Optimal Dictionary-Based Compression with Answer Set Programming: Encodings and Empirical Analysis

Mutsunori Banbara[1] , **Hideo Bannai**[2] , **Takashi Horiyama**[3] , **Dominik Köppl**[4] , **Takuya Mieno**[5] , **Hideotomo Nabeshima**[4]

[1]Nagoya University, Japan
[2]Institute of Science Tokyo, Japan
[3]Hokkaido University, Japan
[4]University of Yamanashi, Japan
[5]The University of Electro-Communications, Japan

banbara@nagoya-u.jp, hdbn.dsc@tmd.ac.jp,
horiyama@ist.hokudai.ac.jp,
{dkppl,nabesima}@yamanashi.ac.jp, tmieno@uec.ac.jp

Abstract

We develop an Answer Set Programming (ASP)-based approach for computing the smallest bidirectional macro schemes (BMSs), a fundamental NP-hard optimization problem in dictionary-based compression. Our approach relies on high-level ASP encodings and delegates both the grounding and solving tasks to an off-the-shelf ASP solver. The proposed encoding is compact and extensible, and leverages advanced ASP techniques to improve scalability, including ASP modulo acyclicity and refined declarative encodings of acyclicity constraints. We further show that our ASP encoding can be naturally extended to compute the smallest straight-line programs (SLPs), another important NP-hard measure of repetitiveness. Furthermore, we establish the competitiveness of our approach by empirically contrasting it with a more dedicated MaxSAT-based approach.

1 Introduction

Dictionary-based compression has received considerable renewed attention in recent years due to the prevalence of very large but *highly repetitive* data, such as versioned document collections or multiple genome sequences of the same species. In such settings, dictionary-based approaches are particularly effective, as they are able to capture long-range repetitions more naturally than compressors based on statistical modeling (Navarro 2021). Dictionary-based compression is a family of compressed representations essentially based on copy-and-paste mechanisms. It encompasses the outputs of many well-known compressors, including run-length compression, LZ77 (Lempel and Ziv 1976; Ziv and Lempel 1977), LZ78 (Ziv and Lempel 1978), grammar compression (e.g., RE-PAIR (Larsson and Moffat 1999), SEQUITUR (Nevill-Manning and Witten 1997), and GREEDY (Apostolico and Lonardi 2000)), as well as the run-length compressed Burrows–Wheeler transform (RLBWT) (Mäkinen and Navarro 2005).

Among various measures used to capture the repetitiveness of a string under dictionary-based compression, the size of a smallest *bidirectional macro scheme* (BMS) plays a central role. BMSs (Storer and Szymanski 1982) constitute the most general and powerful form of dictionary-based compression, and their minimum size can be viewed as a lower bound on the compressibility of a string. The theoretical study of such repetitiveness measures was initiated by Kempa and Prezza (Kempa et al. 2018), who uncovered various relations among them. Understanding the BMS measure is therefore crucial for analyzing the performance of compression algorithms and for developing more effective methods. However, computing the size of a smallest BMS is NP-hard (Storer 1977; Storer and Szymanski 1982), which makes this analysis challenging.

Although computing such repetitiveness measures for genome-scale data may still be challenging, the ability to compute them for moderately sized strings is already highly valuable for theoretical research. This usefulness is illustrated by two recent examples. Bannai et al. discovered a new lower bound on the sensitivity of the repetitiveness measure γ (Bannai et al. 2022), and Mieno et al. uncovered a connection between the RE-PAIR grammar and the smallest grammar for Fibonacci strings (Mieno, Inenaga, and Horiyama 2022). In both cases, key insights were first suggested by computational experiments that

evaluated the corresponding repetitiveness measures on various strings, and were later confirmed by rigorous theoretical proofs. In particular, the former result (Bannai et al. 2022) directly relied on a MaxSAT encoding of the underlying minimization problem.

In this paper, we present an alternative approach to computing the smallest bidirectional macro schemes (BMSs) based on Answer Set Programming (ASP). Our solver, *compass*, takes a string as input and translates it into a set of ASP facts, which are then combined with an ASP encoding of the BMS minimization problem. The resulting ASP program is subsequently solved using an off-the-shelf ASP solver, in our case *clingo* (Gebser et al. 2019).

ASP (Lifschitz 2019) is a declarative programming paradigm for knowledge representation and reasoning in artificial intelligence. ASP provides an expressive modeling language that is particularly well suited for representing combinatorial optimization problems. Due to significant advances in solver technology, ASP has been successfully applied to a wide range of domains, including product configuration, scheduling, planning, computational biology, and many others (Erdem, Gelfond, and Leone 2016). Despite these successes, the potential of ASP for modeling and solving optimization problems arising in dictionary-based compression has so far received little attention.

The main contributions and results of this paper are summarized as follows.

1. We present a basic ASP encoding for computing the smallest bidirectional macro schemes (BMSs). Our encoding is compact, human-readable, and highly extensible.
2. We extend the basic encoding to improve the scalability for large input strings. To this end, we leverage ASP modulo acyclicity (Bomanson et al. 2016) and an advanced ASP encoding of acyclicity constraints (Gebser, Janhunen, and Rintanen 2020).
3. We further show that our ASP encoding can be naturally extended to compute the smallest straight-line programs (SLPs; (Karpinski, Rytter, and Shinohara 1997)), another important NP-hard measure of repetitiveness.
4. Our extensive experiments on the Calgary Corpus (Bell, Witten, and Cleary 1989) show that our *compass* approach scales up to strings of length 7000 and outperforms a dedicated MaxSAT-based approach (Bannai et al. 2022).

This paper demonstrates that ASP provides a practical and scalable framework for dictionary-based compression. In particular, we show how recent advances in ASP technology enhance scalability for BMS minimization. To the best of our knowledge, this is the first systematic application of ASP to dictionary-based compression.

2 Background

2.1 Dictionary-based Compression

Let Σ be an *alphabet*, i.e., a set of characters. The elements of Σ^* are called *strings*. The length of a string T is denoted by $|T|$. $T[i]$ denotes the i-th character of the string T, and $T[i..j]$ denotes the substring of T from the i-th character to the j-th character. When $j = |T|$, $T[i..j]$ is a suffix starting at position i. We sometimes write $T[i..j+1) = T[i..j]$ to omit the last position. A *factorization* of a string T is defined as the partitioning of T into several substrings $F_1, \cdots, F_f$ such that $T = F_1 \cdots F_f$. Each substring F_x is called a *phrase*. The starting position and length of the phrase F_x are denoted by dst_x and λ_x, respectively: $F_x = T[\mathsf{dst}_x..\mathsf{dst}_x + \lambda_x)$.

From now on, the string T will be fixed as input. Let $\sigma := |\Sigma|$ denote the alphabet size and $n := |T|$ the length of $T[1..n] = T$. We use the string $T = \texttt{abaababaabaab}$ as a running example.

A **bidirectional macro scheme** (BMS) of size f representing T is a factorization $T = F_1, \ldots, F_f$, where each phrase F_x is either a single character (a *ground phrase*) or an encoded pair of integers (i, ℓ) indicating that it references the substring $T[i..i+\ell)$. A BMS is *valid*, if T can be reconstructed from the representation of such a factorization.

For example, a valid BMS `(6,6)ba(1,5)` representing $T = \texttt{abaababaabaab}$ is shown in Figure 1. This example is a smallest BMS of size 4 for T. Each phrase is displayed in rounded rectangles. The first phrase `(6,6)` references the substring $T[6..12) = \texttt{abaaba}$. The second `b` and third `a` are ground phrases. The last `(1,5)` references the substring $T[1..6) = \texttt{abaab}$.

More formally, BMS can be defined as follows. Let $R(i)$ denote the reference of each position i that is induced by the phrase references, i.e.,

$$R(i) = \begin{cases} \mathsf{src}_x + (i - \mathsf{dst}_x) & \text{if } F_x \text{ has reference } \mathsf{src}_x \\ \bot & \text{otherwise,} \end{cases} \tag{1}$$

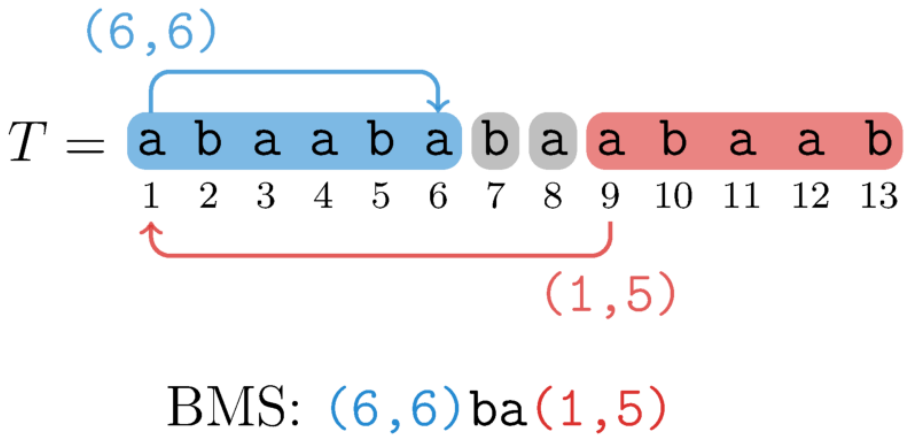


Figure 1: A smallest BMS for $T = \texttt{abaababaabaab}$.

where $\mathsf{dst}_x \leq i \leq \mathsf{dst}_x + \lambda_x - 1$. $R(i) = \bot$ means position i is a ground phrase. Also, consider a directed graph $G = (V, E)$ defined by $R(i)$, i.e., $V := [1..n]$ and $E := \{(i, R(i)) \mid i \in V, \text{where } R(i) \neq \bot\}$. Then, a valid BMS satisfies the following two conditions:

1. For each $x \in [1..f]$, F_x must have a reference $\mathsf{src}_x \in [1..n]$ if $|F_x| \geq 2$, such that $T[\mathsf{src}_x..\mathsf{src}_x + \lambda_x) = F_x$ and $\mathsf{src}_x \neq \mathsf{dst}_x$.
2. The graph $G = (V, E)$ must be acyclic. That is, G is a forest of rooted trees, where the roots are the positions of ground phrases since they have no reference.

A valid BMS `(6,6)ba(1,5)` is shown in Figure 1. This BMS is represented by a rooted forest $G = (V, E)$, where $V := [1..13]$, $E := \{(1,6),(2,7),(3,8),(4,9),(5,10),(6,11),(9,1),(10,2),(11,3),(12,4),(13,5)\}$, and the set of roots is $\{7, 8\}$. For example, position 10 references position 2, which in turn references position 7, a ground phrase `b`.

A **Straight-Line Program** (SLP) for T is a context-free grammar in Chomsky normal form. In an SLP, all production rules are of the form $X \to AB$ or $X \to c$, where X, A, B are non-terminal symbols and c is a terminal symbol, which derives only T. The size of an SLP is the number of production rules. It is known that an SLP is a form of BMS in that an SLP of size g induces a BMS of size $g - \sigma + 1$ (Rytter 2003). This can be seen by considering the *partial parse tree* of the grammar: from the complete parse tree of the grammar where internal nodes are non-terminal symbols and leaves are terminal symbols, prune the children (and all descendants) of each internal node $X \to AB$ except for exactly one occurrence of X in the parse tree. Then, the leaves of the partial parse tree can be seen as phrases of a BMS; any phrase whose descendants have been pruned can reference the single occurrence of the non-terminal whose descendants were not pruned. Otherwise, the phrase is a ground phrase. The number of internal nodes in the partial parse tree corresponds to the number of production rules of the form $X \to AB$, and is equivalent to the number of phrases minus 1. Since the grammar has σ more rules of the form $X \to c$, the claim follows.

On the other hand, it has been proven that a BMS with additional constraints in the referencing induces an SLP (Bannai et al. 2022, Lemma 2). The constraint required is that every reference must start at a phrase boundary and end at a phrase boundary, and that all references must either be nested or disjoint. An example for an SLP representing $T = \texttt{abaababaabaab}$ and its partial parse tree are shown in Figure 2. The production rules consist of $\{X_7 \to X_6X_5,\ X_6 \to X_5X_4,\ X_5 \to X_4X_3,\ X_4 \to X_3X_1,\ X_3 \to X_1X_2,\ X_2 \to \texttt{b},\ X_1 \to \texttt{a}\}$. This example is a smallest SLP of size 7 for T, which induces a BMS $\texttt{aba}(1,2)(1,3)(1,5)$ of size $6 = 7 - 2 + 1$.

2.2 Answer Set Programming

ASP programs are finite sets of *rules* of the form:

$$\texttt{a}_0 \texttt{ :- a}_1\texttt{,...,a}_m\texttt{, not a}_{m+1}\texttt{,...,not a}_n\texttt{.}$$

Each $\texttt{a}_i$ is an *atom* of the form $\texttt{p(t}_1\texttt{,...,t}_k\texttt{)}$, where `p` is a predicate symbol and all $\texttt{t}_i$ are terms. A *literal* is either an atom `a` or its default negation `not a`. The symbols '`:-`', '`,`', and '`not`' mean if,

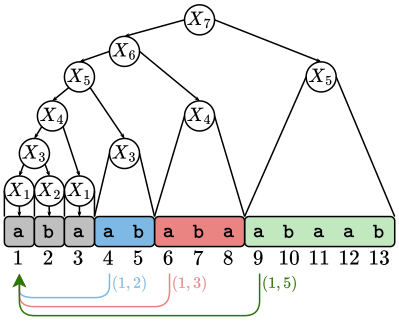


Figure 2: A smallest SLP for $T = \texttt{abaababaabaab}$: its partial parse tree and the induced BMS.

conjunction, and default negation, respectively. The left-hand side of '`:-`' is called the *head*, and the right-hand side is the *body*. Each rule is terminated by a period '`.`'. The intuitive meaning of a rule is that if $\texttt{a}_1$, $\ldots$, $\texttt{a}_m$ are true and $\texttt{a}_{m+1}, \ldots, \texttt{a}_n$ are false, then $\texttt{a}_0$ must be true.

ASP rules have two special cases. The first is a *fact* of the form $\texttt{a}_0$`.`, which must be unconditionally true and is used to represent problem input. The second is an *integrity constraint* of the form:

$$\texttt{:- a}_1\texttt{,...,a}_m\texttt{,not a}_{m+1}\texttt{,...,not a}_n\texttt{.}$$

Integrity constraints specify that the literals in the body must not be satisfied simultaneously and are used to filter solution candidates. Semantically, an ASP program induces a collection of *answer sets*, which are distinguished models under the stable model semantics (Gelfond and Lifschitz 1988). ASP rules with variables are understood as shorthand for the set of all their *ground* instances.

The ASP language provides several useful extensions for expressing combinatorial optimization problems. *Conditional literals* are expressions of the form $\ell_0\texttt{:}\ell_1\texttt{,}\ldots\texttt{,}\ell_m$, where each ℓ_i is a literal. *Cardinality constraints* are of the form $L_b\texttt{\{}c_1\texttt{;}\ldots\texttt{;}c_n\texttt{\}}U_b$, where each c_j is a conditional literal, and L_b and U_b specify lower and upper bounds on the number of satisfied literals. Finally, objective functions that minimize the sum of weights `w` associated with tuples `(w,t)` subject to conditions ℓ are expressed as `#minimize{w, t:`ℓ`}.`

Although we provide only a brief overview of the basic language constructs of ASP, we refer the reader to the literature (Gebser et al. 2012) for a comprehensive treatment.

3 The *compass* approach

Our *compass* approach relies on high-level ASP encodings and delegates both the grounding and solving tasks to a general-purpose ASP solver *clingo*. In this section, we present a collection of *compass* encodings not only for BMS minimization but also for SLP minimization.

Basic encoding for BMS minimization. ASP fact format of our running example $T = \texttt{abaababaabaab}$ is shown in Listing 1. The constant `textlength` indicates the length of a string T. The atom `t(`i`,`a`)` represents that the i-th character of T is a (i.e., $T[i] = a$).

Our basic ASP encoding for computing the smallest BMSs is shown in Listing 2. The rule in Line 2 guesses starting positions $\texttt{dst}_x$ of phrases, characterized by the predicate `p/1`. The atom `p(I)` represents that `I` is a starting position of a phrase. The rule in Line 5 guesses phrase references $R(i)$ by the predicate `r/2`. The atom `r(I,J)` represents that position `I` references position `J`, i.e., $R(\texttt{I}) = \texttt{J}$. This rule ensures that, for two distinct text positions `I` and `J` with the same symbol `C`, either position `I` can reference position `J`, or vice versa, but not both. The rule in Line 8 ensures that each text position has at most one reference. The rule in Line 11 encodes the set of ground phrase positions by enforcing that each text position `I` with $R(\texttt{I}) = \bot$ is the start of a phrase. Subsequently, we address the validity of the BMS we want to construct with the following two rules, each satisfying one of the two conditions for a valid BMS in the same order.

The rule in Line 14 represents Eq. (1), which then implies the first condition for a valid BMS. Given that position `I` references position `J`, but the adjacent previous position `I-1` does not reference position `J-1`, then position `I` must be the start of a phrase. Otherwise, we would obtain a contradiction to Eq. (1) since the offset of the reference at position `I` is different from that of position `I-1`; consequently both text positions must belong to different phrases.

The rule in Lines 17–19 ensures the second condition for a BMS to be valid, namely, to check whether the constructed reference forest is acyclic. We here represent this acyclicity as reachability constraints by shortcutting reference edges E directly to ground phrases. The rules in Lines 17–18 introduce the auxiliary

Listing 1: ASP fact format of $T =$ abaababaabaab

```
#const textlength=13.
t(1,"a").  t(2,"b").  t(3,"a").  t(4,"a").
t(5,"b").  t(6,"a").  t(7,"b").
t(8,"a").  t(9,"a").  t(10,"b").
t(11,"a"). t(12,"a"). t(13,"b").
```

Listing 2: Basic encoding (the *reachability* encoding)

```
1  % starting position of phrase
2  { p(1..textlength) }.
3
4  % phrase reference
5  { r(I,J) ; r(J,I) } 1 :- t(I,C), t(J,C), I < J, J <= textlength.
6
7  % reference uniqueness
8  :- not { r(I,_) } 1, I=1..textlength.
9
10 % ground phrase
11 :- not p(I), 0 { r(I,_) } 0,              I=1..textlength.
12
13 % phrase boundary
14 :- not p(I), r(I,J), not r(I-1,J-1).
15
16 % reachability constraints
17 reachable(I,I) :- 0 { r(I,_) } 0,         I=1..textlength.
18 reachable(X,R) :- r(X,Y), reachable(Y,R).
19 :- p(I), not 1 { reachable(I,_) } 1.
20
21 % objective function
22 #minimize { 1,I : p(I) }.
```

atom `reachable(X,R)`, which represents that position `X` can reach the ground phrase position `R`. The rule in Line 19 enforces that each starting position `I` can reach exactly one ground phrase. We note that it actually suffices to check the reachability of `p(I)` instead of `t(I,_)`. That is because each position can have at most one reference in Line 8.

Finally, the rule in Line 22 represents an objective function to minimize the number of text positions that start with a phrase, and thus, minimizing the number of BMS phrases. We refer to the encoding of Listing 2 as the *reachability* encoding.

Refined encodings. As can be seen above, ASP can elegantly represent the constraints of BMS. On the other hand, the acyclicity constraints (i.e., the second condition for valid BMSs) can be encoded in several ways. The simplest way is to use the transitive closure of R, shown in Listing 3. The atom `closure(X,Y)` represents that position `X` can reach position `Y` through one or more references, and the last integrity constraint ensures that there are no cycles.

Furthermore, we present two refinements of our basic encoding. One refinement is based on *ASP modulo acyclicity*, in our case, `#edge` directive (Bomanson et al. 2016). The acyclicity constraints represented by `#edge` are shown in Listing 4. The statement `#edge (I,J): r(I,J).` enforces that a directed graph $G = (V, E)$ defined by $R(i)$ is acyclic. The `#edge` directive is implemented using *clingo*'s theory propagator interface, which allows us to mitigate the grounding bottleneck and effectively propagate the acyclicity constraints.

The other refinement based on *the order encoding* of acyclicity constraints (Gebser, Janhunen, and Rintanen 2020) is shown in Listing 5. The order encoding ensures acyclicity by inductively deriving positions that do not belong to any cycle. The auxiliary atom `order(X,Y)` represents that the potential reference `r(X,Y)` is not in a cycle, either because the reference is absent or because the successor `Y` has already been derived as acyclic. The atom `order(X)` represents that position `X` is not part of any cycle. A position `X` is derived as `order(X)` if all outgoing edge candidates from `X` satisfy `order(X,Y)`. The integrity

Listing 3: The *closure* encoding

```
closure(X,Y) :- r(X,Y).
closure(X,Z) :- r(Y,Z), closure(X,Y).
:- closure(X,Y), closure(Y,X).
```

Listing 4: The *acyclicity* encoding

```
#edge (I,J): r(I,J).
```

constraint enforces that every node must be derived as acyclic.

We refer to the encodings obtained by replacing the reachability constraints of the *reachability* encoding (in Lines 17–19 in Listing 2) with Listing 3, Listing 4, and Listing 5 as the *closure* encoding, the *acyclicity* encoding, and the *order* encoding, respectively.

Extension to SLP minimization. Finally, we extend the basic encoding in Listing 2 for computing the smallest SLPs. The extended encoding, called *bmsslp*, is shown in Listing 6. We introduce two auxiliary atoms `l(I,L)` and `q(J,L)`. The `l(I,L)` atom represents that $T[\mathtt{I..I+L})$ is a phrase of the grammar parsing. The `q(J,L)` atom represents that $T[\mathtt{J..J+L})$ corresponds to an internal node of the partial parse tree that is referenced by at least one phrase in the grammar.

The main difference from the basic encoding is that the extended encoding has no reachability and acyclicity constraints, and enforces additional constraints in Lines 27–29 and 32. The rules in Lines 27–29 ensure that, if $T[\mathtt{J..J+L})$ is an internal node, $T[\mathtt{J..J+L})$ cannot be a phrase (Line 27), $T[\mathtt{J}]$ and $T[\mathtt{J+L}]$ must be starting positions of phrases (Lines 28 and 29, respectively). The rule in Line 32 represents the *interval* constraints of SLP, and ensures that, for any two such implied internal nodes, they must either be disjoint or nested. We note that the `r(J,I)` atom in Line 5 is restricted so that position `J` can only reference previous position `I`, but not vice versa. This is because, in SLP, only the leftmost occurrences of the non-terminals are internal nodes.

This extension illustrates the elaboration tolerance of ASP. That is, complex additional constraints can be incorporated with minimal changes to the basic encoding.

4 Evaluation

We conducted experiments to evaluate the effectiveness of our *compass* approach for computing the smallest BMSs. Our empirical analysis considers all instances of the Calgary Corpus (Bell, Witten, and Cleary 1989), a widely adopted benchmark set in compression research. This benchmark set consists of plain text, source code, manuals, articles, and some binary files. To assess scalability, we extracted prefixes of 100, 300, 500, and 1000 characters from each file to construct benchmark sets of varying input lengths, resulting in 46 instances for each length (184 instances in total).

We compared our *compass* encodings based on ASP (*closure*, *reachability*, *acyclicity*, and *order*) with the existing MaxSAT-based approach. We used the MaxSAT encoding from (Bannai et al. 2022) with two top solvers from the MaxSAT Evaluation 2024, WMaxCDCL-openwbo (Li et al. 2024b) and MaxCDCL-openwbo (Li et al. 2024a), as well as PySAT (Ignatiev, Morgado, and Marques-Silva 2018), which was also used in (Bannai et al. 2022).

We used *clingo* (Gebser et al. 2019) version 5.7.1 as the ASP solver. In our preliminary experiments, we compared eight built-in solver configurations (including the default `auto`) and two optimization strategies: branch-and-bound (`bb`) and unsatisfiable core–guided optimization (`usc`). We used the built-in configuration `trendy` together with `usc`, since this combination consistently showed good performance across our benchmarks.

For PySAT, we selected Glucose 4.2.1 (Audemard and Simon 2009) and MiniSAT (GitHub version) (Eén and Sörensson 2004) as the SAT solvers, and RC2 (Ignatiev, Morgado, and Marques-Silva 2019) as the MaxSAT algorithm, which is also based on unsatisfiable core–guided optimization. This selection was based on our preliminary testing in which we evaluated all combinations of 17 SAT solvers and 5 MaxSAT algorithms supported by PySAT. Hereafter, we refer to Glucose 4.2.1 as `g42` and MiniSAT (GitHub) as `mgh`, respectively.

We ran experiments on a machine with an Intel Xeon Platinum 8480+ CPU and 512 GiB RAM, using a 3600-second timeout per instance. The timeout includes grounding in the ASP setting but excludes

Listing 5: The *order* encoding

```
node(X) :- X=1..textlength.
pair(I,J) :-
    t(I,C), t(J,C),
    I != J, node(I), node(J).
order(X,Y) :- pair(X,Y), not r(X,Y).
order(X,Y) :- pair(X,Y), order(Y).
order(X) :- node(X), order(X,Y) : pair(X,Y).
:- node(X), not order(X).
```

Table 1: The number of solved instances and the PAR-2 scores for ASP and MaxSAT encodings across different input lengths.

		len100		len300		len500		len1000		all	
Type	*Encoding* / Solver	#sol	PAR-2	#sol	PAR-2	#sol	PAR-2	#sol	PAR-2	#sol	PAR-2
ASP	*closure*	43	551	24	3590	13	5386	1	7044	81	4143
	reachability	45	275	28	3089	6	6483	2	6964	81	4203
	acyclicity	43	502	30	2549	**18**	**4415**	10	5657	101	3281
	order	44	398	**31**	**2399**	17	4545	**11**	**5494**	**103**	**3209**
MaxSAT	MaxCDCL-openwbo	45	268	22	3986	2	6887	2	6893	71	4509
	WMaxCDCL-openwbo	45	245	23	3755	2	6887	2	6892	72	4445
	PySAT-g42	45	179	23	3829	2	6887	2	6889	72	4446
	PySAT-mgh	**46**	**84**	29	3068	3	6761	2	6911	80	4206
VBS		46	25	39	1224	24	3612	13	5191	122	2513

encoding time for MaxSAT. In the MaxSAT setting, instances for which the encoding process exceeded the 3600-second limit, typically those resulting in more than 10^8 variables, were excluded from evaluation. The number of successfully encoded MaxSAT instances was 46, 42, 33, and 26 for input lengths 100, 300, 500, and 1000, respectively.

Table 1 reports the number of instances for which optimality was proven within the timeout for each encoding. The #sol columns indicate the number of solved instances. The PAR-2 columns indicate the penalized average runtime, where unsolved instances are counted as twice the timeout. Figure 3 shows cactus plots of the number of constraints and variables for each encoding. The constraint and variable counts in Figure 3 differ in meaning between ASP and MaxSAT. For ASP encodings, they indicate the number of constraints and variables after grounding, whereas for MaxSAT they reflect the size of the encoded MaxSAT instances.

Overall, Table 1 shows that ASP-*order* solved the largest number of instances (103), followed by ASP-*acyclicity* (101). In particular, these two encodings demonstrate better scalability. More precisely, ASP-*acyclicity* and ASP-*order* solved 10 and 11 instances of length 1000, respectively, whereas the other methods solved only one or two.

MaxSAT-PySAT-mgh solved the largest number of instances for short inputs of length 100, but its performance declined as the input length increased. Indeed, for input lengths of 500 and 1000, the MaxSAT-based approach failed to solve most instances. ASP-*closure* and ASP-*reachability* solved some instances up to length 500, but showed a similar decline at length 1000. In both ASP and MaxSAT approaches, instances with over 100 million constraints or variables proved to be intractable.

In contrast, ASP-*acyclicity* uses the built-in `#edge` directive to enforce acyclicity, rather than explicitly encoding the acyclicity constraints. As shown in Figure 3, the number of constraints and variables is approximately two orders of magnitude smaller than those of ASP-*closure* and ASP-*reachability*. Consequently, ASP-*acyclicity* scaled well to longer inputs. Interestingly, ASP-*order* based on an inductive vertex-ordering formulation also achieved comparable performance in the BMS minimization, despite not relying on solver-specific features such as the `#edge` directive.

Figure 4 shows cactus plots of the solving times for instances grouped by input length. At length 100, the solvers performed almost identically; however, MaxSAT-based solvers showed slightly better performance than ASP-based ones. Among them, MaxSAT-PySAT-mgh solved the most instances, namely 46. At

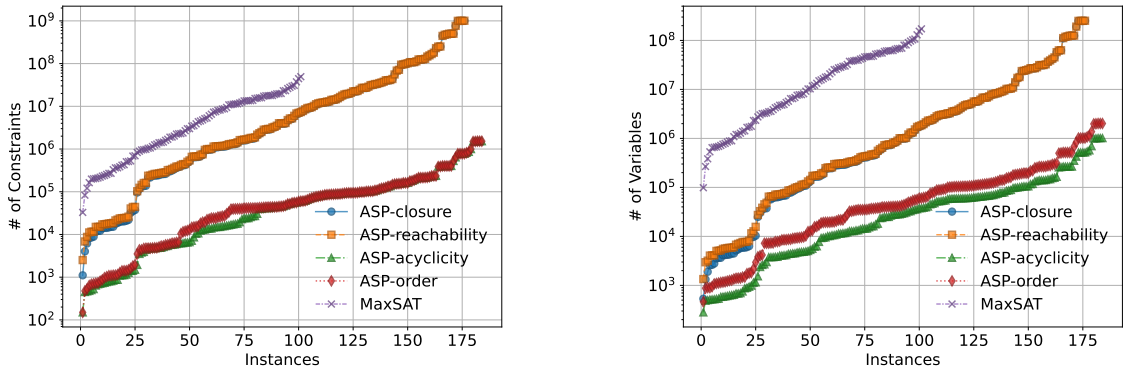
Listing 6: Extension to SLP (the *bmsslp* encoding)

```
% starting position of phrase
{ p(1..textlength) }.

% phrase reference (restricted)
{ r(J,I) } 1 :- t(I,C), t(J,C), I < J,    J <= textlength.

% reference uniqueness (BMS2)
:- not { r(I,_) } 1, I=1..textlength.

% ground phrase (BMS3)
:- not p(I), 0 { r(I,_) } 0,              I=1..textlength.

% phrase boundary (BMS4)
:- not p(I), r(I,J), not r(I-1,J-1).

% objective function (MINP)
#minimize { 1,I : p(I),I <= textlength }.

%%% Extension to SLP
p(textlength+1).

% phrase length
l(I,J-I) :- p(I), p(J), I<J, J <= textlength+1, not p(Z) : Z=I+1..J-1.
q(J,L) :- r(I,J), l(I,L), L > 1.

% start and end at phrase boundaries
:- l(J,L), q(J,L).
:- not p(J), q(J,L).
:- not p(J+L), q(J,L).

% interval constraints
:- q(I,L), q(J,M), I < J, J < I+L, I+L < J+M.
```

length 300, MaxSAT-PySAT-mgh remained the best among MaxSAT solvers, while ASP-*order* solved the most instances overall, namely 31, indicating the emerging advantage of this encoding. At lengths 500 and 1000, MaxSAT solvers were able to solve only a few instances, whereas ASP-*acyclicity* and ASP-*order* maintained high performance, demonstrating their scalability to larger inputs. ASP-*closure* and ASP-*reachability* solved a moderate number of instances at length 500, but their performance dropped significantly at length 1000. The virtual best solver (VBS) solved the highest number of instances across all input lengths, suggesting that the encodings have complementary strengths.

Figure 5 shows the distribution of instances uniquely solved by different solver groups. For example, Group A indicates that 59 instances were solved by all solvers. Some groups, such as Groups B, C, and D, include instances solved only by ASP-based solvers, whereas others, such as Group E, include instances solved only by MaxSAT-based solvers. Among the instances solved by only a subset of solvers, the largest group (14 instances) was solved exclusively by ASP-*order* and ASP-*acyclicity*. Table 2 lists a subset of these uniquely solved instances, further demonstrating that ASP-*order* and ASP-*acyclicity* are particularly effective for large-scale instances.

Table 3 shows the solving time and the best lower bound found for each instance of length 1000. All solvers employed unsatisfiable core-guided optimization, in which the lower bound is progressively refined toward an optimal value. The table reports the best lower bound obtained within the time-limit. The symbol "–" indicates the MaxSAT encoding timed out (time columns) or that no lower bound was found (LB columns). Both ASP-*acyclicity* and ASP-*order* were able to produce the same best lower bounds for most instances. Furthermore, ASP-*acyclicity* and ASP-*order* succeeded in finding optimal solutions for 10 and 11 instances, respectively. Even though MaxSAT solved few instances overall, the lower bound was updated in many cases. However, for most instances, the lower bounds found by MaxSAT were weaker than those obtained with the ASP encodings. These results indicate that ASP-based modeling provides a practical and scalable alternative to dedicated MaxSAT encodings for complex compression measures.



(a) Constraints

(b) Variables

Figure 3: Cactus plots showing the number of constraints and variables in ascending order of instance size, aggregated over all benchmark instances (lengths 100, 300, 500, and 1000), with a logarithmic scale on the vertical axis.

Table 2: Instances uniquely solved by each group with size at least two.

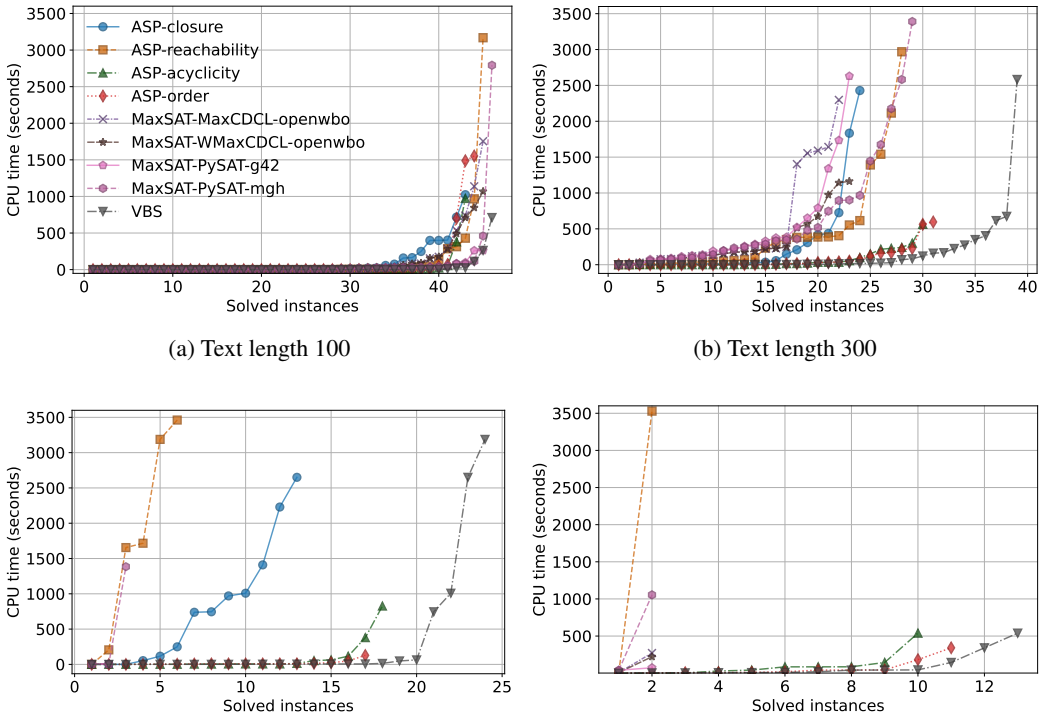
Label	Length	File Name	
B	300	sum	
	500	aaa.txt cp.html ptt5	book2 pic sum
	1000	aaa.txt book1 paper4 ptt5	asyoulik.txt paper3 pic
C	500	fields.c paper2 plrabn12.txt world192.txt	paper1 paper3 trans
D	300	aaa.txt ptt5	pic
	500	obj1	pi.txt
	1000	pi.txt	
E	100	alphabet.txt	
	300	alphabet.txt	paper5
	500	alphabet.txt	
	1000	alphabet.txt	
F	300	binary-de-brujin.11	kennedy.xls
	500	geo	
G	300	bible.txt	
	500	book1	paper4
H	300	news	xargs.1
I	500	asyoulik.txt	bib
J	1000	paper2	world192.txt
K	500	lcet10.txt	
	1000	sum	

Finally, we discuss the scalability limits of our ASP-based approach. Further experimental results show that our ASP-*acyclicity* encoding can optimally compress strings of length up to 7,000. However, no encoding was able to find optimal BMSs for strings of length 8,000 within the time limit.

Table 4 shows the maximum lengths for which optimal BMSs can be obtained using our ASP-*acyclicity*

Table 3: Lower bounds (LB) and CPU times for each solver on benchmark instances of length 1000. *Instance* denotes the file name; "#char types" counts distinct characters in the first 1000 characters (entire instance in parentheses). An asterisk (*) indicates that the solution was proven optimal; bold highlights the best LB. "t.o." indicates timeout; "m.o." memory overflow.

		ASP								MaxSAT	
	#char	closure		reachability		acyclicity		order		PySAT-mgh	
Instance	types	LB	Time	LB	Time	LB	Time	LB	Time	LB	Time
E.coli	4 (4)	202	t.o.	202	t.o.	**203**	t.o.	**203**	t.o.	–	–
aaa.txt	1 (1)	–	t.o.	–	t.o.	**2***	85	**2***	42	–	–
alice29.txt	56 (74)	383	t.o.	382	t.o.	**391**	t.o.	**391**	t.o.	369	t.o.
alphabet.txt	26 (26)	2	t.o.	6	t.o.	3	t.o.	3	t.o.	**27***	39
asyoulik.txt	60 (68)	404	t.o.	383	t.o.	**414***	144	**414***	182	371	t.o.
bib	68 (81)	454	t.o.	434	t.o.	**458**	t.o.	**458**	t.o.	420	t.o.
bible.txt	37 (63)	220	t.o.	185	t.o.	234	t.o.	**235**	t.o.	183	t.o.
binary-de-brujin.11	2 (2)	**104**	t.o.	**104**	t.o.	**104**	t.o.	**104**	t.o.	–	–
book1	60 (82)	468	t.o.	462	t.o.	**471***	3	**471***	7	463	t.o.
book2	53 (96)	405	t.o.	383	t.o.	**416**	t.o.	**416**	t.o.	377	t.o.
cp.html	72 (86)	372	t.o.	337	t.o.	**379**	t.o.	**379**	t.o.	350	t.o.
fibonacci.16	2 (2)	–	t.o.	–	t.o.	**4**	t.o.	**4**	t.o.	–	–
fields.c	72 (90)	378	t.o.	357	t.o.	**386**	t.o.	**386**	t.o.	372	t.o.
geo	165 (256)	522	t.o.	515	t.o.	**527**	t.o.	**527**	t.o.	–	–
grammar.lsp	49 (76)	203	t.o.	173	t.o.	**208**	t.o.	205	t.o.	179	t.o.
kennedy.xls	122 (256)	385	t.o.	**389**	t.o.	387	t.o.	387	t.o.	–	–
kolakoski.17	2 (2)	29	t.o.	29	t.o.	**30**	t.o.	**30**	t.o.	–	–
lcet10.txt	63 (84)	333	t.o.	323	t.o.	342	t.o.	**343**	t.o.	–	–
news	69 (98)	455	t.o.	429	t.o.	**466**	t.o.	**466**	t.o.	421	t.o.
obj1	7 (256)	–	t.o.	–	t.o.	**11**	t.o.	**11**	t.o.	–	–
obj2	101 (256)	429	t.o.	387	t.o.	**436**	t.o.	**436**	t.o.	414	t.o.
paper-folding.10	2 (2)	8	t.o.	7	t.o.	**11**	t.o.	**11**	t.o.	–	–
paper1	66 (95)	437	t.o.	417	t.o.	**440**	t.o.	**440**	t.o.	406	t.o.
paper2	66 (91)	446	t.o.	436	t.o.	436	t.o.	**451***	8	429	t.o.
paper3	59 (84)	430	t.o.	415	t.o.	**431***	42	**431***	19	394	t.o.
paper4	58 (80)	426	t.o.	420	t.o.	**436***	0	**436***	4	401	t.o.
paper5	74 (91)	395	t.o.	385	t.o.	406	t.o.	**412**	t.o.	369	t.o.
paper6	67 (93)	349	t.o.	352	t.o.	362	t.o.	**363**	t.o.	341	m.o.
period-doubling.11	2 (2)	–	t.o.	–	t.o.	**5**	t.o.	**5**	t.o.	–	–
pi.txt	10 (10)	372	t.o.	**374***	3530	**374***	29	**374***	11	**374**	t.o.
pic	1 (159)	–	t.o.	–	t.o.	**2***	85	**2***	44	–	–
plrabn12.txt	61 (81)	390	t.o.	377	t.o.	403	t.o.	**404**	t.o.	366	t.o.
power2.11	2 (3)	–	t.o.	–	t.o.	**11**	t.o.	**11**	t.o.	–	–
progc	68 (92)	353	t.o.	337	t.o.	**369**	t.o.	**369**	t.o.	312	t.o.
progl	38 (87)	225	t.o.	223	t.o.	**261**	t.o.	**261**	t.o.	–	–
progp	58 (89)	411	t.o.	393	t.o.	**415**	t.o.	**415**	t.o.	397	t.o.
ptt5	1 (159)	–	t.o.	–	t.o.	**2***	86	**2***	42	–	–
quaternary-paper-folding.11	4 (4)	9	t.o.	9	t.o.	**11**	t.o.	**11**	t.o.	–	–
random.txt	64 (64)	**892***	2	**892***	2	**892***	0	**892***	0	**892***	1063
sum	89 (255)	–	t.o.	–	t.o.	**225***	539	**225**	t.o.	–	–
thue-morse.11	2 (2)	5	t.o.	5	t.o.	**7**	t.o.	**7**	t.o.	–	–
trans	69 (99)	**468**	t.o.	448	t.o.	**468**	t.o.	**468**	t.o.	441	t.o.
tribonacci.13	3 (3)	4	t.o.	3	t.o.	**5**	t.o.	**5**	t.o.	–	–
vtm.11	3 (3)	7	t.o.	7	t.o.	9	t.o.	**10**	t.o.	–	–
world192.txt	64 (94)	393	t.o.	375	t.o.	401	t.o.	**402***	342	371	t.o.
xargs.1	60 (74)	374	t.o.	347	t.o.	**382**	t.o.	381	t.o.	345	t.o.



(a) Text length 100

(b) Text length 300

(c) Text length 500

(d) Text length 1000

Figure 4: Cactus plots showing the CPU time required to solve instances, sorted by CPU time, for each benchmark set. The legend shown in (a) applies to all four plots.

encoding. The last two instances, marked with $*$, are included for reference purposes. The third column (#char) indicates the number of distinct characters in the prefix of each length shown. Scalability correlates broadly with the repetitiveness of the input. Datasets with few character types (aaa.txt, pic, ptt5) and uniformly random strings (random.txt) perform well, as the lack of repetitive patterns limits reference candidates per position. Among English texts, book1 is notably easy due to its uniform prose style, whereas paper2, paper4, and asyoulik.txt are harder due to mixed styles and symbols. Despite having only 10 character types, pi.txt is limited to length 2,000, since its pseudo-random digit sequence yields few repetitions and thus a large number of BMS phrases. Notably, alice29.txt and plrabn12.txt, although they are English novels similar to book1, are only solvable up to lengths 300 and 500, respectively; the reason for this discrepancy remains unclear.

In contrast, our ASP-*order* encoding introduces more variables than the ASP-*acyclicity* encoding, as illustrated in Figure 3 (b). For example, for aaa.txt of length 5,000, ASP-*acyclicity* has #variables = 25,014,993 and #constraints = 62,532,480, while ASP-*order* has #variables = 50,014,993 and #constraints = 62,532,480. Thus, the number of variables is approximately doubled. While both encodings exhibit comparable solving performance up to length 5,000, the ASP-*order* encoding encounters difficulties in grounding instances of length 6,000 and above. As a result, the ASP-*acyclicity* encoding becomes clearly advantageous for larger instances.

Related Work

The use of MaxSAT for BMS minimization was first studied in (Bannai et al. 2022). This MaxSAT encoding adds the depth of the reference forest to enforce acyclicity constraints. Very recently, an improved MaxSAT encoding using the transitive closure of phrase references was proposed in (Bannai et al. 2025). The underlying idea of the improved MaxSAT encoding (Bannai et al. 2025) is close to our *closure* encoding. Besides BMS and SLP, other important families of dictionary-based compression include run-length SLP (Kawamoto et al. 2024), LZ-End (Kawamoto et al. 2024; Kreft and Navarro 2010; Kreft and Navarro 2013), common string partition (Goldstein, Kolman, and Zheng 2005; Köppl 2023) and collage systems (Kida et al. 2003). By our results, it would be interesting to explore whether ASP can be effective for these measures.

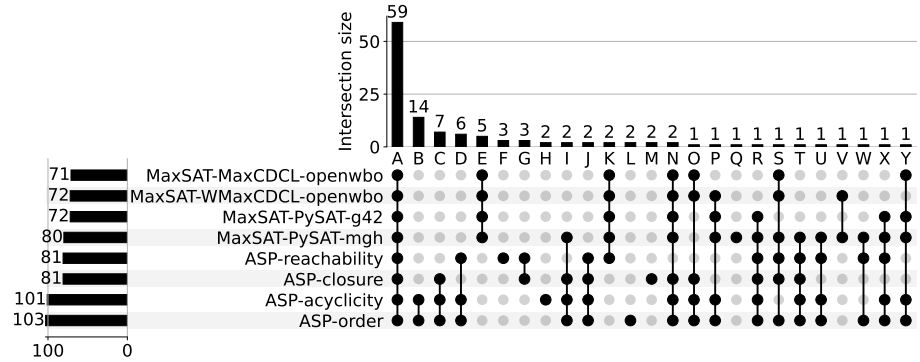


Figure 5: UpSet plot showing the intersections of instances solved by different solver groups (labels A–Y), aggregated over all benchmark instances (lengths 100, 300, 500, and 1000). In each row, the connected dots indicate a solver group (i.e., a specific combination of solvers), and the vertical bar above shows how many instances were solved only by that group. The horizontal bars on the left indicate the total number of instances solved by each individual solver across all groups.

Table 4: The maximum lengths for which optimal BMSs can be obtained by our ASP-*acyclicity* encoding

Length	Instance	#char	Optimal BMS size
7,000	aaa.txt	1	2
7,000	pic	2	4
7,000	ptt5	2	4
6,000	random.txt	64	3,550
4,000	book1	67	1,215
3,000	paper2	71	966
2,000	asyoulik.txt	63	716
2,000	paper4	70	698
2,000	pi.txt	10	658
300∗	alice29.txt	44	145
500∗	plrabn12.txt	53	238

5 Conclusion

We developed an ASP-based approach to dictionary-based compression, focusing on computing the smallest BMSs and the smallest SLPs. The lessons learned from our *compass* approach are as follows. First, the ASP language is well suited to modeling optimization problems in dictionary-based compression, as illustrated by our basic encoding (*reachability*). Second, the elaboration tolerance of ASP facilitates the extension of encodings, as demonstrated by our extended encoding (*bmsslp*). Finally, ASP makes it easy to experiment with advanced solving techniques, as evidenced by our experimental results. In particular, our refined encodings (*acyclicity* and *order*), based on ASP modulo acyclicity and the order encoding of acyclicity constraints, respectively, significantly improve the scalability of BMS solving. All source code, including the *compass* solver, the ASP encodings, and the benchmark instances, is publicly available from the web.[1]

We envision two main use cases for our *compass* tool. First, our tool can serve as an offline *gold standard* generator to evaluate the approximation ratios of linear-time heuristics. This is because the size of the smallest BMS can be viewed as a lower bound on the compressibility of the string in terms of dictionary compression. Second, our tool can support theoretical investigations in the stringology/combinatorics on words community. There is currently strong interest in the study of structural properties of repetitiveness measures such as BMSs and SLPs. In this setting, exact computation on carefully chosen example strings is often valuable for forming, testing, and refining conjectures before proving results in full generality.

Further refinement of our ASP-based approach is left for future work. In particular, the millions of constraints required for the largest solvable instances, even with our compact ASP encodings, motivate the use of specialized propagators for stronger constraints. In our preliminary experiments, by implementing a dedicated propagator for interval constraints in SLPs, we were able to significantly improve the perfor-

[1] https://github.com/nabesima/kr2026

mance of computing the smallest SLPs. We will investigate the potential for performance improvements using propagators in future work.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed and valuable feedback. This work was partially supported by JSPS KAKENHI Grant Numbers JP25K03097, JP25H01114, JP24K02899, JP25K21150, JP24K20734, JP26K02983, JP20H05964, and JP23K24806, and by RIMS, Kyoto University. In this work, we used the supercomputer of ACCMS, Kyoto University.

AI Declaration

A large language model (LLM) was used only for English language proofreading. All content has been reviewed by the authors, who take full responsibility for its accuracy and originality.

References

Apostolico, A., and Lonardi, S. 2000. Off-line compression by greedy textual substitution. *Proc. IEEE* 88(11):1733–1744.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI*, 399–404.

Bannai, H.; Goto, K.; Ishihata, M.; Kanda, S.; Köppl, D.; and Nishimoto, T. 2022. Computing NP-hard repetitiveness measures via MAX-SAT. In *Proc. ESA*, volume 244 of *LIPIcs*, 12:1–12:16.

Bannai, H.; Goto, K.; Ishihata, M.; Kanda, S.; Köppl, D.; Nishimoto, T.; and Subercaseaux, B. 2025. Computing NP-hard repetitiveness measures via MAX-SAT. *ACM Transactions on Algorithms*. Accepted on 12 November 2025.

Bell, T. C.; Witten, I. H.; and Cleary, J. G. 1989. Modeling for text compression. *ACM Comput. Surv.* 21(4):557–591.

Bomanson, J.; Gebser, M.; Janhunen, T.; Kaufmann, B.; and Schaub, T. 2016. Answer set programming modulo acyclicity. *Fundamenta Informaticae* 147(1):63–91.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *Proc. SAT*, volume 2919 of *LNCS*, 502–518. Springer.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine* 37(3):53–68.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19(1):27–82.

Gebser, M.; Janhunen, T.; and Rintanen, J. 2020. Declarative encodings of acyclicity properties. *Journal of Logic and Computation* 30(4):923–952.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Proc. ICLP*, 1070–1080. MIT Press.

Goldstein, A.; Kolman, P.; and Zheng, J. 2005. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.* 12.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A python toolkit for prototyping with SAT oracles. In *Proc. SAT*, volume 10929 of *LNCS*, 428–437.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2019. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.* 11(1):53–64.

Karpinski, M.; Rytter, W.; and Shinohara, A. 1997. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing* 4(2):172–186.

Kawamoto, A.; I, T.; Köppl, D.; and Bannai, H. 2024. On the hardness of smallest RLSLPs and collage systems. In *Proc. DCC*, accepted.

Kempa, D.; Policriti, A.; Prezza, N.; and Rotenberg, E. 2018. String attractors: Verification and optimization. In *Proc. ESA*, volume 112 of *LIPIcs*, 52:1–52:13.

Kida, T.; Matsumoto, T.; Shibata, Y.; Takeda, M.; Shinohara, A.; and Arikawa, S. 2003. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.* 298(1):253–272.

Köppl, D. 2023. Encoding hard string problems with answer set programming. In *Proc. CPM*, volume 259 of *LIPIcs*, 17:1–17:21.

Kreft, S., and Navarro, G. 2010. LZ77-like compression with fast random access. In *Proc. DCC*, 239–248.

Kreft, S., and Navarro, G. 2013. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.* 483:115–133.

Larsson, N. J., and Moffat, A. 1999. Offline dictionary-based compression. In *Proc. DCC*, 296–305.

Lempel, A., and Ziv, J. 1976. On the complexity of finite sequences. *IEEE Trans. Information Theory* 22(1):75–81.

Li, C.-M.; Li, S.; Coll, J.; Habet, D.; Manyà, F.; and He, K. 2024a. MaxCDCL in MaxSAT evaluation 2024. In *Solver and Benchmark Descriptions*, volume B-2024-2 of *Computer Science Report Series B*, 15–16. University of Helsinki.

Li, S.; Li, C.-M.; Coll, J.; Habet, D.; Manyà, F.; and He, K. 2024b. WMaxCDCL in MaxSAT evaluation 2024. In *Solver and Benchmark Descriptions*, volume B-2024-2 of *Department of Computer Science Report Series B*, 17–18. University of Helsinki.

Lifschitz, V. 2019. *Answer Set Programming*. Springer-Verlag.

Mäkinen, V., and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.* 12(1):40–66.

Mieno, T.; Inenaga, S.; and Horiyama, T. 2022. RePair grammars are the smallest grammars for Fibonacci words. In *Proc. CPM*, volume 223 of *LIPIcs*, 26:1–26:17.

Navarro, G. 2021. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.* 54(2):29:1–29:31.

Nevill-Manning, C. G., and Witten, I. H. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.* 7:67–82.

Rytter, W. 2003. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.* 302(1-3):211–222.

Storer, J. A., and Szymanski, T. G. 1982. Data compression via textual substitution. *J. ACM* 29(4):928–951.

Storer, J. A. 1977. NP-completeness results concerning data compression. *Technical Report 234*.

Ziv, J., and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory* 23(3):337–343.

Ziv, J., and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory* 24(5):530–536.