Solving the Minimum String Cover Problem*

Stefan Canzar† Tobias Marschall† Sven Rahmann‡ Chris Schwiegelshohn§

December 12, 2011

Abstract

A string cover C of a set of strings S is a set of substrings from S such that every string in S can be written as a concatenation of the strings in C. Given costs assigned to each substring from S, the Minimum String Cover (MSC) problem asks for a cover of minimum total cost. This NP-hard problem has so far only been approached from a purely theoretical perspective. A previous integer linear programming (ILP) formulation was designed for a special case, in which each string in S must be generated by a (small) constant number of substrings. If this restriction is removed, the ILP has an exponential number of variables, for which we show the pricing problem to be NP-hard. We propose an alternative flow-based ILP formulation of polynomial size, whose structure is particularly favorable for a Lagrangian relaxation approach. By making use of the strong bounds obtained through a repeated shortest path computation in a branch-and-bound manner, we show for the first time that non-trivial MSC instances can be solved to provable optimality in reasonable time. We also provide and solve real-world instances derived from the classic text "Alice in Wonderland". On almost all instances, our Lagrangian relaxation approach outperforms a CPLEX-based implementation by an order of magnitude. Our software is available under the terms of the GNU general public license.

1 Introduction.

Let S be a set of strings. We call a set of substrings of the strings in S a *cover* of S if concatenations of these substrings generate the original strings. In the unweighted *Minimum String Cover* (MSC) problem, we want to find a cover with minimal cardinality. In a more general version, we assign a cost to each substring and aim at a cover of minimal total cost.

The paper is organized as follows. First, we briefly review the history of the problem and define the problem formally. After that, we discuss an existing integer linear programming (ILP) formulation by Hermelin et al. [4] of exponential size (Section 2) and present a new polynomial-size flow-based formulation in Section 3. Additionally, in Section 4, we show that a certain Lagrangian relaxation of our formulation leads to a shortest path problem in a directed acyclic graph associated with the strings of the problem instance. These properties result in the first practical method to solve non-trivial MSC instances. We describe our implementation (Section 5) and evaluate it on benchmark instances (Section 6). A brief discussion concludes the paper (Section 7).

1.1 Previous Work.

Bodlaender et al. [2] used the name *Dictionary Generation* for MSC, because in computer linguistics, it is a common task to find words, stems, suffixes and affixes, or syllables from text corpora with unknown structure, and MSC thus might complement language-specific stemming algorithms by discovering these building blocks (semi-)automatically. Bodlaender et al. [2] also suggested that MSC might be applicable to discover protein domains from collections of protein sequences. Furthermore, MSC may be relevant to data storage as an MSC optimization yields a compact representation of a string set S. Despite these potential applications, in none of the areas mentioned above, real problems are solved with MSC algorithms. This might be due to the lack of efficient and practical algorithms for MSC, as previous work has mostly addressed theoretical aspects of MSC.

NP-completeness of the unweighted MSC problem was determined in 1990 by Néraud [6], who showed that it is co-NP-complete to decide whether a given set of strings is elementary. A set of strings X is elementary if there exists no set of strings Y with $|Y| < |X|$, such that the strings in X can be written as concatenations of the strings in Y. For example, $\{ABC, BCA\}$ is elementary, but $\{ABC, BCA, A\}$ is not because these strings can be

*Supported by DFG SFB 876 "Providing Information by Resource-Constrained Data Analysis"

†Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, Netherlands.

‡Genome Informatics, Faculty of Medicine, University of Duisburg-Essen, and Bioinformatics, Computer Science XI, TU Dortmund, Germany.

§Algorithms and Complexity Theory, Computer Science II, TU Dortmund, Germany.

written as concatenations of strings in the smaller set $\{A, BC\}$.

Hermelin et al. [4] generalized the unweighted MSC problem by assigning costs to each substring and showed that approximating the solution within factors of $c \cdot \ln |S|$ and $\lfloor \max_{s \in S} |s|/2 \rfloor - 1 - \epsilon$, for some $c > 0$ and for all $\epsilon > 0$, is NP-hard. In addition, the authors presented an ILP formulation (see Section 2) and two approximation algorithms based on dynamic programming and LP rounding, respectively.

The ILP formulation appears to never have been implemented, and we are not aware that any other algorithm has ever been used to solve real instances either. In summary, no practical exact method exists yet to solve non-trivial MSC instances. The purpose of this article is to provide such a method and to make a tool available for researchers working in the aforementioned domains.

1.2 Notation and Problem Definition. Given a finite alphabet Σ and a string $s \in \Sigma^*$, the length of s is denoted by $|s|$ and its characters are indexed starting at zero, i.e. $s = s[0] \dots s[|s| - 1]$.

Substrings are written $s[i \dots j] := s[i] \cdots s[j]$. The set of all (distinct) substrings of s excluding the empty string is denoted by $T(s)$. A string can contain the same substring more than once. Therefore, we distinguish between substrings and *intervals* of a string. While substring $t \in T(s)$ is a string from Σ^*, intervals are denoted by tuples (s, i, j) referring to the range from i to j in s. The set of all intervals of s is written $I(s)$. We write $I_t(s)$ to denote the set of all intervals that spell t in s, formally $I_t(s) := \{(s, i, j) \in I(s) : s[i \dots j] = t\}$.

A *factorization* of s is a sequence of intervals of s, $((s, i_1, j_1), \dots, (s, i_K, j_K))$ for some $K \geq 1$ with $0 = i_1$; $i_k \leq j_k$ for $k = 1, \dots, K$; $j_{k-1} + 1 = i_k$ for $k = 2, \dots, K$; and $j_K = |s| - 1$, so that the concatenation of the substrings spelled by the intervals is s. The set of all factorizations of s is denoted by $F(s)$. Note that $|F(s)| = 2^{|s|-1}$ is exponentially large in the string length. For a given $f \in F(s)$, we slightly abuse notation and define $T(f)$ to be the set of all substrings of s spelled by the intervals in f. Furthermore, $F_t(s) := \{f \in F(s) : t \in T(f)\}$ denotes the set of all factorizations containing the substring $t \in T(s)$ at least once.

Throughout this paper, S denotes a finite set of strings. The definitions of T, F and I naturally extend to sets of strings through $T(S) := \bigcup_{s \in S} T(s)$, etc.

A set of strings $C \subset T(S)$ is a *cover* of S if, for every $s \in S$, there exists a factorization $f_s \in F(s)$ such that $T(f_s) \subset C$.

PROBLEM 1. (MINIMUM STRING COVER) *For a given finite alphabet Σ, a finite string set $S \subset \Sigma^*$, and a cost function $w : T(S) \to \mathbb{Q}_0^+$, the* MINIMUM STRING COVER *problem consists of finding a cover C of S such that its total cost $w(C) := \sum_{t \in C} w(t)$ is minimal among all covers of S. The tuple (S, w) is called an* instance *of the* MINIMUM STRING COVER *problem, the underlying alphabet Σ being derived from S. If $w \equiv 1$, the problem is called the* unit cost *(also* unweighted*)* MINIMUM STRING COVER *problem.*

2 An Initial ILP formulation.

We briefly restate the ILP formulation introduced by Hermelin et al. [4]. For every substring $t \in T(S)$ we use a binary variable x_t indicating whether substring t is contained in the sought string cover C. For every string $s \in S$ and every factorization $f \in F(s)$, a binary variable $y_{s,f}$ indicates whether f is used to factorize s. Using these variables, the MINIMUM STRING COVER problem can then be cast as the following ILP $\mathrm{SC}_{\mathrm{fact}}$.

$$(\mathrm{SC}_{\mathrm{fact}}) \quad \min \sum_{t \in T(S)} w(t)\, x_t,$$

$$(2.1) \qquad \text{s.t.} \sum_{f \in F_t(s)} y_{s,f} \leq x_t \qquad \forall\, s \in S, t \in T(S),$$

$$(2.2) \qquad \sum_{f \in F(s)} y_{s,f} \geq 1 \qquad \forall\, s \in S,$$

$$(2.3) \qquad x_t,\ y_{s,f} \in \{0, 1\} \qquad \forall\, t \in T(S),\ s \in S, f \in F(s).$$

The first set of constraints (2.1) ensures that a factorization can be used to cover an input string only if all its substrings are contained in the solution cover C. Constraints (2.2) require that all strings are covered by at least one factorization.

ILPs are solved by commercial solvers by a repeated solution of the linear programming (LP) relaxation of variants of the problem. Therefore, besides the strength of the obtained bound, the ability to solve the LP relaxation efficiently plays a key role in the practical performance of an ILP-based approach. In $\mathrm{SC}_{\mathrm{fact}}$, however, the number of factorizations and thus the number of y-variables grows exponentially with the length of the strings. Previous work, including [4], therefore focused on the ℓ-cover problem, a variant in which each string must be produced by a concatenation of at most ℓ substrings, where ℓ is assumed to be constant. Thus, the number of factorizations is no longer exponential in the string length, and solving the ILP becomes feasible for reasonably small ℓ. For the general MINIMUM STRING COVER problem we consider here, solving this ILP directly is infeasible. Alternatively, one can study the pricing problem, or equivalently, the separation problem for the exponentially large class of constraints in

the dual of its LP relaxation. If the separation problem can be solved in polynomial time, then an optimal solution to the LP relaxation can still be found in polynomial time [7]. In this case, practically efficient algorithms for solving the LP relaxation based on delayed column generation, respectively cutting planes, might exist. However, we show next that an efficient algorithm for solving the separation problem for the dual of the LP relaxation of $\mathrm{SC}_{\mathrm{fact}}$, in which constraints (2.3) are replaced by constraints $x_t, y_{s,f} \geq 0$, is unlikely to exist. By the equivalence of separation and optimization [7] we conclude that there is no polynomial time algorithm to solve the LP relaxation, unless P=NP.

THEOREM 2.1. *The separation problem for the dual of SC_{fact} is $\mathcal{NP}$-hard.*

Proof. Consider the dual of the LP relaxation of $\mathrm{SC}_{\mathrm{fact}}$, taking into account that every optimal solution for the primal problem satisfies $x_t, y_{s,f} \leq 1$,

$$(\mathrm{DSC}_{\mathrm{fact}}) \quad \max \sum_{s \in S} p_s,$$

$$(2.4) \qquad \text{s.t.} \sum_{t \in T(f)} q_{s,t} \geq p_s \qquad \forall s \in S, \forall f \in F(s),$$

$$(2.5) \qquad q_{s,t} \leq w(t) \qquad \forall s \in S, \forall\, t \in T(s),$$

$$(2.6) \qquad p_s, q_{s,t} \geq 0 \qquad \forall s \in S, \forall\, t \in T(s).$$

For details on how to obtain the dual of a linear program, we refer to textbooks such as [1]. The separation problem consists of deciding whether a given vector (q, p) is feasible for $\mathrm{DSC}_{\mathrm{fact}}$ and, if not, to find a violated constraint. For constraints (2.5) and (2.6) this is a trivial task. For constraints (2.4) we have to decide for every string s, whether $\min_{f \in F(s)} \sum_{t \in T(f)} q_{s,t} \geq p_s$. Computing the left hand side of this inequality is equivalent to solving a minimum string cover instance with s as the only input string and costs given by $w'(t) := q_{s,t}$. Note that constraints (2.5) do not pose any restriction on this minimum string cover instance, since the costs $w(t)$ can always be scaled accordingly. The claim now follows from the following lemma. ∎

LEMMA 2.1. MINIMUM STRING COVER *with $|S| = 1$ is $\mathcal{NP}$-hard.*

Proof. Given an instance $I = (S, w)$ of the unweighted MINIMUM STRING COVER problem, i.e. $w \equiv 1$, with $S = \{s_1, s_2, \ldots, s_n\}$ and $n > 1$, we construct an equivalent instance $I' = (S', w')$ with $|S'| = 1$ as follows: We concatenate all strings in S to a single string in S', separated by a character \$, which we assume not to be present in S. Thus $S' = \{s_1\$s_2\$\cdots\$s_n\}$. We define the cost function w' as $w'(t) := 1$ for $t \in T(S)$, $w'(t) := 1+n$ for $t \in T(S') \setminus \{T(S) \cup \{\$\}\}$, and $w'(\$) := 0$. Clearly, the set $C = S \cup \{\$\}$ is a feasible solution for instance I' and thus n provides an upper bound on the cost of an optimal solution $\bar{C}^*$. Therefore, for every substring $t \in \bar{C}^*$ it holds $t = u\$v \Rightarrow u = \epsilon \wedge v = \epsilon$ and thus $\bar{C}^* \setminus \{\$\}$ is a feasible solution for I of same cost. In the reverse direction, we can derive from an optimal solution C^* for I a feasible solution C' for I' of same cost by simply setting $C' := C^* \cup \{\$\}$. ∎

3 A polynomial-size ILP formulation.

We propose a polynomial size ILP formulation for the minimum string cover problem. The idea is to model factorizations of a string by paths in the *substring graph* of a string, which we define in the following. In essence, its directed edges correspond to substring intervals, and the nodes to positions between characters. Formally, for a string s of length n, let

$$V_s := \{(s,0), (s,1), \ldots, (s,n)\},$$
$$E_s := \{((s,p) \to (s,q)) : p < q, (s,p) \in V_s, (s,q) \in V_s\}.$$

The directed edge $((s,p) \to (s,q))$ represents the substring interval $(s, p, q-1)$, spelling a substring of length $q-p$. From now on, we identify the interval (s,i,j) with the edge $(s,i) \to (s,j+1)$.

A factorization of s is now equivalent to a path in the substring graph $G_s = (V_s, E_s)$, starting at $(s,0)$ and ending at $(s,|s|)$. We write $\delta^-(v)$ and $\delta^+(v)$ for the sets of incoming and outgoing edges of $v \in V_s$, respectively.

Our ILP formulation $\mathrm{SC}_{\mathrm{flow}}$ uses a binary variable $z_{s,i,j}$ for every edge, i.e. interval $(s,i,j) \in I(S)$, and models a path from the source $(s,0)$ to the sink $(s,|s|)$ as a unit flow. Let $V_s^{\pm} := V_s \setminus \{(s,0), (s,|s|)\}$.

$$(\mathrm{SC}_{\mathrm{flow}}) \qquad \min \sum_{t \in T(S)} w(t)\, x_t\,, \quad \text{s.t.}$$

$$(3.7) \qquad z_{s,i,j} \leq x_{s[i\ldots j]} \qquad \forall\, (s,i,j) \in I(S),$$

$$(3.8) \qquad \sum_{(s,i,j) \in \delta^+((s,0))} z_{s,i,j} = 1 \qquad \forall\, s \in S,$$

$$(3.9) \qquad \sum_{(s,i,j) \in \delta^-(v)} z_{s,i,j} = \sum_{(s,i,j) \in \delta^+(v)} z_{s,i,j} \qquad \forall\, s \in S, v \in V_s^{\pm},$$

$$(3.10) \qquad x_t,\ z_{s,i,j} \in \{0,1\} \qquad \forall\, t \in T(S),\ (s,i,j) \in I(S).$$

Note that constraints (3.8) and (3.9) together imply that a unit flow arrives at the sink $(s,|s|)$. We also call (3.9) the "flow balance constraints". We solve $\mathrm{SC}_{\mathrm{flow}}$ using a Lagrangian relaxation approach, as described in the next section.

4 Lagrangian Relaxation.

The ILP formulation derived in the previous section exhibits a structure that is favorable for a Lagrangian relaxation approach. The general idea of Lagrangian relaxation is to relax the "complicating" constraints and penalize their violation in the objective function, such that an "easy-to-solve" subproblem remains. In our case, solutions satisfying constraints (3.8)–(3.9) encode a unit flow and thus a path from node $(s,0)$ to node $(s,|s|)$ in the substring graph of every string $s \in S$. The link between these paths and the chosen substrings is established through constraints (3.7). Therefore, by relaxing these "linking constraints" and by penalizing their violation with non-negative multipliers λ in the objective function, an optimal solution to the resulting problem can be obtained by computing shortest paths in the substring graphs independently for each string.

$$(LR_\lambda) \quad \min \sum_{t \in T(S)} w(t)\, x_t + \sum_{(s,i,j) \in I(S)} \lambda_{s,i,j}(z_{s,i,j} - x_{s[i\ldots j]}),$$

such that

$$\begin{aligned} \sum_{(s,i,j) \in \delta^+((s,0))} z_{s,i,j} &= 1 && \forall\, s \in S, \\ \sum_{(s,i,j) \in \delta^-(v)} z_{s,i,j} &= \sum_{(s,i,j) \in \delta^+(v)} z_{s,i,j} && \forall\, s \in S, v \in V_s^{\pm}, \\ x_t,\ z_{s,i,j} &\in \{0,1\} && \forall\, t \in T(S), \\ & && (s,i,j) \in I(S). \end{aligned}$$

We denote the problem of finding an optimal solution to this Lagrangian relaxation for given Lagrangian multipliers $\lambda_{s,i,j} \geq 0$ by (LR_λ), and its optimal cost by $v(LR_\lambda)$.

The shortest paths are computed with respect to weights $\lambda_{s,i,j}$ assigned to the z-variables in the objective function. We set $z_{s,i,j} = 1$ if the edge from node (s,i) to node $(s,j+1)$ lies on the shortest path from $(s,0)$ to $(s,|s|)$ in the substring graph of s and $z_{s,i,j} = 0$ otherwise.

Since the x-variables are not further constrained in (LR_λ), we simply set $x_t = 1$ if its associated coefficient in the objective function is non-positive, and $x_t = 0$ otherwise:

$$(4.11) \qquad x_t = \begin{cases} 1 & \text{if } w(t) - \sum_{(s,i,j) \in I_t(S)} \lambda_{s,i,j} \leq 0, \\ 0 & \text{otherwise.} \end{cases}$$

If $\lambda_{s,i,j} \geq 0$ for all $(s,i,j) \in I(S)$, then the terms in the second sum in objective function (LR_λ) are all non-positive if the solution is also feasible for the original problem formulation $\mathrm{SC}_{\mathrm{flow}}$ and in particular if it satisfies constraints (3.7). Since every solution feasible for the original problem formulation $\mathrm{SC}_{\mathrm{flow}}$ is also feasible for the Lagrangian relaxation (LR_λ), $v(LR_\lambda)$ provides a lower bound on the optimal cost of problem $\mathrm{SC}_{\mathrm{flow}}$. Naturally, we are interested in strong bounds on the optimal cost in order to be able to prune large parts of the solution space during implicit enumeration performed by branch-and-bound approaches. Hence we want to determine multipliers $\lambda^*_{(s,i,j)} \geq 0$ such that the cost of an optimal solution to the Lagrangian relaxation is as large as possible. This problem is referred to as the Lagrangian dual problem:

$$(\mathrm{LD}) \qquad \lambda^* = \operatorname*{argmax}_{\lambda \geq 0} v(LR_\lambda)$$

Since the *Lagrangian function* $f(\lambda) = v(LR_\lambda)$ is a concave function in λ but not differentiable at points where the optimal solution to (LR_λ) is not unique, a commonly used approach [3] to determine near-optimal multipliers efficiently is based on the vector of *subgradients* associated with a given λ. A subgradient at a point λ^0 is given by the vector of slacks of the dualized constraints (3.7) given an optimal solution to (LR_{λ^0}). The iterative approach proposed by Held and Karp [3] generates a sequence of Lagrangian multipliers $\lambda^0, \lambda^1, \ldots$ by taking at iteration $k+1$ a step in the direction opposite to a subgradient of $f(\lambda^k)$, projecting the resulting point onto the non-negative orthant. We refer to [3] for details on this approach.

5 Implementation.

Using the subgradient optimization approach described in the previous section, convergence towards the optimal Lagrangian multipliers can be slow in practice. Therefore, we opt for near-optimal multipliers and employ the resulting lower bounds in a branch-and-bound (b&b) framework to efficiently find the global optimum. Generally, we proceed as described in [5]. In the remainder of this section, we provide the algorithmic details needed to reproduce our results.

A node in the b&b tree represents a set of substrings that must be *included* into the solution and a set of substrings that are *forbidden*. Furthermore, it contains the current Lagrangian multipliers. Branching at a specific substring means cloning the current node into two child nodes and including the given substring in one while forbidding it in the other. Included substrings can be taken into account while solving the Lagrangian dual problem by forcing $x_t = 1$ for every included substring t, while setting the multipliers of the corresponding intervals to zero. To respect forbidden substrings, the respective edges in the substring graphs are deleted.

As the nodes of substring graphs are ordered by

construction, we know a topological sorting and can solve the shortest path problem for every $s \in S$ by straightforward dynamic programming. The sum of lengths of the shortest paths over all $s \in S$ yields a lower bound as explained in Section 4. By selecting each substring being part of a shortest path from the source node to the sink node, we obtain a feasible solution and therefore an upper bound on the optimal cost. In each iteration, we check whether this feasible solution improves the best one known so far and, if so, store it.

Finally, we need to address the questions of node and variable selection. That is, we have to decide at which string to branch for a given branching node and in which order to process the nodes of the b&b tree. To choose the substring to branch on, we consider the multipliers associated with the edges on the shortest path computed in a given branching node. For each substring $t \in T(S)$, we sum the multipliers of the selected edges and divide by the total sum of multipliers for all edges associated with the substring, that is, we compute the quantity

$$r_t := \frac{\sum_{(s,i,j)\in I_t(S)} \lambda_{s,i,j} \cdot z_{s,i,j}}{\sum_{(s,i,j)\in I_t(S)} \lambda_{s,i,j}}.$$

Intuitively, the ratio r_t is a measure for whether the substring t should be included in the final solution or not. We then branch at the substring for which this ratio is closest to 0.5 meaning that we are "most uncertain" whether to include it or not. We keep all branching nodes that have not yet been considered in a priority queue and process them following a *best-node first* strategy which aims to minimize the total number of nodes evaluated in the tree [5]. According to this strategy, always the node with the lowest lower bound, i.e. the node which potentially permits the best solution, is chosen.

The performance of the subgradient optimization can strongly be influenced by the choice of the initial multipliers. We set them as

$$\lambda_{s,i,j} := \frac{w(s[i \dots j])}{|I_{s[i\dots j]}(S)|}.$$

These initial multipliers have the special property that all coefficients of x variables in the objective function (LR_λ) become zero. Therefore, the lower bound is solely determined by the sum of lengths of the shortest paths. If all edges belonging to a substring are chosen, then the complete weight of this substring contributes to the lower bound. Furthermore, these multipliers encourage the use of substrings which occur frequently and/or have low weight. This, intuitively, is beneficial for obtaining a good initial feasible solution.

In the subgradient optimization approach, the size of the step taken in the direction opposite to a subgradient (see Section 4) is controlled by a parameter μ. Concerning its adaption, our approach slightly differs from the classical Held-Karp method [3]. For each branching node, μ is initially set to 2.0 and halved after five iterations in which the lower bound has not been improved or after 15 iteration in which the gap between lower and upper bound has not been reduced by at least 1 %. If μ reaches 0.125 before lower and upper bound meet, we branch. In the branching tree's root node, we invest more effort in computing strong bounds: we decrease μ after 30 non-improving iterations or 90 iterations not reducing the gap by at least 1 % and iterate until it reaches 0.001.

6 Evaluation.

Despite the theoretical considerations in Sections 2 to 4, only experiments can show which approach works best in practice. Until now, however, no practical approach to solve minimum string cover existed and hence no benchmark data sets are publicly available. Therefore, we generate benchmark data sets by random sampling and using the sentences of a novel.

The purpose of Section 6.1 is to provide guidance as to which kind of problem instances (in terms of alphabet size, input size, solution size, etc.) can be solved to provable optimality in reasonable time by our implementation, and to compare the performance of the commercial general-purpose ILP solver CPLEX to our Lagrangian-based b&b approach. In contrast, the purpose of Section 6.2 is to attempt to model a real-world problem (word boundary detection in an English text) with MSC, using an appropriate cost function.

Our software has been implemented in C++ and was compiled using GNU gcc version 4.4.5. It is available under the terms of the GNU general public license at `http://string-cover.googlecode.com`. To solve the ILP introduced in Section 3 directly, the commercial general-purpose ILP solver CPLEX 12.2 (`http://www.cplex.com`) with Concert Technology has been used. Time measurements were taken on a compute cluster whose nodes are equipped with two Intel Quad-Core processors with clock-rates between 2.26 GHz and 2.5 GHz and 24 GB of RAM, running 64 bit Linux.

6.1 Random Instances. In order to compare both approaches in a controlled setting and to provide an overview of runtimes to be expected when facing the string cover problem in practice, we randomly generated a total of 1 800 instances divided into 36 groups (50 instances each). For every group, the instances were sampled using different parameters as detailed below.

We used three different alphabet sizes of 4, 20, and 50, the first two being inspired by the DNA and amino acid alphabets, respectively. To obtain instances with non-trivial solutions, we did not use completely random texts but sampled a solution set first and subsequently generated problem instances by randomly concatenating strings from the solution set. Sampling the solution set was controlled by two parameters, a range for the set size and a range for the string length. Within the given ranges, the set size and the length of each solution string was sampled uniformly. All characters in each solution string were drawn independently and uniformly. From the solution set constructed in this way, a prespecified number of strings were constructed by concatenating randomly drawn strings from the solution set. The length of each string was determined by sampling a lower bound for its length from an interval given as parameter; as long as this lower bound was not reached, another string was drawn from the solution set and appended. All used parameter values are summarized in Table 1. The groups of parameters are obtained by considering all combinations excluding those where the number of strings in the solution set would be larger than the number of generated strings.

For instances with alphabet size four and twenty, we introduced an additional constraint restricting the minimum length of a string in the solution to three and two, respectively. This allows avoiding trivial solutions containing just the input alphabet. For this benchmark, we considered only the unit weight case because choosing an appropriate weight function greatly depends on the specific application. In Section 6.2, we then consider one specific example of a problem instance with an application-tailored weight function. All instances were (attempted to be) solved with CPLEX and our Lagrangian relaxation approach within a time limit of 1h and using up to 8 GB of memory. If either the time or the memory limit was exceeded, the computation was aborted. CPLEX was able to solve 1 432 instances (79.6 %) while our Lagrange implementation successfully solved 1 747 instances (97.1 %). For those instances successfully solved, minimum, median, and maximum runtimes are shown in Table 2 for each group of instances. For all of these groups, our approach outperforms CPLEX in terms of minimum, median, and maximum runtime, often by orders of magnitude.

Table 1: Overview of (alternative) parameters used for the generation of random instances.

General parameters
Alphabet size: {4, 20, 50}
Parameters controlling solution
Solution size: {2–10, 20–30}
Solution string lengths: {3–10, 20–30}
Parameters controlling instance strings
Number of strings: {10, 100}
String lengths: {50–100, 250–300}

6.2 Alice in Wonderland. We investigate to what extent MSC might be useful to recognize building blocks (such as words) of natural language texts. On English texts, the unit cost MSC problem will usually yield the alphabet as the optimal solution. Therefore, choosing a reasonable cost function is essential.

Here we report results on an instance derived from Alice in Wonderland by Lewis Carroll as follows. The text was obtained from `http://www.gutenberg.org/files/11/11.txt` and the header removed. Double dashes (`--`) and potential sentence separators (`?!;:`) were replaced by full stops, simple dashes and newlines by spaces. The text was split into sentences at the resulting full stops. All letters were converted into lower case. In principle, instances with alphabet size 26 can now be obtained by considering each sentence (without the full stop) as one string and removing the spaces between words in each sentence. The goal is to recover the word boundaries as the solution of the MSC problem.

However, to generate non-trivial but still solvable instances, some adjustments were necessary. Allowing words of size 1 or 2 leads to trivial solutions, so we prescribed a minimum word length $m \in \{3, 4\}$. We only kept sentences with at least 6 words with total length at least 50. We also ensured that each word that occurs at all occurs at least twice. We aimed for instances with $n \in \{50, 70, 80, 100, 150, 200\}$ sentences, and maximally many sentences. Due to the above restrictions, the desired values of n could not always be obtained exactly, so the next obtainable larger value was taken. Costs were computed for every occurring substring of length between m and 13 (shorter and longer strings were excluded by assigning infinite costs) as follows. We estimated Markovian text models of orders 0 (i.i.d model) and 1 from the instance by counting the frequency of single letters and 2-grams, respectively. The *p-value* of a string t with k observed occurrences is defined as the probability that t occurs at least k times in a set of the same size as the given one, chosen according to the random text model. The *conditional p-value* is the corresponding conditional probability, given that the string occurs at least once. The *score* of t is the natural logarithm of the conditional p-value. Intuitively, it measures the

Table 2: Performance comparison of CPLEX and the Lagrange-based optimization. For both, the minimal, median, and maximal runtime is reported along with the number of instances aborted due to memory (8gb) or time (1h) constraints (column "abrt.").

Solution size	Solution str. len.	String lengths	Runtime CPLEX [s] abrt. / min / median / max	Runtime Lagrange [s] abrt. / min / median / max
Instances with 10 strings				
Alphabet size 4				
2-10	20-30	250-300	0 / 26.7 / 368.0 / 1088.6	0 / **0.5** / **0.8** / **9.9**
2-10	20-30	50-100	0 / 0.7 / 1.9 / 8.8	0 / **0.01** / **0.04** / **0.6**
2-10	3-10	250-300	0 / 8.2 / 79.3 / 248.3	0 / **0.6** / **4.3** / **30.3**
2-10	3-10	50-100	0 / 0.5 / 1.5 / 6.2	0 / **0.01** / **0.1** / **0.5**
Alphabet size 20				
2-10	20-30	250-300	0 / 13.5 / 310.1 / 1289.6	0 / **0.7** / **0.8** / **19.5**
2-10	20-30	50-100	0 / 0.5 / 1.8 / 6.9	0 / **0.01** / **0.04** / **1.4**
2-10	3-10	250-300	0 / 9.8 / 74.8 / 183.5	0 / **0.7** / **0.9** / **5.6**
2-10	3-10	50-100	0 / 0.6 / 1.6 / 8.6	0 / **0.0** / **0.0** / **0.3**
Alphabet size 50				
2-10	20-30	250-300	0 / 13.5 / 347.5 / 1832.2	0 / **0.7** / **0.8** / **3.2**
2-10	20-30	50-100	0 / 0.9 / 1.9 / 12.3	0 / **0.01** / **0.04** / **0.4**
2-10	3-10	250-300	0 / 9.3 / 75.0 / 318.4	0 / **0.7** / **0.9** / **7.3**
2-10	3-10	50-100	0 / 0.6 / 1.8 / 8.4	0 / **0.01** / **0.03** / **0.3**
Instances with 100 strings				
Alphabet size 4				
20-30	20-30	250-300	50 / – / – / –	**49** / **169.4** / **169.4** / **169.4**
20-30	20-30	50-100	0 / 23.2 / 41.8 / 67.8	0 / **7.2** / **12.8** / **20.3**
20-30	3-10	250-300	**2** / 1265.7 / 1888.5 / 2919.5	4 / **145.6** / **277.4** / **1098.4**
20-30	3-10	50-100	0 / 18.9 / 30.6 / 55.9	0 / **6.9** / **13.0** / **25.7**
2-10	20-30	250-300	50 / – / – / –	**0** / **4.3** / **7.8** / **9.0**
2-10	20-30	50-100	0 / 57.9 / 217.6 / 531.7	0 / **0.1** / **0.3** / **4.2**
2-10	3-10	250-300	28 / 1307.2 / 2757.5 / 3574.6	**0** / **7.4** / **47.6** / **215.3**
2-10	3-10	50-100	0 / 27.6 / 55.2 / 187.8	0 / **0.2** / **2.1** / **11.6**
Alphabet size 20				
20-30	20-30	250-300	27 / 2062.2 / 2915.2 / 3530.9	**0** / **9.7** / **10.9** / **11.8**
20-30	20-30	50-100	0 / 12.0 / 16.5 / 24.2	0 / **0.5** / **0.6** / **0.7**
20-30	3-10	250-300	0 / 433.4 / 707.4 / 1170.3	0 / **10.1** / **14.6** / **77.2**
20-30	3-10	50-100	0 / 11.4 / 14.6 / 20.1	0 / **0.4** / **1.0** / **4.0**
2-10	20-30	250-300	50 / – / – / –	**0** / **4.0** / **8.0** / **9.2**
2-10	20-30	50-100	0 / 39.6 / 214.6 / 489.6	0 / **0.1** / **0.3** / **0.5**
2-10	3-10	250-300	20 / 1209.0 / 2363.4 / 3586.9	**0** / **6.9** / **10.1** / **65.2**
2-10	3-10	50-100	0 / 27.0 / 42.2 / 131.8	0 / **0.2** / **0.4** / **3.0**
Alphabet size 50				
20-30	20-30	250-300	50 / – / – / –	**0** / **10.0** / **11.1** / **11.8**
20-30	20-30	50-100	0 / 13.4 / 21.1 / 38.5	0 / **0.5** / **0.6** / **3.5**
20-30	3-10	250-300	8 / 835.0 / 1275.8 / 1877.4	**0** / **10.4** / **17.9** / **49.5**
20-30	3-10	50-100	0 / 15.9 / 26.1 / 36.9	0 / **0.5** / **1.0** / **3.2**
2-10	20-30	250-300	50 / – / – / –	**0** / **4.3** / **8.0** / **8.9**
2-10	20-30	50-100	0 / 70.2 / 252.0 / 685.9	0 / **0.2** / **0.3** / **0.5**
2-10	3-10	250-300	33 / 1469.2 / 3000.0 / 3608.4	**0** / **6.9** / **10.4** / **70.2**
2-10	3-10	50-100	0 / 32.9 / 84.3 / 275.4	0 / **0.2** / **0.4** / **0.6**

Table 3: Alice in Wonderland MSC results. Instance properties: m: minimum substring/interval length; n: number of strings in instance; size: total number of characters; substrings: number of substrings of length between m and 13. Solution properties: true cost: cost of true solution, using the known word boundaries; optimal: cost of optimal solution found by the ILP solver; prec.: precision of the optimal solution, i.e., percentage of substrings in the solution that are true words; recall: recall of the optimal solution, i.e., percentage of true words that are discovered by the optimal solution.

m	n	size	substrings	true cost > optimal	prec. [%]	recall [%]
3	50	4095	31608	11864 > 11433	87.9	86.4
3	70	5938	44659	23772 > 23206	88.2	87.0
3	80	6683	50083	26064 > 25314	87.6	86.2
3	101	8395	62649	31838 > 30935	86.8	85.7
3	150	12474	89171	70963 > 68838	86.2	84.4
3	200	16460	115100	121529 > 116399	86.3	83.3
3	620	51076	321621	626928 > 581239	86.3	67.8
4	51	3457	25714	14632 > 14536	93.1	93.1
4	71	4823	35355	25110 > 24910	95.9	95.5
4	81	5509	40291	33374 > 33139	95.9	95.4
4	100	6930	50487	44315 > 44044	95.1	94.7
4	150	10548	75419	86398 > 85611	94.0	93.3
4	200	14244	100051	126928 > 125261	94.3	93.3
4	363	26558	178397	301321 > 296109	91.2	90.0

observed exceptionality of t's frequency in comparison to a random text. We take the lower score among the two text models, i.i.d. and Markov order 1. We define the *cost* of t by $w(t) := \text{round}(M - score(t) + 1)$, where $M := \max_{\tau \in T(S)} score(\tau)$. Thus, all costs are positive integers.

Table 3 shows the instance properties and optimal solutions vs. the ground truth (true word boundaries). Note that the optimal solution has slightly lower cost than the true words. This indicates that the chosen cost function cannot model the word boundary problem perfectly, but comes close, as we see from the precision and recall values. For $m = 3$, most precision and recall percentages are well above 80%, the difficult $n = 620$ instance being a notable exception. For $m = 4$, all precision and recall percentages are above 90%. Differences mainly result from the optimal solution using concatenations of words instead of separate words, where this is possible. The optimal solution always uses slightly fewer substrings than there are true words.

Concerning runtimes, most instances were solved in a few seconds by both CPLEX and our Lagrangian approach, with the exception of $(m, n) = (4, 363)$, which took slightly over 2 minutes, and $(3, 620)$, which was solved in about 3 hours by CPLEX but did not finish within 24h using the Lagrangian relaxation approach. This might be due to the more sophisticated branching scheme implemented in CPLEX (see Discussion).

The generally short running times indicate that most of the instances are easy, especially for $m = 4$. Indeed, to avoid trivial solutions and provide a reasonable cost function, we had to give away many hints towards the solution (e.g., substring length constraints). However, we point out that these instances are the first weighted instances for the MSC problem inspired by a real-world application.

7 Discussion.

In the present work, we introduce the first practical algorithm to solve non-trivial instances of the Minimum String Cover problem. As we show, the separation problem for the exponentially sized ILP introduced by Hermelin et al. [4] is NP-hard. Therefore we introduce a novel, polynomially-sized flow-based formulation which is amenable to Lagrangian relaxation with respect to one class of linking constraints. This relaxation leads to a simple shortest path problem on a directed acyclic graph. Combining subgradient optimization and branch-and-bound search leads to a practical algorithm, an implementation of which is available as open source software.

CPLEX is a fast general purpose ILP solver. Thus, we use CPLEX to solve the new flow-based ILP and compare the runtimes to those of our approach. Table 2 shows that the Lagrangian approach indeed outperforms CPLEX by orders of magnitude on most instances.

So far, we know of no work that models a real-world problem usefully as a (weighted) MSC problem. Here

we took a first step by inferring word boundaries in sentences from "Alice in Wonderland". Clearly, designing a cost function that models the real-world problem remains a challenge; yet our approach allows to solve weighted instances of reasonable size efficiently. However, the relative importance of a state-of-the art implementation of the branch-and-bound scheme increases with the complexity of the instances, as indicated by the fact that CPLEX outperforms our implementation for the difficult instance $(3, 620)$, where a lot of branching is required. The strength of the Lagrangian approach lies in the efficient computation of strong bounds. We expect that it can greatly be improved on such difficult instances by tuning the branching behaviour.

We note that the non-existence of a polynomial-time algorithm for the LP relaxation given in [4] (Theorem 2.1) does not necessarily exclude practical useful approaches based on advanced techniques like delayed column generation. We plan to address this question in future research.

Furthermore, we are interested in variants of the Minimum String Cover problem. For instance, allowing a limited number of positions to remain uncovered might broaden the range of applications. Especially when dealing with noisy data, as ubiquitous in computational biology, this might be beneficial as it allows the algorithm to ignore parts of the input that cannot be explained by a string cover.

Acknowledgments. SR and CS are supported by the Collaborative Research Center (Sonderforschungsbereich, SFB) 876 "Providing Information by Resource-Constrained Data Analysis" within projects B1 and C4, respectively (`http://sfb876.tu-dortmund.de`).

We thank the reviewer of an earlier version of this paper for her/his constructive comments and suggestions.

References

[1] Dimitris Bertsimas, John N. Tsitsiklis, Dimitris Bertsimas, and John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, February 1997.

[2] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, Michael T. Hallett, and Harold T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11(1):49–57, 1995.

[3] M. Held and R.M. Karp. The traveling salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[4] Danny Hermelin, Dror Rawitz, Romeo Rizzi, and Stéphane Vialette. The minimum substring cover problem. *Information and Computation*, 206(11):1303–1312, November 2008.

[5] Georg L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley, Chichester, 1988.

[6] Jean Néraud. Elementariness of a finite set of words is co-NP-complete. *Theoretical Informatics and Applications*, 24:459–470, 1990.

[7] Alexander Schrijver. *Theory of linear and integer programming. repr. 94*. Wiley, Chichester, 1986.