Re-Pair in Small Space

Dominik Köppl[1], Tomohiro I[2], Isamu Furuya[3], Yoshimasa Takabatake[2], Kensuke Sakai[2], and Keisuke Goto[4]

[1] Kyushu University, Japan Society for Promotion of Science
dominik.koeppl@inf.kyushu-u.ac.jp,
[2] Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp, takabatake@ai.kyutech.ac.jp,
k_sakai@donald.ai.kyutech.ac.jp
[3] Graduate School of IST, Hokkaido University, Japan
furuya@ist.hokudai.ac.jp
[4] Fujitsu Laboratories Ltd., Kawasaki, Japan
goto.keisuke@fujitsu.com

Abstract. Re-Pair is a grammar compression scheme with favorably good compression rates. The computation of Re-Pair comes with the cost of maintaining large frequency tables, which makes it hard to compute Re-Pair on large scale data sets. As a solution for this problem we present, given a text of length n whose characters are drawn from an integer alphabet with size $\sigma = n^{\mathcal{O}(1)}$, an $\mathcal{O}(n^2) \cap \mathcal{O}(n^2 \lg \log_\tau n \lg\lg\lg n / \log_\tau n)$ time algorithm computing Re-Pair with $\max((n/c)\lg n, n\lceil\lg\tau\rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fix user-defined constant and τ is the sum of σ and the number of non-terminals.

1 Introduction

Re-Pair [16] is a grammar deriving a single string. It is computed by replacing the most frequent bigram in this string with a new non-terminal, recursing until no bigram occurs more than once. Despite this simple-looking description, both the merits and the computational complexity of Re-Pair are intriguing. As a matter of fact, Re-Pair is currently one of the most well-understood grammar schemes.

Besides the seminal work of Larsson and Moffat [16], there are a couple of articles devoted to the compression aspects of Re-Pair: Given a text T of length n whose characters are drawn from an integer alphabet of size $\sigma := n^{\mathcal{O}(1)}$, the output of Re-Pair applied to T is at most $2nH_k(T) + o(n \lg \sigma)$ bits with $k = o(\log_\sigma n)$ when represented naively as a list of character pairs [19], where H_k denotes the empirical entropy of the k-th order. Using the encoding of Kieffer and Yang [12], Ochoa and Navarro [20] could improve the output size to at most $nH_k(T) + o(n \lg \sigma)$ bits. Other encodings were recently studied by Ganczorz [9]. Since Re-Pair is a so-called *irreducible* grammar, its grammar size, i.e., the sum of the symbols on the right hand side of all rules, is upper bounded by $\mathcal{O}(n/\log_\sigma n)$ [12, Lemma 2], which matches the information-theoretic lower bound on the size of a grammar for a string of length n. Comparing this size with the size of the smallest grammar, its approximation ratio has $\mathcal{O}((n/\lg n)^{2/3})$ as an upper bound [5] and $\Omega(\lg n / \lg\lg n)$ as a lower bound [1]. On the practical side, Yoshida and Kida [26] presented an efficient fixed-length code for compressing the Re-Pair grammar.

Although conceived as a grammar for compressing texts, Re-Pair has been successfully applied for compressing trees [17], matrices [23], or images [7]. For different settings or for better compression rates, there is a great interest in modifications to

Dominik Köppl, Tomohiro I, Isamu Furuya, Yoshimasa Takabatake, Kensuke Sakai, Keisuke Goto: Re-Pair in Small Space, pp. 134–147.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

Re-Pair. Charikar et al. [5, Sect. G] gave an easy variation to improve the size of the grammar. Another variant, proposed by Claude and Navarro [6], runs in a user defined working space ($> n \lg n$ bits), and shares with our proposed solution the idea of a table that (a) is stored with the text in the working space, and (b) grows in rounds. The variant of González et al. [11] is specialized on compressing an array of integers delta-encoded (i.e., by the differences of subsequent entries). Sekine et al. [22] provide an adaptive variant whose algorithm divides the input into blocks, and processes each block based on the rules obtained from the grammars of its preceding blocks. Subsequently, Masaki and Kida [18] gave an *online* algorithm producing a grammar mimicking Re-Pair. Ganczorz and Jez [10] modified the Re-Pair grammar by disfavoring the replacement of bigrams that cross Lempel-Ziv-77 (LZ77) [27] factorization borders, which allowed the authors to achieve practically smaller grammar sizes. Recently, Furuya et al. [8] presented a variant, called *MR-Re-Pair*, in which a most frequent maximal repeat is replaced instead of a most frequent bigram.

1.1 Related Work

Re-Pair is a grammar proposed by Larsson and Moffat [16], who presented an algorithm computing it in expected linear time with $5n + 4\sigma^2 + 4\sigma' + \sqrt{n}$ words of working space, where σ' is the number of non-terminals (produced by Re-Pair). González et al. [11, Sect. 4.1] gave another linear time algorithm using $12n + \mathcal{O}(p)$ bytes of working space, where p is the maximum number of distinct bigrams considered at any time. The large space requirements got significantly improved by Bille et al. [3], who presented a randomized linear time algorithm taking $(1 + \epsilon)n + \sqrt{n}$ words on top of the rewriteable text space for a constant ϵ with $0 < \epsilon \leq 1$. Subsequently, they improved their algorithm in [2] to include the text space within the $(1 + \epsilon)n + \sqrt{n}$ words of working space. However, they assume that the alphabet size σ is constant and $\lceil \lg \sigma \rceil \leq w/2$, where w is the machine word size. They also provide a solution for $\epsilon = 0$ running in expected linear time. Recently, Sakai et al. [21] showed how to convert an arbitrary grammar (representing a text) into the Re-Pair grammar in compressed space, i.e., without decompressing the text. Combined with a grammar compression that can process the text in compressed space in a streaming fashion, this result leads to the first Re-Pair computation in compressed space.

1.2 Our Contribution

In this article,[1] we propose an algorithm that computes the Re-Pair grammar in $\mathcal{O}(n^2) \cap \mathcal{O}(n^2 \lg \log_\tau n \lg \lg \lg n / \log_\tau n)$ time (cf. Theorem 3 and Theorem 5) with $\max((n/c) \lg n,\ n \lceil \lg \tau \rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fix user-defined constant and τ is the sum of the alphabet size σ and the number of non-terminals σ'.

Given that the characters of the text are drawn from a large integer alphabet with size $\sigma = \Omega(n)$,[2] the algorithm works in-place. In this setting, we obtain the first non-trivial in-place algorithm, as a trivial approach on a text T of length n would

[1] Parts of this work have already been presented as a poster [15] at the Data Compression Conference 2020 (`https://sigport.org/documents/re-pair-small-space`).

[2] We consider the alphabet as not effective, i.e., a character does not have to appear in the text, as this is a common setting in Unicode texts such as Japanese text. For instance, $n^2 = \Omega(n) \cap n^{\mathcal{O}(1)} \neq \emptyset$ could be such an alphabet size.

compute the most frequent bigram in $\Theta(n^2)$ time by computing the frequency of each bigram $T[i]T[i+1]$ for every integer i with $1 \leq i \leq n-1$, keeping only the most frequent bigram in memory. This sums up to $\mathcal{O}(n^3)$ total time, since there can be $\Theta(n)$ different bigrams considered for replacement by Re-Pair.

To achieve our goal of $\mathcal{O}(n^2)$ total time, we first provide a trade-off algorithm (cf. Lemma 2) finding the d most frequent bigrams in $\mathcal{O}(n^2 \lg d/d)$ time for a trade-off parameter d. We subsequently run this algorithm for increasing values of d, and show that we need to run it $\mathcal{O}(\lg n)$ times, which gives us $\mathcal{O}(n^2)$ time if d is increasing sufficiently fast. Our major tools are appropriate text partitioning, elementary scans, and sorting steps, which we visualize in Section 2.5 by an example, and practically evaluate in Section 2.6. When $\tau = o(n)$, a different approach using word-packing and bit-parallel techniques becomes attractive, leading to an $\mathcal{O}(n \lg \log_\tau n \lg \lg \lg n / \log_\tau n)$ time algorithm, which we explain in Section 3.

1.3 Preliminaries

We use the word RAM model with a word size of $\Omega(\lg n)$ for an integer $n \geq 1$. We work in the restore model [4], in which algorithms are allowed to overwrite the input, as long as they can restore the input to its original form.

Strings. Let T be a text of length n whose characters are drawn from an integer alphabet Σ of size $\sigma = n^{\mathcal{O}(1)}$. A bigram is an element of Σ^2. The *frequency* of a bigram B in T is the number of *non-overlapping* occurrences of B in T, which is at most $|T|/2$. For instance, the frequency of the bigram $\mathtt{aa} \in \Sigma^2$ in the text $T = \mathtt{a} \cdots \mathtt{a}$ consisting of n $\mathtt{a}$'s is $\lfloor n/2 \rfloor$.

Re-Pair. We reformulate the recursive description in the introduction by dividing a Re-Pair construction algorithm into *turns*. Stipulating that T_i is the text after the i-th turn with $i \geq 1$ and $T_0 := T \in \Sigma_0^+$ with $\Sigma_0 := \Sigma$, Re-Pair replaces one of the most frequent bigrams (ties are broken arbitrarily) in T_{i-1} with a non-terminal in the i-th turn. Given this bigram is $\mathtt{bc} \in \Sigma_{i-1}^2$, Re-Pair replaces all occurrences of $\mathtt{bc}$ with a new non-terminal X_i in T_{i-1}, and sets $\Sigma_i := \Sigma_{i-1} \cup \{X_i\}$ with $\sigma_i := |\Sigma_i|$ to produce $T_i \in \Sigma_i^+$. Since $|T_i| \leq |T_{i-1}| - 2$, Re-Pair terminates after $m < n/2$ turns such that $T_m \in \Sigma_m^+$ contains no bigram occurring more than once.

2 Algorithm

A major task for producing the Re-Pair grammar is to count the frequencies of the most frequent bigrams. Our work horse for this task is a frequency table. A *frequency table* in T_i of length f stores pairs of the form $(\mathtt{bc}, x)$, where $\mathtt{bc}$ is a bigram and x the frequency of $\mathtt{bc}$ in T_i. It uses $f \lceil \lg(\sigma_i^2 n_i/2) \rceil$ bits of space since an entry stores a bigram consisting of two characters from Σ_i and its respective frequency, which can be at most $n_i/2$. Throughout this paper, we use an elementary in-place sorting algorithm like heapsort:

Lemma 1 ([25]). *An array of length n can be sorted in-place in $\mathcal{O}(n \lg n)$ time.*

2.1 Trade-Off Computation

By embracing the frequency tables, we present a solution with a trade-off parameter:

Lemma 2. *Given an integer d with $d \geq 1$, we can compute the frequencies of the d most frequent bigrams in a text of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(\max(n, d) n \lg d / d)$ time using $2d \lceil \lg(\sigma^2 n/2) \rceil + \mathcal{O}(\lg n)$ bits.*

Proof. Our idea is to partition the set of all bigrams appearing in T into $\lceil n/d \rceil$ subsets, compute the frequencies for each subset, and finally merge these frequencies. In detail, we partition the text $T = S_1 \cdots S_{\lceil n/d \rceil}$ into $\lceil n/d \rceil$ substrings such that each substring has length d (the last one has a length of at most d). Subsequently, we extend S_j to the left (only if $j > 1$) such that S_j and S_{j+1} overlap by one text position, for $1 \leq j < \lceil n/d \rceil$. By doing so, we take the bigram on the border of two adjacent substrings S_j and S_{j+1} for each $j < \lceil n/d \rceil$ into account. Next, we create two frequency tables F and F', each of length d for storing the frequencies of d bigrams. These tables are at the beginning empty. In what follows, we fill F such that after processing S_i, F stores the most frequent d bigrams among those bigrams occurring in $S_1, \ldots, S_i$ while F' acts as a temporary space for storing candidate bigrams that can enter F.

With F and F', we process each of the n/d substrings S_j as follows: Let us fix an integer j with $1 \leq j \leq \lceil n/d \rceil$. We first put all bigrams of S_j into F' in *lexicographic* order. We can perform this within the space of F' in $\mathcal{O}(d \lg d)$ time since there are at most d different bigrams in S_j. We compute the frequencies of all these bigrams in the *complete* text T in $\mathcal{O}(n \lg d)$ time by scanning the text from left to right while locating a bigram in F' in $\mathcal{O}(\lg d)$ time with a binary search. Subsequently, we interpret F and F' as one large frequency table, sort it with respect to the frequencies while discarding duplicates such that F stores the d most frequent bigrams in $T[1..jd]$. This sorting step can be done in $\mathcal{O}(d \lg d)$ time. Finally, we clear F' and are done with S_j. After the final merge step, we obtain the d most frequent bigrams of T stored in F.

Since each of the $\mathcal{O}(n/d)$ merge steps takes $\mathcal{O}(d \lg d + n \lg d)$ time, we need $\mathcal{O}(\max(d, n) \cdot (n \lg d)/d)$ time. For $d \geq n$, we can build a large frequency table and perform one scan to count the frequencies of all bigrams in T. This scan and the final sorting with respect to the counted frequencies can be done in $\mathcal{O}(n \lg n)$ time.

2.2 Algorithmic Ideas

With Lemma 2, we can compute T_m in $\mathcal{O}(mn^2 \lg d/d)$ time with additional $2d \lceil \lg(\sigma_m^2 n/2) \rceil$ bits[3] of working space on top of the text for a parameter d with $1 \leq d \leq n$. In the following, we show how this leads us to our first algorithm computing Re-Pair:

Theorem 3. *We can compute Re-Pair on a string of length n in $\mathcal{O}(n^2)$ time with $\max((n/c) \lg n, n \lceil \lg \tau \rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space as a rewriteable part in the working space, where $c \geq 1$ is a fixed constant and $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminal symbols.*

In our model, we assume that we can enlarge the text T_i from $n_i \lceil \lg \sigma_i \rceil$ bits to $n_i \lceil \lg \sigma_{i+1} \rceil$ bits without additional extra memory. Our main idea is to store a growing frequency table using the space freed up by replacing bigrams with non-terminals. In detail, we maintain a frequency table F in T_i of length f_k for a growing variable f_k, which is set to $f_0 := \mathcal{O}(1)$ in the beginning. The table F takes $f_k \lceil \lg(\sigma_i^2 n/2) \rceil$ bits,

[3] The variable τ used in the abstract and in the introduction is interchangeable with σ_m, i.e., $\tau = \sigma_m$.

which is $\mathcal{O}(\lg(\sigma^2 n)) = \mathcal{O}(\lg n)$ bits for $k = 0$. When we want to query it for a most frequent bigram, we linearly scan F in $\mathcal{O}(f_k) = \mathcal{O}(n)$ time, which is not a problem since (a) the number of queries is $m \leq n$, and (b) we aim for $\mathcal{O}(n^2)$ overall running time. A consequence is that there is no need to sort the bigrams in F according to their frequencies, which simplifies the following discussion.

Frequency Table F. With Lemma 2, we can compute F in $\mathcal{O}(n \max(n, f_k) \lg f_k / f_k)$ time. Instead of recomputing F on every turn i, we want to recompute it only when it no longer stores a most frequent bigram. However, it is not obvious when this happens as replacing a most frequent bigram during a turn (a) removes this entry in F and (b) can reduce the frequencies of other bigrams in F, making them possibly less frequent than other bigrams not tracked by F. Hence, the variable i for the i-th turn (creating the i-th non-terminal) and the variable k for recomputing the frequency table F the $(k+1)$-st time are loosely connected. We group together all turns with the same f_k and call this group the *k-th round* of the algorithm. At the beginning of each round, we enlarge f_k and create a new F with a capacity for f_k bigrams. Since a recomputation of F takes much time, we want to end a round only if F is no longer useful, i.e., when we no longer can guarantee that F stores a most frequent bigram. To achieve our claimed time bounds, we want to assign all m turns to $\mathcal{O}(\lg n)$ different rounds, which can only be done if f_k grows sufficiently fast.

Algorithm Outline. At the beginning of the k-th round and the i-th turn, we compute the frequency table F storing f_k bigrams, and keep additionally the lowest frequency of F as a threshold t_k, which is treated as a constant during this round. During the computation of the i-th turn, we replace the most frequent bigram (say, $\mathtt{bc} \in \Sigma_i^2$) in the text T_i with a non-terminal X_{i+1} to produce T_{i+1}. Thereafter, we remove $\mathtt{bc}$ from F and update those frequencies in F which got decreased by the replacement of $\mathtt{bc}$ with X_{i+1}, and add each bigram containing the new character X_{i+1} into F if its frequency is at least t_k. Whenever a frequency in F drops below t_k, we discard it. If F becomes empty, we move to the $(k+1)$-st round, and create a new F for storing f_{k+1} frequencies. Otherwise (F still stores an entry), we can be sure that F stores a most frequent bigram. In both cases, we recurse with the $(i+1)$-st turn by selecting the bigram with the highest frequency stored in F. We show in Algorithm 1 a pseudo code of this outlined algorithm. We describe in the following how we update F and how large f_{k+1} can become at least.

2.3 Algorithmic Details

Suppose that we are in the k-th round and in the i-th turn. Let t_k be the lowest frequency in F computed at the beginning of the k-th round. We keep t_k as a constant threshold for the invariant that all frequencies in F are at least t_k during the k-th round. With this threshold we can assure in the following that F is either empty or stores a most frequent bigram. Now suppose that the most frequent bigram of T_i is $\mathtt{bc} \in \Sigma_i^2$, which is stored in F. To produce T_{i+1} (and hence advancing to the $(i+1)$-st turn), we enlarge the space of T_i from $n_i \lceil \lg \sigma_i \rceil$ to $n_i \lceil \lg \sigma_{i+1} \rceil$, and replace all occurrences of $\mathtt{bc}$ in T_i with a new non-terminal X_{i+1}. Subsequently, we would like to take the next bigram of F. For that, however, we need to update the stored frequencies in F. To see this necessity, suppose that there is an occurrence of $\mathtt{abcd}$ with two characters $\mathtt{a}, \mathtt{d} \in \Sigma_i$ in T_i. By replacing $\mathtt{bc}$ with X_{i+1},

Algorithm 1: Algorithmic outline of our proposed algorithm working on a text T with a growing frequency table F. The constants α_i and β_i are explained in Section 2.3. The same section shows that the outer while loop is executed $\mathcal{O}(\lg n)$ times.

```
$k \leftarrow 0, i \leftarrow 0$
$f_0 \leftarrow \mathcal{O}(1)$
$T_0 \leftarrow T$
while highest frequency of a bigram in $T$ is greater than one do        ▷ during the $k$-th round
    $F \leftarrow$ frequency table of Lemma 2 with $d := f_k$
    $t_k \leftarrow$ minimum frequency stored in $F$
    while $F \neq \emptyset$ do                                         ▷ during the $i$-th turn
        bc $\leftarrow$ most frequent bigram stored in $F$
        $T_{i+1} \leftarrow T_i$.replace(bc, $X_{i+1}$)                  ▷ create rule $X_{i+1} \rightarrow$ bc
        $i \leftarrow i + 1$                                             ▷ introduce the $(i+1)$-th turn
        remove all bigrams with frequency lower than $t_k$ from $F$
        add new bigrams to $F$ having $X_i$ as left or right character and a frequency of at
          least $t_k$
    $f_{k+1} \leftarrow f_k + \max(2/\beta_i, (f_k - 1)/(2\beta_i))/\alpha_i$
    $k \leftarrow k + 1$                                                 ▷ introduce the $(k+1)$-th round
Invariant: $i = m$ (the number of non-terminals)
```

1. the frequencies of ab and cd decrease by one[4], and
2. the frequencies of aX_{i+1} and X_{i+1}d increase by one.

Updating F. We can take care of the former changes (1) by decreasing the respective bigram in F (in case that it is present). If the frequency of this bigram drops below the threshold t_k, we remove it from F as there may be bigrams with a higher frequency that are not present in F. To cope with the latter changes (2), we track the characters adjacent to X_{i+1} after the replacement, count their numbers, and add their respective bigrams to F if their frequencies are sufficiently high. In detail, suppose that we have substituted bc with X_{i+1} exactly h times. Consequently, with the new text T_{i+1} we have additionally $h \lg \sigma_{i+1}$ bits of free space[5], which we call D in the following. Subsequently, we scan the text and put the characters of Σ_{i+1} appearing to the left of each of the h occurrences of X_{i+1} into D. After sorting the characters in D lexicographically, we can count the frequency of aX_{i+1} for each character a $\in \Sigma_{i+1}$ preceding an occurrence of X_{i+1} in the text T_{i+1} by scanning D linearly. If the obtained frequency of such a bigram aX_{i+1} is at least as high as the threshold t_k, we insert aX_{i+1} into F, and subsequently discard a bigram with the currently lowest frequency in F if the size of F has become f_k+1. In case that we visit a run of X_{i+1}'s during the creation of D, we must take care of not counting the overlapping occurrences of $X_{i+1}X_{i+1}$. Finally, we can count analogously the occurrences of X_{i+1}d for all characters d $\in \Sigma_i$ succeeding an occurrence of X_{i+1}.

Capacity of F. After the above procedure we have updated the frequencies of F. When F becomes empty, all bigrams stored in F have been replaced or have a frequency

[4] For the border case a = b = c (resp. b = c = d), there is no need to decrement the frequency of ab (resp. cd).

[5] The free space is consecutive after shifting all characters to the left.

that became less than t_k. Subsequently, we end the k-th round and continue with the $(k+1)$-st round by (a) creating a new frequency table F with capacity f_{k+1}, and (b) setting the new threshold t_{k+1} to the minimal frequency in F. In what follows, we (a) analyze in detail when F becomes empty (as this determines the sizes f_k and f_{k+1}), and (b) show that we can compensate the number of discarded bigrams with an enlargement of F's capacity from f_k bigrams to f_{k+1} bigrams for the sake of our aimed total running time.

Next, we analyze how many characters we have to free up (i.e., how many bigram occurrences we have to replace) to gain enough space for storing an additional frequency. Let $\delta_i := \lg(\sigma_{i+1}^2 n_i/2)$ be the number of bits needed to store one entry in F, and let $\beta_i := \min(\delta_i/\lg\sigma_{i+1}, c\delta_i/\lg n)$ be the minimum number of characters that need to be freed to store one frequency in this space. To understand the value of β_i, we look at the arguments of the minimum function in the definition of β_i and simultaneously at the maximum function in our aimed working space of $\max(n\lceil\lg\sigma_m\rceil, (n/c)\lg n) + \mathcal{O}(\lg n)$ bits (cf. Theorem 3):

1. The first item in this maximum function allows us to spend $\lg\sigma_{i+1}$ bits for each freed character such that we obtain space for one additional entry in F after freeing $\delta_i/\lg\sigma_{i+1}$ characters.
2. The second item allows us to use $\lg n$ additional bits after freeing up c characters.[6] Hence, after freeing up $c\delta_i/\lg n$ characters, we have space to store one additional entry in F.

With $\beta_i = \min(\delta_i/\lg\sigma_{i+1}, c\delta_i/\lg n) = \mathcal{O}(\log_\sigma n) \cap \mathcal{O}(\log_n \sigma) = \mathcal{O}(1)$ we have the sufficient condition that replacing a constant number of characters gives us enough space for storing an additional frequency.

If we assume that replacing the occurrences of a bigram stored in F does not decrease the other frequencies stored in F, the analysis is now simple: Since each bigram in F has a frequency of at least two, $f_{k+1} \geq f_k + f_k/\beta_i$. Since $\beta_i = \mathcal{O}(1)$, this lets f_k grow exponentially, meaning that we need $\mathcal{O}(\lg n)$ rounds. In what follows, we show that this is also true in the general case.

Lemma 4. *Given the frequency of all bigrams in F drop below the threshold t_k after replacing the most frequent bigram* `bc`*, then its frequency has to be at least* $\max(2, |F| - 1/2)$*, where $|F| \leq f_k$ is the number of frequencies stored in F.*

Proof. If the frequency of `bc` in T_i is x, then we can reduce at most $2x$ frequencies of other bigrams (both the left character and the right character of each occurrence of `bc` can contribute to an occurrence of another bigram). Since a bigram must occur at least twice in T_i to be present in F, the frequency of `bc` has to be at least $\max(2, (f_k - 1)/2)$ for discarding all bigrams of F.

Suppose that we have enough space available for storing the frequencies of $\alpha_i f_k$ bigrams, where α_i is a constant (depending on σ_i and n_i) such that F and the working space of Lemma 2 with $d = f_k$ can be stored within this space. With β_i and Lemma 4

[6] This additional treatment helps us to let f_k grow sufficiently fast in the first steps to save our $\mathcal{O}(n^2)$ time bound, as for sufficiently small alphabets and large text sizes, $\lg(\sigma^2 n/2)/\lg\sigma = \mathcal{O}(\lg n)$, which means that we might run the first $\mathcal{O}(\lg n)$ turns with $f_k = \mathcal{O}(1)$, and therefore already spend $\mathcal{O}(n^2 \lg n)$ time.

with $|F| = f_k$, we have

$$\begin{aligned} \alpha_i f_{k+1} &= \alpha_i f_k + \max(2/\beta_i, (f_k - 1)/(2\beta_i)) \\ &= \alpha_i f_k \max(1 + 2/(\alpha_i\beta_i f_k), 1 + 1/(2\alpha_i\beta_i) - 1/(2\alpha_i\beta_i f_k)) \\ &\geq \alpha_i f_k(1 + 2/(5\alpha_i\beta_i)) =: \gamma_i\alpha_i f_k \text{ with } \gamma_i := 1 + 2/(5\alpha_i\beta_i), \end{aligned}$$

where we used the equivalence $1+2/(\alpha_i\beta_i f_k) = 1+1/(2\alpha_i\beta_i) - 1/(2\alpha_i\beta_i f_k) \Leftrightarrow 5 = f_k$ to estimate the two arguments of the maximum function.

Since we let f_k grow by a factor of at least $\gamma := \min_{1 \leq i \leq m} \gamma_i > 1$ for each recomputation of F, $f_k = \Omega(\gamma^k)$, and therefore $f_k = \Theta(n)$ after $k = \mathcal{O}(\lg n)$ steps. Consequently, after reaching $k = \mathcal{O}(\lg n)$, we can iterate the above procedure a constant number of times to compute the non-terminals of the remaining bigrams occurring at least twice.

Time Analysis. In total we have $\mathcal{O}(\lg n)$ rounds. At the start of the k-th round, we compute F with the algorithm of Lemma 2 with $d = f_k$ on a text of length at most $n - f_k$ in $\mathcal{O}(n(n - f_k) \cdot \lg f_k / f_k)$ time with $f_k \leq n$. Summing this up, we yield

$$\mathcal{O}\left(\sum_{k=0}^{\mathcal{O}(\lg n)} \frac{n - f_k}{f_k} n \lg f_k\right) = \mathcal{O}\left(n^2 \sum_{k}^{\lg n} \frac{k}{\gamma^k}\right) = \mathcal{O}\big(n^2\big) \text{ time.} \tag{1}$$

In the i-th turn, we update F by decreasing the frequencies of the bigrams affected by the substitution of the most frequent bigram $\mathtt{bc}$ with X_{i+1}. For decreasing such a frequency, we look up its respective bigram with a linear scan in F, which takes $f_k = \mathcal{O}(n)$ time. However, since this decrease is accompanied with a replacement of an occurrence of $\mathtt{bc}$, we obtain $\mathcal{O}(n^2)$ total time by charging each text position with $\mathcal{O}(n)$ time for a linear search in F. With the same argument, we can bound the total time for sorting the characters in D to $\mathcal{O}(n^2)$ overall time: Since we spend $\mathcal{O}(h \lg h)$ time on sorting h characters preceding or succeeding a replaced character, and $\mathcal{O}(f_k) = \mathcal{O}(n)$ time on swapping a sufficiently large new bigram composed of X_{i+1} and a character of Σ_{i+1} with a bigram with the lowest frequency in F, we charge each text position again with $\mathcal{O}(n)$ time. Putting all time bounds together gives the claim of Theorem 3.

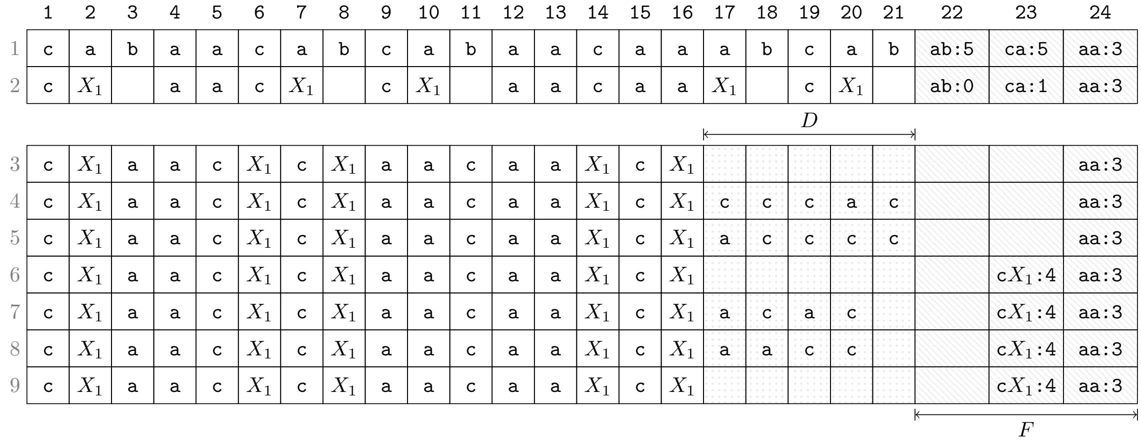
2.4 Storing the Output In-Place

Finally, we show that we can store the computed grammar in text space. More precisely, we want to store the grammar in an auxiliary array A packed at the end of the working space such that the entry $A[i]$ stores the right hand side of the non-terminal X_i, which is a bigram. Thus the non-terminals are represented implicitly as indices of the array A. We therefore need to subtract $2 \lg \sigma_i$ bits of space from our working space $\alpha_i f_k$ after the i-th turn. By adjusting α_i in the above equations, we can deal with this additional space requirement as long as the frequencies of the replaced bigrams are at least three (we charge two occurrences for growing the space of A).

When only bigrams with frequencies of at most two remain, we switch to a simpler algorithm, discarding the idea of maintaining the frequency table F: Suppose that we work with the text T_i. Let λ be a text position, which is 1 in the beginning, but will be incremented in the following turns while holding the invariant that $T[1..\lambda]$ does not contain a bigram of frequency two. We scan $T_i[\lambda..n]$ linearly from left to right and check, for each text position j, whether the bigram $T_i[j]T_i[j+1]$ has another occurrence $T_i[j']T_i[j'+1] = T_i[j]T_i[j+1]$ with $j' > j+1$, and if so,

(a) append $T_i[j]T_i[j+1]$ to A,
(b) replace $T_i[j]T_i[j+1]$ and $T_i[j']T_i[j'+1]$ with a new non-terminal X_{i+1} to transform T_i to T_{i+1}, and
(c) recurse on T_{i+1} with $\lambda := j$ until no bigram with frequency two is left.

The position λ, which we never decrement, helps us to skip over all text positions starting with bigrams with a frequency of one. Thus, the algorithm spends $\mathcal{O}(n)$ time for each such text position, and $\mathcal{O}(n)$ time for each bigram with frequency two. Since there are at most n such bigrams, the overall running time of this algorithm is $\mathcal{O}(n^2)$.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	c	a	b	a	a	c	a	b	c	a	b	a	a	c	a	a	a	b	c	a	b	ab:5	ca:5	aa:3
2	c	X_1		a	a	c	X_1		c	X_1		a	a	c	a	a	X_1		c	X_1		ab:0	ca:1	aa:3
3	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1								aa:3
4	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	c	c	c	a	c			aa:3
5	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	c	c	c	c			aa:3
6	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1							cX_1:4	aa:3
7	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	c	a	c			cX_1:4	aa:3
8	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	a	c	c			cX_1:4	aa:3
9	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1							cX_1:4	aa:3

Figure 1. Step-by-step execution of the first turn of our algorithm on the string T = `cabaacabcabaacaaabcab`. The turn starts with the memory configuration given in Row 1. Positions 1 to 21 are text positions, positions 22 to 24 belong to F ($f_0 = 3$, and it is assumed that a frequency fits into a text entry). Subsequent rows depict the memory configuration during Turn 1. A comment to each row is given in Section 2.5.

2.5 Step-by-Step Execution

Here, we present an exemplary execution of the first turn (of the first round) on the input T = `cabaacabcabaacaaabcab`. We visualize each step of this turn as a row in Fig. 1. A detailed description of each row follows:

Row 1: Suppose that we have computed F, which has the constant number of entries $f_0 = 3$.[7] The highest frequency is five achieved by `ab` and `ca`. The lowest frequency represented in F is three, which becomes the threshold t_0 for a bigram to be present in F such that bigrams whose frequencies drop below t_0 are removed from F. This threshold is a constant for all later turns until F is rebuilt (in the following round). During Turn 1, the algorithm proceeds now as follows:

Row 2: Choose `ab` as a bigram to replace with a new non-terminal X_1 (break ties arbitrarily). Replace every occurrence of `ab` with X_1 while decrementing frequencies in F accordingly to the neighboring characters of the replaced occurrence.

[7] In the later turns when the size f_k becomes larger, F will be put in the text space.

	Prefix Size in KiB				
Data Set	64	128	256	512	1024
Escherichia_Coli	20.68	130.47	516.67	1708.02	10112.47
cere	13.69	90.83	443.17	2125.17	9185.58
coreutils	12.88	75.64	325.51	1502.89	5144.18
einstein.de.txt	19.55	88.34	181.84	805.81	4559.79
einstein.en.txt	21.11	78.57	160.41	900.79	4353.81
influenza	41.01	160.68	667.58	2630.65	10526.23
kernel	20.53	101.84	208.08	1575.48	5067.80
para	20.90	175.93	370.72	2826.76	9462.74
world_leaders	11.92	21.82	167.52	661.52	1718.36
aa···a	0.35	0.92	3.90	14.16	61.74

Table 1. Experimental evaluation of our implementation described in Section 2.6. Table entries are running times in seconds. The last line is the benchmark on the unary string aa···a.

Row 3: Remove from F every bigram whose frequency falls below the threshold. Obtain space for D by aligning the compressed text T_1. (The process of Row 2 and Row 3 can be done simultaneously.)

Row 4: Scan the text and copy each character preceding an occurrence of X_1 in T_1 to D.

Row 5: Sort characters in D lexicographically.

Row 6: Insert new bigrams (consisting of a character of D and X_1) whose frequencies are at least as large as the threshold.

Row 7: Scan the text again and copy each character succeeding an occurrence of X_1 in T_1 to D (symmetric to Row 4).

Row 8: Sort all characters in D lexicographically (symmetric to Row 5).

Row 9: Insert new bigrams whose frequencies are at least as large as the threshold (symmetric to Row 6).

2.6 Implementation

At `https://github.com/koeppl/repair-inplace`, we provide a simplified implementation in C++17. The simplification is that we (a) fix the bit width of the text space to 16 bit, and (b) assume that Σ is the byte alphabet. We further skip the step increasing the bit width of the text from $\lg \sigma_i$ to $\lg \sigma_{i+1}$. This means that the program works as long as the characters of Σ_m fit into 16 bits. The benchmark, whose results are displayed in Table 1, was conducted on a Mac Pro Server with an Intel Xeon CPU X5670 clocked at 2.93GHz running Arch Linux. The implementation was compiled with `gcc-8.2.1` in the highest optimization mode `-O3`. Looking at Table 1, we observe that the running time is super-linear to the input size on all text instances, which we obtained from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl/`). Table 2 gives some characteristics about the used data sets. We see that the number of rounds is the number of turns plus one for every unary string $\texttt{a}^{2^k}$ with an integer $k \geq 1$ since the text contains only one bigram with a frequency larger than two in each round. Replacing this bigram in the text makes F empty such that the algorithm recomputes F after each turn. Note that the number of rounds can drop while scaling the prefix length based on the choice of the bigrams stored in F.

		Turns /1000					Rounds				
		Prefix Size in KiB					Prefix Size in KiB				
Data Set	σ	2^6	2^7	2^8	2^9	2^{10}	2^6	2^7	2^8	2^9	2^{10}
Escherichia_Coli	4	1.8	3.2	5.6	10.3	18.1	6	9	9	12	12
cere	5	1.4	2.8	5.0	9.2	15.1	13	14	14	14	14
coreutils	113	4.7	6.7	10.2	16.1	26.5	15	15	15	14	14
einstein.de.txt	95	1.7	2.8	3.7	5.2	9.7	14	14	15	16	16
einstein.en.txt	87	3.3	3.5	3.8	4.5	8.6	16	15	15	15	17
influenza	7	2.5	3.7	9.5	13.4	22.1	11	12	14	13	15
kernel	160	4.5	8.0	13.9	24.5	43.7	10	11	14	14	13
para	5	1.8	3.2	5.8	10.1	17.6	12	12	13	13	14
world_leaders	87	2.6	4.3	6.1	10.0	42.1	11	11	11	11	14
`aa···a`	1	15	16	17	18	19	16	17	18	19	20

Table 2. Characteristics of our data sets used in Section 2.6. The number of turns and rounds are given for each of the prefix sizes 128, 256, 512, and 1024 KiB of the respective data sets. The number of turns reflecting the number of non-terminals is given in units of thousands. The turns of the unary string `aa···a` are in plain units (not divided by thousand).

3 Bit-Parallel Algorithm

In the case that $\tau = \sigma_m$ is $o(n)$ (and therefore $\sigma = o(n)$), a word-packing approach becomes interesting. We present techniques speeding up previously introduced operations on chunks of $\mathcal{O}(\log_\tau n)$ characters from $\mathcal{O}(\log_\tau n)$ time to $\mathcal{O}(\lg\lg\lg n)$ time. In the end, these techniques allow us to speed up the sequential algorithm of Theorem 3 from $\mathcal{O}(n^2)$ time to the following:

Theorem 5. *We can compute Re-Pair on a string of length n in $\mathcal{O}(n^2 \lg\log_\tau n \lg\lg\lg n / \log_\tau n)$ time with $\max((n/c)\lg n, n\lceil\lg\tau\rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fixed constant and $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminal symbols.*

Note that the $\mathcal{O}(\lg\lg\lg n)$ time factor is due to the popcount function [24, Algo. 1], which has been optimized to a single instruction on modern computer architectures.

3.1 Broadword Search

First, we deal with accelerating the computation of the frequency of a bigram in T by exploiting broadword search thanks to the word RAM model. We start with the search of single characters and subsequently extend this result to bigrams:

Lemma 6. *We can count the occurrences of a character $c \in \Sigma$ in a string of length $\mathcal{O}(\log_\sigma n)$ in $\mathcal{O}(\lg\lg\lg n)$ time.*

See the full version [14] for a proof, which is a variation of broadword searching zero bytes [13, Sect. 7.1.3]. Having Lemma 6, we show that we can compute the frequency of a bigram in T in $\mathcal{O}(n\lg\lg\lg n/\log_\sigma n)$ time. For that, we interpret $T \in \Sigma^n$ of length n as a text $T \in (\Sigma^2)^{\lceil n/2\rceil}$ of length $\lceil n/2\rceil$. Then we partition T into strings fitting into a computer word, and call each string of this partition a *chunk*. For each chunk, we can apply Lemma 6 by treating a bigram $c \in \Sigma^2$ as a single character. The result is, however, not the frequency of the bigram c in general. For computing the frequency a bigram $\mathtt{bc} \in \Sigma^2$, we distinguish the cases $\mathtt{b} \neq \mathtt{c}$ and $\mathtt{b} = \mathtt{c}$.

Case $\mathtt{b} \neq \mathtt{c}$. By applying Lemma 6 to find the character $\mathtt{bc} \in \Sigma^2$ in a chunk S (interpreted as a string of length $\lfloor q/2 \rfloor$ on the alphabet Σ^2), we obtain the number of occurrences of $\mathtt{bc}$ starting at odd positions in S. To obtain this number for all even positions, we apply the procedure to $\mathtt{d}S$ with $\mathtt{d} \in \Sigma \setminus \{\mathtt{b}, \mathtt{c}\}$. Additional care has to be taken at the borders of each chunk matching the last character of the current chunk and the first character of the subsequent chunk with $\mathtt{b}$ and $\mathtt{c}$, respectively.

Case $\mathtt{b} = \mathtt{c}$. This case is more involving as overlapping occurrences of $\mathtt{bb}$ can occur in S, which we must not count. To this end, we watch out for *runs* of $\mathtt{b}$'s, i.e., substrings of maximal lengths consisting of the character $\mathtt{b}$ (here, we consider also maximal substrings of $\mathtt{b}$ with length 1 as a run). We separate these runs into runs ending either at even or at odd positions. We do this because the frequency of $\mathtt{bb}$ in a run of $\mathtt{b}$'s ending at an even (resp. odd) position is the number of occurrences of $\mathtt{bb}$ within this run ending at an even (resp. odd) position. We can compute these positions similarly to the approach for $\mathtt{b} \neq \mathtt{c}$ by first (a) hiding runs ending at even (resp. odd) positions, and then (b) counting all bigrams ending at even (resp. odd) positions. Runs of $\mathtt{b}$'s that are a prefix or a suffix of S are handled individually if S is neither the first nor the last chunk of T, respectively. That is because a run passing a chunk border starts and ends in different chunks. To take care of those runs, we remember the number of $\mathtt{b}$'s of the longest suffix of every chunk, and accumulate this number until we find the end of this run, which is a prefix of a subsequent chunk. With the aforementioned analysis of the runs crossing chunk borders, we can extend this procedure to count the frequency of $\mathtt{bb}$ in T. We conclude:

Lemma 7. *We can compute the frequency of a bigram in a string T of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(n \lg\lg\lg n / \log_\sigma n)$ time.*

3.2 Bit-Parallel Adaption

Similarly to Lemma 2, we present an algorithm computing the d most frequent bigrams, but now with the word-packed search of Lemma 7.

Lemma 8. *Given an integer d with $d \geq 1$, we can compute the frequencies of the d most frequent bigrams in a text of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(n^2 \lg\lg\lg n / \log_\sigma n)$ time using $d \lceil \lg(\sigma^2 n/2) \rceil + \mathcal{O}(\lg n)$ bits.*

Proof. We allocate a frequency table F of length d. For each text position i with $1 \leq i \leq n-1$, we compute the frequency of $T[i]T[i+1]$ in $\mathcal{O}(n \lg\lg\lg n / \log_\sigma n)$ time with Lemma 7. After computing a frequency, we insert it into F if it is one of the d most frequent bigrams among the bigrams we have already computed. We can perform the insertion in $\mathcal{O}(\lg d)$ time if we sort the entries of F by their frequencies, yielding $\mathcal{O}((n \lg\lg\lg n / \log_\sigma n + \lg d)n)$ total time.

Studying the final time bounds of Eq. (1) for the sequential algorithm of Section 2, we see that we spend $\mathcal{O}(n^2)$ time in the first turn, but spend less time in later turns. Hence, we want to run the bit-parallel algorithm only in the first few turns until f_k becomes so large that the benefits of running Lemma 2 outweigh the benefits of the bit-parallel approach of Lemma 8. In detail, for the k-th round, we set $d := f_k$ and run the algorithm of Lemma 8 on the current text if d is sufficiently small, or otherwise the algorithm of Lemma 2. In total, we yield

$$\mathcal{O}\left(\sum_{k=0}^{\mathcal{O}(\lg n)} \min\left(\frac{n-f_k}{f_k} n \lg f_k, \frac{(n-f_k)^2 \lg\lg\lg n}{\log_\tau n}\right)\right) = \mathcal{O}\left(n^2 \sum_{k=0}^{\lg n} \min\left(\frac{k}{\gamma^k}, \frac{\lg\lg\lg n}{\log_\tau n}\right)\right)$$
$$= \mathcal{O}\left(\frac{n^2 \lg \log_\tau n \lg\lg\lg n}{\log_\tau n}\right) \text{ time in total,} \tag{2}$$

where $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminals, and $k/\gamma^k > \lg\lg\lg n / \log_\tau n \Leftrightarrow k = \mathcal{O}(\lg(\lg n/(\lg \tau \lg\lg\lg n)))$.

To obtain the claim of Theorem 5, it is left to show that the k-th round with the bit-parallel approach uses $\mathcal{O}(n^2 \lg\lg\lg n / \log_\tau n)$ time, as we now want to charge each text position with $\mathcal{O}(n/\log_\tau n)$ time with the same amortized analysis as after Eq. (1). We target $\mathcal{O}(n/\log_\tau n)$ time for

(1) replacing all occurrences of a bigram,
(2) shifting freed up text space to the right,
(3) finding the bigram with the highest or lowest frequency in F,
(4) updating or exchanging an entry in F, and
(5) looking up the frequency of a bigram in F.

Items (1) and (2) can be solved by applying elementary bit-parallel techniques.[8]

For the remaining points, our trick is to represent F by a minimum and a maximum heap, both realized as array heaps. For the space increase, we have to lower α_i (and γ_i) adequately. Each element of an array heap stores a frequency and a pointer to a bigram stored in a separate array B storing all bigrams consecutively. A pointer array P stores pointers to the respective frequencies in both heaps for each bigram of B. The total data structure can be constructed at the beginning of the k-th round in $\mathcal{O}(f_k)$ time, and hence does not worsen the time bounds. While B solves Item (5), the two heaps with P solve Items (3) and (4) even in $\mathcal{O}(\lg f_k)$ time.

In case that we want to store the output in working space, we follow the description of Section 2.4, where we now use word-packing to find the second occurrence of a bigram in T_i in $\mathcal{O}(n/\log_{\sigma_i} n)$ time.

4 Conclusion

In this article, we proposed an algorithm computing Re-Pair in-place in sub-quadratic time for small alphabet sizes. Our major tools are simple, which allowed us to parallelize our algorithm or adapt it in the external memory model.

Acknowledgments

This work is funded by the JSPS KAKENHI Grant Numbers JP18F18120 (Dominik Köppl), 19K20213 (Tomohiro I) and 18K18111 (Yoshimasa Takabatake), and the JST CREST Grant Number JPMJCR1402 including AIP challenge program (Keisuke Goto).

[8] A detailed description is found in the full version of this paper [14].

References

1. H. Bannai, M. Hirayama, D. Hucke, S. Inenaga, A. Jez, M. Lohrey, and C. P. Reh: *The smallest grammar problem revisited.* arXiv 1908.06428, 2019.
2. P. Bille, I. L. Gørtz, and N. Prezza: *Practical and effective Re-Pair compression.* arXiv 1704.08558, 2017.
3. P. Bille, I. L. Gørtz, and N. Prezza: *Space-efficient Re-Pair compression*, in Proc. DCC, 2017, pp. 171–180.
4. T. M. Chan, J. I. Munro, and V. Raman: *Selection and sorting in the "restore" model.* ACM Trans. Algorithms, 14(2) 2018, pp. 11:1–11:18.
5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat: *The smallest grammar problem.* IEEE Trans. Information Theory, 51(7) 2005, pp. 2554–2576.
6. F. Claude and G. Navarro: *Fast and compact web graph representations.* TWEB, 4(4) 2010, pp. 16:1–16:31.
7. P. De Luca, V. M. Russiello, R. Ciro Sannino, and L. Valente: *A study for image compression using Re-Pair algorithm.* arXiv 1901.10744, 2019.
8. I. Furuya, T. Takagi, Y. Nakashima, S. Inenaga, H. Bannai, and T. Kida: *MR-RePair: Grammar compression based on maximal repeats*, in Proc. DCC, 2019, pp. 508–517.
9. M. Ganczorz: *Entropy lower bounds for dictionary compression*, in Proc. CPM, vol. 128 of LIPIcs, 2019, pp. 11:1–11:18.
10. M. Ganczorz and A. Jez: *Improvements on Re-Pair grammar compressor*, in Proc. DCC, 2017, pp. 181–190.
11. R. González, G. Navarro, and H. Ferrada: *Locally compressed suffix arrays.* ACM Journal of Experimental Algorithmics, 19(1) 2014.
12. J. C. Kieffer and E. Yang: *Grammar-based codes: A new class of universal lossless source codes.* IEEE Trans. Information Theory, 46(3) 2000, pp. 737–754.
13. D. E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley, 12th ed., 2009.
14. D. Köppl, T. I, I. Furuya, Y. Takabatake, K. Sakai, and K. Goto: *Re-pair in-place.* arXiv 1908.04933, 2019.
15. D. Köppl, T. I, I. Furuya, Y. Takabatake, K. Sakai, and K. Goto: *Re-pair in small space*, in Proc. DCC, 2020, p. 377.
16. N. J. Larsson and A. Moffat: *Offline dictionary-based compression*, in Proc. DCC, 1999, pp. 296–305.
17. M. Lohrey, S. Maneth, and R. Mennicke: *XML tree structure compression using RePair.* Inf. Syst., 38(8) 2013, pp. 1150–1167.
18. T. Masaki and T. Kida: *Online grammar transformation based on Re-Pair algorithm*, in Proc. DCC, 2016, pp. 349–358.
19. G. Navarro and L. M. S. Russo: *Re-Pair achieves high-order entropy*, in Proc. DCC, 2008, p. 537.
20. C. Ochoa and G. Navarro: *RePair and all irreducible grammars are upper bounded by high-order empirical entropy.* IEEE Trans. Information Theory, 65(5) 2019, pp. 3160–3164.
21. K. Sakai, T. Ohno, K. Goto, Y. Takabatake, T. I, and H. Sakamoto: *RePair in compressed space and time*, in Proc. DCC, 2019, pp. 518–527.
22. K. Sekine, H. Sasakawa, S. Yoshida, and T. Kida: *Adaptive dictionary sharing method for Re-Pair algorithm*, in Proc. DCC, 2014, p. 425.
23. Y. Tabei, H. Saigo, Y. Yamanishi, and S. J. Puglisi: *Scalable partial least squares regression on grammar-compressed data matrices*, in Proc. SIGKDD, 2016, pp. 1875–1884.
24. S. Vigna: *Broadword implementation of rank/select queries*, in Proc. WEA, vol. 5038 of LNCS, 2008, pp. 154–168.
25. J. W. J. Williams: *Algorithm 232 - heapsort.* Communications of the ACM, 7(6) 1964, pp. 347–348.
26. S. Yoshida and T. Kida: *Effective variable-length-to-fixed-length coding via a Re-Pair algorithm*, in Proc. DCC, 2013, p. 532.
27. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.