Speeding up Natural Language Text Search using Compression

Majed AbuSafiya
Software Engineering Department
Al-Ahliyya Amman University
Amman, Jordan

***Abstract*—Text search is a well-known problem in computer science where the valid shifts of a pattern *P* in a text string *T* are found. This paper shows how to speed up text search by searching for *P* in a compressed version of *T*. A fast compression algorithm was designed for this aim. This algorithm is based on the assumption that *T* is restricted to the letters of a single natural language. Relying on this assumption, a letter, in *T* or *P*, is encoded into a single byte instead of the two-byte unicode which shortens the string on which a text search algorithm works. The main disadvantage of this approach is the restriction of the alphabet of *T* to be from a single natural language. However, wide range of text documents comply to this assumption. Another issue is the overhead that is required to compress *P* and *T*, but it was found that the proposed compression algorithm is so fast such that its run-time can be paid for and still save text search time. Different approaches to store compressed *T* are also explored. The conducted experimental study showed that this approach does actually reduce the text search time.**

***Keywords*—Text compression; text search; unicode**

I. Introduction

A lot of research was directed towards searching in compressed text. A survey of the approaches to search in compressed text without decompression can be found in [1]. In [2], text search was applied on a directory-based compressed text. In [3], the characters are encoded as a variable-length sequences of base symbols of fixed number of bits. In [4], the input text is already compressed with Lempel-Ziv. In [5], a compression and decompression techniques for natural language text are proposed. The compression scheme that is used is based on semi-static word-based model and Huffman encoding where the coded alphabet is byte oriented rather than bit-oriented. In [6], an approximate search on the compressed search using local decompression is proposed. In [7] the input text is assumed to be Ziv-Lempel Compressed Text. In [8], the text search in compressed text is done using periodicity analysis, with sublinear run time with the size of compressed text. In [9], a directory based compression is used on natural language text.

The main observation that can be noticed in the previous research regarding this problem is that: the primary motivation was to do text search in an input string that is already compressed using known compression algorithms without decompressing it first. This means that compression was not originally done to speed text search. This is the main point that contrasts this work from others work. In this paper, compression is done to speed up the text search first and to save space as a second gain. The compression that was considered in literature is based on known compression algorithms which are known to be complex and time-consuming. On the other hand, the proposed compression is very fast and simple. A similar approach to our approach was found in [10]. However this work differs from our work in many ways: (1) *T* is assumed to be in ASCII while our work is based on unicode encoding, (2) our compression approach is much faster and simpler, (3) the shifts that are found in the compressed *T* can easily be translated to shifts in the original uncoded version of *T*.

The proposed work in this paper is based on the fact that *T* is encoded in unicode [11]. Unicode is an international standard for encoding alphabets of natural languages, two bytes for a letter. Alphabets of different natural languages are encoded in ranges. One observation about unicode is that the alphabet of the same natural language share the same upper byte value. For example, Arabic alphabet unicodes range between 0x0600 up to 0x6FF with the same upper byte code 0x60. This fact will be utilized to compress *T* and *P* to reduce their length to half by excluding the upper byte. For example, the Arabic word (هو) is composed of two letters (Fig. 1). The unicode of letter (هـ) is 0x0647, the upper byte is 0x06 and the lower byte is 0x47. The unicode of the second letter (و) is 0x0648 with upper byte is 0x06 and lower byte is 0x48. The proposed compression is based on using only the lower byte as a code for the letter. This will compress *T* into half size. Note that this compression works only under the assumption that *T* contains only text letters of the same alphabet. Moreover, the code of the first letter (هـ) is placed in the upper byte in the compressed unicode. This is because Arabic script is written from right to left and hence it assures that the encoded letters will be stored in the same order that they have within the original text.

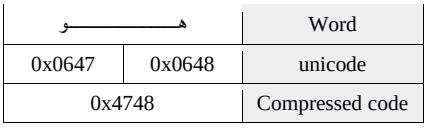

Fig. 1. Compression of Two Letters into One Letter.

II. Compression Algorithm

A. COMPRESS Algorithm

COMPRESS algorithm (Fig. 2) returns a compressed string (*Scompressed*) for an input unicode string *S*. *S* could be *T* or *P*.

Scompressed will be half the length of *S*. *S* is a string, where each letter is encoded with two-byte unicode. *S compressed* is generated by reducing the two-byte unicode code of every letter in *S* into a single byte in *S compressed* (Fig. 1). To show how this algorithm works for the example in Fig. 1, let *S* be the string (هو) with two letters S_0 = (هـ) and S_1=(و). The algorithm will do a left-shift on S_0 by eight bits to generate S'_0 (0x4700). Next, a bit-wise *and* operation is applied between S_1 and 0x00FF to generate S'_1 (0x0048). Finally, a bit-wise *or* operation is applied between S'_0 and S'_1 which will result in one letter unicode (i.e. 0x4748) that will be appended to *S compressed*. The loop will iterate for the letters of *S* in pairs until *S compressed* is complete. In case of *S* is of odd length, a *space* will be appended to make its length even.

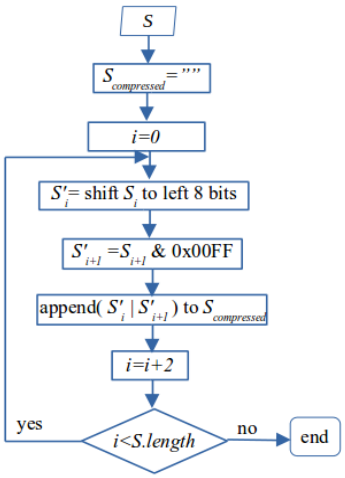

Fig. 2. COMPRESS Algorithm.

B. Saving T compressed to a file

Saving *T compressed* into a file has two advantages: (1) saving disk space, if it replaces the original *T*'s text file since the compressed version is half the size of the *T*'s text file, (2) allowing immediate application of the text search on the compressed version of *T* saves the time to compress *T* every time a text search is required. It is important to point here that the experimental study showed reduction is search time even if *T* is input in its native uncompressed format and compression is done as part of the text search. To save *T* in a compressed form, the text file of *T* is read and COMPRESS algorithm is called to build *T compressed*. Remember that *T compressed* is an array of bytes, one byte encodes one letter in *T*. There are two approaches to write *T compressed*: as a unicode text file or as a binary file.

One way to store *T compressed* is through storing it in a text file using unicode. In this case, every *pair* of bytes of *T compressed* is interpreted as a single unicode code character and the corresponding character of this unicode is written to the file. So, the size of this file will be half the size of the original *T's* file. In addition to compression, it will be encrypted. For example, *T*= "بسم الله الرحمن الرحيم", which is composed of twenty two characters (letters and spaces), will be encoded into eleven unicode characters. Fig. 3 shows how *T compressed* looks like when its file is opened in a text editor.

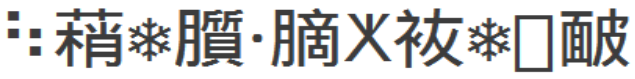

Fig. 3. *T compressed* as Unicode Text.

T compressed may also be stored as binary file (Fig. 4). The letters of *T compressed* are written into a binary file a byte by byte without building unicode letters from pairs of bytes. In this case, the letter is represented as ASCII code. For example, for *T*="بسم الله الرحمن الرحيم", each character is represented by one byte. This byte corresponds to the lower byte of the unicode of that character. For example the first byte is '(' which has the hex value 0x28 is the code of the first letter (بـ) . Note that the unicode for letter (بـ) is 0x0628.

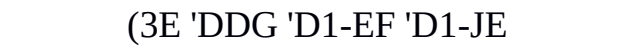

Fig. 4. *T compressed* Stored as Binary.

III. Speeding Text Search with Compression

The compressed text search algorithm is shown in Fig. 5. It takes as input *P* and *T*, compresses them using COMPRESS algorithm and then calls any known string matching algorithm to search for *P compressed* within *T compressed*. Although *T compressed* has half the size of *T*, the length of *T compressed* equals to the length of *T*. This is because *T compressed* is viewed as an array of bytes (on byte for each letter) and *T* is viewed as an array of unicodes. This is also true for *P* and *P compressed*. The equality comparisons within the selected string matching algorithm will be byte-wise comparisons and not unicodes comparisons. So, the calculated valid shifts that are found by the this reused string matching algorithm will be the valid shifts of *P* within *T*.

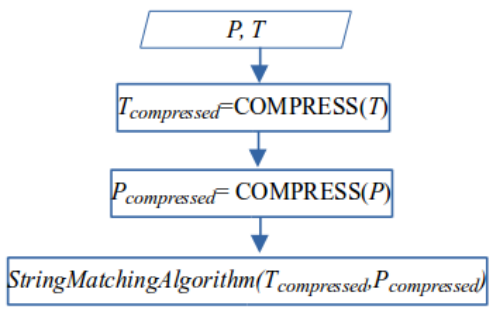

Fig. 5. Compressed Text Search Algorithm.

IV. Experimental Study

The compressed text search algorithm was implemented in Java, where *T* is chosen to be the text of the Holy Quran, which is composed only of Arabic letters, with size of 411,082 letters. The selected string matching algorithm was the known Knuth–Morris–Pratt (KMP) algorithm [12]. To show the reduction in text search time, the search time that is needed to search for *P* in *T* using the compressed text search algorithm is compared with the time that is needed to search for *P* in *T* without compression (Fig. 6). *P* was randomly chosen as a substring of a given length from *T*. This experiment was repeated for varying sizes of *P*. Note that the time to compress *P* and *T* was included in calculating the search time for the compressed text search algorithm. To raise the confidence in the results, this process was repeated 1000 times for each length of *P* and the average time was recorded for both algorithms. It is obvious that when KMP is applied on compressed input, it resulted in significant reduction in search time. The saving in time happened because an equality comparison between two unicode characters, in Java, is actually implemented through a couple of byte-wise equality comparisons. On the other hand, when searching in $T_{compressed}$, the equality comparison is done by a single byte-wise comparison.

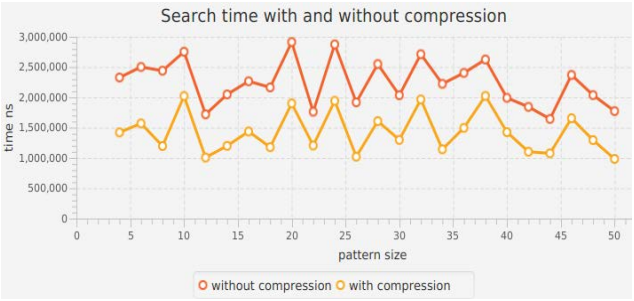

Fig. 6. Comparison between KMP Time with/without Compression of *P* and *T*.

V. Conclusions

In this paper, the natural language text was compressed to speed up text search. By excluding the upper byte of the unicode of letters, we could reduce the size of both *P* and *T* into half and hence have a faster text search. This approach assumes that letters of the text belong to the alphabet of the same natural language. One important result from this research is that exploiting the specifics and constraints of natural languages may open the door for improvements on string algorithms in general. Although these improvements are not generic, they may be useful under certain contexts. One interesting issue to explore is how to do text search when $T_{compressed}$ and $P_{compressed}$ are viewed as arrays of unicodes rather than arrays of bytes. The challenge here is to explore how to compress *P* such that the odd valid shifts of *P* within *T* are also found.

Acknowledgment

All praise and gratitude be to *Allah*, all mighty, for guiding me and giving me the knowledge and strength to accomplish this work.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee, Pattern Matching in Compressed Texts and Images. Now Publishers Inc., Hanover, MA, 2013.

[2] K. Fredriksson and S. Grabowski, “A general compression algorithm that supports fast searching,” Information Processing Letters, vol. 100, 2006, pp.226-232.

[3] J. Rautio, J. Tanninen and J. Tarhio, "String matching with stopper compression," Proceedings DCC 2002, Data Compression Conference, Snowbird, UT, USA, 2002, pp. 469-476.

[4] M. Farach-Colton and M, Thorup, “String Matching in Lempel—Ziv Compressed Strings,” Algorithmica, vol.20 , 1998, pp. 388-404.

[5] E. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, “Fast and flexible word searching on compressed text,” ACM Trans. Inf. Syst. Vol. 18, 2000, pp. 113–139.

[6] G. Navarro, T. Kida, M. Takeda, A. Shinohara and S. Arikawa, "Faster approximate string matching over compressed text," Proceedings DCC 2001. Data Compression Conference, Snowbird, UT, USA, 2001, pp. 459-468.

[7] G. Navarro and J. Tarhio, “Boyer-Moore String Matching over Ziv-Lempel Compressed Text,” Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (COM '00). Springer-Verlag, Berlin, Heidelberg, pp. 166–180, 2000.

[8] A. Amir and G. Benson, “Two-dimensional periodicity and its applications,” In Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA '92). Society for Industrial and Applied Mathematics, USA, 1992, pp. 440–452.

[9] K. Fredriksson and F. Nikitin, “Simple Compression Code Supporting Random Access and Fast String Matching,” In: Demetrescu C. (eds) Experimental Algorithms. WEA 2007. Lecture Notes in Computer Science, vol 4525. Springer, Berlin, Heidelberg.

[10] Udi Manber, “A text compression scheme that allows fast searching directly in the compressed file,” ACM Trans. Inf. Syst. 15, pp. 124–136, 1997.[12] D. Knuth; J. Morris, V. Pratt, “Fast pattern matching in strings,” SIAM Journal on Computing vol.6, 1977, pp. 323-350.

[11] Unicode home, http://home.unicode.org.

[12] D. Knuth; J. Morris, V. Pratt, “Fast pattern matching in strings,” SIAM Journal on Computing vol.6, 1977, pp. 323-350.