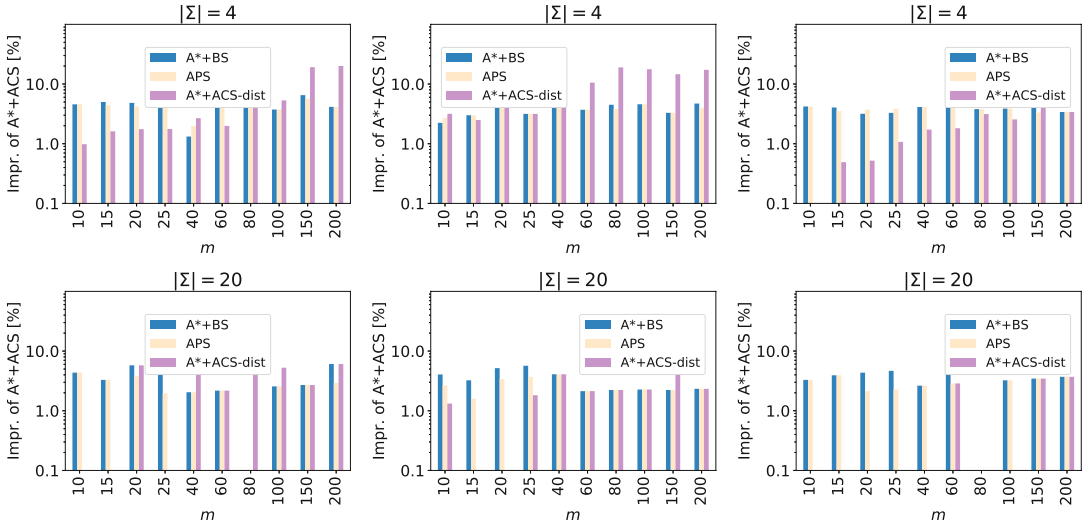


Figure A.2: Improvement of A*+ACS over the competitors in terms of solution quality (in %) for benchmark sets `Rat` (first column of graphs), `Virus` (second column of graphics) and `Random` (last column of graphics).

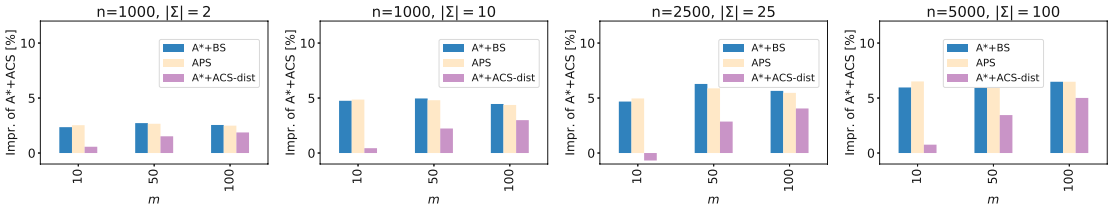


Figure A.3: Improvement of A*+ACS over the competitors in terms of solution quality (in %) for benchmark set ES.

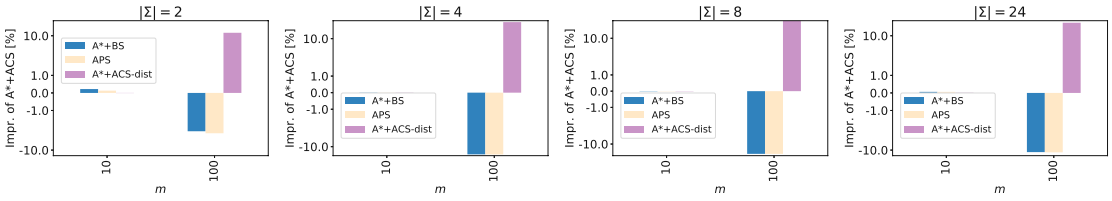


Figure A.4: Improvement of A*+ACS over the competitors in terms of solution quality (in %) for benchmark set BB.

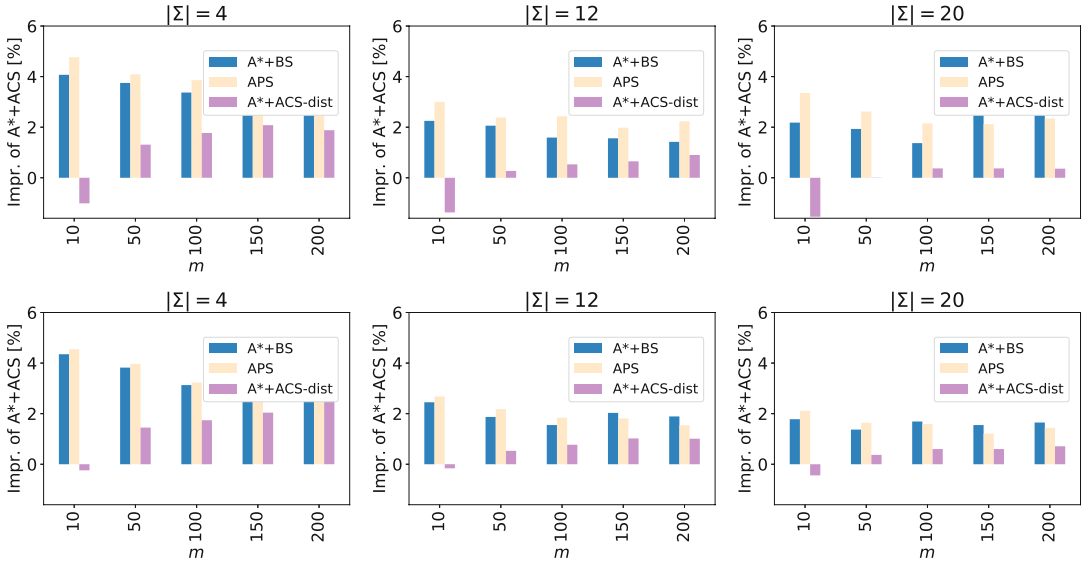


Figure A.5: Improvement of A*+ACS over the competitors in terms of gaps (in %) for benchmark set BL. First row: instances with $n = 500$. Second row: instances with $n = 1000$.

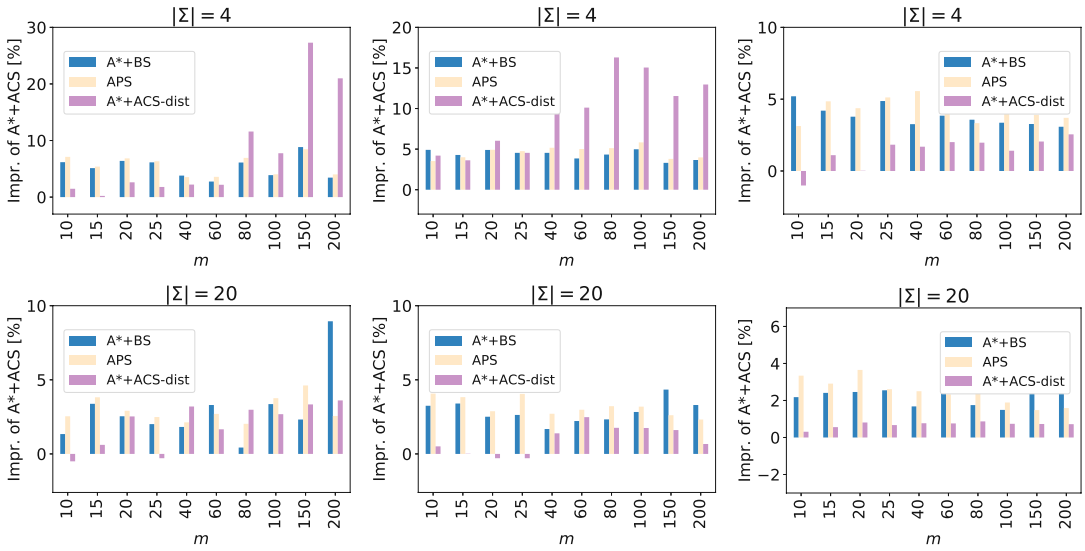


Figure A.6: Improvement of A*+ACS over the competitors in terms of gaps (in %) for benchmark sets `Rat` (first column of graphs), `Virus` (second column of graphics) and `Random` (last column of graphics).

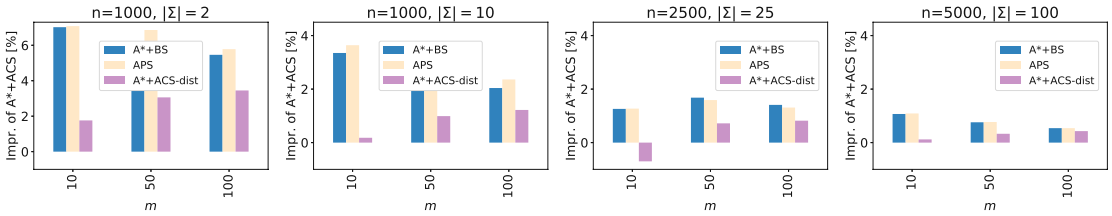


Figure A.7: Improvement of A*+ACS over the competitors in terms of gaps (in %) for benchmark set `ES`.

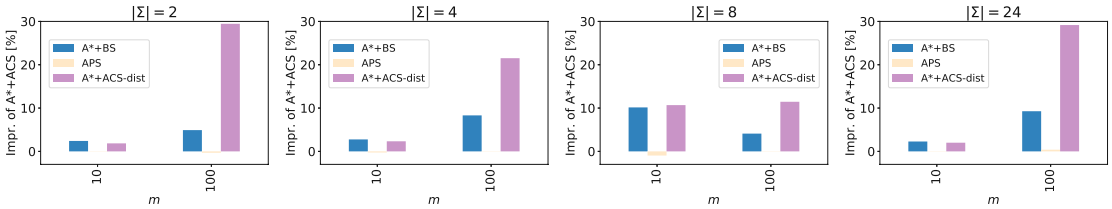


Figure A.8: Improvement of A*+ACS over the competitors in terms of gaps (in %) for benchmark set `BB`.

APPENDIX B

LCPS Problem: Supplementary Material

Appendix B provides supplementary material that concerns of the extended experimental studies for the LCPS problem presented in Chapter 4. The supplementary material includes:

- a CP model for the LCPS problem.
- a complete set of graphics concerning the anytime performance of the proposed algorithms to solve the LCPS problem, both concerning the solution quality and the gaps.
- descriptions of the 2–LCPS algorithms from literature to which the comparison has made, and with the certain implementation aspects we made by our choice due to some open questions and ambiguities found in the corresponding papers.

B.1 Constraint Programming model for the LCPS Problem

For comparison purposes, the following basic CP model for the LCPS problem is considered.

The model uses the following variables. Let $r \in \{1, ..., l\}$, with $l = \min\{|s_i| \,|\, i = 1, \ldots, m\}$ denote the length of the solution string, which shall be maximized. Decision variable $T_{i,j} \in \{1, \ldots, |s_i|\}$ represents the index of the solution string's j-th letter in the i-th input string, for $i = 1, \ldots, m$ and $j = 1, \ldots, r$.

The LCPS is now expressed as follows.

$$\max\ r \tag{B.1}$$

$$T_{i,j} < T_{i,j+1} \qquad \forall i = 1,\dots,m,\ j = 1,\dots,r-1 \tag{B.2}$$

$$s_i[T_{i,j}] = s_{i+1}[T_{i+1,j}] \qquad \forall i = 1,\dots,m-1,\ \forall j = 1,\dots,r \tag{B.3}$$

$$s_1[T_{1,j}] = s_1[T_{1,r-j+1}] \qquad \forall\ j = 1,\dots,\lfloor r/2 \rfloor \tag{B.4}$$

Constraints (B.2) ensure that the sequence of indices $T_{i,1},\dots,T_{i,r}$ is strongly monotonically increasing for each input string s_i. Constraints (B.3) guarantee that for each $j = 1,\dots,r$, the letter at position $T_{i,j}$ in string s_i is the same over all $i = 1,\dots,m$. Last but not least, constraints (B.4) guarantee that the solution is palindromic. Preliminary experiments indicated in this respect that stating these constraints redundantly for all input strings speeds up the solving process.

The model was implemented in MiniZinc 2.1.5. Comparing FlatZinc, Chuffed and Gecode as backbone solvers, we found Gecode to usually work best for this model.

B.2 Anytime plots of the algorithms that show the evolution of the obtained sol. quality

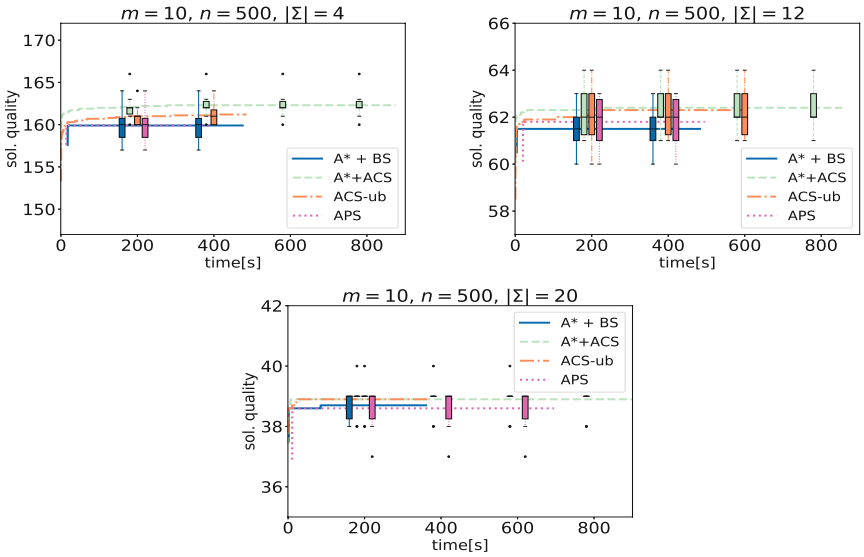


Figure B.1: Instances with m=10 and n=500.

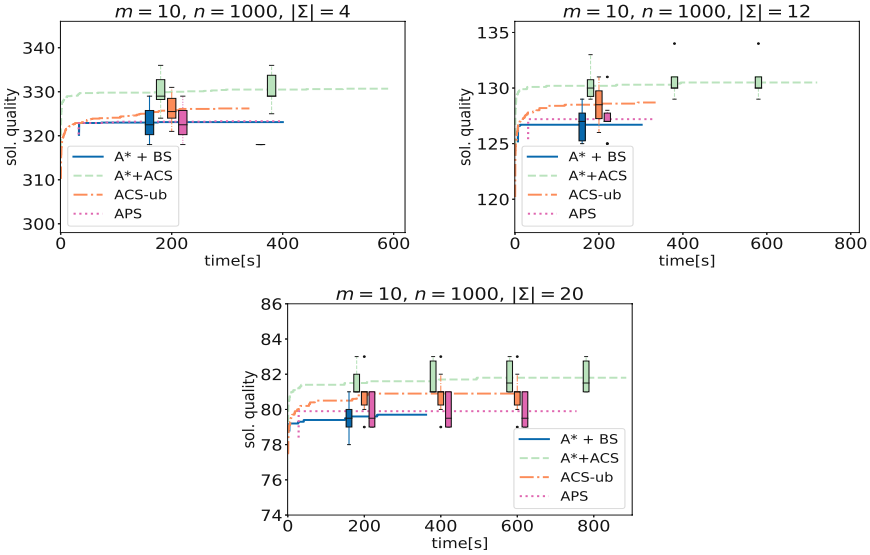


Figure B.2: Instances with m=10 and n=1000.

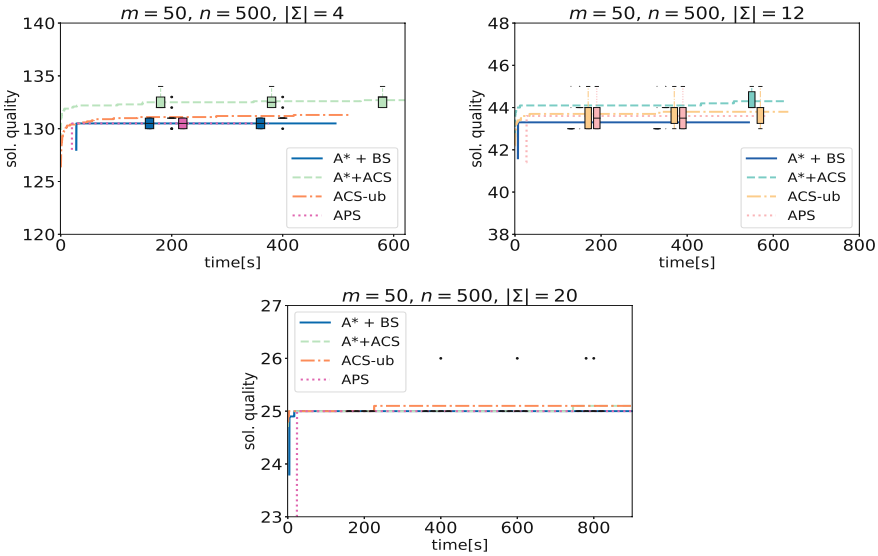


Figure B.3: Instances with m=50 and n=500.

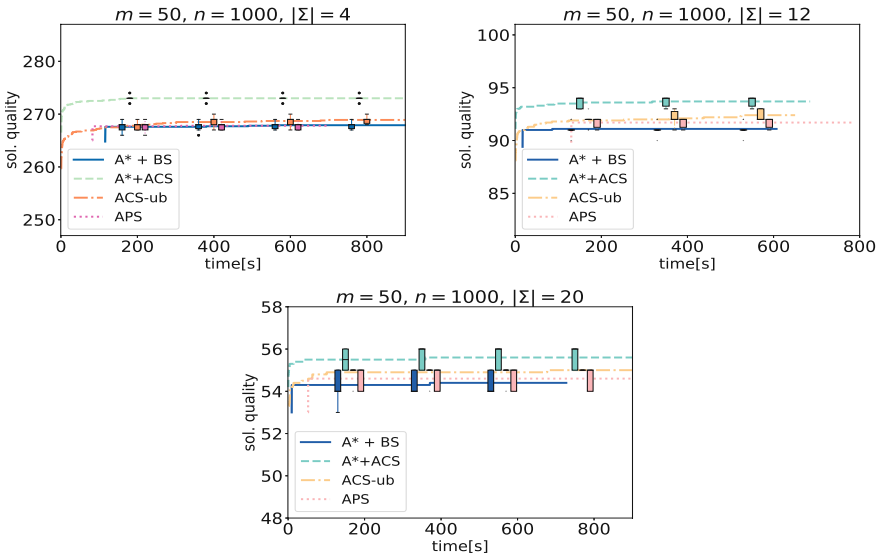


Figure B.4: Instances with m=50 and n=1000.

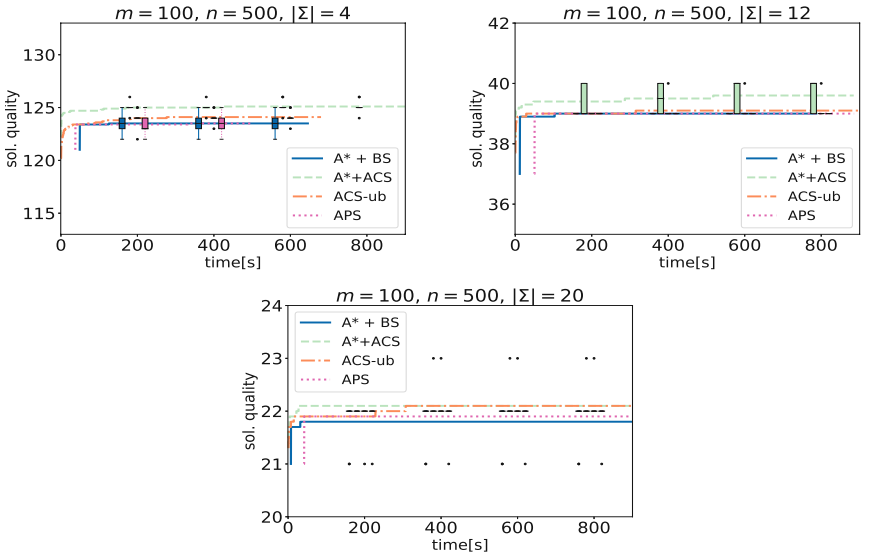


Figure B.5: Instances with m=100 and n=500.

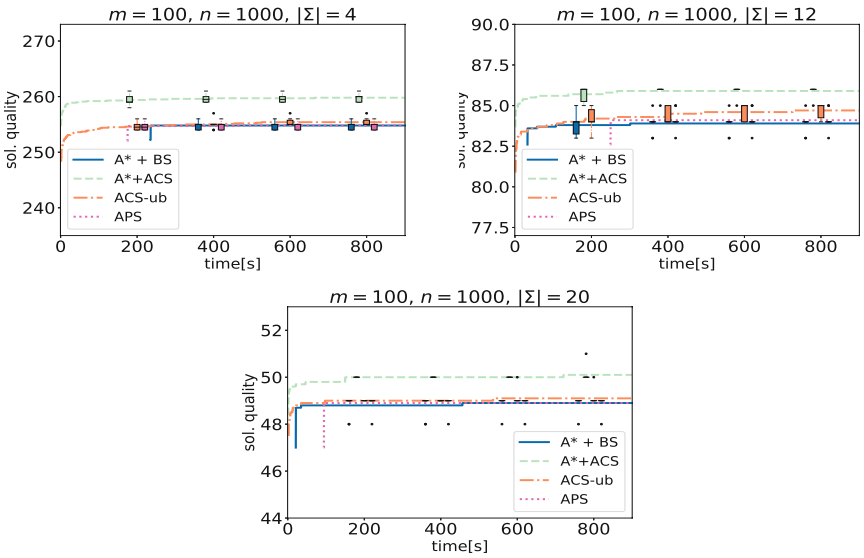


Figure B.6: Instances with m=100 and n=1000.

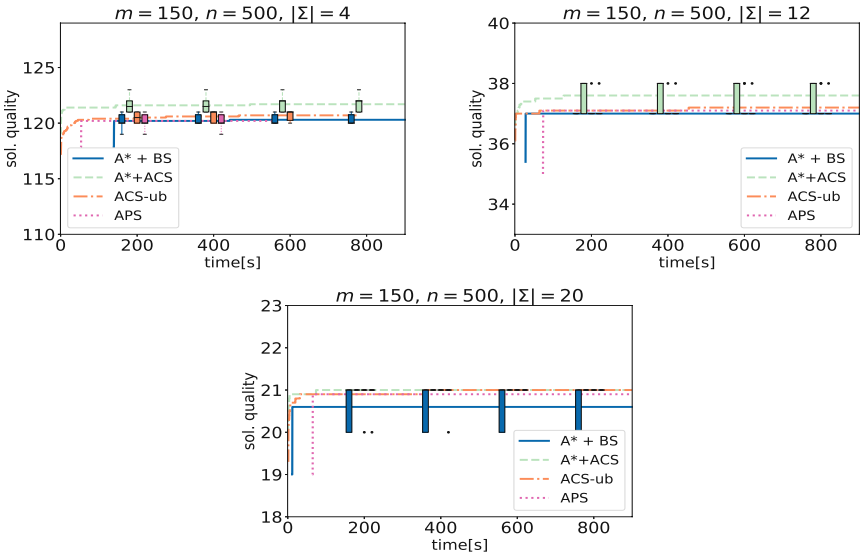


Figure B.7: Instances with m=150 and n=500.

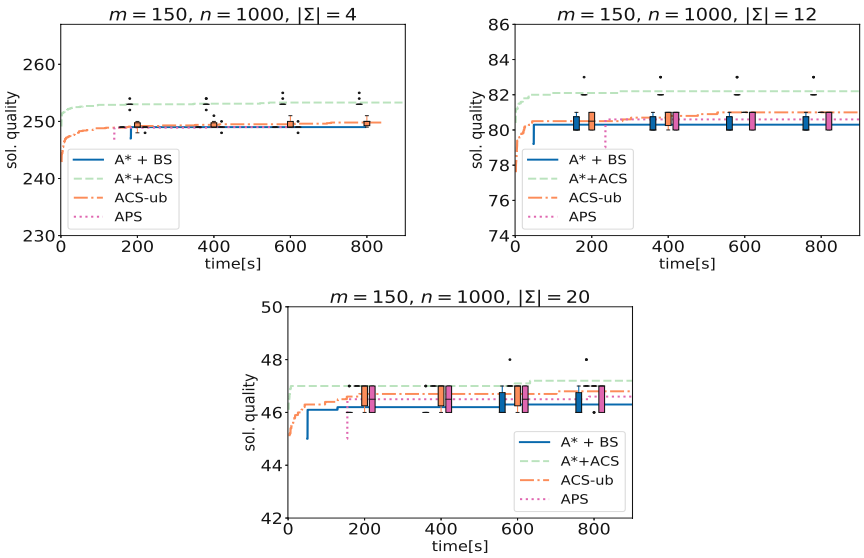


Figure B.8: Instances with m=150 and n=1000.

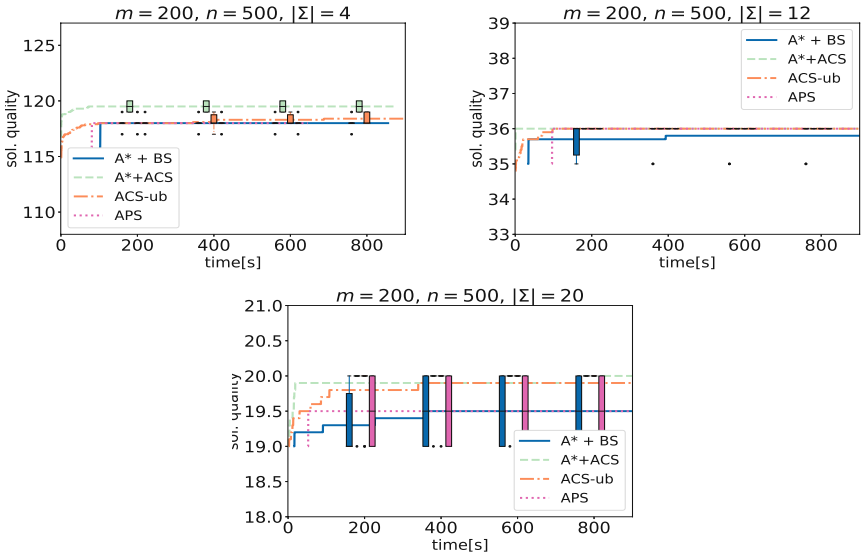


Figure B.9: Instances with m=200 and n=500.

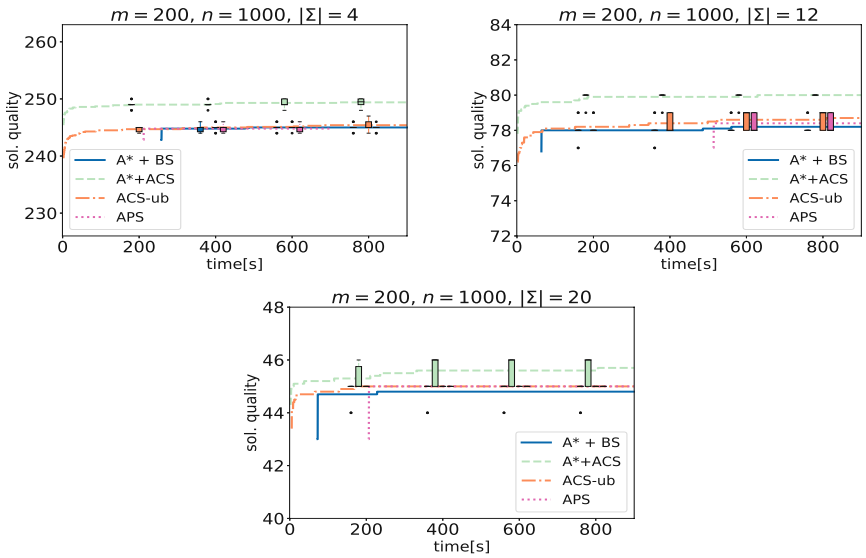


Figure B.10: Instances with m=200 and n=1000.

B.3 Anytime plots of the algorithms that show the evolution of the obtained gaps

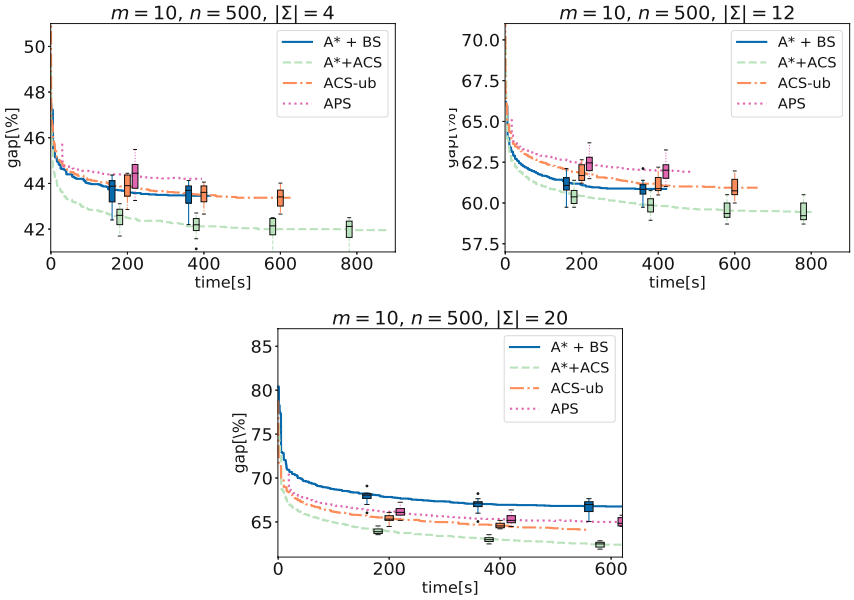


Figure B.11: Instances with m=10 and n=500.

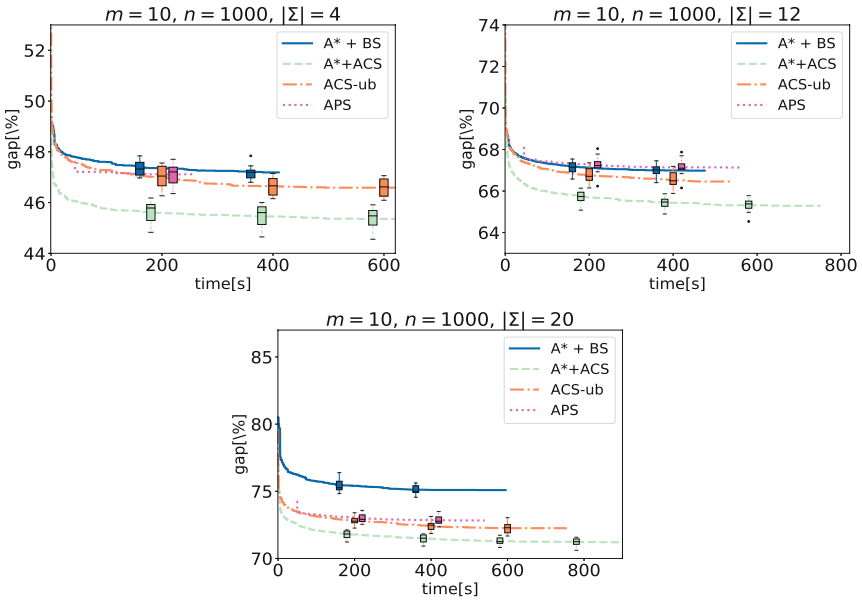


Figure B.12: Instances with m=10 and n=1000.

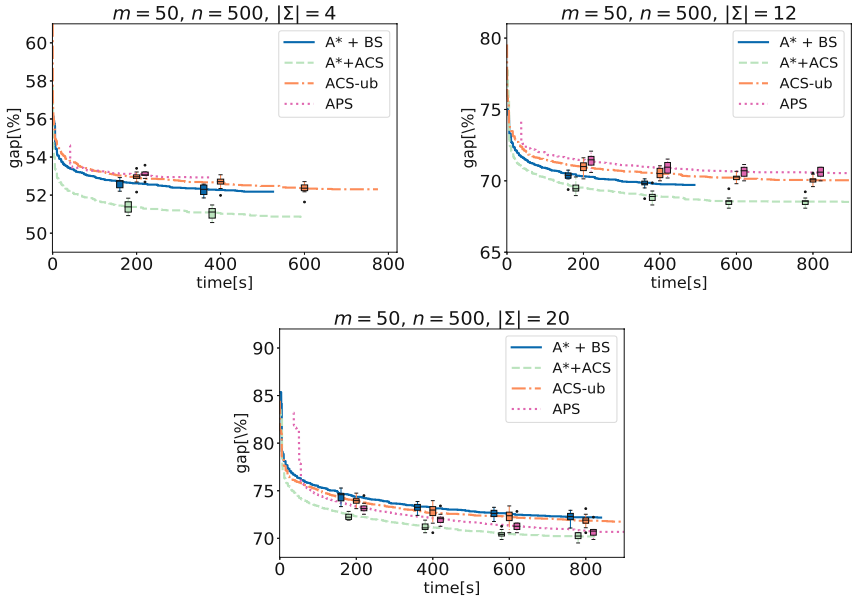


Figure B.13: Instances with m=50 and n=500.

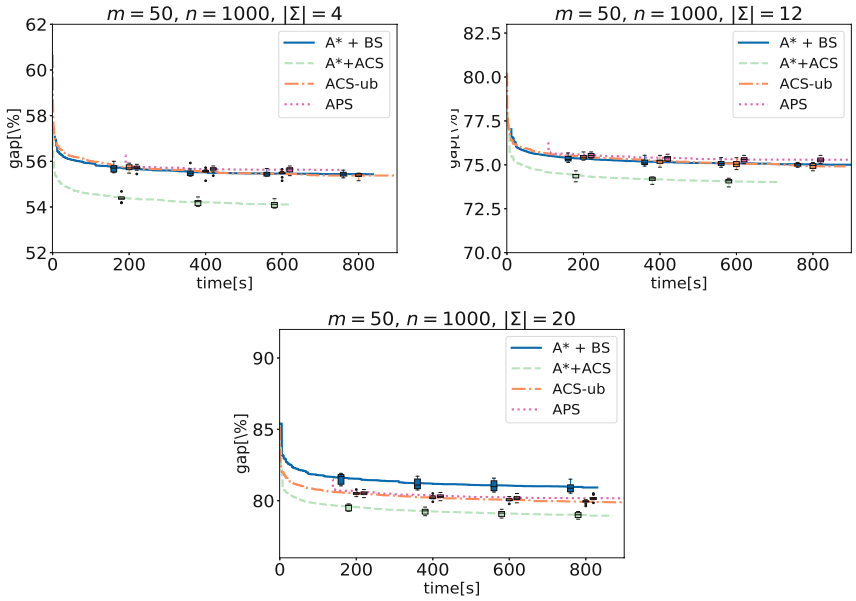


Figure B.14: Instances with m=50 and n=1000.

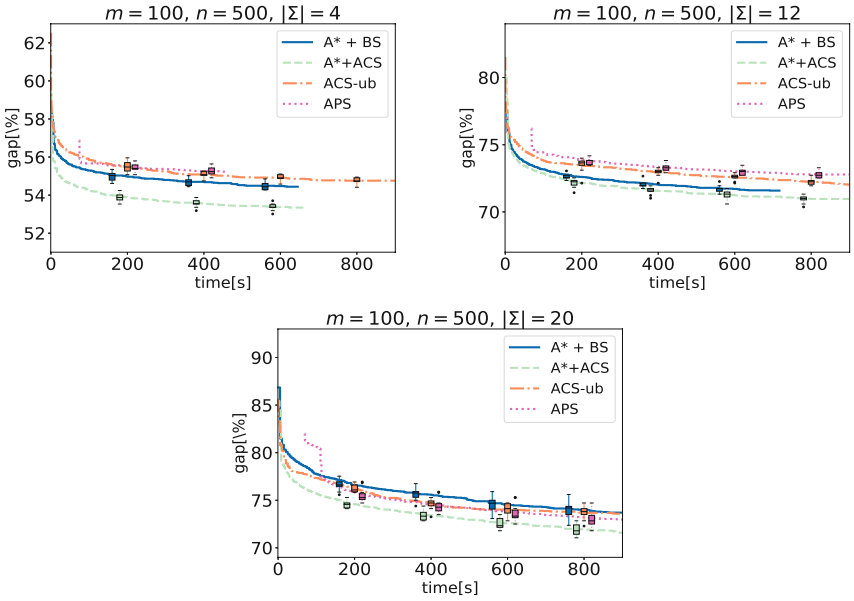


Figure B.15: Instances with m=100 and n=500.

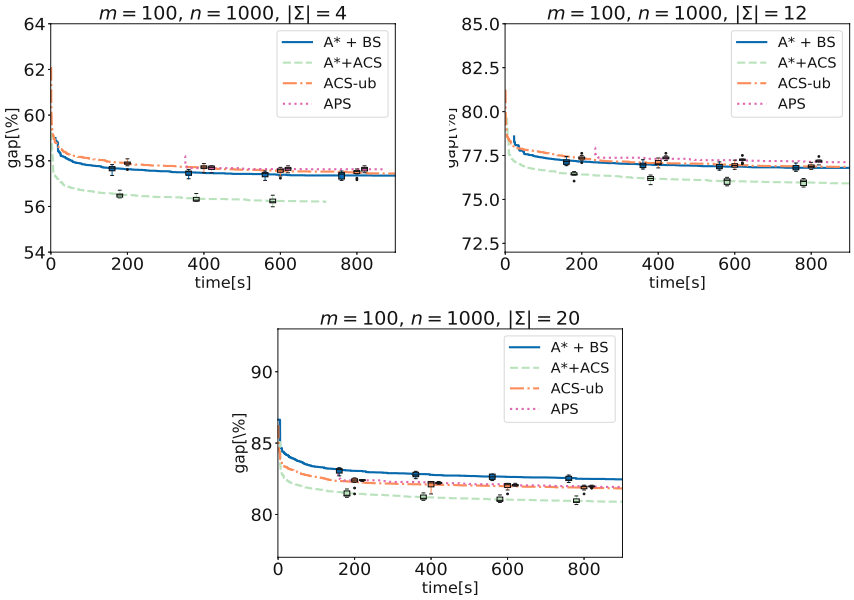


Figure B.16: Instances with m=100 and n=1000.

B.4 The 2–LCPS Approaches from Literature: details of our re-implementations

The existing literature on approaches for solving the 2–LCPS problem is more of theoretical nature. In other words, a computational study comparing the different approaches has

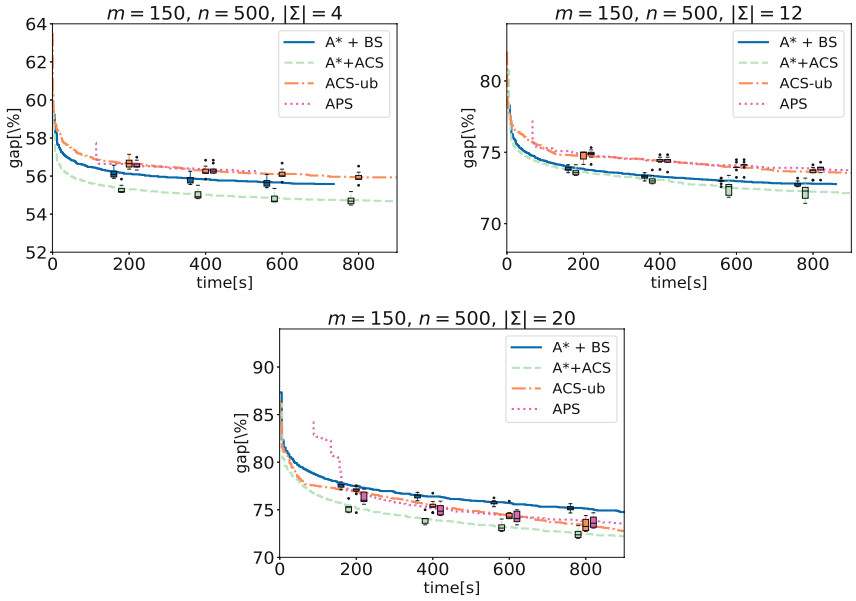


Figure B.17: Instances with m=150 and n=500.

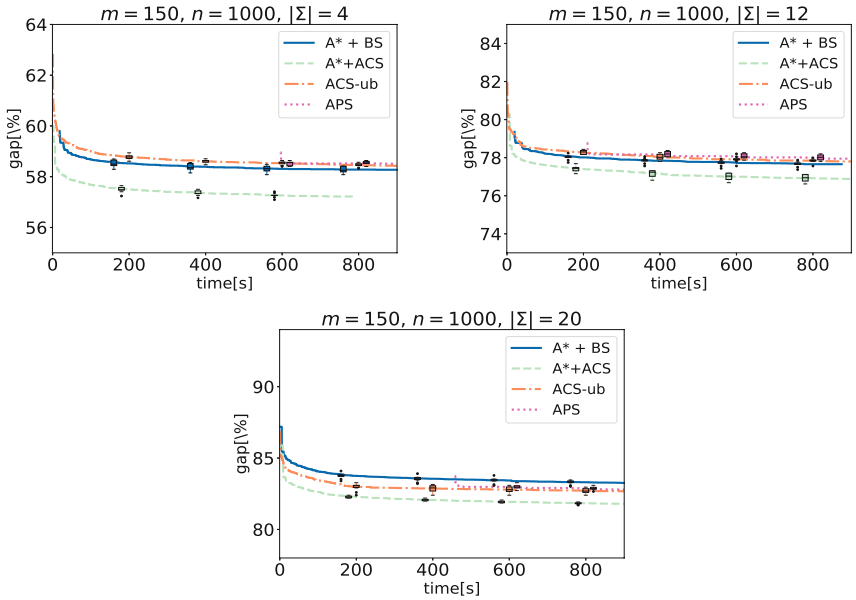


Figure B.18: Instances with m=150 and n=1000.

not been performed so far. In the following, we sketch the existing approaches, putting emphasis on the data structures we chose in order to obtain efficient implementations. Note that a straightforward *Constraint Programming* (CP) model has already been described in the appendix. In the following we focus, therefore, on the description of

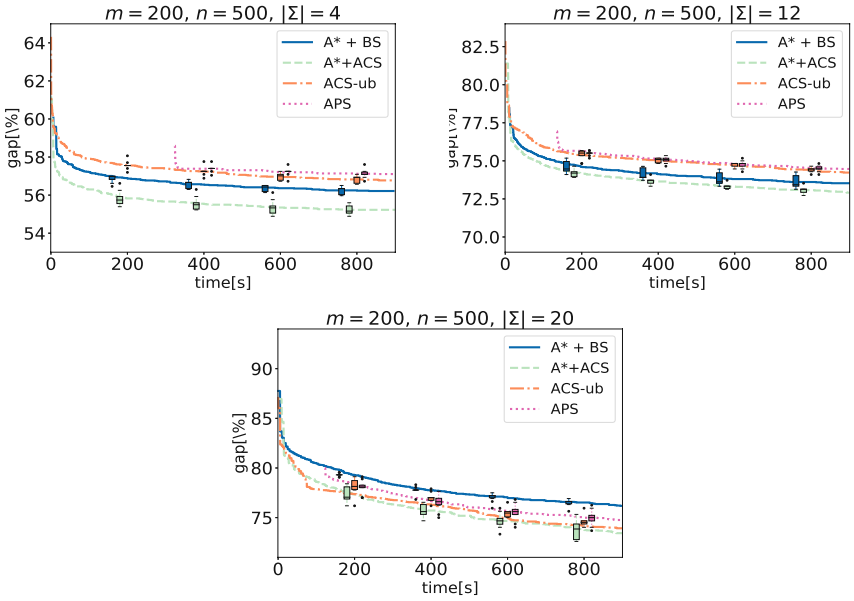


Figure B.19: Instances with m=200 and n=500.

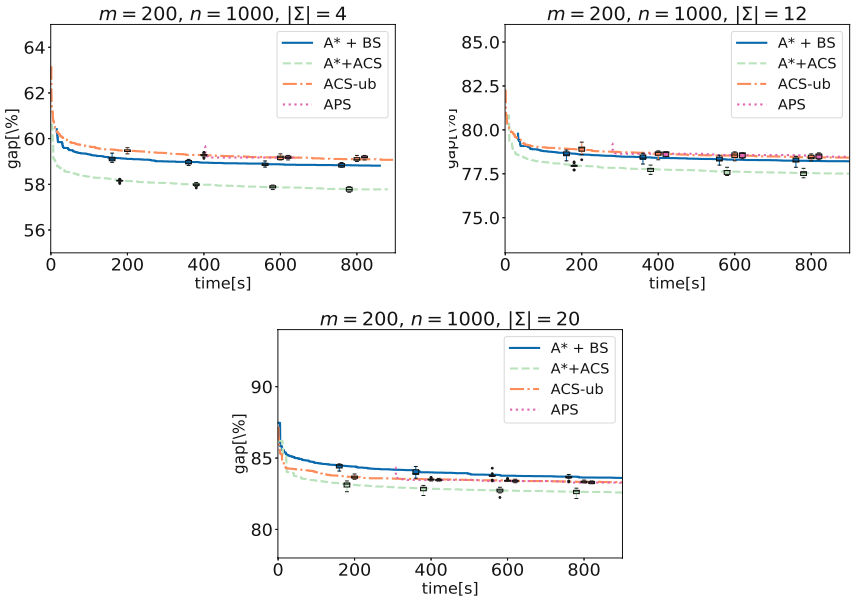


Figure B.20: Instances with m=200 and n=1000.

the remaining three approaches: (*i*) a Dynamic Programming (DP) approach, (*ii*) an algorithm which solves the *Maximum Nesting Depth Rectangle Structures* (MNDRS) problem from computational geometry to which the 2–LCPS problem can be reduced, and (*iii*) the so-called CPSA approach which uses an automaton to solve the 2–LCPS

problem.

B.4.1 Dynamic Programming Approach

Chowdhury et al. [38] presented a dynamic programming approach for solving the 2–LCPS problem. The idea is as follows. Let s_1 and s_2 be the input strings of equal length.[1] Let $\text{LCPS}(i,j,k,l)$, $1 \leq i,j,k,l \leq n$, store the optimal length of the LCPS for substrings $s_1[i,j]$ and $s_2[k,l]$, that is, $\text{LCPS}(i,j,k,l) := |\text{LCPS}(\{s_1[i,j], s_2[k,l]\})|$. Chowdhury proved that the following recursion leads to an optimal solution to the 2–LCPS problem:

$$\text{LCPS}(i,j,k,l) = \begin{cases} 0, & \text{for } i > j \ \vee \ k > l \\ 1, & \text{for } i = j \ \wedge \ k \leq l \\ & \wedge \ s_1[i] = s_2[k] \\ 2 + \text{LCPS}(i+1, j-1, k+1, l-1), & \text{for } i < j \ \wedge \ k < l \\ & \wedge \ s_1[i] = s_1[j] \\ & = s_2[k] = s_2[l] \\ \max(\text{LCPS}(i+1,j,k,l), \text{LCPS}(i,j-1,k,l), & \\ \text{LCPS}(i,j,k+1,l), \mathit{lcps}(i,j,k,l-1)), & \text{otherwise.} \end{cases}$$

The first one of the two main cases is obtained when $i < j$, $k < l$ and $s_1[i] = s_1[j] = s_2[k] = s_2[l]$. In this case, the value of $\text{LCPS}(i,j,k,l)$ can easily be obtained as $\text{LCPS}(i,j,k,l) := 2 + \text{LCPS}(i+1, j-1, k+1, l-1)$. Otherwise, the value of $\text{LCPS}(i,j,k,l)$ can be recursively calculated by solving the four smaller subproblems corresponding to the values of $\text{LCPS}(i+1,j,k,l)$, $\text{LCPS}(i,j-1,k,l)$, $\text{LCPS}(i,j,k+1,l)$, and $\text{LCPS}(i,j,k,l-1)$. The maximum of these values actually corresponds to $\text{LCPS}(i,j,k,l)$.

The DP approach stores the solution of all possible subproblems in a 4-dimensional table of size $n \times n \times n \times n$, which implies a space complexity of $O(n^4)$. The DP recursion generates $O(n^4)$ distinct subproblems, in $O(1)$ time each, in a bottom-up manner, implying a time complexity of $O(n^4)$ for the approach. Note that the value of an optimal solution is stored in $\mathit{lcps}(1,n,1,n)$.

B.4.2 The MNDRS Approach

The *Maximum Nesting Depth Rectangle Structures* (MNDRS) problem is known from computational geometry and can be described as follows. Given a set of rectangles in the Euclidean plane, find a maximum-length sequence of these rectangles such that each rectangle in the sequence contains all following rectangles of the sequence. Chowdhury [38] mapped the 2–LCPS problem to the MNDRS problem by introducing for each pair $((i,k),(j,l))$ of index couples—such that $s_1[i] = s_1[j] = s_2[k] = s_2[l]$—a rectangle

[1] The case of input strings of different length can easily be transformed to the case of input strings with equal length.

in $\mathbb{N}^2$ whose lower left corner is (i, k) and whose upper right corner is (j, l). Moreover, Chowdhury provided a sparse DP approach for solving the MNDRS problem by making use of 3-dimensional balanced range search trees.

Subsequently, Inenaga and Hyyrö [90] proposed an algorithm for solving the MNDRS problem which makes use of two simple but clever data structures. However, since their work is theoretically oriented, they did not deal with the question of how to implement the proposed algorithm in an efficient way. Therefore, the following description of this algorithm presents our own implementation.

The first one of the two data structures mentioned above is used to find—for each combination of a letter $c \in \Sigma$ and a rectangle R—a sub-rectangle R_c of maximum area which is strongly contained in R and whose indexes of the lower left and upper right vertices correspond to letter c. Finding such a rectangle can be done in constant time $O(1)$ by making use of two predecessor and two successor tables (which basically correspond to the Pred and Succ data structures used in the preprocessing of our A*, as described in the main paper). The second data structure is a space-efficient 4D–table used for checking whether or not a rectangle R is processed in the main recursion, as explained in the following. Our implementation uses hash maps to realize this data structure and thus answers a query in $O(1)$ expected time. Note that $O(\mathcal{R}^2)$ memory is needed for these data structures, where $\mathcal{R}$ denotes the number of matching positions in the input strings.

A recursion is used in order to calculate, for each rectangle, its so-called *nesting weight number*, which denotes the maximum length of a sequence of rectangles nested in R (including the rectangle R itself). The initial call of the recursion is applied to the virtual rectangle $R_V = (0, 0, n + 1, n + 1)$ and its final nesting number actually corresponds to the length of an optimal solution to the 2–LCPS problem. Moreover, all rectangles are initially marked as non-processed. Furthermore, a rectangle R will be marked as processed when its nesting weight number is calculated. The information about the nesting numbers is stored in a 4D-hash table. If an already processed rectangle is encountered in the recursion, its nesting number—as obtained from the 4D-hash table—is immediately returned. Let us suppose a rectangle R is currently being processed. For each letter c, the maximum sub-rectangle R_c is being determined (by making use of the Pred and Succ structures) and the recursion is repeatedly called for each R_c until one of the following conditions is encountered: (*i*) R_c is *empty*, returning value 0; (*ii*) R_c is a *point* or a *line*, returning value 1; or (*iii*) R_c is already processed, returning the stored nesting number. Note that a point or a line correspond to a middle letter in the respective solution to the 2–LCPS problem. It can be shown that—concerning the expected asymptotic runtime—the MNDRS approach of Inenaga and Hyyrö for solving the 2–LCPS problem is by a factor of $|\Sigma|$ faster than the DP approach.

The approach of Inenaga and Hyyrö and the approach of Chowdhury et al. have the same memory complexity [90]. Moreover, Chowdhury's MNDRS approach is—with

respect to the expected asymptotic runtime—slower than the DP approach in the case of input strings that were generated uniformly at random, where the number of matchings of positions in the input strings is $\mathcal{R} = O(n^2)$. If the number of matchings is lower, for example $\mathcal{R} = O(n^{1.5})$, MNDRS exhibits an advantage over DP in terms of the asymptotic runtime [38]. Additionally, the data structures used in the approach of Inenaga and Hyyrö [90] are simpler—with respect to the ease of implementation—than the sophisticated data structures from computational geometry used in Chowdhury's MNDRS approach. In general, it is not expected that the MNDRS approach of Chowdhury has a significant advantage over the MNDRS approach of Inenaga and Hyyrö in the context of instances generated uniformly at random. For these reasons we decided to implement the algorithm from [90] for comparison purposes.

B.4.3 A Palindromic Subsequence Automaton Approach

In the following we describe the so-called *Common Palindromic Subsequence Automaton* (CPSA) approach from [80], highlighting the major adaptation we applied in order to obtain an efficient algorithm for solving the 2–LCPS problem. The algorithm is based on a so-called *Palindromic Subsequence Automaton* (PSA) for each input string, that is, a PSA M_1 for input string s_1 and a PSA M_2 for input string s_2. Each of these PSAs works on the space of the first halves of all palindromic subsequences of the corresponding input strings. The major idea for solving the 2–LCPS problem is based on the construction of a so-called intersecting automaton that connects M_1 and M_2.

In the following we describe the structure of a PSA for a string s. The automaton is denoted by $M(Q, \Sigma, \tau, w, F)$, where Q is a set of states, $\tau : Q \times \Sigma \mapsto Q$ is a transition function, $w : Q \times Q \mapsto \mathbb{N}_0$ is a weight function, and F is a set of final states. M has a state $q \in Q$ associated with each pair (i, j), $i, j \geq 1$, such that $s[i] = s^{\text{rev}}[j]$. The initial state of the automaton is defined by $q_0 = (0, 0)$. A transition $\tau(q_1, a) = q_2$ between two states $q_1 = (i', j')$ and $q_2 = (i'', j'')$ is possible if and only if $s[i''] = s^{\text{rev}}[j''] = a$ and there exist no positions k and l, $i' < k < i'' \wedge j' < l < j''$, matching the letter a. Note that the states can be seen as the nodes of a weighted directed acyclic graph, with q_0 as the root node. Moreover, existing transitions between states can be seen as the directed edges of this graph. A state q is therefore a partial LCPS solution that corresponds to a directed path from the root node to q. Note that a transition corresponds to the extension of a partial solution, either by one or by two letters. More specifically, if $i'' + j'' = |s| + 1$, the corresponding transition is an extension by a single letter and otherwise an extension by two letters. The weight function of the PSA is defined accordingly:

$$w(q_1, q_2) = \begin{cases} 2, & \text{if } i'' + j'' < |s| + 1 \\ 1, & \text{if } i'' + j'' = |s| + 1 \\ 0, & \text{else.} \end{cases}$$

Each state can be considered as a final (accepted) state of the automaton, that is, $F = Q$. Any path p from the initial state q_0 to any final state—that is, $p = q_0 \cdots q_{r_i} \cdots q_{r_k}$ with

$k \geq 0$—corresponds to the following palindromic subsequence s^p of s:

$$s^p = \begin{cases} s[i_{r_1}] \cdots s[i_{r_{k-1}}] s[i_{r_k}] \cdot (s[i_{r_1}] \cdots s[i_{r_{k-1}}] s[i_{r_k}])^{\text{rev}} & \text{if } i_k + j_k < |s| + 1, \\ s[i_{r_1}] \cdots s[i_{r_{k-1}}] \cdot s[i_{r_k}] \cdot (s[i_{r_1}] \cdots s[i_{r_{k-1}}])^{\text{rev}} & \text{if } i_k + j_k = |s| + 1. \end{cases}$$

Let $M_1 = (Q_1, \Sigma, \tau_1, w_1, F_1)$ be the PSA of s_1 and $M_2 = (Q_2, \Sigma, \tau_2, w_2, F_2)$ be the PSA for s_2, respectively. The intersecting automaton $M_{\text{isec}}(Q, \Sigma, \tau, w, F) = M_1 \cap M_2$, called *Common Palindromic Subsequence Automaton* (CPSA), is defined as follows. It has an initial state (root node) denoted by $q_{M_{\text{isec}}} = (q'_0, q''_0)$, where $q'_0 \in Q_1$ and $q''_0 \in Q_2$ are the root nodes of M_1 and M_2, respectively. In general, if $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ are the languages accepted by M_1 and M_2, the intersecting automaton M_{isec} will accept all words common to both languages, i.e., $\mathcal{L}(M_{\text{isec}}) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. A maximum path in the directed acyclic graph defined by M_{isec} corresponds to an optimal solution to the 2–LCPS problem.

The transition function τ of M_{isec} is defined as follows. If $q' = (q'_1, q'_2)$, $q'' = (q''_1, q''_2) \in Q \subseteq Q_1 \times Q_2$, then a transition between the nodes—that is, $\tau(q', a) = q''$ for some $a \in \Sigma$—exists if and only if $\tau_1(q'_1, a) = q''_1$ and $\tau_2(q'_2, a) = q''_2$. The weight corresponding to this edge is calculated as follows:

$$w(q', q'') = \begin{cases} 2 & \text{if } w_1(q'_1, q''_1, a) = w_2(q'_2, q''_2, a) = 2 \\ 1 & \text{if } w_1(q'_1, q''_1, a) = 1 \vee w_2(q'_2, q''_2, a) = 1 \\ 0, & \text{else.} \end{cases} \tag{B.5}$$

The final states $q \in F$ of M_{isec} are all states for which $q'_1 \in F_1$ or $q'_2 \in F_2$. In order to construct the intersection automaton M_{isec}, we start by adding the root node $q_{M_{\text{isec}}}$ to a queue Q' and Q. At each step, the top node $q = (q', q'')$ is taken from Q' and the outgoing edges E_1 of q' in M_1 and outgoing edges E_2 of q'' in M_2 are considered. All edges $e_1 = q'r'_1 \in E_1$ and $e_2 = q''r'_2 \in E_2$ for which $\tau_1(q', a_1) = r'_1 \ \wedge \ \tau_2(q'', a_2) = r'_2 \ \wedge \ a_1 = a_2$ will create a new state $r = (r'_1, r'_2)$ which is then added to Q' and to the set of states Q (if not already there). We implemented Q by means of a hash table in order to be able to efficiently check whether or not a state is already in Q. If state r is added to Q, an extension of functions τ and w of M_{isec} is generated for r by determining the corresponding weights for the newly created edge qr, as defined in (B.5). Afterwards, q is removed from the top of Q'. The procedure stops once Q' is empty. A detailed description of this process is provided in [80].

As mentioned above, M_{isec} also defines a directed acyclic graph, and for solving the corresponding 2–LCPS problem it is actually sufficient to find a maximum-length path in M_{isec}. This takes time $O(|Q|) = O(|Q_1| \cdot |Q_2|)$ when applying a topological sort to all nodes of Q. For this purpose, the authors of [80] construct a maximum-length automaton [87], which accepts all the subsequences of maximum length among the subsequences from $\mathcal{L}(M_{\text{isec}})$. The automaton is constructed in $O(|Q|)$ time by using a topological sort of the nodes in M_{isec} followed by removing all the transitions and states which are not

part of any longest path from the initial state to a final state.

Since a main effort of the algorithm is actually to construct the CPSA, and as we are only interested in finding one optimal solution of possibly several ones, and as constructing the maximum-length automaton followed by solving the 2–LCPS problem is more time consuming than a direct application of the maximum-path algorithm to M_{isec}, we decided to simply use the maximum-path algorithm based on the topological sort of the states of the CPSA in order to solve the 2–LCPS problem, without the construction of a maximum-length automaton. This approach yields more directly one optimal solution of the 2–LCPS problem.

APPENDIX C

Application of Max-Clique Solvers to Solve LCS problem: Supplementary Material

In Appendix C, we present the additional results on the studies presented in Chapter 6.

C.1 Numerical Results after graph reduction

Table C.1: Results obtained after graph reduction (RFLCS instances of Rflcs-Set1).

$\|\Sigma\|$	n	Spec. Tech.		Cplex				Lmc				Lscc+Bms	
		result	$\bar{t}$	**result**	$\bar{t}$	$\bar{t}_{\textbf{opt}}$	**#opt**	**result**	$\bar{t}$	$\bar{t}_{\textbf{opt}}$	**#opt**	**result**	$\bar{t}$
$n/8$	32	4.00	0.00	=4.00	0.07	0.07	30	=4.00	0.00	0.03	30	=4.0	0.00
	64	8.00	0.00	=8.00	0.52	0.52	30	=8.00	0.00	0.11	30	=8.0	0.00
	128	16.00	0.00	=16.00	6.10	6.10	30	=16.00	0.00	76.21	30	=16.0	0.00
	256	31.97	0.03	=31.97	202.49	202.49	30	=31.90	18.82	– –	0	=31.97	0.03
	512	63.27	38.89	+**30.97**	1889.32	– –	0	*−62.40*	347.07	– –	0	=63.90	69.94
	1024	111.57	146.30	=0.03	1062.21	– –	0	=112.53	923.56	– –	0	+**116.30** *	1099.26
	2048	182.67	278.15	– –	– –	– –	0	*−182.37*	1415.51	– –	0	+**182.40**	1443.72
	4096	283.33	67.15	– –	– –	– –	0	=281.37	1114.40	– –	0	+**263.83**	2079.95
$n/4$	32	7.83	0.01	=7.83	0.01	0.01	30	=7.83	0.00	0.02	30	=7.83	0.00
	64	14.67	0.08	=14.67	0.08	0.08	30	=14.67	0.00	0.03	30	=14.67	0.00
	128	25.77	3.47	=25.93	1.78	2.34	30	=25.93	0.01	0.12	30	=25.93	0.01
	256	43.70	10.39	=43.97	9.20	19.41	30	=43.97	0.13	0.67	30	=43.97	0.12
	512	67.90	35.97	+**68.57**	233.30	489.73	30	=68.57	95.78	104.92	29	=68.57	3.20
	1024	103.00	50.18	+**104.53**	1169.51	1969.63	14	*−103.73*	487.95	– –	0	+**105.00** *	482.73
	2048	154.33	150.72	+**117.47**	1986.41	1001.01	3	*−152.73*	513.24	– –	0	+**153.93**	1229.79
	4096	226.67	452.60	+**23.17**	648.79	– –	0	+**224.63**	573.22	– –	0	+**215.00**	1510.06
$3n/8$	32	8.77	0.01	=8.77	0.00	0.00	30	=8.77	0.00	0.02	30	=8.77	0.00
	64	15.53	0.05	=15.53	0.03	0.03	30	=15.53	0.00	0.02	30	=15.53	0.00
	128	24.90	0.45	=24.90	0.33	0.34	30	=24.90	0.00	0.04	30	=24.90	0.01
	256	39.97	2.68	=39.97	0.81	0.98	30	=39.97	0.02	0.08	30	=39.97	0.06
	512	59.77	12.80	=59.97	9.15	14.02	30	=59.97	0.14	0.58	30	=59.97	0.39
	1024	90.50	34.73	+**90.73**	30.62	49.38	30	=90.73	2.93	19.30	30	=90.73	5.81
	2048	130.57	51.96	+**131.07** *	323.31	369.50	28	+**130.57**	117.06	410.36	13	+**131.00**	726.79
	4096	191.37	98.06	+**186.27**	739.49	653.72	23	+**190.90**	399.25	– –	0	+**189.07**	1441.64
$n/2$	32	8.87	0.01	=8.87	0.00	0.00	30	=8.87	0.00	0.02	30	=8.87	0.00
	64	14.80	0.01	=14.80	0.01	0.01	30	=14.80	0.00	0.02	30	=14.80	0.00
	128	22.93	0.03	=22.93	0.05	0.06	30	=22.93	0.00	0.03	30	=22.93	0.00
	256	35.10	1.23	=35.20	0.30	0.34	30	=35.20	0.01	0.04	30	=35.20	0.03
	512	53.10	5.70	=53.13	1.41	1.78	30	=53.13	0.03	0.15	30	=53.13	0.16
	1024	79.03	12.43	=79.13	4.46	5.16	30	=79.13	0.24	0.60	30	=79.13	3.37
	2048	115.30	53.49	+**115.70** *	26.72	29.61	30	+**115.67**	116.09	350.85	28	+**115.67**	174.36
	4096	167.47	89.48	+**167.97** *	114.74	158.87	30	+**167.60**	36.16	208.53	18	+**167.43**	664.34
$5n/8$	32	8.60	0.01	=8.60	0.00	0.00	30	=8.60	0.00	0.02	30	=8.60	0.00
	64	13.30	0.08	=13.30	0.00	0.00	30	=13.30	0.00	0.02	30	=13.30	0.00
	128	21.20	0.03	=21.20	0.01	0.01	30	=21.20	0.00	0.02	30	=21.20	0.00
	256	32.53	0.34	=32.53	0.08	0.09	30	=32.53	0.00	0.03	30	=32.53	0.00
	512	47.83	1.81	=47.83	0.65	0.65	30	=47.83	0.02	0.06	30	=47.83	0.01
	1024	70.03	2.62	=70.20	1.29	1.32	30	=70.20	0.04	0.17	30	=70.20	0.25
	2048	103.80	24.71	+**103.97**	4.28	4.32	30	=103.97	0.49	3.18	30	+**103.97**	2.77
	4096	150.00	152.30	+**150.57** *	50.27	50.36	30	+**150.43**	281.64	219.92	24	+**150.40**	365.03
$3n/4$	32	8.17	0.01	=8.17	0.00	0.00	30	=8.17	0.00	0.02	30	=8.17	0.00
	64	12.53	0.01	=12.53	0.00	0.00	30	=12.53	0.00	0.02	30	=12.53	0.00
	128	19.70	0.07	=19.70	0.00	0.00	30	=19.70	0.00	0.02	30	=19.70	0.00
	256	29.97	0.13	=29.97	0.02	0.02	30	=29.97	0.00	0.02	30	=29.97	0.00
	512	44.53	0.59	=44.57	0.17	0.19	30	=44.57	0.00	0.04	30	=44.57	0.02
	1024	65.07	7.84	=65.20	0.52	0.52	30	=65.20	0.02	0.09	30	=65.20	0.07
	2048	94.53	13.11	=94.67	0.87	0.90	30	=94.67	0.04	0.16	30	+**94.67**	1.29
	4096	136.57	60.44	+**136.77** *	11.63	11.80	30	+**136.47**	171.71	3.79	26	+**136.73**	26.59
$7n/8$	32	7.67	0.00	=7.67	0.00	0.00	30	=7.67	0.00	0.02	30	=7.67	0.00
	64	11.57	0.00	=11.57	0.00	0.00	30	=11.57	0.00	0.02	30	=11.57	0.00
	128	18.40	0.01	=18.40	0.01	0.01	30	=18.40	0.00	0.02	30	=18.40	0.00
	256	27.80	0.04	=27.80	0.01	0.01	30	=27.80	0.00	0.02	30	=27.80	0.00
	512	40.57	4.65	=40.60	0.06	0.06	30	=40.60	0.00	0.02	30	=40.60	0.00
	1024	60.50	7.01	=60.57	0.34	0.35	30	=60.57	0.01	0.04	30	=60.57	0.09
	2048	88.00	22.42	=88.00	2.56	2.59	30	=88.00	0.05	8.89	30	=88.00	0.33
	4096	127.20	37.41	+**127.37** *	2.93	2.97	30	+**127.37** *	0.15	1.50	30	+**127.37** *	2.17

Table C.2: Results obtained after graph reduction (RFLCS instances of RFLCS-SET2).

$\|\Sigma\|$	*reps*	Spec. Tech.		CPLEX				LMC				LSCC-BMS	
		result	$\bar{t}$	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$
4	3	3.47	0.00	=3.47	0.00	0.00	30	=3.47	0.00	0.02	30	=3.47	0.00
	4	3.77	0.00	=3.77	0.00	0.00	30	=3.77	0.00	0.02	30	=3.77	0.00
	5	3.83	0.00	=3.83	0.00	0.00	30	=3.83	0.00	0.02	30	=3.83	0.00
	6	3.90	0.00	=3.90	0.00	0.00	30	=3.90	0.00	0.02	30	=3.90	0.00
	7	3.97	0.00	=3.97	0.00	0.00	30	=3.97	0.00	0.02	30	=3.97	0.00
	8	3.97	0.00	=3.97	0.01	0.01	30	=3.97	0.00	0.02	30	=3.97	0.00
8	3	6.23	0.00	=6.23	0.00	0.00	30	=6.23	0.00	0.02	30	=6.23	0.00
	4	6.87	0.00	=6.87	0.00	0.00	30	=6.87	0.00	0.02	30	=6.87	0.00
	5	7.40	0.00	=7.40	0.00	0.00	30	=7.40	0.00	0.02	30	=7.40	0.00
	6	7.53	0.01	=7.53	0.01	0.01	30	=7.53	0.00	0.02	30	=7.53	0.00
	7	7.70	0.00	=7.70	0.02	0.02	30	=7.70	0.00	0.02	30	=7.70	0.00
	8	7.77	0.00	=7.77	0.02	0.02	30	=7.77	0.00	0.02	30	=7.77	0.00
16	3	9.70	0.01	=9.70	0.00	0.00	30	=9.70	0.00	0.02	30	=9.70	0.00
	4	11.57	0.01	=11.57	0.01	0.01	30	=11.57	0.00	0.02	30	=11.57	0.00
	5	12.93	0.01	=12.93	0.02	0.02	30	=12.93	0.00	0.02	30	=12.93	0.00
	6	14.00	0.01	=14.00	0.05	0.05	30	=14.00	0.00	0.02	30	=14.00	0.00
	7	14.93	0.22	=14.93	0.10	0.10	30	=14.93	0.00	0.04	30	=14.93	0.00
	8	14.80	0.39	=14.80	0.17	0.17	30	=14.80	0.00	0.05	30	=14.80	0.01
32	3	16.13	0.02	=16.13	0.01	0.01	30	=16.13	0.00	0.02	30	=16.13	0.00
	4	19.00	0.07	=19.00	0.02	0.03	30	=19.00	0.00	0.02	30	=19.00	0.00
	5	21.63	0.37	=21.63	0.28	0.30	30	=21.63	0.00	0.03	30	=21.63	0.01
	6	23.73	0.48	=23.73	0.62	0.70	30	=23.73	0.00	0.06	30	=23.73	0.01
	7	25.53	0.78	=25.57	1.79	1.93	30	=25.57	0.02	0.13	30	=25.57	0.02
	8	27.40	5.02	=27.50	2.49	2.67	30	=27.50	0.07	0.29	30	=27.50	0.05
64	3	25.43	0.06	=25.43	0.02	0.02	30	=25.43	0.00	0.02	30	=25.43	0.00
	4	30.37	0.77	=30.37	0.24	0.26	30	=30.37	0.00	0.03	30	=30.37	0.01
	5	34.87	1.09	=34.93	1.46	2.21	30	=34.93	0.01	0.09	30	=34.93	0.04
	6	39.07	10.66	=39.13	5.46	8.21	30	=39.13	0.04	0.25	30	=39.13	0.11
	7	43.50	24.28	=43.63	11.06	24.01	30	=43.63	0.16	0.79	30	=43.63	0.18
	8	45.17	35.41	=45.53	35.48	84.08	30	=45.53	1.69	6.06	30	=45.53	0.44
128	3	36.70	0.67	=36.77	0.18	0.18	30	=36.77	0.00	0.03	30	=36.77	0.02
	4	44.90	4.83	=45.03	1.99	2.67	30	=45.03	0.02	0.11	30	=45.03	0.15
	5	53.23	17.58	=53.43	7.90	10.76	30	=53.43	0.12	0.42	30	=53.43	0.30
	6	61.07	34.68	=61.53	28.13	46.99	30	=61.53	4.27	6.76	30	=61.53	0.98
	7	67.90	52.14	+**68.47**	125.08	421.63	30	=68.47	9.47	57.64	30	=68.47	1.98
	8	73.57	105.79	+**74.50**	554.65	1321.01	18	=74.30	608.14	544.04	13	=74.60	10.65
256	3	54.97	0.30	=55.03	0.74	0.79	30	=55.03	0.02	0.06	30	=55.03	0.04
	4	68.70	1.74	=68.93	5.61	6.63	30	=68.93	0.10	1.64	30	=68.93	0.79
	5	81.00	29.22	=81.43	28.37	40.16	30	=81.43	3.51	7.68	30	=81.43	3.13
	6	93.10	36.48	+**93.60** *	227.62	476.17	30	+**93.17**	124.54	309.18	17	+**93.60** *	47.19
	7	103.50	98.94	+**104.27**	667.00	1338.10	24	−*103.13*	156.41	176.14	3	+**104.47** *	262.17
	8	113.70	176.80	+**112.07**	2277.45	1367.53	1	=113.10	344.01	– –	0	+**115.00** *	1009.44
512	3	81.57	29.98	=81.63	0.52	0.54	30	=81.63	0.02	0.35	30	=81.63	0.12
	4	100.83	14.56	+**101.13**	10.22	10.99	30	+**101.10**	17.70	23.64	30	=101.13	4.44
	5	120.43	92.05	+**121.03** *	147.62	209.48	30	+**120.03**	249.84	866.25	14	+**121.03** *	226.89
	6	137.03	128.19	+**136.97**	1335.77	771.58	11	+**136.47**	499.84	47.78	1	+**137.80** *	1064.07
	7	154.57	295.09	+**111.40**	2118.90	– –	0	−*152.27*	823.67	– –	0	+**153.33**	1619.19
	8	172.10	356.01	+**13.97**	577.60	– –	0	−*169.73*	620.92	– –	0	+**168.70**	1557.06

Table C.3: Results for LAPCS instances of set Lapcs-Arti after graph reduction.

n	n_{arcs}	Spec. Tech		Cplex				Lmc				Lscc-Bms	
		result	$\bar{t}$	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$	$\bar{t}_{\text{opt}}$	**#opt**	**result**	$\bar{t}$
100	10	60.17[a]	39.01	=60.20	6.57	8.56	30	=60.20	38.48	85.52	30	=60.20	0.46
	20	58.13[a]	52.11	=58.20	12.46	22.41	30	=58.17	15.96	361.00	29	=58.20	0.80
	50	51.87[a]	50.78	=52.03	637.48	1150.86	24	=52.07	145.51	1105.66	21	=52.10	2.48
200	20	121.70[b]	38.28	+**122.60**	956.11	1072.96	22	−*120.20*	635.33	– –	0	+**122.63** *	671.81
	40	116.70[b]	61.36	+**111.80**	2475.25	1972.97	4	−*115.80*	627.34	– –	0	+**118.40** *	748.94
	100	104.57[a]	143.72	+**0.07**	302.02	– –	0	=104.30	539.94	– –	0	+**106.87** *	1048.04
300	30	181.30[a]	178.30	+**22.10**	659.71	– –	0	+**178.23**	643.25	– –	0	+**178.63**	1495.86
	60	174.97[a]	157.58	– –	– –	– –	0	=171.80	507.46	– –	0	+**172.20**	1338.14
	150	157.13[a]	262.94	– –	– –	– –	0	+**155.97**	991.12	– –	0	+**156.97**	1410.26
400	40	242.70[b]	191.72	– –	– –	– –	0	+**239.73**	650.89	– –	0	+**230.77**	1616.86
	80	233.23[a]	322.25	– –	– –	– –	0	=226.97	578.27	– –	0	+**221.80**	1657.25
	200	208.77[a]	378.41	– –	– –	– –	0	−*205.17*	705.84	– –	0	+**202.27**	2035.29
500	50	302.27[b]	250.46	– –	– –	– –	0	−*295.83*	847.81	– –	0	+**278.47**	1819.17
	100	291.23[a]	181.52	– –	– –	– –	0	+**284.70**	891.25	– –	0	+**266.70**	1470.81
	250	259.50[a]	498.75	– –	– –	– –	0	−*255.80*	1116.97	– –	0	+**242.57**	1686.56
600	60	366.03[b]	324.61	– –	– –	– –	0	+**356.83**	773.68	– –	0	+**323.13**	1598.87
	120	350.97[a]	580.70	– –	– –	– –	0	+**341.40**	987.06	– –	0	– –	– –
	300	309.20[a]	370.43	– –	– –	– –	0	+**306.63**	906.20	– –	0	– –	– –
700	70	418.40[b]	372.52	– –	– –	– –	0	+**386.93**	708.66	– –	0	– –	– –
	140	400.60[a]	6.17	– –	– –	– –	0	+**40.23**	169.30	– –	0	– –	– –
	350	362.74[a]	698.11	– –	– –	– –	0	– –	– –	– –	0	– –	– –
800	80	484.43[b]	420.71	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	160	462.60[b]	572.70	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	400	414.33[a]	797.52	– –	– –	– –	0	– –	– –	– –	0	– –	– –
900	90	542.07[b]	516.09	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	180	522.40[a]	534.96	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	450	463.27[a]	897.39	– –	– –	– –	0	– –	– –	– –	0	– –	– –
1000	100	605.10[b]	535.42	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	200	583.30[a]	889.74	– –	– –	– –	0	– –	– –	– –	0	– –	– –
	500	514.80[b]	664.53	– –	– –	– –	0	– –	– –	– –	0	– –	– –

Table C.4: Results for LAPCS instances of set Lapcs-Real after graph reduction.

Inst.	Spec. Tech.		Cplex			Lmc			Lscc-Bms	
Name	**results**	t	**result**	t	t_{opt}	**result**	t	t_{opt}	**result**	t
Real_1	268[b]	80.92	+**273***	339.70	339.76	=259	2489.55	– –	+**272**	847.60
Real_2	291[b]	84.93	+**291**	21.84	21.85	=283	416.57	– –	+**291**	903.86
Real_3	294[b]	171.78	– –	– –	– –	=284	49.69	– –	+**263**	1360.83
Real_4	374[b]	0.02	+**374**	0.02	0.02	=374	0.01	0.07	+**374**	0.00
Real_5	178[b]	58.96	+**179**	4.62	4.63	=179	1.38	351.62	+**179**	11.32
Real_6	209[b]	131.35	+**0**	2376.75	– –	=206	15.19	– –	+**204**	2441.29
Real_7	330[b]	7.40	+**330**	0.05	0.05	=330	0.73	1.12	+**330**	758.07
Real_8	177[b]	174.40	+**1**	1281.87	– –	=175	2760.20	– –	–	
textit172	3025.38									
Real_9	302[b]	24.89	+**304**	1.47	1.81	=304	16.41	54.18	+**304**	498.12
Real_10	361[a]	0.49	+**361**	0.23	0.23	=361	1.49	10.50	+**361**	74.89

APPENDIX D

CLCS Problem: Supplementary Material

This appendix presents the supplementary material on the computational studies from Chapter 7.

D.1 A short overview over the Algorithms Used for Comparison

Algorithm by Chin et al. [34]. This method is based on dynamic programming. It uses a three-dimensional matrix M to store the lengths of optimal solutions of subproblems $S_{i,j,k} = (s_1[1,i], s_2[1,j], P[1,k], \Sigma)$ for $i = 1,\ldots,|s_1|,\ j = 1,\ldots,|s_2|,\ k = 1,\ldots,|P|$. All these values are obtained recursively on the basis of solutions to smaller subinstances for which optimal values are already known. In essence, the recursive procedure distinguishes the following cases and handles them appropriately: $s_1[i] = s_2[j] = P[k]$, $s_1[i] = s_2[j] \neq P[k]$, or $s_1[i] \neq s_2[j]$. In this way, optimal values of successor entries (representing larger subproblems) are determined in constant time. Due to its simplicity, the algorithm is fast for problem instances of small and medium size but its performance degrades for longer sequences. In general, its time and space complexity is $O(|s_1| \cdot |s_2| \cdot |P|)$.

Algorithm by Arslan and Eğecioğlu [5]. This approach replaces the matrix used in the original dynamic programming algorithm of Tsai [162] by multiple three-dimensional matrices in order to realize some calculations of the approach of Tsai more efficiently. In particular, the recurrence used by Tsai was simplified. In the end, this results in an algorithm with the same time complexity as the algorithm of Chin et al., however with a memory requirement that is by a factor of three higher.

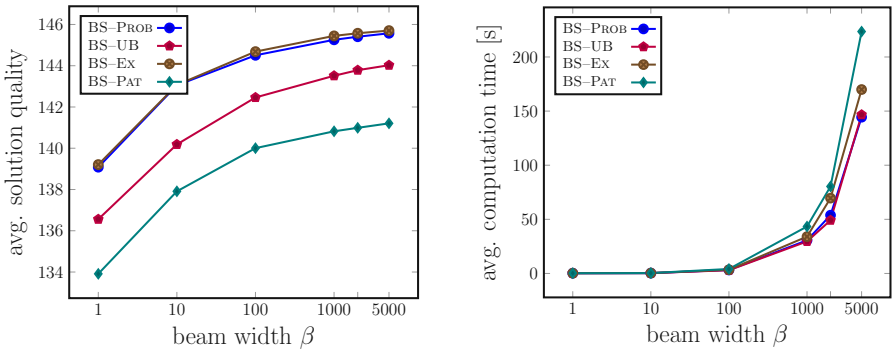
Algorithm by Iliopoulos and Rahman [88]. This method is based on a modification of the dynamic programming formulation from [5]. To perform the matrix calculations of each iteration efficiently, the authors make use of a so-called *bounded heap* data structure [25] that was realized by means of Van Emde Boas (vEB) trees [20]. This data structure allows to calculate intermediate results more efficiently in $O(\log \log n)$ time, leading to a total time complexity of $O(|P| \cdot R \cdot \log \log n + n)$, where R is the number of ordered pairs of positions at which input strings s_1 and s_2 match.

Algorithm by Hung et al. [86]. This method is a more recent development that is particularly suited for input strings that are highly similar. It was developed on the basis of the so-called diagonal concept for the LCS problem by Nakatsu et al. [136]. In general it can be said that the efficiency of the algorithm grows with the length of an optimal CLCS solution. The algorithm uses a table D of dimension $|P| \times L$, where L is an upper bound for the CLCS length. Each cell $D_{i,l}$ stores a triple associated with a partial solution. At each iteration of the algorithm some of the cells are filled with information such that for any triple $(i', j, k) \in D_{i,l}$, where $i' = 1, \ldots, i$, the relation $|\text{CLCS}(s_1[1, i'], s_2[1, j], P[1, |P| - k])| \geq l$ holds. The elements belonging to $D_{i,l}$ are determined by extending all the partial solutions from $D_{i-1,l-1}$, to which all the partial solutions of $D_{i-1,l}$ are added, and by filtering out dominated pairs. If $(i', j, 0) \in D_{i,l}$ and there is no other $(i'', j'', 0) \in D_{i,l}$ with $i' \neq i''$ and $j \neq j''$, it implies that $|\text{CLCS}(s_1[1, i'], s_2[1, j], P)| = l$. In this way an optimal solution is found for the specific subproblem.

Algorithm by Deorowicz [49]. Just like the previous approach, this algorithm is a so-called sparse approach. The matrix utilized for the calculations 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 an optimal solution) the lowest possible column number. Furthermore, a two-dimensional matrix T is used to store computed values from the current and previous levels. For each row i and column j where $s_1[i] = s_2[j]$, the list entries are recalculated. 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. On completion, the highest rank in the list corresponds to the length of an optimal solution.

Improvements of Deorowicz's algorithm were introduced by Deorowicz and Obstoj [50]. They utilize so–called external–entry points (EEP) [81] initially proposed for the pairwise sequence alignment problem, for omitting those cells in the lists that do not contribute to optimal solutions.

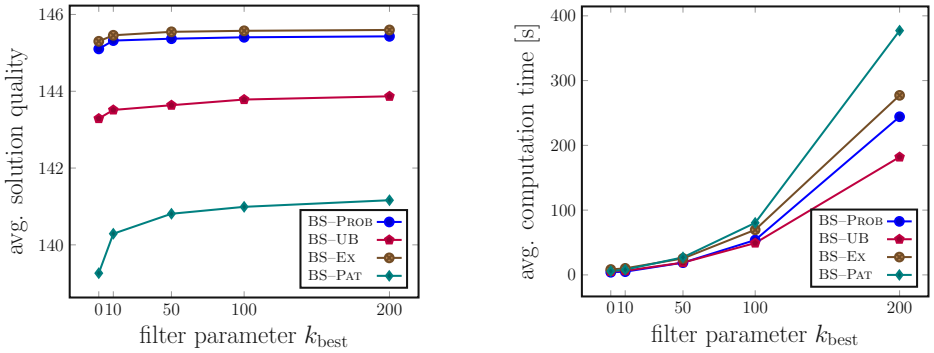
D.2 Tuning of β and k_{best} parameters for different Beam Search Configurations



(a) Average solution qualities (over all instances) ($k_{best} := 100$

(b) Average computation times over all instances

Figure D.1: Results of Beam search with $k_{\text{best}} = 100$ and varying β.



(a) Average solution qualities over all instances ($\beta := 2000$)

(b) Average computation times (over all instances)

Figure D.2: Results of Beam search with $\beta = 2000$ and varying k_{best}.

D.3 The Numerical Results on the Remaining m–CLCS Benchmark Sets

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

			Approx		Greedy		BS–UB		BS–Prob		BS–Ex		BS–Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	#	$\bar{t}[s]$
4	10	100	20.9	<0.1	30	<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	162	<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	329.9	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	20.4	<0.1	24.1	15.5	**24.2**	12.1	**24.2**	16.7	24	22	0	-
4	50	500	109.3	0.1	127.5	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	263.4	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	<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	117.2	0.2	128.4	135	131	118.2	**132.0**	115.2	127.6	160.2	0	-
4	100	1000	225.1	0.9	246.9	0.7	262.4	287.6	270.5	236.6	**272.1**	329.9	261.6	282	0	-
20	10	100	4.3	<0.1	6.8	<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	40.9	<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	82.9	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	24.2	0.1	28.3	49	**28.8**	46.8	**28.8**	100.3	26	135.5	0	-
20	50	1000	42.6	0.5	53.8	0.4	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	22.2	0.2	24.7	60.9	**25.2**	62.6	25.0	118.5	22.8	82.7	0	-
20	100	1000	41.1	1	48.8	1	52.8	166.2	54.7	188.6	**55.0**	334.8	50	342.7	0	-

Table D.2: Instances with $p' = \frac{|P|}{n} = \frac{1}{10}$.

			Approx		Greedy		BS–UB		BS–Prob		BS–Ex		BS–Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	#	$\bar{t}[s]$
4	10	100	22.9	<0.1	29.6	<0.1	**34.6**	14.4	**34.6**	17.4	**34.6**	18.91	23	8	269.1	
4	10	500	121.4	<0.1	163.7	<0.1	182.2	97.6	185.0	137	**186.0**	162.3	165.9	193.9	0	–
4	10	1000	245.5	0.1	329.1	0.1	365	212	375.8	240.5	**377.0**	271.3	330.4	391.7	0	-
4	50	100	19.8	<0.1	21.8	<0.1	24.9	10.1	25.0	11.2	**25.1**	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	**143.6**	129.8	131.2	145.9	0	-
4	50	1000	233.5	0.4	266.5	0.5	279.6	199	289.2	200.6	**290.4**	340.0	266	351.7	0	-
4	100	100	18.9	<0.1	20.8	<0.1	**23.0**	8.8	**23.0**	8.7	**23.0**	14.4	21.5	19.3	3	265.1
4	100	500	111.3	0.2	122	0.2	129.2	63.2	133.3	78.5	**134.2**	128.2	124.3	163.8	0	-
4	100	1000	230.3	0.9	253.2	0.7	262.3	122.7	270.9	183.3	**274.8**	273.9	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**	0.1	***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**	0.2	***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	***100.0**	0.8	***100.0**	0.8	***100.0**	1.1	***100.0**	1	10	0.9

Table D.3: Instances with $p' = \frac{|P|}{n} = \frac{1}{2}$.

			Approx		Greedy		BS–UB		BS–Prob		BS–Ex		BS–Pat		A*	
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	$\overline{\|s\|}$	$\bar{t}[s]$	#	$\bar{t}[s]$
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**	0.5	***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**	0.7	***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**	0.4	***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**	0.2	***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**	1	***500.0**	0.7	***500.0**	0.8	***500.0**	1	***500.0**	1.1	10	0.7

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 – the 56th Annual Symposium on Foundations of Computer Science*, pages 59–78. IEEE, 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] S. Aine, P. P. Chakrabarti, and R. Kumar. AWA* – A window constrained anytime heuristic search algorithm. In M. M. Veloso, editor, *Proceedings of IJCAI 2007 – Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2250–2255. ACM, 2007.

[4] H.-C. An, R. Kleinberg, and D. B. Shmoys. Improving christofides' algorithm for the st path tsp. *Journal of the ACM (JACM)*, 62(5):1–28, 2015.

[5] 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.

[6] M. Baghel, S. Agrawal, and S. Silakari. Survey of metaheuristic algorithms for combinatorial optimization. *International Journal of Computer Applications*, 58(19), 2012.

[7] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012.

[8] R. Beal, T. Afrin, A. Farheen, and D. Adjeroh. A new algorithm for "the LCS problem" with application in compressing genome resequencing data. *BMC Genomics*, 17(4):544, 2016.

[9] R. Bellman and S. Dreyfus. Functional approximations and dynamic programming. *Mathematical Tables and Other Aids to Computation*, pages 247–251, 1959.

[10] C. Berger. Solving a generalized constrained longest common subsequence problem. Master thesis, TU Wien, 2020.

[11] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[12] G. Blin, P. Bonizzoni, R. Dondi, and F. Sikora. On the parameterized complexity of the repetition free longest common subsequence problem. *Information Processing Letters*, 112(7):272–276, 2012.

[13] C. Blum and M. J. Blesa. Probabilistic beam search for the longest common subsequence problem. In T. Stützle, M. Birratari, and H. H. Hoos, editors, *Proceedings of SLS 2007 – the 1st International on Engineering Stochastic Local Search Algorithms*, volume 4638 of *LNCS*, pages 150–161. Springer, 2007.

[14] C. Blum and M. J. Blesa. A comprehensive comparison of metaheuristics for the repetition-free longest common subsequence problem. *Journal of Heuristics*, 24(3):551–579, 2018.

[15] C. Blum and M. J. Blesa. Hybrid techniques based on solving reduced problem instances for a longest common subsequence problem. *Applied Soft Computing*, 62:15–28, 2018.

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

[17] C. Blum, M. Djukanovic, A. Santini, H. Jiang, C.-M. Li, F. Manyà, and G. R. Raidl. Solving longest common subsequence problems via a transformation to the maximum clique problem. *Computers & Operations Research*, 125:105089, 2020.

[18] C. Blum and P. Festa. *Metaheuristics for String Problems in Bioinformatics*, volume 21. Wiley, 2011.

[19] C. Blum and G. R. Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer, 2016.

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

[21] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[22] P. Bonizzoni, G. Della Vedova, R. Dondi, and Y. Pirola. Variants of constrained longest common subsequence. *Information Processing Letters*, 110(20):877–881, 2010.

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

[24] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In *Proceedings of DAC'04 – the 41st Design Automation Conference*, pages 395–400. IEEE press, 2004.

[25] 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.

[26] J. W. Brown. The ribonuclease P database. *Nucleic acids research*, 26(1):351–352, 1998.

[27] A. Bundy and L. Wallen. *Breadth-First Search*, pages 13–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[28] S. Cai, K. Su, and Q. Chen. Ewls: A new local search for minimum vertex cover. In *Proceedings of AAAI-10 – the 24th AAAI Conference on Artificial Intelligence*, 2010.

[29] B. Calvo and G. Santafe. scmamp: Statistical comparison of multiple algorithms in multiple problems. *The R Journal*, 8(1):248–256, 2016.

[30] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990.

[31] H.-T. Chan, C.-B. Yang, and Y.-H. Peng. The generalized definitions of the two-dimensional largest common substructure problems. In *Proceedings of the 33rd Workshop on Combinatorial Mathematics and Computation Theory*, pages 1–12. National Taiwan University, Department of Mathematics, 2016.

[32] Y.-C. Chen and K.-M. Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, (3):383–392, 2016.

[33] K. Chimanga, J. Kalezhi, and P. Mumba. Application of best first search algorithm to demand control. In *Proceedings of 2016 IEEE PES PowerAfrica*, pages 51–55. IEEE, 2016.

[34] 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.

[35] 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.

[36] C. Q. Choi. DNA palindromes found in cancer. *Genome Biology*, 6:1–3, 2005.

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

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

[39] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12(2):306–315, 1975.

[40] A. A. Cire and W.-J. van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[41] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012.

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

[43] I. I. Cplex. V12. 1: User's manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[44] V. Dančík and M. Paterson. Upper bounds for the expected length of a longest common subsequence of two binary sequences. *Random Structures & Algorithms*, 6(4):449–458, 1995.

[45] M. J. De Smith, M. F. Goodchild, and P. Longley. *Geospatial analysis: a comprehensive guide to principles, techniques and software tools*. Troubador publishing ltd, 2007.

[46] T. L. Dean and M. S. Boddy. An analysis of time-dependent planning. In *Proceedings of AAAI-88 – the 7th National Conference on Artificial Intelligence*, volume 88, pages 49–54, 1988.

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

[48] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

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

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

[51] J. D. Dixon. Longest common subsequences in binary sequences. *ArXiv e-prints*, 2013. arxiv:1307.2796v1.

[52] M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. An A* search algorithm for the constrained longest common subsequence problem. *Information Processing Letters*, 166:106041, 2020.

[53] M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. On solving a generalized constrained longest common subsequence problem. In N. Olenev, Y. Evtushenko, M. Khachay, and V. Malkova, editors, *Proceedings of OPTIMA 2020 – the 11th International Conference Optimization and Applications*, volume 12422, pages 55–70, Cham, 2020. Springer International Publishing.

[54] 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.

[55] 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.

[56] 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.

[57] M. Djukanovic, G. R. Raidl, and C. Blum. Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, 114:104827, 2020.

[58] M. Djukanovic, G. R. Raidl, and C. Blum. Finding longest common subsequences: New anytime A* search results. *Applied Soft Computing*, 95:106499, 2020.

[59] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical computer science*, 344(2-3):243–278, 2005.

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

[61] P. A. Evans. *Algorithms and Complexity for Annotated Sequence Analysis*. PhD thesis, University of Victoria, 1999.

[62] P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Annual Symposium on Combinatorial Pattern Matching*, pages 270–280. Springer, 1999.

[63] D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. In *Proceedings of ICAPS'05 – the 15th International workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling*, pages 9–18, 2005.

[64] J. A. Filar, M. Haythorpe, and R. Taylor. Linearly-growing reductions of karp's 21 np-complete problems. *arXiv preprint arXiv:1902.10349*, 2019.

[65] C. B. Fraser. *Subsequences and Supersequences of Strings*. PhD thesis, University of Glasgow, Glasgow, UK, 1995.

[66] D. Furcy and S. Koenig. Limited discrepancy beam search. In *IJCAI*, 2005.

[67] R. A. Gallego, R. Romero, and A. J. Monticelli. Tabu search algorithm for network synthesis. *IEEE Transactions on Power Systems*, 15(2):490–495, 2000.

[68] S. García and F. Herrera. An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. *Journal of Machine Learning Research*, 9:2677 – 2694, 2008.

[69] M. Gendreau, J.-Y. Potvin, et al. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[70] M. Giel-Pietraszuk, M. Hoffmann, S. Dolecka, J. Rychlewski, and J. Barciszewski. Palindromes in proteins. *Journal of Protein Chemistry*, 22(2):109–113, 2003.

[71] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

[72] A. Gorbenko and V. Popov. The c-fragment longest arc-preserving common subsequence problem. *IAENG International Journal of Computer Science*, 39(3):231–238, 2012.

[73] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein. Restricted LCS. In *Proceedings of SPIRE 2010 – the 17th Int. Symposium on String Processing and Information Retrieval*, volume 6394 of *LNCS*, pages 250–257. Springer, 2010.

[74] 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.

[75] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. *ACM Transactions on Algorithms*, 2(1):44–65, 2006.

[76] E. A. Hansen and R. Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28(1):267–297, 2007.

[77] E. A. Hansen, S. Zilberstein, and V. A. Danilchenko. Anytime heuristic search: First results. Technical report, University of Massachusetts, Amherst, MA, USA, 1997.

[78] P. Hansen, N. Mladenović, and J. A. M. Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010.

[79] 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.

[80] M. M. Hasan, A. S. M. Sohidull Islam, M. Sohel Rahman, and A. Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11:219–232, 2017.

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

[82] M. Horn, M. Djukanovic, C. Blum, and G. R. Raidl. On the use of decision diagrams for finding repetition-free longest common subsequences. In *Proceedings of OPTIMA 2020 – the 11th International Conference Optimization and Applications*, pages 134–149. Springer, 2020.

[83] E. Horvitz and G. Rutledge. Time-dependent utility and action under uncertainty. In *Uncertainty Proceedings 1991*, pages 151–158. Elsevier, 1991.

[84] G. Huang and A. Lim. An effective branch-and-bound algorithm to solve the k-longest common subsequence problem. In *Proceedings of ECAI 2004 – the 16th European Conference on Artificial Intelligence*, pages 191–195. IOS Press, 2004.

[85] K. Huang, C.-B. Yang, K.-T. Tseng, et al. Fast algorithms for finding the common subsequence of multiple sequences. In *Proceedings of ICS 2004 – the 13th International Computer Symposium*, pages 1006–1011. IEEE press, 2004.

[86] 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.

[87] 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.

[88] 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.

[89] S. Inenaga and H. Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018. Supplement C.

[90] S. Inenaga and H. Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129(C):11–15, 2018.

[91] T. Inoue, S. Inenaga, H. Hyyrö, H. Bannai, and M. Takeda. Computing longest common square subsequences. In *In Proceedings of CPM 2018 – the 29th Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2018.

[92] A. Islam and D. Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Transactions on Knowledge Discovery from Data*, 2(2):1–25, 2008.

[93] M. R. Islam, C. K. Saifullah, Z. T. Asha, and R. Ahamed. Chemical reaction optimization for solving longest common subsequence problem for multiple string. *Soft Computing*, 23(14):5485–5509, 2019.

[94] H. Jiang, C.-M. Li, and F. Manyà. Combining efficient preprocessing and incremental MaxSAT reasoning for MaxClique in large graphs. In *Proceedings of ECAI 2016 – the 22nd European Conference on Artificial Intelligence*, pages 939–947, 2016.

[95] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[96] 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.

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

[98] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

[99] G. K. Kao, E. C. Sewell, and S. H. Jacobson. A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem. *Journal of Scheduling*, 12(2):163–175, 2009.

[100] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of STOC'84 – the 16th annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[101] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 – the 10th International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[102] A. Kershenbaum and R. R. Boorstyn. Centralized teleprocessing network design. *Networks*, 13(2):279–293, 1983.

[103] S. Khuri, T. Bäck, and J. Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of SAC94 – the ACM symposium on Applied computing*, pages 188–193, 1994.

[104] M. Kiwi, M. Loebl, and J. Matoušek. Expected length of the longest common subsequence for large alphabets. *Advances in Mathematics*, 197(2):480–498, 2005.

[105] V. Klee and G. J. Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

[106] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[107] P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Proceedings of AMTA 2004 – the 4th Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer, 2004.

[108] P. J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management science*, 13(9):723–735, 1967.

[109] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[110] S. Kosub. A note on the triangle inequality for the Jaccard distance. *Pattern Recognition Letters*, 120:36–38, 2019.

[111] A. Kovács, K. N. Brown, and S. A. Tarim. An efficient mip model for the capacitated lot-sizing and scheduling problem with sequence-dependent setups. *International Journal of Production Economics*, 118(1):282–291, 2009.

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

[113] A. Kumar, S. Vembu, A. K. Menon, and C. Elkan. Beam search algorithms for multilabel learning. *Machine learning*, 92(1):65–89, 2013.

[114] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

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

[116] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[117] S. Larionov, A. Loskutov, and E. Ryadchenko. Chromosome evolution with naked eye: Palindromic context of the life origin. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 18(1), 2008.

[118] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[119] E. K. Lee, T. Easton, and K. Gupta. Novel evolutionary models and applications to sequence alignment problems. *Annals of Operations Research*, 148(1):167–187, 2006.

[120] A. S. Lhoussain, G. Hicham, and Y. Abdellah. Adaptating the Levenshtein distance to contextual spelling correction. *International Journal of Computer Science and Applications*, 12(1):127–133, 2015.

[121] C.-M. Li, H. Jiang, and F. Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017.

[122] C. M. Li and F. Manya. Maxsat, hard and soft constraints. In *Handbook of satisfiability*, volume 185, pages 613–631. 2009.

[123] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang. A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments. In *Proceedings of ICDE 2016 – the 32nd International Conference on Data Engineering*, pages 1170–1181. IEEE, 2016.

[124] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS'04 – the 17th International Conference on Neural Information Processing Systems*, pages 767–774, 2004.

[125] G. O. LLC. Gurobi optimizer reference manual, 2018.

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

[127] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

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

[129] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, 2002.

[130] M. Martínez-Porchas and F. Vargas-Albores. An efficient strategy using k-mers to analyse 16s rrna sequences. *Heliyon*, 3(7):e00370, 2017.

[131] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, 1:65–77, 2002.

[132] L. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18(1):24–34, 1970.

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

[134] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79 – 102, 2016.

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

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

[137] D. S. Nau, V. Kumar, and L. Kanal. General branch and bound, and its relation to A^* and AO*. *Artificial Intelligence*, 23(1):29–58, 1984.

[138] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[139] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of CP 2007 – the 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[140] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

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

[142] Z. Peng and Y. Wang. A novel efficient graph model for the multiple longest common subsequences (MLCS) problem. *Frontiers in Genetics*, 8:104, 2017.

[143] C. Prud'homme, J.-G. Fages, and X. Lorca. Choco documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, 6241:64–70, 2014.

[144] L. Rabiner, A. Rosenberg, and S. Levinson. Considerations in dynamic time warping algorithms for discrete word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(6):575–582, 1978.

[145] 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.

[146] K. Rieck, P. Laskov, and K.-R. Müller. Efficient algorithms for similarity measures over sequential data: A look beyond kernels. In *Proccedings of DAGM 2006 – the 28th Joint Pattern Recognition Symposium*, pages 374–383. Springer, 2006.

[147] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[148] R. Ruiz and T. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European journal of operational research*, 177(3):2033–2049, 2007.

[149] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[150] D. Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999.

[151] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

[152] 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.

[153] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computación y Sistemas*, 18(3):491–504, 2014.

[154] S. Sivanandam and S. Deepa. *Genetic algorithm optimization problems*. Springer, 2008.

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

[156] T. Stützle and R. Ruiz. *Iterated Greedy*, pages 547–577. Springer International Publishing, Cham, 2018.

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

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

[159] H. Tanaka, D. A. Bergstrom, M.-C. Yao, and S. J. Tapscott. Large DNA palindromes as a common form of structural chromosome aberrations in human cancers. *Human Cell*, 19(1):17–23, 2006.

[160] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[161] Y. Tsai and J. Hsu. An approximation algorithm for multiple longest common subsequence problems. In *Proceedings of SCI 2020 – the 6th world multiconference on systemics, cybernetics and informatics*, pages 456–460, 2002.

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

[163] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. MAWA* – A Memory–Bounded Anytime Heuristic–Search Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 41(3):725–735, 2011.

[164] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. Anytime pack search. *Natural Computing*, 15(3):395–414, 2016.

[165] S. G. Vadlamudi, P. Gaurav, S. Aine, and P. P. Chakrabarti. Anytime column search. In *Proceedings of AI 2012 – the 16th Australasian Joint Conference on Artificial Intelligence*, pages 254–265. Springer, 2012.

[166] J. van den Berg, R. Shah, A. Huang, and K. Y. Goldberg. Anytime nonparametric A*. In *Proceedings of AAAI 2011 – the 25th Conference on Artificial Intelligence, San Francisco, California, USA, August 7-11, 2011*, 2011.

[167] W.-J. Van Hoeve. Operations research techniques in constraint programming. *Tepper School of Business*, page 532, 2005.

[168] P. J. Van Laarhoven and E. H. Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[169] K. Viswanathan and A. Bagchi. Best-first search methods for constrained two-dimensional cutting stock problems. *Operations Research*, 41(4):768–776, 1993.

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

[171] S. Wan, Y. Lan, J. Xu, J. Guo, L. Pang, and X. Cheng. Match-SRNN: Modeling the recursive matching structure with spatial RNN. In *Proceedings of IJCAI'16 – the 25th International Joint Conference on Artificial Intelligence*, pages 2922–2928. AAAI Press, 2016.

[172] Q. Wang, D. Korkin, and Y. Shang. Efficient dominant point algorithms for the multiple longest common subsequence (MLCS) problem. In *Proceedings of IJCAI'09 – the 25th International Joint Conference on Artificial Intelligence*, pages 1494–1499, 2009.

[173] Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (MLCS) algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):321–334, 2011.

[174] 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.

[175] Y. Wang, S. Cai, and M. Yin. Two efficient local search algorithms for maximum weight clique problem. In *Proceedings of AAAI 2016 – the 30th Conference on Artificial Intelligence*, pages 805–811, 2016.

[176] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[177] L. A. Wolsey. *Heuristic analysis, linear programming and branch and bound*. Springer, 1980.

[178] I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5):249–253, 2005.

[179] J. Yang, Y. Xu, Y. Shang, and G. Chen. A space-bounded anytime algorithm for the multiple longest common subsequence problem. *IEEE Transactions on Knowledge and Data Engineering*, 26(11):2599–2609, 2014.

[180] 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, 2013.

[181] Y. Ye, J. Jiang, B. Ge, Y. Dou, and K. Yang. Similarity measures for time series data classification using grid representation and matrix distance. *Knowledge and Information Systems*, 60(2):1105–1134, 2019.

[182] W. Zhang. Complete anytime beam search. In *Proceedings of AAAI '98/IAAI '98 – the 15th National/10th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 425–430. AAAI Press, 1998.

[183] R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of ICAPS'05 – 15th International Conference on International Conference on Automated Planning and Scheduling*, pages 90–98. AAAI Press, 2005.

[184] A. Zielezinski, S. Vinga, J. Almeida, and W. M. Karlowski. Alignment-free sequence comparison: benefits, applications, and tools. *Genome biology*, 18(1):186, 2017.

[185] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73–73, 1996.

[186] S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms. In *Imprecise and approximate computation*, pages 43–62. Springer, 1995.

[187] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.

[188] S. V. Znamenskij. Approximation of the longest common subsequence length for two long random strings. *Program Systems: Theory and Applications*, 7(4):347–358, 2016.

Curriculum Vitae of Marko Djukanović

Personal Information:

Email	djukanovic.marko90@gmail.com
Date of birth	October 20, 1990

Education:

since 2017	TU Wien, Vienna, Austria
Field of Studies	**Doctoral programme in Engineering Sciences: Computer Science**
Thesis	Exact and Heuristic Approaches for Solving String Problems from Bioinformatics
Advisor	Ao.Univ.Prof. Dipl.-Ing.Dr.techn. Günther Raidl
2014 – 2016	University of Banja Luka, Bosnia and Herzegovina
	Master of Science
Field of Studies	Mathematics
Thesis	Numerical Construction of anti-Gaussian Quadratures
Advisor	prof. dr Miroslav Pranić
2009 – 2013	University of Banja Luka, Bosnia and Herzegovina
	Bachelor of Science
Field of Studies	Mathematics and Informatics

Personal Activities:

since 2017	**Researcher**
	Algorithms and Complexity Group
	Institute of Logic and Computation
2015 – 2016	**Software developer (part time)**
	Bitlab d.o.o, Banja Luka, B&H
2014 – 2017	**Teaching assistant**
	Faculty of Natural Science and Mathematics
	University of Banja Luka
2013 – 2014	**Software developer**
	Inova d.o.o, Banja Luka, B&H

Publications:

2021 C. Blum, M. Djukanovic, A. Santini, H. Jiang, C.-M. Lie, F. Manya, G. Raidl.
Solving longest common subsequence problems via a transformation to the maximum clique problem. *Computers & Operations Research*. 125:105089, 2021.

2020 M. Djukanovic, G. R. Raidl, and C. Blum.
Finding longest common subsequences: New anytime A* search results.
Applied Soft Computing, 95:106499, 2020.

M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum.
An A* search algorithm for the constrained longest common subsequence problem.
Information Processing Letters, 166:106041, 2020.

M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum.
On solving a generalized constrained longest common subsequence problem. *In Proceedings of OPTIMA 2020 – the 11th International Conference Optimization and Applications*, volume 12422, pages 55–70, Cham, 2020. Springer International Publishing.

M. Horn, M. Djukanovic, C. Blum, and G. R. Raidl.
On the use of decision diagrams for finding repetition-free longest common subsequences. *In Proceedings of OPTIMA 2020 – the 11th International Conference Optimization and Applications*, pages 134–149. Cham, 2020. Springer International Publishing.

2019 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.

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*, vol. 11943, pages 154–167. Springer, 2019.

M. Djukanovic, G. R. Raidl, and C. Blum.
Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, 114:104827, 2020.

2018 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.