The Open University of Israel

Department of Mathematics and Computer Science

Shapira/Klein's Compressed Matching

Performance Comparison of Shapira/Klein's Compressed Pattern Matching Method with Manber's and "Decompress and Search" Methods

Final Paper submitted as partial fulfillment of the requirements
towards an M. Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By
Khovansky Alexander

Prepared under the supervision of Dr. Dana Shapira

February 2010

Table of Contents

List of Tables

Abstract

Compressed pattern matching is a problem of performing pattern matching directly in a compressed text. This paper compares the performance of Shapira/Klein's method ([1]) with Manber's ([2]), LZSS([3]) and “Decompress and Search” ones, where the “Decompress and Search” means searching the decompressed text. The methods are compared by (1) compression performance, (2) compression processing time, and (3) pattern matching processing time. Compression processing time and performance are also compared to those of the '*gzip*' UNIX utility ([4], [5]).

The compression method used in “Decompress and Search” is based on LZSS algorithm ([3]). The pattern matching algorithms used in all three methods are based on KMP ([6]).

The Work Goal

The main purpose of this work is to assess the performance of Shapira/Klein's compression and pattern matching methods, by comparing it with another related methods.

The Work Methods

Each compared method is implemented and its performance is measured on standard data sets. The implementation language is Java.

1 Introduction

1.1 Motivation for Compressed Pattern Matching

The general approach for looking for a pattern in a file that is stored in its compressed form is first decompressing and then applying one of the known pattern matching algorithms on the decoded file. But decompression normally requires much more space and time than a search. As a result, files that are searched frequently are in many cases stored in their original form. Efficient compressed pattern matching method would allow these files to be stored in their compressed forms.

1.2 Definition of Compressed Pattern Matching

Amir at al. [7] defines the compressed pattern matching as a problem of finding all occurrences of a pattern in a compressed text in time proportional to the compressed text's size. One approach to achieve this is first to compress the search pattern and then to search the compressed pattern in a compressed file. Manber's pattern matching algorithm ([2]) uses this approach. Another approach, which approximates compressed pattern matching, is to search for the original pattern, but to decompress only the relevant parts of the compressed file and to skip the irrelevant ones. Shapira/Klein's method [1] uses this second approach.

1.3 “Search During Decompress”

The “Decompress and Search” method is first decompressing and then searching within the decompressed file. One of the main disadvantages of this method is that it requires the entire decompressed file to be stored in additional memory storage. One possible compromise between the compressed matching and the “Decompress and Search” method is to perform the search **during** decompression. It usually requires much smaller amount of additional memory than the “Decompress and Search” and might be fast enough in practice. This method can be applied to every compression method that allows decompression from left to right. Let us call this method “Decompress During Search”. This paper compares the performances of Shapira/Klein's [1] and Manber's [2] methods to both “Decompress and Search” and “Search During Decompress” in LZSS-compressed text.

2 Overview of Compared Methods

There are several principal approaches to handle the compressed matching problem. One of them is to try adapting some pattern matching algorithm to a given compression scheme. Both Manber's ([2]) and Shapira/Klein's ([1]) methods use a different road: they suggest changing the compression method itself to allow easy searching in the compressed file. This approach was initially proposed by Manber ([2]). Shapira/Klein's method ([1]) strives to improve the compression performance achieved by Manber using a different compression scheme.

(3)

2.1 Manber's Compression

Manber's compression method ([2]) uses the fact that although a byte can represent 256 different symbols, usually only a subset of them is actually used in an average text file. Manber's compression algorithm utilizes the unused symbols by replacing common pairs of characters by these unused symbols. The savings in space results from the fact that pairs of characters (two bytes) are replaced by single characters (one byte). The mapping between the replaced pairs and the new symbols is stored in the beginning of the compressed file. To search in a compressed file, the search pattern is first encoded using the mapping stored at the beginning of the file, and then the search is performed using any known pattern matching algorithm. However, the pairs to be replaced should be chosen with care, e.g. a word "wood" might be encoded either as $\boxed{wo}\ \boxed{od}$, or as $\boxed{?w}\ \boxed{oo}\ \boxed{d?}$ (here '?' could be any character, e.g. white space). Manber's compression algorithm chooses the pairs that do not cause such ambiguities. Specifically, it chooses the most frequent pairs so that no character will be both a first character of some pair and a second character of (probably another) pair. Manber's compression method implementation is described in more details in section 3.1 .

Note that Manber's method confines itself to bytes, i.e. every new character occupies exactly one byte. This means that virtually every known pattern matching algorithm can be applied to search in the compressed file. Moreover, due to compression, the search time in Manber-compressed text is even smaller than in the original text! This is different from e.g. binary Huffman coding ([8]), which might be more efficient in terms of compression ratio, but using a bit based search it might require longer processing time. The reason is that in case of binary Huffman code ([8]), decoding a character requires processing the compressed text bit by bit, which is much slower than reading characters on a byte level. Another reason is that some fastest pattern matching algorithms allow skipping some text altogether, e.g. Boyer-Moor ([9]), and they cannot be applied to an adaptive compressed text.

2.2 Manber's Pattern Matching

The searching starts by reading the mapping between the old and new characters from the beginning of the compressed file. It then compresses the pattern using this mapping. In some cases, the first and the last character of the pattern should be removed from the compressed search pattern. Some pattern matching algorithm is then invoked as a black box, to locate the compressed pattern in the compressed text. The output of the match is then decompressed. In case the first or last character of the pattern were removed, this output is further filtered. Note that Manber's pattern matching does not impose any restrictions on the core pattern matching algorithm. We used KMP pattern matching algorithm ([6]), which is briefly described in section 2.4 .

2.3 "Decompress and Search" Method: LZSS-Based Compression

In this method we use LZSS ([3]) to compress the original text. In LZSS, a text is encoded as a sequence of elements which are either single characters or pointers to previously occurring strings. These pointers are encoded as ordered pairs of numbers, denoted (*off*, *len*), where *off* is the number of characters from the current location to the previous occurrence of a sub-string matching the one that starts at the current location, and *len* is the length of the matching string.

For example, the LZSS encoding of string "*acdeabc**eabc**deaeab*" is "*acdeabc(4,4)(9,3)(7,3)*". Here the pair (4,4) represents a string "*eabc*", which already occurred 4 characters earlier and has a length of 4.

The basic LZSS algorithm allocates a fixed number of bits to *off* and *len*, thus bounding the maximum value of *off* and *len*. For example, if the number of bits allocated to *off* is 12, its value can not exceed $2^{12}+1$. This value is called "***text window size***". The pair (*off*, *len*) occupies an integer number of bytes, so the number of bits allocated to both *off* and *len* is usually chosen to be a multiple of 8, e.g. 12 bits for *off* and 4 bits for *len.* In fact, Williams ([10]) used 12-bits offset size and 4-bit length size to produce a very fast LZSS-based compression algorithm.

The compression ratio of LZSS can be improved if we don't fix the number of bits allocated to (*off*, *len*) pairs. For instance, the first (*off*, *len*) pair could occupy 8 bits, since the values of *off* and *len* are small, while the second pair could occupy 24 bits, since the values of *off* and *len* are big. But this requires a method to recognize the actual number of bits occupied by a specific (*off*, *len*) pair. The method yielding the best compression ratio would be probably to use Huffman ([8]) or arithmetic coding

([11]) to encode (`off`, `len`) pairs as short as possible. Indeed, `'gzip'` ([4], [5]) uses binary Huffman codings, one for all text characters and all encountered values of `len`, and another one for all encountered values of `off`. The main method used in this paper is different: it confines to a fixed number of (`off`, `len`) bit allocations. However, a method using two binary Huffman trees was also implemented. The details are described in section 3.2 .

2.4 "Decompress and Search" Method: KMP Pattern Matching

To search for a string, we first decompress the entire file and then invoke the Knuth-Morris-Pratt (KMP) pattern matching algorithm ([6]) on the decompressed text.

The algorithm KMP searches for occurrences of a search pattern $S=S_1S_{2..}..S_n$ within a main "text string" $T=T_1T_2...T_k$ by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. To do so, the algorithm first preprocesses the pattern *S* to construct a "***prefix function***" ***π***, which indicates where we need to look for the beginning of a new match in case the current one ends in a mismatch. Specifically, *π[j],* $j \le |S|$ is defined to be the length of the longest prefix of the pattern S[1,..,j] that is also a suffix of S[2,..,j]. The algorithm starts the search by comparing the characters of *T* to the characters of *S*. If a match was assumed to start at *T[m]*, and a mismatch was found at *T[m+i]≠S[1+i]*, the algorithm is able to conclude that the next possible match could only start at *T[m + i - π[i]]*, so it will check *T[m+i]* versus *S[π[i]]*. This is in contrast to a naïve algorithm that in this case would check *T[m+1]* versus *S[1]*. The search time of the algorithm is *O(|T| + |S|)*, since the position in T is never reduced.

For example, let *T="abcabf..."* and the pattern *S="abcabe"*. The algorithm first constructs the prefix function *π* with the following values:

```
π[0] = π[1] = 0
π[2] = 0
π[3] = 0
π[4] = 1
π[5] = 2
```

$\pi[6] = 0$

The meaning of $\pi[5] = 2$, for example, is that there is a prefix of S of length 2 (i.e. "*ab*") ending at $S[5]$. The algorithm starts the search at index 1 of the text by checking character by character until it encounters a mismatch at index 6 (since $T[6]='f' \neq S[6]='e'$). This means that the match cannot start at $T[1]$. The naive algorithm would then check whether a match exists at $T[2]$. However, KMP is able to conclude immediately that the match cannot occur at $T[2]$ and neither at $T[3]$, and for a moment would assume a match starting at $T[4]$ (since $4 = 1 + 5 - \pi[5]$). But it would then immediately understand that this cannot happen by checking $T[6]$ with $S[3]$ and seeing that they differ.

2.5 "Search During Decompress" in LZSS-Compressed Text

As mentioned in section 1.3, the search times of Manber's [2] and Shapira/Klein's [1] algorithms are compared both to "Decompress and Search" and "Search During Decompress" methods in LZSS-compressed text. The "Search During Decompress" algorithm decompresses character after character, and feeds them into the KMP pattern matching algorithm ([6]) described above. The decompression stops once a desired number of matches is found or when it reaches the end of the input, if it looks for all matches. To reduce the memory usage, the decompression algorithm stores the decompressed characters in a simple circular buffer whose size equals the *maximum text window* size, in contrast to "Decompress and Search" algorithm which stores the entire decompressed file.

2.6 Shapira/Klein's Method: Compression Algorithm

The purpose of the algorithm ([1]) is to change the LZSS compression scheme ([3]) to allow searching for a string directly in the compressed text, in time proportional to the size of the compressed file, without the need to decompress the whole file before the search. This section describes the compression, while the pattern matching algorithm is described in the next section.

Unlike LZSS which replaces the reoccurring substring by a pointer *(off, len)*, Shapira/Klein's method stores the pointers immediately after the strings they refer to. For instance, the string "*abcde**abcd**k*", encoded by LZSS as "*abcde(5,4)k*" would now be encoded as "`abcd(1,4)ek`". Here, `'1'` means that "`abcd`" occurs one character after the end of the first occurrence of "`abcd`" (i.e. after `'e'`), while `'4'` is the length of the reoccurring string, as in LZSS.

The problem is that this method will not always work: for instance, in $a_1 b_2 c_3 \boldsymbol{d_4 e_5 a_6} b_7 c_8 d_9 \boldsymbol{d_{10} e_{11} a_{12}} k_{13}$ ("`abcdeabcddeak`") we would like to replace both the second occurrence of "`abcd`" (at index 6) and the second occurrence of "`dea`" (at index 10) by pairs `(off, len)`. But after we replace the second occurrence of "`abcd`" by a pointer and the string becomes "`abcd(1,4)edeak`", the first occurrence of "`dea`" disappears, since it overlaps the pointer reference `(1,4)`. Therefore, it is not clear where to place the pointer to the second `'dea'`.

The method solves this problem by introducing a third parameter, which the authors call "**`slide`**", to the pointer's structure, which now becomes a triple `(off, len, slide)`. The text in the example above becomes "`abcd(1,4)e(3,3,1)k`", where the triple `(3,3,1)` describes the second occurrence of the string "`dea`". `slide 1` here means that after replacing all pointers preceding the `(3, 3, 1)`, it should be shifted `1` character forward: $abc\,\boldsymbol{de}(\mathbf{3,3,1})\,\boldsymbol{a}\,bcdk \rightarrow abc\,\boldsymbol{dea}(\mathbf{3,3})\,bcdk$. The details on the '`slide`' parameter as well as other implementation details are described in section 3.3.

2.7 Shapira/Klein's Method: Pattern Matching Algorithm

The algorithm ([1]) searches for occurrences of a search pattern S within a text C compressed using Shapira/Klein's compression scheme described above. The algorithm traverses the text from left to right, partially replacing the pointers `(off, len[, slide])`. The decompressed characters are fed into an underlying pattern matching algorithm, which is used as a black box, as soon as the characters become available. Any left-to-right pattern matching algorithm can be used for this purpose, but to get an advantage from the method it should be able to efficiently skip any given number of successive characters. We use a slightly modified KMP pattern matching algorithm ([6]) for this purpose.

The main idea is to use the fact that not all characters of the original text appear at the pattern. This fact can be used to perform only partial decompression, namely to decompress only parts consisting of characters from the pattern and skipping the rest. The partially decompressed text is stored in a circular buffer, which holds the relevant characters at their proper places. Each block of successive relevant characters has a pointer to the next block, so the irrelevant characters are skipped not only during the partial decompression, but also during the search. The algorithm uses the fact that the *(off, len[, slide])* pointers appear immediately after the strings they refer to in order to find out which characters can be skipped.

Let us show the process of partial decompression by an example. Let the pattern *S="dk"* and the compressed text *C="dabcd(1,4)ek"* (which corresponds to the original text *"dabcdeabcdk"*). Let us use *'$'* to designate irrelevant characters. First, the algorithm processes the first five characters of the text (*"dabcd"*), character after character. The buffer becomes *"d$$$d"*, where the first *'d'* has also a pointer to the second *'d'*. Next, it partially replaces the pointer *(1,4)*, skipping the first three characters of the original string and copying only *'d'* to the corresponding place at the buffer, which now becomes *d$$$d$$$$d* . Finally, it copies *'e'* and *'k'* into the buffer, which now becomes *d$$$d$$$$dk* and finds the appearance of *"dk"*.

Note that the search time is proportional to the size of the compressed file only in the best case, in which most character blocks pointed by the pointers do not contain the "relevant" characters. In addition, even in the best case, the search time target is reached only when no additional encoding is applied to the pointers and characters produced by the basic Shapira/Klein's compression algorithm. When an additional encoding (such as binary Huffman encoding) is applied to the output of the basic Shapira/Klein's algorithm, the search time is no longer proportional to the size of the compressed file.

The implementation details of the algorithms are described in section 3.4.

2.8 Byte-Based Compression Methods

Let us emphasize that the main implementations of all compression methods used in this paper are based on a byte format, for search efficiency. A bit-format compression was implemented for LZSS compression only, but used only to verify the compression effectiveness achieved by other methods. The problem with a bit-format compression is that while improving the compression ratio it also increases the search time in the compressed text. Let us examine each of the compared methods.

Manber's method: staying with a byte format allowed a search in Manber-compressed text to be even faster than a search in the original text, which would be impossible to achieve if a bit-format compression was also applied. Manber explicitly rejected a bit-format compression in order to improve the search time.

Shapira/Klein's method: one of the main goals of Shapira/Klein's method was to improve the search time, probably at the cost of compression efficiency. In our paper we assumed that decoding of binary Huffman-encoded characters and pointers would take most of the search time, which would shadow any search time improvement in Shapira/Klein's compressed text over the search in LZSS-compressed text. This assumption requires verification, but it is not in the scope of this work.

LZSS method: for the sake of consistency, the main implementation of LZSS used in all comparisons also stays with a byte format. Still, to compute the penalty of staying with a byte format to the compression ratio, a bit-format LZSS compression was also implemented. The compression ratio of both bit-format and byte-format implementations is compared to that of *'gzip'*. This allows to verify the efficiency of our implementations of LZSS algorithm in terms of compression performance (and Shapira/Klein method's implementation which is based on that of LZSS).

Computing the search time penalty of employing a bit-format compression in all compared methods (Manber, LZSS and Shapira/Klein) is outside of the scope of this paper.

3 Implementation Details

This section describes the implementation details of the methods used in the comparison.

3.1 Manber's Compression Implementation

3.1.1 Reduction to Maximum Cut Problem

As described in section 2.1, the main idea of Manber 's compression ([2]) is to replace the most frequent pairs of characters (2 bytes) by new symbols (1 byte). To allow searching, the pairs are chosen so that no character will be both a first character of some pair and a second character of (probably another) pair. This problem is reduced to the problem of finding the maximum cut in a graph. The vertices are single characters, the edges are pairs of characters, and the edge weights are the frequencies of the pairs in the original text. The problem of finding the most frequent non-overlapping pairs of characters is equivalent to the problem of finding the maximum-weight cut in this graph.

3.1.2 Dealing with "Rare" Characters

In the original paper, the input data is assumed to contain only characters in the range [0, 127]. Characters from range [128, 255] are considered '*rare*' and are encoded as 2 bytes. For this purpose, a code '255' is considered special and is used to indicate the encoding of the rare characters. Any character that is considered 'rare' will be replaced by two symbols: first '255' and then the character itself.

In our implementation, we do not assume anything about the input. Instead, characters with frequency less than a user-specified threshold parameter are considered 'rare' and are encoded as 2 bytes, as described above. All other characters are considered '*frequent*' and are matched to a 1-byte symbol in the new alphabet.

Note that since the symbol '255' is reserved, if the input contains all 256 different characters, at least one character must be considered 'rare', even if no pairs are going to be chosen (since there are 256 different characters and at most 255 possible encodings).

3.1.3 Translation Table

The translation table from the new symbols to the old characters and character pairs is stored at the beginning of the compressed file. Let us describe the details of the format in which it is stored in the compressed text.

Let N be the number of 'frequent' characters, $N<=255$. These characters are mapped to codes from 0 to $N-1$. The most frequent pairs of characters are mapped to codes from N to 254. As it was mentioned above, the code 255 is reserved to deal with 'rare' characters. The format for writing the translation table is:

- Size of single-character alphabet (N), 1 byte
- Size of character pairs alphabet, 1 byte
- All single characters from the single-character alphabet in the correct order. That is, the character encoded as '0' appears first, followed by the character encoded as '1' etc., total N characters. Each character takes 1 byte.
- All chosen character pairs in the correct order, each pair takes 2 bytes.

The total number of encoded characters varies from 1 for an alphabet consisting of a single character to 255, and each encoded character may take either 1 or 2 bytes in the table. Hence, the dictionary's size may vary from 2+1=3 to no more than 2+255*2 bytes.

3.1.4 The Main Compression Algorithm

Input: arbitrary text

Output: Manber-compressed text

Algorithm:

1. Read in the original text and count the frequencies of each character and each character pair.
2. Find all `'frequent'` characters, i.e. characters with a frequency greater or equal to the user-specified threshold. Let *N* be the number of `'frequent'` characters.
3. Assign each such character a new code in a range *[0, N-1]*.
4. Compute *(255-N)* the most frequent non-overlapping pairs, using the maximum-weight cut algorithm – see section 3.1.5.
5. Assign a unique code in the range *[N, 254]* for each chosen pair of characters.
6. We received a static dictionary of size at most 2*255 bytes. Write this dictionary to the output.
7. Encode the original text using this dictionary.

3.1.5 Maximum-Weight Cut Algorithm

As it was already mentioned in the previous section, the problem of finding the most-frequent non-overlapping character pairs can be reduced to the problem of finding the maximum cut in a directed weighed graph. In this graph, the vertices are single characters, the edges are pairs of characters, and the edge weights are the frequencies of the pairs in the original text. It turns out that this problem is NP-complete, so Manber suggests an approximation algorithm based on a local search.

The algorithm works as follows: randomly choose an initial partition of vertices into 2 groups, and then try to improve the solution by moving each character from one group to the other. To improve the results, this process can be repeated. The number of repetitions is a user parameter.

Let us now describe the algorithm in more details. Let V be the set of all vertices, V_1 and V_2 be the two groups of vertices defining the cut. We wish to find a maximum-weight cut $\{V_1, V_2\}$ in this graph and then choose the N heaviest edges from this cut, where N is the input of the algorithm. The algorithm uses two arrays W_1 and W_2 of size $|V|$ whose meaning will be explained in the next section.

Input:

- Directed weighed graph $G(V, E)$
- N – the required number of heaviest edges
- X – the number of trials

Output:

N non-overlapping edges with maximum sum weight (two edges are overlapping if they share a vertex)

Algorithm:

```
1. Repeat $X$ times {
2.     Randomly assign each vertex to either $V_1$ or $V_2$ with equal probability
3.     For each vertex $v$ in $V$, compute two values:
4.         $W_1[v] = \sum_{u \in V_1} weight(uv)$
5.         $W_2[v] = \sum_{u \in V_2} weight(vu)$
6.     Put all vertices of $V$ into a queue $Q$
7.     While $Q$ is not empty {
8.         Pop $v$ from $Q$
9.         Compare $W_1[v]$ and $W_2[v]$ to decide whether switching $v$ to the other
               group will increase the $\{V_1, V_2\}$ cut.
10.        If switching v to the other group will increase the cut weight {
11.            Switch v to the other group
12.            For each vertex $u \neq v$, update $W_1[u]$ and $W_2[u]$ by adding or
               subtracting a weight of edge ‘$uv$’ or ‘$vu$’
13.            For each vertex $u \neq v$ that is not in $Q$, compare $W_1[u]$ and $W_2[u]$
                   to decide whether switching $u$ to the other group will in-
crease the
```

```
                  {V1, V2} cut. If yes, push 'u' to Q
14.         }
15.     }
16.     Update the best solution – N most heaviest edges from the maximum cut
17. }
18. Output N edges of the cut with the maximum weight
```

3.1.6 Data Structures for the Maximum-Weight Cut Algorithm

Manber's paper does not specify how to check whether switching a vertex to the other group will increase the cut, but this is the most frequent operation (step 13 of the algorithm). To do it in constant time, we used two arrays, W_1 and W_2, of size $|V|$. $W_1[v]$ specifies the total weight of all edges from the first group into v: $W_1[v]=\sum_{u\in V_1} weight(uv)$. $W_2[v]$ specifies the total weight of all edges from v to the second group: $W_2[v]=\sum_{u\in V_2} weight(vu)$. The vertex *'v'* may belong to either of the two groups, it is not important for the computation of W_1 and W_2.

$W_1[v]$ and $W_2[v]$ are updated each time a vertex v switches a group (step 12 of the algorithm). To do so, we add or subtract the weight of an edge *'uv'* or *'vu'* to/from $W_1[u]$ and $W_2[u]$ for each vertex $u\neq v$.

3.2 LZSS Compression Implementation

This section describes the details of LZSS algorithm ([3]) and its implementation. As described in section 2.3, a text is encoded as a sequence of elements which are either single characters or pointers of kind *(offset, length)* to the previously occurred strings. We will sometimes refer to the previous occurrence of the current string as a '***match***'. Let us also define ***"the best" previous occurrence*** of the current string as the one for which replacing the current string by the pointer saves the maximum space in the output.

3.2.1 Outline of LZSS Algorithm

Input:

- Text, T

- *Maximum allowed text window size*, *W*

Output: LZSS-compressed text

Algorithm:

```
1.  Let p be the current position at the original text, p=1
2. while p < length of T {
3.      Find "the best" previous occurrence of the current string
        (the current string is the substring of T starting at 'p').
        Let (off, len) be the pointer to it from position p.
4.      If such previous occurrence has been found
5.          and replacing the current string by the pointer will save a space
6.      then {
7.          Write the pointer (off, len), probably additionally encoded
8.          p ← p+len
9.          Update the data structure used to find the strings previous occurrences
            with 'len' strings starting at
            T[p], T[p+1], …, and T[p+len-1].
10.     } otherwise {
11.         Write the original character T[p], probably additionally encoded
12.         p ← p+1
13.         Update the data structure used to find strings previous occurrences
            with a string starting at T[p].
14.     }
15. }
```

Note that the *maximum allowed text window size W,* was listed as input to the algorithm rather than a regular user parameter, since it affects not only the quality and the speed of the compression process itself, but also a time required to de-compress the output of the compression algorithm. Note also that the maximum allowed text window size is actually the maximum allowed value for *'offset'* that can be used in *(offset, length)* pair.

3.2.2 Implementation Choices

The algorithm has been implemented using several different ways. User parameters control which implementation is actually run.

First, there are two main methods to encode the characters and pointers: ***the byte-format*** implementation and ***the bit-format*** one. The *byte-format* method always uses an integer number of bytes to encode a single character or pointer, i.e. a pointer can be encoded to span 1, 2, 3 or 5 bytes, but cannot span 17 bits. In contrast, the *bit-format* implementation has no such restriction. The *bit-format* implementation yields a better compression ratio than the *byte-format* one, but it takes more time to search inside or decompress the text compressed using the *bit-format* method. The implementations differ in steps *7* and *11* of the algorithm. They also differ in defining the meaning of the “best” match among several ones, in step *3* of the algorithm.

Second, three ways to store and find strings previous occurrences were implemented. This corresponds to steps *3*, *9* and *13* of the algorithm.

The user can combine either *byte-format* or *bit-format* implementation with any method of finding strings previous occurrences, independently.

3.2.3 Finding a Previous Occurrence of the Current String

Let us define ***the current string*** as the string starting at the current position. This section describes the data structures used to store and find the strings previous occurrences.

Let us see an example of string's previous occurrences. Let $T=$ $a_1 b_2 c_3 d_4 a_5 b_6 a_7 b_8 c_9 d_{10} e_{11}$ and $p=7$. The longest previous occurrence of the current string starts at position *1* and has a length of *4* characters (*'abcd'*). Another previous occurrence starts at position *5* and has a length of *2* characters (*'ab'*).

Which occurrence to prefer is a matter of a specific implementation. In general, the best occurrence maximizes the difference in the number of bits that would take to write the occurrence as is and the number of bits that would take to write the *(offset, length)* pair. For the simplest case when no additional encoding is applied to characters and pointers, the best occurrence is the longest occurrence. When an additional encoding is applied to characters and pointers, the most common heuristics is to find the most recent occurrence among the longest ones. The criteria for choosing one pointer over the other depends on whether the `byte-format` or the `bit-format` implementation is used. The criteria are described later, in sections 3.2.4 and 3.2.5.

Finding a previous occurrence of a current string is usually the bottleneck of LZSS compression algorithm. Finding the really best match might be too expensive, so we look for a match that will be as close to the best as possible. Three techniques were implemented to search for the previous occurrence of a current string. The first one uses a hash table, the second one uses suffix trees ([12], [13]), and the third one combines the two previous techniques to yield the better result.

3.2.3.1 Finding a Previous Occurrence Using a Hash Table

This technique uses a hash table to find previous occurrences of a current string. The hash index is computed for the first *'k'* bytes of each string, where *'k'* is a user parameter (typically *3* or *4*). Strings are stored as a position in the original text, so storing each string takes a constant amount of storage. The hash buckets are circular buffers of a fixed size, which is also a user parameter. When a bucket is full and a new string is going to be inserted, the algorithm removes the oldest string from this bucket. The hash value is computed by the following formula:

$$hash(a_1 a_2 \ldots a_k) = 31 * hash(a_1 \ldots a_{k-1}) + a_k, \quad hash(a_1) = a_1$$

which is similar to how the String class hash function is implemented in Java. (In contrast to Java, each a_i here represents a byte, while Java strings consist of 2-byte characters). To make the run time independent of the value of *'k'*, the algorithm computes the hash value for the next string from the previous hash value.

$$hash_{new}(a_p \ldots a_{p+k-1}) = 31 * (hash_{old} - 31^{(k-1)} * a_{p-1}) + a_{p+k-1}$$

Each bucket stores the starting position of a string whose prefix matches the prefix of $a_1 \ldots a_k$. If a matching bucket is found, the algorithm explores the bucket to find the “best” match among all the strings stored in the bucket.

For example, let $T = a_1 b_2 c_3 a_4 b_5 c_6 d_7$, and $k=3$. If the allowed bucket size is greater than *1*, then the bucket corresponding to the key `hash('abc')` will contain two strings: the first starting at position *1* of *T* and the second starting at position *4* of *T*. Otherwise, the string starting at position *1* of *T* will be removed before the insertion of the string starting a position *4*.

The problem with this technique is that it does not always find the longest match since the maximum allowed bucket size is usually small (1-10 entries). Increasing this number increases the chances to find longer matches, but also increases the running time, which is proportional to the maximum allowed size of the buckets. The advantage of this technique is that it always finds the most recent matches. This allows to keep the size of `'offset'` small, which may result in smaller space taken by `(offset, length)` pair.

3.2.3.2 Finding a Previous Occurrence Using Suffix Trees

This technique uses two suffix trees ([12]) to find the longest previous occurrence of a given string.

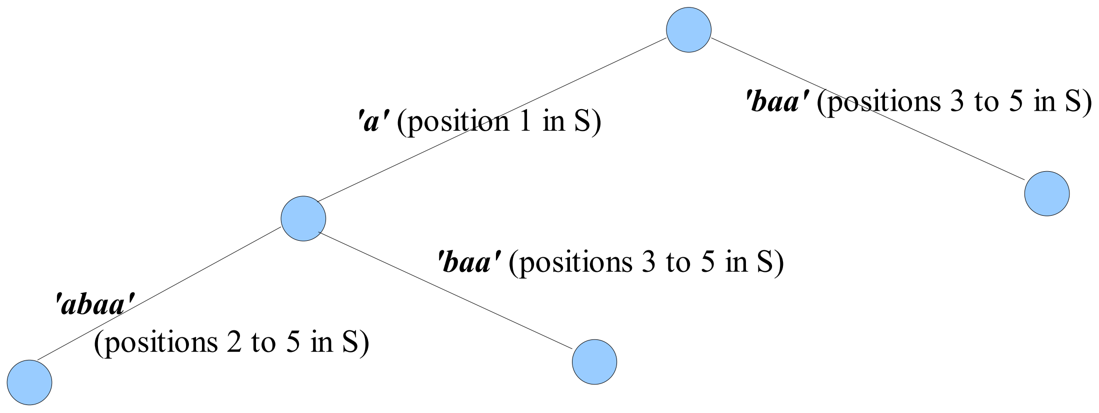
A suffix tree is a data structure that presents the suffixes of a given string T in a way that allows fast search for any substring of T. If such a substring does not exist in the tree, the tree allows to find the longest prefix of the substring that is present in T. The search time is `O(match length)`, which theoretically makes it the fastest possible data structure to find the longest prefix of a given pattern in a text T. The edges of a suffix tree are labeled with strings, such that each suffix of T has one-to-one correspondence to a path from the tree's root to a leaf. Another important property of a suffix tree is that any internal node has at least two children. To satisfy this property, T is padded with a terminal symbol not seen in T. This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of T.

There is a linear-time online-construction algorithm of Ukkonen ([14]). It builds the suffix tree by inserting character after character, from left to right, so inserting a single suffix takes a constant time on average. Mark Nelson ([13]) gives a very good description of suffix trees and the Ukkonen algorithm ([14]).

At a first glance, a single suffix tree can provide the functionality of storing and finding the string's previous occurrences, as required by the LZSS algorithm. It can find the longest previous occurrence of a given string in time linear to the length of the match.

However, a suffix tree has two drawbacks: first, it always finds the oldest possible occurrence of a string, and second, there is no effective mechanism to remove old nodes from a suffix tree. This poses three problems: first, the occurrence it finds may violate the *'maximum text window size'* requirement (i.e. the maximum allowed value for *'offset'*), even if a valid newer occurrence exists. Second, older occurrences have larger *offset* values than the newer ones, which results in less effective encoding, since larger values require more space than smaller ones. Third, a suffix tree of the entire input text requires too much memory.

Let us see why a suffix tree always finds older occurrences by an example. Let *T=* $a_1 a_2 b_3 a_4 a_5 b_6 X_7 X_8 X_9 a_{10} a_{11}$. After the fifth iteration the tree will look as follows:


'a' (position 1 in S)
'baa' (positions 3 to 5 in S)
'abaa' (positions 2 to 5 in S)
'baa' (positions 3 to 5 in S)

It is easy to see that a search for *'aa'* in such a tree yields *T[1,2]*, and not *T[4,5]*. The problem is that there is no explicit leaf on which to store the newer position of *'aa'*.

Artificially creating an explicit node to store newer position would violate the property of a suffix tree that each internal node should have at least two children and empty edges are not allowed.

Removing too old occurrences would solve the first and the third problem. Larsson ([15]) describes an algorithm that allows removing old nodes, but the algorithm is not easy to implement. Instead, our implementation “imitates” a suffix tree with node removal by using two overlapping suffix trees, where each suffix tree is constructed using the Ukkonen's algorithm ([14]). The size of a string represented by a tree cannot exceed the maximum text window size *W*. When the first tree is half-full (i.e. of size *W/2*), the second tree is created and starts growing. When the first tree becomes full, it is cleared, and the second tree becomes the “active” tree used for searches etc.

For example, let us see the behavior of the trees for *W=32K*:

- On `0-16K`: insert values into tree#1, use tree#1 for search
- On `16K-32K`: insert values into both trees, use tree#1 for search
- On `32K`: clear tree#1, start using tree#2 for search
- On `32-48K`: insert values into both trees, use tree#2 for search
- On `48K`: clear tree#2, start using tree#1 for search
- On `48-64K`: insert values into both trees, use tree#1 for search
- etc.

Clearly, this way the longest match within the window size can be missed, since after the trees are just switched, the tree used for searching is only half-full. Still, the technique seems to find longer matches than a hash-based one. Let us repeat that the described method does not solve the second problem, that is it still tends to find older occurrences, requiring larger `offset` values and thus taking more space to encode.

Another trick used in our implementation of a suffix tree deals with potentially large number of a node's children. Since there are 256 different bytes, a node can have as many as 256 children. To keep the search fast we could allocate an array of size 256 in every internal node, but it may require too much memory. We used the technique of Mark Nelson ([13]), which is to store all children in a single global hash table. The hash key is computed from the first byte of the edge and the parent node's number.

To summarize, the suffix trees technique was able to find long, but old matches. Due to the trick of using two trees all found matches are legal. However, preferring old matches may result in a less effective compression, due to larger offsets. Suffix trees may be changed to find more recent matches, but this has not been done in our implementation.

3.2.3.3 Finding a Previous Occurrence Using both Hash Table and Suffix Trees

This technique strives to combine the benefits of both approaches. The technique that uses two suffix trees tends to find matches that are almost as long as possible, but which might be located far from the current position. In contrast, hash tables tend to find matches located as close as possible to the current position, but which might be not very long. The technique that uses both methods compares the match found by the suffix tree method with the one found by the hash table and then chooses the “best” match. The meaning of the “best” depends on a specific implementation (either byte-format LZSS, bit-format LZSS or byte-format Shapira/Klein), see sections 3.2.4.2, 3.2.5.4, and 3.3.4 for details.

3.2.4 Byte-Format Implementation of LZSS

This section describes the byte-format method to additionally encode characters and pointers (steps *7* and *11* of the algorithm). In this method each encoded character or pointer should take an integer number of bytes.

The encoding is as follows. Characters are not encoded, i.e. they are written as is. *(Offset, length)* pairs are encoded using a number of predefined "***encoding rules***". Each encoded pair takes an integer number of bytes. The details will be described in section 3.2.4.1. To differentiate between regular characters and pointers, each 8 elements (characters or pointers) are prefixed with a special byte, as suggested by ([12). Each bit of this byte corresponds to one of the 8 elements following the byte. Bit *0* means that the element is a regular character, while bit *1* means that the element is a pointer. For example, the byte 00000110 of value *6* will prefix the following sequence of LZSS elements: $abcde(4,3)(6,4)k$. Similar to ([10]), each character is written as is, always spanning a single byte. But pointers are written differently: in ([10]), *offsets* always span 12 bits (thus bounding the text window size by 2^{12}) and *lengths* always span 4 bits, thus yielding 2 bytes for a pointer. In contrast, in our implementation the text window size is a user parameter which may be as large as 2^{32} , and the pointers may span from 1 to 7 bytes, as described in the next section.

3.2.4.1 Bits Allocation for *(offset, length)* Pairs

(Offset, length) pointers are encoded using a number of predefined "encoding rules". Each rule specifies the exact amount of bits allocated for *offset*, and the exact amount of bits allocated for *length*. Each rule is prefixed with a rule identifier, indicating the specific rule. The total number of bits, used for rule identifier, *offset* and *length* always sums to a multiple of 8, thus forming an integer number of bytes. Given a specific *(offset, length)* pair, the algorithm looks for the shortest matching rule that could handle this pair. Note that all rules are fixed and known before the compression process.

For example, let *(500, 4)* be a specific *(offset, length)* pair, and suppose there are two "encoding rules". The first rule allocates 2 bits for the rule identifier, 4 bits for `offset` and 2 bits for `length`, the second rule allocates 3 bits for the rule identifier, 10 bits for `offset` and 3 bits for `length`. Using the first rule allows to encode a pointer by a single byte, using the second rule allows encoding a pointer by two bytes. Note that rules identifiers may span a different number of bits in different rules: we may wish to encourage the use of shorter rules giving them shorter identifiers and thus more space to store the actual data. Since the maximum `offset` value that can be stored using the first rule is $2^4=16$ (offset '0' is illegal), it cannot store our pointer. The maximum `offset` value that can be stored using the second rule is $2^{10}=1024$, so it will be used.

Note that since all rules exist prior to compression, some of them might be irrelevant for the given maximum text window size *W*. Continuing the example, if *W=1024*, and a third rule allocates 18 bits for `offset` and 3 bits for `length`, it would be irrelevant, since no `offset` can be larger than *W*, so every legal pointer matching the third rule would also match the second one. But if the third rule allocates more than 3 bits for `lengths`, it still can be relevant, to handle pointers with small `offsets`, but large `lengths`.

If pointer's `length` value is too large for a certain rule, there are choices: to cut the `length` to match a smaller rule or to use a rule with larger length. To make the rules more efficient, each rule also specifies the minimum value of `'length'` of `(offset, length)` pair for which it can be applied. This allows to use its bits allocated for `length` more efficiently. Referring to the example above, the first rule can specify that its minimum `length` is 3, so that the maximum `length` it can store using its 2 bits allocated for `length` is 6 (that is: 3, 4, 5, 6) and not 4. (The same could be done for offsets, but it was not implemented, for simplicity).

After running some experiments, the encoding for the rule identifiers was chosen as follows. Rules are sorted from shortest to longest, and each rule gets its sequence number, which is also its identifier, counting from 0. This number is written in unary format, i.e. the identifier of the first rule is '0', the identifier of the second one is '10', the identifier of the third one is '110'. At a first glance, this unary encoding may seem to be very inefficient, still the experiments show that it beats binary encoding, for example, in which all identifiers span a fixed number of bits, e.g. 3. The reason is that the unary encoding of identifiers maximizes the usage of short rules over the longer ones. For example, it requires only 1 bit for the first, single-byte rule, leaving 7 bits to store a pointer. In contrast, a binary encoding with a fixed number of bits, say 3, would leave only 5 bits to store a pointer in a single byte. This means that many pointers that can be stored in a single byte using the unary approach will be stored in two bytes with the binary approach.

Each rule encodes a pointer to a different number of bytes, i.e. a pointer encoded by the first rule spans a single byte, a pointer encoded by the second rule spans two bytes etc. There are 7 rules total, so a pointer can span at most 7 bytes.

One could argue that this is not purely a byte-format encoding. But with a trick described in section 3.2.4.3, identification of an encoding rule takes exactly one array access operation, regardless of the number of bits allocated to a rule identifier. Once the "encoding rule" is known, the `offset` and `length` may be obtained by one or two bit shift operations, which is also very fast. In contrast, decoding a binary Huffman-encoded character requires a move on a Huffman tree for every bit of the input. The trick described in section 3.2.4.3 can also be used to decode binary Huffman-encoded symbols, but it either cannot guarantee a single array access operation for every symbol or can require too large array, whose preparation can also take a time. Note also that the encoded text is aligned on bytes, in contrast to e.g. Huffman encoding.

3.2.4.2 Assessing Possible Replacements of the Current String

Knowing in advance all possible encodings for `(offset, length)` pair has an advantage that we can precisely measure how much space will be saved by replacing the current string by the pointer. This means that the "best" match for this method is not always the longest match.

For example, let p_1 `=(10000, 5)` and p_2 `=(6, 4)` be two possible `(offset, length)` pairs, and suppose there are only three encoding rules: the first rule allocates *4* bits for `offset` and *2* bits for `length`, the second rule allocates *10* bits for `offset` and *3* bits for `length`, the third rule allocates *17* bits for `offset` and *4* bits for `length`. The pointer p_1 can only be encoded using the third rule thus taking 3 bytes and replacing 5 bytes, so the saving is $5-3=2$ bytes. The pointer p_2 can be replaced by the first rule thus taking 1 byte and replacing 4 bytes, so this saves $4-1=3$ bytes.

In this case using a shorter match is better than using a longer one.

Let `p = (offset, length)` be the pointer to some previous occurrence of the current string, let R be the shortest encoding rule that can handle p, and let $|R|$ be the number of bytes that R spans. The saving from the replacement of the current string by p is: $length-|R|$. Given a previous occurrence of the current string, the algorithm first finds the shortest encoding rule that can store the pointer to this occurrence and then computes the saving using the formula above. As shown in section 3.2.3, the algorithm might have to choose from several previous occurrences. To do so, the algorithm computes the saving for each previous occurrence and then chooses the one offering the best saving. If no replacement offers a positive saving, the algorithm chooses not to replace the current string.

Experiments show that choosing the occurrence offering the best saving instead of just using the longest previous occurrence improves the compression ratio by **3%** on average. The downside of this technique is a larger processing time, since the algorithm has to find the shortest encoding rule for each possible replacement.

3.2.4.3 Fast Encoding Rule Identification

This section describes a trick used to identify an “encoding rule” using a single array access operation. As a reminder, the encoding for rule identifier is Unary codes, i.e. consists of 0 to 6 non-zero bits followed by a zero bit. A naïve approach would be to count the number of non-zero bits, which would mean performing from 1 to 7 bit-format operations. The following trick allows avoiding this.

The encoding rules are fixed and known at the initialization time, prior to any file to decompress or search. Hence, we can prepare an array of size $2^8=256$ which maps a first byte of a pointer to an encoding rule, also at the initialization time. Of course, there is a redundancy, since 256 possible values are mapped into 7 encoding rules. For example, all bytes whose value belongs to [0, 127] will be mapped to the first “encoding rule”. At the time of decompression, the algorithm just reads the first byte of a pointer and immediately obtains the corresponding encoding rule. This idea belongs to ([4]), which used the idea for faster decompression of Huffman-encoded symbols.

3.2.5 Bit-Format Implementation of LZSS

This section describes the `bit-format` method to additionally encode characters and pointers (steps *7* and *11* of the algorithm). The method uses two binary Huffman trees ([8]): the first one is used to encode characters and match lengths, and the second one encodes match offsets. This is similar to *DEFLATE* algorithm ([16]) used in `'gzip'` ([4], [5]). Both trees are written to the output prior to the compressed text itself. Section 3.2.5.3 describes how the trees are serialized.

3.2.5.1 Algorithm Outline

The general algorithm is as follows:

1. Compute binary Huffman encoding of all characters of the input text to estimate an average size of a Huffman-encoded character, using the characters frequencies in the input text. This value is used to decide whether to replace the current string by a pointer at step 4 of the LZSS algorithm (section 3.2.1). The details of using this value are provided in section 3.2.5.4.
2. Run the basic LZSS algorithm, without additional encoding and store its output in a list *L*.
3. Compute the frequencies of all characters, `lengths` and `offsets`.
4. For rare `lengths` and `offsets` compute the frequency of the required number of bytes to store each value. See section 3.2.5.2 for details.
5. Use the obtained frequencies to compute two binary Huffman trees, the first tree for literals and `lengths` and the second one for `offsets`.
6. Write the representations of the Huffman trees to the output and then write Huffman encoding of the LZSS elements from *L* into the output.

3.2.5.2 Dealing with Rare Symbols

Huffman encoding of elements that appear only once in the input is counter productive: not only they take many bits to encode, but they also increase the size of a Huffman tree, which should be also serialized into the output. Usually, such elements take more than twice space in their compressed form than in their original form, due to necessity to write the Huffman trees themselves. This may be tolerable for characters since there may be no more than 256 different symbols spanning one byte. However, the number of possible offsets may be very large.

To deal with such cases, rare lengths and offsets are not inserted into their respective Huffman trees. Let us define the `byte-span` of a value N to be the number of bytes required to store it, which may be computed using the formula: $\log_2(N+1)/8$. Now, instead of storing rare lengths and offsets, only their `byte-spans` are inserted into their Huffman trees. In the compressed output they appear as follows:

Huffman-encoded element's `byte-span`	The original element (length or offset)

So the alphabet for the first Huffman tree consists of (1) all characters that appear in the input text, (2) frequent `lengths`, and (3) `byte-spans` of rare `lengths` (usually, it will be a single "symbol" denoting one byte). The alphabet for the second Huffman tree consists of (1) frequent `offsets` (2) `byte-spans` of rare `offsets` (usually, there will be 2-3 such special "symbols" for `offsets` spanning one, two and three bytes).

For example, assume that 6 different offsets appeared in the output of the basic LZSS algorithm: `o1=1` (3 times), `o2=12` (3 times), and four offsets appeared only once: `o3=3, o4=4, o5=5, o6=6`. The alphabet for the Huffman tree consists of 3 elements:

- `1` with frequency 3
- `12` with frequency 3
- `Offset byte-span of 1 byte` with frequency 4 (corresponds for offsets `o3, o4, o5` and `o6`)

This alphabet will be encoded as follows:

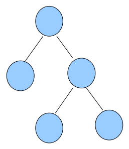
- Offset `1` as `'10'`
- Offset `12` as `'11'`
- O`ffset byte-span of 1 byte` as `'0'`

Using this alphabet, the offset `1` will be encoded as `'10'`, while offset `3` will be encoded as 000000011 that is, a single bit `'0'` followed by a single byte (8 bits) representing the value of `3`.

3.2.5.3 Writing a Huffman Tree

The two binary Huffman trees are written to the output prior to the compressed text itself. This section describes how a Huffman tree is serialized.

First, the algorithm outputs the structure of the Huffman tree. It uses the fact that every internal node in a Huffman tree can have exactly two sons. The algorithm traverses the tree in BFS order, from left to right, and for each node outputs a bit telling whether the node is an internal node or a leaf. If the node is an internal node, it outputs bit *1*, otherwise it outputs bit *0*. For example, the output for the tree below is *'10100'*.



Second, the algorithm outputs the original values that the tree encodes. Each leaf encodes one original value. Their order matches the walk on tree's leaves from left to right. In general, the values may be of three types: (1) text literals (2) integers such as lengths or offsets (3) *byte-spans* of rare integer values in bytes, as described in the previous section. For each value the algorithm first outputs its type id: *'00'* for text literals, *'1'* for lengths or offsets, *'01'* for *byte-spans*. (The tree of offsets has only two types of values: offsets and their *byte-spans*, so a single bit would be sufficient to encode a type id of a *byte-span*. For simplicity, it was not done, since there can be at most 4 different *byte-spans*, for 1, 2, 3 and 4 bytes respectively). Then it outputs the actual value. Text literals and byte-spans always require one byte, so it just writes the byte. Arbitrary integers such as lengths or offsets may span more than one byte. For such an integer, the algorithm first outputs 2 bits telling the number of bytes the value can span (1, 2, 3, or 4 bytes) and then writes the actual value.

3.2.5.4 Assessing Possible Replacements of the Current String

The final encoding of text literals and *(off, len)* pairs is not known at the time of running the basic LZSS algorithm, since their final encoding is based on their frequencies. Suppose, for example, that the algorithm has found that a string *T="abcde"* could be replaced by a pointer *P=(60, 5)*. The algorithm has now to decide whether to replace the string *T* by the pointer *P*. As a reminder, all literals and pointers will be Huffman-encoded after the basic LZSS algorithm completes. It might happen that the Huffman-encoding of *T* would span e.g. 40 bits, while the Huffman-encoding of *P* would span only 12 bits, in which case it is worth to replace *T* by *P*. But it could also happen that the Huffman-encoding of *T* would span 15 bits, while the Huffman-encoding of *P* would span 25 bits, in which case it is not worth to replace *T* by *P*. Hence, the algorithm needs to use heuristics to estimate the final length of a string or a pointer. It then uses this estimation to choose the best replacement and to decide whether to use a replacement at all. The heuristics used to estimate the final length of a string or a pointer is as follows.

Let *C* be the average size of a Huffman-encoded character, computed at step 1 of the bit-format algorithm (see section 3.2.5.1). A final length of a string of *len* literals is estimated to be $len * C$ bits. (Of course, we could achieve a better estimate by just encoding the string with the Huffman tree computed at step 1, but this would require much more processing time). Suppose now that this string may be replaced by *(off, len)* pointer. The final length of *'off'* value is estimated to be $2+7*(byte\ span\ of\ 'off')$ bits. The reason to use the multiplier *7* instead of *8* stems from the assumption that the Huffman-encoded offsets will on average span slightly less than without any encoding, and the summand *2* is added to make the algorithm a bit more reluctant to use replacements over the original literals. The final length of *'len'* is estimated to be $2+C*(byte\ span\ of\ 'len')$. The reason of using *C* as a multiplier is that *'len'* will be encoded with the same Huffman tree as the literals. The reason of using the byte-span of *'len'* as a multiplier is that the original value of *'len'* will have to be written once in the serialization of its Huffman tree. Again, the summand *2* is added to make the algorithm a bit more reluctant to use replacements.

Once the final length of a string to be replaced and the final length of an *(off, len)* pair are estimated, it is easy to compare two *(off, len)* pointers and to decide whether to use a pointer at all. A pointer offering larger savings in bits is preferred on a pointer offering smaller savings. The original string can be replaced by a pointer if the savings offered by this pointer is positive, i.e. is estimated to span less bits than the original string.

3.3 Shapira/Klein's Compression Implementation

This section describes the implementation details of Shapira/Klein's compression algorithm. As described in section 2.6, a text is encoded as a sequence of elements which are either text literals or pointers of the kind *(offset, length, slide)* to the previously occurred strings. The encoding is similar to LZSS, but differs from it by the fact that a pointer is written just after the string it refers to instead of being written at the position of the string it replaces. The implementation uses the basic LZSS algorithm ([3]) as a subroutine. The additional encoding used for literals and pointers is similar to the byte-format implementation of LZSS. That is, literals are written as is, and *(offset, length, slide)* pointers are encoded using a number of predefined *"encoding rules"*. Each encoded pointer takes an integer number of bytes. The details will be described in section 3.3.2. Similar to the byte-format implementation of LZSS, each 8 elements are prefixed with a special byte telling whether the corresponding element is a literal or a pointer.

3.3.1 Outline of Shapira/Klein's Compression Algorithm

1. Run the basic LZSS algorithm, without additional encoding, and store its output in a special data structure *L*, used to convert LZSS pointers to Shapira/Klein pointers. Converting LZSS pointers to Shapira/Klein pointers is described in section 3.3.3. Criteria used to choose a particular pointer in steps *3* and *4* of the basic LZSS algorithm are explained in section 3.3.4.
2. Obtain a list of literals and Shapira/Klein pointers from *L*.
3. Write each element to the output, encoding each pointer using the shortest matching *"encoding rule"* (see section 3.3.2 for details).

3.3.2 Bit Allocation for *(offset, length, slide)* Pointers

The encoding is similar to the encoding of *(offset, length)* pointers used in the byte-format LZSS compression, described in details in section 3.2.4.1. That is, *(offset, length, slide)* pointers are encoded using a number of predefined "encoding rules". Each rule specifies the exact amount of bits allocated for *offset*, the exact amount of bits allocated for *length*, and the exact amount of bits allocated for *slide*. Each rule is prefixed with a rule identifier, telling which specific rule is used. The total number of bits, used for the rule identifier, *offset*, *length* and *slide* always sums to a multiple of 8, thus forming an integer number of bytes. Given a specific pointer, the algorithm looks for the shortest matching rule that could handle it.

The number of rules is 16. The number of bits allocated to the rule identifier is constant and equals to 4. This is in contrast to the byte-format LZSS where a rule identifier spans a different number of bits using the unary encoding for a rule identifier. The reason to use a fixed number of bits for a rule identifier is that the unary coding becomes much less attractive with the increasing of the number of rules from 7 to 16. Also, with the addition of the third parameter, `slide`, there going to be very few pointers which can be encoded to a single byte (the main reason behind using the unary encoding in the byte-format LZSS was to maximize the usage of rules spanning one and two bytes). Yet it introduces some inconsistency between the implementations. Experiments show that the unary encoding for the byte-format LZSS identifiers improves the compression ratio by 2-3% on average over the use of identifiers with a fixed number of bits.

There are several rules encoding a pointer to the same number of bytes, but allocating a different number of bits to an `offset,`a `length` and a `slide`. For instance, one rule may allocate 13 bits for an `offset`, 3 bits for a `length`, and only 4 bits for a `slide` (i.e. the rule spans 24 bits; the remaining 4 bits are used for the rule identifier), to be able to handle pointers with large `offsets`, but with small or zero `slides`. Another rule allocates 8 bits both for an `offset` and a `slide`, and 4 bits for a `length` (again, 4 bits are used as an identifier to sum up to 24 bits). A `slide` cannot be larger than an `offset`, so the number of bits allocated to an `offset` is at least as the number of bits allocated to a `slide`.

3.3.3 Converting Shapira/Klein Pointers from Regular LZSS Pointers

In Shapira/Klein compression, pointers are inserted just after a string they refer to. The number of pointers that will be inserted after a given string is not known in advance. Therefore, the Shapira/Klein's compression algorithm first stores all literals and LZSS pointers in a special structure `L` (step `1`) and only then traverses `L` and computes the list of literals and Shapira/Klein pointers (step `2`).

3.3.3.1 Storing Literals and LZSS Pointers

At the first step, the algorithm just moves LZSS pointers to the new positions, that is, just after the strings they refer to: `pointer's new position`= $(\mathit{LZSS\ pointer\ position})-(\mathit{LZSS\ offset})+\mathit{length}-1$. The literals stay in their original positions. The structure `L` stores a map from every new position to the list of LZSS elements (literals and pointers) located at this position. The offset is computed by the formula: $\mathit{new\ offset}=(\mathit{LZSS\ offset})-\mathit{length}$.

For example, let T= $a_1 b_2 c_3 d_4 e_5 a_6 b_7 c_8 d_9 d_{10} e_{11} a_{12} k_{13}$. LZSS encoding of this string is $abcde(5,4)(6,3)k$. *L* will contain the following map:

- *1 → a*
- *2 → b*
- *3 → c*
- *4 → d, (1,4)*
- *5 → e*
- *6 → (3, 3)*
- *13 → k*

Similar to the LZSS implementations, the *length* cannot be smaller than a user-specified parameter, *3* by default.

To avoid dealing with negative values of Shapira/Klein *offsets*, if LZSS pointer's *length* exceeds LZSS pointer's *offset*, the *length* is reduced to the value of the *offset*. For example, let T= $a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}$ and let the minimal allowed *length* to be *3* bytes. The optimal LZSS encoding of this string would be $a(1,11)$. But since the *length* value cannot exceed the *offset* value, the actual LZSS encoding will be $aaa(3,3)(6,6)$.

3.3.3.2 Computing a New List of Literals and Pointers

At the second step, the algorithm computes the actual sequence of Shapira/Klein elements. First, if all elements of a list are pointers, without a literal, they are moved to a closest preceding position that corresponds to a literal, and a *slide* is inserted to each such pointer. The *slide* is computed by a formula:

$slide=(pointer\ position\ before\ move)-(pointer\ position\ after\ move)$.

In the example above (with T= $a_1 b_2 c_3 d_4 e_5 a_6 b_7 c_8 d_9 d_{10} e_{11} a_{12} k_{13}$), position *4* maps to a list containing a literal *'d'* and a pointer *(1, 4)*, so it will not be changed. But position *6* maps to a list consisting of a single pointer *(3,3)*. Since this list contains no literals, *(3, 3)* should be appended to a list mapped by *5*. The position of pointer *(3,3)* in *L* before the move was *6*, its new position became *5*, so the *slide = 6 - 5 = 1*. After all moves, *L* contains the following map:

- *1 → a*

- *2 → b*
- *3 → c*
- *4 → d, (1,4)*
- *5 → e, (3, 3, 1)*
- *13 → k*

Second, the map is traversed from the smallest position to the largest, and all elements mapped by those positions are inserted to the output list.

In the example above, the output list is: $\{a, b, c, d, (1,4), e, (3,3,1), k\}$.

3.3.4 Assessing Possible Replacement of the Current String

The actual *(offset, length, slide)* triples are not known at the time of the run of the basic LZSS algorithm. Therefore, precise evaluation of savings in space offered by a particular pointer is not possible.

The algorithm compares two encodings of the matches and prefers longer matches over shorter ones. For matches with the same length, it prefers the ones with smaller offsets.

If the match length is equal or larger than the ***minimum length*** then the match will be accepted, i.e. the algorithm will choose to replace an original string by a pointer. Otherwise, the original string will be kept. In our implementation the minimum length is ***3***. This number is computed from the predefined encoding rules: the smallest number of bytes that a triple *(offset, length, slide)* can span is *2*, so a replacement has a chance to save space only if the length of the replaced string is greater than *2*, i.e. if it is *3*.

3.4 Shapira/Klein's Pattern Matching Implementation

This section provides implementation details of Shapira/Klein's pattern matching algorithm, described in section 2.7.

The algorithm traverses the text from left to right. To be able to perform partial decompression, the processed text is stored in a special circular buffer, ***CB***.

Note that we treat regular *(offset, length)* pointers just as *(offset, length, slide)* pointers with *slide*=0.

3.4.1 Dealing With Literals

Characters are copied to *CB* and are also fed into a standard underlying pattern matching algorithm, which is the KMP pattern matching algorithm ([6]) in our case. Blocks of relevant characters (i.e. successive characters present in the searched pattern) are additionally linked. That is, the last character of a block points on the first character of the following block. To link to the next block, a character just stores a distance to the next block. The circular buffer also stores the position of the `last processed literal`.

For example, let the searched pattern be *S="dkAB"* and the compressed text *C=* $\boldsymbol{d}_1 a_2 b_3 c_4 \boldsymbol{d}_5 \boldsymbol{d}_6 e_7 f_8 \boldsymbol{k}_9 \ldots$ (*C* would of course contain pointers, but let us assume that they appear after these literals, so they are not shown here for brevity). The blocks of the relevant characters are first *"d"*, then *"dd"* and then *"k"*. After reading these characters, *CB* will contain the following: $d^{+4}\$\$\$d\,d^{+3}\$\$\hat{\boldsymbol{k}}$. Here *'$'* has been used to denote an irrelevant character. (Note that *CB* is implemented using an array, so even though there is a linked list of blocks of relevant characters, there are array entries between these blocks. Those entries have been denoted by *'$'*). Superscript *'+4'* of d_1 designates a pointer to the next relevant character (which is d_5) and means that the distance from d_1 to the next relevant character is *4*. The position of the `last processed literal` is *9*. We use ^ to denote the last processed literal, which is the *'k'* in the last example.

3.4.2 Dealing With `(offset, length slide)` Pointers

Since each `(offset, length, slide)` pointer appears immediately after the string it refers to, the algorithm can use the linked list of relevant blocks prepared when copying the literals. That is, it copies only the relevant characters and strives to skip the irrelevant ones. No character is fed into the underlying pattern matching algorithm (KMP) at this stage. The position of the `last processed literal` is also not changed.

Let us continue with the previous example. Suppose that the next element of C is a pointer *(1, 6, 1)*, so that *C=* $\boldsymbol{d}_1 a_2 b_3 c_4 \boldsymbol{d}_5 \boldsymbol{d}_6 e_7 f_8 \boldsymbol{k}_9 (1,6,1) \ldots$. The triple *(1, 6, 1)* means that the algorithm should copy *6* characters with the total offset *1+1=2* from the current position, i.e. after position *10*. After reading the text, *CB* will contain the following: $d_1^{+4}\$_2\$_3\$_4 d_5 d_6^{+3}\$_7\$_8\widehat{\boldsymbol{k}_9}@_{10}@_{11}\$_{12} d_{13} d_{14}^{+3}@_{15}@_{16}k_{17}$. Here `'@'` has been used to designate an uninitialized buffer entry.

Let us explain the difference between `'@'` and `'$'`. The entry '@ `'` means that **so far** the algorithm spent no time on this entry. Of course, this entry might become initialized at the next step of the algorithm. In contrast, as `'$'` stands for an updated buffer entry holding an irrelevant character, the algorithm spent some time updating this entry (in our implementation, by copying the character). Once updated, this entry will "never" be changed (it may only be change in a completely different context after the capacity of the circular buffer is exhausted). For instance, the algorithm could skip positions `15` and `16` altogether by using the link from one block of relevant characters to the next one. In contrast, it had to update position `12` with an irrelevant character.

3.4.3 Finding the Destination Position for a Literal

Things get more complicated when literals and pointers are intermixed. Characters cannot be just copied to the next position in the buffer, since this position might be used by a character copied as instructed by some previous `(offset, length, slide)` pointer. To find the actual position in which the literal should be copied to, the algorithm looks for the first buffer entry, not occupied by a character that was copied as instructed by some `(offset, length, slide)` pointer. This is the first uninitialized entry in the buffer, which does not lie between two **interlinked** blocks of relevant characters.

To find such an entry, the algorithm looks at the entry just next to the entry of the `last processed literal`. Let `E` be this entry and let `i` be the index of `E` in the circular buffer `CB`. If `E` is not updated, the search is done. Otherwise, the algorithm inspects whether `E` has a pointer to another block of relevant characters. If such a pointer does not exist, the algorithm just moves one step forward and inspects the next entry, at `CB[i+1]`. But if such pointer exists, the algorithm follows the pointer and inspects the entry just next to the one pointed by `E`. This process is repeated until the algorithm encounters an initialized entry. At every move, the algorithm updates the underlying KMP algorithm: it feeds the encountered relevant characters into the KMP algorithm or instructs the KMP to skip a specified number of characters when it moves from one block to the other. Skipping the specified number of characters in the KMP algorithm in O(1) time is described in section 3.4.5. The next section provides a formal description of this algorithm.

3.4.3.1 Finding the Destination Position for a Literal: the Find_Next_Position Algorithm

Algorithm Find_Next_Position

Input:

- Circular Buffer, *CB*
- The index in *CB* of the last processed literal, *j*
- The *KMP* algorithm, willing to receive characters or skip them

Output:

- The entry in *CB* to which the next literal should be written

Algorithm:

```
1. $i \leftarrow j + 1$
2. while $CB[i]$ is updated {
3.     $E \leftarrow CB[i]$
4.     Feed $KMP$ with $E$'s character
5.     Update the list of the found matches if needed
6.     if $E$ has a pointer to the next block of relevant characters, $E.next$ {
7.         Instruct $KMP$ to skip ($E.next - i - 1$) characters
8.         Feed $KMP$ with the character at $CB[E.next]$
9.         $i \leftarrow E.next + 1$
10.    } else {
11.        $i \leftarrow i + 1$
12.    }
13. }
14. return $CB[i]$
```

3.4.3.2 Finding the Destination Position for a Literal: the Performance of the Find_Next_Position Algorithm

Let *U* be the number of update operations performed by the Shapira/Klein search algorithm on the circular buffer *CB*. The "*Find_Next_Position*" algorithm always moves forward, using the **updated** entries, and stops immediately when it encounters an uninitialized entry. Hence the sum contribution of all invocations of the "*Find_Next_Position*" is bounded by *U*.

3.4.3.3 Finding the Destination Position for a Literal: Example

Let us continue with the example that we dealt with in section 3.4.2. Let us add two more literal elements to the compressed text:

$\boldsymbol{d_1} a_2 b_3 c_4 \boldsymbol{d_5} \boldsymbol{d_6} e_7 f_8 \widehat{\boldsymbol{k_9}}(1,6,1) A\,B\,C$. As shown in the previous section, after processing the *(1,6,1)* pointer, *CB* becomes

$\boldsymbol{d_1^{+4}} \$_2 \$_3 \$_4 \boldsymbol{d_5} \boldsymbol{d_6^{+3}} \$_7 \$_8 \widehat{\boldsymbol{k_9}} @_{10} @_{11} \$_{12} \boldsymbol{d_{13}} \boldsymbol{d_{14}^{+3}} @_{15} @_{16} \boldsymbol{k_{17}} @_{18} @_{19} @_{20} \cdots$.

The next element of the compressed text is a literal *'A'*. The *position of the last processed literal* is *9*. The first uninitialized buffer entry that does not lie between the <u>interlinked</u> blocks of relevant characters is at position *10*. (Position *10* does not lie between interlinked blocks, since $\boldsymbol{k_9}$ has no *next* pointer, so that $\boldsymbol{k_9}$ and $\boldsymbol{d_{13}} \boldsymbol{d_{14}^{+3}}$ are not interlinked). The algorithm copies *'A'* to position *10* of *CB* and feeds *'A'* into the KMP algorithm. *CB* becomes

$d_1^{+4} \$_2 \$_3 \$_4 d_5 d_6^{+3} \$_7 \$_8 k_9 \widehat{\boldsymbol{A_{10}}} @_{11} \$_{12} d_{13} d_{14}^{+3} @_{15} @_{16} k_{17} @_{18} @_{19} @_{20} \cdots$.

Similarly, *'B'* is copied into position *11* of *CB* and feeds *'B'* into the KMP algorithm, so *CB* becomes $d_1^{+4} \$_2 \$_3 \$_4 d_5 d_6^{+3} \$_7 \$_8 k_9 A_{10} \widehat{\boldsymbol{B_{11}}} \$_{12} d_{13} d_{14}^{+3} @_{15} @_{16} k_{17} @_{18} @_{19} @_{20} \cdots$.

Now the next element of the compressed text is a literal *'C'*, and *the position of the last literal* is *11*. The algorithm examines *'$\$_{12}$'*, *'$d_{13}$'* and *'$d_{14}$'* and feeds them to the KMP algorithm. Then it moves to *'k_{17}'*, skipping 2 characters, and instructs the KMP algorithm to skip 2 characters. Then *'k_{17}'* is examined and fed to the KMP algorithm and then the needed uninitialized buffer entry is found at position *18*. The algorithm copies *'C'* to this position and feeds it to the KMP algorithm. The buffer becomes

$d_1^{+4} \$_2 \$_3 \$_4 d_5 d_6^{+3} \$_7 \$_8 k_9 A_{10} B_{11} \$_{12} d_{13} d_{14}^{+3} @_{15} @_{16} k_{17} \widehat{\boldsymbol{C_{18}}} @_{19} @_{20} \cdots$.

3.4.4 Detecting Uninitialized Buffer Entries

The algorithm needs to detect whether a buffer entry is initialized or not. Simply using NULL values will not work, since this is a circular buffer, so a non-NULL entry may still be uninitialized. The algorithm uses another method.

Let K be the capacity of a circular buffer. A circular buffer has an interface of a regular buffer with infinite capacity. That is, as the algorithm runs forward, the external indexes grow, while internally they are mapped to indexes within $[0, K)$ using the formula: $internal\ index = (external\ index)\ mod\ K$.

When a character is copied into a buffer entry, the **external** index is also stored in this entry. The entry at external index i is initialized if and only if its stored external index is equal to i.

3.4.5 Using KMP Algorithm to Skip Irrelevant Characters

Usually the KMP algorithm ([6]) is described as being able to read that characters that it wishes to process (e.g. running in a loop on the input text). However, it can be modified to respond to an event of reading a new character. The search object stores the search pattern, the prefix function, the variable t holding the current position in the text, and the last value, q, returned by the prefix function. On receiving a character, this search object executes a single step of the search, i.e. performs exactly the same operations that are performed by a regular KMP algorithm inside a loop on the text characters.

To skip s characters, $s>0$, t is incremented by s and q is assigned to -1. This has exactly the same effect as if a single irrelevant character (the one not present in the search pattern) has been encountered in the text except that the current position in text is incremented by s and not by 1.

3.4.6 Bounding the Capacity of the Circular Buffer

Let us find some upper bound on the required capacity, K, of the circular buffer, CB, used in Shapira/Klein's pattern matching algorithms.

Processing any pointer $(offset,\ length,\ slide)$ requires no more than $offset + 2 * length + slide$ space (the $length$ is doubled since both the source and the destination should be present in the buffer in order to copy the characters). In other words, $K \leq (max\ offset) + 2 * (max\ length) + (max\ slide)$.

Let W be the maximum text window size. Then the following holds:

$max\ offset \leq W\ ,\ max\ slide \leq W\ ,\ max\ length < max\ offset / 2 = W/2$.

The maximum length is usually much smaller than the maximum offset, hence we could safely assume that $max\ length < max\ offset\ /\ 2$.

Hence, $K \leq W + 2 * W/2 + W = 3 * W$.

3.4.7 The Algorithm Outline

Input:

- Shapira/Klein compressed text, *T*
- Searched pattern, *S*
- *Maximum text window size, W*

Output:

- The positions of all matches of *S* in *T*

Algorithm:

```
1. Compute a relevancy array. For each byte it tells whether the character is present in S
or not.
2. Create a circular buffer CB with capacity 3 *W
3. last_char_index ← 0
4. For each element e of T {
5.      If e is a character {
6.            E ← Find_Next_Position (see section 3.4.3)
7.            last_char_index ← the index of E
8.            Copy e into CB[last_char_index]
9.            Feed e into the KMP algorithm
10.           Update the list of the found matches if needed
11.           Update the linked list of relevant characters blocks if needed
12.     } else (e is a pointer of kind (offset, length, slide)) {
13.           Copy the relevant characters and skip the irrelevant ones
14.     }
15. }
16.     Feed CB's characters into the KMP algorithm,
        until the first uninitialized entry of CB,
        updating the list of the found matches if needed
```

4 Experimental Results

This section shows experimental results for each technique and compares the compression performance and the compression and pattern matching processing time.

The tests were run on the standard benchmarking data set, which is "The Canterbury Corpus" ([17]), and "The Large Calgary Corpus" ([18]). The tests have been run on 32-bit Windows 7 machine with *"Intel(R) Core(TM)2 Quad CPU Q8200"* processor and *3GB* RAM memory. The program has been implemented in Java. All runs were performed with *"-Xms64m -Xmx1500m"* java parameters. It is especially required for large files, since for simplicity a whole file is read into a memory before the compression. The compressed text is also accumulated in a memory before storing it on the disk. The search time specifies the time of a search in the in-memory text and does not include the load time of a file into a memory.

4.1 Pattern Matching: Methods Comparison

This section compares the search times achieved by different methods. The compared techniques are:

- "Decompress and Search" in byte-format LZSS-compressed file (described in sections 2.3 and 2.4)
- "Search During Decompress" in byte-format LZSS-compressed file (described in sections 2.5 and 3.2.4)
- Manber's compressed matching (described in section 2.2)
- Shapira/Klein's pattern matching in byte-format Shapira/Klein-compressed file (described in sections 2.7 and 3.4)

For our experiments we repeated the pattern matching algorithm 100 times and averaged the results. Each experiment searched for a substring taken randomly from the original file. The length of a searched pattern randomly varies from 4 to 10 characters. The time is given in seconds, and the best results are shown in a **bold** font, the worst results are <u>underlined</u>. In the first table the search is performed until all matches are found. In the second table the search is performed until the first match is found.

<u>Search for all matches:</u>

File	**File Size (bytes)**	**"Decompress & Search"**	**"Search During Decompress"**	**Manber**	**Shapira/Klein**
Canterbury Corpus					

alice29.txt	152089	1.436s	0.877s	**0.406s**	9.213s
asyoulik.txt	125179	1.210s	0.751s	**0.296s**	8.124s
cp.html	24603	0.220s	**0.121s**	0.63s	1.372s
fields.c	11150	**0.105s**	0.61s	0.30s	0.491s
grammar.lsp	3721	0.33s	0.21s	**0.10s**	0.156s
kennedy.xls	1029744	9.100s	5.6s	**4.602s**	45.683s
lcet10.txt	426754	3.961s	2.434s	**1.28s**	22.451s
plrabn12.txt	481861	4.796s	3.10s	**1.104s**	25.941s
ptt5	513216	7.924s	**6.33s**	18.890s	31.701s
sum	38240	0.353s	0.205s	**0.138s**	2.512s
xargs.1	4227	0.43s	0.25s	**0.11s**	0.195s
Large Canterbury Corpus					
bible.txt	4047392	36.294s	21.62s	**9.51s**	156.146s
E.coli	4638690	47.131s	29.599s	**12.802s**	191.221s
world192.txt	2473400	21.194s	12.255s	**6.28s**	93.321s
Large Calgary Corpus					
bib	111261	0.969s	0.567s	**0.265s**	7.312s
book1	768771	7.425s	4.703s	**1.766s**	40.367s
book2	610856	5.528s	3.313s	**1.474s**	28.652s
geo	102400	1.70s	0.713s	**0.363s**	7.532s
news	377109	3.386s	2.77s	**0.971s**	19.151s
obj1	21504	0.194s	**0.118s**	0.85s	1.479s
obj2	246814	2.252s	1.236s	**0.859s**	12.709s
paper1	53161	0.478s	0.286s	**0.130s**	3.279s
paper2	82199	0.770s	0.461s	**0.193s**	5.215s
paper3	46526	0.445s	0.265s	**0.110s**	2.971s
paper4	13286	**0.132s**	0.79s	0.32s	0.676s
paper5	11954	**0.116s**	0.77s	0.30s	0.595s
paper6	38105	0.348s	**0.198s**	0.93s	2.330s
pic	513216	7.600s	**5.519s**	17.215s	30.75s
progc	39611	0.357s	0.209s	**0.101s**	2.430s

progl	71646	0.597s	0.314s	**0.184s**	3.962s
progp	49379	0.404s	0.218s	**0.132s**	2.722s
trans	93695	0.758s	0.387s	**0.238s**	4.868s

Search for the first match:

File	**File Size (bytes)**	**“Decompress & Search”**	**“Search During Decompress”**	**Manber**	**Shapira/Klein**
Canterbury Corpus					
alice29.txt	152089	0.985s	0.219s	**0.100s**	2.643s
asyoulik.txt	125179	0.865s	**0.214s**	0.80s	2.506s
cp.html	24603	0.153s	0.49s	**0.21s**	0.439s
fields.c	11150	0.73s	0.24s	**0.11s**	0.186s
grammar.lsp	3721	0.26s	**0.10s**	0.5s	0.65s
kennedy.xls	1029744	5.439s	0.895s	**0.719s**	8.823s
lcet10.txt	426754	2.672s	0.537s	**0.232s**	5.739s
plrabn12.txt	481861	3.242s	0.625s	**0.230s**	6.168s
ptt5	513216	2.7s	0.127s	**0.106s**	1.698s
sum	38240	**0.261s**	0.97s	0.56s	1.91s
xargs.1	4227	0.32s	**0.17s**	0.5s	0.89s
Large Canterbury Corpus					
bible.txt	4047392	22.763s	2.397s	**1.69s**	19.830s
E.coli	4638690	27.943s	0.372s	**0.153s**	3.44s
world192.txt	2473400	13.291s	1.544s	**0.748s**	12.655s
Large Calgary Corpus					
bib	111261	0.669s	**0.133s**	0.59s	1.674s
book1	768771	5.106s	0.880s	**0.343s**	8.362s
book2	610856	3.802s	0.762s	**0.340s**	7.725s
geo	102400	0.870s	0.379s	**0.190s**	4.74s

news	377109	2.391s	0.548s	**0.253s**	5.692s
obj1	21504	**0.147s**	0.54s	0.30s	0.572s
obj2	246814	1.582s	0.467s	**0.316s**	5.366s
paper1	53161	0.368s	**0.128s**	0.53s	1.568s
paper2	82199	0.562s	**0.176s**	0.70s	2.153s
paper3	46526	0.330s	**0.104s**	0.41s	1.283s
paper4	13286	0.99s	0.42s	**0.14s**	0.332s
paper5	11954	0.97s	0.43s	**0.14s**	0.310s
paper6	38105	**0.261s**	0.88s	0.37s	1.56s
pic	513216	2.52s	0.149s	**0.133s**	2.117s
progc	39611	**0.277s**	0.96s	0.41s	1.131s
progl	71646	0.433s	**0.119s**	0.71s	1.515s
progp	49379	**0.295s**	0.89s	0.44s	0.992s
trans	93695	0.513s	**0.147s**	0.72s	1.598s

The tables above show that LZSS search-during-decompress outperforms Shapira/Klein's pattern matching in the vast majority of the test cases. Manber's pattern matching usually outperforms the LZSS search-during-decompress, especially in larger files, but it is less stable, since it may first search for a shorter pattern which might produce false matches. When the search is performed until the first match, Shapira/Klein's pattern matching outperforms byte-format LZSS-based “Decompress and Search” on large files and is usually slower in smaller files. Note that in rare cases of small files byte-format LZSS-based “Decompress and Search” outperforms all other techniques because of its simplicity (even LZSS search-during-decompress has its overhead of maintaining a circular buffer).

4.2 Shapira/Klein's Pattern Matching: Methods' Dependency on the Pattern Length

This section explores the dependency of Shapira/Klein's pattern matching on the length of the searched pattern. Again, the search was performed 100 times for each technique and the searched pattern was randomly selected from the corresponding original file. Each column represents the results for a fixed length pattern. The search is performed until the first match is found. The table shows the ratio between the search-during-decompress time of byte-format LZSS method and the search time of Shapira/Klein's one. The smaller the ratio the better the LZSS search-during-decompress performs as compared to Shapira/Klein's pattern matching.

File	**File Size (bytes)**	**Len=1**	**Len=2**	**Len=3**	**Len=4**	**Len=15**	**Len= 100**
Canterbury Corpus							
alice29.txt	152089	0.4	0.15	0.08	0.07	0.08	0.08
asyoulik.txt	125179	0.33	0.11	0.07	0.08	0.08	0.08
cp.html	24603	0.6	0.15	0.13	0.11	0.1	0.08
fields.c	11150	0.46	0.44	0.15	0.14	0.12	0.12
grammar.lsp	3721	0.83	0.22	0.14	0.14	0.13	0.13
kennedy.xls	1029744	0.57	0.1	0.1	0.1	0.1	0.1
lcet10.txt	426754	0.23	0.11	0.08	0.08	0.09	0.09
plrabn12.txt	481861	0.67	0.24	0.1	0.09	0.11	0.1
ptt5	513216	0.04	0.08	0.08	0.08	0.08	0.08
sum	38240	0.14	0.07	0.08	0.07	0.07	0.08
xargs.1	4227	0.5	0.17	0.12	0.18	0.12	0.12
Large Canterbury Corpus							
bible.txt	4047392	0.4	0.18	0.09	0.12	0.12	0.12
E.coli	4638690	0.93	1.25	0.65	0.17	0.14	0.14
world192.txt	2473400	0.9	0.2	0.11	0.1	0.11	0.11
Large Calgary Corpus							
bib	111261	0.48	0.09	0.08	0.07	0.07	0.07
book1	768771	0.86	0.11	0.07	0.09	0.1	0.11
book2	610856	0.35	0.08	0.08	0.09	0.1	0.1
geo	102400	0.28	0.08	0.08	0.08	0.08	0.08
news	377109	0.17	0.08	0.07	0.08	0.09	0.09
obj1	21504	0.16	0.08	0.08	0.09	0.08	0.09
obj2	246814	0.11	0.07	0.08	0.07	0.08	0.08
paper1	53161	0.21	0.1	0.08	0.08	0.07	0.08
paper2	82199	0.73	0.14	0.08	0.07	0.07	0.07

paper3	46526	0.53	0.14	0.12	0.08	0.08	0.08
paper4	13286	0.38	0.15	0.16	0.13	0.11	0.12
paper5	11954	0.64	0.13	0.17	0.12	0.13	0.13
paper6	38105	0.17	0.1	0.08	0.08	0.07	0.08
pic	513216	0.03	0.07	0.07	0.07	0.07	0.08
progc	39611	0.24	0.11	0.08	0.08	0.08	0.07
progl	71646	0.63	0.13	0.08	0.08	0.07	0.07
progp	49379	0.6	0.13	0.1	0.1	0.08	0.07
trans	93695	0.1	0.12	0.07	0.07	0.07	0.07

The table shows that the ratio is usually larger for short strings up to 4 characters, and then remains more or less constant for each file. That is, Shapira/Klein performs relatively better for very short strings. This can be explained by the fact that in average short patterns leave more characters "irrelevant". These characters are skipped by Shapira/Klein's pattern matching.

4.3 Compression: Methods Comparison

This section compares the compression ratio and compression times achieved by different methods. The compared techniques are:

- Bit-format implementation of LZSS (described in section 3.2.5)
- Byte-format implementation of LZSS (described in section 3.2.4)
- Shapira/Klein's method (described in sections 2.6 and 3.3)
- Manber's method (described in sections 2.1 and 3.1)
- *Gzip*: Linux/UNIX utility run with default options

LZSS and Shapira/Klein's compressions use both hash-table and suffix trees to find each string's previous occurrences, as described in section 3.2.3.3. The maximum text window size is 32000 byte, the maximum bucket size in the hash table that stores the string's previous occurrences is 5. Manber's compression uses 100 trials to improve the original partition, as recommended by Manber. The full list of parameters with their description and the default values is shown in "Appendix B". All the values used in this run are the default ones.

Each entry in the table consists of two parts: the first one presents the compression ratio in percent, while the second one specifies the compression time, in seconds.

The compression time of *Gzip* utility is not shown. The best results are shown in a **bold** font, the worst results are underlined. If the best compression ratio is achieved by *Gzip*, the next best result is also shown in a **bold** font.

File	File Size	BIT LZSS	BYTE LZSS	Shapira-Klein	Manber	GZIP
Canterbury Corpus						
alice29.txt	152089	**39.16%** 1.484s	44.83% 1.191s	58.50% 1.333s	66.90% 0.168s	**35.79%**
asyoulik.txt	125179	**42.23%** 1.151s	49.12% 0.987s	63.11% 1.62s	65.61% 0.120s	**39.10%**
cp.html	24603	**34.76%** 0.133s	42.29% 0.101s	48.09% 0.158s	70.69% 0.155s	**32.51%**
fields.c	11150	**31.80%** 0.51s	35.51% 0.30s	42.19% 0.37s	72.22% 0.131s	**28.19%**
grammar.lsp	3721	**36.58%** 0.17s	42.27% 0.14s	49.45% 0.12s	76.57% 0.87s	**33.49%**
kennedy.xls	1029744	**13.93%** 5.689s	34.02% 4.819s	41.46% 5.738s	100.05% 1.266s	20.08%
lcet10.txt	426754	**37.08%** 4.63s	42.29% 3.422s	55.44% 3.711s	66.66% 0.252s	**33.95%**
plrabn12.txt	481861	**44.47%** 4.961s	51.72% 4.83s	67.16% 4.505s	64.39% 0.221s	**40.51%**
ptt5	513216	**10.48%** 3.73s	14.53% 2.877s	29.19% 9.878s	96.04% 0.556s	11.00%
sum	38240	**36.41%** 0.219s	38.89% 0.179s	47.86% 0.200s	100.67% 1.4s	**33.80%**
xargs.1	4227	**45.78%** 0.21s	51.24% 0.8s	60.73% 0.16s	74.54% 0.80s	**41.54%**
Large Canterbury Corpus						
bible.txt	4047392	**32.03%** 42.71s	37.00% 27.660s	49.80% 35.4s	64.93% 0.986s	**29.43%**
E.coli	4638690	**27.16%** 54.146s	36.43% 38.200s	65.74% 40.89s	74.01% 0.865s	28.91%
world192.txt	2473400	**32.10%** 18.197s	37.49% 18.216s	46.86% 19.820s	67.88% 0.713s	**29.30%**

Large Calgary Corpus						
bib	111261	**34.87%** 1.13s	40.70% 0.844s	49.98% 0.960s	63.47% 0.184s	**31.51%**
book1	768771	**45.09%** 8.335s	51.83% 6.724s	66.70% 7.888s	65.32% 0.310s	**40.76%**
book2	610856	**37.26%** 5.450s	42.18% 5.105s	54.54% 5.439s	67.13% 0.347s	**33.84%**
geo	102400	**68.50%** 0.857s	79.44% 0.815s	93.60% 0.986s	100.27% 1.587s	**66.89%**
news	377109	**40.67%** 3.639s	47.85% 3.91s	58.06% 3.503s	72.24% 0.338s	**38.41%**
obj1	21504	**52.23%** 0.110s	54.76% 0.70s	65.14% 0.119s	101.20% 1.140s	**48.01%**
obj2	246814	**35.96%** 2.92s	39.22% 1.820s	46.41% 2.38s	100.11% 1.733s	**33.07%**
paper1	53161	**38.43%** 0.418s	42.89% 0.331s	53.85% 0.358s	68.02% 0.180s	**34.94%**
paper2	82199	**39.48%** 0.665s	44.70% 0.625s	57.96% 0.695s	66.39% 0.173s	**36.20%**
paper3	46526	**42.36%** 0.417s	48.07% 0.299s	60.66% 0.328s	66.71% 0.144s	**38.90%**
paper4	13286	**45.42%** 0.68s	51.45% 0.50s	62.78% 0.112s	69.83% 0.115s	**41.67%**
paper5	11954	**45.96%** 0.56s	51.05% 0.44s	61.91% 0.49s	70.17% 0.152s	**41.79%**
paper6	38105	**38.46%** 0.262s	42.52% 0.223s	53.34% 0.232s	68.36% 0.171s	**34.73%**
pic	513216	**10.48%** 3.195s	14.53% 2.839s	29.19% 8.294s	96.05% 0.690s	11.00%
progc	39611	**36.85%** 0.277s	41.53% 0.217s	51.07% 0.264s	70.74% 0.163s	**33.51%**
progl	71646	**24.69%** 0.539s	28.80% 0.461s	36.14% 0.498s	71.15% 0.161s	**22.71%**
progp	49379	**25.09%** 0.342s	28.37% 0.276s	34.91% 0.328s	72.00% 0.167s	**22.77%**
trans	93695	**22.61%** 0.667s	26.44% 0.582s	32.83% 0.701s	71.40% 0.217s	**20.26%**

The table shows that the bit-format LZSS compression ratio is usually within **3-4%** worse than *Gzip*'s. This verifies our implementation of LZSS. *Gzip*'s better compression ratio can be attributed mainly to its more sophisticated way of encoding LZSS offsets. (*Gzip* prefixes each offset with a Huffman-encoded value from 1 to 32 to designate one of 32 *offset* size groups, while our implementation just stores the Huffman-encoded *offset*'s size in bytes). The byte-format LZSS compression ratio is usually within **5-7%** of the bit-format one. This is roughly the "price" of staying with a byte format. Shapira/Klein method's compression ratio is usually within **10-15%** of the byte-format LZSS. This is roughly the "price" of adding the third parameter, *slide*, to the pointer structure. Manber's method achieves the worst compression ratio and is usually the fastest. On average, it achieves **30%** reduction of the input size, similar to the figures reported by Manber.

4.4 Compression: LZSS Text Window Implementations Comparison

This section compares the relative performance of several implementations of LZSS text windows, i.e. methods to store and find the previous occurrences. The first table shows the results for the bit-format LZSS method. The second table shows the results for the byte-format compression. The string prefix length that serves as a hash key is 4 characters for the bit-format compression and 3 characters for the byte-format one. These numbers yield the best compression ratios. The full list of parameters with their description and the default values is shown in "Appendix B". All the values used in this run are the default ones. The compared implementations are:

- ***"ST+Hash[100]"***: Suffix Trees + Hash Table (described in section 3.2.3.3) with hash bucket size=100. It shows the best possible results achievable by the given algorithm and the given encoding.
- ***"ST+Hash[5]"***: Suffix Trees + Hash Table (described in section 3.2.3.3) with hash bucket size=5
- ***"ST"***: Suffix Trees only (described in section 3.2.3.2)
- ***"Hash[5]"***: Hash Table (described in section 3.2.3.1) with hash bucket size=5
- "***Hash[100]***": Hash Table (described in section 3.2.3.1) with hash bucket size=100

Hash bucket size of 5 is chosen as a reasonable default, while hash bucket size of 100 is chosen to show the compression ratio limits of a given implementation.
The tests have been run on the Canterbury Corpus data set ([17]). The first table shows both the compression ratio and the compression time for each method. The second table only shows the compression ratios.

The compression times are less relevant in the byte-format LZSS: as explained in section 3.2.4.2, the byte-format LZSS compression algorithm chooses from several previous occurrences of the current string by computing the precise saving from using each occurrence. The time overhead of choosing the best previous occurrence from several ones is comparable to the time required to find them. In contrast, the bit-format LZSS algorithm uses a fast heuristics to choose the best previous occurrence from several ones (see section 3.2.5.4), and its time is negligible compared to the time of finding them.

Compression ratio and compression times for the bit-format LZSS method:

File	**File Size**	**ST + Hash[100]**	**ST + Hash[5]**	**ST**	**Hash[5]**	**Hash[100]**
Canterbury Corpus						
alice29.txt	152089	38.78% 1.900s	39.16% 1.428s	40.49% 1.108s	40.79% 0.402s	38.82% 0.643s
asyoulik.txt	125179	41.97% 1.425s	42.23% 1.164s	43.56% 0.885s	43.28% 0.378s	41.98% 0.562s
cp.html	24603	34.68% 0.165s	34.76% 0.135s	35.15% 0.98s	36.16% 0.52s	34.69% 0.69s
fields.c	11150	31.78% 0.62s	31.80% 0.76s	33.27% 0.35s	33.52% 0.16s	31.78% 0.29s
grammar.lsp	3721	36.52% 0.21s	36.58% 0.12s	38.08% 0.11s	36.93% 0.5s	36.52% 0.7s
kennedy.xls	1029744	13.77% 7.127s	13.93% 5.664s	16.96% 4.441s	14.23% 1.739s	13.79% 2.859s
lcet10.txt	426754	36.77% 4.964s	37.08% 4.65s	38.52% 3.226s	38.38% 1.170s	36.82% 1.862s
plrabn12.txt	481861	44.12% 5.949s	44.47% 5.49s	45.83% 3.711s	45.78% 1.523s	44.15% 2.562s

ptt5	513216	10.40% 3.768s	10.48% 3.154s	10.53% 2.726s	11.63% 0.568s	10.62% 1.67s
sum	38240	36.35% 0.282s	36.41% 0.231s	38.06% 0.188s	36.95% 0.70s	36.37% 0.112s
xargs.1	4227	45.75% 0.25s	45.78% 0.22s	46.89% 0.9s	46.11% 0.16s	45.75% 0.7s
bible.txt	4047392	31.69% 54.323s	32.03% 36.518s	33.02% 31.227s	33.77% 8.964s	31.81% 16.908s
E.coli	4638690	27.15% 193.459s	27.16% 55.779s	27.51% 45.121s	27.07% 16.218s	27.16% 154.116s
world192.txt	2473400	31.79% 24.713s	32.10% 21.93s	33.79% 12.842s	33.02% 5.50s	31.82% 7.246s

The table shows that “Hash[5]” achieves compression ratio which is usually within 1-1.5% of the “best possible“ results achieved by “ST+Hash[100]”. The compression ratio of “ST+Hash[5]” is very close to that of “ST+Hash[100]”. The compression ratio of “ST” is usually comparable with that of “Hash[5]”, but is **much slower**. In many cases it is even slower than “Hash[100]”!

Compression ratios for the byte-format LZSS method:

File	File Size	ST + Hash[100]	ST + Hash[5]	ST	Hash[5]	Hash[100]
Canterbury Corpus						
alice29.txt	152089	44.00%	44.83%	50.18%	48.20%	44.10%
asyoulik.txt	125179	48.44%	49.12%	55.25%	52.01%	48.51%
cp.html	24603	42.35%	42.29%	44.80%	44.06%	42.38%
fields.c	11150	35.35%	35.51%	37.70%	37.35%	35.39%
grammar.lsp	3721	42.30%	42.27%	43.54%	42.76%	42.30%
kennedy.xls	1029744	33.85%	34.02%	34.22%	34.68%	33.95%
lcet10.txt	426754	41.56%	42.29%	47.53%	45.60%	41.65%
plrabn12.txt	481861	50.79%	51.72%	58.31%	54.76%	50.89%
ptt5	513216	14.49%	14.53%	15.30%	15.81%	15.13%
sum	38240	38.89%	38.89%	42.82%	40.18%	39.04%
xargs.1	4227	51.12%	51.24%	52.97%	51.90%	51.12%
bible.txt	4047392	36.24%	37.00%	41.39%	40.74%	36.47%

E.coli	4638690	34.41%	36.43%	41.44%	40.14%	34.89%
world192.txt	2473400	36.78%	37.49%	42.09%	40.31%	36.84%

The table shows that a combination of suffix trees and hash tables "ST+Hash[5]" improves the compression ratio by 2-3% in average. On the other hand, the compression ratio of "ST" is slightly worse than that of "Hash[5]". That is, for the byte-format LZSS method, hash tables outperform the suffix trees both in compression ratios and in compression times. As has been already mentioned, the naïve usage of suffix trees in the LZSS algorithm finds the oldest matches, yielding longer pointers.

5 Analysis

5.1 Performance of Shapira/Klein's Method

It has been shown in section 4 that Shapira/Klein's pattern matching method is actually slower than a search-during-decompress in LZSS-compressed text. There are several reasons behind it.

First, Shapira/Klein's compressed matcher can perform faster than LZSS compressed matcher only due to skipped characters. That is, irrelevant characters referenced by a pointer are (1) not copied to the circular buffer (2) KMP algorithm ([6]) can be instructed to skip several irrelevant characters at once. In practice this saving is overshadowed by the time required to implement a more complex logic of the pattern matching algorithm, which includes preserving linked list of relevant character blocks, finding the actual place for the next literal etc. In addition, the number of skipped characters decreases in average when increasing the length of a searched pattern.

It should be noted that most probably the replacement of LZSS pointers is not the bottleneck in compressed search within e.g. *gzip*-compressed text. After all, it only requires bytes copy from one position in an array to another which is a relatively fast operation. Most probably, the additional compression of LZSS pointers and literals (e.g. Huffman encoding in case of DEFLATE) slows the decompression process much greater. Shapira/Klein's method mainly attempts to optimize the first part (i.e. replacement of LZSS pointers), offering nothing to optimize the second part (decompression of additionally compressed LZSS pointers and literals). Changing the way that the LZSS elements are additionally encoded can yield a compression ratio better than Manber's and a search time better than that in a bit-format LZSS compressed text. That is, it can serve as a compromise between the two techniques.

5.2 Compression of LZSS Pointers and Literals

It should be noted that the “byte-format” LZSS-compression scheme used in this work is very far from perfect. After all, designing this scheme was not the main purpose of this work. This is in contrast to well-thought and mature bit-format techniques such as binary Huffman or arithmetic coding. Despite the relatively primitive technique employed in the “byte-format” LZSS compression, the compression ratio it achieves is within *8-12%* of the one achieved by *'gzip'*, and is usually much better than Manber's. It is reasonable to assume that smarter byte-format techniques would achieve better compression ratio while still providing the same or better compressed matching speed.

The “encoding rule” technique can be improved in a variety of ways: dynamically deduce the encoding rules, drop the restriction on characters and pointers to span an integer number of bytes (some internal statistics suggest that the most frequent pointers would span 10-14 bits) - it still can require only a constant number of fast operations to decompress an element, and others. Still, “mimicking” the well-thought *DEFLATE* algorithm ([16]) and replacing the binary Huffman encoding ([8]) or the Arithmetic Coding (10) by some compromise would most probably achieve better results.

Example: *DEFLATE* uses a binary Huffman tree to encode literals and lengths. The encoding algorithm can replace the original Huffman tree by the one in which no Huffman-encoded symbol spans more than N bits, where N may be for example 10. Symbols whose original Huffman encoding spans more than N bits will be replaced by a single special symbol in the second Huffman tree. Such symbols will be written as is to the output, prefixed with the Huffman encoding of the special symbol. For offsets, *DEFLATE* uses 32 possible symbols, Huffman encoded, to encode the number of bits the corresponding offset can span. Again, the encoding algorithm will replace the original Huffman tree by the one in which no Huffman-encoded symbol spans more than N bits. It could be done by artificially increasing the frequency of elements whose original Huffman encoding spans more than N bits. The advantage of bounding the Huffman-encoding by N bits is that decoding of every element (either character, length or offset prefix) would now always take a constant number of simple operations, using a prebuilt array of size 2^N (This technique appears in ([4]) and is also described in section 3.2.4.3 of this paper). Of course, this would decrease the compression ratio, but improve the decompression and search-during-decompress processing time.

Another possible technique to encode offsets is as follows. Assume that the technique used to write offsets is similar to that of DEFLATE, i.e. each offset is prefixed with a short symbol telling the number of bits that the offset spans. In this context "`writing a value`" means first writing such symbol and then writing the value itself. Now, enumerate N most frequent offsets by numbers from 1 to N (0 is reserved as a special symbol), thus mapping each such offset to a small number. The map is appended to the compressed text. Offset values larger than N and not found in the dictionary are written as usual. Offset values smaller than N and not found in the dictionary are prefixed with a special symbol 0 (as usual, both values are prefixed with the number of bits they span). Frequent offset values (found in the map) are written using this map.

5.3 Performance of Suffix Trees for LZSS

As shown in section 3.2.3.2, suffix trees have been used in implementations of LZSS and Shapira/Klein's methods to find the longest previous occurrence of a given string. Results presented in section 4 show that suffix trees were *2-3* times slower than hash tables, while yielding similar compression ratios as hash tables. Still, their combination improved the compression ratio by *1-1.5%* in average for the bit-format LZSS method and by *2-3%* in average for the byte-format one. It should be noted that the suffix tree may be changed to store more recent matches, in which case the results would be different, but this was not implemented.

5.4 Alternative LZSS Search Structures

Results presented in section 4 show that increasing the maximum size of hash buckets improves the compression ratio, but also can drastically increase the processing time of compression. As the k-length prefix of a string is used to compute a hash key, the number of text substrings with the same 3-characters' prefix may be large.

One of possible improvements of the performance of the suffix trees is to change it to store more recent matches. If on insertion of a new suffix the algorithm goes down one edge, it means that the new suffix contains the characters of this edge. Therefore, the algorithm may update the edge to point to the characters of the suffix, i.e. to a more recent location in the text having the same characters as the edge.

Another improvement relates to the hash tables performance used in LZSS compression: instead of using a single hash table, use two or three of them simultaneously. Each table works with string prefixes of different lengths, e.g. 3, 4, and 6. That is, to find or insert a string into the first table, one should compute the hash function of the first 3 characters of the string. To find or insert a string into the second table, one should compute the hash function of the first 4 characters etc. Each step of the algorithm inserts the current string into the three hash tables. To find the longest match, the algorithm first searches in the table based on the longest prefix (6 in our example). If it yields no match, the algorithm concludes that there is no match with length greater than or equal to the prefix length (6 in our example), and checks in the table based on a shorter prefix etc., otherwise it just uses the match. As shown in section 3.2.3.1, the hash functions can be computed in constant time, regardless of the prefix size.

Probably the best data structure for LZSS-based compression is a truncated suffix tree ([19]), which maintains a suffix-tree of a given depth k only. It allows finding the most recent matches among the ones with length up to k characters.

5.5 KMP Is Not the Fastest Pattern Matching Algorithm

The experimental results shown in section 4 have a bias against the Manber methods. The underlying pattern matching algorithm used both in Shapira/Klein's pattern matching and in Manber's pattern matching was KMP ([6]). But it is known that in most practical applications, Boyer-Moor algorithm ([9] performs faster than KMP. This algorithm could be freely used as the underlying algorithm for the Manber's pattern matching. But it is not clear how it could be used inside LZSS-compressed or Shapira/Klein-compressed text, since it violates the methods' requirement from the underlying algorithm to process characters from left to right. Still, it should be noted that Manber's technique does not allow a search for regular expressions, since it compresses the searched pattern, while LZSS and Shapira/Klein's methods can feed the original characters to any pattern matching algorithm.

6 Conclusions

The main conclusion of this paper is that Shapira/Klein's method apparently did not achieve its goals. It takes more time to search inside the Shapira/Klein-compressed text than in LZSS-compressed text and the search requires the same amount of additional memory as search-during-decompress in LZSS. At the same time, LZSS achieves better compression ratio than Shapira/Klein's method because of an additional parameter, '*slide*', added by the latter to the pointer structure.

Second, a compromise between Manber's compression offering fast search times and bit-format LZSS compression offering a good compression ratio can be achieved by employing a byte-format LZSS compression. The byte-format LZSS compression employed in this paper is quite naïve and can be improved in many ways. The actual search time improvement of a byte-format LZSS compression over a bit-format one such as *gzip* is yet to be verified.

Third, although suffix trees can find long matches in LZSS-based compression method, in practice they are slower compared to the hash tables and tend to find old matches with large offsets. This may be improved by changing the suffix tree insertion algorithm to update the edges it passes to point to newer positions in text, but this has not been implemented. Still, combination of suffix trees and hash tables can improve the compression ratio, especially for the byte-format LZSS compression. There are several possible techniques to improve the performance of the LZSS search structure, but the research and comparison of possible techniques is outside of the scope of this work.

7 Further Research

The restricted scope of this work left several important questions on the subject unanswered.

First of all, before trying to improve the search speed in *DEFLATE* ([16], [4]) compressed files, one could verify how slow or how fast it is. Libraries providing decoded “streams” to *DEFLATE*-encoded files exist in many programming languages. Of course, such streams can only be used in pattern matching algorithms traversing the input from left to right, such as the KMP algorithm ([6]), and are not applicable to e.g. the Boyer-Moore one ([9]), since they decode every character of the input. To answer this question, one could compare the search time using the KMP algorithm inside a *DE-FLATE*-compressed file with the speed of the KMP pattern matching and the Boyer-Moore pattern matching in an uncompressed text. In addition, there are methods to adapt the Boyer-Moore pattern matching algorithm to search Huffman encoded text, e.g. ([20]). It is probably possible to adapt the Boyer-Moore algorithm to search inside DEFLATE-encoded text as well.

A second question is to determine which component is the bottleneck of the *DE-FLATE*-compressed pattern matching: decoding LZ77/LZSS or decoding the binary Huffman-encoded symbols? To answer this question one could compare the speed of a compressed pattern matching inside a LZSS-compressed text with that in a *DEFLATE*-compressed text.

References

[1] S. Klein, D. Shapira, (2000) "A New Compression Method for Compressed Matching", Data Compression Conference – DCC-2000, 400-409

[2] U. Manber, "A Text Compression Scheme That allows Fast Searching Directly in the compressed File", ACM Trans. on Inf. Sys. 15 (1997) 124-136.

[3] J. A. Storer, T. G. Szymanski, (1982) "Data Compression via Textual Substitution", Journal of ACM, 29(4), 928-951.

[4] J-L. Gailly, M. Adler, GZIP [Internet]
Available from: http://www.gzip.org/algorithm.txt

[5] Wikipedia, gzip [Internet]
Available from: http://en.wikipedia.org/wiki/Gzip

[6] D. E. Knuth, J. Morris, V. Pratt, "Fast Pattern Matching in Strings", SIAM J. on Computing 6 (1977) 323-350.

[7] A. Amir, G. Benson, M. Farach, "Let Sleeping Files Lie: Pattern Matching in Z-compressed Files", Journal of Computer and System Sciences 52, 2 (April 1996), 299-307

[8] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E. (September 1952) 1098–1102

[9] R. S. Boyer, J. S. Moore, "A Fast String Searching Algorithm", Comm. ACM 20 (1997) 762–772

[10] R. N. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm", Data Compression Conference 1991 (DCC'91), 8-11 April , 1991, Snowbird, Utah, 362-371

[11] Wikipedia, Arithmetic coding [Internet]
Available from: http://en.wikipedia.org/wiki/Arithmetic_coding

[12] Wikipedia, Suffix tree [Internet]
Available from: http://en.wikipedia.org/wiki/Suffix_tree

[13] M. Nelson, "Fast String Searching With Suffix Trees" [Internet]
Available from: http://marknelson.us/1996/08/01/suffix-trees

[14] E. Ukkonen, "On-Line Construction of Suffix Trees", Algorithmica 14, 3 (1995) 249-260

[15] N. J. Larsson, "Extended Application of Suffix Trees To Data Compression", Data Compression Conference – DCC-1996 (1996) 190

[16] Wikipedia, DEFLATE [Internet]
Available from: http://en.wikipedia.org/wiki/DEFLATE

[17] R. Arnold, T. Bell, "The Canterbury Corpus" [Internet]. New Zealand, University of Canterbury; 1997
Available from: http://corpus.canterbury.ac.nz/

[18] I. Witten, T. Bell, J. Cleary, "The Calgary Corpus" [Internet]. Canada, University of Calgary; 1987
Available from: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/

[19] J.C. Na, A. Apostolico, C.S. Iliopoulos, K. Park, "Truncated Suffix Trees and Their Application to Data Compression", Theoretical Computer Science 304, 1-3 (2003), 87-101

[20] D. Cantone, S. Faro, E. Giaquinta, "Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts", The Prague Stringology Conference 2009 (PSC'09), 29-39

Appendix A

Running the Program

The program is implemented in Java and as such is invoked using 'java' command:

```
$ java -cp <path to the program's classes>  cs.main.Main
<command line arguments>
```

Note that no special efforts have been done to reduce the amount of the used memory, preferring the simplicity of the implementation. For instance, the program just reads the whole file into its memory and then starts to either compress or search it. To increase the default Java heap size for large files, the following `java` options can be used: "`-Xms`" for initial Java heap size and "`-Xmx`" for maximum heap size, e.g. "`-Xms64m -Xmx1500m`".

The program's command line options can be grouped into 4 "commands". Each "command" has its own main option followed by command-specific arguments. In addition, option `'-help'` prints the synopsis of all allowed options.

The "commands" are:

Main option	Description
-compress	Compresses a file and writes the compressed one. In addition, verifies the compression by internally decompressing the compressed text and comparing it to the original text.
-search	Search inside a compressed file
-runcompressions	For statistics gathering: Run compression on all files in a directory and output the compression ratio and compression time in a tabular format. In addition, verifies the compression of each file by internally decompressing the compressed text and comparing it to the original text.
-runsearches	For statistics gathering: Run compression and then run a specified number of searches inside a compressed file and output the search time in a tabular format.

The arguments of each "command" are:

Option	Description
-compress	
-file <file>	Original uncompressed file. The output file will have the following name: <original file name>.<compression method name>
-comprmethod	One of: BIT_LZSS|BYTE_LZSS|MANBER|SHAPIRA_KLEIN

-configfile <config file>	Optional: the configuration file as described in Appendix B.
-search	
-file <file>	Compressed file. Should have an extension bearing the name of the used compression method. The compression method will be determined from the extension. Can be used with all compression methods besides “BIT_LZSS”.
-decompressandsearch	Optional: if present, use the “decompress and search” strategy. Can be used only with “BYTE_LZSS” compression method.
-pattern <pattern>	The searched pattern
-configfile <config file>	Optional: the configuration file as described in Appendix B.
-runcompressions	
-dir <directory>	Directory containing uncompressed files
-comprmethod <comma-separated list of compression methods>	Optional: list of compression methods to run. Allowed compression methods: BIT_LZSS, BYTE_LZSS, MANBER, SHAPIRA_KLEIN, for example: “BIT_LZSS, MANBER”. Compression ratio and compression time will be output for each compression method and for each file, in a tabular format. If missing, all compression methods will be run.
-gzdir <directory with gzipped files>	Directory containing files with “.gz” extension. It is used to verify the compression ratio of “gzip” command.
-configfile <config file>	Optional: the configuration file as described in Appendix B.
-runsearches	
-dir <directory>	Directory containing uncompressed files. The files will be internally compressed, and then a random substring of an original text will be searched in the compressed text.
-decompressandsearch	Optional: if present, use the “decompress and search” strategy to search in a text compressed with “BYTE_LZSS” method.
-comprmethod <comma-separated list of compression methods>	Optional: list of compression methods to run. Allowed compression methods: BYTE_LZSS, MANBER, SHAPIRA_KLEIN, for example: “BYTE_LZSS, MANBER”. Search times will be output for each compression method and for each file, in a tabular format. If missing, all allowed compression methods will be tried.
-ptrnlen <pattern length>	Optional: the length of a random search pattern. Overrides “run.search.ptrn_max_length” and “run.search.ptrn_min_length” configuration properties, described in Appendix B. The search pattern will always be a substring of an original text.
-searchcnt <search count>	Optional: the number of search patterns to search. Overrides “run.search.ptrn_cnt” configuration property, described in Appendix B.
-configfile <config file>	Optional: the configuration file as described in Appendix B.

Appendix B

Configuring the Program

The algorithms implemented by the program can be configured using a properties file. Each property entry has a format `<property name> = <value>`. Lines pre-fixed with `'#'` are comments. Property names are chosen to reflect both their domain and their meaning inside this domain. For instance, property name *“manber.min_char_frequency”* relates to Manber compression.

Property “domain” names and their meaning:

run	Configures how to gain statistics for statistics-gathering actions
manber	Relates to Manber's compression
lzss	Relates to LZSS compression
sk	Relates to Shapira/Klein's compression

Available properties:

Property name	Default value	Description
Search		
run.search.ptrn_cnt	100	The number of random search patterns on which to test the search speed of a given compressed matching method
run.search.ptrn_max_length	10	The maximum allowed length for a random search pattern
run.search.ptrn_min_length	4	The minimum allowed length for a random search pattern
Manber Compression		
manber.min_char_frequency	1	Characters with frequency below this one are considered 'rare' and will be encoded with 2 bytes
manber.partition_improvement_trials	100	The number of trials to improve the initial partition of characters into 2 groups
LZSS and Shapira/Klein's Compression		
lzss.max_text_window_size sk.max_text_window_size	32000	The maximum allowed LZSS text window size
lzss.bit_level.min_word_length lzss.byte_level.min_word_length sk.min_word_length	4 3 3	Minimum length to be used in (offset, length) pair. This is also the length of string prefix to be used as a hash key for hash-based implementation of a text window.

lzss.byte_level.max_bits_for_offset sk.byte_level.max_bits_for_offset	32	Encoding rules allowing the number of bits for an offset larger than this value will be cut off. Note that in order to take an effect, this value should be more restrictive than 'max_text_window_size', which also cuts off irrelevant rules. Large value has no effect since no rule will be cut off based on this value.
lzss.byte_level.max_bits_for_length sk.byte_level.max_bits_for_length	16	Encoding rules allowing the number of bits for a length larger than this value will be cut off. Large value has no effect since no rule will be cut off based on this value.
lzss.tw_impl.type sk.tw_impl.type	ON_HASH_AND_SUFFIX_TREE	The type of the text window implementation. Possible values: - ON_HASH: hash based implementation - ON_SUFFIX_TREE: suffix-trees based implementation - ON_HASH_AND_SUFFIX_TREE: implementation using both hash and suffix trees
lzss.tw_impl.hash.max_bucket_size sk.tw_impl.hash.max_bucket_size	5	For hash-based text window implementation: the size of a hash bucket. This is the number of words stored for the same prefix
lzss.tw_impl.hash.common_string_check_length sk.tw_impl.hash.common_string_check_length	10	For hash-based text window implementation: two matches are compared mostly by their matched length, the longer the better, but checking too many characters can be too slow. This value specifies how many characters will be actually checked. If both of compared matches have the same value of matched characters (but up to 'common_string_check_length' ones), the most recent match will be preferred.