universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Novel Methods in Utilizing Multivalued Decision Diagrams for Solving Repetition-Free Longest Common Subsequence Problems

verfasst von | submitted by

Georg Braun BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Ass.-Prof. Dr. Kathrin Hanauer B.Sc. M.SC.
Mitbetreut von | Co-Supervisor:	Dr.techn. Maximilian Vötsch BSc MSc

Acknowledgements

I would like to thank Enes Bajrović for his support during the early stages of this work. I am also grateful to my supervisors, Kathrin Hanauer and Maximilian Vötsch, for their guidance and valuable feedback throughout all stages of the research.

Abstract

This work presents an enhanced version of an existing multivalued decision diagram (*MDD*) algorithm for solving the Repetition-Free Longest Common Subsequence (*RFLCS*) problem, which involves finding the longest subsequence common to two strings without repeated characters. Our approach combines a randomized heuristic that consistently achieves optimal solutions with minimal exceptions and an *MDD* utilized in a novel way. The new approaches applied to the *MDD* lead to better *MDD* growth behaviour and enhanced runtime efficiency. We employ extensive experiments and comparisons to demonstrate the superiority of our method, showing substantial gains in both solution quality and computational efficiency across a wide range of common problem instances for the problem. Furthermore, our approach successfully solves problem instances that were previously unsolved in the literature, marking a significant advancement in the field.

Kurzfassung

Diese Arbeit stellt eine verbesserte Version eines Algorithmus auf Basis mehrwertiger Entscheidungsdiagramme (Multivalued Devision Diagram *MDD*) zur Lösung des Wiederholungsfreien Längsten Gemeinsamen Teilsequenz-Problems (Repetition-Free Longest Common Subsequence *RFLCS*) vor, bei dem die längste gemeinsame Teilsequenz zweier Zeichenketten ohne Wiederholungen von Zeichen gesucht wird. Unser Ansatz integriert eine randomisierte Heuristik, die in der Praxis konsistent optimale Lösungen findet, und nutzt *MDDs* auf neuartige Weise, um die Leistung zu verbessern. Diese Neuerungen führen zu einer reduzierten Größe der *MDDs* und schnelleren Laufzeiten. Umfangreiche Experimente zeigen Verbesserungen sowohl in der Lösungsqualität als auch in der Effizienz über eine Vielzahl von Benchmark-Instanzen. Darüber hinaus löst unsere Methode Instanzen, die bisher in der Literatur als ungelöst galten, und stellt somit einen bedeutenden Fortschritt gegenüber bestehenden Ansätzen dar.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

The Longest Common Subsequence (LCS) problem is a fundamental and well-established problem in computer science, with wide-ranging applications in fields such as bioinformatics [PL88, AGM+90], text similarity analysis [BNM17], and version control systems [HM76]. The objective of the LCS problem is identifying the longest subsequence that is common to two sequences, where the subsequence preserves the order of characters but not necessarily their consecutiveness. Over the years, various efficient algorithms have been developed to solve this problem [HS77, IR09, KHO21].

However, the Repetition-Free Longest Common Subsequence (*RFLCS*) problem introduces an additional layer of complexity. In the *RFLCS* variant, the objective is to find the longest subsequence common to two given sequences, while ensuring that no character appears more than once within the subsequence. This added constraint significantly increases the problem's complexity, making it NP-hard. The *RFLCS* problem is particularly useful in applications where avoiding duplicates is essential, such as in genome family analysis.

One of the earliest and most influential contributions to the *RFLCS* problem came from Sankoff [San99], who introduced the *exemplar model* in the context of genome analysis. This model led to further research, including the formal definition of the *RFLCS* problem by Adi et al. [ABF+08]. Despite its relevance, the *RFLCS* problem has received comparatively little attention in the literature, leaving opportunities for further research and improvement.

The current research on solving the *RFLCS* problem includes both heuristic methods and exact solvers. A notable contribution of an optimal solver is the work by Horn et al. [HDBR20], who employed a Multivalued Decision Diagram (*MDD*) approach. Their method has been successful in solving previously unknown optimal solutions and has also provided improved heuristic results compared to earlier works. This *MDD*-based approach represents a meaningful step forward in the field, demonstrating the potential of *MDD*s in the *RFLCS* context.

In this thesis, we aim to improve the current state of the art by proposing a randomized heuristic and introducing techniques that utilize the Multivalued Decision Diagram in a new way. Notably, our approach is capable of finding previously unknown optimal solutions and offers a heuristic that, with few exceptions, always finds the optimal solution. Through a series of extensive experiments, we will evaluate the performance of this algorithm and compare it against the current state-of-the-art methods.

This thesis is structured as follows: Chapter 2 defines the formal problem and provides the necessary theoretical background; Chapter 3 reviews the state of the art, highlighting key approaches to solving the *RFLCS* problem, including both exact and heuristic methods; Chapter 4 presents the proposed algorithm, detailing the novel techniques introduced; Chapter 5 discusses the experimental results and evaluates the performance of the algorithm; and Chapter 6 concludes with directions for future research. Additionally, Appendix A introduces and discusses a new Integer Linear Programming (ILP) model based on a computed *MDD*. Appendix B provides additional experimental results, and Appendix C contains a necessary proof of correctness.

2. Problem Definition

The *Repetition-Free Longest Common Subsequence Problem* is defined as follows: Given two strings $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_n)$, with characters over an alphabet Σ, the goal is to find the longest subsequence

$$C = (c_1, c_2, \ldots, c_{\ell(A,B)})$$

of length $\ell(A, B)$, such that C is a subsequence of both A and B, and each character appears at most once in C. Formally, the sequence C consists of elements $c_1, c_2, \ldots, c_{\ell(A,B)}$, where there exist indices

$$1 \leq i_1 < i_2 < \cdots < i_\ell \leq m, \quad 1 \leq j_1 < j_2 < \cdots < j_\ell \leq n$$

such that:

$$c_1 = a_{i_1} = b_{j_1}, \quad c_2 = a_{i_2} = b_{j_2}, \quad \ldots, \quad c_\ell = a_{i_\ell} = b_{j_\ell}.$$

Additionally, C must be *repetition-free*, meaning that $\neg\exists p, q$: such that $p \neq q \wedge c_p = c_q$.

Note that C is not a function of A and B because it is not unique, but $\ell(A, B)$ is indeed a function of A and B.

Example 2.0.1. To demonstrate an *RFLCS*, consider the sequences $A =$ `daebabadceadca` and $B =$ `aedcdcababdbed`, as shown in Figure 2.1. One possible *RFLCS* $C =$ `dabe`, which corresponds to the indices $I = 1, 2, 4, 10$ in A and $J = 3, 7, 8, 13$ in B. Note that the index sequences I and J are strictly increasing, and that C contains no repeating characters.

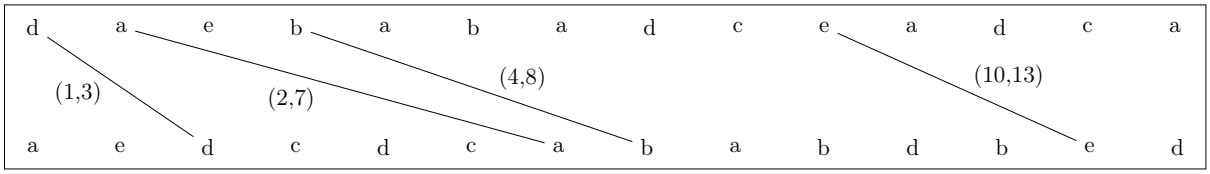


Figure 2.1.: Two strings, A and B, with the highlighting of an *RFLCS* C, showing the corresponding indices of the characters in the solution sequence within strings A and B.

3. Related Work

In this section, we first address the hardness of the *RFLCS* problem and the expected lengths of optimal solutions. We then review various approaches to solve *RFLCS*, including heuristic methods and optimal solvers.

3.1. Hardness of the Problem

While the *LCS* problem is solvable in polynomial time [HS77], Adi et al. [ABF$^+$08], in their introduction of the *RFLCS* problem, established that *RFLCS* is APX-hard [Joh73]. This implies that there exists no polynomial-time approximation scheme for *RFLCS*, unless $P = NP$.

Their proof involves a reduction from a variant of the Max 2-SAT problem to *RFLCS*, leveraging known hardness results for Max 2-SAT to demonstrate the complexity of *RFLCS*. The APX-hardness of *RFLCS* implies its NP-hardness. Additionally, the APX-hardness result implies that *RFLCS* does not admit a polynomial-time approximation scheme (PTAS) unless P = NP.

3.2. Expected RFLCS Lengths for Random Strings

Fernandes et al. [FK16] investigated the asymptotic behavior of the expected lengths of optimal solutions to the *RFLCS* problem, assuming that both strings A and B have equal length m and each character in the input strings is randomly, uniformly, and independently distributed over a common alphabet Σ, where $|\Sigma| = k$. They argue that understanding the behavior of random input strings can, for instance, aid in distinguishing between random noise and actual similarities in the *RFLCS* context.

Three key results were presented, demonstrating how the lengths of two input strings of equal length ($m = n$) grow in relation to the size of the alphabet, which changes with the size of the input. These results are as follows:

1. If $m = o\left(k\sqrt{k}\right)$, then
$$\lim_{m\to\infty} \frac{\mathbb{E}(l(A,B))}{m/\sqrt{k(m)}} = 2.$$
In this regime, the expected length of the *RFLCS* grows slower than the length of the strings, scaled by $\sqrt{k}$. This indicates that for strings growing slower than $k\sqrt{k}$, the alphabet size plays a role in limiting the expected length of the *RFLCS*.

2. If $m = \frac{1}{2}\rho k\sqrt{k}$ for $\rho > 0$, then
$$\lim_{m\to\infty} \frac{\mathbb{E}(l(A,B))}{k(m)} \geq 1 - e^{-\rho}.$$
In this regime, as the string length grows relative to $k\sqrt{k}$, the expected *RFLCS* length approaches a limit determined by ρ. This regime marks the transition where the *RFLCS* length becomes less constrained by the alphabet size.

3. If $m = \left(\frac{1}{2} + \xi\right) k\sqrt{k} \ln k$ for some $\xi > 0$, then

$$\lim_{m\to\infty} \frac{\mathbb{E}(l(A,B))}{k(m)} = 1.$$

In this regime, the string length is large enough that the expected *RFLCS* length approaches the alphabet size k, indicating that the natural upper bound - the alphabet size - is consistently reached.

3.3. Approaches to Solve RFLCS

Over the years, several approaches have been proposed to solve the *RFLCS* problem. These methods can broadly be categorized into heuristic approaches, which aim to find near-optimal solutions efficiently, and optimal solvers, which guarantee exact solutions but may be computationally expensive for large instances. In this section, we outline the most notable methods and provide a comparative analysis of their strengths and weaknesses. In this section, we highlight a selection of representative methods.

3.3.1. Preliminaries

A key concept in common subsequence problems is the notion of *matches* v, which correspond to pairs of indices from strings A and B, where the characters at these positions are identical.

Definition 1. *The set of all matches is defined as:*

$$V(A,B) = \{(i,j) \mid A[i] = B[j]\},$$

When the relevant strings are clear from context, we will omit explicit references to them and denote the structure simply as V. Additionally, we define the character of a match $char(v)$ as follows:

$$char(v) = char((i,j)) = A[i] = B[j],$$

which returns the character at the indices i and j in strings A and B, respectively.

3.3.2. The Integer Linear Programming Model

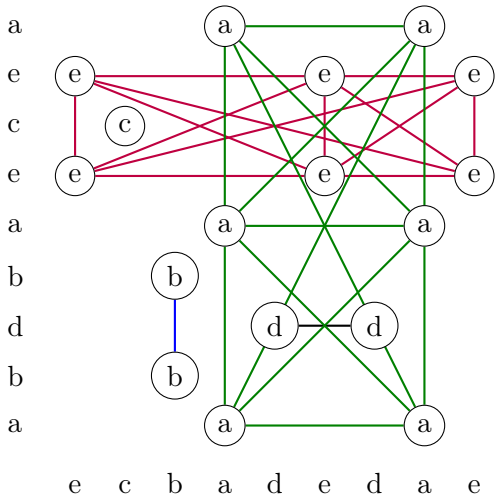
Adi et al. [ABF+08] introduced an Integer Linear Programming (ILP) formulation to optimally solve the *RFLCS* problem. The *RFLCS* instance is represented by a set of matches (see section 3.3.1 for a detailed explanation). Each match v is then associated with a binary decision variable z_v, indicating whether the match is included in the solution of the ILP.

The ILP formulation is structured as follows:

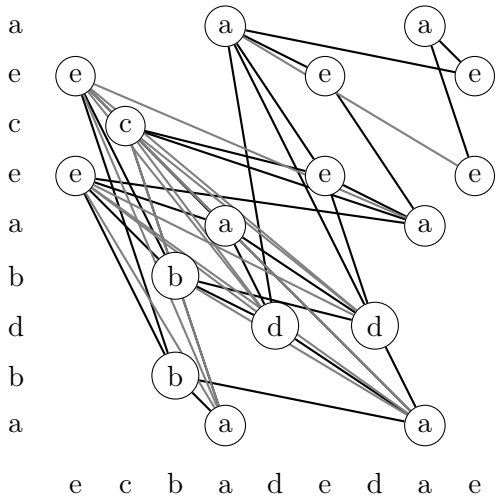
$$\begin{aligned}
&\textbf{Objective:} && \text{maximize} \sum_{v\in V} z_v && \text{(Maximize the subsequence length)}\\
&\textbf{Subject to:} && \sum_{\substack{v\in V\\ char(v)=\sigma}} z_v \le 1 \quad \forall \sigma \in \Sigma && \text{(Repetition-free constraint)}\\
& && z_v + z_{v'} \le 1 \quad \forall v, v' \in V \text{ s.t. } i < i' \wedge j > j' && \text{(Subsequence constraint)}\\
& && z_v \in \{0,1\} \quad \forall v \in V && \text{(Binary decision variable)}
\end{aligned}$$

This ILP formulation effectively captures the problem's constraints, ensuring both repetition-free and valid subsequence properties. It can also be interpreted as a Maximum Independent Set (MIS) model [Ber57], where conflicts between matches are represented as edges in a conflict graph,

as illustrated in fig. 3.1. A MIS is a subset of vertices in a graph such that no two vertices in the subset are adjacent. The conflict graph $G = (V, E)$ is defined where V is the set of matches, and E represents edges between vertices (matches) that cannot both be included in the solution of the ILP, due to conflicts between them.



(a) Repetition-Free Constraint Conflict Edges forming Cliques of Identical Character Matches



(b) Common subsequence constraint conflict edges

Figure 3.1.: Visualization of the ILP MIS interpretation for $A =$ `ecbadedae` and $B =$ `abdbaecea`. The MIS graph includes all edges from both graphs 3.1a and 3.1b, with matches represented by their characters rather than pairs of indices.

While this approach guarantees an optimal solution, it may face scalability challenges due to the large number of potential matches in M and the associated constraints.

3.3.3. Ant Colony Optimization and Beam Search

Blum et al. [BBC13] introduced a hybrid method that combines Ant Colony Optimization (ACO) with Beam Search to address the *RFLCS* problem. In this approach, the ACO component employs a pheromone model that tracks pheromone levels for each character in the sequences x and y. These pheromone values guide the search process by influencing the selection of matches based on their accumulated pheromone intensities.

Each iteration begins with a Beam Search, which uses the pheromone information to prioritize and explore promising solution paths. After the completion of a Beam Search phase, the pheromone levels are updated according to the quality of the solutions found, reinforcing paths that contribute to higher-quality solutions. If the algorithm detects stagnation in solution improvement, the pheromone values are reset to introduce variability and explore new solution spaces. This iterative cycle of Beam Search, pheromone updating, and potential resetting continues until a predefined time limit is reached.

3.3.4. A Genetic Algorithm

Castelli et al. [CBV13] proposed a genetic algorithm to solve the *RFLCS* problem, where candidate solutions are represented as m-bit arrays. In this encoding scheme, the set bits identify the positions in string A that are building a potential solution subsequence.

The genetic algorithm evolves generations by evaluating the quality of solutions through a fitness function and promoting diversity through probabilistic mutation operators. To further

refine the solution and enforce the repetition-free constraint, an Estimation of Distribution Algorithm (EDA) [HP11] is then applied, enhancing the search process.

3.3.5. Application of the Construct, Merge, Solve, and Adapt Algorithm

Blum et al. [BB16] applied the Construct, Merge, Solve, and Adapt (CMSA) framework to tackle the *RFLCS* problem. In each iteration, k candidate solutions are probabilistically generated (Construct). The matches from these solutions are then added (Merge) to a set of matches Z, which serves as the input for an ILP model described in section 3.3.2. This ILP model is then solved to optimality (Solve). To maintain the efficiency of the process and prevent the IP model from becoming overly large, unpromising matches are removed from the set Z (Adapt) based on an age property, ensuring that the model remains manageable and solvable.

3.3.6. A Branch-and-Cut Approach

Ferreira et al. [FT10] proposed a Branch-and-Cut approach to solve the *RFLCS* problem optimally. In this method, an Integer Linear Programming (ILP) model is designed to maximize a set of binary decision variables, each representing a match between the input sequences. The key aspect of this ILP formulation is the use of *extended star inequalities* as constraints. These extended star inequalities represent sets of matches, allowing at most one match from each set to be included in the final solution. The extended star inequalities not only impose valid constraints but also possess properties that help steer the branching process effectively within the Branch-and-Cut framework. In addition, the authors employ heuristics applied to the current ILP relaxation to obtain high-quality solutions early in the search process.

3.3.7. Reduction to Maximum Clique

Blum et al. [BDS+21] propose a reduction to the Maximum Clique Problem. This reduction is straightforward, as we have already discussed the MIS interpretation of *RFLCS* fig. 3.1, and since the maximum independent set problem is equivalent to the maximum clique problem in the complementary graph, the reduction is trivial; however, it enables the application of different solvers to the problem.

3.3.8. A Multivalued Decision Diagram (MDD) Representation

Horn et al. [HDBR20] propose representing an *RFLCS* instance as an *MDD*, which is a compact, directed acyclic graph (DAG) used to represent and solve combinatorial optimization problems. Unlike traditional binary decision diagrams, an *MDD* allows for more than two possible transitions at each node, making it well-suited for problems involving sequences or sets. The structure consists of multiple levels, with each level corresponding to a position within a subsequence. Each node in a level represents a state of the problem, and the edges between nodes encode possible transitions between states.

Traversing the nodes of an *MDD* corresponds to identifying a common subsequence for an *RFLCS* instance. The authors leverage this data structure to construct high-quality heuristic solutions by selecting traversals of the *MDD* based on the information maintained within the nodes. By refining the nodes of the *MDD*, repetition-free conflicts can be detected, enabling subsequences and matches to be pruned from the search space. This pruning significantly reduces the complexity of the solution process by focusing only on feasible repetition-free subsequences.

To illustrate the data structure and the impact of the refinement step, refer to fig. 4.2. Note that character repetitions along traversals of a refined *MDD* are easily identifiable and are consequently pruned from the search space. Moreover, a completely refined *MDD* can be used to extract an optimal solution of the *RFLCS* instance. However, refining a single node with respect to a character may cause it to split into two separate nodes. Since such splitting can occur for many

nodes, and may be repeated for each character in the alphabet, the size of the *MDD* can grow exponentially with the size of the alphabet. Therefore, the growth of the *MDD* must be carefully controlled, and complete refinement cannot always be guaranteed.

Following an incomplete *MDD* refinement process, the remaining matches serve as input for the Integer Linear Programming model discussed in section 3.3.2.

4. Proposed Algorithm

We propose a randomized heuristic (section 4.2) that builds upon a dominating match graph representation (section 4.1), where each vertex encodes a character match between the two input strings, and edges preserve their relative order. It generates lower bounds by combining solutions from both the original and reversed inputs. To reduce the search space, we efficiently compute upper bounds by relaxing the repetition-free constraint for most characters in the alphabet Σ, while enforcing it only for a selected subset (section 4.3). Remarkably, even enforcing this constraint on a small number of characters can significantly tighten the upper bounds. Based on these bounds, we prune dominating match graph vertices whose best-case contribution cannot exceed the current lower bound. A directional heuristic (section 4.4.4), informed by metrics derived from the *MDD*, creates two separate *MDD*s: one for the original input and one for the reversed input strings. The heuristic then selects the direction corresponding to the *MDD* with fewer matches, as this is expected to lead to more efficient *MDD* behavior. The *MDD* compactly represents all common subsequences, which may not yet be repetition-free, and is iteratively refined through a character-by-character refinement and filtering process to enforce uniqueness (section 4.4). During refinement, *MDD* nodes are split based on the presence or absence of a specific character. This results in finer knowledge of possible subsequences when a node is present in a potential solution. The subsequent filtering phase removes infeasible paths that violate the repetition-free constraint, effectively reducing the diagram's complexity. The refinement order is guided by an impact measure that quantifies each character's potential to prune the search space, as well as its effect on the *MDD*'s size (section 4.4.6). While refinement can cause the *MDD* to grow due to node splitting, the filtering process can reduce its size. The growth scale measures the overall impact of a character on the *MDD*'s size after both refinement and filtering, ensuring an optimal pruning-to-*MDD*-size ratio. If the *MDD* exceeds memory limits during the procedure, we compress it (section 4.5) and restart the *MDD* process. If optimality cannot be established within the timeout, we switch to an ILP formulation (section 4.6) on the reduced instance, using the matches from the remaining *MDD* nodes as input.

4.1. Dominating match graph (DMG) Representation

In this section, we introduce a dominating match graph, where the vertices correspond to character matches between the two input strings. Directed edges between vertices indicate possible sequence continuations when forming a common subsequence (CS).

Definition 2. *The dominating relation between two matches $v = (i, j)$ and $v' = (i', j')$ is defined as:*

$$v \prec v' \iff v \neq v' \wedge i \leq i' \wedge j \leq j'$$

Definition 3. *The set of appendable matches after a given match v is defined as:*

$$R_v = \{v' \in V \mid v \prec v'\}$$

This set includes all matches that could occur in a CS after the given match.

Match $v = (i, j)$ can be viewed as virtually creating a subinstance of the original problem, where only the substrings of A and B starting at positions $i + 1$ and $j + 1$, respectively, are considered.

Algorithm 1: Algorithm overview of *DMG* creation stage, heuristic stage, repetition-free subset *LCS* stage, *MDD* stage and ILP stage

Input : Strings A and B
Output : Optimal solution C^*, or suboptimal solution $\hat{C}$ and an upper bound ub
Create dominating match graph *DMG* `/* section 4.1 */`
Run randomized heuristic, updating c_{best} and lb `/* section 4.2 */`
Prune dominating match graph via Repetition-Free Subset LCS relaxation, updating ub `/* section 4.3 */`
Create and filter initial *MDD*s for forward and backward instance `/* section 4.4.2 */`
Decide which *MDD* to use `/* section 4.4.4 */`
while *MDD not completely refined and timeout not reached* **do**
  MDD refinement and filtering procedure `/* algorithm 2 */`
end
Prune *DMG* according to *MDD*
Reduce remaining *DMG* to ILP and solve it, updating $\hat{C}$, lb and ub `/* section 4.6 and appendix A */`
return $\hat{C}$

Definition 4. *The set of dominating appendable matches for a given match v is defined as:*

$$D_v = \{v' \in R_v \mid \nexists v'' \in R_v \text{ such that } v'' \prec v'\}$$

This set includes all matches that are directly dominated by the given match, with no intermediate match dominating them.

Definition 5. *The dominating match graph is defined as* $DMG = (V, \{(v, v_d) | v_d \in V \wedge v_d \in D_v\})$.

Figure 4.1 illustrates four dominating match graphs. The first graph depicts the *DMG* for the original strings A and B. The second graph represents the reverse *DMG*, where the input strings are reversed. The last two graphs show versions where certain matches have been removed because they cannot contribute to any subsequence that would improve an intermediate solution. The graphs show the identified matches and their dominating successors. Matches in the regular dominating match graph are denoted by v, whereas corresponding matches in the reverse graph are denoted with a left arrow above the symbol, i.e., $\overleftarrow{v}$. Please observe, that $\overleftarrow{\overleftarrow{v}} = v$.

Based on the *DMG* and its corresponding reverse *DMG*, we can derive upper bounds. To do so, we combine two complementary upper bounds: on one hand, the familiar formulation for the common LCS [BDS+21], and on the other hand, the number of distinct characters that can appear on paths passing through a match.

Definition 6. *We define an RFLCS sub-instance with respect to a match $v = (i, j)$ as the RFLCS problem restricted to the suffixes of A and B starting immediately after the indices of v. Formally, this is given by*

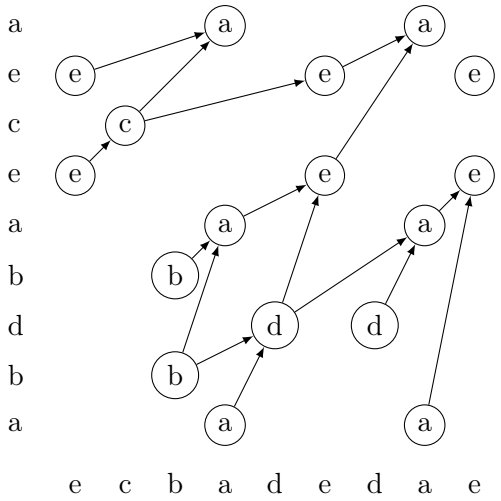
$$RFLCS(A[i+1\colon m], B[j+1\colon n]),$$

where the bracket notation denotes substrings.

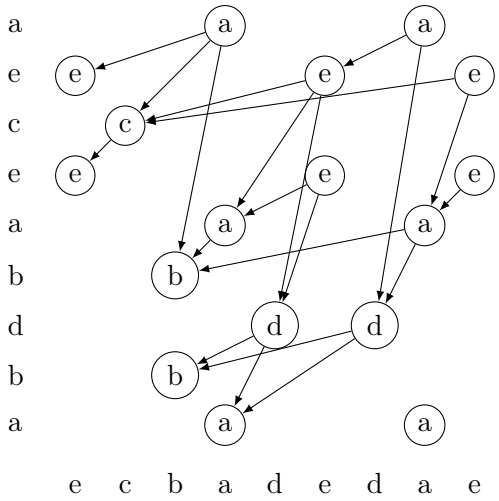
Definition 7. *For each match v, let us_v denote the upper bound for the RFLCS sub-instance w.r.t. match v. These upper bounds are computed using the following formulation:*

$$us_v = \min\left(\underbrace{\max(1 + \{us_{v'} | v' \in D_v\})}_{\text{LCS-analogue}}, \underbrace{\left|\{\sigma' | char(v') = \sigma' \wedge v' \in R_v\}\right|}_{\text{appendable characters}}\right)$$

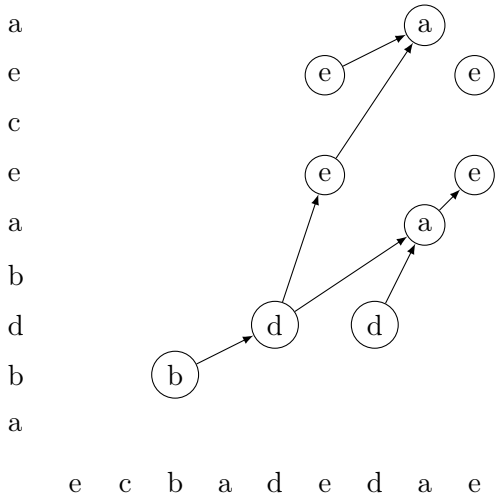
With the convention that $\max(\emptyset) = 0$.



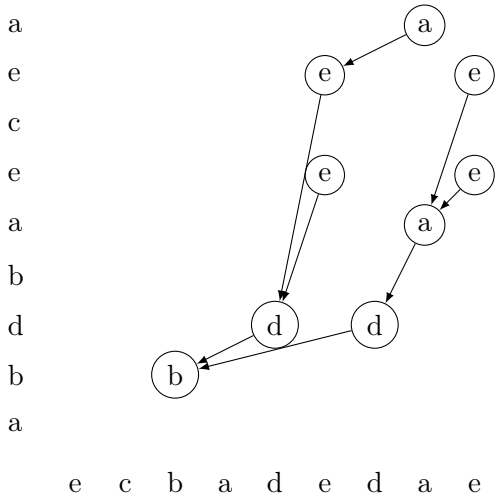
(a) Initial *DMG*



(b) Initial reverse *DMG*



(c) Reduced *DMG*



(d) Reduced Reverse *DMG*

Figure 4.1.: Various dominating match graphs for $A =$ `ecbadedae` and $B =$ `abdbaecea`. The strings A and B are shown along the bottom and left side respectively, with matches indicated by their characters.

LCS is less constrained than *RFLCS* and, as a result, always serves as an upper bound for *RFLCS* subinstances. Additionally, *RFLCS* is restricted by the distinct characters contained in the subinstance.

Definition 8. *For each match v, let ub_v denote the RFLCS upper bound for a match v. These upper bounds are computed using the following formulation:*

$$ub_v = \min\left(\underbrace{1 + us_v + us_{\overleftarrow{v}}}_{\textit{combined upper bounds}}, \underbrace{\left|\{char(v)\} \cup \{\sigma' | char(v') = \sigma' \wedge \sigma' \in R_v \cup R_{\overleftarrow{v}}\}\right|}_{\textit{combined characters}}\right)$$

These upper bounds are used to identify matches that cannot be part of any feasible solution capable of improving the current lower bound. Specifically, the upper bounds give an upper limit on the *RFLCS* length that can be achieved when a match is included in the solution. If we identify a match whose upper bound is less than or equal to the current lower bound lb, we can

safely remove that match from the graph. This is because no solution containing this match can improve upon the best solution found so far.

4.2. A Randomized Heuristic

Building upon the dominating match graph representation, we propose a randomized heuristic that leverages the structure of the *DMG* to find high-quality solutions. The heuristic operates by maintaining two repetition-free subsequences for every match in the graph: one that can occur before the match and another that can occur after the match. These subsequences are selected in an alternating fashion. Greedily for each match we pick a subsequence from the subproblems of the match, that best complement the sequence maintained in the opposing direction. This aims to maximize the unique characters in the combined subsequences. While this approach aims to choose the longest possible subsequences, it does not guarantee optimality.

Definition 9. *For each match v, let h_v denote the longest directly obtainable repetition-free subsequence that occurs after match v. These subsequences are computed using the following formulation:*

$$h_v = char(v) \cdot argmax_{\{h_{v'} | v' \in D_v\}} \left| h_{v'} \cup h_{\overleftarrow{v}} \right|$$

In this formulation, $\cdot$ represents string concatenation, indicating that the solution h_v is obtained by concatenating the subsequences associated with match v and its predecessors. The $\cup$ symbol denotes the union of all characters contained in the sequences.

To resolve ties between subsequences of equal length, the heuristic introduces randomness, breaking ties randomly. This randomness allows the heuristic to explore different feasible solutions, potentially avoiding local optima. For each match, sequences that follow the current match are generated by evaluating h_v in the regular *DMG*. Conversely, sequences that precede the match are found evaluating $h_{\overleftarrow{v}}$ in the reverse *DMG*. In each pass, a match v is only traversed when all matches $v' \in D_v$ have already been traversed. Since the *DMG* and the reverse *DMG* are acyclic, such a traversal is always possible. The traversal of the *DMG* and the reverse *DMG* alternates, because changes in h_v may lead to changes in $h_{\overleftarrow{v}}$, and vice versa. By alternating the traversal direction, we ensure that all updates to h_v are fully propagated before triggering potential changes in $h_{\overleftarrow{v}}$, potentially leading to an efficient convergence process.

When a new incumbent solution is found, e.g., if $\left|\mathrm{h}_{\overleftarrow{v}} \cup \mathrm{h}_v\right| > lb$, the solution is derived by reconstructing a regular subsequence from h_v and $\mathrm{h}_{\overleftarrow{v}}$, followed by the removal of potential duplicate characters. Note that the removal of repeated characters does not decrease $\left|\mathrm{h}_{\overleftarrow{v}} \cup \mathrm{h}_v\right|$. Furthermore, the lower bound is updated to $lb \leftarrow \left|\mathrm{h}_{\overleftarrow{v}} \cup \mathrm{h}_v\right|$. After updating lb, we prune every match from both *DMG*s that satisfies $ub_v \leq lb$, as these matches cannot contribute to a new incumbent solution. If no further improvement is observed after a set number of traversals, the subsequences associated with each match are reset to allow for fresh exploration of potential solutions. This process continues iteratively until no further improvement is observed after a specified number of resets, signaling that the heuristic has likely converged on a (near-)optimal solution.

We can derive an upper bound on the worst-case running time for our heuristic. To do so, we first establish that the worst-case number of matches in the graph is in $\mathcal{O}(m \cdot n)$. Furthermore, we assume that each match consists of $\Sigma - 1$ edges. Since a fixed series of set operations is performed for each edge, involving up to Σ characters, we assume that these operations can be executed in $\mathcal{O}(\Sigma)$ time, which can be achieved using for instance a bitset. Consequently, the asymptotic

runtime for a single pass of the heuristic is given by

$$t_{\text{hpass}} = \mathcal{O}(m \cdot n \cdot |\Sigma|^2).$$

Additionally, we continue executing passes until stagnation is detected. Let us denote this repetition limit by $rep(A, B)$. Since the stagnation counter is reset whenever an incumbent solution is found, this reset can occur up to $l(A, B)$ times in the worst case. Thus, the overall heuristic runtime is given by

$$t_{\text{heur}} = \mathcal{O}(m \cdot n \cdot |\Sigma|^2 \cdot rep(A, B) \cdot l(A, B)).$$

4.3. A Repetition-Free Subset LCS Relaxation

We propose a relaxed variant of *RFLCS*, where the goal is to find an LCS with the additional constraint that characters from a given set F may appear at most once in the solution. By calculating upper bounds for this problem, we can also derive upper bounds for the *RFLCS*. These upper bounds are then used to prune the *DMG*, removing matches that cannot contribute to a solution longer than the current lower bound lb. This relaxation technique is particularly valuable. By providing a quick pass to reduce the size of the dominating match graph, it speeds up the subsequent decision diagram process described in section 3.3.8. Since one pass of this relaxation is computationally efficient, we iteratively apply it to different sets of characters F until no further pruning of the dominating match graph occurs. We select $\log_2(|\Sigma|)$ to be in F and the selection of characters is guided by the following criteria:

- Maximum number of occurrences of the character in any common subsequence (CS).
- Number of matches for this character in the dominating match graph.
- Sum of all upper bounds of matches in the dominating match graph.

For a chosen set of characters $F \subset \Sigma$, and for each match v, we define a mapping for each element ρ of the powerset of F, where $\rho \in \mathcal{P}(F)$, to an upper bound $f(v, F, \rho)$ of the LCS subinstance of v. This subinstance considers only characters from $(\Sigma \setminus F) \cup \rho$ while ensuring that characters from F appear repetition-free.

For a given match v, the maximum of these upper bounds $f(v, \cdot, \cdot)$ is itself an upper bound for the *RFLCS* of the subinstance associated with v. We can derive these upper bounds using the *DMG* and the following formulation:

$$f(v, F, \rho) = \begin{cases} 0, & \text{if } D_v = \emptyset \\ 1 + \max(\{f(v', F, \rho) \mid v' \in D_v\}), & \text{if } char(v) \notin F \\ \max\left(\{f(v', F, \rho) \mid v' \in D_v\} \cup \{1 + f(v', F, \rho \setminus \{char(v)\}) \mid v' \in D_v\}\right) & \text{otherwise} \end{cases}$$

We then apply those new bounds to the bounds in the *DMG*:

$$us_v \leftarrow \min\left(us_v, \max(\{f(v, F, \rho) \mid \rho \in \mathcal{P}(F)\})\right).$$

Following the updates of the upper bounds us_v in the *DMG* and the reverse *DMG*, we can also update the upper bounds ub_v as defined in definition 8, allowing for further pruning of matches from the *DMG*.

4.4. Multivalued Decision Diagram

The Multivalued Decision Diagram (*MDD*) is designed to represent all relevant common subsequences (CS), which may initially contain character repetitions. These repetitions are identified

and removed as the diagram undergoes a process of refinement and filtering. This chapter provides a formal definition of the *MDD* and a detailed explanation of its initialization. We then describe the refinement and filtering phases applied to the diagram. The algorithm 2 provides an overview of the key phases involved in constructing, refining, and filtering the *MDD*. Each phase is explained in greater detail in the subsequent sections.

Algorithm 2: Overview of Multivalued Decision Diagram Refinement, Updating, and Pruning Phases

Input: Current initial *MDD*, lower bound *lb*
Output: Repetition-free longest common subsequence
Phase 2: Find Refinement Order for Characters (4.4.6);
Determine the order of characters for refining the current decision diagram;
while *The diagram is not refined by all characters*
and *memory is not exhausted*
and *the lower bound lb is less than the upper bound ub* **do**
  Phase 3: Refinement Phase (4.4.5);
  Refine the current *MDD* by applying the next character from the character refinement sequence (4.4.6);
  Phase 4: Filter Phase (4.4.3);
  while *Update and Pruning did not converge* **do**
    Phase 4.1: Update Phase (4.4.3);
    Derive the current attributes of a node based on the current decision diagram;
    Phase 4.2: Prune Phase (4.4.3);
    Prune the current decision diagram by removing conflicting or unnecessary edges and nodes;
  end
end
Phase 5: *MDD* **compression** (4.5);

4.4.1. Multivalued Decision Diagram Definitions

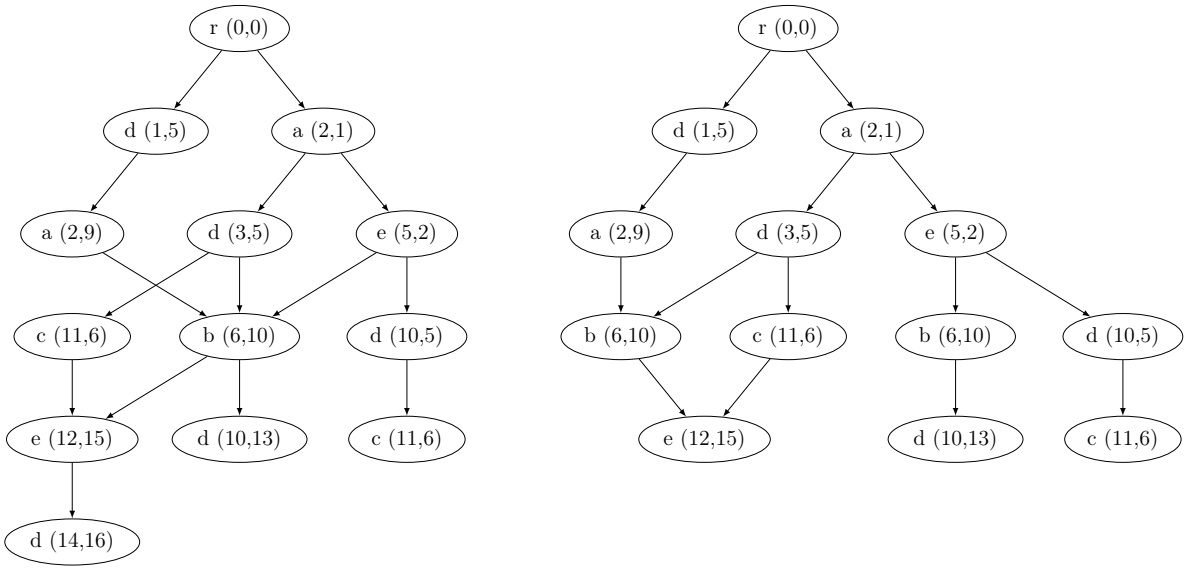
In the multivalued decision diagram (*MDD*), each node represents a match from the *DMG*, augmented with additional state information. The *MDD* includes an artificial root node at level 0, serving as the starting point of each CS encoded by the *MDD*. Subsequent levels correspond to positions within a solution string, where each level contains nodes representing matches at the respective position in a potential solution string.

Each node η in the decision diagram is a tuple

$$\eta = (v, pred, succ, pre, prefix, app, postfix)$$

representing its attributes:

- **Match *v*:** The specific match represented by node η.
- **Predecessor Nodes *pred*:** Connections from nodes at the level immediately above. These edges represent transitions from matches that would precede the match of node η in a solution string.
- **Successor Nodes *succ*:** Connections to nodes at the level immediately below. These edges represent potential transitions from the current node η to matches that would follow it in the solution string.
- **Prependable Characters *pre*:** Characters appearing on at least one path leading from root node r to node η, including $char(v)$ itself.



(a) Initial Multivalued Decision Diagram (b) Refined Multivalued Decision Diagram

Figure 4.2.: Multivalued Decision Diagrams for the input strings $A =$ `daebabadceadca` and $B =$ `aedcdcababdbed`, where nodes are represented by their characters followed by their matches. The diagrams illustrate the pruning of the match $(14, 16)$ from the *MDD* and the decline of the deduced upper bound from 5 to 4.

- **Prefix Characters *prefix*:** Characters that occur on all paths from node η to the root node r, including $char(v)$. These characters are guaranteed to be part of the prefix of every solution string containing this node η.
- **Appendable Characters *app*:** Characters appearing on at least one path from a node $\eta' \in succ$ to nodes on deeper levels of the *MDD*, but always excluding $char(v)$.
- **Postfix Characters *postfix*:** The set of characters that appear along all paths from all nodes $\eta' \in succ$ to nodes at depth $lb + 1$. These characters are guaranteed to be part of the postfix of any solution string that includes node η. Note that characters from levels beyond $lb + 1$ are not necessarily included in an *RFLCS*, as the actual length l may be precisely $lb + 1$.

The *MDD* is a partitioned graph, where each partition is called a level, denoted by MDD_i. Each level MDD_i consists of a list of nodes. As an additional convention, we denote the total number of levels by $|MDD|$. Furthermore, we use $\eta \in MDD$ to indicate that the node η appears in at least one level MDD_i. Similarly, we write $\eta \in MDD_i$ to specify that the node η occurs in level i of the *MDD*. To improve clarity, we use the redundant adjacency lists *pred* and *succ*. By redundant, we mean that if and only if $\eta' \in succ$, then $\eta \in pred'$ holds. Moreover, a successor node can only reside one level deeper than its predecessor. More formally, if $\eta' \in succ$ and $\eta \in MDD_i$ then $\eta' \in MDD_{i+1}$.

The *MDD* graph is given by $G_{MDD} = (N, \{(n, n') \mid n' \in succ\})$, where N is the set of all nodes. A visual representation is provided in fig. 4.2.

4.4.2. Initialising a Multivalued Decision Diagram

The initial *MDD* is a compact representation of all relevant common subsequences, that still allows for possible repetitions of characters. The construction process begins by introducing an artificial root node r at level 0. The *MDD* is then expanded level by level, where each node at a given level may generate a successor node at the next level. Nodes representing the same match are merged to maintain the compactness of the *MDD*. This expansion continues up to level $ub - 1$, since we already know that no *RFLCS* can exceed length ub, ensuring that all relevant

sub sequences are included. If on every *MDD* level there are no two nodes representing the same match, then we call it a compressed *MDD* and an initial *MDD* is always compressed.

During initialization, character sets for all nodes are assigned as follows:

- The sets *pre* and *app* are initialized to the alphabet Σ.
- The sets *prefix* and *postfix* are initialized to the empty set $\emptyset$.

The first filter pass from section 4.4.3 restores the invariants described in section 4.4.1.

To systematically construct the decision diagram, we require a set that, given a match, provides possible successor matches.

Definition 10. *The set of successor matches S_v w.r.t. a current DMG $= (M, \cdot)$ and for a given match v, is defined as:*

$$S_v = \left\{ v' \in R_v \mid v' \in M \wedge \nexists v'' \in R_v \text{ s.t. } v' \neq v'' \wedge char(v') = char(v'') \wedge v'' \prec v' \right\}.$$

This set S_v contains matches corresponding to appendable characters according to a current DMG, ensuring that the matches have the smallest possible indices while still being dominated by v.

Using this definition, we formalize the construction of the initial *MDD*:

$$r \leftarrow (0, 0), \quad char(r) = \mathbf{0}.$$

Here, we consider the character of r as non-existent, meaning it is not a character in Σ and is ignored in set operations.

The initialization of the *MDD* is given by:

$$MDD_0 \leftarrow \langle (r, d_r, \emptyset, \emptyset, \emptyset, \Sigma, \emptyset) \rangle.$$

For each subsequent level $i + 1$, the *MDD* is expanded as follows:

$$MDD_{i+1} \leftarrow \langle (v', \text{in}(MDD_i, v'), \emptyset, \Sigma, \emptyset, \Sigma, \emptyset) \mid \text{in}(MDD_i, v') \neq \emptyset \rangle,$$

where $\langle \cdot \rangle$ denote lists and the function $\text{in}(MDD_i, v')$ is defined as:

$$\text{in}(MDD_i, v') = \{ (v, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \in MDD_i \mid v' \in S_v \}.$$

After initialization, a filtering phase is applied to eliminate redundant nodes and edges, as described in section 4.4.3.

4.4.3. Multivalued Decision Diagram Filter

After initialisation and each refinement (section 4.4.5), the *MDD* undergoes a filtering process comprising two sub phases: updating and pruning. These sub phases are applied iteratively until no further changes occur in the diagram.

The *update* sub phase involves deriving the current attributes of the nodes based on the current paths of the *MDD*, ensuring that all node information accurately reflects the latest refinement.

Following the update, the *pruning* sub phase removes edges and nodes from the decision diagram, where a conflict occurs. Both updating and pruning are repeated as long as modifications continue to occur, ensuring the diagram is pruned as thoroughly as possible before proceeding to subsequent phases.

Multivalued Decision Diagram Node Updates

During the update process, each node's character sets are adjusted accordingly. Although it is not entirely accurate, but to build some intuition, the sets *pre* and *prefix* are computed by uniting

respectively intersecting all pre' and $prefix'$ for all $\eta' \in pred$. And the sets app and $postfix$ are computed by uniting respectively intersecting all app' and $postfix'$ for all $\eta' \in succ$. Please note that the sets pre and app can not grow in size, because a character that could neither be pre- or appended cannot appear on a path before or after a node. Conversely the sets $prefix$ and $postfix$ can only grow in size, because a character that was present on all paths, must remain so. Formally the update phase is defined by following formulation:

$$\begin{aligned}\eta = (v, pred, succ, pre, prefix, app, postfix) \leftarrow \\ (v, pred, succ, \\ pre \cap \{\sigma' | n' \in pred \wedge \sigma' \in pre'\} \cup \{char(v)\}, \\ prefix \cup \{\sigma' | \forall n' \in pred : \sigma' \in prefix'\}, \\ app \cap \{\sigma' | n' \in succ \wedge \sigma' \in (\{char(v')\} \cup app')\}, \\ postfix \cup \{\sigma' | \forall n' \in succ : n' \in MDD_i \wedge i \leq lb + 1 \wedge \sigma' \in (\{char(v')\} \cup postfix'\})).\end{aligned}$$

The update phase always terminates because the character sets can only either expand or contract, while the number of nodes remains constant throughout the process. In fact, a single pass from the root to the sinks, followed by a pass from the sinks back to the root, is already sufficient for completion. Please note, that by our approach to compute $prefix$ a character that is present on all paths from r to η, is not guaranteed to be in $prefix$, analogous the same is true for $postfix$.

Multivalued Decision Diagram Node and Edge Pruning

Following the update phase, the prune phase is initiated. The goal of this phase is to eliminate conflicting or redundant nodes and edges, ensuring that only the most relevant paths are retained.

A node is removed if the size of the union of its prepending and appending character sets is less than or equal to the lower bound lb. Specifically, if $|pre \cup app| \leq lb$, the node is deemed to contribute to paths that cannot form repetition-free subsequences longer than lb, and thus it is eliminated from the diagram.

Additionally, a node is pruned if the intersection of its $prefix$ and its $postfix$ is non-empty, i.e., $prefix \cap postfix \neq \emptyset$. This indicates that the node would lead to a subsequence that is not repetition-free within the first $lb + 1$ positions, making it unsuitable for an *RFLCS*.

Nodes without predecessors ($pred = \emptyset$) that are not the root node r are also removed, as they do not contribute to any valid paths in the *MDD*.

Furthermore, nodes $n \in MDD_i$ that satisfy the condition $i + |app| \leq lb$ are pruned. This condition indicates that there are insufficient characters following the node to form a repetition-free common subsequence longer than the current lower bound lb.

Edges are also subject to pruning based on the properties of their connected nodes. An edge between nodes η and η' is removed if the combined character sets of the prepending characters of node η, the match character of node η', and the appending characters of node η' do not provide enough characters to form a subsequence longer than lb. Specifically, if $|pre \cup \{char(v')\} \cup app'| \leq lb$, the edge is pruned, as this indicates that placing the two nodes consecutively in a solution cannot yield a repetition-free subsequence long enough to exceed the current lower bound.

Edges are pruned when there is a conflict with the repetition-free constraint between two connected nodes. Specifically, the edge between nodes η and η' is removed if the intersection of the prefix character set of node η and the combined set of the match character of node η' and its postfix characters is non-empty. Formally, if $prefix \cap \left(\{char(v')\} \cup postfix'\right) \neq \emptyset$, the edge is pruned. This indicates that each long enough common subsequence including these two nodes, would violate the repetition-free constraint.

An edge from node $\eta \in MDD_i$ is also pruned if the target node η' is dominated by at least $i - |prefix| + 1$ other target nodes, or if it is dominated by another target node η'' under the condition that $char(v'') \notin pre$. A proof that we do not lose all optimal solutions by this rule, is

given in appendix C. Although removing such an edge could potentially exclude a solution better than the current lower bound lb, it is guaranteed that the remaining edges still contribute to an equally good or even better solution.

An *MDD* level is pruned if it contains no nodes. This step is crucial since we use $|MDD| - 1$ as an additional upper bound for *RFLCS*.

These pruning conditions are applied iteratively within a loop, continuing until no further reductions in nodes or edges are detected. This ensures that the *MDD* is as tight as possible, retaining only relevant paths.

4.4.4. Deciding Solving direction

Finding an *RFLCS* or reversing the *RFLCS* of reversed input strings leads to the same solution. This insight motivates selecting the direction that simplifies solving an instance within our *MDD* approach.

We propose a heuristic that predicts whether solving the original problem or its reversed counterpart will result in better runtime performance. To achieve this, we compute two complexity metrics $\kappa(MDD)$ based on the initial *MDD*s (section 4.4.2) and choose the direction with the smaller complexity metric. The complexity metric is computed by counting the number of distinct matches across all levels of the *MDD*. Formally, we define the complexity as: $\kappa(MDD) = |\{v \mid (v, \cdot, \cdot, \cdot, \cdot, \cdot) \in MDD\}|$.

4.4.5. Refinement Phase

In this phase, for each path in the *MDD*, we enforce the occurrence of the refinement character $\sigma \in \Sigma$. As a result, each node has σ either in *pre* or in *app*, but not in both, hence, we removed all repetitions of character σ in all paths. We do this by refining each node w.r.t. σ. This refinement targets nodes where σ appears in both *pre* and *app*. When refining a node η by σ, η is split into two nodes, η' and η'' and η is removed from the *MDD*. The node η' includes σ in its $prefix'$, while removing σ from app', representing paths where σ is definitively included on paths to the root, but not on paths to the sinks. The other node, η'', excludes σ from pre'', indicating the absence of σ on paths leading to the root.

Furthermore, the predecessor edges are rerouted. For node η', only those predecessor edges are retained where the predecessor node definitively has σ on all paths from the root to it. Similarly, for node η'', only those predecessor edges are kept where the predecessor definitively does not have σ on paths from the root to itself. The successor edges remain unchanged from the original node.

More formally, the refinement w.r.t. σ of the MDD_i is defined as:

$$\begin{aligned} MDD_i \leftarrow & \langle \eta \in MDD_i \mid \sigma \notin (pre \cap app) \rangle \\ & \oplus \langle \text{incl}(\eta, \sigma) \mid \eta \in MDD_i \wedge \sigma \in (pre \cap app) \rangle \\ & \oplus \langle \text{excl}(\eta, \sigma) \mid \eta \in MDD_i \wedge \sigma \in (pre \cap app) \rangle \end{aligned}$$

where $\oplus$ denotes list concatenation and $\text{incl}(\eta, \sigma)$ and $\text{excl}(\eta, \sigma)$ respectively are defined as:

$$\text{incl}(\eta, \sigma) = (v, \{\hat{\eta} \in pred \mid \sigma \in \hat{prefix}\}, succ, pre, prefix \cup \{\sigma\}, app \setminus \{\sigma\}, postfix)$$

$$\text{excl}(\eta, \sigma) = (v, \{\hat{\eta} \in pred \mid \sigma \notin \hat{prefix}\}, succ, pre \setminus \{\sigma\}, prefix, app, postfix).$$

The following filter pass from section 4.4.3 restores the invariants described in section 4.4.1.

4.4.6. Character Refinement Sequence

The objective is to derive a character sequence that optimally leverages the *MDD* refinement (section 4.4.5). We denote a *MDD* that is refined by character σ and subsequently filtered by MDD^σ. A greedy scoring approach evaluates differences between an *MDD* and its MDD^σ for all $\sigma \in \Sigma$. The score for each character is denoted as $\varphi_\sigma(MDD, MDD^\sigma)$. This score quantifies the impact of the refinement on both the size of the *MDD* and its ability to prune matches. φ_σ is a composite metric, incorporating multiple evaluation criteria.

Definition 11. *The number of distinct match edges in the MDD is defined as*

$$\varepsilon(MDD) = \left| \left\{ (v, v') \;\middle|\; \begin{array}{l} (v, \cdot, next, \cdot, \cdot, \cdot, \cdot) \in MDD_i, \\ (v', \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \in MDD_{i+1}, \\ v' \in next \end{array} \right\} \right|.$$

Definition 12. *The count of nodes in the MDD is defined as*

$$\mathcal{K}(MDD) = \sum_{i=0}^{|MDD|} |MDD_i|\,.$$

Definition 13. *The count of edges in the MDD levels is defined as*

$$\mathcal{E}(MDD) = \sum_{i=0}^{|MDD|} \sum_{n \in MDD_i} |next|\,.$$

Definition 14. *The refinement potential score of a character σ is defined as*

$$\varphi_\sigma(MDD, MDD^\sigma) = \frac{\overbrace{(\kappa(MDD) - \kappa(MDD^\sigma)) \cdot (\varepsilon(MDD) - \varepsilon(MDD^\sigma))}^{Pruning\ Potential}}{\underbrace{\mathcal{K}(MDD^\sigma) \cdot \mathcal{E}(MDD^\sigma)}_{MDD\,Scale}}$$

Subsequently, we rank each character σ based on its refinement potential score φ_σ, ordering them from the highest to the lowest score. This order then is used to determine the next character for refinement in the while loop of algorithm 2.

4.5. Multivalued Decision Diagram Compression

An *MDD* can be compressed by merging nodes that represent the same match, while still preserving all represented subsequences. Formally, this compression step is defined as:

$$MDD_i \leftarrow \left\langle \left(v, \bigcup_{(v, pred, \cdot, \cdot, \cdot, \cdot) \in MDD_i} pred, \bigcup_{(v, \cdot, succ, \cdot, \cdot, \cdot) \in MDD_i} succ, \cdot, \cdot, \cdot, \cdot \right) \;\middle|\; \exists \eta \in MDD_i \right\rangle$$

This is followed by an update phase to restore the character set invariants.

4.6. Integer Linear Program

We employ two different ILPs, we utilized the already discussed match based ILP model described in section 4.6, but we also formulated an *MDD* based ILP model, that we will only discuss in appendix A, because despite differences in their performance, the differences are only small in

contrast to other parts of this work. The input matches for our ILP considers only the set of matches, that still occur in the *MDD*: $\{v|(v,\cdot,\cdot,\cdot,\cdot,\cdot) \in MDD\}$.

5. Experimental Evaluation

In this section, we outline the methodology and results of our experiments. We begin by describing the experimental setup, followed by a detailed presentation of the results. The results are then categorized into quantitative and qualitative assessments, offering both numerical comparisons and deeper insights into the algorithm's performance. The implementation used in our experiments is publicly available at `https://github.com/sch0rschi/rflcs/tree/masters_thesis`.

5.1. Experimental Methodology

To evaluate and compare our approach, we used the same benchmark sets as those used by Horn et al. [HDBR20]. These consist of two randomly generated benchmark sets first introduced by Blum et al. [BB18].

The first set, referred to as Instance Set 1, comprises 1680 instances. For each combination of input string lengths $m \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$ and alphabet sizes $|\Sigma| \in \{\frac{m}{8}, \frac{m}{4}, \frac{3m}{8}, \frac{m}{2}, \frac{5m}{8}, \frac{3m}{4}, \frac{7m}{8}\}$, there are 30 instances. The second set, referred to as Instance Set 2, contains 30 instances for each combination of alphabet sizes $|\Sigma| \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and maximum repetitions of each character, $\text{reps} \in \{3, 4, 5, 6, 7, 8\}$. This set includes a total of 1440 instances. Together, both sets encompass 3120 instances. We refer to combinations of alphabet size and string length, or alphabet size and the number of repetitions, as a complexity category.

Furthermore, we extended the benchmark with two additional complexity categories following the structure of Instance Set 1. Specifically, we introduced input string lengths $m \in \{640, 768\}$ with an alphabet size of $|\Sigma| = \frac{m}{8}$. The instances in these categories were generated by selecting characters from the alphabet uniformly and independently at random to construct the input strings. This inclusion was motivated by our observation of a transition from solvable to unsolvable instances between $n = 512$ and $n = 1024$ for $|\Sigma| = \frac{m}{8}$. Additionally, we provide these instance sets alongside our implementation of the algorithm and all results of each instance to facilitate further research and ensure reproducibility.

In our work, we directly use the results reported by Horn et al. [HDBR20], as we were unable to get an implementation artifact despite our attempts to contact the authors. While reproducing their approach independently would have been ideal, a best-effort implementation was not feasible due to the lack of sufficient details in their paper.

All experiments were conducted on a machine running Ubuntu 24.04.2 LTS, using a single thread of an Intel® Xeon® Gold 6130 CPU running at 2.10 GHz, with 32 GB of RAM and 512-bit SIMD registers. We want to point out that the hardware setup of the original *MDD* approach is weaker; however, this does not account for the performance disparity observed in the conducted experiments. The implementation was developed in C++ and compiled using the Intel® oneAPI DPC++/C++ Compiler version 2025.0.4 (build 20241205), along with Ubuntu GLIBC version 2.39-0ubuntu8.4. The compiler was given flags `-O3` and `-march=native`. For solving the ILP, we used GUROBI version 12.0.0. A timeout of 1800 seconds was imposed for the combined runtime of Heuristic, Repetition-Free Subset LCS Relaxation and *MDD* refinement and filtering steps, in addition there were 1800 seconds for solving the ILP. GUROBI was configured to prioritize reducing the upper bound as quickly as possible during the solving process. The algorithm executable was configured to adhere to the specified timeouts, with a limit on heuristic graph traversals without finding an incumbent solution set to $l_t = \frac{m \cdot n}{|\Sigma|}$. Additionally, a threshold

was imposed before resetting the heuristic state, given by $l_r = \sqrt{l_t}$.

Tables 5.1 and 5.2 provide a detailed comparison of the performance metrics for three approaches: the original *MDD* approach (*MDD* + CPLEX), our heuristic, and our modified *MDD* approach (*MDD* + Gurobi). The first two columns, $|\Sigma|$ and m, define the *complexity category*, where $|\Sigma|$ represents the alphabet size, and m represents the string length. The third column, *obj best*, indicates the best-known average solution quality for the respective complexity category found in the literature. The next six columns represent the performance measures of the original *MDD* approach (*MDD* + CPLEX). The column *obj* shows the achieved average solution quality, while *gap* [%] presents the average gap between the upper and lower bounds. The column t_{prep}[s] indicates the time required to complete the *MDD* reduction steps, and t_{tot}[s] represents the total runtime of the algorithm. The column #opt provides two values: the number of instances where the *MDD* reduction was fully completed (value before the slash) and the number of instances solved optimally (value after the slash). Finally, *red*[%] shows the average reduction of matches compared to the total number of matches, with higher values being better. The subsequent four columns describe the performance measures of our heuristic. The column *obj* represents the average solution quality. The column *best* counts the number of times the heuristic either found an optimal solution, or if the optimum is unknown, it then instead counts cases where no better solution was found during subsequent optimization phases. Column t_{sol}[s] indicates the time at which the heuristic solution is found, and t_{tot}[s] is the runtime until the heuristic terminates. The final six columns refer to the performance measures of our modified *MDD* approach (*MDD* + Gurobi), which are identical to those of the original *MDD* approach, namely: the achieved average solution quality (*obj*), the average gap between the upper and lower bounds (*gap*[%]), the time required to complete the *MDD* reduction steps (t_{prep}[s]), the total runtime of the algorithm (t_{tot}[s]), the number of instances that were solved by the ILP and the number of instances where the *MDD* reduction was fully completed separated by a / (#opt), and the average reduction of matches (*red*[%]). The respective best values for each complexity category (*obj*, *gap*, t_{tot}, t_{prep}, #opt, and *red*) are highlighted in bold.

5.2. Experimental Analysis

This section presents the comprehensive analysis of the experimental results, based on the experimental results Tables 5.1 and 5.2. We first discuss the quantitative results, which focus on the performance metrics and computational efficiency of the proposed approach. Following that, we provide a qualitative discussion.

5.2.1. Quantitative Results

Our approach successfully solves 76 problems to optimality that were previously unsolved in Instance Set 1, and 118 problems in Instance Set 2. There is no single instance that is solved previously that cannot be solved by out approach. In general, the runtime, the gap and the reduction percentages of our approach consistently outperforms the original *MDD* approach across every complexity category. We reach optimality for all instances except for instances with $m \in \{1024, 2048, 4096\}, \Sigma = \frac{m}{8}$, where none of the instances could be solved to optimality.

Complexity categories that are quick to solve by our algorithm are processed in under 10 milliseconds on average, there is a wide range of complexity categories that are solved by our solver in this time, but not by the original *MDD* approach. The most notable difference arises with the instance set $m = 512, \Sigma = \frac{m}{8}$, where our approach maintains an average solution time of 9 milliseconds, while the competing method requires, on average, 3435 seconds and is able to only solve 2 of these 30 instances (see table 5.1). This suggests a significant improvement offered by our approach. For mid-range challenging complexity categories that are completely solved to optimality by the original *MDD* approach, our method outperforms it by a factor of 5 to 40 in

Table 5.1.: Instance Set 1 Results

$\|\Sigma\|$	m	obj best	*MDD* + CPLEX (Horn)						Heuristic				*MDD* + Gurobi						
			obj	gap[%]	t_{prep}[s]	t_{tot}[s]	#opt	red[%]	obj	best	t_{sol}[s]	t_{tot}[s]	obj	gap[%]	t_{prep}[s]	t_{tot}[s]	#opt	red[%]	ram[MB]
	32	**4.00**	**4.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	94.94	**4.00**	**30**	< 0.01	< 0.01	**4.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**8.00**	**8.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	78.23	**8.00**	**30**	< 0.01	< 0.01	**8.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	128	**16.00**	**16.00**	**0.00**	0.04	0.04	**30/30**	44.45	**16.00**	**30**	< 0.01	< 0.01	**16.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	256	**31.97**	**31.97**	**0.00**	0.42	0.42	**30/30**	25.47	**31.97**	**30**	< 0.01	< 0.01	**31.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
$\frac{m}{8}$	512	63.90	62.80	1.82	60.82	3434.53	2/2	20.29	**63.93**	**30**	0.08	0.09	**63.93**	**0.00**	**< 0.01**	**0.09**	**30/30**	**100.00**	< 0.01
	640	-	-	-	-	-	-	-	79.33	28	0.52	1.72	**79.40**	**0.58**	**443.81**	**865.77**	**23/23**	**86.48**	8034.54
	768	-	-	-	-	-	-	-	**92.97**	**30**	1.10	7.27	**92.97**	**3.02**	**1794.62**	**3603.93**	**0/0**	**35.58**	32966.92
	1024	116.30	113.47	11.22	**271.15**	3946.52	**0/0**	23.34	**117.37**	**30**	2.69	15.25	**117.37**	**8.16**	1787.11	**3606.90**	**0/0**	**29.36**	32967.02
	2048	186.23	186.23	16.29	**1134.30**	4135.18	**0/0**	28.49	**192.03**	**30**	12.11	61.48	**192.03**	**13.28**	1740.25	**3614.95**	**0/0**	**36.24**	32502.27
	4096	292.03	292.03	11.05	4297.07	4297.07	**0/0**	36.59	**298.63**	**30**	41.79	311.00	**298.63**	**9.13**	**1491.87**	**3638.15**	**0/0**	**44.14**	7670.50
	32	**7.83**	**7.83**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	79.54	**7.83**	**30**	< 0.01	< 0.01	**7.83**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**14.67**	**14.67**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	84.06	**14.67**	**30**	< 0.01	< 0.01	**14.67**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	128	**25.93**	**25.93**	**0.00**	0.47	0.47	**30/30**	87.34	**25.93**	**30**	< 0.01	0.03	**25.93**	**0.00**	**0.04**	**0.07**	**30/30**	**100.00**	47.93
$\frac{m}{4}$	256	**43.97**	**43.97**	**0.00**	4.61	4.61	**30/30**	90.67	**43.97**	**30**	0.01	0.09	**43.97**	**0.00**	**0.13**	**0.22**	**30/30**	**100.00**	55.70
	512	**68.57**	**68.57**	**0.00**	28.43	38.93	**30**/28	91.01	**68.57**	**30**	0.03	0.44	**68.57**	**0.00**	**0.53**	**0.97**	**30/30**	**100.00**	67.20
	1024	**105.07**	**105.07**	**0.00**	122.81	188.27	**30**/24	91.56	**105.07**	**30**	0.15	1.67	**105.07**	**0.00**	**1.35**	**3.02**	**30/30**	**100.00**	87.07
	2048	156.87	156.87	0.37	576.39	1571.46	24/15	85.91	**156.93**	**30**	0.31	6.19	**156.93**	**0.00**	**3.64**	**9.83**	**30/30**	**100.00**	144.64
	4096	230.37	230.37	0.73	1996.26	4036.48	15/6	83.99	**230.57**	**30**	1.33	26.80	**230.57**	**0.00**	**14.20**	**41.00**	**30/30**	**100.00**	382.02
	32	**8.77**	**8.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	78.03	**8.77**	**30**	< 0.01	< 0.01	**8.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**15.53**	**15.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	81.16	**15.53**	**30**	< 0.01	< 0.01	**15.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	3.42
	128	**24.90**	**24.90**	**0.00**	0.06	0.06	**30/30**	86.44	**24.90**	**30**	< 0.01	< 0.01	**24.90**	**0.00**	**< 0.01**	**0.01**	**30/30**	**100.00**	1.73
$\frac{3 \cdot m}{8}$	256	**39.97**	**39.97**	**0.00**	0.50	0.50	**30/30**	89.20	**39.97**	**30**	< 0.01	0.03	**39.97**	**0.00**	**< 0.01**	**0.04**	**30/30**	**100.00**	8.84
	512	**59.97**	**59.97**	**0.00**	4.26	4.26	**30/30**	91.16	**59.97**	**30**	0.02	0.10	**59.97**	**0.00**	**0.03**	**0.14**	**30/30**	**100.00**	15.12
	1024	**90.73**	**90.73**	**0.00**	13.59	13.59	**30/30**	93.59	**90.73**	**30**	0.05	0.35	**90.73**	**0.00**	**0.07**	**0.42**	**30/30**	**100.00**	18.71
	2048	**131.17**	**131.17**	**0.00**	50.39	50.39	**30/30**	94.79	**131.17**	**30**	0.17	1.62	**131.17**	**0.00**	**0.32**	**1.94**	**30/30**	**100.00**	40.24
	4096	**193.37**	**193.37**	**0.00**	163.90	163.99	**30**/29	96.65	**193.37**	**30**	0.66	6.36	**193.37**	**0.00**	**0.78**	**7.15**	**30/30**	**100.00**	133.98
	32	**8.87**	**8.87**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	75.08	**8.87**	**30**	< 0.01	< 0.01	**8.87**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**14.80**	**14.80**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	81.22	**14.80**	**30**	< 0.01	< 0.01	**14.80**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	128	**22.93**	**22.93**	**0.00**	**0.01**	**0.01**	**30/30**	82.57	**22.93**	**30**	< 0.01	< 0.01	**22.93**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
$\frac{m}{2}$	256	**35.20**	**35.20**	**0.00**	0.07	0.07	**30/30**	87.37	**35.20**	**30**	< 0.01	0.01	**35.20**	**0.00**	**< 0.01**	**0.01**	**30/30**	**100.00**	< 0.01
	512	**53.13**	**53.13**	**0.00**	0.92	0.92	**30/30**	90.49	**53.13**	**30**	0.01	0.04	**53.13**	**0.00**	**< 0.01**	**0.04**	**30/30**	**100.00**	1.85
	1024	**79.13**	**79.13**	**0.00**	3.45	3.45	**30/30**	93.05	**79.13**	**30**	0.04	0.13	**79.13**	**0.00**	**0.01**	**0.14**	**30/30**	**100.00**	2.27
	2048	**115.70**	**115.70**	**0.00**	19.65	19.65	**30/30**	94.59	**115.70**	**30**	0.14	0.61	**115.70**	**0.00**	**0.05**	**0.65**	**30/30**	**100.00**	3.32
	4096	**167.97**	**167.97**	**0.00**	26.89	26.89	**30/30**	95.84	**167.97**	**30**	0.55	2.18	**167.97**	**0.00**	**0.09**	**2.28**	**30/30**	**100.00**	6.97
	32	**8.60**	**8.60**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	72.45	**8.60**	**30**	< 0.01	< 0.01	**8.60**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**13.30**	**13.30**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	78.35	13.27	29	< 0.01	< 0.01	**13.30**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**99.57**	1.70
	128	**21.20**	**21.20**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	83.47	**21.20**	**30**	< 0.01	< 0.01	**21.20**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
$\frac{5 \cdot m}{8}$	256	**32.53**	**32.53**	**0.00**	0.02	0.02	**30/30**	86.36	**32.53**	**30**	< 0.01	< 0.01	**32.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	512	**47.83**	**47.83**	**0.00**	0.10	0.10	**30/30**	88.68	**47.83**	**30**	0.01	0.02	**47.83**	**0.00**	**< 0.01**	**0.02**	**30/30**	**100.00**	< 0.01
	1024	**70.20**	**70.20**	**0.00**	0.76	0.76	**30/30**	91.35	**70.20**	**30**	0.04	0.08	**70.20**	**0.00**	**< 0.01**	**0.08**	**30/30**	**100.00**	< 0.01
	2048	**103.97**	**103.97**	**0.00**	3.27	3.27	**30/30**	93.98	**103.97**	**30**	0.13	0.32	**103.97**	**0.00**	**< 0.01**	**0.33**	**30/30**	**100.00**	< 0.01
	4096	**150.57**	**150.57**	**0.00**	10.26	10.26	**30/30**	95.77	**150.57**	**30**	0.50	1.29	**150.57**	**0.00**	**0.03**	**1.33**	**30/30**	**100.00**	6.63
	32	**8.17**	**8.17**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	71.83	**8.17**	**30**	< 0.01	< 0.01	**8.17**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**12.53**	**12.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	71.72	**12.53**	**30**	< 0.01	< 0.01	**12.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	128	**19.70**	**19.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	79.50	**19.70**	**30**	< 0.01	< 0.01	**19.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
$\frac{3 \cdot m}{4}$	256	**29.97**	**29.97**	**0.00**	**0.01**	**0.01**	**30/30**	84.41	**29.97**	**30**	< 0.01	< 0.01	**29.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	512	**44.57**	**44.57**	**0.00**	0.03	0.03	**30/30**	88.28	**44.57**	**30**	0.01	0.02	**44.57**	**0.00**	**< 0.01**	**0.02**	**30/30**	**100.00**	< 0.01
	1024	**65.20**	**65.20**	**0.00**	0.26	0.26	**30/30**	92.07	**65.20**	**30**	0.04	0.05	**65.20**	**0.00**	**< 0.01**	**0.06**	**30/30**	**100.00**	< 0.01
	2048	**94.67**	**94.67**	**0.00**	0.51	0.51	**30/30**	94.18	**94.67**	**30**	0.12	0.19	**94.67**	**0.00**	**< 0.01**	**0.19**	**30/30**	**100.00**	< 0.01
	4096	**136.77**	**136.77**	**0.00**	4.19	4.19	**30/30**	95.22	**136.77**	**30**	0.47	0.92	**136.77**	**0.00**	**0.02**	**0.93**	**30/30**	**100.00**	< 0.01
	32	**7.67**	**7.67**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	64.92	**7.67**	**30**	< 0.01	< 0.01	**7.67**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	64	**11.57**	**11.57**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	73.63	**11.57**	**30**	< 0.01	< 0.01	**11.57**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	128	**18.40**	**18.40**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	76.54	**18.40**	**30**	< 0.01	< 0.01	**18.40**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
$\frac{7 \cdot m}{8}$	256	**27.80**	**27.80**	**0.00**	**0.01**	**0.01**	**30/30**	84.25	**27.80**	**30**	< 0.01	< 0.01	**27.80**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	512	**40.60**	**40.60**	**0.00**	0.03	0.03	**30/30**	87.14	**40.60**	**30**	0.01	0.01	**40.60**	**0.00**	**< 0.01**	**0.01**	**30/30**	**100.00**	< 0.01
	1024	**60.57**	**60.57**	**0.00**	0.11	0.11	**30/30**	91.16	**60.57**	**30**	0.03	0.04	**60.57**	**0.00**	**< 0.01**	**0.04**	**30/30**	**100.00**	< 0.01
	2048	**88.00**	**88.00**	**0.00**	0.63	0.63	**30/30**	91.89	**88.00**	**30**	0.12	0.18	**88.00**	**0.00**	**< 0.01**	**0.19**	**30/30**	**100.00**	< 0.01
	4096	**127.37**	**127.37**	**0.00**	2.17	2.17	**30/30**	94.71	**127.37**	**30**	0.45	0.64	**127.37**	**0.00**	**< 0.01**	**0.65**	**30/30**	**100.00**	< 0.01

terms of runtime. Beyond this point, further runtime comparisons become inconclusive, as they would involve comparing instances where timeouts occur. Our upper and lower bound gaps are tighter across all three complexity categories in which we fail to prove optimality, compared to the original *MDD* approach. The heuristic achieves optimal solutions with only four exceptions. None of the instances that were solved to optimality required more than 3,71GB of memory.

5.2.2. Qualitative Discussion

Our algorithm for finding the *RFLCS* introduces significant innovations that surpass the state of the art. The remarkable performance of our heuristic in solution quality enables more efficient pruning of the search space, allowing for earlier termination compared to previous approaches. A key advantage of the heuristic is its ability to efficiently traverse the dominating match graph, enabling the algorithm to generate a large number of solutions and perform several resets upon reaching local maxima, all while maintaining fast runtimes. High-quality solutions are crucial for the subsequent algorithm steps. The dominating match graph implementation was specifically optimized for fast traversals. Additionally, due to the simpler structure of the dominating match graph, this approach is more suitable than the similar method employed on an *MDD*.

The character-by-character refinement process represents a major improvement over *MDD*

Table 5.2.: Instance Set 2 Results

$\|\Sigma\|$	reps	obj best	*MDD* + CPLEX (Horn)						Heuristic				*MDD* + Gurobi						
			obj	gap[%]	t_{prep}[s]	t_{tot}[s]	#opt	red[%]	obj	best	t_{sol}[s]	t_{tot}[s]	obj	gap[%]	t_{prep}[s]	t_{tot}[s]	#opt	red[%]	ram[MB]
4	3	**3.47**	**3.47**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	65.78	**3.47**	**30**	< 0.01	< 0.01	**3.47**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**3.77**	**3.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	76.59	**3.77**	**30**	< 0.01	< 0.01	**3.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	5	**3.83**	**3.83**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	81.44	**3.83**	**30**	< 0.01	< 0.01	**3.83**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	6	**3.90**	**3.90**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	85.92	**3.90**	**30**	< 0.01	< 0.01	**3.90**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	7	**3.97**	**3.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	88.59	**3.97**	**30**	< 0.01	< 0.01	**3.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	8	**3.97**	**3.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	89.54	**3.97**	**30**	< 0.01	< 0.01	**3.97**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
8	3	**6.23**	**6.23**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	66.94	**6.23**	**30**	< 0.01	< 0.01	**6.23**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**6.87**	**6.87**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	71.70	**6.87**	**30**	< 0.01	< 0.01	**6.87**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	5	**7.40**	**7.40**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	80.10	**7.40**	**30**	< 0.01	< 0.01	**7.40**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	6	**7.53**	**7.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	79.22	**7.53**	**30**	< 0.01	< 0.01	**7.53**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	7	**7.70**	**7.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	80.36	**7.70**	**30**	< 0.01	< 0.01	**7.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	8	**7.77**	**7.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	80.65	**7.77**	**30**	< 0.01	< 0.01	**7.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
16	3	**9.70**	**9.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	71.78	**9.70**	**30**	< 0.01	< 0.01	**9.70**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**11.57**	**11.57**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	79.44	**11.57**	**30**	< 0.01	< 0.01	**11.57**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	5	**12.93**	**12.93**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	79.97	**12.93**	**30**	< 0.01	< 0.01	**12.93**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	3.42
	6	**14.00**	**14.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	83.96	**14.00**	**30**	< 0.01	< 0.01	**14.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	7	**14.93**	**14.93**	**0.00**	**0.01**	**0.01**	**30/30**	84.81	**14.93**	**30**	< 0.01	< 0.01	**14.93**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	8	**14.80**	**14.80**	**0.00**	**0.01**	**0.01**	**30/30**	86.09	**14.80**	**30**	< 0.01	< 0.01	**14.80**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
32	3	**16.13**	**16.13**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	78.26	**16.13**	**30**	< 0.01	< 0.01	**16.13**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**19.00**	**19.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	81.46	**19.00**	**30**	< 0.01	< 0.01	**19.00**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	1.72
	5	**21.63**	**21.63**	**0.00**	0.04	0.04	**30/30**	85.76	**21.63**	**30**	< 0.01	< 0.01	**21.63**	**0.00**	**< 0.01**	**0.01**	**30/30**	**100.00**	12.12
	6	**23.73**	**23.73**	**0.00**	0.10	0.10	**30/30**	85.91	**23.73**	**30**	< 0.01	0.01	**23.73**	**0.00**	**0.02**	**0.03**	**30/30**	**100.00**	27.93
	7	**25.57**	**25.57**	**0.00**	0.61	0.61	**30/30**	89.21	**25.57**	**30**	< 0.01	0.02	**25.57**	**0.00**	**0.03**	**0.05**	**30/30**	**100.00**	37.12
	8	**27.50**	**27.50**	**0.00**	0.99	0.99	**30/30**	88.80	**27.50**	**30**	< 0.01	0.04	**27.50**	**0.00**	**0.07**	**0.10**	**30/30**	**100.00**	44.78
64	3	**25.43**	**25.43**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	82.34	**25.43**	**30**	< 0.01	< 0.01	**25.43**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**30.37**	**30.37**	**0.00**	0.04	0.04	**30/30**	86.45	**30.37**	**30**	< 0.01	< 0.01	**30.37**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	1.74
	5	**34.93**	**34.93**	**0.00**	0.74	0.74	**30/30**	86.26	**34.93**	**30**	< 0.01	0.02	**34.93**	**0.00**	**0.01**	**0.04**	**30/30**	**100.00**	15.85
	6	**39.13**	**39.13**	**0.00**	2.03	2.03	**30/30**	90.35	**39.13**	**30**	< 0.01	0.05	**39.13**	**0.00**	**0.05**	**0.10**	**30/30**	**100.00**	43.75
	7	**43.63**	**43.63**	**0.00**	7.51	7.92	**30**/29	89.54	**43.63**	**30**	< 0.01	0.10	**43.63**	**0.00**	**0.18**	**0.27**	**30/30**	**100.00**	59.18
	8	**45.53**	**45.53**	**0.00**	14.14	27.62	**30**/25	84.73	**45.53**	**30**	0.02	0.17	**45.53**	**0.00**	**0.42**	**0.59**	**30/30**	**100.00**	67.16
128	3	**36.77**	**36.77**	**0.00**	0.02	0.02	**30/30**	85.18	**36.77**	**30**	< 0.01	< 0.01	**36.77**	**0.00**	**< 0.01**	**< 0.01**	**30/30**	**100.00**	< 0.01
	4	**45.03**	**45.03**	**0.00**	0.37	0.37	**30/30**	87.78	**45.03**	**30**	< 0.01	0.03	**45.03**	**0.00**	**< 0.01**	**0.03**	**30/30**	**100.00**	1.79
	5	**53.43**	**53.43**	**0.00**	3.30	3.30	**30/30**	90.42	**53.43**	**30**	0.01	0.07	**53.43**	**0.00**	**0.03**	**0.10**	**30/30**	**100.00**	14.77
	6	**61.53**	**61.53**	**0.00**	8.17	8.17	**30/30**	92.79	**61.53**	**30**	0.02	0.19	**61.53**	**0.00**	**0.12**	**0.31**	**30/30**	**100.00**	43.36
	7	**68.47**	**68.47**	**0.00**	29.22	36.33	**30**/28	91.40	**68.47**	**30**	0.03	0.45	**68.47**	**0.00**	**0.44**	**0.89**	**30/30**	**100.00**	65.77
	8	**74.60**	**74.60**	0.11	74.85	1010.01	28/12	68.99	**74.60**	**30**	0.08	0.84	**74.60**	**0.00**	**3.41**	**4.24**	**30/30**	**100.00**	136.83
256	3	**55.03**	**55.03**	**0.00**	0.07	0.07	**30/30**	89.31	**55.03**	**30**	0.01	0.03	**55.03**	**0.00**	**< 0.01**	**0.03**	**30/30**	**100.00**	< 0.01
	4	**68.93**	**68.93**	**0.00**	1.82	1.82	**30/30**	92.27	**68.93**	**30**	0.02	0.07	**68.93**	**0.00**	**< 0.01**	**0.08**	**30/30**	**100.00**	< 0.01
	5	**81.43**	**81.43**	**0.00**	12.95	12.95	**30/30**	93.83	**81.43**	**30**	0.04	0.25	**81.43**	**0.00**	**0.06**	**0.32**	**30/30**	**100.00**	18.98
	6	**93.60**	**93.60**	**0.00**	45.63	48.11	**30**/28	93.00	93.57	29	0.05	0.72	**93.60**	**0.00**	**0.43**	**1.15**	**30/30**	**99.90**	58.28
	7	**104.50**	**104.50**	**0.00**	129.27	369.43	**30**/20	87.91	**104.50**	**30**	0.10	1.46	**104.50**	**0.00**	**1.18**	**2.64**	**30/30**	**100.00**	84.53
	8	115.07	115.03	2.70	375.68	3167.78	10/2	62.22	**115.23**	**30**	0.17	3.17	**115.23**	**0.00**	**11.39**	**14.56**	**30/30**	**100.00**	288.42
512	3	**81.63**	**81.63**	**0.00**	0.58	0.58	**30/30**	92.84	**81.63**	**30**	0.04	0.08	**81.63**	**0.00**	**< 0.01**	**0.08**	**30/30**	**100.00**	< 0.01
	4	**101.13**	**101.13**	**0.00**	9.01	9.01	**30/30**	93.71	**101.13**	**30**	0.07	0.29	**101.13**	**0.00**	**0.02**	**0.32**	**30/30**	**100.00**	2.66
	5	**121.03**	**121.03**	**0.00**	37.06	37.06	**30/30**	95.16	**121.03**	**30**	0.11	1.04	**121.03**	**0.00**	**0.21**	**1.24**	**30/30**	**100.00**	37.59
	6	**138.60**	**138.60**	**0.00**	143.51	148.66	**30**/27	94.86	**138.60**	**30**	0.23	3.09	**138.60**	**0.00**	**1.17**	**4.26**	**30/30**	**100.00**	103.81
	7	**156.00**	**156.00**	0.70	679.41	2115.36	20/10	82.00	**156.00**	**30**	0.28	6.56	**156.00**	**0.00**	**4.64**	**11.20**	**30/30**	**100.00**	156.38
	8	174.23	174.23	3.25	1221.36	4499.14	3/1	64.24	**174.77**	**30**	0.54	13.85	**174.77**	**0.00**	**36.95**	**50.80**	**30/30**	**100.00**	624.00

methods, as it eliminates the need for fine-tuning a fixed maximum width. Fixed widths often hinder refinement by preventing the *MDD* from expanding adequately on important levels, thereby blocking essential exploration.

Furthermore, the refinement scheme incorporates precise knowledge about whether a character is definitively present or absent on paths entering a node, enabling more effective pruning of edges and improving the overall quality of refinement.

The ordering of characters during refinement is also critically important. Although determining character orderings can dominate the *MDD* phase runtime, it is worthwhile to invest computational resources in finding well behaving orderings. By leveraging a character's impact measure on the *MDD*, the algorithm effectively prunes matches from the search space while simultaneously managing the size of the *MDD*, thereby enhancing both efficiency and overall performance.

Our results suggest that in practice, optimally solving instances that are not already solved by the *MDD* reduction step of our algorithms is in many cases hard for the ILP solver, and there are only rare cases where running it at all will still yield a better solution than the one determined by the *MDD*. Specifically, given the same time and computational resources, we are more likely to solve instances with the *MDD* than the ILP.

In summary, we solved 68% of the previously unsolved problem instances. Also the runtime for solving both instance sets is 27 times faster than the original *MDD* approach, when comparing only the complexity categories in which the original *MDD* approach solves all 30 instances optimally.

6. Future Work

One potential approach to solving more instances optimally is to maintain two *MDD*s, each solving the *RFLCS* problem for result lengths up to a predefined threshold—one for the forward problem and one for the backward problem. By exhaustively combining nodes from both *MDD*s, regarding their solution characters, it may be possible to construct longer solutions or demonstrate that no solution longer than the current lower bound exists. Reducing the depths of both *MDD*s should improve computational efficiency compared to constructing a full *MDD* for the entire problem. However, it should be emphasized that this approach is unlikely to be a breakthrough; rather, it serves as a way to extend the capabilities of the *MDD* approach.

The heuristic and *MDD* phases are parallelizable using MPI. Multiple MPI ranks can run the heuristic independently while sharing the current lower bound. In the *MDD* phase, each MPI rank can be assigned several consecutive levels of the *MDD*, with communication occurring at the boundary levels between ranks.

The *MDD* approach can be extended to solve the *RFLCS*-IG problem [WZ23] by incorporating pruning rules that account for index genes. A sufficient condition to ensure that the resulting *RFLCS* includes all index genes is to enforce the constraint that the set $\text{pre} \cup \text{app}$ contains all index genes.

Bibliography

[ABF+08] Said Sadique Adi, Marília DV Braga, Cristina G Fernandes, Carlos Eduardo Ferreira, Fábio Viduani Martinez, M-F Sagot, Marco Aurelio Stefanes, Christian Tjandraatmadja, and Yoshiko Wakabayashi. Repetition-free longest common subsequence. *Electronic Notes in Discrete Mathematics*, 30:243–248, 2008.

[AGM+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[BB16] Christian Blum and Maria J Blesa. Construct, merge, solve and adapt: application to the repetition-free longest common subsequence problem. In *Evolutionary Computation in Combinatorial Optimization: 16th European Conference, EvoCOP 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings 16*, pages 46–57. Springer, 2016.

[BB18] Christian Blum and Maria J Blesa. A comprehensive comparison of metaheuristics for the repetition-free longest common subsequence problem. *Journal of Heuristics*, 24(3):551–579, 2018.

[BBC13] Christian Blum, Maria J Blesa, and Borja Calvo. Beam-aco for the repetition-free longest common subsequence problem. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 79–90. Springer, 2013.

[BDS+21] Christian Blum, Marko Djukanovic, Alberto Santini, Hua Jiang, Chu-Min Li, Felip Manyà, and Günter R Raidl. Solving longest common subsequence problems via a transformation to the maximum clique problem. *Computers & Operations Research*, 125:105089, 2021.

[Ber57] Claude Berge. *Théorie des graphes et ses applications*. Dunod, 1957.

[BNM17] Kensuke Baba, Tetsuya Nakatoh, and Toshiro Minami. Plagiarism detection using document similarity based on distributed representation. *Procedia computer science*, 111:382–387, 2017.

[CBV13] Mauro Castelli, Stefano Beretta, and Leonardo Vanneschi. A hybrid genetic algorithm for the repetition free longest common subsequence problem. *Operations Research Letters*, 41(6):644–649, 2013.

[FK16] Cristina G Fernandes and Marcos Kiwi. Repetition-free longest common subsequence of random sequences. *Discrete Applied Mathematics*, 210:75–87, 2016.

[FT10] Carlos E Ferreira and Christian Tjandraatmadja. A branch-and-cut approach to the repetition-free longest common subsequence problem. *Electronic Notes in Discrete Mathematics*, 36:527–534, 2010.

[HDBR20] Matthias Horn, Marko Djukanovic, Christian Blum, and Günther R Raidl. On the use of decision diagrams for finding repetition-free longest common subsequences. In *Optimization and Applications: 11th International Conference, OPTIMA 2020, Moscow, Russia, September 28–October 2, 2020, Proceedings 11*, pages 134–149. Springer, 2020.

[HM76] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[HP11] Mark Hauschild and Martin Pelikan. An introduction and survey of estimation of distribution algorithms. *Swarm and evolutionary computation*, 1(3):111–128, 2011.

[HS77] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[IR09] Costas S Iliopoulos and M Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory of Computing Systems*, 45(2):355–371, 2009.

[Joh73] David S Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49, 1973.

[KHO21] Masashi Kiyomi, Takashi Horiyama, and Yota Otachi. Longest common subsequence in sublinear space. *Information Processing Letters*, 168:106084, 2021.

[PL88] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[San99] David Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999.

[WZ23] Qichen Wang and Shu Zhang. Hybrid heuristics: for the repetition-free longest common subsequence problem with index-gene. In *2023 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 3382–3389. IEEE, 2023.

A. An MDD-Based ILP

We formulate the *RFLCS* problem as an ILP based on a precomputed *MDD*, with the objective of maximizing the number of traversed nodes while ensuring compliance with the repetition-free constraint. To achieve this, we define a decision variable for each node in the *MDD*. Additionally, we introduce predecessor edge constraints to further restrict the solution space. The ILP formulation is defined as follows:

$$
\begin{aligned}
\textbf{Objective:} \quad & \text{maximize} \sum_{n \in MDD} z_n && \text{(Maximizing subsequence length)} \\
\textbf{Subject to:} \quad & \sum_{\substack{n \in MDD \\ \text{char}(n) = \sigma}} z_n \leq 1 \quad \forall \sigma \in \Sigma && \text{(Repetition-free constraint)} \\
& \sum_{n \in MDD_i} z_n \leq 1 \quad \forall MDD_i && \text{(One node per level constraint)} \\
& z_n \leq \sum_{n' \in pred(n)} z_{n'} \quad \forall n \in MDD \setminus \{r\} && \text{(Predecessor inclusion constraint)} \\
& z_n \in \{0, 1\} \quad \forall n \in MDD && \text{(Binary decision variable)}
\end{aligned}
$$

The objective function aims to maximize the number of selected nodes, thereby identifying the longest valid traversal through the *MDD* that satisfies all constraints. The repetition-free constraint ensures that no character appears more than once in the selected subsequence. The level constraint enforces that at most one node is chosen per *MDD* level. Additionally to preserve the common subsequence constraint of *RFLCS*, the predecessor inclusion constraint guarantees that a node can only be selected if one of its predecessors is also included in the solution.

Finally, we handle the root node separately, as it does not contribute to the actual *RFLCS* and is therefore excluded from the final solution.

A.1. Experimental Methodology

In this section, we replicate the experiments described in section 5.1, and also in addition utilizing the *MDD*-based ILP. To generate ILP instances that are non-trivial to solve, we omit the *MDD* refinement and filtering steps. Additionally, we exclude complexity categories where the ILP fails to produce meaningful results, leaving us with the data presented in table A.1 and table A.2.

A.2. Results Discussion

The results reveal variations in runtime behavior between the two approaches; however, neither consistently outperforms the other. For instance, in instance set 1, where $\Sigma = \frac{n}{4}$, no clear trend emerges favoring one approach over the other, with performance advantages alternating as input string lengths increase. Notably, in instance set 2, we identified a single case where the *MDD*-based ILP successfully derived the optimal upper bound, thereby proving optimality—whereas the match-based ILP did not. Conversely, the match-based ILP found a better solution than the *MDD*-based ILP in one of the two complexity categories that were generated by us.

Table A.1.: No *MDD* Instance Set 1 Results

$\|\Sigma\|$	m	Match ILP			*MDD* ILP		
		obj	t_{ilp}[s]	gap[%]	obj	t_{ilp}[s]	gap[%]
$\frac{m}{8}$	32	**4.00**	**< 0.01**	**0.00**	**4.00**	**< 0.01**	**0.00**
	64	**8.00**	**< 0.01**	**0.00**	**8.00**	**< 0.01**	**0.00**
	128	**16.00**	**< 0.01**	**0.00**	**16.00**	**< 0.01**	**0.00**
	256	**31.97**	**< 0.01**	**0.00**	**31.97**	**< 0.01**	**0.00**
	512	**63.93**	**< 0.01**	**0.00**	**63.93**	**< 0.01**	**0.00**
	640	**79.37**	**532.91**	**0.62**	79.33	660.10	0.75
$\frac{m}{4}$	32	**7.83**	**< 0.01**	**0.00**	**7.83**	**< 0.01**	**0.00**
	64	**14.67**	**< 0.01**	**0.00**	**14.67**	**< 0.01**	**0.00**
	128	**25.93**	0.85	**0.00**	**25.93**	**0.35**	**0.00**
	256	**43.97**	**10.19**	**0.00**	**43.97**	14.15	**0.00**
	512	**68.57**	**73.21**	**0.00**	**68.57**	101.70	**0.00**
	1024	**105.07**	232.11	**0.00**	**105.07**	**111.88**	**0.00**
$\frac{3 \cdot m}{8}$	32	**8.77**	**< 0.01**	**0.00**	**8.77**	**< 0.01**	**0.00**
	64	**15.53**	**< 0.01**	**0.00**	**15.53**	**< 0.01**	**0.00**
	128	**24.90**	**< 0.01**	**0.00**	**24.90**	**< 0.01**	**0.00**
	256	**39.97**	**< 0.01**	**0.00**	**39.97**	**< 0.01**	**0.00**
	512	**59.97**	0.57	**0.00**	**59.97**	**< 0.01**	**0.00**
	1024	**90.73**	0.99	**0.00**	**90.73**	**0.03**	**0.00**
	2048	**131.17**	6.83	**0.00**	**131.17**	**0.41**	**0.00**
	4096	**193.37**	10.64	**0.00**	**193.37**	**0.16**	**0.00**
$\frac{m}{2}$	32	**8.87**	**< 0.01**	**0.00**	**8.87**	**< 0.01**	**0.00**
	64	**14.80**	**< 0.01**	**0.00**	**14.80**	**< 0.01**	**0.00**
	128	**22.93**	**< 0.01**	**0.00**	**22.93**	**< 0.01**	**0.00**
	256	**35.20**	**< 0.01**	**0.00**	**35.20**	**< 0.01**	**0.00**
	512	**53.13**	**< 0.01**	**0.00**	**53.13**	**< 0.01**	**0.00**
	1024	**79.13**	**< 0.01**	**0.00**	**79.13**	**< 0.01**	**0.00**
	2048	**115.70**	**< 0.01**	**0.00**	**115.70**	**< 0.01**	**0.00**
	4096	**167.97**	**< 0.01**	**0.00**	**167.97**	**< 0.01**	**0.00**
$\frac{5 \cdot m}{8}$	32	**8.60**	**< 0.01**	**0.00**	**8.60**	**< 0.01**	**0.00**
	64	**13.30**	**< 0.01**	**0.00**	**13.30**	**< 0.01**	**0.00**
	128	**21.20**	**< 0.01**	**0.00**	**21.20**	**< 0.01**	**0.00**
	256	**32.53**	**< 0.01**	**0.00**	**32.53**	**< 0.01**	**0.00**
	512	**47.83**	**< 0.01**	**0.00**	**47.83**	**< 0.01**	**0.00**
	1024	**70.20**	**< 0.01**	**0.00**	**70.20**	**< 0.01**	**0.00**
	2048	**103.97**	**< 0.01**	**0.00**	**103.97**	**< 0.01**	**0.00**
	4096	**150.57**	**< 0.01**	**0.00**	**150.57**	**< 0.01**	**0.00**
$\frac{3 \cdot m}{4}$	32	**8.17**	**< 0.01**	**0.00**	**8.17**	**< 0.01**	**0.00**
	64	**12.53**	**< 0.01**	**0.00**	**12.53**	**< 0.01**	**0.00**
	128	**19.70**	**< 0.01**	**0.00**	**19.70**	**< 0.01**	**0.00**
	256	**29.97**	**< 0.01**	**0.00**	**29.97**	**< 0.01**	**0.00**
	512	**44.57**	**< 0.01**	**0.00**	**44.57**	**< 0.01**	**0.00**
	1024	**65.20**	**< 0.01**	**0.00**	**65.20**	**< 0.01**	**0.00**
	2048	**94.67**	**< 0.01**	**0.00**	**94.67**	**< 0.01**	**0.00**
	4096	**136.77**	**< 0.01**	**0.00**	**136.77**	**< 0.01**	**0.00**
$\frac{7 \cdot m}{8}$	32	**7.67**	**< 0.01**	**0.00**	**7.67**	**< 0.01**	**0.00**
	64	**11.57**	**< 0.01**	**0.00**	**11.57**	**< 0.01**	**0.00**
	128	**18.40**	**< 0.01**	**0.00**	**18.40**	**< 0.01**	**0.00**
	256	**27.80**	**< 0.01**	**0.00**	**27.80**	**< 0.01**	**0.00**
	512	**40.60**	**< 0.01**	**0.00**	**40.60**	**< 0.01**	**0.00**
	1024	**60.57**	**< 0.01**	**0.00**	**60.57**	**< 0.01**	**0.00**
	2048	**88.00**	**< 0.01**	**0.00**	**88.00**	**< 0.01**	**0.00**
	4096	**127.37**	**< 0.01**	**0.00**	**127.37**	**< 0.01**	**0.00**

Table A.2.: No *MDD* Instance Set 2 Results

$\|\Sigma\|$	reps	Match ILP			*MDD* ILP		
		obj	t_{ilp}[s]	gap[%]	obj	t_{ilp}[s]	gap[%]
4	3	**3.47**	**< 0.01**	**0.00**	**3.47**	**< 0.01**	**0.00**
	4	**3.77**	**< 0.01**	**0.00**	**3.77**	**< 0.01**	**0.00**
	5	**3.83**	**< 0.01**	**0.00**	**3.83**	**< 0.01**	**0.00**
	6	**3.90**	**< 0.01**	**0.00**	**3.90**	**< 0.01**	**0.00**
	7	**3.97**	**< 0.01**	**0.00**	**3.97**	**< 0.01**	**0.00**
	8	**3.97**	**< 0.01**	**0.00**	**3.97**	**< 0.01**	**0.00**
8	3	**6.23**	**< 0.01**	**0.00**	**6.23**	**< 0.01**	**0.00**
	4	**6.87**	**< 0.01**	**0.00**	**6.87**	**< 0.01**	**0.00**
	5	**7.40**	**< 0.01**	**0.00**	**7.40**	**< 0.01**	**0.00**
	6	**7.53**	**< 0.01**	**0.00**	**7.53**	**< 0.01**	**0.00**
	7	**7.70**	**< 0.01**	**0.00**	**7.70**	**< 0.01**	**0.00**
	8	**7.77**	**< 0.01**	**0.00**	**7.77**	**< 0.01**	**0.00**
16	3	**9.70**	**< 0.01**	**0.00**	**9.70**	**< 0.01**	**0.00**
	4	**11.57**	**< 0.01**	**0.00**	**11.57**	**< 0.01**	**0.00**
	5	**12.93**	**< 0.01**	**0.00**	**12.93**	**< 0.01**	**0.00**
	6	**14.00**	**< 0.01**	**0.00**	**14.00**	**< 0.01**	**0.00**
	7	**14.93**	**< 0.01**	**0.00**	**14.93**	**< 0.01**	**0.00**
	8	**14.80**	**< 0.01**	**0.00**	**14.80**	**< 0.01**	**0.00**
32	3	**16.13**	**< 0.01**	**0.00**	**16.13**	**< 0.01**	**0.00**
	4	**19.00**	**< 0.01**	**0.00**	**19.00**	**< 0.01**	**0.00**
	5	**21.63**	**< 0.01**	**0.00**	**21.63**	**< 0.01**	**0.00**
	6	**23.73**	0.06	**0.00**	**23.73**	**0.05**	**0.00**
	7	**25.57**	0.39	**0.00**	**25.57**	**0.13**	**0.00**
	8	**27.50**	**1.00**	**0.00**	**27.50**	11.35	**0.00**
64	3	**25.43**	**< 0.01**	**0.00**	**25.43**	**< 0.01**	**0.00**
	4	**30.37**	**< 0.01**	**0.00**	**30.37**	**< 0.01**	**0.00**
	5	**34.93**	0.06	**0.00**	**34.93**	**< 0.01**	**0.00**
	6	**39.13**	1.83	**0.00**	**39.13**	**0.11**	**0.00**
	7	**43.63**	**14.08**	**0.00**	**43.63**	23.97	**0.00**
	8	**45.53**	**48.16**	**0.00**	**45.53**	261.40	**0.00**
128	3	**36.77**	**< 0.01**	**0.00**	**36.77**	**< 0.01**	**0.00**
	4	**45.03**	**< 0.01**	**0.00**	**45.03**	**< 0.01**	**0.00**
	5	**53.43**	0.31	**0.00**	**53.43**	**< 0.01**	**0.00**
	6	**61.53**	11.01	**0.00**	**61.53**	**0.15**	**0.00**
	7	**68.47**	64.23	**0.00**	**68.47**	**27.69**	**0.00**
256	3	**55.03**	**< 0.01**	**0.00**	**55.03**	**< 0.01**	**0.00**
	4	**68.93**	**< 0.01**	**0.00**	**68.93**	**< 0.01**	**0.00**
	5	**81.43**	0.07	**0.00**	**81.43**	**< 0.01**	**0.00**
	6	**93.60**	29.39	**0.00**	**93.60**	**11.92**	**0.00**
512	3	**81.63**	**< 0.01**	**0.00**	**81.63**	**< 0.01**	**0.00**
	4	**101.13**	**< 0.01**	**0.00**	**101.13**	**< 0.01**	**0.00**
	5	**121.03**	2.17	**0.00**	**121.03**	**0.02**	**0.00**
	6	**138.60**	212.66	0.05	**138.60**	**11.46**	**0.00**

B. Further Experimental Result Visualizations

As a complement to the results presented in Tables 5.1 and 5.2, the following pages contain box plots that illustrate the minimum, maximum, mean, and the 25th and 75th percentiles of runtime, solution lengths, and *MDD* memory usage. In addition, we include box plots for the experiments shown in Tables A.1 and A.2, where the *MDD* stage was omitted, to compare the two proposed ILP approaches in terms of runtime and solution length. We also present three time series for the complexity category with $\Sigma = 512$ and 8 repetitions, showing the *MDD* behavior over time (by refined characters) with respect to node count, edge count, and edge-to-node ratio.

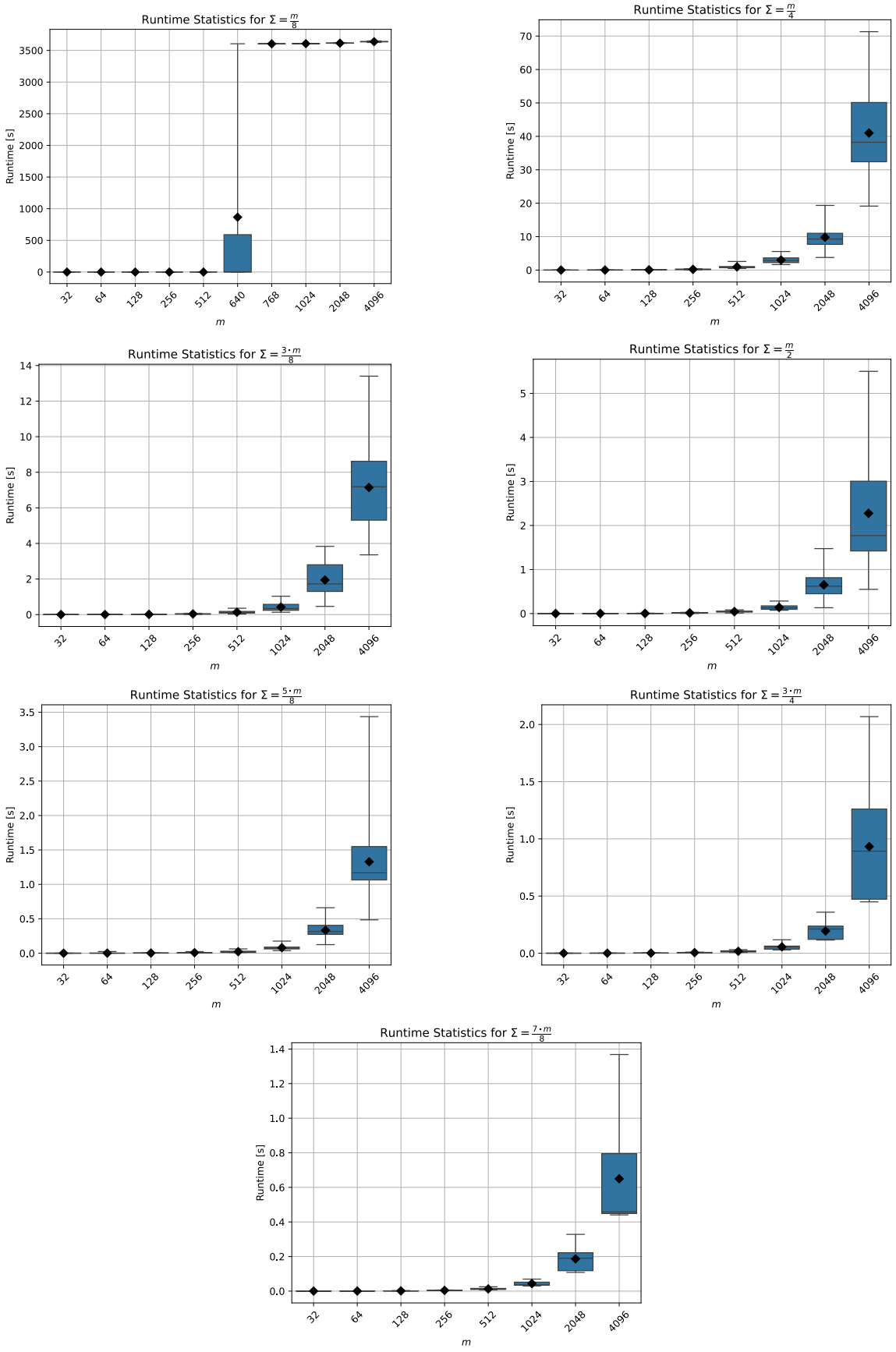


Figure B.1.: Box plots of runtime (in s) for Instance Set 1. Each subfigure corresponds to a different relative alphabet size. The y-axis shows runtimes, and the x-axis shows input string lengths. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

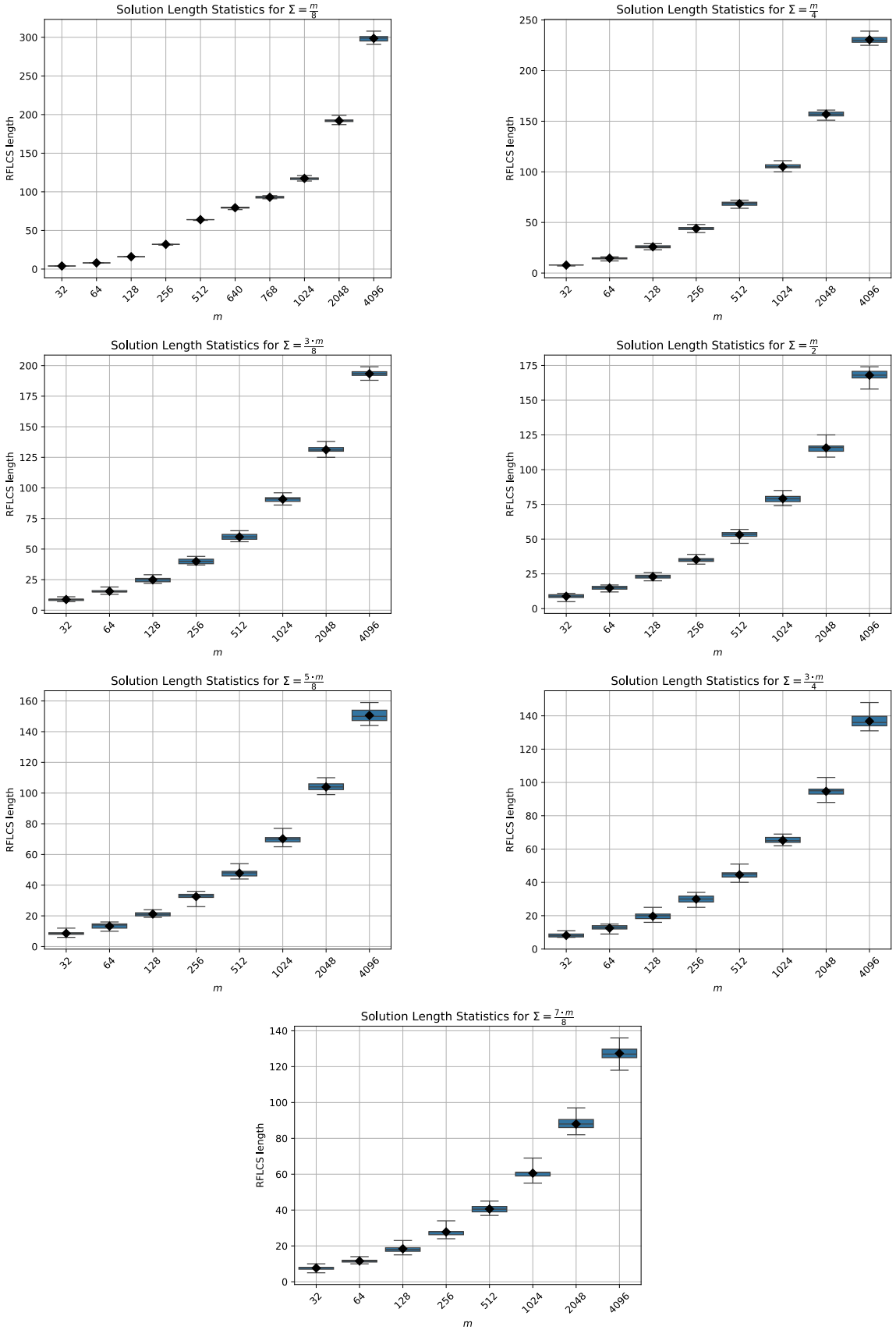


Figure B.2.: Box plots of solution lengths for Instance Set 1. Each subfigure corresponds to a different relative alphabet size. The y-axis shows solution lengths, and the x-axis shows input string lengths. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

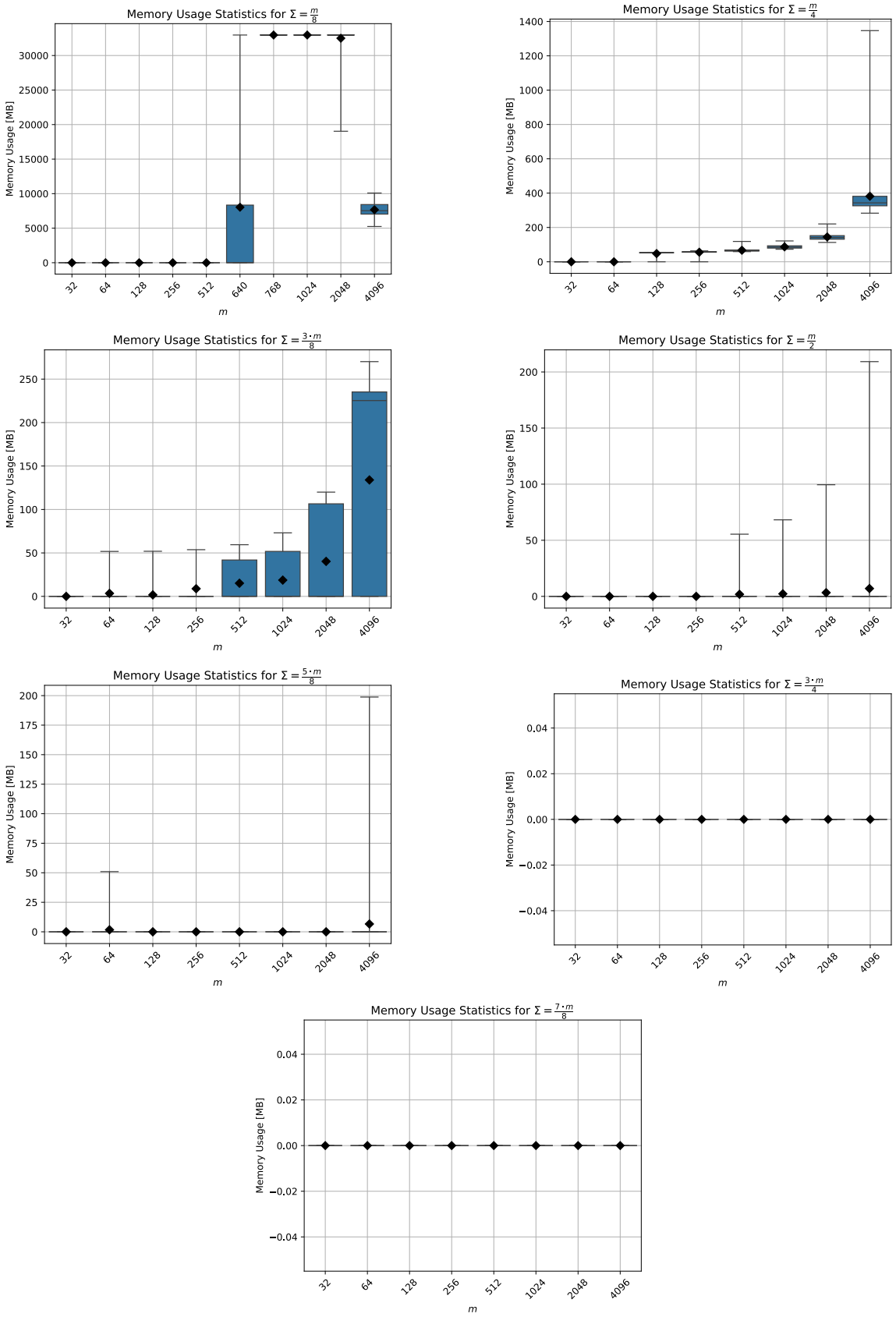


Figure B.3.: Box plots of memory consumption (in MB) for Instance Set 1. Each subfigure corresponds to a different relative alphabet size. The y-axis shows memory usage, and the x-axis shows input string lengths. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

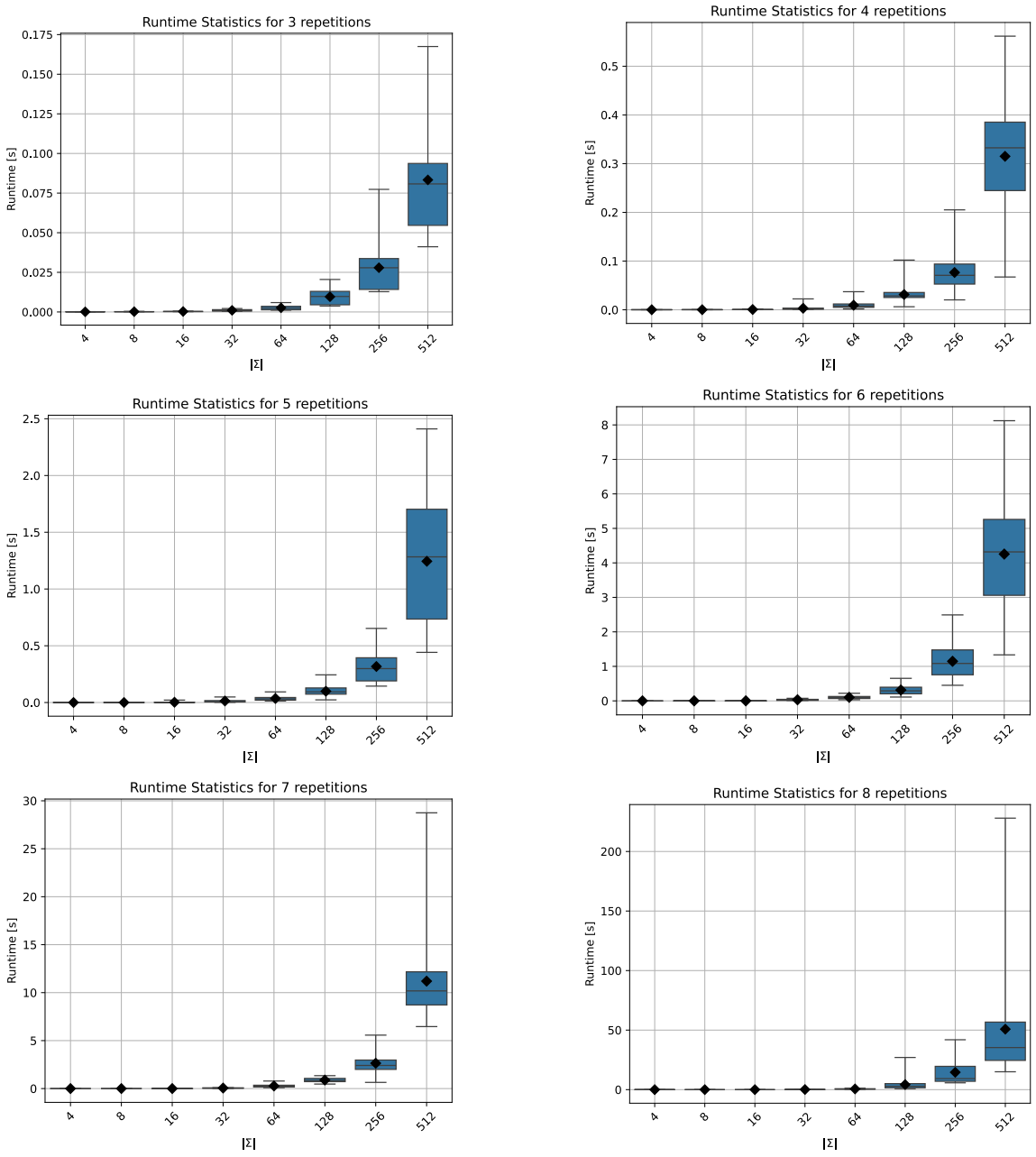


Figure B.4.: Box plots of runtime (in s) for Instance Set 2. Each subfigure corresponds to a different number of character repetitions. The y-axis shows runtimes, and the x-axis shows the alphabet size. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

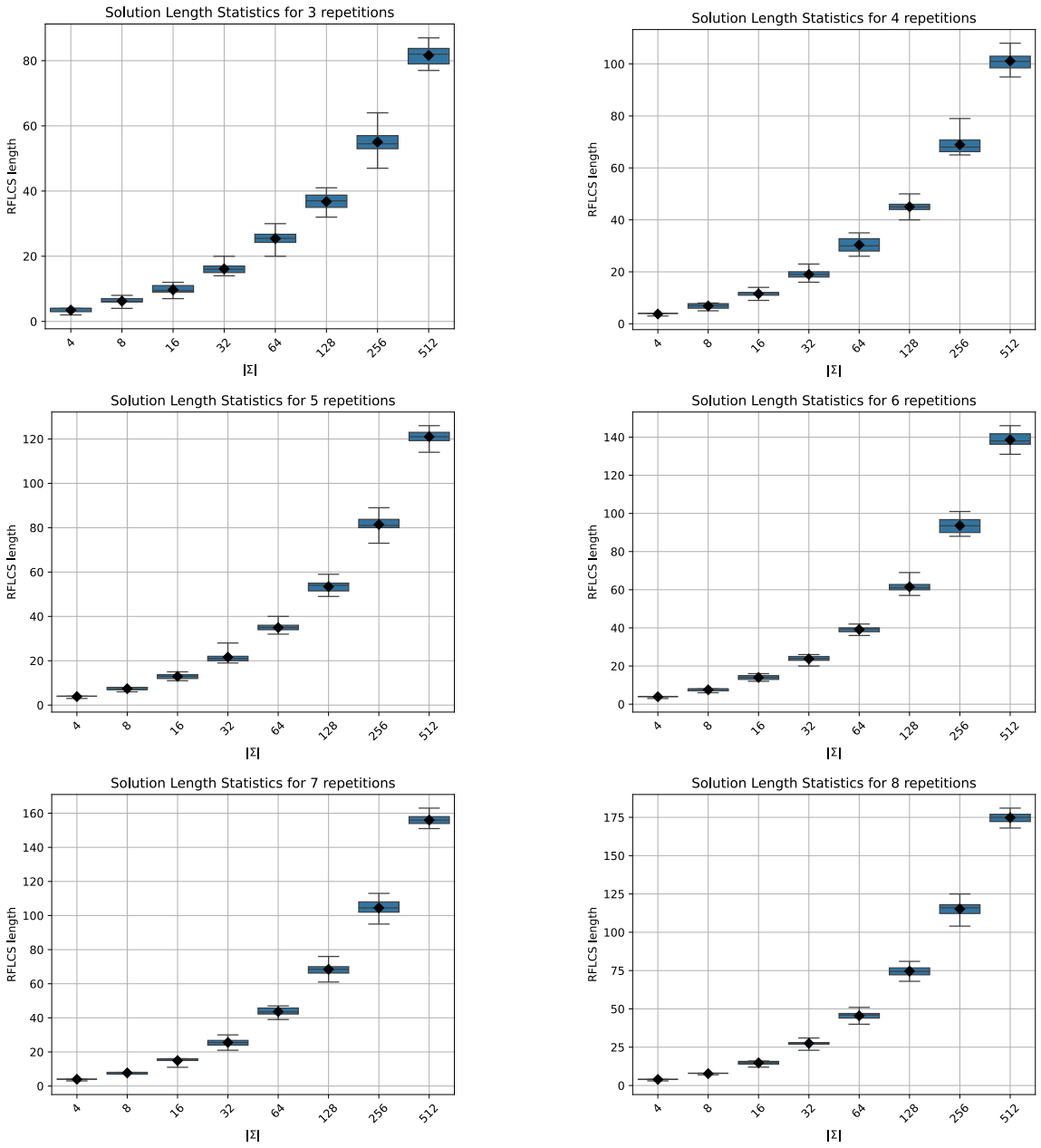


Figure B.5.: Box plots of solution lengths for Instance Set 2. Each subfigure corresponds to a different number of character repetitions. The y-axis shows solution lengths, and the x-axis shows the alphabet size. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

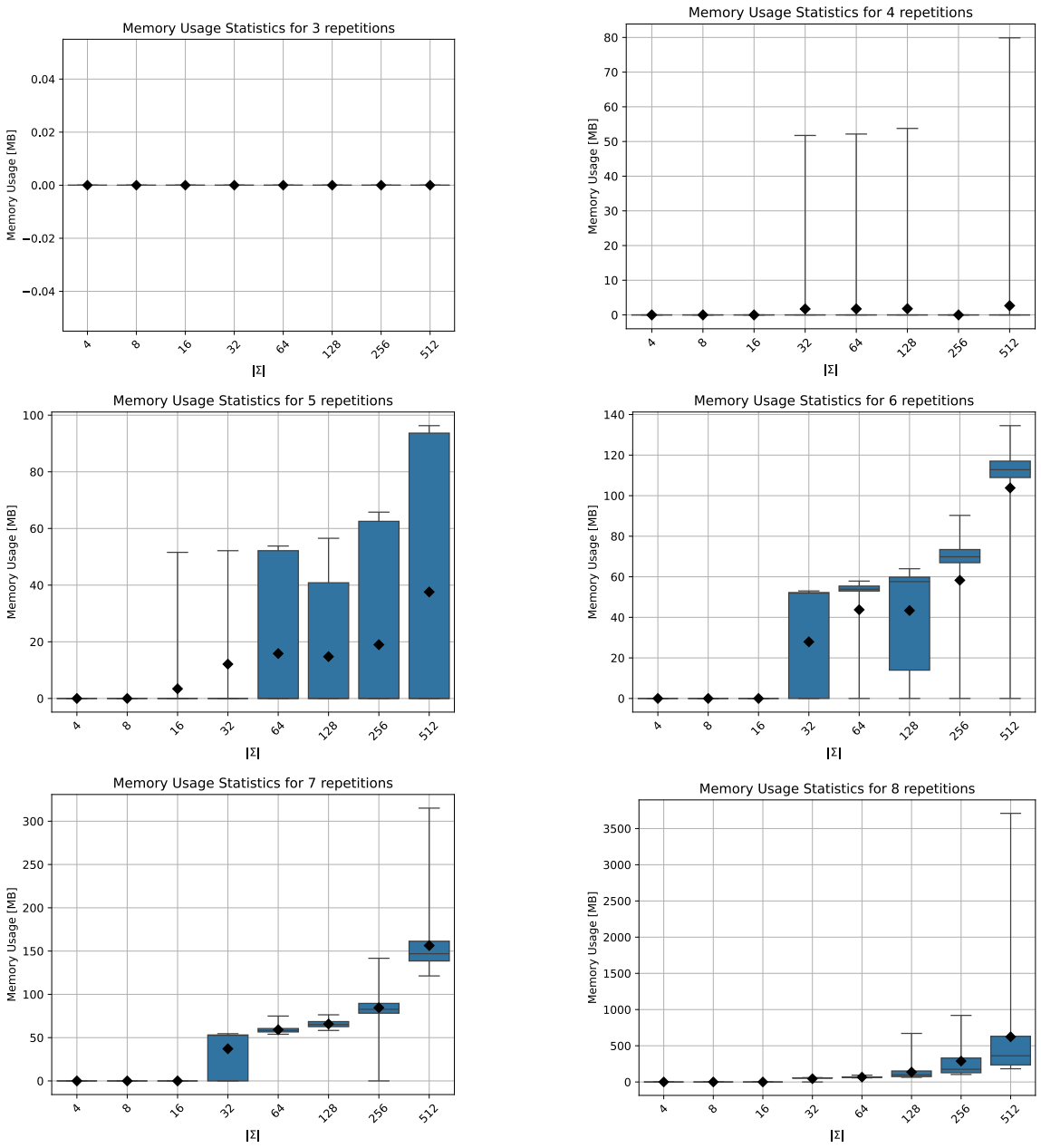


Figure B.6.: Box plots of memory consumption (in MB) for Instance Set 2. Each subfigure corresponds to a different number of character repetitions. The y-axis shows memory usage, and the x-axis shows the alphabet size. The box plots represent the minimum, maximum, average, as well as the 25% and 75% quantiles across runs.

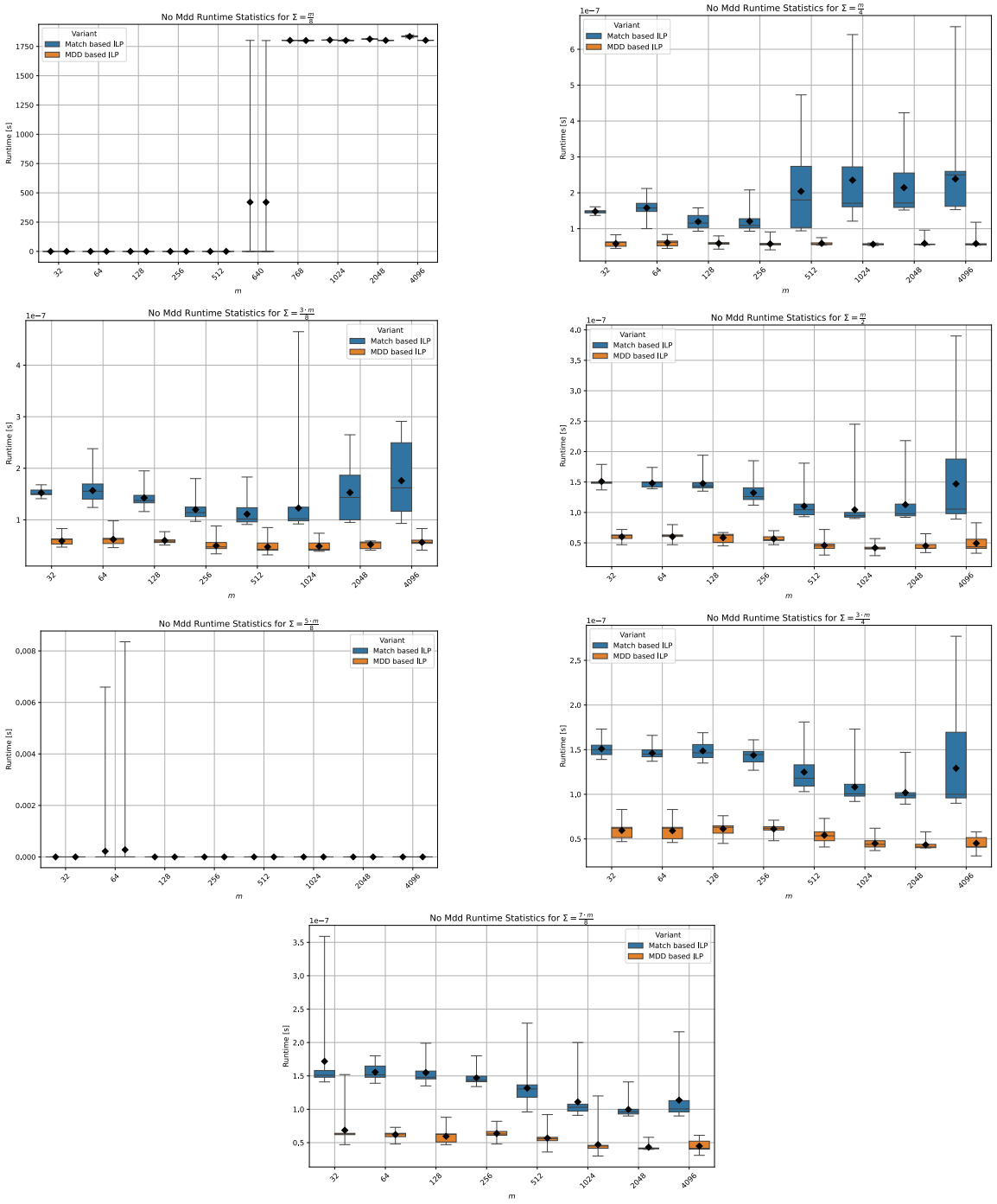


Figure B.7.: Box plots of runtime (in seconds) for Instance Set 1 when the *MDD* phase is skipped, comparing the match-based and MDD-based ILP approaches. Each subfigure corresponds to a different relative alphabet size. The y-axis shows runtime, and the x-axis shows input string lengths. The box plots indicate the minimum, maximum, average, and the 25% and 75% quantiles across runs.

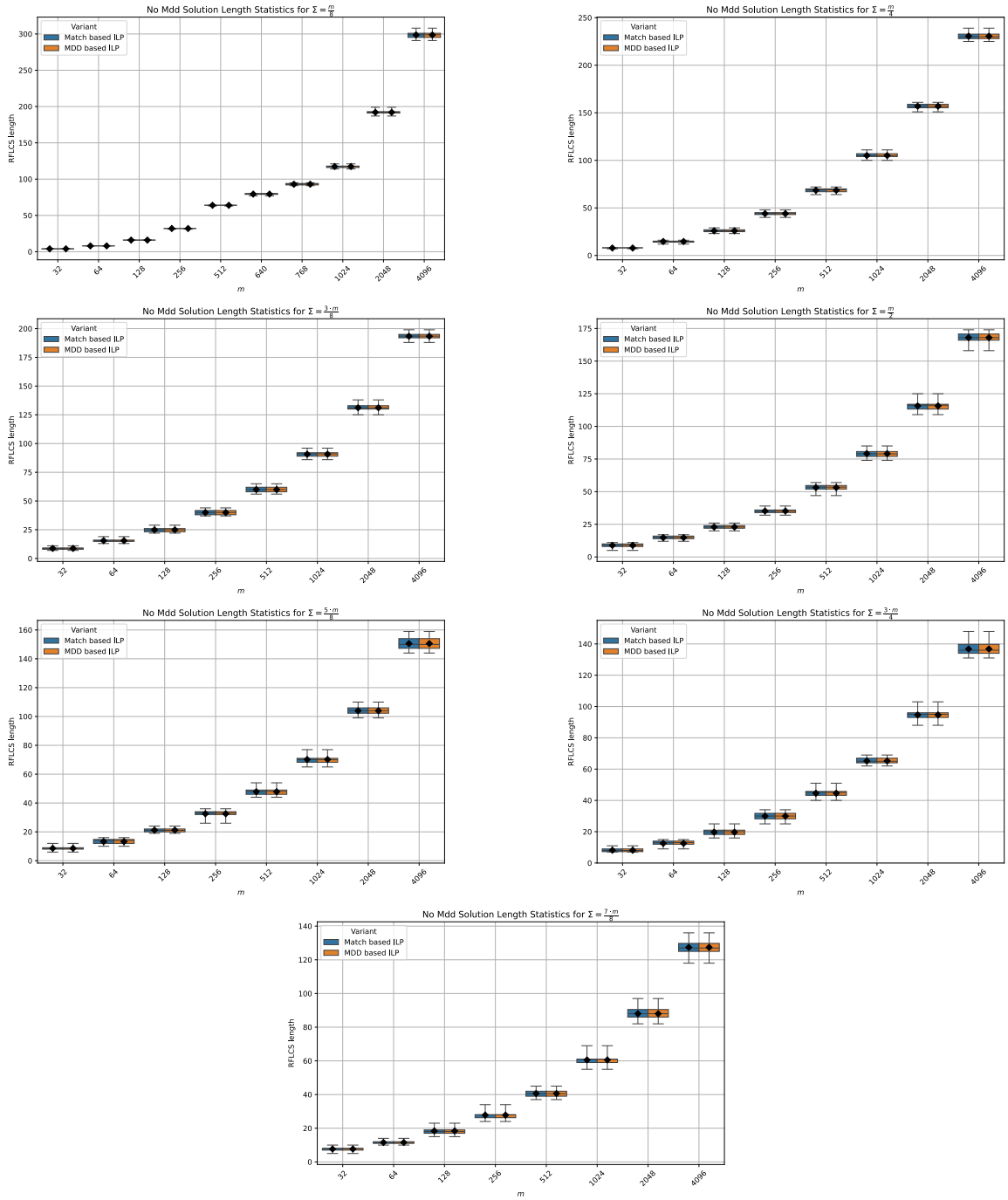


Figure B.8.: Box plots of solution lengths for Instance Set 1 when the *MDD* phase is skipped, comparing the match-based and MDD-based ILP approaches. Each subfigure corresponds to a different relative alphabet size. The y-axis shows solution lengths, and the x-axis shows input string lengths. The box plots indicate the minimum, maximum, average, and the 25% and 75% quantiles across runs.

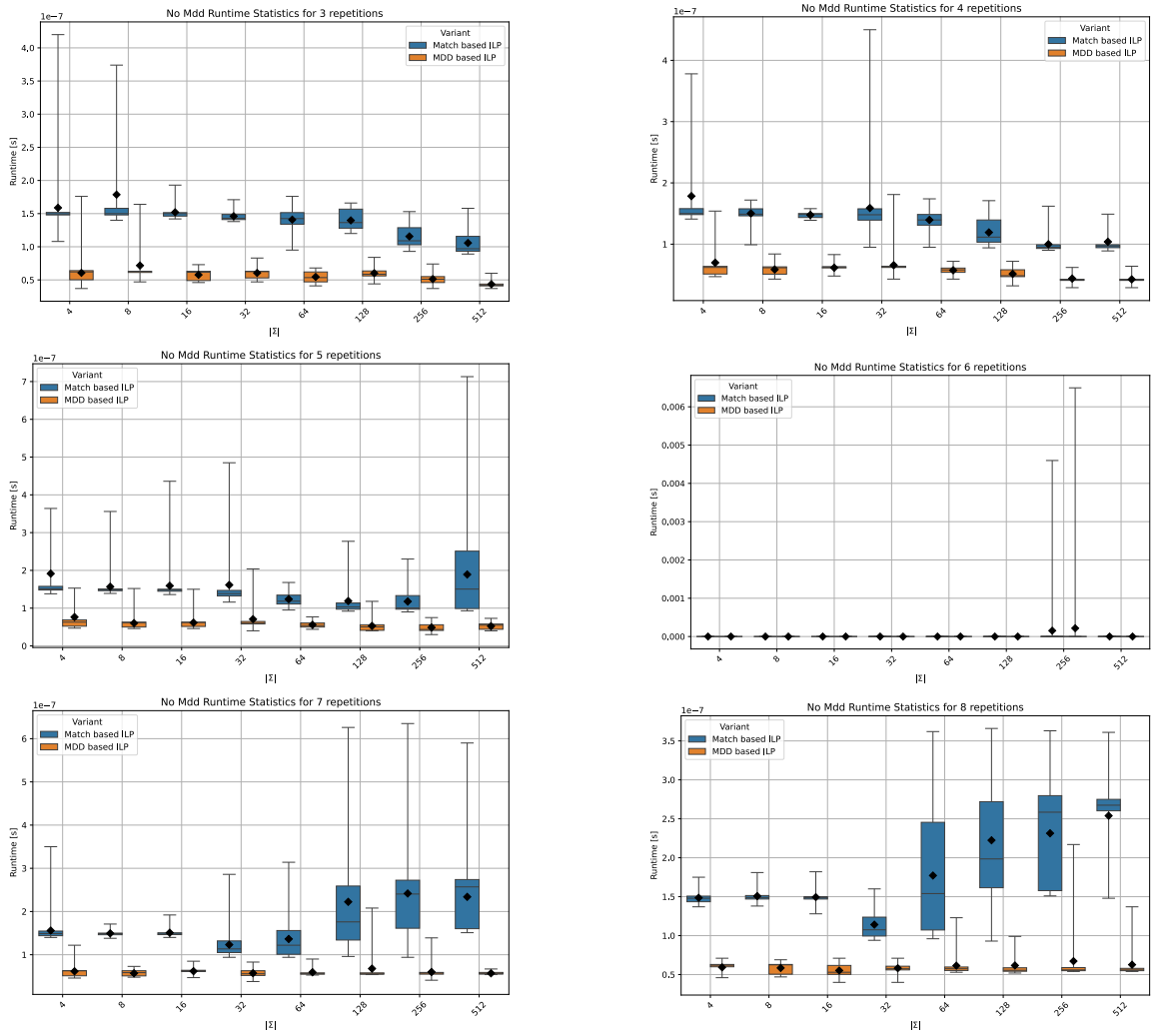


Figure B.9.: Box plots of runtime (in seconds) for Instance Set 2 when the *MDD* phase is skipped, comparing the match-based and MDD-based ILP approaches. Each subfigure corresponds to a different number of character repetitions. The y-axis shows runtime, and the x-axis shows the alphabet size. The box plots indicate the minimum, maximum, average, and the 25% and 75% quantiles across runs.

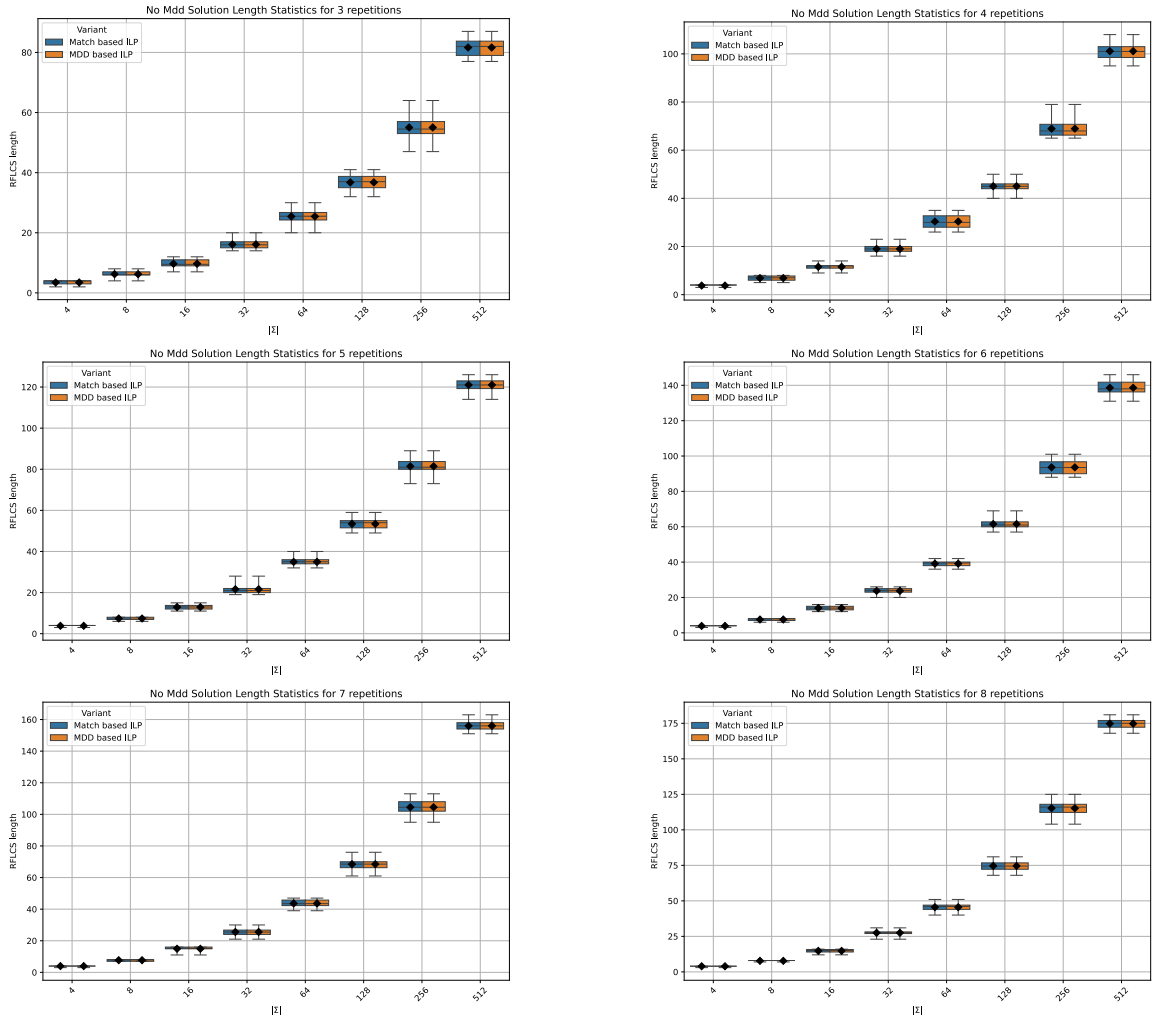


Figure B.10.: Box plots of solution lengths for Instance Set 2 when the *MDD* phase is skipped, comparing the match-based and MDD-based ILP approaches. Each subfigure corresponds to a different number of character repetitions. The y-axis shows solution lengths, and the x-axis shows the alphabet size. The box plots indicate the minimum, maximum, average, and the 25% and 75% quantiles across runs.

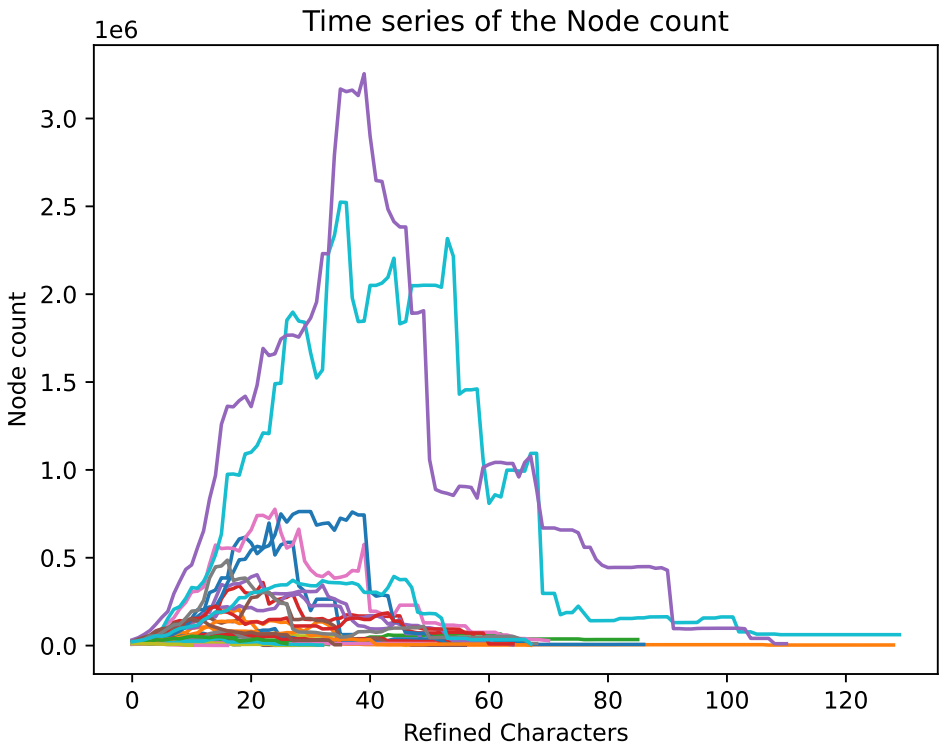


Figure B.11.: MDD node count series for all 30 instances in the complexity category with $\Sigma = 512$ and 8 repetitions. The y-axis shows the number of MDD nodes, and the x-axis indicates the number of characters used to refine the MDD.

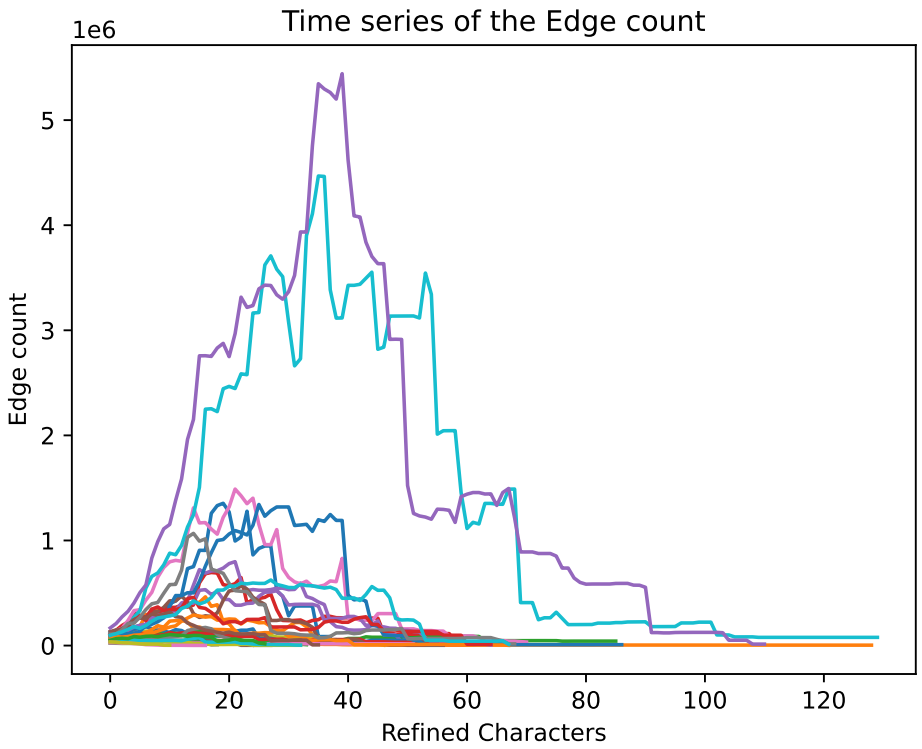


Figure B.12.: MDD edge count series for all 30 instances in the complexity category with $\Sigma = 512$ and 8 repetitions. The y-axis shows the number of edges in the MDD, and the x-axis indicates the number of characters used to refine the MDD.

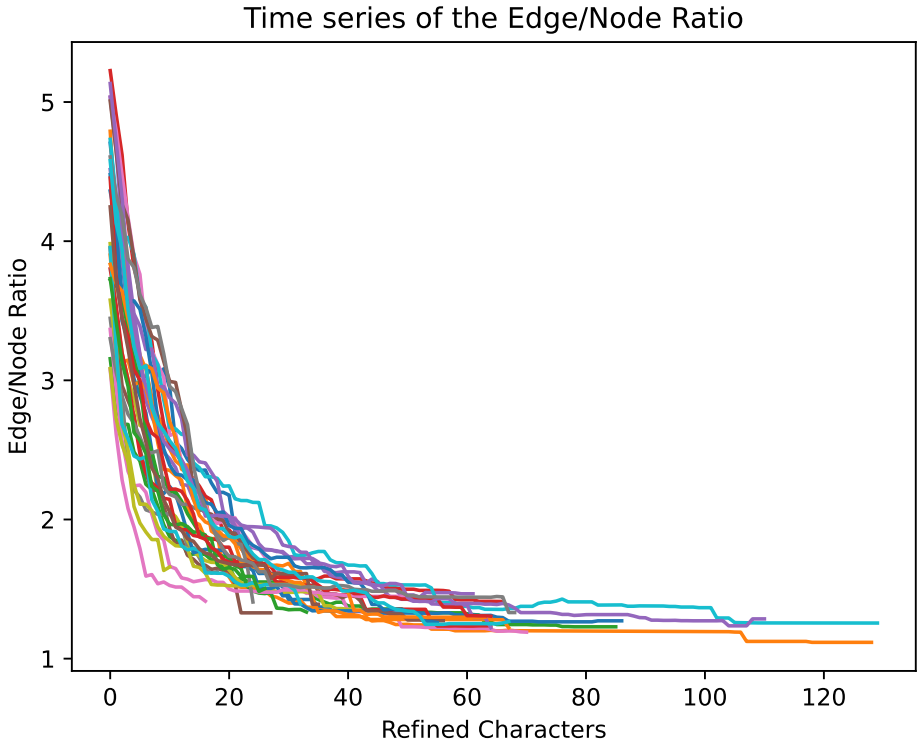


Figure B.13.: Average number of edges per node in the MDD for all 30 instances in the complexity category with $\Sigma = 512$ and 8 repetitions. The y-axis shows the average number of edges per node, and the x-axis indicates the number of characters used to refine the MDD.

C. Proofs

For correctness proofs concerning refinement and pruning, we refer to the original work by Horn et al. [HDBR20]. While most of the newly introduced pruning rules are straightforward, we provide a proof for the pruning rules based on the domination relation, which require a more detailed justification.

Lemma 1. *An edge from node $\eta \in MDD_i$ to node η' can be pruned if*

$$|\{\eta^d \mid \eta^d \in succ \wedge v^d \prec v'\}| > i - |prefix| \quad \vee \quad \exists n^d \in succ : v^d \prec v' \wedge char(v^d) \notin pre,$$

without the algorithm losing optimality.

Proof. **Case** $|\{\eta^d \mid \eta^d \in succ \wedge v^d \prec v'\}| > i - |prefix|$: Assume that there exists a path starting at r that is traversing node η and continuing through node η', which yields a longer solution than any path that continues through a node η^d with $v^d \prec v'$ after η. Since η' is still reachable after any η^d, such a path could only exist if no traversal from r to node η, would permit continuing with any node η^d after η. So every path from the root to η must have had already included nodes corresponding to every character in the set $\Lambda = \{char(v^d) \mid v_d \in succ \wedge v^d \prec v'\}$, and by definition all characters in *prefix*. From $|\{\eta^d \mid \eta^d \in succ \wedge v^d \prec v'\}| = |\Lambda|$, $\Lambda \cap prefix = \emptyset$ and our case condition, we get $|\Lambda| + |prefix| > i$. So these paths would need to traverse more characters than the level on which node η resides in permits, which results in a contradiction.

Case $\exists \eta^d \in succ : v^d \prec v' \wedge char(v^d) \notin pre$: Assume again that there exists a path starting from r through node η that continues with node η', resulting in a longer solution than any path continuing through node η^d with $v^d \prec v'$ after η. As before, since η' remains reachable after η^d, this would imply that there is no valid path that includes η^d directly after η. Therefore, all paths from r to η must have already included a node with character $char(v')$. However, by assumption, $char(v') \notin pre$, resulting in a contradiction. □