Compression with the tudocomp Framework

Patrick Dinklage[1], Johannes Fischer[1], Dominik Köppl[1], Marvin Löbel[1], and Kunihiko Sadakane[2]

1 **Department of Computer Science, TU Dortmund, Germany**
pdinklag@gmail.com, johannes.fischer@cs.tu-dortmund.de,
dominik.koeppl@tu-dortmund.de, loebel.marvin@gmail.com

2 **Grad. School of Inf. Science and Technology, University of Tokyo, Japan**
sada@mist.i.u-tokyo.ac.jp

Abstract

We present a framework facilitating the implementation and comparison of text compression algorithms. We evaluate its features by a case study on two novel compression algorithms based on the Lempel-Ziv compression schemes that perform well on highly repetitive texts.

1998 ACM Subject Classification D.3.3 Frameworks, D.2.2 Software libraries

Keywords and phrases lossless compression, compression framework, compression library, algorithm engineering, application of string algorithms

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Engineering novel compression algorithms is a relevant topic, shown by recent approaches like bzip [6], Brotli [1], or Zstandard[1]. Engineers of data compression algorithms face the fact that it is cumbersome (a) to build a new compression program from scratch, and (b) to evaluate and benchmark a compression algorithm against other algorithms objectively. We present the highly modular compression framework tudocomp that addresses both problems. To tackle problem (a), tudocomp contains standard techniques like VByte, Elias-γ/δ, or Huffman coding. To tackle problem (b), it provides automatic testing and benchmarking against external programs and implemented standard compressors like Lempel-Ziv compressors. As a case study, we present the two novel compression algorithms lcpcomp and LZ78U, their implementations in tudocomp, and their evaluations with tudocomp. lcpcomp is based on Lempel-Ziv 77, substituting greedily the longest remaining repeated substring. LZ78U is based on Lempel-Ziv 78, with the main difference that it allows a factor to introduce multiple new characters.

1.1 Related Work

There are many[2] compression benchmark websites measuring compression programs on a given test corpus. Although the compression ratio of a novel compression program can be compared with the ratios of the programs listed on these websites, we cannot infer which program runs faster or more memory efficiently if these programs have not been compiled and run on the same machine. Efforts in facilitating this kind of comparison have been made by wrapping the source code of different compression algorithms in a single executable that benchmarks the algorithms on the same machine with the same compile flags. Examples include lzbench[3] and Squash[4].

[1] https://github.com/facebook/zstd
[2] e.g., http://www.squeezechart.com or http://www.maximumcompression.com
[3] https://github.com/inikep/lzbench
[4] https://quixdb.github.io/squash-benchmark

Considering frameworks aiming at easing the comparison *and* implementation of new compression algorithms, we are only aware of the C++98 library ExCom [12]. The library contains a collection of compression algorithms. These algorithms can be used as components for a compression pipeline. However, ExCom does not provide the same flexibility as we had in mind; it provides only character-wise pipelines, i.e., it does no bitwise transmission of data. Its design does not use meta-programming features; a header-only library has more potential for optimization since the compiler can inline header-implemented (possibly performance critical) functions easily.

A broader focus is set in Giuseppe Ottaviano's succinct library [11] and Simon Gog's Succinct Data Structure Library 2.0 (SDSL) [10]. These two libraries provide integer coders and helper functions for working on the bit level.

1.2 Our Results/Approach

Our lossless compression framework ***tudocomp*** aims at supporting and facilitating the implementation of novel compression algorithms. The philosophy behind tudocomp is to support building a pipeline of modules that transforms an input to a compressed binary output. This pipeline has to be flexible: appending, exchanging and removing a module in the pipeline in a plug-and-play manner is in the main focus of the design of tudocomp. Even a module itself can be refined into submodules.

To this end, tudocomp is written in modern C++11. On the one hand, the language allows us to write compile time optimized code due to its meta programming paradigm. On the other hand, its fine-grained memory management mechanisms support controlling and monitoring the memory footprint in detail. We provide a tutorial, an exhaustive documentation of the API, and the source code at `http://tudocomp.org` with the permissive Apache License 2.0 to encourage developers to use and foster the framework.

In order to demonstrate its usefulness, we added reference implementations of common compression and encoding schemes (see Section 2). On top of that, we present two novel algorithms (see Section 3) which we have implemented in our framework. We give a detailed evaluation of these algorithms in Section 4, thereby exposing the benchmarking and the visualization tools of tudocomp.

2 Description of the tudocomp Framework

On the topmost abstraction level, tudocomp defines the abstract types `Compressor` and `Coder`. A ***compressor*** transforms an input into an output so that the input can be losslessly restored from the output by the corresponding ***decompressor***. A ***coder*** takes an elementary data type like a character and writes it to a compressed bit sequence. As with compressors, each coder is accompanied by a ***decoder*** taking care of restoring the original data from its compressed bit sequence. By design, a coder can take the role of a compressor, but a compressor may not be suitable as a coder (e.g., a compressor that needs random access on the whole input).

tudocomp provides implementations of the compressors and the coders shown in the tables below. Each compressor and coder gets an identifier (right column of each table).

Compressors	
BWT	`bwt`
Coder wrapper	`encode`
LCPComp (Section 3.2)	`lcpcomp`
LZ77 (Def. 1), LZSS [25] output	`lzss_lcp`
LZ78 (Def. 2)	`lz78`
LZ78U (Section 3.3)	`lz78u`
LZW [26]	`lzw`
Move-To-Front	`mtf`
Re-Pair [18]	`repair`
Run-Length-Encoding	`rle`

Integer Coders	
Bit-Compact Coder	`bit`
Elias-γ [5]	`gamma`
Elias-δ [5]	`delta`

String Coders	
Canonical Huffman Coder [27]	`huff`
A Custom Static Low Entropy Encoder (Section 3.2)	`sle`

The behavior of a compressor or coder can be modified by passing different parameters. A parameter can be an elementary data type like an integer, but it can also be an instance of a class that specifies certain subtasks like integer coding. For instance, the compressor `lzss_lcp(threshold, coder)` takes an integer `threshold` and a `coder` (to code an LZ77 factor) as parameters. The coder is supplied as a parameter such that the compressor can call the coder directly (instead of alternatively piping the output of `lzss_lcp` to a coder).

The support of class parameters eases the deployment of the design pattern ***strategy*** [9]. A strategy determines what algorithm or data structure is used to achieve a compressor-specific task.

Library and Command Line Tool. tudocomp consists of two major components: a standalone compression library and a command line tool `tdc`. The library contains the core interfaces and implementations of the aforementioned compressors and coders. The tool `tdc` exposes the library's functionality in form of an executable that can run compressors directly on the command line. It allows the user to select a compressor by its identifier and to pass parameters to it, i.e., the user can specify the exact compression strategy *at runtime*. For instance, the LZ78U compressor (Section 3.3) expects a compression strategy, an integer coder, and an integer variable specifying a threshold. Its strategy can define parameters by itself, like which string coder to use. A valid call is `./tdc -a 'lz78u(coder = bit, comp = buffering(string_coder = huff), threshold = 3)' input.txt -o output.tdc`, where `tdc` compresses the file `input.txt` and stores the compressed bit sequence in the file `output.tdc`. To this end, it uses the compressor `lz78u` parametrized by the coder `bit` for integer values, by the compression strategy `buffering` with `huff` to code strings, and by a threshold value of 3.

After compressing an input using a certain compression strategy, the tool adds a header to the compressed file so that it can decompress it without the need for specifying the compression strategy again. However, this behavior can be overruled by explicitly specifying a decompression strategy, e.g., to test different decompression strategies.

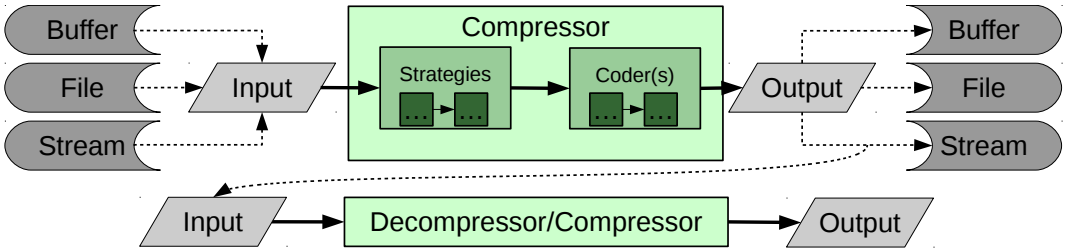


Figure 1 Flowchart of a possible compression pipeline. The compressors of tudocomp work with abstract data types for input and output, i.e., a compressor is unaware of whether the input or the output is a file, is stored in memory, or is accessed using a stream. A compressor can follow one or more compression strategies that can have (nested) parameters. Usually, a compressor is parametrized with one or more coders (e.g., for different integer ranges or strings) that produce the final output.

Helper classes. tudocomp provides several classes for easing common tasks when engineering a new compression algorithm, like SA, ISA or LCP. tudocomp generates SA with `divsufsort`[5], and LCP with the Φ-algorithm [15]. The arrays SA, ISA, and LCP can be stored in plain arrays or in packed arrays with a bit width of $\lceil \lg n \rceil$, i.e., in a ***bit-compact representation***. We provide the modes `plain`, `compressed`, and `delayed` to describe when/whether a data structure should be stored in a bit-compact representation: In `plain` mode, all data structures are stored in plain arrays; in `compressed` mode, all data structures are built in a bit-compact representation. In `delayed` mode, tudocomp first builds a data structure A in a plain array; when all other data structures are built whose constructions depended on A, A gets transformed into a bit-compact representation.

[5] `https://github.com/y-256/libdivsufsort`

While `direct` and `compressed` are the fastest or the memory-friendliest modes, respectively, the data structures produced by `delayed` are the same as `compressed`, though `delayed` is faster than `compressed`.

If more elaborated algorithms are desired (e.g., for producing compressed data structures like the compressed suffix array), it is easy to use tudocomp in conjunction with SDSL for which we provide an easy binding.

Combining streaming and offline approaches. A compressor can stream its input (online approach) or request the input to be loaded into memory (offline approach). Compressors can be chained to build a pipeline of multiple compression modules, like as in Figure 1.

2.1 Example

(a) C++ Source Code

```
1  #include <tudocomp/tudocomp.hpp>
2  class BWTComp : public Compressor {
3    public: static Meta meta() {
4      Meta m("compressor", "bwt");
5      m.option("ds").templated<TextDS<>>();
6      m.needs_sentinel_terminator();
7      return m; }
8    using Compressor::Compressor;
9    void compress(Input& in, Output& out) {
10     auto o = out.as_stream();
11     auto i = in.as_view();
12     TextDS<> t(env().env_for_option("ds"),i);
13     const auto& sa = t.require_sa();
14     for(size_t j = 0; j < t.size(); ++j)
15       o << ((sa[j] != 0) ? t[sa[j] - 1]
16                          : t[t.size() - 1]);
17   }
18   void decompress(Input&, Output&){/*[...]*/}
19 };
```

(b) Execution with `tdc`

```
1  > echo -n 'aaababaaabaababa' > ex.txt
2  > ./tdc -a bwt -o bwt.tdc ex.txt
3  > hexdump -v -e '"%-2_c"' bwt.tdc
4  b w t % a b b \0a b a b b a a a a a a a a
5  > ./tdc -a 'bwt:rle' -o rle.tdc ex.txt
6  > hexdump -v -e '"%-3_c"' rle.tdc
7  b  w  t  :  r  l  e  %  a  b  b  \0 \0 a  b
      a  b  b  \0 a  a  006
```

The source code (a) on the left implements a compressor that computes the Burrows-Wheeler transform (BWT) (see Section 3.1) of an input. To this end, it loads the input into memory using (line 11) `in.as_view()` and computes the suffix array using (line 13) `t.require_sa()`. In the function `meta`, we state that we assume the unique terminal symbol (represented by the byte '\0') as part of the text, and that we want to register the class `BWTComp` as a `Compressor` with the identifier `bwt`. By doing so, we can call the compressor directly in the command line tool `tdc` using the argument `-a bwt`. In the shell code (b) on the left, you can see how we produced the BWT of our running example. The program `hexdump` outputs each character of a file such that non-visible characters are escaped. A `%`-sign separates the header from the body in the output. Next, we use the binary composition operator `:` connecting the output of its left operand with the input of its right operand. In the shell code, this operator pipes the output of `bwt` to the run-length encoding compressor `rle`, which transforms a substring $\texttt{aa}\underbrace{\texttt{a}\cdots\texttt{a}}_{m \text{ times}}$ to $\texttt{aa}m$ with $m \geq 0$ encoded in VByte (the output is a *byte* sequence).

(c) Assembling a compression pipeline

```
1  > ./tdc -a bwt -o bwt.tdc pc_english.200MB
2  > ./tdc -a 'bwt:rle:mtf:encode(huff)' -o bzip
       .tdc pc_english.200MB
3  > stat -c"%s␣%n" pc_english.200MB *.tdc
4  209715200 pc_english.200MB
5  209715209 bwt.tdc
6  66912437 bzip.tdc
```

Finally, the compressor `bwt` can be used as part of a pipeline to achieve good compression quality: Given a move-to-front compressor `mtf` and a Huffman coder `huff`, we can build a chain `bwt:rle:mtf:encode(huff)`. The compressor `encode` is a wrapper that turns a coder into a compressor. The last code fragment (c) on the left shows the calls of this pipeline and a call of `bwt` only. Using `stat`, we measure the file sizes (in bytes) of the input PC-ENGLISH (see Section 4) and both outputs.

2.2 Specific Features

Build Requirements. To deploy tudocomp, the build management software `cmake`, the version control system `git`, Python 3, and a C++11 compiler are required. `cmake` automatically downloads and builds other third-party software components like the SDSL. We tested the build process on Unix-like build environments, namely Debian Jessie, Ubuntu Xenial, Arch Linux 2016, and the Ubuntu shell on Windows 10.

Unit Tests. tudocomp offers semi-automatic unit tests. For a registered compressor, tudocomp can automatically generate test cases that check whether the compressor can compress and decompress a set of selected inputs successfully. These inputs include border cases like the empty string, a run of the same character, samples on various subranges in UTF-8, Fibonacci strings, Thue-Morse strings, and strings with a high number of runs [20]. These strings can be generated on-the-fly by `tdc` as an alternative input.

Type Inferences. The C++ standard does neither provide a syntax for constraining type parameters (like type wildcards in Java) nor for querying properties of a class at runtime (i.e., reflection). To address this syntactic lack we augmented each class exposed to `tdc` and to the unit tests with a so-called ***type***. A type is a string identifier. We expect that classes with the same type provide the same public methods. Types resemble *interfaces* of Java, but contrary to those, they are not subject to polymorphism. Common types in our framework are `Compressor` and `Coder`. The idea is that, given a compressor that accepts a `Coder` as a parameter, it should accept all classes of type `Coder`. To this end, each typed class is augmented with an identifier and a description of all parameters that the class accepts. All typed classes are exposed by the tool `tdc` that calls a typed class by its identifier with the described parameters. Types provide a uniform, but simple declaration of all parameters (e.g., integer values, or strategy classes). The aforementioned exemplaric call of `lz78u` at the beginning of Section 2 illustrates the uniform declaration of the parameters of a compressor.

Evaluation tools. To evaluate a compressor pipeline, tudocomp provides several tools to facilitate measuring the compression ratio, the running time, and the memory consumption. By adding `--stats` to the parameters of `tdc`, the tool monitors these measurement parameters: It additionally tracks the running time and the memory consumption of the data structures in all phases. A phase is a self-defined code division like a pre-processing phase, or an encoding phase. Each phase can collect self-defined statistics like the number of generated factors. All measured data is collected in a JSON file that can be visualized by the web application found at `http://tudocomp.org/charter`. An example is given in Figure 7.

In addition, we have a command line ***comparison tool*** called `compare.py` that runs a predefined set of compression programs (that can be tudocomp compressors or external compression programs). Its primary usage is to compare tudocomp compression algorithms with external compression programs. It monitors the memory usage with the tool `valgrind -tool=massif -pages-as-heap=yes`. This tool is significantly slower than running `tdc` with `--stats`.

3 New Compression Algorithms

With the aid of tudocomp, it is easy to implement new compression algorithms. We demonstrate this by introducing two novel compression algorithms: ***lcpcomp*** and ***LZ78U***. To this end, we first recall some definitions.

3.1 Theoretical Background

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| \leq n^{\mathcal{O}(1)}$ for a natural number n. We call an element $T \in \Sigma^*$ a ***string***. The empty string is ϵ with $|\epsilon| = 0$. Given $x, y, z \in \Sigma^*$ with $T = xyz$, then x, y and z are called a ***prefix***, ***substring*** and ***suffix*** of w, respectively. We call $T[i..]$ the i-th suffix of T, and denote a substring $T[i] \cdots T[j]$ with $T[i..j]$.

For the rest of the article, we take a string T of length n. We assume that $T[n]$ is a special character $\$ \notin \Sigma$ so that no suffix of T is a prefix of another suffix of T.

SA and ISA denote the suffix array [19] and the inverse suffix array of T, respectively. $\mathsf{LCP}[2..n]$ is an array such that $\mathsf{LCP}[i]$ is the length of the longest common prefix of the *lexicographically* i-th smallest suffix with its lexicographic predecessor for $i = 2, \ldots, n$. The BWT [3] of T is the string BWT with

$$\mathsf{BWT}[j] = \begin{cases} T[n] & \text{if } \mathsf{SA}[j] = 1, \\ T[\mathsf{SA}[j] - 1] & \text{otherwise,} \end{cases}$$

for $1 \leq j \leq n$. The arrays SA, ISA, LCP and BWT can be constructed in time linear to the number of characters of T [16].

As a running example, we take the text $T :=$ `aaababaaabaababa`\$. The arrays SA, LCP and BWT of this example text are shown in Figure 2.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	$
SA[i]	17	16	7	1	8	11	2	14	5	9	12	3	15	6	10	13	4
LCP[i]	-	0	1	5	2	4	6	1	3	4	3	5	0	2	3	2	4
BWT[i]	a	b	b	$	a	b	a	b	b	a	a	a	a	a	a	a	a

Figure 2 Suffix array, LCP array and BWT of the running example.

Given a bit vector B with length $|B|$, the operation $B.\mathrm{rank}_1(i)$ counts the number of '1'-bits in $B[1..i]$, and the operation $B.\mathrm{select}_1(i)$ yields the position of the i-th '1' in B.

There are data structures [13, 4] that can answer rank and select queries on B in constant time, respectively. Each of them uses $o(|B|)$ additional bits of space, and both can be built in $\mathcal{O}(|B|)$ time.

The ***suffix trie*** of T is the trie of all suffixes of T. The ***suffix tree*** of T, denoted by ST, is the tree obtained by compacting the suffix trie of T. It has n leaves and at most n internal nodes. The string stored in an edge e is called the ***edge label*** of e, and denoted by $\lambda(e)$. The ***string depth*** of a node v is the length of the concatenation of all edge labels on the path from the root to v. The leaf corresponding to the i-th suffix is labeled with i.

Each node of the suffix tree is uniquely identified by its pre-order number. We can store the suffix tree topology in a bit vector (e.g., DFUDS [2] or BP [13, 23]) such that rank and select queries enable us to address a node by its pre-order number in constant time. If the context is clear, we implicitly convert an ST node to its pre-order number, and vice versa. We will use the following constant time operations on the suffix tree:

- parent(v) selects the parent of the node v,
- level-anc(ℓ, d) selects the ancestor of the leaf ℓ at depth d (level ancestor query), and
- leaf-select(i) selects the i-th leaf (in lexicographic order).

A ***factorization*** of T of size z partitions T into z substrings $T = F_1 \cdots F_z$. These substrings are called ***factors***. In particular, we have:

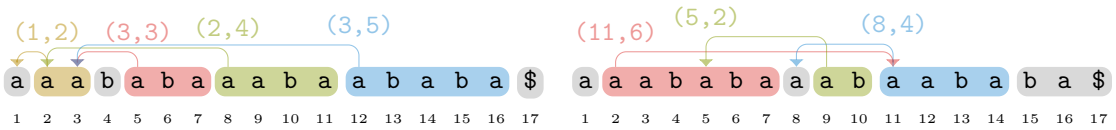
▶ **Definition 1.** A factorization $F_1 \cdots F_z = T$ is called the ***Lempel-Ziv-77 (LZ77) factorization*** [28] of T with a threshold $\vartheta \geq 1$ iff F_x is either the longest substring of length at least ϑ occurring at least twice in $F_1 \cdots F_x$, or, if such a substring does not exist, a single character. We merge successive occurrences of the latter type of factors to a single factor and call it a ***remaining substring***.

The usual definition of the LZ77 factorization fixes $\vartheta = 1$. We introduced the version with a threshold to make the comparison with lcpcomp (Section 3.2) fairer.

▶ **Definition 2.** A factorization $F_1 \cdots F_z = T$ is called the ***Lempel-Ziv-78 (LZ78) factorization*** [29] of T iff $F_x = F_y \cdot c$ with $F_y = \mathrm{argmax}_{S \in \{F_y : y < x\} \cup \{\epsilon\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$.

3.2 lcpcomp

The idea of lcpcomp is to search for long repeated substrings and substitute one of their occurrences with a reference to the other. High values in the LCP-array indicate such long repeated substrings. There are two major differences to the LZ77 compression scheme: (1) while LZ77 only allows back-references, lcpcomp allows both back *and forward* references; and (2) LZ77 factorizes T greedily from left to right, whereas lcpcomp makes substitutions at *arbitrary* positions in the text, greedily chosen such that the number of substituted characters is maximized. This process is repeated until all remaining repeated substrings are shorter than a threshold ϑ. On termination, lcpcomp



(a) LZ77 **(b)** lcpcomp

Figure 3 References of the (a) LZ77 factorization with the threshold $\vartheta = 2$, and of the (b) lcpcomp factorization with the same threshold. The output of the LZ77 and the lcpcomp algorithms are `a(1,2)b(3,3)(2,4)(3,5)$` and `a(11,6)a(5,2)(8,4)ba$`, respectively.

has generated a factorization $T = F_1 \cdots F_z$, where each F_j is either a remaining substring, or a reference (i, ℓ) with the intended meaning "copy ℓ characters from position i" (see Figure 3b for an example).

Algorithm. The LCP array stores the longest common prefix of two lexicographically neighbored suffixes. The largest entries in the LCP array correspond to the longest substrings of the text that have at least two occurrences. Given a suffix $T[\mathsf{SA}[i]..]$ whose entry $\mathsf{LCP}[i]$ is maximal among all other values in LCP, we know that $T[\mathsf{SA}[i]..\mathsf{SA}[i]+\mathsf{LCP}[i]-1] = T[\mathsf{SA}[i-1]..\mathsf{SA}[i-1]+\mathsf{LCP}[i]-1]$, i.e., we can substitute $T[\mathsf{SA}[i]..\mathsf{SA}[i]+\mathsf{LCP}[i]-1]$ with the reference $(\mathsf{SA}[i-1], \mathsf{LCP}[i])$. In order to find a suffix whose LCP entry is maximal, we need a data structure that maintains suffixes ordered by their corresponding LCP values. We use a maximum heap for this task. To this end, the heap stores suffix array indices whose keys are their LCP values (i.e., insert i with key $\mathsf{LCP}[i]$, $2 \le i \le n$). The heap stores only those indices whose keys are at least ϑ.

While the heap is not empty, we do the following:

1. Remove the maximum from the heap; let i be its value.
2. Report the reference $(\mathsf{SA}[i-1], \mathsf{LCP}[i])$ and the position $\mathsf{SA}[i]$ as a triplet $(\mathsf{SA}[i-1], \mathsf{LCP}[i], \mathsf{SA}[i])$.
3. For every $1 \le k \le \mathsf{LCP}[i]-1$, remove the entry $\mathsf{ISA}[\mathsf{SA}[i]+k]$ from the heap (as these positions are covered by the reported reference).
4. Decrease the keys of all entries j with $\mathsf{SA}[i] - \mathsf{LCP}[i] \le \mathsf{SA}[j] < \mathsf{SA}[i]$ to $\min(\mathsf{LCP}[j], \mathsf{SA}[i] - \mathsf{SA}[j] - 1)$. (If a key becomes smaller than ϑ, remove the element from the heap.) By doing so, we prevent the substitution of a substring of $T[\mathsf{SA}[i]..\mathsf{SA}[i]+\mathsf{LCP}[i]-1]$ at a later time.

```
template<class text_t>
class MaxHeapStrategy : public Algorithm {
 public: static Meta meta() {
  Meta m("lcpcomp_strategy", "heap");
  return m; }
 using Algorithm::Algorithm;
 void create_factor(int pos,int src,int len);
 void factorize(text_t& text, const int t) {
  text.require(text_t::SA | text_t::ISA |
       text_t::LCP);
  auto& sa = text.require_sa();
  auto& isa = text.require_isa();
  auto lcpp = text.release_lcp()->relinquish
       ();
  auto& lcp = *lcpp;
  ArrayMaxHeap<typename text_t::lcp_type::
       data_type> heap(lcp, lcp.size(), lcp.
       size());
  for(int i = 1; i < lcp.size(); ++i)
   if(lcp[i] >= t) heap.insert(i);
  while(heap.size() > 0) {
   int m = heap.top(), fpos = sa[m],
       fsrc = sa[m-1], flen = heap.key(m);
   create_factor(fpos, fsrc, flen);
   for(int k=0; k < flen; k++)
    heap.remove(isa[fpos + k]);
   for(int k=0; k < flen && fpos > k; k++) {
    int s = fpos - k - 1;
    int i = isa[s];
    if(heap.contains(i)) {
     if(s + lcp[i] > fpos) {
      int l = fpos - s;
      if(l >= t)
       heap.decrease_key(i, l);
      else heap.remove(i);
}}}}}};
```

As an invariant, the key ℓ of a suffix array index i stored in the heap will always be the maximal number of characters such that $T[i..i+\ell-1]$ occurs at least twice in the *remaining* text.

The reported triplets are collected in a list. To compute the final output, we sort the triplets by their third component (storing the starting position of the substring substituted by the reference stored in the first two components). We then scan simultaneously over the list and the text to generate the output.

The code on the left implements the compression strategy of lcpcomp that uses a maximum heap. We transfered the code from the compressor class to a strategy class since the lcpcomp compression scheme can be implemented in different ways. Each strategy receives a text. Its goal is to compute all factors (created by the `create_factor` method). In the depicted strategy, we use a maximum

heap to find all factors. The heap is implemented in the class `ArrayMaxHeap`. An instance of that class stores an array A of keys and an array heap maintaining (key-value)-pairs of the form $(A[i], i)$ with the order $(A[i], i) < (A[j], j) :\Leftrightarrow A[i] < A[j]$. To access a specific element in the heap by its value, the class has an additional array storing the position of each value in the heap.

Correctness. Although a reference r can refer to a substring that has been substituted by another reference after the creation of r, it is still possible to restore the text due to the following lemma:

▶ **Lemma 3.** *The output of lcpcomp contains enough information to restore the original text.*

Proof. We want to show that the output is free of cycles, i.e., there is no text position i for that $i \underbrace{\rightarrow \cdots \rightarrow}_{\text{cycle length}} i$ holds, where $\rightarrow$ is a relation on text positions such that $i \rightarrow j$ holds iff there is a substring $T[i'..i'+\ell-1]$ with $i \in [i', i'+\ell-1]$ that has been substituted by a reference $(j-i+i', \ell)$. If the text is free of cycles, then each substituted text position can be restored by following a finite chain of references.

First, we show that is not possible to create cycles of length two. Assume that we substituted $T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i]+\ell_i-1]$ with $(\mathsf{SA}[i-1], \ell_i)$ for $\vartheta \le \ell_i \le \mathsf{LCP}[i]$. The algorithm will not choose $T[\mathsf{SA}[i-1]+k..\,\mathsf{SA}[i-1]+k+\ell_k-1]$ for $0 \le k \le \ell_i$ and $\vartheta \le \ell_k \le \mathsf{LCP}[i]-k$ to be substituted with $(\mathsf{SA}[i]+k, \ell_k)$, since $T[\mathsf{SA}[i]+k..] > T[\mathsf{SA}[i-1]+k..]$ and therefore $\mathsf{ISA}[\mathsf{SA}[i]+k] > \mathsf{ISA}[\mathsf{SA}[i-1]+k]$. Finally, by the transitivity of the lexicographic order (i.e., the order induced by the suffix array), it is neither possible to produce larger cycles. ◀

Time Analysis. We insert at most n values into the heap. No value is inserted again. Finally, we use the following lemma to get a running time of $\mathcal{O}(n \lg n)$:

▶ **Lemma 4.** *The key of a suffix array entry is decreased at most once.*

Proof. Let us denote the key of a value i stored in the heap by $K[i]$. Assume that we have decreased the key $K[j]$ of some value j stored in the heap after we have substituted a substring $T[i..i+\ell-1]$ with a reference. It holds that $K[j] = \mathsf{SA}[i] - \mathsf{SA}[j] - 1 > \mathsf{SA}[i] - \mathsf{SA}[j] - 1 - m \ge K[\mathsf{ISA}[\mathsf{SA}[j]+m]]$ for all m with $1 \le m \le K[j]$, i.e., there is no suffix array entry that can decrease the key of j again. ◀

3.2.1 Decompression

Decompressing lcpcomp-compressed data is harder than decompressing LZ77, since references in lcpcomp can refer to positions that have not yet been decoded. Figure 3 depicts the references built on our running example by arrows.

In order to cope with this problem, we add, for each position i of the original text, a list L_i storing the text positions waiting for this text position getting decompressed.

First, we determine the original text size (the compressor stores it as a VByte before the output of the factorization). Subsequently, while there is some compressed input, we do the following, using a counting variable i as a cursor in the text that we are going to rebuild:

- If the input is a character c, we write $T[i] \leftarrow c$, and increment i by one.
- If the input is a reference consisting of a position s and a length ℓ, we check whether $T[s+j]$ is already decoded, for each j with $0 \le j \le \ell - 1$:
  - If it is, then we can restore $T[i+j] \leftarrow T[s+j]$.
  - Otherwise, we add $i+j$ to the list L_{s+j}.

  In either case, we increment i by ℓ.

An additional procedure is needed to restore the text completely by processing the lists: On writing $T[i] \leftarrow c$ for some text position i and some character c, we further write $T[t] \leftarrow T[i]$ for each t stored in L_i (if L_t is not empty, we proceed recursively). Afterwards, we can delete L_i since it will be no longer needed. The decompression runs in $\mathcal{O}(n)$ time, since we perform a linear scan over the decompressed text, and each text position is visited at most twice.

3.2.2 Related Work

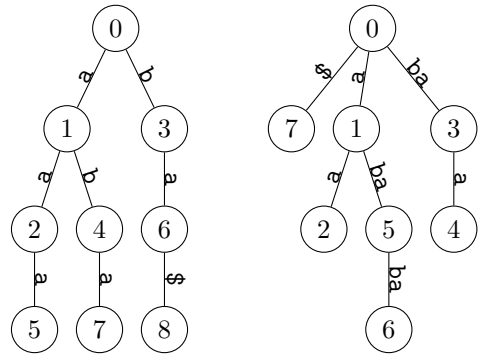
Rather than adapting the LZ77 compression scheme, [21] and [22] compute a grammar by subsequently substituting a longest substring in the text occurring at least twice without overlapping. After computing such a substring, they exchange each occurrence of it with a new non-terminal. Compressing the text with a grammar has two main characteristics: On the one hand, it substitutes *every* non-overlapping occurrence of a substring with the *same* new non-terminal. On the other hand, it enforces that every substring that is going to be substituted has at least two occurrences *without* overlapping.

3.2.3 Implementation Improvements

Compression. We will present an alternative strategy to the heap for the lcpcomp compression, since the heap takes $\mathcal{O}(n \lg n)$ time for deleting elements (it stores all suffix array entries at the beginning). In this strategy, we compute an array A_ℓ storing all suffix array entries j with $\mathsf{LCP}[j] = \ell$, for each ℓ with $\vartheta \leq \ell \leq \max_k \mathsf{LCP}[k]$. To compute the references, we sequentially scan the arrays in decreasing order, starting with the array that stores the suffixes with the maximum LCP value. On substituting a substring $T[\mathsf{SA}[i] \,..\, \mathsf{SA}[i] + \mathsf{LCP}[i] - 1]$ with the reference $(\mathsf{SA}[i-1], \mathsf{LCP}[i])$, we update the LCP array (instead of updating the keys in the heap). We set $\mathsf{LCP}[\mathsf{ISA}[\mathsf{SA}[i] + k]] \leftarrow 0$ for every $1 \leq k \leq \mathsf{LCP}[i] - 1$ (deletion), and $\mathsf{LCP}[j] \leftarrow \min(\mathsf{LCP}[j], \mathsf{SA}[i] - \mathsf{SA}[j] - 1)$ for every j with $\mathsf{ISA}[\mathsf{SA}[i] - \mathsf{LCP}[i]] \leq j < i$ (decrease key). Unlike the heap implementation, we do *not* delete an entry from the arrays. Instead, we look up the current LCP value of an element when we process it: Assume that we want to process $A_\ell[i]$. If $\mathsf{LCP}[A_\ell[i]] = \ell$, then we proceed as above. Otherwise, we have updated the LCP value of the suffix starting at position $A_\ell[i]$ to the value $\ell' := \mathsf{LCP}[A_\ell[i]] < \ell$. In this case, we append $A_\ell[i]$ to $A_{\ell'}$ (if $\ell' < \vartheta$, we do nothing), and skip computing the reference for $A_\ell[i]$. By doing so, we either omit the substring $A_\ell[i]$ if $\ell' < \vartheta$, or delay the processing of the value $A_\ell[i]$. A suffix array entry gets delayed at most once, analogously to Lemma 4.

Decompression. We use a heuristic to improve the memory usage. The heuristic defers the creation of the lists L_i storing the text positions that are waiting for the position i to get decompressed. If a reference needs a substring that has not yet been decompressed, we store the reference in a list L. By doing so, we have reconstructed at least all substrings that have not been substituted by a reference during the compression. Subsequently, we try to decompress each reference stored in L, removing successfully decompressed references from L. If we repeat this step, more and more text positions can become restored. Clearly, after at most n iterations, we would have restored the original text completely, but this would cost us $\mathcal{O}(n^2)$ time. Instead, we run this algorithm only for a fixed number of times α. Afterwards, we mark all not yet decompressed positions in a bit vector B, and build a rank data structure on top of B. Next, we create a list L_i for each marked text position $B.\mathrm{rank}(i)$ as in the original algorithm. The difference to the original algorithm is that L_i now corresponds to $B.\mathrm{rank}(i)$. Finally, we run the original algorithm using the lists L_i to restore the remaining characters.

Encoding. After computing the lcpcomp factorization, we encode the remaining substrings of its output by a static low entropy encoder `sle`. The coder is similar to a Huffman coder, but it additionally treats all 3-grams of the remaining substrings as symbols of the input.



(a) LZ78-Trie **(b)** LZ78U-Tree

Figure 6 Dictionary trees of LZ78 and LZ78U. LZ78 factorizes our running example into $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{b}}|\overset{4}{\texttt{ab}}|\overset{5}{\texttt{aaa}}|\overset{6}{\texttt{ba}}|\overset{7}{\texttt{aba}}|\overset{8}{\texttt{ba\$}}$, where the vertical bars separate the factors. The LZ78 factorization is output as tuples: `(0,a)(1,a)(0,b)(1,b)(2,a)(3,a)(4,a)(6,$)`. This output is represented by the left trie (a). The LZ78U factorization of the same text is $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{ba}}|\overset{4}{\texttt{baa}}|\overset{5}{\texttt{aba}}|\overset{6}{\texttt{ababa}}|\overset{7}{\texttt{\$}}$. We output it as `(0,a)(1,a)(0,ba)(3,a)(1,ba)(5,ba)(0,$)`. This output induces the right tree (b).

3.3 LZ78U

A factorization $F_1 \cdots F_z = T$ is called the ***LZ78U factorization*** of T iff $F_x := T[i..j + \ell]$ with $T[i..j] = \text{argmax}_{S \in \{F_y : y < x\} \cup \{\epsilon\}} |S|$ and

$$\ell := \begin{cases} 1 \text{ if } T[i..j+1] \text{ is a unique substring of } T, \text{otherwise:} \\ 1 + \max\left\{\ell \in \mathbb{N}_0 \mid \forall k = 1, \ldots, \ell \nexists c \in \Sigma \setminus \{T[j+k+1]\} : T[i..j+k]c \text{ occurs in } T\right\}, \end{cases}$$

for all $1 \leq x \leq z$. Informally, we enlarge an LZ78 factor representing a repeated substring $T[i..i + \ell - 1]$ to $T[i..i+\ell]$ as long as the number of occurrences of $T[i..i+\ell-1]$ and $T[i..i+\ell]$ are the same.

Having the LZ78U factorization $F_1, \ldots, F_z$ of T, we can output each factor F_x as a tuple (y, S_x) such that $F_x = F_y S_x$, where F_y $(0 \leq y < x)$ is the longest previous factor (set $F_0 := \epsilon$) that is a prefix of F_x, and S_x is the suffix determined by the factorization. We call y the ***referred index*** and S_x the ***factor label*** of the x-th factor. Transforming the factors to this output induces a dictionary tree, called the ***LZ78U-tree***, in which

- every node corresponds to a factor,
- the parent of a node v corresponds to the referred index of v, and
- the edge between the node of the x-th factor and its parent is labeled with the factor label of the x-th factor.

Figure 6 shows a comparison to the LZ78-trie. By the definition of the factorizations, the LZ78-trie is a subtree of the suffix *trie*, whereas the LZ78U-tree[6] is a subtree of the suffix *tree*. The latter can be seen by the fact that the suffix tree compacts the unary paths of the suffix trie. This fact is the foundation of two algorithms we will subsequently show. Both algorithms build the LZ78U-tree on top of the suffix tree. They are easier computable variants of the LZ78 algorithms in [8, 17]. We present

(1) a streaming algorithm with $n \lg n + |\mathsf{ST}|$ bits of working space, and

(2) an offline algorithm using $|\mathsf{ST}| + n + z(\lg(2n) + \lg z) + 2z \lg n + o(n)$ bits of working space.

(1) Streaming Algorithm. The internal suffix tree nodes can be mapped to the pre-order numbers $[1..n]$ injectively by using rank/select data structures on the suffix tree topology. This allows us to use $n \lg n$ bits for storing a factor id in each internal suffix tree node. To this end, we create an array R of $n \lg n$ bits. All elements of the array are initially set to zero. In order to compute the factorization, we scan the text from left to right. Given that we are at text position i, we locate the suffix tree leaf ℓ corresponding to the i-th suffix. Let p be ℓ's parent.

[6] we named the algorithm LZ78U because each factor label represents a **u**nary path of the suffix trie, unless the path leads to leaf.

- If $R[p] \neq 0$, then p corresponds to a factor F_x. Let c be the first character of the edge label $\lambda(p, \ell)$. The substring $F_x c$ occurs exactly once in T, otherwise ℓ would not be a leaf. Consequently, we output a factor consisting of the referred index $R[p]$ and the string label c. We further increment i by the string depth of p plus one.
- Otherwise, using level ancestor queries, we search for the highest node v with $R[v] = 0$ on the path between the root (exclusively) and p. We set $R[v] \leftarrow z + 1$, where z is the current number of computed factors. We output the referred index $R[\text{parent}(v)]$ and the string $\lambda(\text{parent}(v), v)$. Finally, we increment i by the string depth of v.

Since level ancestor queries can be answered in constant time, we can compute a factor in time linear to its length. Summing over all factors we get linear time overall.

The code that computes the LZ78U factorization with variant (1) is shown in the appendix.

(2) Offline Algorithm. Instead of directly constructing the array R that is necessary to determine the referred indices, we create a list F storing the marked LZ-trie nodes, and a bit vector B marking the internal nodes belonging to the LZ-tree. Initially, only the root node is marked in B. Let i, p and ℓ be defined as in the above tree traversal. If $B[p]$ is set, then we append ℓ to F and increment i by one. Otherwise, by using level ancestor queries, we search for the highest node v with $B[v] = 0$ on the path between the root and p. We set $B[v] \leftarrow 1$, and append v to F. Additionally, we increment i by $|\lambda(\text{parent}(v), v)|$. By doing so, we have computed the factorization.

In order to generate the final output, we augment B with a rank data structure, and create a permutation N that maps a marked suffix tree node to the factor it belongs. The permutation N is represented as an array of $z \lg z$ bits, where $N[B.\operatorname{rank}_1(F[x])] \leftarrow x$, for $1 \leq x \leq z$. At this point, we no longer need F. The rest of the algorithm sorts the factors in the factor index order. To this end, we create an array R with $z \lg z$ bits to store the referred indices, and an array S with $z \lg n$ bits to store the factor labels. To compute S and R, we scan all marked nodes in B: Since the x-th marked node v corresponds to the $N[x]$-th factor, we can fill up S easily: If v is a leaf, we store the first character of $\lambda(\text{parent}(v), v)$ in $S[N[x]]$; otherwise (v is an internal node), we store the whole string. Filling R is also easy if v is a child of the root: we simply store the referred index 0. Otherwise, the parent p of v is not the root; p corresponds to the y-th factor, where $y := N[B.\operatorname{rank}_1(p)]$.

We get linear running time with the same argument as for (1).

Improved Compression Ratio. To achieve an improved compression ratio, we factorize the factor labels: If S_x is the label of the x-th factor f_x, then we factorize $S_x = G_1 \cdots G_m$ with $G_j := \operatorname{argmax}_{S \in \{F_y : y < x, |F_y| \geq \vartheta\} \cup \Sigma} |S|$ greedily chosen for ascending values of j with $1 \leq j \leq m$, with a threshold $\vartheta \geq 1$. By doing so, the string S_x gets partitioned into characters and former factors longer than ϑ. The factorization of S_x is done in $\mathcal{O}(|S_x|)$ time by traversing the suffix tree with level ancestor queries, as above (the only difference is that we do not introduce a new factor to the LZ78U factorization).

4 Practical Evaluation

Experimental Data. Table 1 shows the text collections used for the evaluation in the tudocomp benchmarks. We provide a tool that automatically downloads and prepares a superset of the collections used in this evaluation. The collections with the prefixes PC or PCR belong to the Pizza&Chili Corpus[7]. The Pizza&Chili Corpus is divided in a real text corpus (PC), and in a repetitive corpus (PCR). The collection HASHTAG is a tab-separated values file with five columns (integer values, a hashtag and a title) [7]. The collection TAGME is a list of Wikipedia fragments[8]. Finally, we present two new text collections. The first collection, called WIKI-ALL-VITAL, consists of the approx. 10,000 most vital Wikipedia articles[9]. We gathered all articles and processed them with the Wikipedia extractor of TANL [24] to convert each article into plain text. The second collection,

[7] http://pizzachili.dcc.uchile.cl
[8] http://acube.di.unipi.it/tagme-dataset
[9] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Expanded

collection	σ	max lcp	avg_{LCP}	bwt-runs	z	$\max_x \lvert F_x \rvert$	H_0	H_1	H_2	H_3
HASHTAG	179	54,075	84	63,014K	13,721K	54,056	4.59	3.06	2.69	2.46
PC-DBLP.XML	97	1084	44	29,585K	7035K	1060	5.26	3.48	2.17	1.43
PC-DNA	17	97,979	60	128,863K	13,970K	97,966	1.97	1.93	1.92	1.92
PC-ENGLISH	226	987,770	9390	72,032K	13,971K	987,766	4.52	3.62	2.95	2.42
PC-PROTEINS	26	45,704	278	108,459K	20,875K	45,703	4.20	4.18	4.16	4.07
PCR-CERE	6	175,655	3541	10,422K	1447K	175,643	2.19	1.81	1.81	1.80
PCR-EINSTEIN.EN	125	935,920	45,983	153K	496K	906,995	4.92	3.66	2.61	1.63
PCR-KERNEL	161	2,755,550	149,872	2718K	775K	2,755,550	5.38	4.03	2.93	2.05
PCR-PARA	6	72,544	2268	13,576K	1927K	70,680	2.12	1.88	1.88	1.87
PC-SOURCES	231	307,871	373	47,651K	11,542K	307,871	5.47	4.08	3.10	2.34
TAGME	206	1281	26	65,195K	13,841K	1279	4.90	3.77	3.20	2.60
WIKI-ALL-VITAL	205	8607	15	80,609K	16,274K	8607	4.56	3.62	3.03	2.45
COMMONCRAWL	115	246,266	1727	45,899K	10,791K	246,266	5.37	4.30	3.55	2.78

Table 1 Datasets of size 200MiB. The alphabet size σ includes the terminating `$`-character. The expression avg_{LCP} is the average of all LCP values. z is the number of LZ77 factors with $\vartheta = 1$. The number of runs consisting of one character in BWT is called bwt-runs. H_k denotes the k-th order empirical entropy.

named COMMONCRAWL, is composed of a random subset of a web crawl[10]; this subset contains only the plain texts (i.e., without header and HTML tags) of web sites with ASCII characters. A detailed description of the text collections is available at `http://tudocomp.org/text-collection.html`.

Setup. The experiments were conducted on a machine with 32 GB of RAM, an Intel Xeon CPU `E3-1271 v3` and a Samsung SSD `850 EVO 250GB`. The operating system was a 64-bit version of Ubuntu Linux 14.04 with the kernel version 3.13. We used a single execution thread for the experiments. The source code was compiled using the GNU compiler `g++ 6.2.0` with the compile flags `-O3 -march=native -DNDEBUG`.

lcpcomp Strategies. For lcpcomp we use the heap strategy and the list decompression strategy described in Section 3.2. We call them `heap` and `compact`, respectively. The strategies described in Section 3.2.3 are called `arrays` (compression) and `scan` (decompression). The decompression strategy `scan` takes the number of scans α as an argument. We evaluated lcpcomp only with the coder `sle`, since it provided the best compression ratio. We produced SA, ISA and LCP in the `delayed` mode.

LZ78U Implementation. We used the suffix tree implementation `cst_sada` of SDSL, since it provides all required operations like level ancestor queries.

Figure 7 visualizes the execution of lcpcomp with the strategy `arrays` in different phases for the collection PC-ENGLISH. The figure is generated with the JSON output of `tdc` by the chart visualization application on our website `http://tudocomp.org/charter`. We loaded the text (200MiB), constructed SA (800MiB, 32 bits per entry), computed LCP (500MiB, 20-bits per entry), computed ISA (700MiB, 28 bits per entry), and shrunk SA to 700MiB. Summing these memory sizes gives a memory offset of 1.9GiB when lcpcomp started its actual factorization. The factorization is divided in LCP value ranges. After the factorization, the factors were sorted and finally transformed to a binary bit sequence by `sle`. Most of the running time was spent on building SA, roughly 1GiB was spent for creating the lists L_i containing the suffix array entries with an LCP value of i.

Finally, we compare the implemented algorithms of tudocomp with some classic compression programs like `gzip` by our comparison tool `compare.py`. The output of the tool is shown in Table 2. The compressor `lzss_lcp` computes the LZ77 factorization (Def. 1) by a variant of [14]. The

[10] `http://commoncrawl.org`

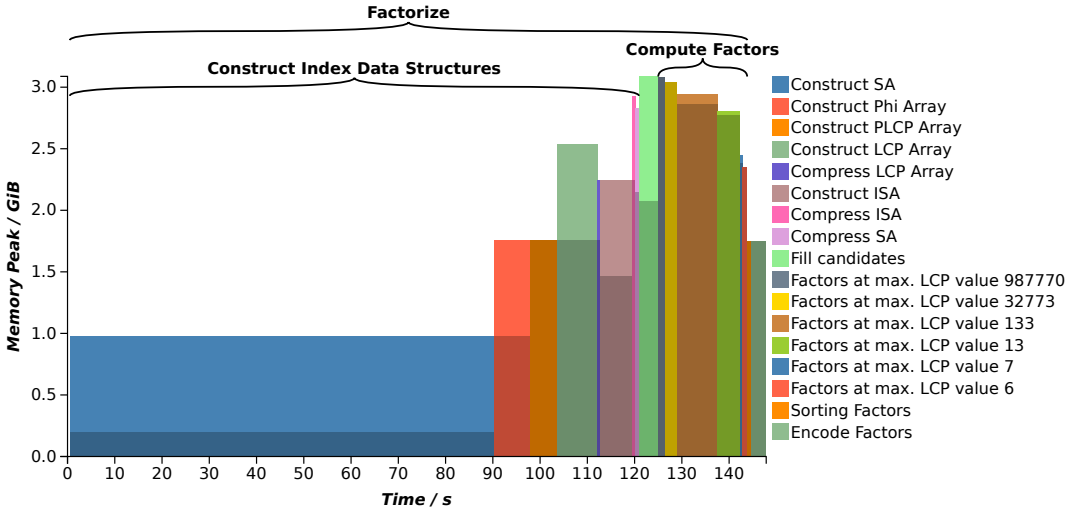


Figure 7 Compression of the collection PC-ENGLISH with `lcpcomp(coder=sle, threshold=5, comp=arrays)`. SA and LCP are built in `delayed` mode. Each phase of the algorithm (like the construction of SA) is depicted as a bar in the diagram. Each bar is additionally highlighted in a different color with a light and a dark shade. The darker part of a phase's bar is the amount of memory already reserved when entering the phase; the lighter part shows the memory peak on top of the already reserved space of the current phase. The memory consumption of a phase on termination is equal to the darker bar of the next phase. Coherent phases are grouped together by curly braces on the top.

compressor `bwtzip` is an alias for the compression pipeline `bwt:rle:mtf:encode(huff)` devised in Section 2.1. The programs `bzip2` and `gzip` do not compress the highly repetitive collection PCR-CERE as well as any of the tudocomp compressors (excluding the plain usage of a coder). Still, our algorithms are inferior to `lzma -9` in the compression ratio and the decompression speed. The high memory consumption of `LZ78U` is mainly due to the usage of the compressed suffix tree.

5 Conclusions

The framework tudocomp consists of a compression library, the command line executable `tdc`, a comparison tool, and a visualization tool. The library provides classic compressors and standard coders to facilitate building a compressor, or constructing a complex compression pipeline. Since the library was built with a focus on high modularity, a compression pipeline does not have to get statically compiled. Instead, the tool `tdc` can assemble a compression pipeline at runtime. Such a pipeline, given as a parameter to `tdc`, can be adjusted in detail at runtime.

We demonstrated tudocomp's capabilities with the implementation of two new compressors: lcpcomp, a variant of LZ77, and LZ78U, a variant of LZ78. Both new variants show better compression ratios than their respective originals, but have a higher memory consumption and also slower decompression times. Further research is needed to address these issues.

Future Research. The memory footprint of lcpcomp could be dropped by exchanging the array implementations of SA, ISA and LCP with compressed data structures like a compressed suffix array, an inverse suffix array sampling, and a permuted LCP (PLCP) array, respectively. We are currently investigating a variant that only observes the peaks in the PLCP array to compute the same output as lcpcomp. If the number of peaks is π, then this algorithm needs at most $\pi \lg n$ bits on top of SA, ISA and the PLCP array.

We are optimistic that we can easily improve the compression ratio of our algorithms by using adaptive coders like an adaptive arithmetic coder.

```
pcr_cere.200MB (200.0MiB, sha256=577486b84633ebc71a8ca4af971eaa4e6a91bcddda17f0464ff79038cf928eab)

                       Compressor |     C Time |   C Memory |     C Rate |     D Time |   D Memory | chk |
----------------------------------------------------------------------------------------------------------
                  lz78u(t=5,huff) |     280.2s |     9.2GiB |   12.4643% |      5.1s |   286.9MiB |  OK |
         lcpcomp(t=5,heap,compact) |     235.5s |     3.4GiB |    2.8436% |     36.4s |     7.6GiB |  OK |
       lcpcomp(t=5,arrays,compact) |     103.1s |     3.2GiB |    2.8505% |     36.6s |     7.6GiB |  OK |
  lcpcomp(t=5,arrays,scans(a=25)) |     104.6s |     3.2GiB |    2.8505% |     37.2s |     4.6GiB |  OK |
                lzss_lcp(t=5,bit) |      98.5s |     2.9GiB |    4.0530% |      4.3s |   230.6MiB |  OK |
                            code2 |      16.4s |   230.6MiB |   28.4704% |      6.6s |    30.6MiB |  OK |
                             huff |       2.7s |   230.5MiB |   28.1072% |      5.9s |    30.6MiB |  OK |
                              lzw |      14.3s |   480.9MiB |   23.4411% |      5.5s |   452.6MiB |  OK |
                             lz78 |      13.6s |   480.8MiB |   29.1033% |     10.3s |   142.9MiB |  OK |
                            bwtzip |      83.6s |     1.7GiB |    6.8688% |     22.6s |     1.4GiB |  OK |
                           gzip -1 |       2.6s |     6.6MiB |   30.7312% |      1.4s |     6.6MiB |  OK |
                           gzip -9 |     107.6s |     6.6MiB |   26.2159% |      1.0s |     6.6MiB |  OK |
                          bzip2 -1 |      13.1s |     9.3MiB |   25.3806% |      5.1s |     8.6MiB |  OK |
                          bzip2 -9 |      13.8s |    15.4MiB |   25.2368% |      5.6s |    11.7MiB |  OK |
                           lzma -1 |      12.6s |    27.2MiB |   27.6205% |      3.4s |    19.7MiB |  OK |
                           lzma -9 |     138.6s |   691.7MiB |    1.9047% |    337.3ms |    82.7MiB |  OK |
```

Table 2 Output of the comparison tool for the collection PCR-CERE. `C` and `D` denote the compression and decompression phase, respectively. `a` and `t` are the parameters α and ϑ, respectively. The tool checks at the last column whether the `sha256`-checksum of the decompressed output matches the input file.

References

1 J. Alakuijala and Z. Szabadka. Brotli Compressed Data Format. RFC 7932, 2016.

2 D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

3 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

4 D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

5 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

6 A. Farruggia, P. Ferragina, and R. Venturini. Bicriteria data compression: Efficient and usable. In *Proc. ESA*, volume 8737 of *LNCS*, pages 406–417. Springer, 2014.

7 P. Ferragina, F. Piccinno, and R. Santoro. On analyzing hashtags in Twitter. In *Proc. ICWSM*, pages 110–119, 2015.

8 J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.

9 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, first edition, 1995.

10 S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.

11 R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, volume 7933 of *LNCS*, pages 5–17. Springer, 2013.

12 J. Holub, J. Reznicek, and F. Simek. Lossless data compression testbed: ExCom and Prague corpus. In *Proc. DCC*, page 457. IEEE Computer Society, 2011.

13 G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

14 J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

15 J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proc. CPM*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

16 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):1–19, 2006.

17 D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12. IEEE Computer Society, 2016.

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

19 U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

20 W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara. A series of run-rich strings. In *Proc. LATA*, volume 5457 of *LNCS*, pages 578–587. Springer, 2009.

21 R. Nakamura, S. Inenaga, H. Bannai, T. Funamoto, M. Takeda, and A. Shinohara. Linear-time text compression by longest-first substitution. *Algorithms*, 2(4):1429–1448, 2009.

22 S. Ristov and D. Korencic. Using static suffix array in dynamic application: Case of text compression by longest first substitution. *Inf. Process. Lett.*, 115(2):175–181, 2015.

23 K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

24 M. Simi and G. Attardi. Adapting the tanl tool suite to universal dependencies. In *Proc. LREC*. European Language Resources Association, 2016.

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

26 T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

27 I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

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

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

A More Evaluation

In this section, the execution time is measured in second, and all data sizes are measured in mebibytes (MiB).

In Table 3, we selected the ϑ with the best compression ratio and the α with the shortest decompression time. Although ϑ and α tend to correlate with the compression speed and decompression memory, respectively, selecting values for ϑ and α that yield a good compression ratio or a fast decompression speed seems difficult.

In Table 4, we fixed two values of ϑ and three values of α. The compression ratio of the strategies `heap` and `arrays` differ slightly, since the lcpcomp compression scheme does not specify a tie breaking rule for choosing a longest repeated substring.

Figure 8 compares the number of factors of `lzss_lcp` with `lcpcomp`'s `arrays` strategy on all aforementioned datasets. We varied the threshold ϑ from 4 up to 22 and measured for each ϑ the number of created factors. In all cases, lcpcomp produces less factors than `lzss_lcp` with the same threshold.

	compression					decompression		
collection	ϑ	#factors	ratio	memory	time	α	memory	time
HASHTAG	5	10,088,662	25.47%	3179.9	100	17	1726	50
PC-DBLP.XML	5	5,547,102	14.4 %	2929.7	99	28	1993.5	65
PC-DNA	21	1,091,010	26.03%	2925	122	11	291.2	8
PC-ENGLISH	5	11,405,635	27.66%	3162	123	25	792.6	36
PC-PROTEINS	10	1,749,917	35.91%	2900	124	13	362	11
PCR-CERE	22	236,551	2.45 %	3126	113	6	454.2	7
PCR-EINSTEIN.EN	8	24,672	0.1 %	3288.8	113	40	1777.3	47
PCR-KERNEL	6	512,047	1.51 %	3356.3	116	40	2129.6	37
PCR-PARA	22	388,195	3.27 %	3060.8	117	6	402.3	7
PC-SOURCES	5	8,922,703	23.36%	3271	98	30	1019.6	36
TAGME	5	10,986,096	27.29%	2987.7	113	25	985.4	41
WIKI-ALL-VITAL	5	13,338,470	32.46%	3163	117	27	870.4	45
COMMONCRAWL	4	8,402,041	21.49%	3254.6	101	36	1206.11	41

Table 3 Compression and decompression with the lcpcomp strategies `arrays` and `scan`, for fixed parameters ϑ and α. For each collection we chose the ϑ with the best compression ratio. Having ϑ fixed, we chose the $\alpha \leq 40$ with the shortest decompression running time.

	compression			decompression		
compressor	memory	output size	time	strategy	memory	time
external programs						
`gzip -1`	6.6	61.3	2.19		6.6	1.045
`bzip2 -1`	9.3	55.4	14.455		8.6	4.7
`lzma -1`	27.2	46.7	9.395		19.7	2.37
`gzip -9`	6.6	53.4	6.86		6.6	0.97
`bzip2 -9`	15.4	50.7	14.78		11.7	4.955
`lzma -9e`	691.7	29.4	104.375		82.7	1.56
tudocomp algorithms						
`encode(sle)`	265.2	137.7	24.145		30.6	10.095
`encode(huff)`	230.4	135	5.7		30.4	9.045
`bwtzip`	1730.6	43.7	83.035		1575	21.44
`lcpcomp(`$\vartheta = 5$`,heap)`	3598.9	44.1	228.055	`compact`	6592.2	33.24
`lcpcomp(`$\vartheta = 22$`,heap)`	3161.7	58.5	175.21	`compact`	3981.2	14.065
`lcpcomp(`$\vartheta = 5$`,arrays)`	3354.2	44.3	107.34	`scan(`$\alpha = 6$`)`	4930	43.1
				`scan(`$\alpha = 25$`)`	2584.5	33.995
				`scan(`$\alpha = 60$`)`	1164.8	38.925
`lcpcomp(`$\vartheta = 22$`,arrays)`	2980.6	58.5	109.245	`scan(`$\alpha = 6$`)`	1308	10.925
				`scan(`$\alpha = 25$`)`	520.9	11.265
				`scan(`$\alpha = 60$`)`	368.7	15.635
`lzss(bit)`	2980.4	60.2	108.59		230.6	6.045
`lz78(bit)`	480.8	83.1	17.96		254.9	11.46
`lzw(bit)`	480.8	70.3	18.97		663.1	7.05

Table 4 Evaluation of external compression programs and algorithms of the tudocomp framework on the collection COMMONCRAWL.

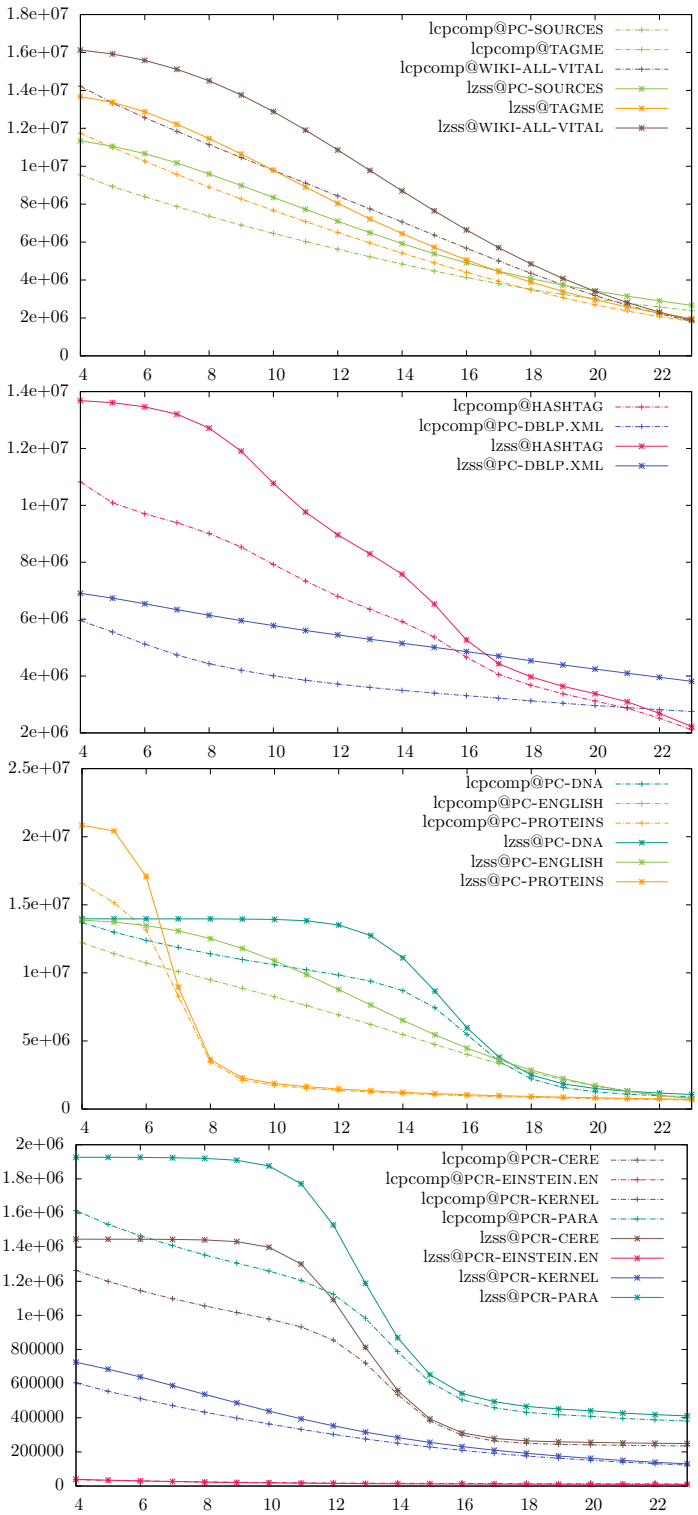


Figure 8 Number of factors (y-axis) of lcpcomp and LZ77 on varying the given threshold ϑ (x-axis).

B Elaborated Example of lcpcomp

Figure 9 demonstrates how the lcpcomp factorization of the running example is done step-by-step.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	$
SA$[i]$	17	16	7	1	8	11	2	14	5	9	12	3	15	6	10	13	4
LCP$[i]$	-	0	1	5	2	4	6	1	3	4	3	5	0	2	3	2	4
LCP$^1[i]$	-	0	0	1	2	4	0	1	0	4	3	0	0	0	3	2	0
LCP$^2[i]$	-	0	0	1	2	0	0	0	0	2	0	0	0	0	1	0	0
LCP$^3[i]$	-	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9 Step-by-step computation of the lcpcomp compression scheme in Figure 3b. We scan for the largest LCP value in LCP and overwrite values in LCP instead of using a heap. Each row LCP$^i[i]$ shows the LCP array after computing a substitution. The LCP value of the starting position of the selected largest repeated substring has a green border. The updated values are colored, either due to deletion (red) or key reduction (blue). Ties are broken arbitrarily. The number of red zeros in each row is equal to the number above the green bordered zero in the corresponding row minus one.

C LZ78U Code Snippets

```
void factorize(TextDS<>& T, SuffixTree& ST, std::function<void(int begin, int end, int ref)>
     output){
 typedef SuffixTree::node_type node_t;
 sdsl::int_vector<> R(ST.internal_nodes,0,bits_for(T.size() * bits_for(ST.cst.csa.sigma) /
      bits_for(T.size())));
 int pos = 0, z = 0;
 while(pos < T.size() - 1) {
  node_t l = ST.select_leaf(ST.cst.csa.isa[pos]);
  int leaflabel = pos;
  if(ST.parent(l) == ST.root || R[ST.nid(ST.parent(l))] != 0) {
   int parent_strdepth = ST.str_depth(ST.parent(l));
   output(pos + parent_strdepth, pos + parent_strdepth + 1, R[ST.nid(ST.parent(l))]);
   pos += parent_strdepth+1;
   ++z;
   continue;
  }
  int d = 1;
  node_t parent = ST.root;
  node_t node = ST.level_anc(l, d);
  while(R[ST.nid(node)] != 0) {
   parent = node;
   node = ST.level_anc(l, ++d);
  }
  pos += ST.str_depth(parent);
  int begin = leaflabel + ST.str_depth(parent);
  int end = leaflabel + ST.str_depth(node);
  output(begin, end, R[ST.nid(ST.parent(node))]);
  R[ST.nid(node)] = ++z;
  pos += end - begin;
 }
}
```

Figure 10 Implementation of the LZ78U algorithm streaming the output

Algorithm 1: Streaming LZ78U

```
1  ST ← suffix tree of T
2  R ← array of size n        // maps internal suffix tree nodes to LZ trie ids
3  initialize R with zeros
4  pos ← 1                                                    // text position
5  z ← 0                                                      // number of factors
6  while pos ≤ |T| do
7      ℓ ← leaf-select(ISA[pos])
8      if R[parent(ℓ)] ≠ 0 or parent(ℓ) = root then
9          output the first character of λ(parent(ℓ), ℓ)
10         output referred index R[parent(node)]
11         z ← z + 1
12         pos ← pos + str_depth(parent) + 1
13     else
14         d ← 1                                                  // the current depth
15         while R[level-anc(ℓ, d)] ≠ 0 do
16             d ← d + 1
17             pos ← pos + |λ(level-anc(ℓ, d − 1), level-anc(ℓ, d))|
18         node ← level-anc(ℓ, d)
19         z ← z + 1
20         R[node] ← z
21         output string λ(parent(node), node)
22         output referred index R[parent(node)]
23         pos ← pos + |λ(parent(node), node)|
```

Algorithm 2: Computing LZ78U memory-efficiently

```
1  ST ← suffix tree of T
2  pos ← 1
3  B ← bit vector of size n     // marking the ST nodes belonging to the LZ-trie
4  F ← list of integers      // storing the LZ-trie nodes in the order when they
   got explored
5  node ← root of ST
6  while pos ≤ |T| do
7      node ← child(node, T[pos])                    // use level-anc to get O(1) time
8      pos ← pos + (is-leaf(node) ? 1 : |λ(parent(node), node)|
9      if is-leaf(node) or B[node] = 0 then
10         B[node] ← 1
11         F.append(node)
12         node ← root of ST
13 add_rank_support(B)
14 N ← array of length z     // stores for each marked ST node to which factor
   it belongs
15 for 1 ≤ x ≤ z do N[B.rank_1(F[x])] ← x
16 F ← integer array of size z                       // storing the referred indices
17 S ← string array of size z                  // storing the string of each factor
18 for 1 ≤ x ≤ z do
19     node ← B.rank_1(x)
20     if is-leaf(node) then S[N[x]] ← first character of λ(parent(node), node)
21     else S[N[x]] ← λ(parent(node), node)
22     if parent(node) = root then F[N[x]] ← 0
23     else F[N[x]] ← N[B.rank_1(parent(node))]
24 return (F,S)
```