Recompressing compressed data

Dominik Köppl[1], Francesco Pio Marino[2,3]

[1]University of Yamanashi, Japan
[2]Università di Catania, Italy
[3]Université de Rouen Normandie, France

Abstract

We study the problem of *recompression*, namely the application of additional lossless compression stages to data that has already been compressed. Focusing on binary outputs produced by state-of-the-art compressors such as `7zip`, we introduce recompression algorithms that operate directly on the compressed bitstream, without requiring decompression or format-specific parsing. Our approach is based on fixed-length window substitutions that detect and replace repeated segments in the compressed stream with positional references, in the spirit of Lempel–Ziv schemes. We provide theoretical guarantees showing that, for sufficiently large window sizes, the proposed method is *non-expansive*, i.e., it never increases the size of the input. We further extend this strategy to a recursive, multi-scale recompression framework and discuss the algorithmic implications of optimal window selection.

Keywords

recompression, data compression, compressed data, sliding window algorithms

1. Introduction

Data compression has long been a cornerstone of information processing and storage efficiency [1]. Once a dataset has been compressed, it is generally assumed that little to no additional redundancy remains to be exploited. However, the rapid growth of digital data and the widespread use of multi-stage storage pipelines raise a natural question: can already compressed data be further compressed without decompressing it first? This question lies at the core of the emerging field of *recompression* [2, 3, 4]. Unlike traditional compression, recompression operates on data that has already undergone one or more compression passes. The goal is not to replace existing compressors, but to apply an additional lightweight transformation that identifies residual or structural redundancies within compressed streams.

Recompression represents a non-trivial challenge: standard compressors produce outputs designed to appear statistically random, leaving little obvious structure to exploit. Nevertheless, practical observations suggest that compressed data often contains patterns arising from header structures, metadata, padding, container formats, and local statistical dependencies between compressed blocks. Understanding and leveraging these residual regularities opens up new opportunities for improving storage efficiency, especially in large-scale archival systems and data deduplication pipelines.

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy
dkppl@dkppl.de (D. Köppl); francesco.marino@phd.unict.it (F. P. Marino)
0000-0002-8721-4444 (D. Köppl); 0000-0003-4722-9542 (F. P. Marino)

Public awareness of recompression is present in the context of *double compression*, where applying a general-purpose compressor to already compressed data can lead to compression gain for artificial data[1] but often results in counterproductive results, increasing the file size due to the lack of exploitable redundancy and the overhead of additional compression metadata[2].

From a more general perspective, many approaches to lossless compression can be interpreted through the lens of *textual substitution* [5]. In this paradigm, repeated substrings are replaced by references to a single occurrence, explicitly exploiting structural redundancy rather than purely statistical regularities. Textual substitution underlies several families of compression schemes, including dictionary-based and grammar-based methods, and has been studied in both on-line and off-line settings. Among the latter, greedy textual substitution has been investigated extensively, most notably in the work of Apostolico and Lonardi [6], where substrings are iteratively selected to maximize compression gain. These methods, however, operate on uncompressed text and rely on global substring statistics, resulting in algorithmic designs and objectives that differ substantially from those arising in the recompression setting considered here.

In a similar vein, a closely related perspective arises from the recently introduced framework of substring equation systems (SES) [7]. In this model, a string is represented by a set of equality constraints between substrings together with explicit character assignments, providing a unifying abstraction for copy-based compression schemes. From this viewpoint, recompression can be interpreted as constructing an SES over the compressed bitstream, where detected repetitions induce substring-equality constraints. This connection places our approach within a broader theoretical framework and helps explain the combinatorial nature of substitution selection. This perspective will guide our interpretation of recompression strategies and their limitations. In particular, it provides a natural abstraction for reasoning about substitution selection and its complexity. Building on these perspectives, we investigate whether meaningful redundancy can still be identified in the output of state-of-the-art compressors. We introduce and evaluate a recompression strategy that operates directly on the bit-level representation of compressed files, without requiring partial decompression or format-specific parsing.

The proposed approach, called *Fixed-Length Window* (FLW), relies on a sliding window of constant size that scans the compressed stream and identifies repeated segments. Whenever a repeated window is detected, the algorithm replaces it with a positional reference, in the spirit of Lempel–Ziv schemes. We derive a theoretical bound on the minimum effective window size, ensuring that the recompression process is non-expansive.

We experimentally assess this approach on a variety of datasets compressed with `7zip`. While recompression does not universally guarantee further reduction, our results show that specific configurations—particularly those based on fixed-length windows—can achieve measurable gains on structured inputs. By construction, the FLW method never increases the data size: depending on the presence of repeated windows, it either preserves the input size or yields a reduction. Overall, the experiments confirm that recompression can serve as a complementary layer in modern compression pipelines, providing small but consistent improvements in storage efficiency without sacrificing losslessness.

[1] https://medium.com/@abdulsamie488/the-illusion-of-double-compression-why-gzip-wont-work-miracles-on-already-compressed-data-c683a3f5890a

[2] https://designvault.medium.com/double-compression-the-risks-of-compressing-already-compressed-http-responses-4fef6254489c

1.1. Substring equation systems

We briefly introduce the notion of *substring equation systems* (SESs), recently proposed as a general framework for representing strings via equality constraints between substrings [7]. Let $x \in \Sigma^n$ be a string. A SES for x consists of two types of constraints: substring-equality constraints of the form $x[i..i+\ell-1] = x[j..j+\ell-1]$, and character-assignment constraints of the form $x[k] = c$, for $c \in \Sigma$. An SES represents x if x is the unique string satisfying all constraints, and the size of the system is defined as the total number of constraints. SES provide a unifying abstraction for copy-based compression schemes. In contrast to classical approaches such as Lempel–Ziv or bidirectional macro schemes, SES do not rely on an explicit parsing or directional references; instead, redundancy is expressed purely through equality relations between substrings. In particular, any representation based on copying substrings can be naturally translated into an SES of comparable size.

This framework also offers a natural interpretation of recompression. Given a compressed bitstream, each detected repetition of a substring can be modeled as a substring-equality constraint, while explicitly stored bits correspond to character assignments. Under this view, recompression can be interpreted as constructing an SES over the compressed data, where the goal is to identify a set of constraints that captures residual redundancy.

In general, finding a smallest SES representation amounts to selecting a set of substring equalities that maximizes compression while maintaining global consistency. As we discuss later, this selection problem naturally leads to combinatorial optimization formulations and becomes computationally hard under general interaction models.

2. A naïve fixed length window recompression

We introduce a simple fixed-length window recompression scheme designed to capture residual repetitions in compressed bitstreams while guaranteeing non-expansion. The method is *naïve* in that it relies solely on local window comparisons, without global substring statistics, gain optimization, or non-overlapping substitution selection beyond a single left-to-right scan. The resulting scheme is closely related to the sliding-window recompression method introduced in [8]. We deliberately restate it here in a simplified and self-contained form, and provide a more explicit theoretical foundation, as it will serve as the basic building block for the more refined techniques presented later in the paper.

Given as input a binary stream that has already been compressed by a standard algorithm, the recompressor scans the data using a window of a fixed length w. For each possible window position, the algorithm examines the corresponding substring of length w and compares it to all previously encountered substrings of the same size. This process is conceptually similar to the hashing strategy used in the Rabin–Karp string matching algorithm [9], enabling efficient detection of repeated patterns across the compressed bitstream.

The use of rolling fingerprints makes the implementation Monte Carlo: distinct windows may collide and therefore generate an incorrect back-reference. Assuming fingerprints are uniformly distributed over a universe of size 2^f, where f is the fingerprint length, the probability of any collision among at most n windows is bounded by $\binom{n}{2}/2^f \leq n^2/2^{f+1}$ by the union bound. Thus, choosing $f = \Theta(\log n)$ makes the failure probability polynomially small in n.

Whenever a match is found the algorithm records a reference to the earlier occurrence rather than storing the raw bits again. For positions that do not exhibit repetition, the corresponding bits of the input are emitted directly. As a result, the output alternates between literal data and back-references, in a manner analogous to Lempel–Ziv (LZ) style compression schemes [10], but operating directly on already compressed data. This representation allows to preserve the recoverability of the input stream while reducing its size when redundant fragments exist.

From the perspective of substring equation systems introduced in section 1.1, the fixed-length window (FLW) recompression scheme can be interpreted as constructing a restricted SES over the compressed bitstream B. Each detected repetition of a window of length w induces a substring-equality constraint of the form $B[i..i+w-1] = B[j..j+w-1]$, while emitted literals correspond to character-assignment constraints. In contrast to general SES representations, the constraints produced by FLW all have the same length and are discovered through a local, left-to-right procedure without global optimization. This interpretation clarifies that FLW is a restricted SES and motivates the substitution-selection problem studied in Section 4.

As illustrated in Figure 1, the recompression process identifies repeated windows within the compressed sequence. The first occurrence of a window, starting at position i, is stored explicitly as literal data. When the same substring reappears at positions j and k, instead of writing the raw bits again, the encoder emits back-references (e.g., $j \to i$ and $k \to i$) pointing to the previously stored segment. This mechanism avoids redundant storage of identical fragments and forms the basis of the fixed-length window recompression scheme. Throughout the paper, positions in bitstreams are indexed from 0.

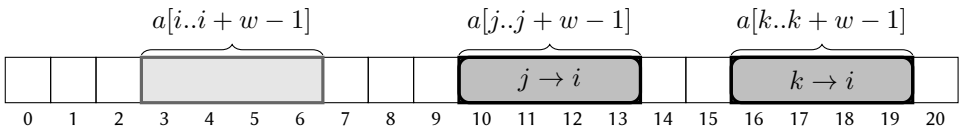

Figure 1: Fixed-length window recompression on a single array $a[0..n-1]$. The window at i represents the first occurrence of $a[i..i+w-1]$. At positions j and k, the same substring reappears, so those segments are not stored as raw bits; the encoder emits references $j \to i$ and $k \to i$ (of length w) instead of literals. Literals are emitted only for positions not covered by a matching window.

Example 1. To illustrate the behavior of the fixed-length window algorithm, consider the following bit vector of length $n = 10$, and choose a window size $w = 3$:

i	0	1	2	3	4	5	6	7	8	9
$B[i]$	0	1	0	1	1	0	0	1	0	1

At the start, the algorithm computes the hash of the first window $B[0..2] = 010$ and stores its starting position 0 in the hash table of already seen patterns. Since this is the first occurrence, the bits in this window are emitted as literals in the output.

The window is then shifted by one position, generating $B[1..3] = 101$, which does not appear in the table. The algorithm stores position 1 and outputs the next literal. The same applies to $B[2..4] = 011$ and $B[3..5] = 110$, both of which are new patterns. When the window reaches $B[4..6] = 100$, this pattern is again new and is inserted into the table.

Sliding forward once more yields the substring $B[5..7] = 001$, also unseen so far. However, the next window is $B[6..8] = 010$, and this pattern has already appeared at position 0. At this point, instead of emitting the three literal bits again, the algorithm outputs the reference $(6 \to 0)$ and skips the next $w - 1 = 2$ bits, since they are covered by the reference itself. The scan then continues from the first uncovered position.

In this example, the substring $B[6..8]$ is not stored a second time; it is encoded compactly through a back-reference. The resulting output therefore consists of literals for each first occurrence of a window and occasional references when repetitions are detected.

For a binary representation Y, let $\mathrm{enc}(Y)$ denote its total length in bits, including literals, references, and any metadata required by the encoding format. The metadata used by the encodings considered in this paper is summarized in Table 2; its total size is independent of n and therefore contributes only $O(1)$ bits.

Lemma 1. Let n be the total input size in bits, and w the window length used by the recompression algorithm. To ensure that positional references can be represented efficiently, w must satisfy the bound $w \geq 2\lceil \log_2 n \rceil$.

Proof. Each back-reference requires storing two positions, each of $\lceil \log_2 n \rceil$ bits for addressing within the input stream. Hence, the total overhead per reference is $2\lceil \log_2 n \rceil$ bits. Compression gain is achievable only when the window is large enough for this overhead to be offset by the repeated content captured within it. Therefore, w must be at least $2\lceil \log_2 n \rceil$ to ensure that representing references is asymptotically more efficient than storing the literals directly. □

In practice, the choice of w defines a trade-off between computational cost and compression ratio. Larger windows increase the likelihood of finding repetitions and therefore improve redundancy capture, but also require more memory and incur higher comparison costs. Conversely, very small windows may fail to identify meaningful overlaps, resulting in no gain or even slight expansion due to reference overhead. Empirical tuning of w around the theoretical lower bound often yields a good balance between efficiency and gain.

Theorem 2 (Non-expansion). Let X be an input bitstream of length n, and let $\mathsf{FLW}(X)$ denote its recompressed representation obtained with window length $w \geq 2\lceil \log_2 n \rceil$. Then $\mathrm{enc}(\mathsf{FLW}(X)) \leq \mathrm{enc}(X) + O(1)$.

Proof. Each uncovered bit is copied verbatim. Each substituted window of length w is replaced by a back-reference of cost $2\lceil \log_2 n \rceil$ bits. Since $w \geq 2\lceil \log_2 n \rceil$, every substitution is non-expansive. Hence the total encoding size cannot increase, except for possible constant-size metadata, and therefore $\mathrm{enc}(\mathsf{FLW}(X)) \leq \mathrm{enc}(X) + O(1)$. □

2.1. Improving the naïve FLW via shared-origin references

While the fixed-length window recompression scheme already reduces redundancy by replacing repeated windows with back-references, additional space savings can be achieved by optimizing how these references are represented.

This modification preserves the scanning strategy of the naïve scheme, but improves its space efficiency by reducing redundancy in the representation of back-references.

In the baseline model, each repeated window generates an individual reference of the form $j \to i$, $k \to i$, and so on, where i is the index of the first occurrence, and j, k are later positions where the same window reappears. This results in repeated storage of the origin index i across multiple references. To avoid this redundancy, we group references that share the same source window. That is, for each unique window occurrence at position i, we record a single origin followed by a list of all positions that refer back to it. Instead of emitting multiple reference pairs, we emit a grouped structure: $i \to \{i_1, i_2, \ldots, i_r\}$, as illustrated in Figure 2. Each origin position (e.g., i) appears only once, followed by a sequence of target positions where the same window reoccurs. This representation compactly encodes repeated patterns while avoiding redundant storage of the origin index.

Lemma 3. Let $G = i \to \{i_1, \ldots, i_r\}$ be a shared-origin group and let $I(G)$ denote the corresponding collection of individual references. Assume that both encodings store the same auxiliary information, whose total cost is c bits (e.g., window length, group cardinality, and format markers). Then $\mathrm{enc}(I(G)) = c + 2r\lceil \log_2 n \rceil$, and $\mathrm{enc}(G) = c + (r+1)\lceil \log_2 n \rceil$. Consequently, $\mathrm{enc}(G) = \mathrm{enc}(I(G)) - (r-1)\lceil \log_2 n \rceil$.

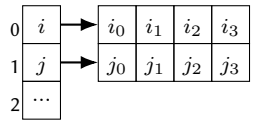

Figure 2: Grouped back-references from multiple origins. Each vertical origin (e.g., i, j) points to a horizontal sequence of repeated positions (e.g., i_0 through i_4). This reduces the redundancy of emitting multiple individual references.

Theorem 4 (Shared-Origin Non-Expansion). Let $\mathsf{SO}(X)$ denote the shared-origin encoding of the FLW representation $\mathsf{FLW}(X)$. Then $\mathrm{enc}(\mathsf{SO}(X)) \le \mathrm{enc}(\mathsf{FLW}(X))$. Consequently, if $\mathrm{enc}(\mathsf{FLW}(X)) \le \mathrm{enc}(X) + O(1)$, then $\mathrm{enc}(\mathsf{SO}(X)) \le \mathrm{enc}(X) + O(1)$.

Proof. By Lemma 3, each group $i \to \{i_1, \ldots, i_r\}$ saves $(r-1)\lceil \log_2 n \rceil$ bits compared to storing the corresponding individual references. Summing over all groups shows that $\mathrm{enc}(\mathsf{SO}(X)) \le \mathrm{enc}(\mathsf{FLW}(X))$. The second claim follows immediately from the non-expansion of FLW. □

3. Recursive fixed-length window

The fixed-length window recompression scheme introduced in Section 2 operates at a single resolution determined by the chosen window length. While this approach already guarantees non-expansion and is able to capture residual redundancy in compressed bitstreams, it is inherently limited to repetitions occurring at a fixed scale. In particular, repetitions that are either smaller than the selected window length or only partially aligned with it may remain undetected. To address this limitation, we extend the fixed-length recompression strategy to a

recursive setting, applying recompression across multiple window scales to capture finer-grained redundancy while preserving the non-expansive property of the base method.

Formally, let B be a compressed bitstream of length n, and let the initial window length w_0 be chosen so as to satisfy the non-expansion condition $w_0 \geq 2\lceil \log_2 n \rceil$. The recursive recompression proceeds in rounds $t = 0, 1, 2, \ldots$, where at each round the fixed-length window algorithm is applied using a window size w_t. The window length is decreased deterministically according to $w_{t+1} = w_t - \alpha$, with $\alpha \geq 1$, until it falls below the threshold required to offset the cost of positional references, for some positive integer α.

In round t, the algorithm scans only those portions of the bitstream that were not replaced by back-references in previous rounds. On these unreduced segments, the same matching and substitution mechanism described in Section 2 is applied using window length w_t. Segments already represented by references are left untouched, ensuring that substitutions performed at earlier stages are preserved. The recursion is applied to the concatenation of all residual literals remaining after a recompression round, not to the residual segments individually.

This recursive process induces a hierarchical decomposition of the bitstream. Early rounds, operating with large window sizes, tend to capture coarse-grained and structurally significant repetitions, whereas later rounds progressively identify finer and more local patterns that may have escaped detection at larger scales. The recompression terminates when $w_t < 2\lceil \log_2 n \rceil$, at which point the overhead of encoding references would dominate any potential gain.

From the perspective of substring equation systems (SES), this recursive procedure can be viewed as constructing a hierarchical system of substring constraints over the compressed bitstream. While the base FLW method introduces equality constraints of fixed length, the recursive scheme progressively adds constraints at decreasing scales, thereby enriching the representation with finer-grained dependencies. In this sense, recursive recompression corresponds to building a multi-scale SES, where earlier constraints capture coarse structure and later ones refine local redundancy. From a computational perspective, the recursive strategy increases the runtime by requiring multiple passes over the data. However, this additional cost is offset by a substantially improved ability to detect nested or partially overlapping repetitions. As a result, the recursive fixed-length window approach provides a principled multi-scale recompression framework that balances compression effectiveness and computational overhead, making it well suited for compressed data streams with heterogeneous residual redundancy.

Overall, the recursive fixed-length window strategy provides a natural multi-scale extension of the base recompression scheme, enabling the detection of residual redundancy at progressively finer resolutions. By organizing substitutions according to decreasing window sizes, the method induces a hierarchical structure that is effective in practice and preserves the non-expansive guarantees of the underlying fixed-length approach.

Corollary 1 (Non-expansion of RFLW). Let $\mathsf{RFLW}(X)$ denote the recursive fixed-length window recompression of X, where every round uses a window length $w_t \geq 2\lceil \log_2 n \rceil$. Then $\mathrm{enc}(\mathsf{RFLW}(X)) \leq \mathrm{enc}(X) + O(1)$.

Proof. Each round applies the FLW transformation to portions of the stream that remain encoded as literals. By Theorem 2, every such transformation is non-expansive. Therefore the encoding size cannot increase from one round to the next. By induction on the recursion depth, $\mathrm{enc}(\mathsf{RFLW}(X)) \leq \mathrm{enc}(X) + O(1)$. □

At the same time, this hierarchy is constructed using a predetermined and greedy schedule of window sizes and substitutions. In particular, prioritizing larger windows implicitly constrains the set of substitutions that can be applied at smaller scales, even when alternative combinations might yield a higher overall compression gain. As a result, the quality of recursive recompression depends not only on the availability of multiple window sizes, but also on how substitutions across different scales interact and are selected. This observation highlights the need for a more global view of substitution selection. In the next section, we move beyond greedy multi-scale strategies and study the problem of selecting substitutions jointly, with the goal of understanding under which conditions optimal or near-optimal selection can be achieved efficiently.

4. Toward optimal window selection

The recursive fixed-length window strategy introduced in the previous section organizes recompression as a multi-scale process, in which substitutions are applied at progressively decreasing window lengths. While this hierarchical approach is effective in practice and preserves the non-expansive guarantees of the base method, it implicitly relies on greedy decisions at multiple levels. In particular, window sizes are selected according to a predetermined schedule, and substitutions are applied as soon as they are detected at a given scale. Such greedy choices are natural and computationally efficient, as they prioritize large-scale repetitions before refining the representation at finer granularities. However, they do not necessarily yield an optimal global selection of substitutions. Two sources of suboptimality arise naturally in this setting. First, substitutions performed at different window lengths may interfere with one another: applying a replacement at a given scale can prevent other potentially beneficial replacements at smaller scales. Second, even substitutions corresponding to the same window content may be mutually incompatible due to overlapping occurrences in the input stream.

Example 2. Consider the string $x = \texttt{abaababaabaababaababababababaa}$, and the window word $y = \texttt{aba}$. The substring y occurs 11 times in x, with starting positions $\{0, 3, 5, 8, 11, 13, 16, 18, 20, 22, 24\}$. Several of these occurrences overlap. For instance, the occurrences at positions 3 and 5, as well as those starting at positions 11 and 13, share common symbols and therefore cannot be selected simultaneously. As a consequence, not all occurrences of y can be used for substitution. In this example, at most 7 pairwise disjoint occurrences can be chosen. One possible selection consists of the occurrences at positions $\{0, 3, 8, 11, 16, 20, 24\}$. Thus, even when all candidate substitutions correspond to the same window word, the selection of replacements is constrained by overlap and requires a global decision.

These overlaps motivate a global view of the replacement selection problem, in which substitutions must be chosen jointly rather than greedily. We now formalize this perspective. Each candidate substitution can be represented as a weighted interval (s, e, z), where s and e denote the start and end positions of the window in the original input stream, and z is the compression gain achieved by applying the substitution. Two substitutions are incompatible if their intervals overlap, since overlapping replacements cannot be applied simultaneously.

From the viewpoint of substring equation systems (SES), each candidate substitution naturally corresponds to a substring-equality constraint of the form $x[i..i + \ell - 1] = x[j..j + \ell - 1]$.

Indeed, a back-reference can be interpreted as replacing an explicit occurrence of a substring by a reference to another occurrence known to be equal. Under this interpretation, a recompressed representation may be viewed as a collection of substring-equality constraints together with the remaining literal symbols needed to reconstruct the original string. This is closely related to the SES framework of Shibata and Bannai [7], where strings are represented through systems of substring equations and character assignments. A key difference is that general SESs allow arbitrary collections of constraints, including overlapping and mutually interacting equations. Such interactions require global consistency reasoning and may propagate information across multiple constraints. In contrast, the present framework deliberately restricts attention to non-overlapping substitutions. Each selected equality constraint is therefore localized to a single interval and does not affect the validity of other selected constraints.

This restriction substantially simplifies the optimization problem. Rather than reasoning about global consistency among interacting equations, we only need to determine which candidate constraints can coexist without overlap. As a consequence, the selection problem reduces to weighted interval scheduling, yielding a polynomial-time optimal algorithm.

In the following, we restrict attention to the case in which incompatibilities between substitutions arise exclusively from interval overlap in the original input stream. Under this incompatibility model, selecting an optimal set of substitutions amounts to choosing a maximum-weight subset of pairwise disjoint intervals. This is exactly the classical *weighted interval scheduling* problem, which admits an optimal solution through dynamic programming [11].

After ordering the m candidate intervals by increasing end position, the optimal solution can be computed in $O(m \log m)$ time in the general case, or in $O(m)$ time when the intervals are already sorted. The latter situation arises naturally in our setting, since candidate substitutions are generated by a left-to-right scan of the input stream.

This formulation provides an optimal alternative to greedy selection strategies when incompatibilities arise solely from interval overlap. It applies both to the case of a fixed window length and to collections of windows of different lengths, as long as substitutions are evaluated on the original input and do not introduce additional semantic interactions.

More general interaction models, such as dependencies introduced by recursive rewriting or substitutions that affect the availability of other candidates beyond simple overlap, lead to substantially richer conflict structures. Such models appear closely related to maximum-weight independent set and other packing problems on general graphs, suggesting a significantly harder optimization problem. Characterizing these interactions and designing efficient approximation or hybrid strategies is left for future work.

We note that, once an optimal set of substitutions has been selected, the actual recompression step can be performed in a single left-to-right pass over the input stream. In contrast to the recursive fixed-length window approach, no further window-size scanning is required at this stage, as all substitution positions are known in advance. As a result, the application phase of the interval-based strategy is computationally simpler than recursive recompression, although this comes at the cost of a more expensive upfront selection step.

4.1. A more general rule-selection model

The interval-based formulation considered above assumes that each candidate substitution corresponds to a single fixed interval in the original input stream, with a fixed associated gain. Under this assumption, incompatibilities arise solely from interval overlap, and the optimal selection problem reduces to weighted interval scheduling. We now consider a more general formulation in which substitutions are defined at the level of *rules*. A rule X is specified by a window length ℓ and a set of starting positions $L \subseteq \{0, \dots, n-\ell\}$ corresponding to occurrences of the same window word. Each rule induces a family of intervals $\mathcal{I}(X) = \{[p, p+\ell-1] : p \in L\}$, and selecting X covers the union $\bigcup_{I \in \mathcal{I}(X)} I$. For instance, under a simple encoding scheme, the gain of X may be expressed as $w(X) = \ell|L| - (\ell + |L|)$, reflecting the cost of storing one origin and $|L|$ references. In contrast to the interval-based model, selecting a rule X may invalidate some occurrences of another rule Y whenever $\bigcup_{I \in \mathcal{I}(X)} I \cap \bigcup_{J \in \mathcal{I}(Y)} J \neq \emptyset$. Consequently, the effective gain of Y depends on the subset of rules selected so far. Thus, gains become selection-dependent and incompatibilities no longer captured by a static interval graph.

Definition 1 (General Rule-Selection Problem (GRSP)). Given a family of rules $\mathcal{X} = \{X_1, \dots, X_m\}$, find a subset $\mathcal{S} \subseteq \mathcal{X}$ maximizing the total gain, where (i) rules are incompatible whenever their covered regions intersect, and (ii) the gain of each rule depends on the occurrences that remain feasible after previous selections.

Let T be a string. A rule X is specified by a window length ℓ, a window word $P_X \in \Sigma^\ell$, and the set L_X of starting positions at which P_X occurs in T. The rule covers the region $C(X) := \bigcup_{p \in L_X} [p, p+\ell-1]$. Given a family of rules $\mathcal{X} = \{X_1, \dots, X_m\}$ with fixed gains $g(X_i)$, the problem Static Rule-Selection asks for a subset $\mathcal{S} \subseteq \mathcal{X}$ maximizing $\sum_{X \in \mathcal{S}} g(X)$ subject to $C(X) \cap C(Y) = \emptyset$ for all distinct $X, Y \in \mathcal{S}$. In what follows, we give a reduction from Independent Set in 3-regular graphs, for which NP-hardness is known [12].

Theorem 5. Static Rule-Selection is NP-hard, even when all rules have the same window length and each rule has exactly three occurrences.

Proof. We reduce from Independent Set on 3-regular graphs. Let $G = (V, E)$ be a 3-regular graph and let k be the target independent set size. We construct a string T and one candidate rule X_v for each vertex $v \in V$, such that two candidate rules overlap if and only if the corresponding vertices are adjacent. For every vertex $v \in V$, introduce a distinct symbol a_v. We also use a separator symbol $\#$. Fix the window length to be $\ell = 3$. For every edge $e = \{u, v\} \in E$, choose an arbitrary orientation, say u before v, and create the edge gadget $H_e = \#a_u\#a_v\#$. The string T is obtained by concatenating all edge gadgets. Between two consecutive edge gadgets, we insert a separator block $\$_i^3$, where each $\$_i$ is a fresh symbol used only in that one separator block. Thus every length-3 substring containing some $\$_i$ occurs only once in T.

For every vertex $v \in V$, define the window word $P_v = \#a_v\#$. Let L_v be the set of all starting positions of P_v in T, and define the candidate rule $X_v = (\ell, L_v)$. The candidate rule family is $\mathcal{X} = \{X_v : v \in V\}$. In particular, separator substrings such as $\$_i^3$, $\#\$_i^2$, or $\$_i^2\#$ are not candidate rules. Moreover, since the symbols $\$_i$ are fresh, such substrings occur only once and hence would have no positive gain under the usual rule-score $g(X) = \ell|L_X| - (\ell + |L_X|)$. By

construction, P_v occurs exactly once in each edge gadget corresponding to an edge incident to v, and nowhere else. Since G is 3-regular, $|L_v| = 3$ for every v.

We now show that two candidate rules X_u and X_v overlap if and only if $\{u, v\} \in E$. First suppose that $\{u, v\} \in E$. In the corresponding edge gadget, up to orientation, we have $\#a_u\#a_v\#$. The occurrence of $P_u = \#a_u\#$ covers the first three positions of this gadget, while the occurrence of $P_v = \#a_v\#$ covers the last three positions. These two intervals overlap in the middle $\#$-position. Therefore $C(X_u) \cap C(X_v) \neq \emptyset$. Conversely, suppose that $C(X_u) \cap C(X_v) \neq \emptyset$. Occurrences belonging to different edge gadgets cannot overlap: distinct edge gadgets are separated by a block $\$_i^3$, and no candidate rule contains any symbol $\$_i$. Thus the overlap must occur inside a single edge gadget. But an edge gadget contains occurrences of $P_z = \#a_z\#$ only for its two endpoint vertices. Hence u and v are the endpoints of the same edge, and therefore $\{u, v\} \in E$. Thus the conflict graph of the constructed rule-selection instance is exactly G. A set $S \subseteq V$ is an independent set in G if and only if the corresponding set of candidate rules $\{X_v : v \in S\}$ is feasible.

Finally, assign every candidate rule the same gain. For example, using the gain expression above, each candidate rule has gain $g(X_v) = 3 \cdot 3 - (3 + 3) = 3$, because $\ell = 3$ and $|L_v| = 3$. Therefore G has an independent set of size at least k if and only if the constructed static rule-selection instance has a feasible solution of total gain at least $3k$. The construction is polynomial in the size of G. Hence, Static Rule-Selection is NP-hard. □

Remark 1. GRSP generalizes Static Rule-Selection by allowing selection-dependent gains through occurrence-removal updates. Hence, GRSP is NP-hard. A full treatment of the dynamic, order-dependent variant requires a precise formalization of gain updates under different orders.

Remark 2. The proof above uses a finite alphabet with one symbol a_v per vertex. If a binary construction is required, each symbol can be replaced by a fixed-length binary code and a delimiter code chosen not to occur inside symbol codes. This scales all window lengths and positions by only a polynomial factor and preserves the overlap pattern used in the reduction.

For this reason, the framework developed in this paper deliberately restricts attention to the interval-based incompatibility model, where each substitution is evaluated independently on the original input and conflicts arise solely from geometric interval overlap. Under this restriction, the conflict graph is an interval graph and optimal selection is solvable in polynomial time via weighted interval scheduling. Exploring approximation algorithms or structural properties of the fully general rule-selection model remains an interesting direction for future work.

Example 3. Consider three candidate substitutions represented by intervals $[0, 4]$ with gain 6, $[3, 7]$ with gain 7, and $[5, 9]$ with gain 5. The first two intervals overlap, as do the second and third, while the first and third are disjoint. A greedy strategy selecting the interval with maximum gain would choose $[3, 7]$ with gain 7. However, the optimal solution consists of selecting $[0, 4]$ and $[5, 9]$, yielding a total gain of 11. This simple example illustrates why global optimization via weighted interval scheduling is necessary.

While the weighted interval scheduling formulation provides a polynomial-time optimal selection strategy, its practical implementation and evaluation are left for future work.

5. Experimental results

All implementations were developed in C++[3]. The goal of the evaluation is to measure the effectiveness of the proposed recompression methods on compressed data. The dataset consists of two parts. First, we considered a collection of heterogeneous real-world files. All inputs were first compressed using `7zip` with maximum compression (`-mx=9`). The resulting `.7z` archives were then processed directly by our recompression algorithms without decompression. For each dataset, we report the relative percentage reduction in size with respect to the original `7zip` archive. Figure 3 summarizes the experimental results. We compare the Fixed-Length Window method (FLW; Section 2) and its recursive variant (RFLW; Section 3).[4] For reference, we also report the median and average longest common prefix (LCP) values of the compressed sequences as a coarse measure of residual redundancy. Negative values in the "Difference" rows indicate compression gains relative to the original archive, whereas positive values indicate expansion. As expected, FLW yields modest but consistent improvements on most datasets, showing that residual repetitions remain even after `7zip` compression.

Size	LCP_{mdn}	LCP_{avg}	FLW	RFLW
5 888 320	21	23	5 881 226	5 868 489
	Difference (%)		−0.120%	−0.336%
22 196 608	23	23	22 194 275	22 192 157
	Difference (%)		−0.010%	−0.021%
38 779 736	24	24	38 779 508	38 778 296
	Difference (%)		−0.000 6%	−0.003 7%
49 936 688	24	24	49 929 427	49 914 361
	Difference (%)		−0.014%	−0.044%
92 080 664	25	25	92 080 226	92 077 922
	Difference (%)		−0.000 04%	−0.002 9%

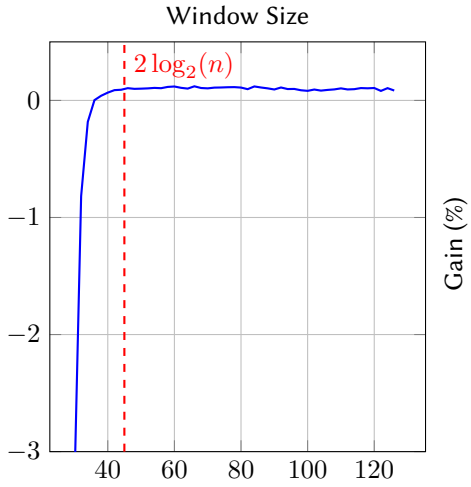

Figure 3: Comparison of recompression methods (left) and gain achieved by the FLW algorithm across different window sizes (right). Reported sizes correspond to the complete serialized compressed representation written to disk, including literals, references, and encoding metadata. The dashed red line marks the theoretical threshold $2\lceil \log_2 n \rceil$.

We also evaluate the impact of the window size w on FLW. Motivated by the theoretical threshold $w \geq 2\lceil \log_2 n \rceil$ established in Section 2, we vary w and measure the resulting compression gain. The right side of Figure 3 reports the results and shows the transition from expansion to compression. In practice, this transition occurs slightly before the theoretical threshold, due to the shared-origin optimization of Section 2.1, which reduces reference overhead and allows compression gains even for window sizes marginally below $2\lceil \log_2 n \rceil$.

[3]The source code is publicly available at https://github.com/marfra99x/recompression.
[4]An experimental comparison with [2] was not possible due to the unavailability of the implementation.

Declaration on Generative AI

Generative AI was used exclusively to assist with the linguistic refinement of the manuscript, including the rephrasing of selected sentences and paragraphs to improve clarity, conciseness, and readability.

References

[1] D. A. Lelewer, D. S. Hirschberg, Data compression, ACM Comput. Surv. 19 (1987) 261–296. URL: https://doi.org/10.1145/45072.45074. doi:10.1145/45072.45074.

[2] B. T. Akgün, A methodology for recursive compression of the compressed data files, in: 2024 4th Interdisciplinary Conference on Electrics and Computer (INTCEC), 2024, pp. 1–6. doi:10.1109/INTCEC61833.2024.10602935.

[3] M. A. Bassiouni, B. Ok, Double encoding - A technique for reducing storage requirement of text, Inf. Syst. 11 (1986) 177–184. URL: https://doi.org/10.1016/0306-4379(86)90006-2. doi:10.1016/0306-4379(86)90006-2.

[4] Z. Li, Y. Ding, C. Yi, A double-compression method for searchable network packets in network forensics and analysis, Comput. Electr. Eng. 119 (2024) 109535. URL: https://doi.org/10.1016/j.compeleceng.2024.109535. doi:10.1016/J.COMPELECENG.2024.109535.

[5] A. Apostolico, S. Lonardi, Compression of biological sequences by greedy off-line textual substitution, in: Data Compression Conference, DCC 2000, Snowbird, Utah, USA, March 28-30, 2000, IEEE Computer Society, 2000, pp. 143–152. URL: https://doi.org/10.1109/DCC.2000.838154. doi:10.1109/DCC.2000.838154.

[6] A. Apostolico, S. Lonardi, Some theory and practice of greedy off-line textual substitution, in: Data Compression Conference, DCC 1998, Snowbird, Utah, USA, March 30 - April 1, 1998, IEEE Computer Society, 1998, pp. 119–128. URL: https://doi.org/10.1109/DCC.1998.672138. doi:10.1109/DCC.1998.672138.

[7] H. Shibata, H. Bannai, String representation in suffixient set size space, 2026. URL: https://arxiv.org/abs/2604.04377. arXiv:2604.04377.

[8] J. L. Bentley, M. D. McIlroy, Data compression with long repeated strings, Inf. Sci. 135 (2001) 1–11. URL: https://doi.org/10.1016/S0020-0255(01)00097-4. doi:10.1016/S0020-0255(01)00097-4.

[9] R. M. Karp, M. O. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (1987) 249–260. URL: https://doi.org/10.1147/rd.312.0249. doi:10.1147/RD.312.0249.

[10] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (1977) 337–343. doi:10.1109/TIT.1977.1055714.

[11] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, B. Schieber, A unified approach to approximating resource allocation and scheduling, J. ACM 48 (2001) 1069–1090. URL: https://doi.org/10.1145/502102.502107. doi:10.1145/502102.502107.

[12] M. Chlebík, J. Chlebíková, Approximation hardness for small occurrence instances of NP-hard problems, in: CIAC, volume 2653 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 152–164.

[13] J. Berstel, Fibonacci Words — A Survey, Springer Berlin Heidelberg, Berlin, Heidelberg,

1986, pp. 13–27. URL: https://doi.org/10.1007/978-3-642-95486-3_2. doi:10.1007/978-3-642-95486-3_2.

Appendix

The full procedure described in Section 2 is formally defined in Algorithm 1.

Algorithm 1: Fixed-Length Window

Input: Bit vector $B[0..n-1]$, window size w
Output: Bit vector B', list R of reference pairs $(i \to j)$

Function FLW(B, w):
  initialize empty hash table tmp // pattern → positions
  initialize empty pair list R // references $(i \to j)$
  initialize empty bit vector B' // output literals
  if $w = 0$ ***or*** $n < w$ **then**
    return (B, R) ;
  $mask \leftarrow 2^w - 1$ // bitmask for window width
  $h \leftarrow 0$;
  for $i = 0$ **to** $w - 1$ **do**
    $h \leftarrow ((h \ll 1) \mid B[i]) \,\&\, mask$;
  append $B[0..w-2]$ to B' ;
  append 0 to $tmp[h]$ // first occurrence at position 0
  $i \leftarrow w - 1$;
  while $i < n$ **do**
    $h \leftarrow ((h \ll 1) \mid B[i]) \,\&\, mask$;
    $pos \leftarrow i - w + 1$ // starting position of current window
    if $h \in tmp$ **then**
      $origin \leftarrow tmp[h][0]$ // first occurrence of this pattern
      append $(pos \to origin)$ to R ;
      // the next $w - 1$ bits are covered by this reference
      for $j = 1$ **to** $w - 1$ ***and*** $i + j < n$ **do**
        $h \leftarrow ((h \ll 1) \mid B[i + j]) \,\&\, mask$;
        $newpos \leftarrow i + j - w + 1$;
        append $newpos$ to $tmp[h]$;
      $i \leftarrow i + w$ // skip positions covered by the reference
    else
      append pos to $tmp[h]$;
      append $B[i]$ to B' // this bit is kept as a literal
      $i \leftarrow i + 1$
  return (B', R) ;

The algorithm maintains a rolling hash of the current window and a hash table mapping each previously observed window to the position of its first occurrence. As the window slides through the input stream, the hash is updated in constant time, and the algorithm checks whether the

newly obtained window has appeared before.

If the pattern is new, the bit at the current position is emitted as a literal and the window starting position is stored in the hash table. If the pattern has been seen previously, the algorithm emits a back-reference $(i \to j)$, skips the next $w - 1$ positions (since they are covered by that reference), and continues from the first uncovered position. This ensures that references do not overlap and that the resulting representation contains exactly one literal for the first occurrence of each distinct window.

Additional Experimental Results

To study the effect of the initial compression strength, we additionally varied the `7zip` compression level from `-mx=1` to `-mx=9`. For real-world data, the recompression gain generally decreases as the compression level increases, indicating that stronger initial compression leaves less residual redundancy. For the synthetic Fibonacci dataset ($\mathrm{Fib}(45)$), the gain instead increases up to approximately `-mx=6` before decreasing again. This behavior is partly explained by practical limits on the window sizes that can be explored, which restrict the maximum detectable repetition length. The corresponding results are shown in Figure 4.

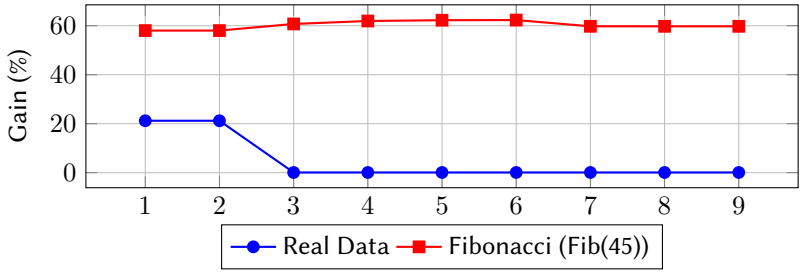

Figure 4: Recompression gain as a function of the `7zip` compression level (`-mx=1` to `-mx=9`). The blue curve corresponds to a representative real-world dataset, while the red curve corresponds to the synthetic Fibonacci word $\mathrm{Fib}(45)$. The reported values are computed from the complete serialized compressed representation written to disk and show the relative percentage size reduction achieved by our method with respect to the original `7zip` archive.

Highly Repetitive Synthetic Data

To evaluate the behavior of FLW on highly structured inputs, we conducted additional experiments using Fibonacci words [13], a classical benchmark for compression algorithms due to their strong self-similarity and deterministic repetition patterns. For increasing values of k, we generated $\mathrm{Fib}(k)$, compressed it using `7zip`, and applied FLW directly to the resulting bitstream. The objective was to assess whether significant redundancy remains detectable even after state-of-the-art compression.

Preliminary results indicate that FLW achieves significantly stronger gains on this class of datasets compared to standard real-world inputs. This confirms the capability of FLW to exploit latent structural redundancy that traditional compressors leave unexposed. Table 1 summarizes the obtained compression performance, reported as in the previous experiment.

Table 1
Recompression results on Fibonacci-word datasets.

Fib(x)	Size	LCP_{mdn}	LCP_{avg}	FLW	RFLW
45	41 990 064	9 847	14 133	16 897 651	16 240 344
			Difference (%)	−59.758%	−61.323%
46	67 939 336	14 766	40 754	26 951 654	26 262 599
			Difference (%)	−60.329%	−61.344%
47	109 927 424	20 331	62 551	48 954 903	42 826 371
			Difference (%)	−55.466%	−61.041%

Metadata and Decodability

For all encoding formats considered in this paper, the compressed representation consists of (i) a constant-size header, followed by (ii) the encoded literals and references.

The header stores only global parameters required by the decoder, such as the encoding format identifier, the window length, and, for RFLW, the initial window length and the decrement step. Since each parameter is an integer bounded by the input length n, each requires at most $\lceil \log_2 n \rceil$ bits. The number of such parameters is constant, so the total header size is $O(\log n)$ bits.

Given the header, the decoder can uniquely determine the encoding format and reconstruct the original bitstream by processing the serialized literals and references from left to right. No additional side information is required.

Table 2
Constant-size metadata included in the encoding length $\text{enc}(\cdot)$. None of these quantities depends on the input length n.

Encoding	Metadata included in $\text{enc}(\cdot)$
FLW	format identifier, window length w
Shared-Origin FLW	format identifier, window length w
RFLW	format identifier, initial window length w_0, step α