Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 4355 – 4367

doi:10.3906/elk-1407-178

A new compression algorithm for fast text search

Aydın CARUS*, Altan MESUT
Computer Engineering Department, Trakya University, Edirne, Turkey

Received: 24.07.2014 • **Accepted/Published Online:** 04.08.2015 • **Final Version:** 20.06.2016

Abstract: We propose a new compression algorithm that compresses plain texts by using a dictionary-based model and a compressed string-matching approach that can be used with the compressed texts produced by this algorithm. The compression algorithm (CAFTS) can reduce the size of the texts to approximately 41% of their original sizes. The presented compressed string matching approach (SoCAFTS), which can be used with any of the known pattern matching algorithms, is compared with a powerful compressed string matching algorithm (ETDC) and a compressed string-matching tool (Lzgrep). Although the search speed of ETDC is very good in short patterns, it can only search for exact words and its compression performance differs from one natural language to another because of its word-based structure. Our experimental results show that SoCAFTS is a good solution when it is necessary to search for long patterns in a compressed document.

Key words: Compressed string matching, text compression, dictionary-based compression, exact pattern matching, CAFTS

1. Introduction

As the amount of text data increases day by day, the problems of storing such data and finding a string in these texts are becoming more important. The solution to the former is to compress the text data and the solution to the latter is to use an efficient string-matching algorithm. Some compressed texts can be searched directly without decompression and this process is generally called compressed pattern matching or compressed string matching. In recent years, many algorithms have been developed to solve the compressed pattern matching problem. A compressed pattern matching algorithm is generally specific to one compression algorithm, and cannot be used with another.

The term compressed string matching was first seen in Amir and Benson's work [1]. Manber offered a compression scheme that allows compressed pattern matching [2]. Although any string matching algorithm can be used with this scheme, its compression ratio and string matching speed are moderate. It was shown in [3] that searching simple patterns in texts that are compressed with a byte-oriented word-based Huffman coding algorithm is nearly 2 times faster than searching them in regular texts. Dense codes are statistical codes that use a word-based strategy, and they are efficient when searching for a single word in a compressed text. Two different semistatic dense codes, which were developed in 2003, are called the End-Tagged Dense Code (ETDC) [4] and (s,c)-Dense Code (SCDC) [5]. Subsequently developed dynamic counterparts of them, Dynamic ETDC (DETDC) and Dynamic SCDC (DSCDC), are better in compression speed, but worse in decompression speed [6]. The other two latest developed dense codes, Dynamic Lightweight ETDC (DLETDC)

*Correspondence: aydinc@trakya.edu.tr

and Dynamic Lightweight SCDC (DLSCDC), are able to perform decompression as fast as semistatic dense codes [7]. Although the compression speed of dynamic lightweight dense codes is worse than that of dynamic dense codes, it is still better than that of semistatic dense codes. On the other hand, dynamic lightweight dense codes are not better than dynamic dense codes, and dynamic dense codes are not better than semistatic dense codes in terms of compression ratio. Apart from these, Culpepper and Moffat described methods for searching in byte-coded compressed text and phrase-based searching in a restricted type of byte code [8].

Most of the compressed pattern matching studies deal with the problem of searching patterns in texts that are compressed using the two most well-known dictionary based compression methods, LZ77 [9] and LZ78 [10], or their variants like LZSS [11] and LZW [12]. Farach and Thorup suggested a pseudo-optimal compressed matching algorithm for LZ77 [13]. However, it cannot find all the occurrences of the pattern in the compressed text. The algorithm proposed in [14] finds only the first occurrence of a single pattern in LZW compressed text, while the algorithms proposed in [15,16] find all occurrences of multiple patterns by simulating the move of the Aho–Corasick pattern matching machine [17]. Furthermore, Navarro and Raffinot proposed a hybrid compression scheme that allows a search as fast as LZ78 and has a similar compression ratio to LZ77 [18]. Later, Navarro and Tarhio modified the Boyer–Moore string matching algorithm [19] so as to perform string matching over LZ78 and LZW compressed texts [20]. In addition, Klein and Shapira presented an adaptation of LZSS that is suitable for compressed matching [21].

It is also possible to perform pattern matching over texts that are compressed with an algorithm that uses an n-gram model. BPE [22] and Re-Pair [23] are the best-known examples of digram coding. The algorithm described in [24] is able to search BPE compressed texts. Moffat and Wan constructed a system for browsing and searching over texts compressed with a variant of the Re-Pair algorithm [25]. In one of our previous works [26], we devised a slightly different digram coding algorithm (Iterative Semi-Static Digram Coding: ISSDC). We think that texts that are compressed with ISSDC can be searched directly with a similar approach and this is a subject of our ongoing work.

In the current paper, we present a dictionary-based compression algorithm that uses a semistatic model. We name our algorithm CAFTS (Compression Algorithm for Fast Text Search), since texts that are compressed with CAFTS can be searched directly without decompression, which is generally faster than a search on uncompressed texts. The presented algorithm takes its origin from STECA (Static Text Compression Algorithm) [27], which is one of our previous works. The main difference between STECA and CAFTS is that while STECA uses static dictionaries CAFTS uses a semistatic approach. Since CAFTS has to build a dictionary for each source, its compression speed is slower but its compression ratio is higher and it is not language dependent. Another difference is that while CAFTS builds sub-dictionaries that contain the most frequently used trigrams that come after 'each different digram', STECA deals with only digrams/trigrams that come after 'each different character'. For this reason, the total dictionary size in STECA is relatively small and this contributes to its compression speed. Even though the compression ratio of CAFTS cannot be better than that of dynamic dictionary-based encoders like Gzip and Unix Compress, it can be better than semistatic and dynamic dense codes in some languages (see Section 4.1). We also present a compressed string matching approach that is used with texts that are compressed with CAFTS (SoCAFTS: Search on CAFTS). Although it is not possible to search for short patterns that contain less than 5 characters with SoCAFTS (the reason will be explained in the last paragraph of Section 3), the decompress-and-search technique on CAFTS (DSCAFTS) can be used as a fast solution to search for short patterns (see Section 4.2).

We organized the paper as follows: we explain the CAFTS compression algorithm in Section 2 and the SoCAFTS search method in Section 3. We give the compression ratio and compression/decompression time results of CAFTS, and also the compressed pattern matching time results of SoCAFTS in Section 4 with a comparison with other methods. Finally we conclude this paper in Section 5.

2. Compression algorithm for fast text search (CAFTS)

CAFTS performs compression by using a semistatic model and a dictionary that contains the most frequently used trigrams. The main dictionary consists of several sub-dictionaries, and each of them contains the most frequently used trigrams after a particular digram. The algorithm selects the corresponding sub-dictionary by looking at the last encoded digram. For example, if the trigram to be compressed is 'eir' and the last encoded digram is 'th', the algorithm searches 'eir' in the sub-dictionary of 'th'. The explanations of the alphabet and dictionary generation (first pass of the algorithm) and the compression method (second pass of the algorithm) are given in the next two subsections.

2.1. Alphabet and dictionary generation in CAFTS

In the first pass of CAFTS, the frequencies of the characters in the text are found and they are sorted in descending order of their frequency. The first σ characters having total frequencies above 99.95% are selected for the alphabet Σ. The selected σ characters are inserted into Σ in ASCII code order. The generated Σ with the most frequently used 60 characters that forms the 99.95% of the file 'dickens.txt' (an English text file, which is taken from http://introcs.cs.princeton.edu/data/) is given in Table 1 as an example. 'Σ Code = 0' is reserved for the escape character that is to be used when the current encoded character is not in Σ, like Q, U, V, X, and Z. These capital letters do not exist in Σ, because they are rarely used in this text file. We used this file as an English test file in Section 4 (Experimental results). If you look at Table 2, you can see that there are 91 different characters in the English file. This means that the total frequency of these 5 capital letters and 26 more characters is below 0.05%. The reason for not adding all of the characters used in the text to Σ is that the compression ratio will be affected negatively if Σ includes the characters that exist very rarely in the text. By eliminating these characters, we can add more trigrams to the sub-dictionaries (31 more trigrams for each sub-dictionary in our example). We choose this 0.05% value, because it gives the best compression ratio in our experiments. We also limited the size of the σ to 127 for the same reason.

After a code table is prepared for Σ, the different digrams and the trigrams placed after those digrams are searched in the text and the trigram dictionaries (sub-dictionaries) are prepared for each digram by using a trie data structure. Because CAFTS uses the dictionary size of 256 and one code must be used for the escape character, there are $\beta = 256 - \sigma - 1$ codes left to represent trigrams. Since $\sigma \leq 127$, β will be at least 128 (for our example in Table 1, $\beta = 256 - 60 - 1 = 195$). Therefore, a sub-dictionary for a digram can contain maximum β most frequently used trigrams. If the total number of the trigrams placed after a particular digram more than once is less than β, all of them are added to the sub-dictionary. The access method to the elements of a sub-dictionary is a binary search and in order to use this method the trigrams in sub-dictionaries are sorted in alphabetical order. The structures of the dictionary and the hash tables that are used to access the sub-dictionaries are shown in Figure 1.

As seen in Figure 1, there are 3600 (60^2) elements in both the 'sub-dictionary size hash table' and 'sub-dictionary address hash table'. However, because some digrams do not have any trigrams that are placed after them more than once (like 'hj' in our example), there are less than 3600 sub-dictionaries in the main dictionary.

It can also be observed in Figure 1 that some sub-dictionaries contain less than 195 trigrams. Therefore, the number of trigrams in the dictionary will be less than 195 times the number of sub-dictionaries. The 'σ value', 'Σ', 'size hash table', and 'dictionary' are added to a corresponding dictionary file. It is not necessary to add the 'address hash table', because its elements can be calculated by the help of the 'size hash table'.

Table 1. A sample Σ for 'dickens.txt' when $\sigma = 60$

Σ Code	ASCII Code	Character	Σ Code	ASCII Code	Character	Σ Code	ASCII Code	Character
1	10	Line Feed	21	72	H	41	103	G
2	32	Space	22	73	I	42	104	h
3	33	!	23	74	J	43	105	i
4	34	”	24	75	K	44	106	j
5	39	’	25	76	L	45	107	k
6	40	(	26	77	M	46	108	l
7	41	)	27	78	N	47	109	m
8	44	,	28	79	O	48	110	n
9	45	-	29	80	P	49	111	o
10	46	.	30	82	R	50	112	p
11	58	:	31	83	S	51	113	q
12	59	;	32	84	T	52	114	r
13	63	?	33	87	W	53	115	s
14	65	A	34	89	Y	54	116	t
15	66	B	35	97	a	55	117	u
16	67	C	36	98	b	56	118	v
17	68	D	37	99	c	57	119	w
18	69	E	38	100	d	58	120	x
19	70	F	39	101	e	59	121	y
20	71	G	40	102	f	60	122	z

Table 2. Statistical information about used text files in comparison.

File Name	The Number of Different:			The Most Frequently Used:								
	Chars	Digrams	Trigrams	Characters			Digrams			Trigrams		
Dutch	116	3436	31443	Space 16%	e 15%	n 8%	[en] 4.3%	[n] 4.0%	[e] 2.5%	[en] 2.7%	[de] 1.0%	[de] 0.9%
English	91	2548	25702	Space 17%	e 9%	t 7%	[e] 2.7%	[t] 2.3%	[th] 2.1%	[th] 1.5%	[the] 1.3%	[he] 1.1%
Finnish	118	3323	30313	Space 13%	a 9%	i 9%	[n] 3.2%	[a] 2.2%	[in] 1.9%	[en] 0.9%	[in] 0.8%	[an] 0.6%
French	129	3328	27793	Space 17%	e 12%	s 6%	[e] 4.1%	[s] 2.6%	[d] 2.1%	[de] 1.2%	[es] 1.0%	[de] 0.9%
German	128	4622	41910	Space 15%	e 13%	n 8%	[en] 3.1%	[er] 2.9%	[n] 2.7%	[en] 1.7%	[er] 1.3%	[de] 1.0%
Italian	138	3577	28117	Space 16%	e 9%	a 9%	[e] 3.1%	[a] 2.8%	[i] 2.4%	[di] 0.7%	[la] 0.6%	[co] 0.6%
Spain	134	3833	33601	Space 18%	e 10%	a 9%	[e] 2.8%	[a] 2.6%	[s] 2.1%	[de] 1.4%	[de] 1.0%	[os] 0.8%
Turkish	125	4650	46733	Space 13%	a 9%	e 7%	10,13 2.2%	[n] 1.8%	[ar] 1.4%	46,13,10 1.0%	32,32,32 0.8%	[lar] 0.6%
enwik8	205	18611	249172	Space 14%	e 8%	t 6%	[e] 2.0%	[t] 1.4%	[th] 1.4%	[th] 1.0%	[the] 0.9%	[he] 0.9%

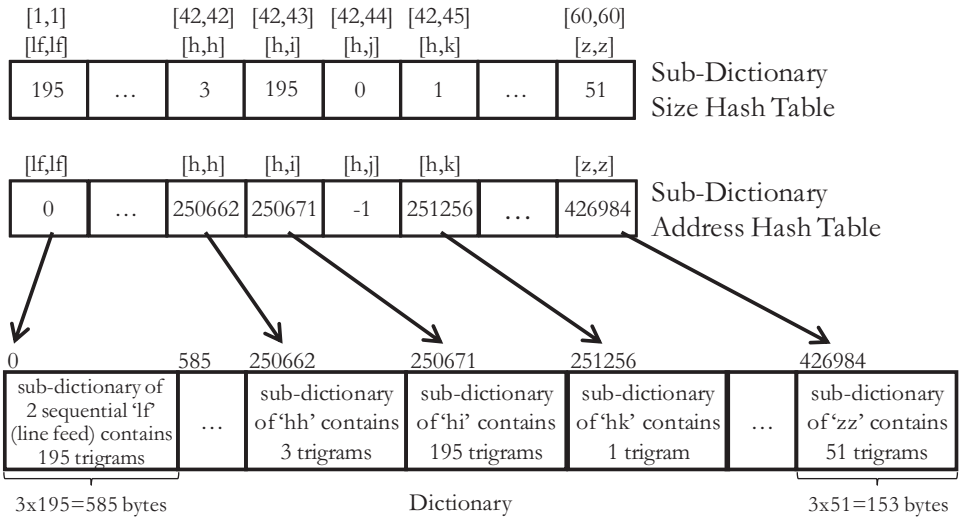

Figure 1. Dictionary and hash tables.

2.2. Compression method in CAFTS

CAFTS determines which sub-dictionary will be used by looking at the last encoded digram (LED) and combines the next three characters after this LED to form the searched trigram. After that, the sub-dictionary size of the LED is obtained from the 'size hash table'. If the sub-dictionary size is 0, it means that there is no sub-dictionary for that LED in the 'dictionary'. If it is not 0, the address of the beginning of that sub-dictionary is obtained from the 'address hash table', and the trigram is searched in that sub-dictionary with a binary search mechanism. If the trigram is found, the offset address of that trigram (the index number of the trigram in that sub-dictionary) $+\sigma$ is encoded and the last two characters in the trigram are assigned to the LED. If the trigram is not found in the corresponding sub-dictionary a shifting process is performed: the second character of the old LED becomes the first character of the new LED, the first character of the old trigram becomes the second character of the new LED, the last two characters of the old trigram become the first two characters of the new trigram, and a new character is read from the source for the last character of the new trigram. The algorithm continues in this fashion. The encoding algorithm is given in Figure 2 and an example is given in the following paragraph to clarify the encoding process.

Suppose that the text file to be compressed is 'dickens.txt' and we are now compressing the phrase "Oxford University" using the alphabet in Table 1. The algorithm reads two characters (O and x) and checks their existence in Σ. Because they are found in Σ, they are assigned to the LED and their Σ codes (28 and 58) are encoded. The algorithm reads the next three characters (f, o and r), which are also found in Σ. The trigram is formed with these three characters and this trigram searched in the sub-dictionary of 'Ox'. Since 'for' is the only trigram in the sub-dictionary of 'Ox', it is in the first index. After the σ value + index (60 + 1 = 61) is encoded, the second and the third characters of the trigram (o and r) are assigned to the LED. To form the new trigram, the next three characters are taken from the source (d, 'space', and U). Although the first two of them are found in Σ, the last one is not. Therefore, the Σ codes of 'd' and 'space' (38 and 2) are encoded, and to encode 'U', first the escape character (0) is encoded, and later the ASCII code of this character (85) is encoded. Two characters are taken from the source (n and i) to form the new LED and three characters are taken from the source (v, e, and r) to form the new trigram (they are all found in Σ). The Σ codes of 'n' and 'i' (48 and

```
Generate the Alphabet(Σ) and the Dictionary
LED = NULL, trigram = NULL
digram count = 1, trigram count = 1
While not end of source {
   chr = Read a character from source
   if chr exists in Σ {
      if digram count ≠ 2 {
         Σ code of chr is encoded
         LED[digram count] = chr
         digram count = digram count + 1
      }
      else {
         trigram[trigram count] = chr
         if trigram count is equal 3 {
            search the trigram in the sub-dictionary using binary search
            if the trigram is found {
               Sub-dictionary index + σ is encoded
               LED = trigram[2] & trigram[3]
            }
            else {
               trigram[1] is encoded
               LED = LED[2] & trigram[1]
               characters of the trigram are shifted from right to left
               trigram count = trigram count - 1
            }
         }
         trigram count = trigram count + 1
      }
   }
   else {
      Σ code of each character of the trigram are encoded individually
      LED = NULL, trigram = NULL
      digram count = 1, trigram count = 1
      Σ code of the escape character (0) is encoded
      ASCII code of chr is encoded
   }
}
```

Figure 2. CAFTS encoding algorithm.

43) are encoded and the trigram 'ver' is searched in the sub-dictionary of the digram 'ni'. The trigram is found in the 188th index of this sub-dictionary and 60 + 188 = 248 is encoded. The second and third characters of the trigram (e and r) are assigned to the LED and, to form the new trigram, the next three characters (s, i and t) are taken from the source. All of them are found in Σ and assigned to the trigram, but this trigram is not found in the sub-dictionary of 'er'. This time, the dictionary index of the first character of the trigram (53 for s) is encoded and all characters in the LED and trigram are shifted one position from right to left. A new character is read from the source (y) and assigned to the third character of the trigram. The trigram 'ity' is searched in the sub-dictionary of the digram 'rs' and again it is not found. The dictionary index of the first character of the trigram (43 for i) is encoded and, after another shifting process, the LED becomes 'si' and the trigram becomes 'ty'. This trigram is found in the 170th index of the sub-dictionary of 'si' and therefore 60 + 170 = 230 is encoded. The compressed form of the phrase "Oxford University" is shown in Figure 3.

Although the compression ratio of CAFTS in this example is 72% (13/18), the actual compression ratio is much better (see Section 4.1) as a consequence of the much rarely used escape mode. The time complexity of the CAFTS compression algorithm is $O(u \times log(k))$ in the worst case, where k is the sub-dictionary size and u is the text size, while the time complexity of the CAFTS decompression algorithm is $O(n)$, where n is the compressed text size.

	1	2	3	4	5	6	7	8		9	10	11	12	13	14	15	16	17	18
	O	x	f	o	r	d		U		n	i	v	e	r	s	i	t	y	
(a)	79	120	102	111	114	100	32	85		110	105	118	101	114	115	105	116	121	32
(c)	28	58	61			38	2	0	85	48	43	248			53	43	230		
	1	2	3			4	5	6	7	8	9	10			11	12	13		

Figure 3. The ASCII (a) and compressed (c) forms of "Oxford University".

3. Search on CAFTS compressed texts (SoCAFTS)

SoCAFTS has two extra phases: 'preprocessing' and 'postprocessing'. The searched pattern is compressed with maximum 3 different combinations using CAFTS in the preprocessing phase and the obtained compressed patterns are searched in the compressed text with any fast pattern matching algorithm in the searching phase. Whenever a match is found the postprocessing phase performs some partial decompression before the beginning and after the end of the matched part of the compressed text to verify that it is exactly the same as the searched pattern. SoCAFTS uses the same dictionary that is prepared for compression in both of these extra phases.

Because CAFTS uses trigram coding, a pattern may be compressed at most in 3 different forms according to the starting position of compression. We explain this situation with an example: suppose that we want to search for the word "compression" on the compressed form of 'dickens.txt'. Firstly, all of the different compressed forms of the searched pattern should be determined. This process is illustrated in Figure 4. In order to find all of the occurrences of the searched pattern in the text, the search procedure must be performed for all different compressed forms of the searched pattern. Note that in the Compressed Form 1 of Figure 4, if the trigram 'mpr' could not be found in the sub-dictionary of 'co', the compression procedure would try to search for 'pre' in the sub-dictionary of 'om'. In this situation, there would be no difference between Compressed Form 1 and Compressed Form 2, and because of that there would be no reason to use both of them in the searching phase. Similarly if 'pre' could not be found in the sub-dictionary of 'om', Compressed Form 2 and 3 will be the same. For this kind of situation, only Compressed Form 3 can be used to increase search speed.

Wherever a compressed form of the searched pattern is found in the compressed text, the postprocessing phase is initialized. This phase checks whether the uncompressed characters residing at the beginning and end of the pattern match the characters in the text or not. It requires partial decompression to perform this check. For example, while checking that if the character sequence in the text before 247 is 'com' or not in 'Compressed Form 2' of Figure 4, the algorithm may encounter a value that is bigger than 60. This means that the value represents a compressed trigram and the algorithm must know the corresponding sub-dictionary. If the two characters before this value are uncompressed characters (<60), the sub-dictionary can easily be found. If they are not, it is unable to know the content of the LED that comes before this value. For this reason the postprocessing phase searches for two consecutive uncompressed characters in the backward direction. When it finds them, it forms the LED and starts to perform partial decompression until it reaches characters before 247. After that, it can be checked whether the character sequence before 247 is equal to 'com' or not. If this verification is successful, then a similar process is performed for the last two characters of the searched pattern. If the pair after 237 is 'on' in the compressed text, the postprocessing phase ends with success.

The searched pattern that we have used in our example contains 11 characters. If the pattern contains less than 10 characters, the number of characters in the compressed pattern might be only one. It can be seen from Figure 4 that if the searched pattern was "compress", 231 in 'Compressed Form 1', 237 in 'Compressed Form 2', and 209 in 'Compressed Form 3' would not exist in the searching phase. If the compressed pattern contains only one character, the number of occurrences of this character in the text would be too much (most

Compressed Form 1
pre-processing phase:
LED = 'co', trigram = 'mpr'
'mpr' exists in the 72th index of the sub-dict. of 'co' (60 + 72 = 132 is encoded).
LED = 'pr', trigram = 'ess'
'ess' exists in the 55th index of the sub-dict. of 'pr' (60 + 55 = 115 is encoded).
LED = 'ss', trigram = 'ion'
'ion' exists in the 171th index of the sub-dict. of 'ss' (60 + 171 = 231 is encoded).
searching phase: (132, 115, 231) sequence is searched in the compressed text
wherever it is found:
post-processing phase: 'co' is searched before 132 for confirmation

Compressed Form 2
pre-processing phase:
LED = 'om', trigram = 'pre'
'pre' exists in the 187th index of the sub-dict. of 'om' (60 + 187 = 247 is encoded).
LED = 're', trigram = 'ssi'
'ssi' exists in the 177th index of the sub-dict. of 're' (60 + 177 = 237 is encoded).
LED = 'si', trigram = (There are not enough characters to form a trigram)
searching phase: (247, 237) sequence is searched in the compressed text
wherever it is found:
post-processing phase: 'com' is searched before 247 and 'on' is searched after 237 for confirmation

Compressed Form 3
pre-processing phase:
LED = 'mp', trigram = 'res'
'res' exists in the 140th index of the sub-dict. of 'mp' (60 + 140 = 200 is encoded).
LED = 'es', trigram = 'sio'
'sio' exists in the 149th index of the sub-dict. of 'es' (60 + 149 = 209 is encoded).
LED = 'io', trigram = (There are not enough characters to form a trigram)
searching phase: (200, 209) sequence is searched in the compressed text
wherever it is found:
post-processing phase: 'comp' is searched before 200 and 'n' is searched after 209 for confirmation

Figure 4. Searching for the word 'compression' in 3 different forms.

of them do not belong to our searched pattern). This situation greatly increases the number of unnecessary postprocessing phases and slows down searching. If the pattern length (m) is 6, 'Compressed Form 3' cannot be formed; if $m = 5$, 'Compressed Form 2' also cannot be formed; and if $m \leq 4$, none of them can be formed. Briefly, SoCAFTS cannot find any result when $m \leq 4$, it cannot find all the occurrences of the pattern when m is 5 or 6, and it can slowly find all the occurrences of the pattern when m is 7, 8, or 9. The time complexity of SoCAFTS is $O(n+rd)$, where r is the number of matches and d is the distance between the beginning of the match and two consecutive uncompressed characters in the backward direction.

4. Experimental results

In this section, we present the results of our compression ratio, compression speed, decompression speed, and string matching tests. We compared CAFTS with ETDC [4], SCDC [5], DLETDC [7], Unix Compress, and Gzip for compression and decompression performance. We use only DLETDC to represent dynamic dense codes, because it can be seen in [7] that the most successful of them in terms of compression and decompression time results is DLETDC.

Table 3. Compression ratios (in bits/character), compression and decompression times (in seconds).

	Gzip	Compress	DLETDC	ETDC	SCDC	CAFTS
Dutch	2.86	3.04	2.91	2.84	**2.80**	3.23
English	3.06	3.22	2.82	2.78	**2.70**	3.23
Finnish	**3.03**	3.25	4.12	4.02	3.99	3.36
French	**2.87**	3.11	3.15	3.09	3.04	3.18
German	**2.98**	3.24	3.40	3.32	3.28	3.45
Italian	**3.10**	3.24	3.29	3.21	3.17	3.29
Spanish	**2.93**	3.15	3.33	3.26	3.21	3.32
Turkish	**3.09**	3.43	4.16	4.08	4.01	3.44
Average ratio	**2.99**	3.21	3.40	3.33	3.28	3.31
Compression time	20.65	5.15	**4.83**	6.24	6.32	36.20
Decompression time	1.66	1.55	1.95	1.94	1.91	**1.26**

In our string matching tests, we compared SoCAFTS with ETDC and Lzgrep [20]. Lzgrep is a string matching tool that can search in compressed files that are compressed with Unix Compress (LZW) without decompression and can also search in compressed files that are compressed with Gzip (LZ77) with a fast decompress-and-search technique. In order to make a fair comparison with Lzgrep, we have implemented BM [19] and BOM [28,29] pattern matching algorithms to SoCAFTS, by modifying the program codes of Thierry Lecroq [30]. We have selected only ETDC as a representative of dense codes, because it had given the best results in [7] for single pattern matching.

We prepared a multilanguage corpus to show the difference in compression ratio when the source texts are written in different natural languages. In this corpus, there are 8 plain text files that are written in a different language, and each of them is exactly 15 MB (15.728.640 bytes) in size. The Turkish text file is a simplified version of *The Metu Corpus* [31]. Some characters and words belonging to XML format were eliminated to obtain plain text. The English text file is the first half of the 30 MB 'dickens.txt' file, which is taken from 'http://introcs.cs.princeton.edu/java/data/'. The Spanish, French, Italian, German, Dutch, and Finnish text files are all generated from free eBooks of Project Gutenberg (http://www.gutenberg.org). The total number of different *Characters*, *Digrams*, and *Trigrams* and the most frequently used 3 *Characters*, *Digrams*, and *Trigrams* in these files are given in Table 2.

The computer that we used in our comparison tests has an Intel Core i5-2450M (2.5 GHz) CPU and 6 GB DDR3 (1333 MHz) main memory. The operating system of this computer was Linux (Ubuntu) having kernel version *3.0.0-17-generic*. All of the C codes used in these tests were compiled with *GCC ver. 4.6.1* (optimization parameter: *-O9*). The time results were obtained using the Linux 'command time' command.

4.1. Compression and decompression performance of CAFTS

The compression ratios for each test file and the total compression and decompression times for the whole test corpus are given in Table 3. Gzip was used with '-9' parameter for the best compression ratio. Note that the compression ratio results of ETDC, SCDC, and CAFTS are the ratios of the sum of their compressed files and semistatic dictionary files.

As seen in Table 3, Gzip is the best algorithm in terms of compression ratio, but its situation is slightly different, because compressed string matching cannot be done in Gzip compressed files (Lzgrep can perform decompress-and-search over Gzip compressed files). SCDC is always better than ETDC and ETDC is always

better than DLETDC in terms of compression ratio (as expected because of the results found in [7]). SCDC has the best compression ratios in English and Dutch, while ETDC takes second place in these languages. However, it is also seen in Table 3 that these two compression methods are not successful in Finnish and Turkish. Because Finnish and Turkish are agglutinative languages, the numbers of different words in these languages are more than those of the other languages, and it is probably the reason why word-based compression methods like dense codes are not successful in these languages. As a side note, it is seen in Table 2 that the space character is used less in these two languages (13%) and this shows that the average word lengths in these languages are greater than those of the other languages.

Although DLETDC is the best method in terms of compression time, decompression time is a more important parameter in decompress-and-search methods, and dynamic dense codes are not better than their semistatic counterparts in terms of this parameter. Compression speed is the weakness of CAFTS, but as seen in Table 3 CAFTS is powerful in terms of decompression speed.

4.2. String matching performance of SoCAFTS

In order to obtain more accurate time results, a larger text file (*enwik8*: the first 100 MB of the XML text dump of the English version of Wikipedia, which is taken from http://www.cs.fit.edu/~mmahoney/compression/textdata.html) was used in string matching tests. For each pattern length (m = 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, and 50), approximately 50 patterns were chosen randomly from *enwik8*. The average search times of these patterns are given in Figure 5.

Lzgrep was used with 6 different search modes in our tests: D77-BM, D77-BOM, DW-BM, DW-BOM, BM-simple, and BM-blocks. The first four of them are decompress-and-search techniques: 'D77' represents decompression with LZ77 (Gzip) and 'DW' represents decompression with LZW (Unix Compress). The other two are direct searches with BM algorithm. We have chosen these 6 parameters because they had given the best results in [20]. We have also given the results of the decompress-and-search technique that is used with CAFTS as DSCAFTS.

It is seen in Figure 5 that ETDC performs best when pattern length is less than or equal to 15 and SoCAFTS BM performs best when pattern length is greater than 15. Because the current implementation of the ETDC algorithm in the Dense Codes website (http://vios.dc.fi.udc.es/codes/) can only search for exact words, we cannot use it with long patterns (m >20). It is also seen that SoCAFTS BOM is better than SoCAFTS BM when $m \leq 15$. As a result of the high decompression speed of CAFTS, the search speed of DSCAFTS is always better than the speed of the decompress-and-search of Lzgrep (D77-BM, D77-BOM, DW-BM, and DW-BOM) and close to the speed of the direct search of Lzgrep (BM-blocks and BM-simple).

Figure 5 shows clearly that the search time of SoCAFTS steadily increases while the pattern length decreases (the reason is explained in the last paragraph of Section 3). It is also explained that SoCAFTS cannot find all the occurrences of the searched pattern in the compressed text when the pattern length is 5 or 6 (for this reason these results are shown with dashed lines). It can find nearly 1/3 of them when $m = 5$ (only one compressed form is searched) and nearly 2/3 of them when $m = 6$ (two compressed forms are searched). If the search method is not determined by a parameter, our implementation code automatically selects DSCAFTS when $m < 7$.

5. Conclusion

The presented compression algorithm, which is named CAFTS, compresses plain texts approximately 41% of their original sizes (3.31 bpc $\cong$ 41%) by using multiple sub-dictionaries. Using multiple dictionaries instead of a

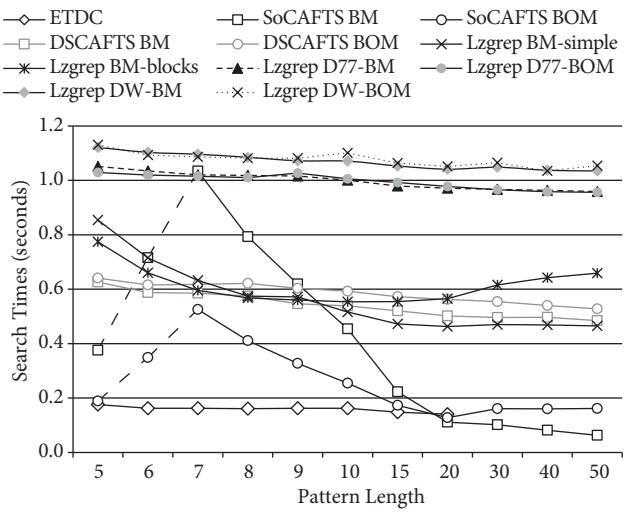

Figure 5. Average search times (in seconds).

single dictionary provides a better compression ratio, because this approach increases the finding probability of digrams and trigrams in the dictionary. Although this structure causes slow encoding, the ability of performing string matching without decompression makes this algorithm useful. We think that proposed technique can be used in document matching in compressed collections.

Like most of the other compressed string matching methods, SoCAFTS is also able to use any kind of string matching algorithm. With some implementations like Lzgrep, compressed string matching can be done on files that are compressed with a dynamic dictionary-based compression technique. However, Section 4.2 shows that the search speed of Lzgrep is much slower than that of ETDC and SoCAFTS. If a single word is to be searched for in a compressed document, semistatic dense codes (ETDC and SCDC) and dynamic dense codes (DETDC, DSCDC, DLETDC, and DLSCDC) are very good solutions. However, since their implementations on the Internet are not able to search for consecutive words, we think that SoCAFTS will be a good alternative when it is necessary to search for long patterns in compressed large text databases or encyclopedias. For example, it can be used when the "President of the United States" pattern needs to be searched for in the compressed enwik8 file (this pattern includes 30 characters and repeats 383 times in this file).

Since some words may be very long in agglutinative languages like Turkish and Finnish, when a user wants to search for a part of a long word, the current implementations of dense codes cannot be used (they can only search for an exact word). In this kind of situation, if the part of a word that will be searched for is larger than 7 characters SoCAFTS can be used; otherwise DSCAFTS can be used.

References

[1] Amir A, Benson G. Efficient two-dimensional compressed matching. In: Data Compression Conference; 24–27 March 1992; Snowbird, Utah, USA. Los Alamitos, CA, USA: IEEE. pp. 279-288.

[2] Manber U. A text compression scheme that allows fast searching directly in the compressed file. ACM T Inform Syst 1997; 15: 124-136.

[3] Moura ES, Navarro G, Ziviani N, Baeza-Yates R. Fast and flexible word searching on compressed text. ACM T Inform Syst 2000; 18: 113-139.

[4] Brisaboa NR, Iglesias ER, Navarro G, Paramá JR. An efficient compression code for text databases. In: European Conference on IR Research; 14–16 April 2003; Pisa, Italy. LNCS 2633, Berlin, Germany: Springer-Verlag. pp. 468-481.

[5] Brisaboa NR, Fariña A, Navarro G, Esteller MF. (S,C)-Dense Coding: an optimized compression code for natural language text databases. In: Symposium on String Processing and Information Retrieval; 8–10 October 2003; Manaus, Brazil. LNCS 2857, Berlin, Germany: Springer-Verlag. pp. 122-136.

[6] Brisaboa NR, Fariña A, Navarro G, Paramá JR. New adaptive compressors for natural language text. Software Pract Exper 2008; 38: 1429-1450.

[7] Brisaboa NR, Fariña A, Navarro G, Paramá JR. Dynamic lightweight text compression. ACM T Inform Syst 2010; 28: 1-32.

[8] Culpepper JS, Moffat A. Phrase-based pattern matching in compressed text. In: Symposium on String Processing and Information Retrieval; 11–13 October 2006; Glasgow, Scotland, UK. LNCS 4209, Berlin, Germany: Springer-Verlag. pp. 337-345.

[9] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE T Inform Theory 1977; 23: 337-343.

[10] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. IEEE Inform Theory 1978; 24: 530-536.

[11] Storer JA, Szymanski TG. Data compression via textual substitution. J ACM 1982; 29: 928-951.

[12] Welch TA. Technique for high-performance data compression. IEEE Computer 1984; 17: 8-19.

[13] Farach M, Thorup M. String matching in LempelZiv compressed strings. In: ACM Symposium on Theory of Computing; 29 May–1 June 1995; Las Vegas, NV, USA. New York, NY, USA: ACM. pp. 703-712.

[14] Amir A, Benson G, Farach M. Let sleeping files lie: pattern matching in Z-compressed files. J Comput Syst Sci 1996; 52: 299-307.

[15] Kida T, Takeda M, Shinohara A, Miyazaki M, Arikawa S. Multiple pattern matching in LZW compressed text. In: Data Compression Conference; 30 March–1 April 1998; Snowbird, Utah, USA. Los Alamitos, CA, USA: IEEE. pp. 103-112.

[16] Tao T, Mukherjee A. Pattern matching in LZW compressed files. IEEE T Comput 2005; 54: 929-938.

[17] Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM 1975; 18: 333-340.

[18] Navarro G, Raffinot M. A general practical approach to pattern matching over Ziv-Lempel compressed text. In: Combinatorial Pattern Matching; 22–24 July 1999; Warwick University, UK. LNCS 1645, Berlin, Germany: Springer-Verlag. pp. 14-36.

[19] Boyer RS, Moore JS. A fast string searching algorithm. Commun ACM 1977; 20: 762-772.

[20] Navarro G, Tarhio J. LZgrep: A Boyer–Moore string matching tool for Ziv–Lempel compressed text. Software Pract Exper 2005; 35: 1107-1130.

[21] Klein ST, Shapira D. A new compression method for compressed matching. In: Data Compression Conference; 28–30 March 2000; Snowbird, Utah, USA. Los Alamitos, CA, USA: IEEE. pp. 400-409.

[22] Gage P. A new algorithm for data compression. C Users Journal 1994; 12: 23-28.

[23] Larsson NJ, Moffat A. Offline dictionary-based compression. P IEEE 2000; 88: 1722-1732.

[24] Shibata Y, Kida T, Fukamachi S, Takeda M, Shinohara A, Shinohara T, Arikawa S. Byte pair encoding: a text compression scheme that accelerates pattern matching. Technical Report DOI-TR-161; 1999; Dept. of Informatics, Kyushu University.

[25] Moffat A, Wan R. Re-Store: A system for compressing, browsing, and searching large documents. In: 8th International Symposium on String Processing and Information Retrieval; 13–15 November 2001; Laguna de San Rafael, Chile. Los Alamitos, CA, USA: IEEE. pp. 162-174.

[26] Mesut A, Carus A. ISSDC: Digram coding based lossless data compression algorithm. Comput Inform 2010; 29: 741-756.

[27] Carus A, Mesut A. Fast text compression using multiple static dictionaries. Information Technology Journal 2010; 9: 1013-1021.

[28] Allauzen C, Crochemore M, Raffinot M. Factor oracle: a new structure for pattern matching. In: 26th Conference on Current Trends in Theory and Practice of Informatics; 27 November–4 December 1999; Milovy, Czech Republic. LNCS 1725, Berlin, Germany: Springer-Verlag. pp. 295-310.

[29] Faro S, Lecroq T. Efficient variants of the Backward-Oracle-Matching algorithm. Int J Found Comput S 2009; 20: 967-984.

[30] Charras C, Lecroq T. Handbook of Exact String-Matching Algorithms. London, UK: King's College Publications. 2004.

[31] Say B, Zeyrek D, Oflazer K, Ozge U. Development of a corpus and a treebank for present-day written Turkish. In: Eleventh International Conference of Turkish Linguistics; 7–9 August 2002; Gazimağusa, KKTC. Ankara, Turkey: METU. pp. 183-192.