Diploma Thesis on

“Algorithms for the Minimum String Cover Problem”

Chris Schwiegelshohn
November 10, 2010

Advisors:
Dipl.-Inform. Tobias Marschall
Prof. Dr. Sven Rahmann

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
## Contents

1 Introduction and Algorithmic Background .......................... 1
   1.1 Introduction ................................................. 1
   1.2 Algorithmic Background .................................... 2
      1.2.1 Strings ............................................... 2
      1.2.2 Abelian Patterns ...................................... 3
      1.2.3 Graphs and Graph Algorithms ......................... 3
      1.2.4 Branch and Bound Method .............................. 4
      1.2.5 Integer Linear Programming ............................. 5
   1.3 Terminology .................................................. 6
   1.4 Definition of the Minimum String Cover Problem (MSC) ........ 7
   1.5 Variations of the MSC Problem ................................ 8
      1.5.1 Minimum Partial String Cover (MPSC) .................. 8
      1.5.2 Minimum String Cover with Forbidden Substrings (MSC/FB) .. 10
      1.5.3 Minimum Abelian Cover MAC ............................ 11
   1.6 Previous Work ............................................... 11
      1.6.1 Cover Problems ....................................... 12
      1.6.2 String Problems ...................................... 12
      1.6.3 MSC Results .......................................... 13

2 Algorithms and Algorithmic Analysis ............................... 15
   2.1 Index Graphs ............................................... 15
      2.1.1 Lower Bound Calculation for MSC ....................... 15
      2.1.2 Lower Bound Calculation for MPSC ..................... 20
      2.1.3 Lower Bound Calculation for MSC/FB and MAC ............ 22
      2.1.4 Branch and Bound Scheme .............................. 23
   2.2 Linear Integer Programming ................................ 27
      2.2.1 IP and MIP Formulations for MSC ...................... 27
      2.2.2 IP Formulation for MPSC ............................... 34
      2.2.3 IP Formulations for MSC/FB and MAC ................... 40
3 Experimental Evaluation
   3.1 Resource Description ........................................ 43
   3.2 Experiment Methodology ..................................... 44
   3.3 Instance Generation ......................................... 45
   3.4 Implementation ............................................. 47
      3.4.1 Brute Force Algorithm ................................. 47
      3.4.2 Existing IP Algorithm ................................. 48
      3.4.3 New IP and MIP Algorithms ......................... 49
      3.4.4 Branch and Bound Algorithm using Index Graphs ...... 49
   3.5 MSC Results .................................................. 49
      3.5.1 Brute Force for MSC .................................... 50
      3.5.2 Index Graphs for MSC .................................. 50
      3.5.3 IP and MIP for MSC .................................... 51
      3.5.4 Result Assessment ...................................... 52
   3.6 MPSC Results ................................................ 54
   3.7 Gene Clustering ............................................... 54

4 Conclusion and Future Work
   4.1 Conclusion ................................................... 57
   4.2 Future Work .................................................. 58

A Additional Information
   A.1 Glossary ..................................................... 62

Bibliography ...................................................... 64

Erklärung ........................................................... 64
Chapter 1

Introduction and Algorithmic Background

1.1 Introduction

Both string and cover problems are seminal topics in computer science and complexity theory. However, there are few instances where an actual problem falls into both problem sets. The Minimum String Cover Problem is such an example. Its input consists of a finite set \( S \) of strings over a symbolic alphabet, and the task is to find a set of substrings generating \( S \) and having a smaller cardinality than the alphabet. At a first glance, the Minimum String Cover Problem strongly resembles the Minimum Set Cover Problem, one of the first problems known to be NP-complete [16]. However, a more thorough analysis reveals that on the one hand, the string cover problem is restricted by the sequential alignment of strings as it does not allow any overlaps of the symbols. On the other hand, its solution space is larger due to the increased number of string permutations compared to subsets. Although the NP-completeness of this problem was already determined in 1990 [17], there has been very little additional work on the subject since then. In particular, we are not aware of any study that addresses the efficient computation of solutions. On the one hand, this little interest may be due to a lack of practical applications for the Minimum String Cover Problem. On the other hand, the existence of efficient algorithms may lead to more applications in practice. In this thesis, we therefore have the goal to develop algorithms with a reasonable computational performance. In addition to a theoretical analysis of these algorithms, a thorough experimental evaluation is required as well.

In the remaining sections of this chapter, we first describe basic definitions and algorithms that are later used in the thesis. Then we introduce our terminology and formally define the Minimum String Cover Problem and some of its variants. Finally, we discuss previous work on the topic and on related problems. Chapter 2 contains the main part of the thesis. There we first introduce a method to determine lower bounds for the optimal
solution. This method is the key element of a branch and bound scheme for the Minimum String Cover Problem. Afterwards, we propose an alternative approach using integer linear programming. We prove the correctness of our approaches and their properties. In Chapter 3, we provide an experimental analysis and evaluation of our approaches presented in Chapter 2 as well as two reference algorithms. This analysis contains the methodology of the experiments, the generation of the test cases, some implementation details, and the experimental results. Finally, we give a brief conclusion and mention some potential applications for the algorithms.

1.2 Algorithmic Background

1.2.1 Strings

String is an umbrella term for sequences of symbols. To formally approach strings, we need an alphabet typically denoted by $\Sigma$ and the grammar of the language or the set of strings that we accept. However, not every grammar can express every property of a string set. In this thesis, we only consider finite languages, that is, sets of strings with a finite number of elements, which can be modeled with regular expressions as defined in Hopcroft and Ullman [9]:

A regular expression over an alphabet $\Sigma$ is either one of the following constant symbols:

- the empty set $\emptyset$,
- the empty string $\epsilon$,
- a single symbol of $\Sigma$

or a combination of regular expressions $\alpha$ and $\beta$ using the following operators:

- the concatenation $\alpha\beta$
- the alteration $\alpha + \beta$
- the Kleene star $\alpha^*$

We will not go further into the semantics of regular expressions and instead explain their meaning informally. A regular expression $r$ describes a possible infinite set of strings that can be constructed by applying the mentioned operations on subexpressions of $r$. The concatenation of $\alpha$ and $\beta$ describes all strings that consist of exactly one string of $\alpha$ followed by exactly one string of $\beta$. The alteration of two regular expressions merges the languages, that is, all strings represented by $\alpha$ and all strings represented by $\beta$. For example $r = (AB + BA)B$ only represents the strings $ABB$ and $BAB$. The Kleene star represents all strings that can be constructed by concatenating strings represented by $\alpha$ an
arbitrary number of times. The regular expression corresponding to all possible strings over an alphabet \( \Sigma = \{ A, B, C, \ldots \} \) is often written as \( \Sigma^* \) as opposed to \( (A + B + C + \ldots)^* \). Note that any finite set of strings \( S = \{ s_1, s_2, \ldots, s_{|S|} \} \) can be represented by the regular expression \( r = (s_1 + s_2 + \ldots + s_{|S|}) \).

Let \( s = \alpha_1 \alpha_2 \ldots \alpha_n \) and \( s' = \alpha'_1 \alpha'_2 \ldots \alpha'_m \) be two strings with \( n \geq m \) consisting of symbols \( \alpha_i \) and \( \alpha'_i \), respectively. We say that \( s' \) is a prefix of \( s \) if and only if \( \alpha'_i = \alpha_i \) for all \( i \in \{ 1, \ldots, m \} \). Analogously, \( s' \) is a suffix of \( s \) if and only if \( \alpha'_{m-i+1} = \alpha_{n-i+1} \) for all \( i \in \{ 1, \ldots, m \} \). A substring is a prefix of a suffix. We say that \( s \) is a superstring of strings \( s' \) and \( s'' \) if and only if \( s' \) and \( s'' \) are both substrings of \( s \).

Assume a string \( s \). If we delete symbols of \( s \) then we obtain a subsequence \( s' \). Therefore, every substring is also a subsequence. Similarly, \( s \) is a supersequence of \( s' \) and \( s'' \) if and only if \( s' \) and \( s'' \) are both subsequences of \( s \).

### 1.2.2 Abelian Patterns

While the order of symbols defines a string, an Abelian pattern only counts the number of symbols in a string pattern. Formally, given an alphabet \( \Sigma \), an Abelian pattern is a sum \( a := \sum_{c \in \Sigma} m_c \cdot c \) with a nonnegative integer \( m_c \) [19]. The factor \( m_c \) corresponds to the number of occurrences of a symbol \( c \). We say that a string \( s \) matches an Abelian pattern \( a \), if the number of occurrences of symbol \( c \) in \( s \) is equal to \( m_c \) for all \( c \in \Sigma \). For example, the four strings \( ABBB \), \( BABB \), \( BBAB \), and \( BBBA \) all match the same Abelian pattern \( 1A + 3B \).

### 1.2.3 Graphs and Graph Algorithms

The definitions in this section are based on Jungnickel [13].

A directed graph \( G, (V, E) \) contains a non-empty set \( V \) of nodes and a set \( E \subseteq V^2 \) of edges. An edge \( e = (u, v) \in E \) starts at node \( u \) and ends at node \( v \). Then, we say that \( u \in V \) and \( v \in V \) are interconnected or adjacent. Further, both nodes are incident with edge \( e \). \( E \) is not necessarily symmetrical, that is, \( (u, v) \in E \) does not mean that \( (v, u) \in E \) as well. A weighted directed graph \( G \) is a triple of \( (V, E, f) \) with \( G(V, E) \) representing a directed graph and a weight function \( f : E \rightarrow \mathbb{R} \).

A path \( b \) from node \( v \in V \) to node \( u \in V \) in a weighted directed graph \( G(V, E, f) \) is a sequence of edges \( e_1, e_2, \ldots, e_n \) for some positive integer \( n \) such that edge \( e_i \) ends at the same node at which edge \( e_{i+1} \) starts. Furthermore, edge \( e_1 \) starts at node \( v \) and edge \( e_n \) ends at node \( u \). The weighted length of path \( b \) is the sum of its edge weights, that is

\[
f(b) = \sum_{i=1}^{n} f(e_i) .
\]

If there exists a path in \( G(V, E, f) \) from node \( v \in V \) to node \( u \in V \), we say that \( u \) is reachable from \( v \).
A path starting and ending with the same node is called a cycle. A graph without any cycles is called acyclic. In the context of this thesis, we only consider directed acyclic graphs also called DAGs. To determine shortest paths in a DAG, we use a topological sorting of the nodes. A topological sorting of \( V \) in \( G(V, E) \) exists if and only if we can assign every node \( v_i \in V \) a unique number \( i \in \{1, \ldots, |V|\} \) such that

\[
e = (v_i, v_j) \in E \Rightarrow i < j \ \forall e \in E.
\]

As there are no cycles in a DAG, there exists a topological sorting of the nodes. It can be determined in \( O(|V| + |E|) \). Given a topological sorting, we can calculate the shortest and the longest paths from node \( v \) to all nodes reachable from \( v \) in \( O(|V| + |E|) \). To this end, we can apply the Breadth First Search algorithm (BFS). We provide an exemplary pseudo code for BFS, see Algorithm 1, as BFS is central to one of our algorithms.

<table>
<thead>
<tr>
<th>Data:</th>
<th>DAG ( G(V, E, f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a starting node ( s )</td>
<td>a topological sorting ( R ) of all nodes reachable from ( s ) with ( s = R(1) )</td>
</tr>
</tbody>
</table>

**Result:** The shortest paths to all nodes reachable from \( s \)

1. begin
2. forall \( v \in V \setminus \{s\} \) do
3. \( f(b(s, v)) = \infty \)
4. \( f(b(s, s)) = 0 \)
5. for \( i = 1; i \leq |R|; i = i + 1 \) do
6. node \( v = R(i) \)
7. forall edges \( e = (v, u) \) do
8. if \( f(b(s, v)) + f(e) < f(b(s, u)) \) then
9. \( f(b(s, u)) = f(b(s, v)) + f(e) \)
10. end

**Algorithm 1:** BFS - We determine the shortest paths from starting node \( s \) to all reachable nodes.

A directed tree is a DAG \( G(V, E) \) with \( |E| = |V| - 1 \). A node \( v \in V \) is a leaf, if it is not starting node of any edge. If all nodes in the tree are reachable from node \( v \) then \( v \) is called the root. Note that there must not necessarily exist such a node in a every given directed tree.

1.2.4 Branch and Bound Method

For problems with high computational time complexity, we can decide to apply a heuristic that has an acceptable computational effort on average but cannot always produce an
optimal solution within a reasonable time. Approximation algorithms are specific heuristics with polynomial time complexity. They always produce a solution that is at most a factor \(a\) away from the optimum. The factor \(a\) is called the approximation factor. Alternatively, there are methods that will find an optimal solution by examining all possible solutions if required. For many input data, these methods are nevertheless able to complete within an acceptable time frame. The branch and bound scheme belongs to the latter category. It is based on a directed tree data structure. The leaves of the tree represent all possible solutions. For minimization problems, a method is required to determine a lower bound for all candidate solutions in a given subtree. If this lower bound for such a subtree is larger or equal to an already found candidate solution then the whole subtree can be ignored. It is the goal of the method to quickly eliminate large parts of the branching tree such that only few candidate solutions must be evaluated.

### 1.2.5 Integer Linear Programming

The definitions in this section are based on Nemhauser and Wolsey [7]. A linear mixed integer program \(MIP\)

\[
\text{max}\{c \cdot X + h \cdot Y : A \cdot X + G \cdot Y \leq b, X \in \mathbb{Z}_+^n, Y \in \mathbb{R}_+^p\}
\]

consists of an \(n\) dimensional vector \(X\) of nonnegative integer variables and an \(p\) dimensional vector \(Y\) of nonnegative real variables, an \(n\) dimensional vector \(c\) and an \(p\) dimensional vector \(h\), both consisting of real numbers, an \(m \times n\) matrix \(A\), an \(m \times p\) matrix \(G\) and a \(m\) dimensional vector of real numbers \(b\). If \(p = 0\), the problem is called a linear (purely) integer program \(IP\) and analogously, if \(n = 0\), the problem is called a linear program \(LP\). We call \(\text{max}\{c \cdot X + h \cdot Y\}\) the optimization function whereas \(A \cdot X + G \cdot Y \leq b\) are the constraints. In practice, we explicitly state the meaning of every constraint via an inequality rather than to use the matrix representation \(A \cdot X + G \cdot Y \leq b\).

MIPs are called feasible if there exists a variable assignment for \(X\) and \(Y\) such that \(X\) and \(Y\) satisfy the constraints of the MIP and \(\text{max}\{c \cdot X + h \cdot Y\}\) is greater or equal than the value of the optimization function for any other variable assignment that satisfies the constraints of the MIP. Such an assignment of \(X\) and \(Y\) is an optimal solution. If there is no assignment such that no constraint is violated then we say that the MIP is infeasible. If the MIP is neither infeasible nor feasible then it is called unbounded.

Often, combinatorial problems can be modeled with binary variables, that is \(X \in \{0,1\}^n\). Typically, \(x = 1\) means that the underlying statement of variable \(x\) is true. If we wish to solve minimization problems then we simply multiply the elements of \(c\) and \(h\) with -1. We can model greater or equal inequalities similarly. When presenting IPs in this thesis we use the most compact and understandable formulation.

Linear mixed integer programming and integer programming are both NP-complete [4] [14] [7], whereas every linear program is solvable in polynomial time [7]. Nevertheless,
MIPs have been studied intensively in the past and there are many efficient algorithms available to solve them. Therefore, it is common to solve an NP-complete problem by modeling it as an MIP and applying a commercial solver to find a solution. There are two common methods to solve a MIP.

**Branch and Bound:** The set of all assignments of the variables of a MIP: \( \max\{c \cdot X + h \cdot Y : A \cdot X + G \cdot Y \leq b, X \in \mathbb{Z}_+^n, Y \in \mathbb{R}_+^p\} \) is included in the set of all assignments of the variables of the LP: \( \max\{c \cdot X + h \cdot Y : A \cdot X + G \cdot Y \leq b, X \in \mathbb{Z}_+^n, Y \in \mathbb{R}_+^p\} \). Therefore, a branch and bound approach for MIPs consists of solving the corresponding LP, selecting a binary variable and fixing its value for all nodes of the subtree.

**Cutting Plane Algorithm:** Given a MIP: \( \max\{c \cdot X + h \cdot Y : A \cdot X + G \cdot Y \leq b, X \in \mathbb{Z}_+^n, Y \in \mathbb{R}_+^p\} \), we do not immediately attempt to find a solution that satisfies all constraints. Instead, we determine an optimum of a relaxed LP problem and remove the non integer solution with the help of a cut.

### 1.3 Terminology

Our input always consists of a set of strings \( S \) over a symbolic alphabet \( \Sigma \). Every string of \( S \) is denoted by \( s_i \) for a counting index \( i \in \{1, \ldots, |S|\} \). String \( s_i \) contains \( |s_i| \) symbols and a pair \( (i, p) \) with \( i \in \{1, \ldots, |S|\} \) and \( p \in \{1, \ldots, |s_i|\} \) denotes position \( p \) of string \( s_i \). A triple \( (i, p, q) \) with \( i \in \{1, \ldots, |S|\} \) and position indexes \( p, q \in \{1, \ldots, |s_i|\} \) with \( p \leq q \) is an occurrence and represents positions \( p \) to \( q \) of string \( s_i \). The set of all occurrences in \( S \) is denoted by \( C(S) \). It is easy to see that \( |C(S)| = \sum_{i=1}^{|S|} |s_i| \cdot (|s_i| + 1)/2 \). We denote by \( s_i(p, q) \in \Sigma^* \) with \( i \in \{1, \ldots, |S|\} \), position indexes \( p, q \in \{1, \ldots, |s_i|\} \), and \( p \leq q \) the substring that occurs at positions \( p \) to \( q \) in string \( s_i \). \( T(S) \subseteq \Sigma^* \) is the set of all substrings in \( S \). Note that every string of \( S \) also belongs to \( T(S) \) and hence \( S \subseteq T(S) \). We use \( t_j \) with \( j \in \{1, \ldots, |T(S)|\} \) to denote a specific substring in \( S \). Informally, an occurrence is one instance of a substring in \( S \).

Furthermore, we assign to every occurrence \((i, p, q)\) the corresponding substring \( s_i(p, q) \) with \( i \in \{1, \ldots, |S|\} \), \( p, q \in \{1, \ldots, |s_i|\} \), and \( p \leq q \). Therefore, we can define the set \( C_j = \{(i, p, q) \in C(S) \mid s_i(p, q) = t_j\} \) with \( j \in \{1, \ldots, |T(S)|\} \), that is, \( C_j \) contains all occurrences that are an instance of substring \( t_j \).

We say that two occurrences \((i, p, q)\) and \((i', p', q')\) overlap if \( i = i', p \leq q', \) and \( p' \leq q \). Similarly, occurrence \((i, p, q)\) contains position \((i', p')\) if \( i = i' \) and \( p \leq p' \leq q \), see Figure 1.1.
1.4. Definition of the Minimum String Cover Problem (MSC)

1 Definition. A set of occurrences \( D \subseteq C(S) \) is a factorization of string \( s_i \in S \) with \( i \in \{1, \ldots, |S|\} \) if we can order the elements of \( D = \{(i, p_0, q_0), (i, p_1, q_1), \ldots, (i, p_n, q_n)\} \) such that \( p_0 = 1, q_n = |s_i| \) and \( p_k+1 = q_k + 1 \) for \( k \in \{0, n-1\} \).

Note that a factorization \( D \) of \( s_i \) only contains occurrences with the associated string index \( i \in \{1, \ldots, |S|\} \). For example, consider \( S = \{s_1 = ABCDAE, s_2 = ADC\} \). The set of occurrences \( \{(1,1,2),(1,3,5),(1,6)\} \) is a factorization of \( s_1 \), whereas the set of occurrences \( \{(1,1,2),(1,3,5),(1,6),(2,1,2)\} \) is not a factorization as \( (2,1,2) \) occurs in string \( s_2 \). Similarly, the sets \( \{(1,1,1),(1,3,6)\} \) and \( \{(1,1,4),(1,4,6)\} \) are no factorizations either as position \( (1,2) \) is not contained in any occurrence of the first set and position \( (1,4) \) is contained in two occurrences of the second set. This example is illustrated in Figure 1.2.

\[
\begin{array}{cccccccc}
S_1 & A & B & C & D & A & B & E & C \\
\text{Positions} & (1,1) & (1,2) & (1,3) & (1,4) & (1,5) & (1,6) & (1,7) & (1,8) \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Occurrences} & (1,1,2) & (1,5,6) & (1,3,5) & (1,4,6) \\
\end{array}
\]

Figure 1.1: The first eight symbols of string \( s_1 \). Occurrences \((1,1,2)\) and \((1,5,6)\) both correspond to substring AB. Occurrence \((1,3,5)\) and \((1,4,6)\) overlap as both contain position \((1,4)\).

2 Corollary. A set of occurrences \( D \) contains a factorization of a string \( s \) if every position of \( s \) is contained in exactly one occurrence of \( D \).
**Chapter 1. Introduction and Algorithmic Background**

**Proof** The proof follows directly from Definition 1.

---

**3 Definition.** Let $S$ be a set of strings. Then $M \subseteq T(S)$ is a cover of $S$ (or $M$ covers $S$) if and only if the set of occurrences $D = \{(i, p, q) \in C(S) \mid s_i(p, q) \in M\}$ contains a factorization for every $s \in S$. If every position of every string $s \in S$ is contained in exactly one occurrence of $D$, we say that $D$ is a non-redundant factorization set of $S$.

Let $S = \{ABCA, CAAB\}$ be a set of strings. Then $\{A, B, C\}$ covers $S$ while $\{ABC, A\}$ does not cover $S$.

We are now able to provide a formal definition of the Minimum String Cover Problem.

**4 Definition.** Let $S$ be a set of strings with a weight function $w : T(S) \mapsto \mathbb{R}_0^+$. The Minimum String Cover Problem (MSC) consists of finding a set of strings $M \subseteq T(S)$ such that the following two conditions are met:

1. $M$ covers $S$,
2. $\sum_{t \in M} w(t)$ is minimal.

An optimal solution of the MSC for $(S, w)$ is called a minimum string cover. The most basic and also classic version of the problem uses unit weights for all substrings, i.e. $w(t) = 1$ for all $t \in T(S)$. In this case, the minimum string cover is a cover of $S$ with minimal cardinality. We have based all experiments in Chapter 3 on this variant, partly because it is the most intuitive one but also because it is the easiest to visualize. Therefore, $w$ is implicitly assumed to be the unit weight function unless specified otherwise.

For example, $\{AB, C\}$ is the only minimum string cover of $\{ABC, CAB, CC\}$ while $\{A, B, C\}, \{AB, C, ACB\}$, or $\{ABC, CAB, ACB\}$ are each minimum string covers for the set of strings $\{ABC, CAB, ACB\}$. For non-unit weight functions, the optimal solutions may be different. Given the weight function $w(t) = |t|$, $\{A, B, C\}$ and $\{AB, C\}$ are the minimum string covers for the first set of strings while $\{A, B, C\}$ is the minimum string cover for the second set of strings, see Figure 1.1.

---

**1.5 Variations of the MSC Problem**

**1.5.1 Minimum Partial String Cover (MPSC)**

Before introducing our new problem, we expand our definitions on factorizations (see Definition 1) and covers (see Definition 3).

**5 Definition.** A set of occurrences $D \subseteq C(S)$ is a partial factorization of string $s_i \in S$ with $i \in \{1, \ldots, |S|\}$ if we can order the elements of $D = \{(i, p_0, q_0), (i, p_1, q_1), \ldots, (i, p_n, q_n)\}$ such that $p_0 \geq 1$, $q_n \leq |s_i|$, and $p_{k+1} > q_k + 1$ for $k \in \{0, n - 1\}$.
1.5. VARIATIONS OF THE MSC PROBLEM

Figure 1.3: The optimal covers $M$ for the instance $S = \{ABC, CAB, CC\}$ with unit weight function and length-proportional weight function $w(t) = |t|$. The colors show how the strings of $S$ can be factorized by substrings appearing in the minimum string cover.

Note that not all positions of $s_i$ must be contained in an occurrence of a partial factorization of $s_i$. This leads to the definition of gap.

6 Definition. Let $D$ be a partial factorization of a string $s$. Every position that is not contained in an occurrence of $D$ is called a gap.

The number of gaps in a partial factorization $D$ of a string $s$ is called $g(D)$. Furthermore, we can state a corollary similar to Corollary 2.

7 Corollary. A set of occurrences $D$ contains a partial factorization of a string $s$ if every position of $s$ is contained in at most one occurrence of $D$.

Proof The proof follows directly from Definition 5.

The occurrence set $D = \{(1, 2, 3)\}$ is a partial factorization of $s_1 = ABCA$ with two gaps, while $D' = \{(2, 1, 2), (2, 3, 3)\}$ is a factorization of $s_2 = CAD$ and therefore a partial factorization with zero gaps.

8 Definition. Let $S$ be a set of strings. Then $M \subseteq T(S)$ is a partial cover of $S$ with at most $g$ gaps if and only if there exists a set of occurrences $D = \{(i, p, q) \in C(S) \mid s_i(p, q) \in M\}$ such that $D$ contains a partial factorization $D_i$ for every string $s_i \in S$ with

$$\sum_{i=1}^{|S|} g(D_i) \leq g.$$

If every position of every string $s \in S$ is contained in at most one occurrence of $D$ then we say that $D$ is a non redundant partial factorization set of $S$. 
CHAPTER 1. INTRODUCTION AND ALGORITHMIC BACKGROUND

Figure 1.4: Effects of gaps on minimal string covers: Any minimum string cover of \{ABC, AB\} has a cardinality two but a partial minimum string cover with one gap has cardinality one. The cardinality of a minimum partial cover for \{ACB, AB\} does not change from the cardinality of the minimum string cover when allowing one gap. However, introducing the gap now results in multiple optimal solutions.

Next, we define our new problem:

9 Definition. Let \(S\) be a set of strings with a weight function \(w : T(S) \rightarrow \mathbb{R}_0^+\) and a nonnegative integer \(g\). The Minimum Partial String Cover Problem (MPSC) consists of finding a set of strings \(M \subseteq T(S)\) such that the following two conditions are met:

1. \(M\) is partial cover of \(S\) with at most \(g\) gaps,
2. \(\sum_{t \in M} w(t)\) is minimal.

We implicitly assume \(g = 0\) if the maximum number of gaps \(g\) is not specified. Then the MPSC degenerates to the standard MSC. An optimal solution of the MPSC is a minimum partial cover.

For example, \{AB\} is a minimum partial cover of \{ABC, AB\} with \(g = 1\), as only position \((1, 3)\) is a gap. Next consider the problem \{ACB, AB\} with \(g = 1\). It is easy to see that \{AB\} is now a partial cover with three gaps. In this case, there is no minimum partial cover of cardinality one, see Figure 1.4.

A slight modification of this problem specifies the maximum number of gaps for each string instead of/in addition to a global restriction in the number of gaps. Here, there exists a maximum number of gaps \(g_i\) for string \(s_i\) with \(i\{1, \ldots, |S|\}\). Then the set of occurrences \(D = \{(i, p, q) \in C(S) \mid s_i(p, q) \in M\}\) for a minimum partial cover must contain a factorization \(D_i\) such that \(g(D_i) < g_i\).

1.5.2 Minimum String Cover with Forbidden Substrings (MSC/FB)

We can also forbid certain substrings from appearing in our cover. For instance, the elements of the optimal solution can have a minimum and a maximum length. Other substrings can be disregarded for application-specific reasons. Below we give a formal definition of this problem.
10 Definition. Let \( S \) be a set of strings with a weight function \( w : T(S) \mapsto \mathbb{R}_0^+ \) and let \( F \subset T(S) \) be a set of substrings of \( S \). The Minimum String Cover Problem with Forbidden Substrings (MSC/FB) consists of finding a set of strings \( M \subseteq T(S) \setminus F \) such that the following two conditions are met:

1. \( M \) covers \( S \),
2. \( \sum_{t \in M} w(t) \) is minimal.

The definition of the Partial Minimum String Cover Problem with Forbidden Strings can be formulated analogously. Consider \( S = \{ABC, CAB, CC\} \) and \( F = \{A, B, C\} \subset T(S) \). \( \{AB, C\} \) is the minimum string cover of \( S \) while \( S \) is the only solution for the MSC/FS.

1.5.3 Minimum Abelian Cover MAC

We denote the Abelian pattern matching with the string \( s \) by \( a(s) \). For example, if given the string \( ABCAB \) then \( a(ABCAB) = 2A + 2B + C \). Let \( A(\Sigma) \) be the set of all Abelian patterns of an alphabet \( \Sigma \). Assume \( S \subset \Sigma^* \) to be a set of strings. We say that \( A(T) = \{a(t) \in A(\Sigma) \mid t \in T\} \) is the Abelian pattern set of a set of substrings \( T \subseteq T(S) \). We assign to every occurrence \((i, p, q)\) appearing in a set of strings \( S \) the matching Abelian pattern \( a(s_i(p, q)) \).

11 Definition. Let \( S \) be a set of strings. Then \( A \subseteq A(T(S)) \) is an Abelian cover of \( S \) if and only if there exists a set of occurrences \( D = \{(i, p, q) \in C(S) \mid a(s_i(p, q)) \in A\} \) such that \( D \) contains a factorization \( D_i \) for every string \( s_i \in S \).

12 Definition. Let \( S \) be a set of strings with a weight function \( w : A(T(S)) \mapsto \mathbb{R}_0^+ \). The Minimum Abelian Cover Problem (MAC) consists of finding a set of Abelian patterns \( A \subseteq A(T(S)) \) such that the following two conditions are met:

1. \( A \) is an Abelian cover of \( S \),
2. \( \sum_{a \in A} w(a) \) is minimal.

Again, variants of MAC to include gaps, forbidden substrings, or forbidden Abelian patterns can be formulated analogously. Let \( \{ABCA, CAB, AAA\} \) be a set of strings. The minimum Abelian cover consists of \( \{1A + 1B + 1C, 1A\} \), see Figure 1.5.

1.6 Previous Work

Previous work on the MSC has been comparatively sparse. In this section, we document which results have been achieved so far. Additionally, there are also a number of problems that share several characteristics with the MSC. These problems are introduced as well.
1.6.1 Cover Problems

Two of the most well known cover problems are the Minimum Set Cover and the Minimum Vertex Cover Problems, both of which are NP-complete [16] [14]. Especially the latter is also very important to string problems, as we explain below. Given a set of elements \( U \) and a collection \( V \) of subsets of \( U \), the Minimum Set Cover Problem consists of finding a collection of subsets \( W \subseteq V \), such that

- \( u \in U \Rightarrow \exists W' \in W \mid u \in W' \)
- \( |W| \) is minimal.

For example, given the set \( U = \{A, B, C\} \) and the collections \( V = \{\{A, B\}, \{A\}, \{A, C\}\} \), the optimal solution is \( W = \{\{A, B\}, \{A, C\}\} \).

Given a graph \( G(V, E) \), the Minimum Vertex Cover Problem consists of finding a set of nodes \( V' \subset V \) such that

- \( e \in E \Rightarrow \exists v \in V' \mid e \) is incident with \( v \)
- \( |V'| \) is minimal.

We omit a written example for this problem as it is easier to understand visually, see Figure 1.6.

Generally, cover problems consist of a set \( S \), a collection of targeting structures \( T \) and the objective to include all elements of \( S \) with a subset of \( T \) with minimal costs. Minimum Set Cover and Minimum Vertex Cover both allow overlaps of the target structure. For example, an edge \( e \) can be incident with two elements of a vertex cover \( V' \), and an element of \( U \) can be contained in multiple elements of the set cover \( W \). Our string cover problem differs from Set Cover in not allowing overlaps, see Definition 1.

1.6.2 String Problems

String problems are, as the name already indicates, a loose set of combinatorial problems dealing with strings in one way or the other. A very simple example is the Longest Common Substring Problem. Given a set of strings \( S \) it consists of finding a string \( t \) of maximum
1.6. PREVIOUS WORK

Figure 1.6: Minimum Vertex Cover Problem - The set of nodes \{B, D, E\} is a vertex cover as every edge is incident with a red node.

length such that \( t \) is a substring of all \( s \in S \). This problem is in P [21] and there are a number of algorithms and data structures available to approach it. Further examples are the **Longest Common Subsequence Problem** and **Shortest Common Supersequence Problem**, which Maier proved to be NP-complete via reduction from Minimum Vertex Cover [15]. The **Shortest Common Superstring** is NP-complete as well [6] which can easily be proven via transformation from the **Hamiltonian Circuit Problem**. Once again this also underlines the similarity between string and cover problems as the original proof of the NP-completeness of the Hamiltonian Circuit Problem is also based on Vertex Cover [14].

We already introduced regular expressions. Similar in intent to the MSC, Angluin showed that the **Minimum Inferred Regular Expression Problem** is NP-complete [2] [16].

Given two not necessarily finite sets of strings \( S \) and \( T \) over an alphabet \( \Sigma \), the Minimum Inferred Regular Expression Problem consists of finding a regular expression \( r \) and corresponding language \( L \) with a minimum number of symbols from \( \Sigma \) such that \( S \subseteq L \) and \( T \subseteq \Sigma^* \setminus L \).

1.6.3 MSC Results

The Minimum String Cover Problem is the first string problem to combine string and cover problems. The earliest paper published on the subject was written by Néraud in 1990 [17]. It may be a bit surprising that the unique merging of these two problem types has not been addressed much earlier as for many string problems, the NP-completeness was proven by transforming a cover problem into an instance of the string problem in question. Néraud's paper deals with the elementariness of a set of strings. A set of strings \( X \) is defined to be elementary if there exists no set of strings \( Y \) with \( |Y| < |X| \) and \( X \in Y^* \).

For example, \( X' = \{ABC, BCA\} \) is elementary and \( X'' = \{ABC, BCA, A\} \) is not due to \( Y = \{A, BC\} \) and \( X'' \in Y^* \). Néraud proves the decision problem whether a given set of strings is elementary to be co-NP-complete. Consequently, the MSC is NP-complete for
unit weights. Further results were published by Hermelin et. al. [8]. It is the main result of their paper that the approximation of the problem beyond a certain bound is already NP-complete as well. Our own definition of the MSC, see Section 4, is based heavily on the paper by Hermelin et. al., though their paper defined factorizations differently. In addition, the paper introduces a few approximation algorithms whose factors did not match the proved lower bound. Other than that, we only found a paper from 1995 by Bodlaender et. al. [3] in which it was mistakenly stated that the MSC (which they termed Dictionary Generation) is an unsolved problem of interest in computational biology [3]. We are not aware of any other research on the MSC, and this thesis is the first attempt at developing practically applicable algorithms to solve the problem.
Chapter 2

Algorithms and Algorithmic Analysis

In this chapter, we present two algorithms for the MSC. The first method determines lower bounds for the sum of all weights of strings appearing in a minimum string cover $M$ based on a graph representation of strings an BFS. The other method solves the problem by using a linear integer program.

2.1 Index Graphs

2.1.1 Lower Bound Calculation for MSC

This section describes a graph based method to find lower bounds for $w(M) := \sum_{t \in M} w(t)$. Before focusing on the bounds and the optimization scheme building upon it, we first define a data structure that allows us to convert strings into a manageable graph based form.

13 Definition. Let $S$ be a set of strings. An index graph $G_i(V_i, E_i)$ of a string $s_i$ with $i \in \{1, \ldots, |S|\}$ and $V_i = \{(i, 0), (i, 1), \ldots, (i, |s_i|)\}$ has a directed edge $e$ from node $(i, p-1)$ to node $(i, q)$ for every occurrence $(i, p, q)$. This edge $e$ is annotated with string $s_i(p, q)$.

There are $|S|$ index graphs for a set of strings $S$. Informally, every position of a string in $S$ is a node. Additionally, there is a start node for every string. We use the notation $E(j)$ to describe the set of all edges annotated with substring $t_j \in T(S)$ for some $j \in \{1, \ldots, |T(S)|\}$. Note that every index graph is a DAG as $p \leq q$ for all occurrences $(i, p, q)$. Then arranging the nodes according to ascending position indexes produces a unique topological order.

14 Corollary. Let $s_i$ be a string of $S$ with $i \in \{1, \ldots, |S|\}$. There exists a bijective mapping between the factorizations of $s_i$ and the paths from node $(i, 0)$ to node $(i, |s_i|)$ in the corresponding index graph $G_i$.

Proof The claim follows directly from Definitions 1 and 13. 

□
To determine a lower bound for $w(M)$, we introduce a weight function for the edges of the index graphs such that

- the weight of edge $e$ considers the weight of its annotated substring, and
- it considers the frequency of its annotated substring in $S$.

Let $t_j \in T(S)$ with $j \in \{1, \ldots, |T(S)|\}$ be a substring of $S$. Then its frequency is equal to the number of its occurrences $|C_j| = |E(j)|$, see Section 1.3.

15 Definition. Let $S$ be a set of strings with index graphs $G_1, G_2, \ldots, G_{|S|}$ and weight function $w$. An index graph family for $S$ is a weighted DAG $G = (V, E, \ell)$ with

$$G = \left( V = \bigcup_{i=1}^{|S|} V_i, E = \bigcup_{i=1}^{|S|} E_i, f : E \mapsto \mathbb{R} \right)$$

For every edge $e \in E$, we have $f(e) = w(t_j)/|E(j)|$ with $j \in \{1, \ldots, |T(S)|\}$ and $t_j \in T(S)$ being the annotated substring of $e$.

The property

$$\sum_{e \in E(j)} f(e) = w(t_j) \ \forall j \in \{1, \ldots, |T(S)|\}$$

(2.1)

directly follows from this definition.

Figure 2.1 shows an index graph family for the set of substrings $\{ABC, CAB, CC\}$. An analysis of space and time complexity for the construction and shortest path computation of the index graphs follows later in this section.

**Figure 2.1:** The index graph family for the set of strings $S = \{ABC, CAB, CC\}$ and unit weights. For every edge, the substring annotation is followed by the weight (in red).
16 Theorem. Let $S$ be a set of strings with weight function $w$ and a minimum string cover $M$. Furthermore, let $b_i$ be a shortest weighted path from $(i,0)$ to $(i,|s_i|)$ in the index graph $G_i$ for each $i \in \{1, \ldots, |S|\}$. Then we have

$$\sum_{i=1}^{|S|} f(b_i) \leq \sum_{t \in M} w(t). \quad (2.2)$$

Proof Due to Equation (2.1) we have

$$\sum_{j:t_j \in M} \sum_{e \in E(j)} f(e) = \sum_{t \in M} w(t).$$

Since $M$ covers $S$, there exists a set of occurrences $D = \{(i,p,q) \in C(S) \mid s_i(p,q) \in M\}$ such that $D$ contains a factorization for every $s_i \in S$, see Definition 3. Due to Corollary 14, we can determine paths $b'_i$ from $(i,0)$ to $(i,|s_i|)$ using only edges that are annotated with elements of $M$. For the total weighted length of all these paths, we have

$$\sum_{i=1}^{|S|} f(b'_i) \leq \sum_{j:t_j \in M} \sum_{e \in E(j)} f(e) = \sum_{t \in M} w(t).$$

Since a shortest path’s weighted length is less than the weighted length of any other path having the same start and end nodes our claim follows:

$$\sum_{i=1}^{|S|} f(b_i) \leq \sum_{i=1}^{|S|} f(b'_i) \leq \sum_{t \in M} w(t).$$

The proof only requires that the sum of the weights of all edges annotated with substring $t$ is equal to $w(t)$ for all $t \in T(S)$, see Equation (2.1), while the exact weights of the edges are not relevant. Thus the weights can be adjusted according to necessity as long as they are nonnegative. Therefore, we can extent Definition 15 and modify the weights according to our needs. For instance, a higher and therefore better lower bound may be found by increasing the weights of edges in the shortest paths by a certain amount and lowering the weights of equally annotated edges that do not appear in any shortest path such that Equation (2.1) still holds and the edge weights remain nonnegative. Of course, this may change the shortest paths and we cannot rule out that any newly computed lower bound is worse than the old one. Algorithm 2 is an example for an iterative lower bound computation based on these observations.

Also note that while the sum of the lengths of the shortest paths in all index graphs is not necessarily an integer, the cardinality of a minimum string cover always is. For
\textbf{Data:} previously computed lower bound $\beta$
index graph family $G$
set of strings $S$

\textbf{Result:} final lower bound

1 \begin{algorithm}
2 \begin{align*}
3 & \text{new lower bound } \rho = 0 \\
4 & \text{for } i = 1 \text{ to } |S| \text{ do} \quad \text{// compute the lower bound} \\
5 & \quad \text{BFS}(G,(i,0)) \\
6 & \quad \rho = \rho + \text{shortestPathLength}((i,|s_i|)) \\
7 & \text{if } \rho \leq \beta \text{ then} \\
8 & \quad \text{return } \text{ceil}(\beta) \\
9 & \text{else} \\
10 & \quad M = \text{extractCover}(G) \quad \text{// see Algorithm 9} \\
11 & \quad H = \text{extractEdges}(G) \quad \text{// } H \text{ contains the edges of the shortest path} \\
12 & \quad \text{foreach } t_j \in M \text{ do} \\
13 & \quad \quad \text{foreach } e \in E(j) \text{ do} \\
14 & \quad \quad \quad \text{if } e \in H \text{ then} \quad \text{// } e \text{ belongs to the selected shortest path} \\
15 & \quad \quad \quad \quad e\.weight = w(t_j)/|H \cap E(j)| \\
16 & \quad \quad \quad \text{else} \\
17 & \quad \quad \quad \quad e\.weight = 0 \quad \text{// } e \text{ does not belong to the selected shortest} \\
18 & \quad \text{return } \text{Iterative Bound Computation}(\rho, G) \\
19 & \end{align*}
\end{algorithm}

Algorithm 2: Iterative Bound Computation - We evenly redistribute $w(t)$ among all edges $e$ that are annotated with $t$ and appear in a shortest path.
2.1. INDEX GRAPHS

Figure 2.2: This figure represents the index graph family also illustrated in Figure 2.1. Using a topological order, we reduce the graphs by merging the last node appearing in the topological order of one index graph with the first node of the next. The merged nodes are marked by two labels \((1,3)\) and \((2,3)\).

a problem with unit weights, we can therefore round the lower bound value to the next higher integer and still preserve the lower bound property.

Finally, the shortest path computation in all index graphs implicitly determines a cover due to Corollary 14. Although such a cover is not necessarily optimal or even better than the trivial solutions \(\Sigma\) and \(SSS\), we can use this observation in the branch and bound scheme described in Section 2.1.4.

Let \(S\) be a set of strings. Due to Definition 13, index graph \(G_i\) corresponding to \(s_i \in S\) has exactly \(|s_i| + 1\) nodes resulting in \(|S| + \sum_{i=1}^{|S|} |s_i|\) nodes for the whole index graph family. We can reduce this number to \(1 + \sum_{i=1}^{|S|} |s_i|\) by merging the last node \((i, |s_i|)\) of \(G_i\) with the first node \((i+1, 0)\) of \(G_{i+1}\) for all \(i \in \{1, \ldots, |S| - 1\}\), see Figure 2.2. Note that the resulting graph is still a DAG. Node \((i, p-1)\) has exactly \(|s_i| - p + 1\) outgoing edges as this is the amount of occurrences in \(s_i\) starting at position \(p\), see Definition 13. Therefore, we have \(\sum_{p=0}^{|s_i|} (|s_i| - p) = \sum_{p=0}^{|s_i|} p = (|s_i| + 1) \cdot |s_i|/2\) edges in \(G_i\) and \(\sum_{i=1}^{|S|} (|s_i| + 1) \cdot |s_i|/2\) edges in the index graph family \(G\). Since all our graphs are DAGs, the shortest paths can be determined by exploring every edge exactly once using BFS. Therefore, the lower bound can be computed in \(O\left(\sum_{i=1}^{|S|} |s_i|^2\right)\) time. While the worst case running time is nearly quadratic in the size of the input, we have an almost linear running time if the number of strings dominates the individual string lengths. Note that after merging the index graphs, the BFS can no longer be computed for each index graph in parallel. If we were to compute the shortest paths concurrently and had \(|S|\) machines available, the lower bound can be determined in \(O\left(\max\{|s_i|^2 \mid i \in \{1, \ldots, |S|\}\}\right) + K\) with \(K\) being the time required for synchronizing the results of the BFS in each index graph. Being able to compute the shortest paths in parallel also allows us to distribute the index graphs among the available resources.
The deviation of the lower bound from the optimal value cannot not bounded by a constant factor. For a minimum string cover $M$ and shortest paths $b_i$ in index graph $G$, we define the bound performance as $(\sum_{t \in M} w(t)) / (\sum_{i=1}^{S} f(b_i))$. As an example, we consider the MSC of the single string $\{A^{2n}\}$. It is clear that the cardinality of a minimum string cover is 1 as the corresponding index graph family consists of a single index graph. An edge annotated with the substring $A^p$ has the weight $f(A^p) = 1/(2n + 1 - p)$. Accordingly, the path $b = ((1, 0)(1, n)), ((1, n)(1, 2n))$ has a weighted length of $2 \cdot 1/(2n + 1 - n) = 2/(n + 1)$. Since the weighted length of a shortest path is not larger than $f(b)$, inserting the values for $w(\{A\}) = 1$ and $f(b)$ yields the following lower bound for the bound performance in this case

$$\frac{1}{n+1} = \frac{n + 1}{2} \in \Omega(n).$$

### 2.1.2 Lower Bound Calculation for MPSC

We adapt the lower bound heuristic to allow a maximum number of $g_i$ gaps within each string $s_i \in S$ with $i \in \{1, \ldots, |S|\}$. To this end, we replicate every index graph $G_i$ exactly $g_i$ times, that is, we generate $g_i + 1$ identical layers of $G_i$ such that index graph $G_i(h)$ with $0 \leq h \leq g_i$ belongs to layer $h$. Node $(i, p)$ of index graph $G_i(h)$ is denoted by $(i, p)_h$. To model the skipping of a symbol, we connect the layers with additional edges $((i, p)_h, (i, p+1)_{h+1})$ with $p \in \{0, \ldots, |s_i| - 1\}$ and $h \in \{0, \ldots, g_i - 1\}$, see Figure 2.3. These additional edges have the weight 0. In the index graph family, there is now a total of $\sum_{i=1}^{S} (g_i \cdot |s_i| \cdot (|s_i| + 1)/2 + g_i \cdot |s_i|)$ edges.

The resulting graph is still a DAG and we can apply BFS to determine shortest paths. Our new lower bound now considers the smallest weight of a shortest path from $(i, 0)_0$ to a node $(i, |s_i|)_h$ with $h \in \{0, \ldots, g_i\}$. Similar to Corollary 14, there exists a bijective mapping between partial factorizations of $s_i$ and paths from $(i, 0)_0$ to $(i, |s_i|)_h$. An edge $((i, p)_h, (i, p+1)_{h+1})$ belonging to such a path means that a gap is introduced in the corresponding partial factorization at position $(i, p+1)$.

To determine a partial cover with at most $g$ gaps in total, we first merge the index graphs as described in Section 2.1.1 and illustrated in Figure 2.2. Afterwards we apply the procedure introduced above for a single index graph. Note that a parallel distribution of the index graphs is possible for maximum number of gaps within each string. However, for a global maximum number of gaps, we must compute the lower bound in each index graph successively.

The described construction of an index graph family contains a significant amount of redundancy as the weighted edges appear in all layers. Therefore, we model the interconnection between two nodes of index graph $G_i$ only once instead of $g_i + 1$ times. We combine all nodes $(i, p)_h$ with $h \in \{0, \ldots, g_i\}$ into one structure containing an array with $g_i + 1$ entries. Entry $h$ represents node $(i, p)_h$. The new structure corresponding to the index
2.1. INDEX GRAPHS

Figure 2.3: This figure shows the layered index graph for the string $ABC$ and a maximum of two gaps. The weights are the same as in Figure 2.1. Choosing one of the blue edges between two nodes $(i, d)$ and $(i, d + 1)$ means that the computed lower bound uses a gap at position $d + 1$ in string $s_i$ independent of the layers connected by the specific blue edge. For a global number of gaps, the ending nodes $(1, 3)_0$, $(1, 3)_1$, and $(1, 3)_2$ are merged with the starting nodes of the next index graph. Note that we can delete the dotted nodes $(1, 0)_1$, $(1, 0)_2$ and $(1, 1)_2$ and any incident edges as these nodes are not reachable from node $(1, 0)_0$. This can be done for each index graph if we only have a maximum number of gaps $g_i$ in each string $s_i$ as opposed to a global number of gaps $g$. 
CHAPTER 2. ALGORITHMS AND ALGORITHMIC ANALYSIS

Graph family $G$ of $S$ has a total of $\sum_{i=1}^{\|S\|} (|s_i| \cdot (|s_i| + 1)/2 + |s_i|) = \sum_{i=1}^{\|S\|} (|s_i| \cdot (|s_i| + 3)/2$) edges. The total number of array entries in the structure is $\sum_{i=1}^{\|S\|} ((g_i + 1) \cdot (|s_i| + 1))$.

Altogether, we can reduce the required memory from $O(\sum_{i=1}^{\|S\|} g_i \cdot |s_i|^2)$ to $O(\sum_{i=1}^{\|S\|} |s_i|^2)$ as $g_i < |s_i|$ always holds. The runtime is not affected as we still have to consider each outgoing edge $e(v, u) \in E_i$ for all $g_i + 1$ entries of node $v \in V_i$, see Algorithm 3.

**Data:** index graph $G_i (V_i, E_i)$ with edge weights

maximum number of gaps $g_i$

**Result:** array containing the shortest paths for 0 to $g_i$ gaps

begin

1. for $h = 0$ to $g_i$ do

2. $(i, 0).shortestPath[h] = 0$ // shortestPath contains $g_i + 1$ entries

3. // for the shortest path in each node

4. for $v = (i, 0)$ to $(i, |s_i| - 1)$ do

5. for $g = 0$ to $g_i$ do

6. forall outgoing edges $e = (v, u)$ of $v$ do

7. if $e$ is a layer connecting edge then

8. $h' = h + 1$

9. else

10. $h' = h$

11. if $h' > g_i$ then

12. break

13. else

14. newLength = $v.\text{shortestPath}[h] + e.\text{weight}$

15. if newLength < $u.\text{shortestPath}[h']$ then

16. $u.\text{shortestPath}[h'] = \text{newLength}$

17. return $(i, |s_i|).\text{shortestPath}$

18. end

19. end

Algorithm 3: Shortest Paths with Gaps - We compute the shortest path from $(i, 0)$ to all other nodes. If the index graphs are merged we only initialize the shortest path lengths in $(1, 0)$.

2.1.3 Lower Bound Calculation for MSC/FB and MAC

To determine lower bounds for the MSC/FB and MPSC/FB, we delete any edges annotated with substrings of the forbidden substring set $F$ from the index graph family.
Abelian patterns only influence the weight function $f$ of the index graph family. Now every edge $e$ annotated with the substring $t$ is weighted with the reciprocal frequency of the placeholder Abelian pattern $a(t)$ in $S$ multiplied with $w(a)$, that is

$$f(e) = \frac{w(a)}{\big| \bigcup_{j:a(t_j)=a} E(j) \big|} \quad \forall e \in E(j).$$

### 2.1.4 Branch and Bound Scheme

Our first algorithm for solving the MSC is based on the branch and bound approach. It uses a directed binary branching tree, that is, every internal node (or every node that is not a leaf) has exactly two successors. Furthermore, every internal node of the branching tree is associated with a substring $t_j \in T(S)$ with $j \in \{1, \ldots, |T(S)|\}$ such that on any directed path from the root to a leaf each substring is encountered exactly once. Let node $v$ be associated with substring $t$. Then $t$ is not included in any cover represented by the leaves of the subtree rooted at the left child $l$ of $v$. Analogously $t$ is included in every cover represented by the leaves of the subtree rooted at the right child $r$ of $v$. Informally, any node inherits the decisions of its ancestors.

Next, we calculate the lower bounds with the help of the shortest paths in the index graph family. Note that we must consider the decisions of the ancestor nodes when determining the lower bound for a subtree.

- Substring $t$ is included: The weights of all edges annotated with $t$ are set to 0 and the calculated lower bound value is increased by $w(t)$.

- Substring $t$ is not included: All edges annotated with $t$ are deleted.

Recall that any lower bound calculation also generates a cover $M' \subseteq T(S)$. We determine $w(M') = \sum_{t \in M'} w(t)$ and update our current cover if $w(M')$ is less than the previous upper bound.

Finding a good branching sequence is the key to a good performance in any branch and bound scheme. The sequence can either be predetermined, see Algorithm 4, or it can be generated dynamically while exploring the search tree, see Algorithm 5. The biggest deficiency of a predetermined branching sequence is that without any further context, the selection of an edge annotated with a certain substring often has little or no influence at all in the next level of the branching tree. This leads to unnecessary computations. While any dynamic computation of the next branch requires more time, due to the immediate return, this approach almost always pays off.

Our final implementation is a hybrid of Algorithms 4 and 5, see Algorithm 6. We collect the substrings that are annotations of edges appearing in the shortest path of the last lower bound computation and that have not been previously selected in our branching tree. Of these substrings $t_j$ with $j \in \{1, \ldots, |T(S)|\}$ we select the one with the largest
value $|t_j| \cdot |C_j|/w(t_j)$. This strategy prefers long substrings with many occurrences and little weight.

```
Data: array $R$ of all substrings in $T(S)$
Result: sorted array $R$ containing the branching sequence
1 begin
2 for $i = 0$ to $|R| - 1$ do
3   $t_j = R[i]$
4   $t_j.sortingKey = |t_j| \cdot |C_j|/w(t_j)$
5 sort($R$)  // we sort the substrings according to increasing sorting key
6 return $R$
7 end
```

**Algorithm 4**: Example of a Predetermined Branching Strategy - This branching strategy uses the weight function of the index graph family. Longer substrings with many occurrences are preferred, while substrings with high weights are penalized. On every path we select the substrings in the same order.

```
Data: set $H$ of substrings found by the last lower bound computation
set $I$ of already selected substrings
Result: the next selected substring
1 begin
2 $H = H \setminus I$
3 return random($H$)
4 end
```

**Algorithm 5**: Example of a Dynamic Branching Strategy - We choose a non selected substring randomly.

After explaining the branching strategy, we describe our branch and bound algorithm. First, we need to create the index graph family and sort the nodes in topological order. Our initial cover is the better of either the set of all strings $S$ or the input alphabet $\Sigma$, see Algorithm 7.

Then we start the branching procedure, see Algorithm 8. Here, we first compute the lower bound, see Algorithm 10. If the determined lower bound for a subtree is at least as large as the weighted sum of the current cover, we do not need to explore the remaining part of this subtree. Otherwise we examine the cover induced by the current shortest path calculation, see Algorithm 9. If this new cover is better than the previously best found cover we update accordingly. Note that if the extracted cover is included in all solutions
2.1. INDEX GRAPHS

Algorithm 6: Hybrid Branching Strategy - We consider substrings that belong to the last determined cover and have not been selected before. These substrings are valued according to the criteria in Algorithm 4.

Algorithm 7: Initialization of the Branch and Bound Approach - This part creates the index graphs and starts the branching procedure.
**Data:** index graph family $G$

coverWeight is the weighted sum of all already selected and included substrings

cover is the best cover found so far

**Result:** the optimal cover of the current subtree or null if the current subtree can be ignored

```plaintext
1 begin
2   lowerBound = compute lower bound(G, coverWeight) // see Algorithm 10
3   if lowerBound ≥ w(cover) then
4     return null
5   tmp = extract cover(G) // get intermediate solution to be possibly
6   if w(tmp) < w(cover) then // used as upper bound, see Algorithm 9
7     cover = tmp
8   substring t = next branch() // This is a placeholder for any possible
9     oldWeight = t.edgeWeight // branching strategy such as Algorithm 6
10    foreach edge e annotated with t do
11      e.weight = 0
12      branch(G, coverWeight + w(t), cover)
13    foreach edge e annotated with t do
14      e.weight = ∞
15      branch(G, coverWeight, cover)
16    foreach edge e annotated with t do
17      e.weight = oldWeight
18   return cover
19 end
```

**Algorithm 8:** Branching Procedure - We compute the lower bound and return to the predecessor node if this subtree can be ignored. If the last lower bound computation yields a better cover then we update the cover. Afterwards we select a substring and continue to branch.
2.2 Linear Integer Programming

2.2.1 IP and MIP Formulations for MSC

Let $S$ be a set of strings with weight function $w$. We first describe the use of IP to solve the MSC. To this end, we introduce the following variables and notations:
CHAPTER 2. ALGORITHMS AND ALGORITHMIC ANALYSIS

\[ \min \sum_{j=1}^{\vert T(S) \vert} x_j \cdot w(t_j) \] (2.3)

\[ \sum_{(i,p,q) \in C(i,d)} y_{i,p,q} = 1 \quad \forall i \in \{1, \ldots, \vert S \vert \}, \forall d \in \{1, \ldots, \vert s_i \vert \} \] (2.4)

\[ y_{i,p,q} \leq x_j \quad \forall j \in \{1, \ldots, \vert T(S) \vert \}, \forall i \in \{1, \ldots, \vert S \vert \} \]
\[ \forall p,q \in \{1, \ldots, \vert s_i \vert \}, p \leq q, \quad t_j = s_i(p,q) \] (2.5)

\[ x_j \in \{0,1\} \quad \forall j \in \{1, \ldots, \vert T(S) \vert \} \] (2.6)

\[ y_{i,p,q} \in \{0,1\} \quad \forall i \in \{1, \ldots, \vert S \vert \}, \forall p,q \in \{1, \ldots, \vert s_i \vert \}, p \leq q \] (2.7)

Table 2.1: \textit{IP} \textit{MSC}

1. Each substring \( t_j \in T(S) \) with \( j \in \{1, \ldots, \vert T(S) \vert \} \) is represented by a binary variable \( x_j \in \{0,1\} \). Substring \( t_j \) is included in the minimum string cover if and only if \( x_j = 1 \).

2. Every occurrence \( (i,p,q) \) with \( i \in \{1, \ldots, \vert S \vert \}, p,q \in \{1, \ldots, \vert s_i \vert \}, \) and \( p \leq q \) is represented by a binary variable \( y_{i,p,q} \in \{0,1\} \). Similar to the substring variables, \( y_{i,p,q} = 1 \) if and only if \( (i,p,q) \) appears in a set \( D \) that contains a factorization of every string in \( S \).

3. The set of all occurrences containing position \( (i,d) \) with \( i \in \{1, \ldots, \vert S \vert \} \) and \( d \in \{1, \ldots, \vert s_i \vert \} \) is denoted by \( C(i,d) \), that is
\[ C(i,d) = \{(i,p,q) \in C(S) \mid p \leq d \leq q\} \]

\textit{IP} \textit{MSC} is described in Equations (2.3) to (2.7). Note that \textit{IP} \textit{MSC} is feasible as \( S \) is a feasible solution and variable values are bounded. Definition 17 describes the result of \textit{IP} \textit{MSC} while Theorem 18 shows that \textit{IP} \textit{MSC} solves the MSC.

17 Definition. Let \( S \) be a set of strings with weight function \( w \). For a feasible variable assignment of \textit{IP} \textit{MSC} containing substring variables \( \vec{x} \) and occurrence variables \( \vec{y} \), we call \( M(\vec{x}) = \{t_j \in T(S) \mid x_j = 1\} \) the extracted cover and \( C(\vec{y}) = \{(i,p,q) \in C(S) \mid y_{i,p,q} = 1\} \) the extracted factorizations with \( j \in \{1, \ldots, \vert T(S) \vert \}, i \in \{1, \ldots, \vert S \vert \}, p,q \in \{1, \ldots, \vert s_i \vert \} \) and \( p \leq q \).
18 Theorem. Let $S$ be a set of strings with weight function $w$. The variable assignment $\vec{x}$ and $\vec{y}$ is an optimal solution of $IP_{MSC}$ if and only if $M(\vec{x})$ is a minimum string cover of $S$ and $C(\vec{y})$ contains a factorization for all strings of $S$.

Proof First we prove that $\vec{x}$ and $\vec{y}$ are a feasible solution of $IP_{MSC}$ if and only if $M(\vec{x})$ covers $S$ and $C(\vec{y})$ contains a factorization of all strings in $S$.

$\Leftarrow$: Let $M$ be a cover of $S$ and let $D \subseteq C(S)$ be a corresponding set of occurrences as defined in Definition 3. $D$ contains a factorization for each string of $S$. For every string $s_i \in S$ with $i \in \{1, \ldots, |S|\}$, we select a factorization $D_i \subseteq D$ and set the associated occurrence variable of every occurrence in $D_i$ to 1 while all other occurrence variables corresponding to occurrences in $s_i$ are set to 0. For every substring in $M$, we set the associated substring variable to 1 while all other substring variables are set to 0. Conditions (2.6) and (2.7) are clearly satisfied. Condition (2.5) is valid due to Definition 3. Due to Corollary 2, Condition (2.4) is satisfied as well.

$\Rightarrow$: Pick any position $(i, d)$ with $i \in \{1, \ldots, |S|\}$ and $d \in \{1, \ldots, |s_i|\}$. Constraint (2.4) assures that this position is contained in at least one occurrence of $C(\vec{y})$ while Condition (2.7) guarantees that it is contained in at most one of these occurrences. Therefore, position $(i, d)$ is contained in exactly one occurrence of $C(\vec{y})$, and $C(\vec{y})$ contains a factorization of every string in $S$, see Corollary 2. Constraint (2.5) and Constraint (2.6) guarantee that $s_i(p, q) \in M(\vec{x})$ for every occurrence $(i, p, q) \in C(\vec{y})$. Hence, $M(\vec{x})$ is a cover of $S$.

Since the optimization value of any feasible solution $\vec{x}, \vec{y}$ of $IP_{MSC}$ is equal to the weighted sum of the extracted cover $M(\vec{x})$, $(\vec{x}, \vec{y})$ is optimal if and only if $M(\vec{x})$ is a minimum string cover.

\[ \Box \]

Equation (2.4) contains $2 \cdot \sum_{s_i \in S} |s_i|$ individual constraints. It is not possible to replace the $=$ symbol in Equation (2.4) by the symbol $\geq$ as shown in the following example. Consider strings $\{ABA, ABABA\}$. Every minimum string cover has cardinality 2. If we removed the $\leq$ inequality portion of Covering Constraint (2.4) then the solution corresponding to $\{ABA\}$ becomes feasible, see Figure 2.4.

Due to Equations (2.5) and (2.7), there are $\sum_{s_i \in S} |s_i| \cdot (|s_i| + 1)/2$ occurrence variables and occurrence constraints in $IP_{MSC}$. This value is also an upper bound for the number of substring variables, see Equation (2.6).

Next, we propose a MIP formulation of the MSC denoted by $MIP_{MSC}$ and subsequently show it to be correct. We transform $IP_{MSC}$ to $MIP_{MSC}$ by replacing Constraint (2.7) with

\[ y_{i, p, q} \in [0, 1] \ \forall i \in \{1, \ldots, |S|\}, \ \forall p, q \in \{1, \ldots, |s_i|\}, \ p \leq q. \quad (2.8) \]

Consider a set of string $S$ with weight function $w$. The extracted factorization of a $MIP_{MSC}$ solution does not necessarily contain factorizations for all strings. It is even pos-
Let it suffice to show that the extracted cover of \( y \) for every \( \text{FFF} \) holds. Due to Equation (2.9) and Covering Constraint (2.4) we have Equation (2.10) guaranteeing that the extracted factorization is empty, that is, no variable \( y_{i,p,q} \) with \( i \in \{1, \ldots, |\mathcal{S}|\} \), \( p, q \in \{1, \ldots, |\mathcal{T}(\mathcal{S})|\} \), and \( p \leq q \) has the value 1. Therefore, we expand Definition 17.

19 Definition. Let \( \mathcal{S} \) be a set of strings with weight function \( w \). For a feasible variable assignment of \( \text{MIP}_{\text{MSC}} \) containing substring variables \( \bar{x} \) and occurrence variables \( \bar{y} \), we call \( \bar{M}(\bar{x}) = \{ t_j \in \mathcal{T}(\mathcal{S}) \mid x_j = 1 \} \) the extracted cover, \( \mathcal{F}(\bar{y}) = \{(i,p,q) \in \mathcal{C}(\mathcal{S}) \mid y_{i,p,q} > 0\} \) the extracted factorization set, and \( \mathcal{Y}(\bar{y}) = \{ y \in \bar{y} \mid y > 0 \} \) the factorization variables with \( j \in \{1, \ldots, |\mathcal{T}(\mathcal{S})|\} \), \( i \in \{1, \ldots, |\mathcal{S}|\} \), \( p, q \in \{1, \ldots, |s_i|\} \), and \( p \leq q \).

Now, we address the relationship of the solutions for \( \text{IP}_{\text{MSC}} \) and \( \text{MIP}_{\text{MSC}} \) applied to the same problem instance.

20 Lemma. Let \( \mathcal{S} \) be a set of strings with weight function \( w \). For every optimal solution \((\bar{x}, \bar{y})\) of \( \text{MIP}_{\text{MSC}} \), there exists an optimal solution \((\bar{x}, \bar{y}')\) of \( \text{IP}_{\text{MSC}} \).

Proof It suffices to show that the extracted cover of \( \text{MIP}_{\text{MSC}} \) is a cover of \( \mathcal{S} \) as every feasible solution of \( \text{IP}_{\text{MSC}} \) is a feasible solution of \( \text{MIP}_{\text{MSC}} \) as well. Let \( \mathcal{F}(\bar{y}) \) be the extracted factorization set and \( \mathcal{Y}(\bar{y}) \) be the factorization variables of \( \text{MIP}_{\text{MSC}} \).

For \( i \in \{1, \ldots, |\mathcal{S}|\} \) and \( d \in \{1, \ldots, |s_i|\} \), we define \( \mathcal{F}(i,d) := \mathcal{F}(\bar{y}) \cap \mathcal{C}(i,d) \). Let \( y_{i,p,q} \in \mathcal{Y}(\bar{y}) \) be an occurrence variable with \( q < |s_i| \). Then

\[
\sum_{u,v:(i,u,v) \in \mathcal{F}(i,q)} y_{i,u,v} - y_{i,p,q} < \sum_{u,v:(i,u,v) \in \mathcal{F}(i,q+1)} y_{i,u,v} \tag{2.9}
\]

is a sufficient condition for the existence of an occurrence variable \( y_{i,q+1,q'} \in \mathcal{Y}(\bar{y}) \).

Similarly, for an occurrence variable \( y_{i,p,q} \in \mathcal{Y}(\bar{y}) \) with \( p > 1 \), there will always be an occurrence variable \( y_{i,p',p-1} \) if

\[
\sum_{u,v:(i,u,v) \in \mathcal{F}(i,p)} y_{i,u,v} - y_{i,p,q} < \sum_{u,v:(i,u,v) \in \mathcal{F}(i,p-1)} y_{i,u,v} \tag{2.10}
\]

holds. Due to Equation (2.9) and Covering Constraint (2.4) we have \( \mathcal{F}(i,q+1) \setminus \mathcal{F}(i,q) \neq \emptyset \) for every \( y_{i,p,q} \in \mathcal{Y}(\bar{y}) \) and \( q < |s_i| \), see Figure 2.5. The Covering Constraint (2.4) and Equation (2.10) guarantee \( \mathcal{F}(i,p-1) \setminus \mathcal{F}(i,p) \neq \emptyset \) for every \( y_{i,p,q} \in \mathcal{Y}(\bar{y}) \) and \( p > 1 \).
2.2. LINEAR INTEGER PROGRAMMING

Figure 2.5: Illustration to Lemma 20: The occurrence variable $y_{i,p,q}$ terminates at position $q$. Since the Covering Constraint (2.4) ensures that the sum of all occurrence variables appearing in $q$ and $q + 1$ equals 1, at least one positive occurrence variable with corresponding occurrence $(i, q + 1, q')$ exists.

□

Now we prove that the occurrence variable assignments of any optimal solution $(\vec{x}, \vec{y})$ of $MIP_{MSC}$ can be characterized as a linear combination of occurrence variable assignments in an optimal solution of $IP_{MSC}$.

We use the factorization variables $Y(\vec{y})$ and extracted factorization set $F(\vec{y})$ to compute the linear combinations via Algorithms 11 and 12. Algorithm 11 first initializes an empty list of variables $\vec{y}$ corresponding to a factorization of each string in a minimum string cover. Then it repeatedly determines the variable with minimum value of the current factorization variables. Algorithm 12 uses the current factorization set to generate a factorization that includes the occurrence corresponding to the above mentioned variable. This factorization is included in the list along with the value of the mentioned variable. Finally, we update the extracted factorization set and the factorization variables. Due to Lemma 20, Algorithm 12 always finds a valid factorization.

We formally describe the relationship between optimal solutions of $IP_{MSC}$ and an optimal solution of $MIP_{MSC}$ in the following lemma.

21 Lemma. For every solution $(\vec{x}, \vec{y})$ of $MIP_{MSC}$, there exist $m \in \{1, \ldots, |Y(\vec{y})|\}$ optimal solutions $(\vec{x}, \vec{y}_1), (\vec{x}, \vec{y}_2), \ldots, (\vec{x}, \vec{y}_m)$ of $IP_{MSC}$ with $\vec{y} = \sum_{h=1}^{m} k_h \cdot \vec{y}_h$ and positive values $k_h$ such that $\sum_{h=1}^{m} k_h = 1$ holds.

Proof $MIP_{MSC}$ produces the solution $(\vec{x}, \vec{y})$. Let $k_1$ be the smallest positive value in $\vec{y}$ and let $(i, p, q)$ be the corresponding occurrence. Algorithm 12 determines a vector $\vec{y}_1$. Note that

$$F(\vec{y}_1) \subseteq F(\vec{y}).$$ (2.11)

$F(\vec{y}_1)$ contains a factorization of all input strings. Therefore, $\vec{y}_1$ and the corresponding vector $\vec{x}_1$ are a feasible solution of $IP_{MSC}$. Due to Equation (2.11), $M(\vec{x}_1) \subseteq M(\vec{x})$ and hence $M(\vec{x}_1) = M(\vec{x})$ due to Lemma 20.

We set $\vec{y}' = \vec{y} - k_1 \cdot \vec{y}_1$. Note that Equation (2.9) and (2.10) continue to hold as $\vec{y}_1$ is a factorization of all strings. All entries of the new vector $\vec{y}'$ are in the interval $[0, 1 - k_1]$. Furthermore, we have $|Y(\vec{y}')| \leq |Y(\vec{y})| - 1$. We continue with the next iteration of
Data: set of strings $S$ with weight function $w$
optimal solution $\bar{x}, \bar{y}$ of $MIP_{MSC}$
$Y(\bar{y})$ factorization variables
$F(\bar{y})$ factorization set

Result: list $L$ of $m$ elements containing variable assignments $\bar{y}_h$ and coefficients $k_h$
for $h \in \{1 \ldots , m\}$

begin
1 $L = \emptyset$
2 $h = 1$
3 while $Y(\bar{y}) \neq \emptyset$ do
4 $\delta = \min(Y(\bar{y}))$
5 $k_h = \delta \cdot \text{value}$
6 $\bar{y}_h = \text{extractFactorization}(\bar{y}, \delta, S, F(\bar{y}))$ // see Algorithm 12
7 $\bar{y} = \bar{y} - k_h \cdot \bar{y}_h$
8 $L.insert(\bar{y}_h, k_h)$
9 $h = h + 1$
10 update $Y(\bar{y})$
11 update $F(\bar{y})$
12 return $L$
end

Algorithm 11: Determine Linear Combinations for the MSC - $h$ marks the current iteration. Each iteration computes one further coefficient $k_h$ and vector $\bar{y}_h$. 
Data: variable assignment $\vec{y}$
smallest nonzero variable $\delta \in \vec{y}$
set of strings $S$
extracted factorization set $F(\vec{y})$

Result: vector $\vec{y}_h$ being the occurrence variables of an optimal solution of $IP_{MSC}$

```
1 begin
2   $\vec{y}_h = 0 \cdot \vec{y}$ // $\vec{y}_h$ and $\vec{y}$ have the same dimension
3   $(i, p, q) = \delta$.occurrence // $s_i$ is the string in which the smallest
4   $\vec{y}_h[(i, p, q)] = 1$ // occurrence appears
5   while $q \neq |s_i|$ do
6       pick positive occurrence variable $y_{i,q+1,q'} \in \vec{y}$
7       $\vec{y}_h[(i, q + 1, q')] = 1$
8       $q = q'$
9       $(i, p, q) = \delta$.occurrence
10      while $p \neq 1$ do
11         pick positive occurrence variable $y_{i,p',p-1} \in \vec{y}$
12         $\vec{y}_h[(i, p', p-1)] = 1$
13         $p = p'$
14      foreach $s_f \in S \setminus \{s_i\}$ do // determine an arbitrary factorization
15         $p = 1$ // for each string $s_f$
16         while $p \leq |s_f|$ do
17             pick positive occurrence variable $y_{f,p,q} \in \vec{y}$
18             $\vec{y}_h[(f, p, q)] = 1$
19             $p = q + 1$
20      return $\vec{y}_h$
21 end
```

Algorithm 12: Extract Factorization - We determine a factorization in the current $F(\vec{y})$ containing the occurrence $(i, p, q)$ corresponding to $\delta$. For any string other than $s_i$, the factorization can be arbitrary. The existence of bordering occurrences in $Y(\vec{y})$ in lines 6 and 17 is guaranteed by Equation (2.9), and the existence of bordering occurrences in $Y(\vec{y})$ in line 11 is guaranteed by Equation (2.10).
\[
\min \sum_{j=1}^{T(S)} x_j \cdot w(t_j) \tag{2.12}
\]

\[
\sum_{(i,p,q) \in C(i,d)} y_{i,p,q} = z_{i,d} \quad \forall i \in \{1, \ldots, |S|\}, \forall d \in \{1, \ldots, |s_i|\} \tag{2.13}
\]

\[
y_{i,p,q} \leq x_j \quad \forall j \in \{1, \ldots, |T(S)|\}, \forall i \in \{1, \ldots, |S|\}, \forall p, q \in \{1, \ldots, |s_i|\}, p \leq q, t_j = s_i(p, q) \tag{2.14}
\]

\[
\sum_{i=1}^{S} \sum_{d=1}^{|s_i|} z_{i,d} \geq \left( \sum_{i=1}^{S} |s_i| \right) - g \tag{2.15}
\]

\[
x_j \in \{0, 1\} \quad \forall j \in \{1, \ldots, |T(S)|\} \tag{2.16}
\]

\[
y_{i,p,q} \in \{0, 1\} \quad \forall i \in \{1, \ldots, |S|\}, \forall p, q \in \{1, \ldots, |s_i|\}, p \leq q \tag{2.17}
\]

\[
z_{i,d} \in \{0, 1\} \quad \forall i \in \{1, \ldots, |S|\}, \forall d \in \{1, \ldots, |s_i|\} \tag{2.18}
\]

**Table 2.2: IP_{MPSC}**

Algorithm 11. After \(h\) iterations, we have \(\bar{y}' = \overline{y} - \sum_{i=1}^{h} k_i \cdot \overline{y}_i, |Y(\bar{y}')| \leq |Y(\overline{y})| - h\), and all variables of \(\bar{y}'\) are in the interval \([0, 1 - \sum_{i=1}^{h} k_i]\). The algorithm terminates if \(|Y(\bar{y}')| = 0\), that is after at most \(|Y(\overline{y})|\) iterations.

\[\square\]

### 2.2.2 IP Formulation for MPSC

Table 2.2 presents an IP formulation to solve MPSC, called IP_{MPSC}. IP_{MPSC} contains three modifications compared to IP_{MSC}, see Table 2.1. Again, let \(S\) be a set of strings with weight function \(w\) and let the positive integer \(g\) be the maximum number of allowed gaps.

For every string \(s_i\) with \(i \in \{1, \ldots, |S|\}\) and every position \((i,d)\) in \(s_i\) with \(d \in \{1, \ldots, |s_i|\}\), we introduce a binary variable \(z_{i,d} \in \{0, 1\}\), see the new Condition (2.18). If position variable \(z_{i,d} = 1\) then position \((i,d)\) is contained by a factorization belonging to a minimum partial string cover whereas \(z_{i,d} = 0\) denotes that position \((i,d)\) is a gap. This relationship is expressed in Constraint (2.13), a modification of Constraint (2.4). Finally, the total number of allowed gaps must be less than \(g\) leading to the other new Con-
2.2. LINEAR INTEGER PROGRAMMING

Alternatively or additionally, further constraints, like \( \sum_{d=1}^{s_i} z_{i,d} \geq |s_i| - g_i \), can model the maximum number of allowed gaps \( g_i \) for the individual string \( s_i \).

Definition 22 and Theorem 23 for \( IP_{MP SC} \) correspond to Definition 17 and Theorem 18 for \( IP_{MSC} \), respectively.

22 Definition. Let \( S \) be a set of strings with weight function \( w \) and let the positive integer \( g \) be the maximum number of allowed gaps in \( S \). For a feasible variable assignment of \( IP_{MP SC} \) containing substring variables \( \vec{x} \) and occurrence variables \( \vec{y} \), we call \( MMM(\vec{x}) = \{ t_j \in T(S) | x_j = 1 \} \) the extracted partial cover, \( CCC(\vec{y}) = \{ (i, p, q) \in C(S) | y_{i,p,q} = 1 \} \) the extracted partial factorizations, and \( Z(\vec{z}) = \{ (i, d) | z_{i,d} = 0 \} \) the extracted gaps with \( j \in \{1, \ldots, |T(S)|\} \), \( i \in \{1, \ldots, |S|\} \), \( p, q, d \in \{1, \ldots, |s_i|\} \), and \( p \leq q \).

23 Theorem. Let \( S \) be a set of strings with weight function \( w \) and let the positive integer \( g \) be the maximum number of allowed gaps in \( S \). The variable assignment \( \vec{x}, \vec{y}, \) and \( \vec{z} \) is an optimal solution of \( IP_{MP SC} \) if and only if \( MMM(\vec{x}) \) is a minimum partial string cover of \( S \) with at most \( g \) gaps and \( CCC(\vec{y}) \) contains a partial factorization \( D_i \) for all strings \( s_i \in S \) such that \( \sum_{i=1}^{|S|} g(D_i) \leq g \).

Proof We say that two gap positions \((i, p)\) and \((i, q)\) in a string are neighboring if and only if there is no gap position \((i, d)\) such that \( p < d < q \) with \( i \in \{1, \ldots, |S|\} \), \( p, d, q \in \{1, \ldots, |s_i|\} \). Given a set of gap positions \( Z \), we transform the MPSC into an MSC by removing the gap positions and considering the substrings between two neighboring gap positions as two separate strings. Similarly, the substring between the beginning of a string and the first gap position in this string as well as the substring between the last gap position of a string and the end of this string are separate strings. Then we can apply Theorem 18. For the \( \Leftarrow \) direction, the partial factorizations define the gap positions while for the \( \Rightarrow \) direction, the gap positions are given by the extracted gaps \( Z(\vec{z}) \).

\( \square \)

As in Section 2.2.1, we propose an MIP formulation of the MSC denoted by \( MIP_{MP SC} \) and then show that it produces a minimum partial string cover with not more than the allowed number of gaps. To transform \( IP_{MP SC} \) to \( MIP_{MP SC} \), we replace Constraints (2.17) and (2.18) with Constraints (2.19) and (2.20), respectively.

\[
y_{i,p,q} \in [0,1] \quad \forall i \in \{1, \ldots, |S|\}, \forall p, q \in \{1, \ldots, |s_i|\}, p \leq q \quad (2.19)
\]

\[
z_{i,d} \in [0,1] \quad \forall i \in \{1, \ldots, |S|\}, \forall d \in \{1, \ldots, |s_i|\} \quad (2.20)
\]

The extracted cover \( MMM(\vec{x}) \), the extracted factorization set \( F(\vec{y}) \), the factorization variables \( Y(\vec{y}) \) of Definition 19 also apply to \( MIP_{MP SC} \) although we are actually using partial covers and partial factorizations. In Lemma 24, we will show the relationship of an
MIP_{MPSC} solution to individual partial covers. To this end, we use Algorithms 13 and 14 that are adaptations of Algorithms 11 and 12 discussed in Section 2.2.1.

**Data:** set of strings $S$ with weight function $w$

optimal solution $\vec{x}, \vec{y}, \vec{z}$ of MIP_{MPSC}

$Y(\vec{y})$ factorization variables

$F(\vec{y})$ factorization set

**Result:** list $L$ of $m$ elements containing variable assignments $\vec{y}_h, \vec{z}_h$ and coefficients $k_h$ for $h \in \{1, \ldots, m\}$

```plaintext
1 begin
2    $L = \emptyset$
3    $h = 1$
4    while $Y(\vec{y}) \neq \emptyset$ do
5        $\delta = \min(Y(\vec{y}))$
6        $k_h = \delta$.value
7        $\vec{y}_h, \vec{z}_h = \text{extractFactorization}(\vec{y}, \vec{z}, \delta, S, F(\vec{y}))$  // see Algorithm 14
8        $\vec{y} = \vec{y} - k_h \cdot \vec{y}_h$
9        $\vec{z} = \vec{z} - k_h \cdot \vec{z}_h$
10       $L$.insert($\vec{y}_h, \vec{z}_h, k_h$)
11       $h = h + 1$
12      update $Y(\vec{y})$
13      update $F(\vec{y})$
14    return $L$
15 end
```

**Algorithm 13:** Determine Linear Combinations for MPSC - $h$ marks the current iteration. Each iteration computes one further coefficient $k_h$ and vectors $\vec{y}_h$ and $\vec{z}_h$.

**24 Lemma.** For every solution $(\vec{x}, \vec{y}, \vec{z})$ of MIP_{MPSC}, there exist $m \in \{1, \ldots, |Y(\vec{y})|\}$ binary valued vectors $(\vec{y}_1, \vec{z}_1), (\vec{y}_2, \vec{z}_2), \ldots, (\vec{y}_m, \vec{z}_m)$ with $\vec{y} = \sum_{h=1}^{m} k_h \cdot \vec{y}_h$, $\vec{z} = \sum_{h=1}^{m} k_h \cdot \vec{z}_h$, and $\sum_{h=1}^{m} k_h \leq 1$ with $k_h > 0$.

**Proof** Algorithms 13 and 14 produce the binary valued vectors $(\vec{y}_1, \vec{z}_1), (\vec{y}_2, \vec{z}_2), \ldots, (\vec{y}_m, \vec{z}_m)$ with $\vec{y} = \sum_{h=1}^{m} k_h \cdot \vec{y}_h$, $\vec{z} = \sum_{h=1}^{m} k_h \cdot \vec{z}_h$ and $k_h > 0$. Therefore, we need to address the condition $\sum_{h=1}^{m} k_h \leq 1$. We prove that the results of the algorithms fulfill this condition by induction on $m$:

- **Induction Base:** For every entry of $\vec{z}$, we have $z_{i,d} \leq 1 - \sum_{k=1}^{0} k_h = 1$ for all $i \in \{1, \ldots, |S|\}$ and $d \in \{1, \ldots, |s_i|\}$ due to Constraint (2.13).
- **Induction Assumption:** The condition holds before the execution of iteration $r$. 
**Data:** variable assignment $\vec{y}, \vec{z}$  
smallest nonzero variable $\delta \in \vec{y}$  
set of strings $S$  
extracted factorization set $F(\vec{y})$

**Result:** vectors $\vec{y}_h, \vec{z}_h$

1. $\vec{y}_h = 0 \cdot \vec{y}$ \hspace{1cm} // $\vec{y}_h$ and $\vec{y}$ have the same dimension
2. $\vec{z}_h = 1$ \hspace{1cm} // $\vec{z}_h$ and $\vec{z}$ have the same dimension
3. $(i, p, q) = \delta.\text{occurrence}$ \hspace{1cm} // $s_i$ is the string in which the smallest
4. $\vec{y}_h[(i, p, q)] = 1$ \hspace{1cm} // occurrence appears

while $q \neq |s_i|$ do

7. if $\exists$ positive occurrence variable $y_{i,q+1,q'} \in \vec{y}$ then
   8. $\vec{y}_h[(i, q + 1, q')] = 1$
   9. $q = q'$

else
   10. $\vec{z}_h[(i, q + 1)] = 0$
   11. $q = q + 1$

13. $(i, p, q) = \delta.\text{occurrence}$

while $p \neq 1$ do

15. if $\exists$ positive occurrence variable $y_{i,p',p-1} \in \vec{y}$ then
   16. $\vec{y}_h[(i, p', p - 1)] = 1$
   17. $p = p'$

else
   18. $\vec{z}_h[(i, p)] = 0$
   19. $p = p - 1$

21. foreach $s_k \in S \setminus \{s_i\}$ do

22. $p = 1$

23. while $q \leq |s_k|$ do

24. if $\exists$ positive occurrence variable $y_{k,p,q} \in \vec{y}$ then
   25. $\vec{y}_h[(k, p, q)] = 1$
   26. $p = q + 1$

else
   27. $\vec{z}_h[(k, p)] = 0$
   28. $p = p + 1$

return $\vec{y}_h, \vec{z}_h$

Algorithm 14: Extract Partial Factorization - We determine a partial factorization in the current $F(\vec{y})$ containing the occurrence $(i, p, q)$ corresponding to $\delta$. For any string other than $s_i$, the partial factorization can be arbitrary. As some positions are not contained in a partial factorization, we have to consider each position of a string.
Figure 2.6: Illustration to Lemma 24. We subtract exactly $k_r$ from every position variable whose position is contained in a picked occurrence. Since none of the occurrences containing $z_{i,p}$ (green, blue, and purple) were picked, they must all overlap with the last occurrence that was picked (red). This results in $z_{i,q'} \geq z_{i,p} + k_r$ which is a contradiction to the assumption $z \leq 1 - \sum_{h=1}^{r-1} k_h$.

- Induction Proof: If a position $(i, p)$ belongs to an occurrence whose variable has been picked in iteration $r$ then we subtract $k_r$ from the corresponding position variable, see line 9 of Algorithm 13. The inequality $z_{i,p} \leq 1 - \sum_{h=1}^{r} k_h$ holds after iteration $r$ due to the induction assumption.

Let $(i, p)$ be a position with $z_{i,p} > 1 - \sum_{h=1}^{r} k_h$ that is not contained in any occurrence whose variable has been picked in iteration $r$. Let us assume that occurrences are picked in increasing order of positions (lines 7 and 24 in Algorithm 14). The proof for the reverse direction (line 15 of Algorithm 14) is analogous. Then all occurrences with a positive occurrence variable that contain position $(i, p)$ must overlap with the occurrence $(i, p', q')$ whose variable was the last one to be picked. Therefore, $z_{i,q'} \geq z_{i,p} + k_r > 1 - \sum_{h=1}^{r} k_h + k_r = 1 - \sum_{h=1}^{r-1} k_h$ before iteration $r$. This a contradiction to the induction assumption, see Figure 2.6.

\[\square\]

25 Lemma. Let $S$ be a set of strings with weight function $w$ and let the positive integer $g$ be the maximum number of gaps in $S$. For every optimal solution $(\vec{x}, \vec{y}, \vec{z})$ of $MIP_{MPSC}$, there exists an optimal solution $(\vec{x}, \vec{y}', \vec{z}')$ of $IP_{MPSC}$.

Proof We only need to show that for one of the vectors $\vec{y}_h$ of Lemma 24, $C(\vec{y}_h)$ contains a partial factorization for every string in $S$ with at most $g$ gaps in total. This is the case if the Gap Constraint (2.15) is fulfilled.
2.2. LINEAR INTEGER PROGRAMMING

We apply a proof by contradiction and assume that for every \( \vec{z}_h \)
\[
|S| \sum_{i=1}^{m} z_{i,d}(h) < \left( \sum_{i=1}^{m} |s_i| \right) - g
\]
holds with \( z_{i,d}(h) \) being an entry of \( \vec{z}_h \). Lemma 24 leads to
\[
\sum_{i=1}^{m} |s_i| \sum_{p=1}^{k} z_{i,d}(h) = m \sum_{i=1}^{m} k_h \sum_{p=1}^{k} z_{i,d}(h).
\]
With our assumption and \( \sum_{k=1}^{m} k_h \leq 1 \) from Lemma 24, we have
\[
\sum_{k=1}^{m} k_h \sum_{i=1}^{m} |s_i| z_{i,d}(h) < \sum_{k=1}^{m} k_h \left( \sum_{i=1}^{m} |s_i| - g \right) = \left( \sum_{i=1}^{m} |s_i| - g \right) \sum_{k=1}^{m} k_h \leq \left( \sum_{i=1}^{m} |s_i| - g \right).
\]
This is a contradiction, as \( \sum_{k=1}^{m} \sum_{i=1}^{m} |s_i| z_{i,d} < \left( \sum_{i=1}^{m} |s_i| \right) - g \) would mean that \((\vec{x}, \vec{y}, \vec{z})\) is not a feasible solution of the \( MIP_{MPSC} \).

\( \square \)

Not every binary valued vectors of Lemma 24 is necessarily a feasible solution of \( IP_{MPSC} \). We support this claim with an example. For the string set \{ABAB\} and one allowed gap, \( MIP_{MPSC} \) can find the partial cover \{AB\} with \( y_{1,1,2} = 1 \) and \( y_{1,3,4} = 0.5 \) resulting in \( z_{1,1} = 1, z_{1,2} = 1, z_{1,3} = 0.5, \) and \( z_{1,4} = 0.5 \). This solution is feasible as \( z_{1,1} + z_{1,2} + z_{1,3} + z_{1,4} = 3 = 4 - 1 \). Applying Algorithm 14, the occurrence variables of \( \vec{y} \) results in \( k_1 = 0.5 \) with \( y_{1,1,2} = y_{1,3,4} = 1 \) in \( \vec{y_1} \) and \( k_2 = 0.5 \) with \( y_{1,1,2} = 0.5 \) and \( y_{1,3,4} = 0 \) in \( \vec{y_2} \). The second vector \( \vec{y_2} \) is not a feasible solution. Now, we only consider the feasible optimal solutions of \( IP_{MPSC} \) for this instance. These are \{AB\} (with \( y_{1,1,2} = y_{1,3,4} = 1 \), \{ABAB\}, \{ABA\} and \{BAB\}. Using only these solutions, it is not possible to find a linear combination.

Finding the Largest Cover

Lemma 25 states that \( MIP_{MPSC} \) finds an optimal partial cover. However, it is not easy to determine the occurrences of a suitable partial factorization. For example, we again consider the string set \{ABA, ABABA\} with a maximum of two gaps. An optimal solution is given by \( M = \{ABA\} \). \( MIP_{MPSC} \) may find the solution \( y_{2,1,3} = y_{2,3,5} = 0.5 \) and \( y_{1,1,3} = 1 \), see Figure 2.7, which is an infeasible solution for \( IP_{MPSC} \). For a practical application, it is likely not useful to only establish the existence of the minimum partial cover. Therefore, we briefly describe how to obtain the factorization with the least number
CHAPTER 2. ALGORITHMS AND ALGORITHMIC ANALYSIS

Figure 2.7: We are given the set of strings \{ABA, ABABA\} with a maximum of two allowed gaps. The only optimal solution consists of the substring \{ABA\}. The left side shows the optimal solution found by IP\textsubscript{MPSC}. Here \(y_{1,1,3} = y_{2,1,3} = 1\) and \(z_{2,4} = z_{2,5} = 0\). A solution of MIP\textsubscript{MPSC} on the right side illustrates a cover with \(y_{2,1,3} = y_{2,3,5} = 0, 5\) and corresponding index variables. This solution is feasible for the relaxed IP and not feasible for the binary IP. Its existence only guarantees that IP\textsubscript{MPSC} could find factorizations with an equal or less number of gaps.

of gaps of a string \(s_i \in S\) with \(i \in \{1, \ldots, |S|\}\) in a minimum partial string cover \(M\). To this end, we construct the index graph family, see Definition 15, using only edges annotated with an element in \(M\). Bridging edges remain unweighted, every other edge \(e(v_p, v_q)\) is given the weight \(q - p\). Computing the longest path from starting node \(v_{i,0}\) to destination node \(v_{i,|s_i|}\) yields the wanted factorization for string \(s_i\). In a DAG, the longest path can be computed with BFS [13].

2.2.3 IP Formulations for MSC/FS and MAC

To solve MSC/FB, we do not create any substring and occurrence variables for elements of the forbidden set \(F\).

Abelian patterns require a modification of all variables \(x\). Instead of representing a substring, \(x_j\) now represents an Abelian pattern \(a_j\) for \(j \in \{1, \ldots, |A(T(S))|\}\), and \(x_j = 1\) holds if and only if \(a\) is included in the computed optimal solution. Note that we only create an Abelian pattern \(a\) if it corresponds to at least one substring occurring in \(T(S)\), that is \(a \in A(T(S))\). We also modify our objective to \(\min \sum_{j=1}^{|A(T(S))|} x_j \cdot w(a_j)\). Lastly, we must associate the substring occurrences with the corresponding Abelian pattern. For every occurrence variable, the occurrence constraint now consists of \(y_{i,p,q} \leq x_a\) with \(j \in \{1, \ldots, |A(T(S))|\}, \; i \in \{1, \ldots, |S|\}, \; p, q \in \{1, \ldots, |s_i|\}, \; p \leq q, \; \text{and} \; a_j = a(s_i(p, q))\). We provide IP\textsubscript{MAC} in Table 2.3.
\[ \begin{align*}
\min & \quad \sum_{j=1}^{\|A(T(S))\|} x_j \cdot w(a_j) \\
\sum_{(i,p,q) \in C(i,d)} y_{i,p,q} &= 1 \quad \forall i \in \{1, \ldots, |S|\}, \forall d \in \{1, \ldots, |s_i|\} \quad (2.22) \\
y_{i,p,q} &\leq x_j \quad \forall j \in \{1, \ldots, |A(T(S))|\}, \forall i \in \{1, \ldots, |S|\}, \\
&\quad \forall p, q \in \{1, \ldots, |s_i|\}, p \leq q, a_j = a(s_i(p, q)) \quad (2.23) \\
x_j &\in \{0, 1\} \quad \forall j \in \{1, \ldots, |A(T(S))|\} \quad (2.24) \\
y_{i,p,q} &\in \{0, 1\} \quad \forall i \in \{1, \ldots, |S|\}, \forall p, q \in \{1, \ldots, |s_i|\}, p \leq q \quad (2.25)
\end{align*} \]

Table 2.3: \(IP_{MAC}\)
Chapter 3

Experimental Evaluation

Recall that the MSC and its related problems are NP-hard [17] and that even a constant approximation of the MSC is NP-hard [8]. Nevertheless, we can still hope for algorithms that quickly produce good results for many input data sets. Such efficiency is particularly relevant for any practical application of the MSC or its related problems as such application is likely to use large input data sets. However, given the complexity of the MSC, a theoretical worst case analysis is only of limited use in determining the performance of the developed algorithms. Therefore, we present in this chapter an experimental evaluation of the algorithms. To provide a comprehensible description of the experiments, we first give an overview of the resources used during the experiments. Then we explain our methodology and our approach to generate test instances. Due to the total size of the generated data, it is not possible to include all data in this thesis. To comply with the standards of proper scientific conduct, we have stored the data on a separate medium. Implementation details of the tested algorithms that go beyond the information of Chapter 2 are given next. Afterwards we show the results of the experiments in a condensed form while the individual results are again available electronically. Based on these results, we compare our algorithms.

3.1 Resource Description

The experiments were executed on a distributed system consisting of twelve computers. The system enabled us to run different and independent experiments on different computers in parallel. However, we did not use parallel code that concurrently runs on several computers. Computing jobs were submitted to a batch system that automatically distributed them to the various computers. The computers in the distributed system all had the same configuration:

Processor Intel Core™2 Quad processor with 2.66 GHz clock frequency

L2-Cache 3 MB
Main Memory 8 GB RAM

Operation System 64 bit Ubuntu 9.04

The algorithms were implemented in C++ using the Gnu GCC 4.2.1 compiler. In addition, procedures and classes for regular expressions were taken from the boost library [11] [10]. To implement IP and MIP algorithms, we used the ABACUS framework [5] [12] and the CPLEX 11.2 solver. Up to eight CPLEX licenses enabled us to execute several instances in parallel.

3.2 Experiment Methodology

To perform the experiments, we always used the same methodology consisting of four steps:

1. Instance Generation: We produced a test instance as described in Section 3.3. The data of the instance was stored in an input text file.

2. Preprocessing: The input data was used to generate the required strings and data structures.

3. Optimization: After preprocessing, the optimization run was started and its running time was measured by using the C++ function *difftime*(). The program produced an output text file containing the running time and the generated minimum string cover.

4. Verification: The minimum string cover was verified with the help of regular expressions and the input text file.

The preprocessing step was identical for all experiments except for the existing IP approach [8]. For the latter algorithm, a more elaborate preprocessing step was necessary as all possible factorizations are required while substrings are sufficient for the other algorithms. The running time of the standard preprocessing step never exceeded 10 seconds even for large input files. Note that the preprocessing step was not included in the measurement of the running time.

Verification of the result was simple if the minimum string cover obtained during input generation was identical to the minimum cover found by the optimization algorithms. However, it was possible that the optimization algorithms produced a different minimum string cover. In this case, we determined whether the optimization result was able to generate the original minimum cover. Note that such a generation is a sufficient but not a necessary condition for the validity of the optimization result. However, we never encountered a case in which this approach failed.

Apart from producing the desired result, some experiments also failed to complete due to an excessive running time. Experiments that exceeded a running time of 8 hours were
aborted. For some instances of the existing IP approach, we were not even able to complete the preprocessing step. To guarantee a fair comparison, we did not apply any fine tuning for any of the algorithms. This also includes the memory modification of the index graph formulation for MPSC, see Section 2.1.2. Altogether, we ran about 2000 experiments using more than 2000 CPU hours.

Our experiments were focussed on MSC with unit weight function as we do not see a significant algorithmic difference when allowing general weight functions. We also executed a few experiments for MPSC as MPSC leads to significantly larger index graphs. For the experiments, we ignored MSC/FB and MAC. Using our random approach to generate inputs, MSC/FB will typically result in a smaller computational effort as the solution space is reduced. However, note that for real input data, MSC/FB can increase the computation time when trivial solutions are eliminated. As an Abelian pattern represents multiple substrings we also expect a smaller computation time for MAC than for MSC.

### 3.3 Instance Generation

Randomly generated strings often lead to instances with $S$ or $\Sigma$ being the minimum string cover. We generated input instances of this kind to test the ability of our algorithms to also handle these cases. But primarily, we were interested in instances with the cardinality of the minimum string cover being strictly smaller than $\min\{|S|, |\Sigma|\}$. To this end, we chose a backward approach. First, we considered a set of strings of a would-be minimum set cover $M \subset \Sigma^*$. Then we constructed a set of input strings $S \subset M^*$. Prior to generating any strings, we had to specify several parameters:

- $|\Sigma|$ and $|S|$: Usually, we selected the same value for both.
- Maximum and minimum length of strings in $M$.

Following additional parameters were determined randomly using a single random seed that was obtained from the name of the destination file for the input data.

- $|M|$ not exceeding $\min(|\Sigma|, |S|) - 1$.
- Length of the strings in $M$ within the specified interval.
- Length of the strings in $S$.
- Composition of the strings in $M$.
- Composition of the elements of $S$.

Algorithm 15 describes the input generation. $M$ is not guaranteed to be the only minimum string cover of the generated set of strings $S$ or even a minimum string cover
**Data:** input parameters, see Section 3.3  
random seed  
**Result:** set of strings $S$

```plaintext
begin

1 $M = \emptyset$

2 determine size of $M$

3 for $i = 1$ to $|M|$ do

4 determine length of $M[i]$

5 for $j = 1$ to $M[i].\text{length}$ do

6 $M[i][j] = \text{random}(\Sigma)$ // random selection of a symbol in $\Sigma$

7 for $i = 1$ to $|S|$ do

8 randomly determine minimum length of $S[i]$

9 $j = 0$

10 $s_i = \epsilon$ // initialization of $s_i$

11 repeat

12 $t = \text{random}(M)$ // random selection of a string in $M$

13 $s_i = s_i.\text{concatenate}(t)$

14 $j = j + t.\text{length}$

15 until $j \geq S[i].\text{length}$;

16 return $S$

end
```

**Algorithm 15:** Instance Generation
at all. For instance, if $M$ includes strings $A$ and $AA$, $M \setminus \{AA\}$ covers any set of strings generated using $M$. As demonstrated in Section 2.1.1 a significant amount of repetition is a challenge for the index graph based branch and bound algorithm. Therefore, we produced some examples that exhibit this property to particularly test the performance of the index graph approach. As instances with short strings and little repetition do not pose any problems for our algorithms, we included only few of these instances in our experiments.

For MPSC experiments, we did not use the instances of the MSC experiments. Due to the described approach of input generation, it is unlikely that for a small maximum number of gaps, a problem instance has a minimum partial string cover different from the minimum string cover. Furthermore, the verification of optimality is very difficult. Therefore, we inserted a symbol not appearing in any string of our generation set $M$ at $g$ randomly introduced positions between concatenated strings in $M$ with $g$ being the maximum number of allowed gaps. Note that this approach is based on the same idea as the proof of Theorem 23.

An ideal experimental evaluation of algorithms uses real input data as random data rarely adequately represent real problems. However, this approach is only feasible if the potential application of these algorithms is already well established. In case of the MSC, we have only identified potential uses of the algorithms but such use must be verified in a separate study that considers practical constraints. For instance, in biology applications, the general MSC will almost always produce $\Sigma$ or $SSS$ as a minimum string cover. To obtain a useful result, we must use MSC/FB with a careful selection of the forbidden strings. Unfortunately, such a study is out of the scope of this thesis. Nevertheless, we believe that the information based on the randomly generated input data already gives sufficient hints regarding the practical performance of the algorithms.

### 3.4 Implementation

#### 3.4.1 Brute Force Algorithm

Brute force algorithms typically evaluate all possible solution candidates and randomly select the next candidate. We propose an algorithm consisting of four steps based on regular expressions.

1. Determination of all substrings $T(S)$ occurring in $S$.
2. Selection of a candidate solution $M' \subseteq T(S)$.
3. Construction of the regular expression describing the language $L(M') = M'^*$
4. Evaluation whether $S \subset L(M')$. 
Individually, each step including the evaluation can be done in polynomial time \[9\]. However, there are \(2^{|T(S)|}\) subsets of \(T(S)\) which means that an exponential number of repetitions for steps 2 to 4 is typically necessary. Note that the selection procedure might be modified to pick a promising candidate with a high probability. However, in our view, it is difficult to obtain useful information from the knowledge that a given set of substrings is not a string cover.

3.4.2 Existing IP Algorithm

Hermelin et. al. \[8\] proposed an IP formulation to the MSC. Their formulation was used to prove a non approximability result. We are not aware of its application as an efficient algorithm. Nevertheless, we wanted to use this algorithm for the purpose of comparison with our algorithms. Their formulation employs the set of all factorizations of string \(s\) with a maximum of \(l\) occurrences. Using our terminology, we denote this set as \(\mathcal{F}_l(s)\) and the set of all factorizations in \(S\) as \(\mathcal{F}_l(S) = \bigcup_{s \in S} \mathcal{F}_l(s)\). The IP then uses the following variables and constraints:

1. Each substring \(t_j \in T(S)\) with \(j \in \{1, \ldots, |T(S)|\}\) is represented by a binary variable \(x_j \in \{0, 1\}\). Substring \(t_j\) is included in the minimum string cover if and only if \(x_j = 1\). The same variable representation is also used in our IP and MIP formulations, see Table 2.1.

2. Each factorization \(D \in \mathcal{F}_l(S)\) is represented by a binary variable \(y_D \in \{0, 1\}\). \(D\) is a chosen factorization if and only if \(y_D = 1\).

3. For every string \(s_i\), there exists the inequality \(\sum_{D \in \mathcal{F}_l(s_i)} y_D \geq 1\).

The original paper specified the next constraint to be \(\sum_{t_j \in D \in \mathcal{F}_l(S)} y_D \leq x_j\) for all substrings \(t_j\) and factorizations \(D\). Since in our terminology a factorization consists of occurrences, this constraint requires a little modification. Basically, the constraint ensures that if a certain factorization \(D\) is used to cover the corresponding string then all substrings included in \(D\) must be part of the minimum string cover. Therefore, we adjust the constraint as follows:

4. For every factorization \(D \in \mathcal{F}_l(S)\) we use the constraint

\[\sum_{(i,p,q) \in D} x_j \geq y_D,\]

with \(i \in \{1, \ldots, |S|\}, j \in \{1, \ldots, |T(S)|\}, p, q \in \{1, \ldots, |s_i|\}, p \leq q,\) and \(t_j = s_i(p, q)\).

5. Finally, the total weighted sum of all substrings must be minimized:
\[
\min \left( \sum_{t_j \in T} w(t_j) \cdot x_j \right).
\]

In the paper of Hermelin et. al. \cite{8}, \( l \) is regarded as a constant and ignored when discussing complexity. But for an implementation, we must consider

\[
\sum_{s \in S} \sum_{k=0}^{l-1} \binom{|s|}{k}
\]

factorizations. Furthermore, in order to properly compare the various formulations, we had to set \( l = \max_{s \in S} (|s|) \) and thus created \( 2^{|s| - 1} \) factorization variables for each string \( s \).

### 3.4.3 New IP and MIP Algorithms

Our algorithms were implemented in C++. The MIPs and IPs were modeled using the ABACUS framework (A BRanch And CUt System), developed by M. Jünger and S. Thienel \cite{5} \cite{12}. Basically, ABACUS provides default implementations and interfaces of basic activities of integer linear programming such as LP-solving, bounding and cutting. The user is only required to implement constraint and variable classes derived from the abstractions of ABACUS. For our IP and MIP formulations, we simply used Tables 2.1 and 2.2, and applied a default branching strategy of ABACUS. After solving the corresponding LP, this strategy determines the binary variable with its value being closest to 0.5 and fixes this value to 1 and 0 in the respective child nodes of the branching tree.

### 3.4.4 Branch and Bound Algorithm using Index Graphs

We implemented Algorithms 2 to 10 of Section 2.1. As already mentioned we have identified two approaches of determining lower bounds. The first approach computes the shortest path only once while second approach iteratively reweights the edges and recomputes the shortest paths until the lower bound no longer increases, see Algorithm 2. For the branching, we only employed our hybrid strategy, see Algorithm 6. We did not distribute the index graphs among the computers and computed the shortest path for each index graph successively.

### 3.5 MSC Results

We present our results in two-dimensional plots. The y-axis represents the performance of an algorithm, that is the runtime in our case. The x-axis uses a number that is best suited to represent a test instance. We decided to use the total number of symbols in all strings for this purpose. We are aware that the size of an instance does not solely determine the
runtime, as there are easy large instances and difficult small instances. However, we felt that combining input composition and input size into a representative scalar is difficult and may result in a plot that is not intuitive.

### 3.5.1 Brute Force for MSC

We checked the correctness of our implementation for some very small instances. Even for those instances with sizes of about 40 symbols, the computation time was more than an hour. The regular expression of the language $L(M')$ gave no feedback regarding the quality of the current solution, even if the minimum string cover was detected by chance. The experiments showed that a simple brute force approach is not applicable in practical situations. Furthermore, we could not identify a simple method to efficiently use previous results in the selection of the next candidate.

### 3.5.2 Index Graphs for MSC

Our basic branching strategy is already described in Chapter 2.1.4, see Algorithm 6. Compared to both IP formulations (for an exemplary comparison with the $IP_{MSC}$, see Figure 3.1) this approach was far more effective. Though there were occasional instances for which the index graphs based algorithms would not almost immediately find a solution, the performance was always better than the corresponding IP or MIP. Sometimes, the IP or MIP algorithm was not even able to find a solution in a reasonable period of time.

![Figure 3.1: Performance of $IP_{MSC}$ (green) and index graph-based branch and bound (red).](image-url)
Also, the slightly more problematic instances always had an optimal solution consisting of exactly one substring, which would be highly improbable in a practical application. We already explained in Section 2.1.1 why these instances pose the greatest difficulties for the index graphs. A further analysis is given in Section 3.5.4.

Surprisingly, even as we increased the number of symbols, the optimization time increased very little. Without any visible observation of a runtime curve, we applied a regression analysis. But this analysis hardly provided any new or useful insights. At about 4300 symbols, we could no longer fit the index graphs into our main memory during preprocessing. Due to permanent swapping, we had to abort the optimization. Approximately 15% of all instances took more than 10 seconds to compute and roughly 5% took more than 50 seconds. However, the instances with an optimization time of more than 50 seconds were large at sizes of 2000 and more symbols.

The performance of the branch and bound algorithm with an iterative lower bound computation was mixed compared to the described basic approach. It is clear that for every visited node of the branching tree, the lower bound had to be computed at least twice. However, we had hoped that the improved bounds would permit us to bound earlier, thereby improving running time. While we often obtained better lower bounds and achieved an equal and sometimes even slightly better performance, there were also a few cases where multiple iterations increased the running time to an unacceptable amount, see Figure 3.2. On average, the iterative lower bound optimization took between 2 and 3 times as long as the basic approach and even without the outliers of either algorithm, the normal index graphs was about 1.5 times faster. Our basic algorithm was generally more dependable and therefore preferable. However in the future, we may consider to apply a hybrid approach by limiting the number of iterations to improve the lower bound.

### 3.5.3 IP and MIP for MSC

As in the brute force case, we verified our implementation of the IP from the literature [8] on some very small instances. However, when we applied this approach to slightly larger instances, the time required to create all variables and constraints was unreasonably long and we had not enough memory to start the optimization without extensive swapping.

Then we tested the new variants $IP_{MSC}$ and $MIP_{MSC}$. In either case, the branching strategy consisted of determining the binary variable closest to 0.5 and fixing it to either 1 or 0 as suggested by ABACUS, see Section 3.4. Although $IP_{MSC}$ has significantly more candidates for the branching strategy than $MIP_{MSC}$ which did not use binary occurrence variables, the run time results were similar, see Figure 3.3. Neither $IP_{MSC}$ nor $MIP_{MSC}$ could compete with the index graph based branch and bound method.

Additionally, we also performed a regression analysis on the runtimes of $IP_{MSC}$ and $MIP_{MSC}$. According to the graphic representation in Figure 3.3, we assumed an exponen-
Figure 3.2: Performance of the iterative lower bound computation (purple) and the normal lower bound computation (red). Generally, they have similar results, but in some cases, the iterative computation takes a very long time.

The regression curve for $IP_{MSC}$ and $MIP_{MSC}$ resulted in $f(x) = 8.01600365 \cdot 1.005472842^x$ and $f(x) = 8.016484624 \cdot 1.005458021^x$, respectively, see Figure 3.4.

3.5.4 Result Assessment

The index graphs had a clear advantage over any IP or MIP formulation. On the one hand, our branching strategy is based on substrings annotating the edges of the shortest paths, see Algorithm 6. We can informally say that these substrings are binary valued as no edge is only partly included in a shortest path. The relaxed LPs of $IP_{MSC}$ and $MIP_{MSC}$, on the other hand, produce variables with values in the interval $[0, 1]$. Hence, the index graphs are able to use problem specific information in the branching strategy with immediate effects, while the IPs must resort to a generic strategy.

We also examined special instances for which the index graphs took a relatively long time to compute compared to inputs of similar size. These instances contained a lot of repetition, which led to a bad performance of our lower bound, see Section 2.1.1. Recall that our branching strategy assigns each substring $t_j$ the value $|t_j| \cdot |C_j|/w(t_j)$ with $j \in \{1, \ldots, |T(S)|\}$ and accordingly chooses the substring with the highest value, see Algorithm 6. In the special instance in which an optimal solution consists only of one element, this value favors longer substrings over shorter ones. We demonstrate this by a
3.5. MSC RESULTS

Figure 3.3: Performance of \( \text{IP}_{\text{MSC}} \) (green) and \( \text{MIP}_{\text{MSC}} \) (blue). The red dots near the x-axis denote the comparative performance for the branch and bound algorithm based on index graphs.

short example. Let \( \{\alpha\} \) be a minimum string cover set of strings \( \mathcal{S} \). For each concatenation of \( \alpha = \alpha^1, \alpha^2, \ldots, s = \alpha^n \), we count the number of occurrences of \( \alpha^p \) with \( p \in \{1, \ldots, m\} \) in string \( s \). For \( 1 \leq p \leq m \), \( \alpha^p \) occurs exactly \( m - p + 1 \) times. Normalizing the string lengths in the computed value of the branching strategy yields the value function \( v(p) = p \cdot \left( \sum_{i=1}^{\mid\mathcal{S}\mid} m_i - p + 1 \right) \) with \( \alpha^m_i = s_i \). This function can be transformed into the form \( v(p) = -\mid\mathcal{S}\mid \cdot p^2 + p \cdot \left( \sum_{i=1}^{\mid\mathcal{S}\mid} m_i + 1 \right) \). We differentiate \( v \) with respect to \( p \) and obtain \( v'(p) = -2 \cdot \mid\mathcal{S}\| \cdot p + \left( \sum_{i=1}^{\mid\mathcal{S}\mid} m_i + 1 \right) \). For \( v'(p) = 0 \), the best value for \( p \) regarding our weighting function is \( p_{\text{max}} = \left( \sum_{i=1}^{\mid\mathcal{S}\mid} m_i + 1 \right) / (2 \cdot \mid\mathcal{S}\mid) \). Unless \( \{\alpha^q\} \) with \( 1 < q < 2 \cdot p_{\text{max}} \) is optimal, our branching strategy only finds the solution after \( 2 \cdot p_{\text{max}} - 1 \) iterations. Therefore, an instance with a few strings and a large amount of repetition was the most difficult one for the index graphs to solve.

In addition to our MSC-instances, we also tested the index graphs and \( \text{MIP}_{\text{MSC}} \) on strings consisting of uniformly distributed symbols. Here, we alternated the (expected) minimum string covers between the input strings \( \mathcal{S} \) and the input alphabet \( \Sigma \) by generating sufficiently long sequences with either \( \mid\Sigma\| = \mid\mathcal{S}\| - 1 \) or \( \mid\Sigma\| = \mid\mathcal{S}\| + 1 \). In the former case, the input consisted of 1330 symbols while the other case had inputs of 1490 symbols. Again the index graph based algorithms rapidly found the optimal solution always in less than 2 seconds. For these instances, the optimization time for \( \text{MIP}_{\text{MSC}} \) was on an average 700 to 800 seconds which was less than the 10300 second average for our instances.
CHAPTER 3. EXPERIMENTAL EVALUATION

Figure 3.4: Regression Analysis of $IP_{MSC}$ and $MIP_{MSC}$. There is little visible difference between the regression curves for the runtime function of either formulation.

of comparable length and with non-trivial optimal solutions. But still took more than 500 times longer to compute than the index graph based algorithms, see Figure 3.5.

3.6 MPSC Results

Due to time constraints, the MPSC was not tested and analyzed in the thorough manner of the MSC. The input was generated such that there were no more gaps than 5% of the total number of symbols. Despite the smaller input size, we can still see that the index graphs with basic lower bound computation outperformed $IP_{MPSC}$, see Figure 3.6. Considering the results of MSC, it is likely that this trend continues for larger input.

3.7 Gene Clustering

Genes encode the information of all proteins that are synthesized in an organism [1]. This genetic information is part of an organism’s DNA sequence also known as genome. In addition, there are non-encoding parts in the genome. A human genome contains about 25000 genes which account for 1.5% of the entire DNA sequence, though the exact number of genes is not known. Parts of the genomes of closely related species have similar regions known as conserved regions. Therefore, applying the MSC on multiple genomes could help
3.7. GENE CLUSTERING

Figure 3.5: This figure compares the runtime of $IP_{MSC}$ and the index graph based algorithms for strings with uniformly distributed symbols. The runtime for the index graph based algorithms was only valid for the first decimal digit.

Figure 3.6: The index graph based algorithms (red points) always computed the minimum partial cover faster than the $IP_{MSC}$ for instances of MPSC with more than 300 symbols.
in finding recurring sequences of genes. Note that the conserved regions do not solely contain genetic information.

We set up an MSC optimization for a set of strings representing 46 bacteria genomes containing a total of 3851 different genes. The bacteria genomes are part of the COG (Clusters of Orthologous Groups of proteins) database [18] and are described with more details in a paper by Tatusov et. al. [20]. The minimum string cover was supposed to contain recurring gene sequences. Note that non-protein encoding DNA was also included in the genomes.

The index graph based branch and bound algorithm for the MSC only found the trivial solution consisting of the 46 genomes. Then we had the option of either reducing the maximum length of strings appearing in a minimum cover or modifying the input. Since the non-protein encoding fragments were of little interest, a MPSC run with as many gaps as non-protein encoding fragments appearing in the genomes is a possible approach. However, we decided against this approach for two reasons. Firstly, we already knew the gap positions prior to optimization which made it unnecessary to have our algorithms find them similar to applying error correction when only erasures must be considered. Secondly, the number of non-protein encoding fragments occurred more often than any gene. Hence, it was more probable that the MPSC would have deleted the recurring substrings of genes we were interested in rather than the non-protein encoding fragments. Therefore, we deleted these fragments from the input and added a new string for every substring between two fragments. Recall that this idea is also used in the proof of Theorem 23.

The new input now consisted of 29960 strings. We ran the index graph based branch and bound algorithm for the MSC again. This time, we only found the set of genes as a solution. As a counter-measure, we used MSC/FB and included all strings consisting of a single gene in the set of forbidden substrings. However, the MSC/FB run did not produce a solution as some strings of the input consisted of a sole gene. This example demonstrates that a significant amount of tuning is required to apply MSC or its related problems to real problems in computational biology.

Furthermore, we believe that a gene cluster is not a suitable model for the standard MSC as the DNA sequences contained too much randomness. Abelian patterns may have produced better results but they had not been implemented at this time. But even a MAC solution is unlikely to produce the desired result without a careful preparation of the input data and the algorithmic parameters. Such a preparation requires expertise and was beyond the scope of this thesis.
Chapter 4

Conclusion and Future Work

4.1 Conclusion

The Minimum String Cover Problem (MSC) is an established NP-complete problem. However, to our knowledge, there are no algorithms that efficiently find solutions for most instances. In this thesis, we address this problem and suggest two new solution methods. Our study is not restricted to the MSC but also includes the three variants partial string covers (MPSC), forbidden substrings (MSC/FB) and Abelian patterns (MAC). Our first result is a method to calculate a lower bound computation for the value of an MSC solution by using a DAG representation for a set of strings and determining shortest paths in this graphs. Such a shortest path corresponds to a string cover and therefore also provides an upper bound for the solution. As this method also allows additional constraints, we introduce developed a branch and bound approach. Our branching strategy is based on the edges chosen in the detected string cover. Additionally, we showed how the graphs can be modified to solve our extended problem variants. Furthermore, we presented an IP formulation for the various problem definitions and proved its correctness. To increase our flexibility, we transformed the IP into an MIP by relaxing some variable conditions. In addition to the theoretical analysis of our algorithms, we tested them on a significant amount of test data for a wide range of string sizes. As reference, we used a simple brute force approach based on regular expressions and an IP formulation that was used in the literature to prove a non approximability result. We established a relationship between the IP and MIP solutions. The graph based branch and bound algorithm clearly outperformed our IP and MIP approaches while the reference algorithms could only be applied to small problem sizes.
4.2 Future Work

We are not aware of any attempt to use the MSC model in a practical application. The Gene Cluster experiment described in Section 3.7 and our attempts to adjust our problem parameters to find meaningful results confronted us with a number of difficulties. Based on the information obtained during our experimental evaluation, we identified a few problems that may be promising candidates for MSC application.

There are two main directions of MSC application. Firstly, we may use MSC to find recurring strings. To a certain extent, such an identification can also be done by various other algorithms and data structures in polynomial time. However, the "cover" aspect of the MSC also emphasizes the importance of a substring as a structural part of $S$. Therefore, the MSC and its variants MPSC, MSC/FB, and MAC can be used in an identification scenario where the observation of such structural properties of $S$ is also relevant.

Secondly, we may desire a compact representation. Apart from structural insights that can be possibly gained by such a representation, we can also reduce the amount of memory required to store $S$, provided the storage cost of $M$ is less than the storage cost of $S$ or the alphabet $\Sigma$. This can be used in data storage or business process models.

Protein Domains

A protein consists of a sequence of 20 amino acids [1]. Large parts of the sequence form distinguished units called domains. A domain determines function and characteristics of the protein. The domain sizes vary greatly but usually contain between 40 and 350 amino acids. Although $20^{40}$ different amino acid sequences of length 40 are theoretically possible, the number of domains and proteins thus far encountered is far less. The small number of found domains may be due to evolution. Even small alterations to a domain’s structure can render it useless and the protein no longer fulfills its function. Therefore, this protein is unlikely to appear in future generations. Contrary to the rate of genes in a genome, much of a protein’s amino acid sequence is covered by domains with short bits of polypeptide chain connecting the domains.

Therefore, the following MSC type model may be applicable for finding the protein domains:

- The input string set $S$ consists of multiple proteins. The proteins should be related to share as many domains as possible.

- A unit weight function $w$ is applied as all domains included in the substring are searched for, regardless of length and composition.

- Substrings of less than 40 and more than 350 amino-acids can be disregarded.
4.2. FUTURE WORK

- The number of gaps is derived from an upper estimation of expected number of amino acids in a protein not belonging to a domain. The gaps are only considered for each protein (string of $S$) individually with no global maximum.
Appendix A

Additional Information

A.1 Glossary

<table>
<thead>
<tr>
<th>notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>the input string set</td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td>the input alphabet</td>
</tr>
<tr>
<td>( w : \Sigma^* \mapsto \mathbb{R} )</td>
<td>weight function, omitted if ( \Sigma^*.w(\alpha) = 1 ) for all ( \alpha \in \Sigma )</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>the minimum string cover</td>
</tr>
<tr>
<td>( s_i )</td>
<td>the ( i^{th} ) string of ( S )</td>
</tr>
<tr>
<td>((i, d))</td>
<td>the position corresponding to the ( d^{th} ) symbol of the string ( s_i )</td>
</tr>
<tr>
<td>( s_i(p, q) )</td>
<td>the substring from position ( p ) to ( q ) of string ( s_i )</td>
</tr>
<tr>
<td><strong>T(S)</strong></td>
<td>the set of all substrings of ( S )</td>
</tr>
<tr>
<td>( t_j )</td>
<td>the ( j^{th} ) element of ( T(S) )</td>
</tr>
<tr>
<td>((i, p, q))</td>
<td>the substring occurrence from position ( p ) to ( q ) of string ( s_i )</td>
</tr>
<tr>
<td><strong>C(S)</strong></td>
<td>the set of all occurrences in ( S )</td>
</tr>
<tr>
<td>( C_j )</td>
<td>the set of all occurrences ( (i, p, q) ) with ( s_i(p, q) = t_j \in T(S) )</td>
</tr>
<tr>
<td>( C(i, d) )</td>
<td>the set of all occurrences containing position ( (i, d) )</td>
</tr>
<tr>
<td>( A(T(S)) )</td>
<td>the set of Abelian patterns for all substrings of ( S )</td>
</tr>
<tr>
<td>( g )</td>
<td>the maximum number of gaps in all strings of ( S )</td>
</tr>
<tr>
<td>( g_i )</td>
<td>the maximum number of gaps in string ( s_i )</td>
</tr>
</tbody>
</table>

**General**

**IP and MIP**

| \( x_j \) | the binary-valued variable corresponding to \( t_j \) |
| \( y_{i,p,q} \) | the variable corresponding to \( (i, p, q) \) |
| \( z_{i,d} \) | the variable corresponding to \( (i, d) \) |

**Index Graphs**

| \( G_i(V_i, E_i) \) | the index graph for \( s_i \) |
| \( G = (V, E, f) \) | the index graph family with weight function \( f \) |
| \( E(j) \) | the set of all edges annotated with substring \( t_j \) |
| \( b_i \) | the shortest path in \( G_i \) from \( (i, 0) \) to \( (i, |s_i|) \) |
Bibliography


Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den November 10, 2010

Chris Schwiegelshohn