Space-Efficient Re-Pair Compression

Philip Bille[†], Inge Li Gørtz[†], and Nicola Prezza[†*]

Technical University of Denmark, DTU Compute
{phbi,inge,npre}@dtu.dk

Abstract

Re-Pair [5] is an effective grammar-based compression scheme achieving strong compression rates in practice. Let n, σ, and d be the text length, alphabet size, and dictionary size of the final grammar, respectively. In their original paper, the authors show how to compute the Re-Pair grammar in expected linear time and $5n + 4\sigma^2 + 4d + \sqrt{n}$ words of working space on top of the text. In this work, we propose two algorithms improving on the space of their original solution. Our model assumes a memory word of $\lceil \log_2 n \rceil$ bits and a re-writable input text composed by n such words. Our first algorithm runs in expected $\mathcal{O}(n/\epsilon)$ time and uses $(1+\epsilon)n+\sqrt{n}$ words of space on top of the text for any parameter $0 < \epsilon \leq 1$ chosen in advance. Our second algorithm runs in expected $\mathcal{O}(n \log n)$ time and improves the space to $n + \sqrt{n}$ words.

1 Introduction

Re-Pair (short for recursive pairing) is a grammar-based compression invented in 1999 by Larsson and Moffat [5]. Re-Pair works by replacing a most frequent pair of symbols in the input string by a new symbol, reevaluating all new frequencies on the resulting string, and then repeating the process until no pairs occur more than once. Specifically, on a string S, Re-Pair works as follows. (1) It identifies the most frequent pair of adjacent symbols ab. If all pairs occur once, the algorithm stops. (2) It adds the rule $A \to ab$ to the dictionary, where A is a new symbol not appearing in S. (3) It repeats the process from step (1).

Re-Pair achieves strong compression ratios in practice and in theory [2, 4, 7, 10]. Re-Pair has been used in wide range of applications, e.g., graph representation [2], data mining [9], and tree compression [6].

Let n, σ, and d denote the text length, the size of the alphabet, and the size of the dictionary grammar (i.e. number of nonterminals), respectively. Larsson et al. [5] showed how to implement Re-Pair in $O(n)$ expected time and $5n + 4\sigma^2 + 4d + \sqrt{n}$ words of space in addition to the text (for simplicity, we ignore any additive $+O(1)$ terms in all space bounds). The space overhead is due to several data structures used to track the pairs to be replaced and their frequencies. As noted by several authors this makes Re-Pair problematic to apply on large data, and various workarounds have been devised (see e.g. [2, 4, 10]).

[†] Partially supported by the Danish Research Council (DFF 4005-00267, DFF 1323-00178)

[*] This work was done while the author was a PhD student at the University of Udine, Italy

Surprisingly, the above bound of the original paper remains the best known complexity for computing the Re-Pair compression. In this work, we propose two algorithms that significantly improve this bound. As in the previous work we assume a standard unit cost RAM with memory words of $\lceil \log_2 n \rceil$ bits and that the input string is given in n such word. Furthermore, we assume that the input string is *re-writeable*, that is, the algorithm is allowed to modify the input string during execution, and we only count the space used in addition to this string in our bounds. Since Re-Pair is defined by repeated re-writing operations, we believe this is a natural model for studying this type of compression scheme. Note that we can trivially convert any algorithm with a re-writeable input string to a read-only input string by simply copying the input string to working memory, at the cost of only n extra words of space. We obtain the following result:

Theorem 1. *Given a re-writeable string S of length n we can compute the Re-Pair compression of S in*

(i) $\mathcal{O}(n/\epsilon)$ expected time and $(1+\epsilon)n + \sqrt{n}$ words of space for any $0 < \epsilon \leq 1$, or

(ii) $\mathcal{O}(n \log n)$ expected time and $n + \sqrt{n}$ words.

Note that since $\epsilon = O(1)$ the time in Thm. 1(i) is always at least $\Omega(n)$. For any constant ϵ, (i) matches the optimal linear time bound of Larsson and Moffat [5], while improving the leading space term by almost $4n$ words to $(1+\epsilon)n + \sqrt{n}$ words (with careful implementation it appears that [5] may be implemented to exploit a re-writeable input string. If so, our improvement is instead almost $3n$ words). Thm. 1(ii) further improves the space to $n + \sqrt{n}$ at the cost of increasing time by a logarithmic factor. By choosing $1/\epsilon = o(\log n)$ the time in (i) is faster than (ii) at the cost of a slight increase in space. For instance, with $\epsilon = 1/\log\log n$ we obtain $\mathcal{O}(n \log\log n)$ time and $n + n/\log\log n + \sqrt{n}$ words.

Our algorithm consists of two main phases: high-frequency and low-frequency pair processing. We define a *high-frequency* (resp. *low frequency*) pair as a character pair appearing at least (resp. less than) $\lceil \sqrt{n}/3 \rceil$ times in the text (we will clarify later the reason for using constant 3). Note that there cannot be more than $3\sqrt{n}$ distinct high-frequency pairs. Both phases use two data structures: a queue $\mathcal{Q}$ storing character pairs (prioritized by frequency) and an array TP storing text positions sorted by character pairs. $\mathcal{Q}$'s elements point to ranges in TP corresponding to all occurrences of a specific character pair. In Section 4.2 we show how we can sort in-place and in linear time any subset of text positions by character pairs. The two phases work exactly in the same way, but use two different implementations for the queue giving different space/time tradeoffs for operations on it. In both phases, we extract (high-frequency/low-frequency) pairs from $\mathcal{Q}$ (from the most to least frequent) and replace them in the text with fresh new dictionary symbols.

When performing a pair replacement $A \rightarrow ab$, for each text occurrence of ab we replace a with A and b with the blank character '␣'. This strategy introduces a potential problem: after several replacements, there could be long (super-constant size) runs of blanks. This could increase the cost of reading pairs in the text by too much. In Section 4.1 we show how we can perform pair replacements while keeping the cost of skipping runs of blanks constant.

2 Preliminaries

Let n be the input text's length. Throughout the paper we assume a memory word of size $\lceil \log_2 n \rceil$ bits, and a rewritable input text T on an alphabet Σ composed by n such words. In this respect, the working space of our algorithms is defined as the amount of memory used *on top* of the input. For reasons explained later, we reserve two characters (*blank* symbols) denoted as '*' and '_'. We encode these characters with the integers $n-2$ and $n-1$, respectively[1].

The Re-Pair compression scheme works by replacing character pairs (with frequency at least 2) with fresh new symbols. We use the notation $\mathcal{D}$ to indicate the *dictionary* of such new symbols, and denote by $\bar{\Sigma}$ the extended alphabet $\bar{\Sigma} = \Sigma \cup \mathcal{D}$. It is easy to prove (by induction[2] on n) that $|\bar{\Sigma}| \leq n$: it follows that we can fit both alphabet characters and dictionary symbols in $\lceil \log_2 n \rceil$ bits. The output of our algorithms consists in a set of rules of the form $X \to AB$, with $A, B \in \bar{\Sigma}$ and $X \in \mathcal{D}$. Our algorithms stream the set of rules directly to the output (e.g. disk), so we do not count the space to store them in main memory.

3 Algorithm

We describe our strategy top-down: first, we introduce the queue $\mathcal{Q}$ as a blackbox, and use it to describe our main algorithm. In the next sections we describe the high-frequency and low-frequency pair processing queues implementations.

3.1 The queue as a blackbox

Our queues support the following operations:
- **new_low_freq_queue**$(\mathbf{T}, \mathbf{TP})$. Return the low-frequency pairs queue.
- **new_high_freq_queue**$(\mathbf{T}, \mathbf{TP})$. Return the high-frequency pairs queue.

[1]If the alphabet size is $|\Sigma| < n-1$, then we can reserve the codes $n-2$ and $n-1$ without increasing the number of bits required to write alphabet characters. Otherwise, if $|\Sigma| \geq n-1$ note that the two (or one) alphabet characters with codes $n-2 \leq x, y \leq n-1$ appear in at most two text positions i_1 and i_2, let's say $T[i_1] = x$ and $T[i_2] = y$. Then, we can overwrite $T[i_1]$ and $T[i_2]$ with the value 0 and store separately two pairs $\langle i_1, x\rangle$, $\langle i_2, y\rangle$. Every time we read a value $T[j]$ equal to 0, in constant time we can discover whether $T[j]$ contains 0, x, or y. Throughout the paper we will therefore assume that $|\Sigma| \leq n$ and that characters from $\Sigma \cup \{*, \_\}$ fit in $\lceil \log_2 n \rceil$ bits.

[2]For $n = 2$ the result is trivial. To carry out the inductive step, consider how $|\Sigma|$, $|\mathcal{D}|$, and n grow when extending the text by one character. Three cases can appear: (i) we append a new alphabet symbol, (ii) we append an existing alphabet symbol and the introduced pair's frequency is equal to 2, (iii) we append an existing alphabet symbol and the new pair's frequency is different than 2.

- $\mathcal{Q}[\mathbf{ab}]$, $ab \in \bar{\Sigma}^2$. If ab is in the queue, return a triple $\langle P_{ab}, L_{ab}, F_{ab} \rangle$, with $L_{ab} \geq F_{ab}$ such that: (i) ab has frequency F_{ab} in the text, and (ii) all text occurrences of ab are contained in $TP[P_{ab}, \dots, P_{ab} + L_{ab} - 1]$. Note that—for reasons explained later—L_{ab} can be strictly greater than F_{ab}.
- $\mathcal{Q}.\mathbf{max}()/\mathcal{Q}.\mathbf{min}()$: return the pair ab in $\mathcal{Q}$ with the highest/lowest F_{ab}.
- $\mathcal{Q}.\mathbf{remove}(\mathbf{ab})$: delete ab from $\mathcal{Q}$.
- $\mathcal{Q}.\mathbf{contains}(\mathbf{ab})$: return true iff $\mathcal{Q}$ contains pair ab.
- $\mathcal{Q}.\mathbf{size}()$ return the number of pairs stored in $\mathcal{Q}$.
- $\mathcal{Q}.\mathbf{decrease}(\mathbf{ab})$: decrease F_{ab} by one.
- $\mathcal{Q}.\mathbf{synchronize}(\mathbf{AB})$. If $F_{AB} < L_{AB}$, then $TP[P_{AB}, \dots, P_{AB} + L_{AB} - 1]$ contains occurrences of pairs $XY \neq AB$ (and/or blank positions). The procedure sorts $TP[P_{AB}, \dots, P_{AB} + L_{AB} - 1]$ by character pairs (ignoring positions containing a blank) and, for each such XY, removes the least frequent pair in $\mathcal{Q}$ and creates a new queue element for XY pointing to the range in TP corresponding to the occurrences of XY. If XY is less frequent than the least frequent pair in $\mathcal{Q}$, XY is not inserted in the queue. Before exiting, the procedure re-computes P_{AB} and L_{AB} so that $TP[P_{AB}, \dots, P_{AB} + L_{AB} - 1]$ contains all and only the occurrences of AB in the text (in particular, $L_{AB} = F_{AB}$).

3.2 Main algorithm

Our main procedure taking as input the text T and computing its RePair grammar works as follows. First, we initialize global variables n (T's length) and $X = |\Sigma|$ (next free dictionary symbol). We then start replacing pairs in two phases: high-frequency and low-frequency pair processing. The high-frequency pair processing phase repeats the following loop until the highest pair frequency in the text becomes smaller than $\sqrt{n}/3$. We initialize array TP containing T's positions sorted by pairs. We create the high-frequency queue $\mathcal{Q}$ by calling $new\_high\_freq\_queue(T, TP)$. Then, we call $substitution\_round(\mathcal{Q})$ (see next section) until $\mathcal{Q}$ is empty. Finally, we free the memory allocated for $\mathcal{Q}$ and TP and we compact T's characters by removing blanks. The low-frequency pair processing phase works exactly as above, except that: (i) we build the queue with $new\_low\_freq\_queue(T, TP)$ and (ii) we repeat the main loop until the highest pair frequency in the text becomes smaller than 2.

3.3 Replacing a pair

In Algorithm 1 we describe the procedure substituting the most frequent pair in the text with a fresh new dictionary symbol. We use this procedure in the main algorithm to compute the re-pair grammar. Variables T (the text), TP (array of text positions), and X (next free dictionary symbol) are global, so we do not pass them from the main algorithm to Algorithm 1. Note that—in Algorithm 1—new pairs appearing after a substitution can be inserted in $\mathcal{Q}$ only inside procedure $\mathcal{Q}.synchronize$ at Lines 14, and 15. However, the operation at Line 14 is executed only under a certain condition. As discussed in the next sections, this trick allows us to amortize operations while preserving correctness of the algorithm. In Lines 4, 5, and 12 of Algorithm 1 we assume that—if necessary—we are skipping runs of blanks while extracting text characters (constant time, see Section 4.1). In Line 5 we extract AB and the two symbols x, y preceding and following it (skipping runs of blanks if necessary). In Line 12, we extract a text substring s composed by X and the symbol preceding it (skipping runs of blanks if necessary). After this, we replace each X with AB in s and truncate s to its suffix of length 3. This is required since we need to reconstruct AB's context *before* the replacement took place. Moreover, note that the procedure could return BAB if we replaced a substring $ABAB$ with XX.

Algorithm 1: *substitution_round*($\mathcal{Q}$)

```
input    : The queue $\mathcal{Q}$
behavior: Pop the most frequent pair from $\mathcal{Q}$ and replace its occurrences with a new symbol
1  $AB \leftarrow \mathcal{Q}.max()$;
2  $\overline{ab} \leftarrow \mathcal{Q}.min()$;                /* global variable storing least frequent pair */
3  output $X \rightarrow AB$;                                    /* output new rule */
4  for $i = TP[P_{AB}], \dots, TP[P_{AB} + L_{AB} - 1]$ and $T[i, i+1] = AB$ do
5      $xABy \leftarrow get\_context(T, i)$;           /* $AB$'s context (before replacement) */
6      $replace(T, i, X)$;                              /* Replace $X \rightarrow AB$ at position $i$ in $T$ */
7      if $\mathcal{Q}.contains(xA)$ then
8          $\mathcal{Q}.decrease(xA)$;
9      if $\mathcal{Q}.contains(By)$ then
10         $\mathcal{Q}.decrease(By)$;
11 for $i = TP[P_{AB}], \dots, TP[P_{AB} + L_{AB} - 1]$ and $T[i] = X$ do
12     $xAB \leftarrow get\_context'(T, i)$;                            /* $X$'s left context */
13     if $\mathcal{Q}.contains(xA)$ and $F_{xA} \le L_{xA}/2$ then
14         $\mathcal{Q}.synchronize(xA)$;
15 $\mathcal{Q}.synchronize(AB)$;                 /* Find new pairs in $AB$'s occurrences list */
16 $\mathcal{Q}.remove(AB)$;
17 $X \leftarrow X + 1$;                                         /* New dictionary symbol */
```

3.4 Amortization: correctness and complexity

Assuming the correctness of the queue implementations (see next sections), all we are left to show is the correctness of our amortization policy at Lines 13 and 14 of Algorithm 1. More formally: in Algorithm 1, replacements create new pairs; however, to amortize operations we postpone the insertion of such pairs in the queue (Line 14 of Algorithm 1). To prove the correctness of our algorithm, we need to show that every time we pick the maximum AB from $\mathcal{Q}$ (Line 1, Algorithm 1), AB is the pair with the highest frequency in the text (i.e. all postponed pairs have lower frequency than AB). Suppose, by contradiction, that at Line 1 of Algorithm 1 we pick pair AB, but the highest-frequency pair in the text is $CD \neq AB$. Since CD is not in $\mathcal{Q}$, we have that (i) CD appeared after some substitution $D \rightarrow zw$ which generated occurrences of CD in portions of the text containing Czw, and[3] (ii) $F_{Cz} > L_{Cz}/2$, otherwise the synchronization step at Line 14 of Algorithm 1 ($\mathcal{Q}.synchronize(Cz)$) would have been executed, and CD would have been inserted in $\mathcal{Q}$. Note that all occurrences of CD are contained in $TP[P_{Cz}, \dots, P_{Cz} + L_{Cz} - 1]$. Inequality $F_{Cz} > L_{Cz}/2$ means that *more than half* of the entries $TP[P_{Cz}, ..., P_{Cz} + L_{Cz} - 1]$ contain an occurrence of Cz, which implies than *less than half* of such entries contain occurrences of pairs different than Cz (in particular CD, since $D \neq z$). This, combined with the fact that all occurrences of CD are stored in $TP[P_{Cz}, ..., P_{Cz} + L_{Cz} - 1]$, yields $F_{CD} \leq L_{Cz}/2$. Then, $F_{CD} \leq L_{Cz}/2 < F_{Cz}$ means that Cz has a higher frequency than CD. This leads to a contradiction, since we assumed that CD was the pair with the highest frequency in the text.

Note that operations $\mathcal{Q}.synchronize(xA)$ and $\mathcal{Q}.synchronize(AB)$ at Lines 14 and 15 scan xA's and AB's occurrences list ($\Theta(L_{xA})$ and $\Theta(L_{AB})$ time). However, to keep time under control, we are allowed to spend only time proportional to F_{AB}. Since L_{xA} and L_{AB} could be much bigger than F_{AB}, we need to show that our strategy amortizes operations. Consider an occurrence $xABy$ of AB in the text. After replacement $X \rightarrow AB$, this text substring becomes xXy. In Lines 8-10 we decrease by one in constant time the two frequencies F_{xA} and F_{By} (if they are stored in $\mathcal{Q}$). Note: we manipulate just F_{xA} and F_{By}, and not the actual intervals associated with these two pairs. As a consequence, for a general pair ab in $\mathcal{Q}$, values F_{ab} and L_{ab} do not always coincide. However, we make sure that, when calling $\mathcal{Q}.max()$ at Line 1 of Algorithm 1, the invariant $F_{ab} > L_{ab}/2$ holds for every pair ab in the priority queue. This invariant is maintained by calling $\mathcal{Q}.synchronize(xA)$ (Line 14, Algorithm 1) as soon as we decrease by "too much" F_{xA} (i.e. $F_{xA} \leq L_{xA}/2$). It is easy to see that this policy amortizes operations: every time we call procedure $\mathcal{Q}.synchronize(ab)$, either—Line 15—we are replacing ab with a fresh new dictionary symbol (thus $L_{ab} < 2 \cdot F_{ab}$ work is allowed), or—Line 14—we just decreased F_{ab} by too much ($F_{ab} \leq L_{ab}/2$). In the latter case, we already have done at least $L_{ab}/2$ work during previous replacements (each one has decreased ab's frequency by 1), so $\mathcal{O}(L_{ab})$ additional work does not asymptotically increase running times.

4 Details and Analysis

We first describe how we implement character replacement in the text and how we efficiently sort text positions by pairs. Then, we provide the two queue implementations. For the low-frequency pairs queue, we provide two alternative implementations leading to two different space/time tradeoffs for our main algorithm.

[3]Note that, if CD appears after some substitution $C \rightarrow zw$ which creates occurrences of CD in portions of the text containing zwD, then all occurrences of CD are contained in $TP[P_{zw}, \dots, P_{zw} + L_{zw} - 1]$, and we insert CD in $\mathcal{Q}$ at Line 15 of Algorithm 1 within procedure $\mathcal{Q}.synchronize(zw)$

4.1 Skipping blanks in constant time

As noted above, pair replacements generate runs of the blank character '_'. Our aim in this section is to show how to skip these runs in constant time. Recall that the text is composed by $\lceil \log_2 n \rceil$-bits words. Recall that we reserve *two* blank characters: '*' and '_'. If the run length r satisfies $r < 10$, then we fill all run positions with character '_' (skipping this run takes constant time). Otherwise, ($r \geq 10$) we start and end the run with the string `_*i*_`, where $i = r - 1$, and fill the remaining run positions with '_'. It is not hard to show that this representation allows us to perform the following actions in constant time: skipping a run, accessing a text character, apply a pair substitution and—if needed—merge two runs. For space reasons, we do not discuss full details here.

4.2 Sorting pairs and frequency counting

We now show how to sort the pairs of an array T of n words lexicographically in linear time using only n additional words. Our algorithm only requires read-only access to T. Furthermore, the algorithm generalizes to substrings of any constant length in the same complexity. As an immediate corollary, this implies that we can compute the frequency of each pair in the same complexity simply by traversing the sorted sequence. We need the following results on in-place sorting and merging.

Lemma 1 (Franceschini et al. [3]). *Given an array A of length n with $\mathcal{O}(\log n)$ bit entries, we can in-place sort A in $\mathcal{O}(n)$ time.*

Lemma 2 (Salowe and Steiger [8]). *Given arrays A and B of total length n, we can merge A and B in-place using a comparison-based algorithm in $\mathcal{O}(n)$ time.*

The above results immediately provide simple but inefficient solutions to sorting pairs. In particular, we can copy each pair of T into an array of n entries each storing a pair using 2 words, and then in-place sort the array using Lemma 1. This uses $\mathcal{O}(n)$ time but requires $2n$ words space. Alternatively, we can copy the positions of each pair into an array and then apply a comparison-based in-place sorting algorithm based on 2. This uses $\mathcal{O}(n \log n)$ time but only requires n words of space.

Our algorithm works as follows. Let A be an array of n words. We greedily process T from left-to-right in phases. In each phase we process a contiguous segment $T[i, j]$ of overlapping pairs of T and compute and store the corresponding sorted segment in $A[i, j]$. Phase $i = 0, \ldots, k$ proceeds as follows. Let r_i denote the number of *remaining pairs* in T not yet processed. Initially, we have that $r_0 = n$. Note that r_i is also the number of unused entries in A. We copy the next $r_i/3$ pairs of T into A. Each pair is encoded using the two characters of the pair and the position of the pair in T. Hence, each encoded pair uses 3 words and thus fills all remaining r_i entries in A. We sort the encoded segment using the in-place sort from Lemma 1, where each 3-words encoded pair is viewed as a single key. We then compact the segment back into $r_i/3$ only entries of A by throwing away the characters of each pair and only keeping the position of the pair. We repeat the process until all pairs in T have been processed. At the end A consists of a collection of segments of sorted pairs. We merge the segments from right-to-left using the in-place comparison-based merge from Lemma 2. See the full version [1] for the analysis. We obtain:

Lemma 3. *Given a string T of length n with $\lceil \log_2 n \rceil$-bit characters, we can sort the pairs of T in $\mathcal{O}(n)$ time using n words.*

Lemma 4. *Given a string T of length n with $\lceil \log_2 n \rceil$-bit characters, we can count the frequencies of pairs of T in $\mathcal{O}(n)$ time using n words.*

4.3 High-Frequency Pairs Queue

The capacity of the high-frequency pairs queue is $\sqrt{n}/11$. We implement $\mathcal{Q}$ with the following two components:
(i) **Hash** $\mathcal{H}$. We keep a hash table $\mathcal{H} : \bar{\Sigma}^2 \rightarrow [0, \sqrt{n}/11]$ with $(2/11)\sqrt{n}$ entries. $\mathcal{H}$ will be filled with at most $\sqrt{n}/11$ pairs (hash load ≤ 0.5). Collisions are solved by linear probing. The size of the hash is $(6/11)\sqrt{n}$ words (1 pair and integer per entry)
(ii) **Queue array** B. We keep an array B of quadruples from $\bar{\Sigma}^2 \times [0, n) \times [0, n) \times [0, n)$. B will be filled with at most $\sqrt{n}/11$ entries. We denote with $\langle ab, P_{ab}, L_{ab}, F_{ab} \rangle$ a generic element of B. Every time we pop the highest-frequency pair from the queue, the following holds: (i) ab has frequency F_{ab} in the text, and (ii) ab occurs in a *subset* of text positions $TP[P_{ab}], \ldots, TP[P_{ab} + L_{ab} - 1]$. The size of B is $(5/11)\sqrt{n}$ words. $\mathcal{H}$'s entries point to B's entries: at any stage of the algorithm, if $\mathcal{H}$ contains a pair ab, then $B[\mathcal{H}[ab]] = \langle ab, P_{ab}, L_{ab}, F_{ab} \rangle$. Overall, $\mathcal{Q} = \langle \mathcal{H}, B \rangle$ takes $\sqrt{n}$ words of space.

For space reasons, we do not show how operations on the queue are implemented. All operations run in constant (expected) time, except $\mathcal{Q}.max()$ and $\mathcal{Q}.min()$—which are supported in $\mathcal{O}(\sqrt{n})$ time—and $\mathcal{Q}.synchronize(AB)$—which is supported in $\mathcal{O}(L_{AB} + N \cdot \sqrt{n})$ time, where L_{AB} is AB's interval length at the moment of entering in this procedure, and N is the number of new pairs XY inserted in the queue.

Time complexity To find the most frequent pair in $\mathcal{Q}$ we scan all $\mathcal{Q}$'s elements; since $|\mathcal{Q}| \in \mathcal{O}(\sqrt{n})$ and there are at most $3\sqrt{n}$ high-frequency pairs, the overall time spent inside procedure $max(\mathcal{Q})$ does not exceed $\mathcal{O}(n)$. Since we insert at most $\sqrt{n}/11$ pairs in $\mathcal{Q}$ but there may be up to $3\sqrt{n}$ high-frequency pairs, once $\mathcal{Q}$ is empty we may need to fill it again with new high-frequency pairs. We need to repeat this process at most $(3\sqrt{n})/(\sqrt{n}/11) \in \mathcal{O}(1)$ times, so the number of rounds is constant. We call $\mathcal{Q}.min()$ in two cases: (i) after extracting the maximum from $\mathcal{Q}$ (Line 2, Algorithm 1), and (ii) within procedure $\mathcal{Q}.synchronize$, after discovering a new high-frequency pair XY and inserting it in $\mathcal{Q}$. Case (i) cannot happen more than $3\sqrt{n}$ times. As for case (ii), note that a high-frequency pair can be inserted at most once per round in $\mathcal{Q}$ within procedure $\mathcal{Q}.synchronize$. Since the overall number of rounds is constant and there are at most $3\sqrt{n}$ high-frequency pairs, the time spent inside $\mathcal{Q}.min()$ is $\mathcal{O}(n)$. Finally, considerations of Section 3.4 imply that sorting occurrences lists inside operation $\mathcal{Q}.synchronize$ takes overall linear time thanks to our amortization policy.

4.4 Low-Frequency Pairs Queue

We describe two low-frequency queue variants, denoted in what follows as *fast* and *light*. We start with the fast variant.

Fast queue Let $0 < \epsilon \leq 1$ be a parameter chosen in advance. Our fast queue has maximum capacity $(\epsilon/13) \cdot n$ and is implemented with three components:
(i) **Set of doubly-linked lists** B. This is a set of lists; each list is associated to a distinct frequency. B is implemented as an array of elements of the form $\langle ab, P_{ab}, L_{ab}, F_{ab}, Prev_{ab}, Next_{ab}\rangle$, where $Prev_{ab}$ points to the previous B element with frequency F_{ab} and $Next_{ab}$ points to the next B element with frequency F_{ab} (NULL if this is the first/last such element, resp.). Every B element takes 7 words. We allocate $\epsilon \cdot (7/13) \cdot n$ words for B (maximum capacity: $(\epsilon/13) \cdot n$)
(ii) **Doubly-linked frequency vector** $\mathcal{F}$. This is a word vector $\mathcal{F}[0, \ldots, \sqrt{n}/3 - 1]$ indexing all possible frequencies of low-frequency pairs. We say that $\mathcal{F}[i]$ is *empty* ($\mathcal{F}[i] = NULL$) if i is not the frequency of any pair in T. Non-empty $\mathcal{F}$'s entries are doubly-linked: we associate to each $\mathcal{F}[i]$ two values $\mathcal{F}[i].prev$ and $\mathcal{F}[i].next$ representing the two non-empty pair's frequencies immediately smaller/larger than i. We moreover keep two variables MAX and MIN storing the largest and smallest frequencies in $\mathcal{F}$. If i is the frequency of some character pair, then $\mathcal{F}[i]$ points to the first B element in the chain associated with frequency i
(iii) **Hash** $\mathcal{H}$. We keep a hash table $\mathcal{H} : \Sigma^2 \rightarrow [0, n]$ with $\epsilon \cdot (2/13) \cdot n$ entries. The hash is indexed by character pairs. $\mathcal{H}$ will be filled with at most $\epsilon \cdot n/13$ pairs (hash load ≤ 0.5). Collisions are solved by linear probing. The overall size of the hash is $\epsilon \cdot (6/13) \cdot n$ words: 3 words (one pair and one integer) per hash entry. $\mathcal{H}$'s entries point to B's entries: if ab is in the hash, then $B[\mathcal{H}[ab]] = \langle ab, P_{ab}, L_{ab}, F_{ab}, Prev_{ab}, Next_{ab}\rangle$

For space reasons, we do not show how operations on the queue are implemented. All operations run in constant (expected) time, except $\mathcal{Q}.synchronize(AB)$—which is supported in $\mathcal{O}(L_{AB})$ expected time, L_{AB} being AB's interval length at the moment of entering in this procedure.

Since we insert at most $(\epsilon/13) \cdot n$ pairs in $\mathcal{Q}$ but there may be up to $\mathcal{O}(n)$ low-frequency pairs, once $\mathcal{Q}$ is empty we may need to fill it again with new low-frequency pairs. We need to repeat this process $\mathcal{O}(n/(n \cdot \epsilon/13)) \in \mathcal{O}(1/\epsilon)$ times before all low-frequency pairs have been processed. Since—in our main algorithm—computing array TP, building the queue, and compacting the text take $\mathcal{O}(n)$ time, the overall time spent inside these procedures is $\mathcal{O}(n/\epsilon)$. Using the same reasonings of the previous section, it is easy to show that the time spent inside $\mathcal{Q}.synchronize$ is bounded by $\mathcal{O}(n)$ thanks to our amortization policy. Since all queue operations except $\mathcal{Q}.synchronize$ take constant time, we spend overall $\mathcal{O}(n)$ time operating on the queue. These considerations imply that the high-frequency pair processing phase of our main algorithm takes overall $\mathcal{O}(n/\epsilon)$ randomized time. Theorem 1(i) follows.

Light queue We observe that we can re-use the space of *blank text characters* generated after replacements to store B and $\mathcal{H}$. Let S_i be the capacity (in terms of number of pairs) of the queue at the i-th time we re-build it in our main algorithm; at the beginning, $S_1 = 1$. After replacing all $\mathcal{Q}$'s pairs and compacting text positions, new blanks are generated and this space is available at the end of the memory allocated for the text, so we can accumulate it on top of S_i obtaining space $S_{i+1} \geq S_i$. At the next execution of the `while` loop, we fill the queue until all the available space S_{i+1} is filled. We proceed like this until all pairs have been processed.

Replacing a pair ab generates at least $F_{ab}/2$ blanks: in the worst case, the pair is of the form aa and all pair occurrences overlap, e.g. in $aaaaaa$ (which generates 3 blanks). Moreover, replacing a pair with frequency F_{ab} decreases the frequency of at most $2F_{ab}$ pairs in the active priority queue (these pairs can therefore disappear from the queue). Note that $F_{ab} \geq 2$ (otherwise we do not consider ab for substitution). After one pair ab is replaced at round i, the number M_i of elements in the active priority queue is at least $M_i \geq S_i - (1 + 2F_{ab})$. Letting $f_1, f_2, \ldots$ be the frequencies of all pairs in the queue, we get that after replacing all elements the number (0) of elements in the priority queue is: $0 \geq S_i - (1 + 2f_1) - (1 + 2f_2) - \cdots$ which yields $S_i \leq (1 + 2f_1) + (1 + 2f_2) + \cdots \leq 2.5f_1 + 2.5f_2 + \cdots = 2.5\sum_i f_i$, since $f_i/2 \geq 1$ for all i. So when the active priority queue is empty we have at least $\sum_i f_i/2 \geq S_i/5$ new blanks. Recall that a pair takes 13 words to be stored in our queue. In the next round we therefore have room for a total of $(1 + 1/(5 \cdot 13))S_i = (1 + 1/65)S_i$ new pairs. This implies $S_i = (1 + 1/65)^{i-1}$. Since $S_i \leq n$ for any i, we easily get that the number R of rounds is bounded by $R \in \mathcal{O}(\log n)$. With the same reasonings used before to analyze the overall time complexity of our algorithm, we get Theorem 1(ii).

5 References

[1] P. Bille, I. L. Gørtz, and N. Prezza. Space-Efficient Re-Pair Compression. *arXiv preprint arXiv:1611.01479*, 2016.

[2] F. Claude and G. Navarro. Fast and Compact Web Graph Representations. *ACM Trans. Web*, 4(4):16:1–16:31, 2010.

[3] G. Franceschini, S. Muthukrishnan, and M. Patrascu. Radix Sorting with No Extra Space. In *Proc. 15th ESA*, pages 194–205, 2007.

[4] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th CPM*, pages 216–227.

[5] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[6] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150 – 1167, 2013.

[7] G. Navarro and L. Russo. Re-pair Achieves High-Order Entropy. In *Proc. 18th DCC*, page 537, 2008.

[8] J. S. Salowe and W. L. Steiger. Simplified Stable Merging Tasks. *J. Algorithms*, 8(4):557–571, 1987.

[9] Y. Tabei, H. Saigo, Y. Yamanishi, and S. J. Puglisi. Scalable Partial Least Squares Regression on Grammar-Compressed Data Matrices. In *Proc. 22Nd KDD*, pages 1875–1884, 2016.

[10] R. Wan. *Browsing and Searching Compressed Documents.* PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne., 1999.