WORDMATCH: WORD BASED STRING MATCHING OVER COMPRESSED TEXTS

Aydın CARUS
aydinc@trakya.edu.tr
Computer Engineering Department
Trakya University – Edirne / TURKEY

H. Nusret BULUŞ
nusretb@trakya.edu.tr
Computer Engineering Department
Trakya University – Edirne / TURKEY

Altan MESUT
altanmesut@trakya.edu.tr
Computer Engineering Department
Trakya University – Edirne / TURKEY

Abstract

In this study, Word Based String Matching Algorithm named as WordMatch is represented. This algorithm makes pattern matching over compressed natural language text documents with word based text compression algorithm that is developed by us. Since the compression algorithm based on words, only word based string matching can be done on compressed text documents. WordMatch uses the same static dictionaries with our word based text compression algorithm. The results of pattern matching with WordMatch on compressed text documents and the results of pattern matching on the same text document which is in uncompressed form are compared. It is seen that the results taken from *WordMatch Algorithm is as efficient as the results of other pattern matching algorithms.*

Key Words: String Matching, Data Compression, Compressed Documents, Word Based.

INTRODUCTION

String matching is the process of finding the exact locations that the pattern of length n is included in the text of length m. Many algorithms are developed in order to make pattern matching. These algorithms start the matching process from the beginning of the pattern, from the end of the pattern, from the middle the pattern or from any point of the pattern and go through different directions. The main idea of starting from the beginning, from the end or from any point of the text which are probably in the same sequence of the pattern is making less attempts. Pattern matching on compressed text is widely used like pattern matching on normal text. Pattern matching on compressed text studies that are LZW based, Word based and byte oriented compression algorithms have been represented [1,2,3,4]. In addition to those, it is possible to use pattern matching algorithms on compressed text which are compressed by Word Based Huffman algorithm [5].

Especially in recent years pattern matching on large compressed text files becomes very popular. It is obviously seen from the previous studies that the performance of pattern matching in compressed text is better than doing it after decompressing. The WordMatch Algorithm that we present in this study allows word based pattern matching in word based compressed files.

The next section explains the word-based compression algorithm. The following section explains The WordMatch Algorithm and after that brief descriptions of the other pattern matching algorithms are given. The following section includes the test results of WordMatch and other pattern matching algorithms. Conclusion is given in the last chapter.

WORD BASED COMPRESSION

Word based lossless data compression algorithm includes a static dictionary of the most frequent words in texts that are written in English. The words are grouped by their number of the letters and stored in the dictionary. 2 bytes

are coded for each word in compression process. The first coded byte indicates the block of the word and the second coded byte is the index of the word in this block. In the compression process, the first word of the text which will be compressed is taken firstly and the space character is added to the word, then the word is searched in the dictionary. If the word is found in the dictionary, the block number and the index in this block are coded. If the word is not found in the dictionary, the space character is dropped and the word is written exactly between two escape characters. The compression test results of this algorithm shows that compression ratio of this algorithm is approximately 50%.

WORD BASED PATTERN MATCHING ALGORITHM

Word based pattern matching using WordMatch Algorithm is possible on texts that are compressed by the algorithm described above. The matching algorithm is given in Figure 1. While the word based pattern matching is being done, first the pattern is searched in the static dictionaries as it is done in the compression algorithm. If the pattern is included in the dictionary, the block number and the index number is calculated like it is done in the compression algorithm. After that, the first byte in the compressed data set is read. If the byte is an escape character, we move until the second escape character is seen. If the character is not an escape character, then this byte indicates the block of the word coded by 2 bytes. In this case this byte is compared with the block number of the pattern. In addition to this, the index of the pattern is compared with the following byte. If these two bytes are same, then a location which pattern occurs in the text is found. After that, appropriate moves are done and so whole text is searched. If the pattern is a word that is not included in the dictionary, then the first comparison process is done between the characters that are following the escape character and the character in the same sequence in the pattern. This is done until the second escape character which shows the end of the word is read. In this way, compressed and uncompressed words can be controlled and the exact locations of the pattern in the text can be determined.

OTHER USED PATTERN MATCHING ALGORTIHMS

- **Raita Algorithm:** This algorithm starts comparison from the last character of the pattern. If they are same then it compares the first character of the pattern with the character in the same location of the text. If the second match occurs it compares the character in the middle. If the matching still continues, it makes the comparison from the second character to the end [6].
- **Boyer-Moore Algorithm:** It has two pre-processing phases which are bad character shifting and good suffix shifting. It compares characters from right to left. In the searching phase, the moving step is the maximum shifting values calculated in the pre-processing phase [7].
- **Berry-Ravindran Algorithm:** It has a pre-processing phase. It uses shift values taken from a rule like bad character rule of Boyer-Moore algorithm [8].
- **Backward Oracle Algorithm:** This algorithm that uses last situation automats takes the pattern backwards. On this reverse pattern least suffix privilege x^R is found in the pre-processing phase. In searching phase, algorithm divides the characters in the window from right to left by $\boldsymbol{O}(x^R)$ automat as *q0* will be the initial state. This state continues until no more definite transition left from the valid state of automat to the valid character of the window. At this point the longest prefix of the pattern is the suffix of the part shaded on the text. By using this information the shift values on the automat is calculated [9].
- **Apostolico Giancarlo Algorithm:** Differently from the Boyer-Moore algorithm, at the end of each try, it remembers the longest suffix of the pattern in the right end of the window. This information is kept in the table named *skip.* In addition to this, algorithm uses the bad character and good suffix pre-processing phases of Boyer-Moore Algorithm. Again searching process is done from right to left. In the case of mismatch, it uses good suffix, bad character and skip values differently from Boyer-Moore Algorithm [10].

```
search the pattern in the dictionary
if pattern exists in word_based_dictionary{
  compute block_number and index for pattern
  while(not end of compressed text){
    if(text[n] equals escape character){
      n←n+1
      while(text[n]not equals to escape character)
      {
        n←n+1
      }
      n←n+1
    }
    else if(text[n]equals to block_number and text[n+1] equals to index){
     show the position
     n←n+2
    }
    else{
      n←n+2
    }
  }
}

if pattern absent in word_based_dictionary{
  while(not end of compressed text){
  if(text[n] equals to escape character){
    chr_count←0
    pattern_pointer←0
    n←n+1
    while(text[n] not equals to Escape character){
      if(text[n] equals to pattern[pattern_pointer]){
        chr_count←+1
        pattern_pointer← pattern_pointer+1
      }
      n←n+1
    }
    if(lenght of pattern equals to chr_count){
      show the position
      n←n+1
    }
    else{
      n←n+2
    }
  }
}
```

Figure 1. WordMatch Algorithm

PATTERN MATCHING TESTS AND RESULTS

The programs of WordMatch and the other algorithms are written in C programming language in .NET 2005 platform and compiled in *Release* mode. The C code of other pattern Matching algorithms that are used in our comparison is developed by Christian Charras and Thierry Lecroq **[**11**].** A corpus about 30 MB including stories, novels and texts in English in different subjects is prepared for matching. The corpus is compressed by using word based compression algorithm to make the searching process with WordMatch algorithm. This test is performed with searching 100 words with WordMatch in compressed corpus and with the other matching algorithms in uncompressed corpus. The values on the Figure 2 shows the search time for 100 patterns.

According to Figure 2 which gives us the matching times of WordMatch algorithm in compressed corpus and the other algorithms in original corpus;

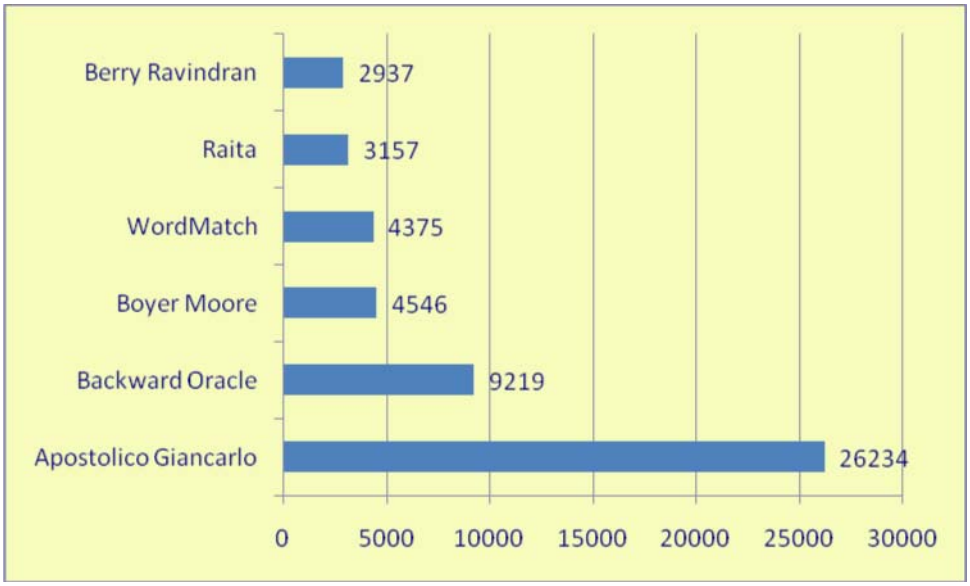

Figure 2. Search time results for 100 patterns (Milisecond)

- It is seen that WordMatch on compressed text has a performance that is two times better than the performance of Backward Oracle Algorithm and six times better than Apostolico Giancarlo Algorithm.
- Berry Ravindran algorithm has better performance than WordMatch about 33%. Raita algorithm has better than WordMatch about 27 % and Boyer-Moore algorithm has worse than WordMatch about 1 %.

CONCLUSION

The WordMatch Algorithm that can search words over texts which are compressed by our word-based compression algorithm is presented in this paper. If a string matching algorithm can make string matching over compressed texts it has a time advantage since decompression is not necessary. Therefore if a string is searched in a compressed text, WordMatch algorithm can be better than all algorithms in Figure 2.

REFERENCES

[1] Kida T., Takeda M., Shinohara A., Miyazaki M., Arikava S., "Multiple pattern matching in LZW compressed text", *Data Compression Conference* 1998:103-112, 1999.

[2] Kida T., Takeda M., Shinohara A., Arikava S., "Shift-and approach to pattern-matching in LZW compressed text", *in Combinatorial Pattern-Matching 1999, Lecture Notes in Computer Science* 1645:1–1, 1999.

[3] Navarro G., Tarhio J., "LZgrep: A Boyer-Moore String Matching Tool for Ziv-Lempel Compressed Text", *Software Practice and Experience*, 35(12):1107-1130, 2005.

[4] Culpepper J. S., Moffat A., "Phrase-based pattern matching in compressed text*", In Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006)*:337-345, 2006.

[5] Moura E., Navarro G., Ziviani N., Baeza-Yates R., "Fast and Flexible Word Searching on Compressed Text", *ACM Transactions on Information Systems*, 18(2):113-139, 2000.

[6] Raita, T., "Tuning the Boyer-Moore-Horspool string searching algorithm", *Software - Practice & Experience*, 22(10):879-884, 1992.

[7] Boyer, R. S., Moore, J. S., "A fast string searching algorithm". *Communications of the ACM*. 20:762-772, 1977.

[8] Berry, T., Ravindran S., "A fast string matching algorithm and experimental results", *in Proceedings of the Prague Stringology Club Workshop`99, Collaborative Report DC-99-05:16-26*, 1999.

[9] Allauzen C., Crochemore M., Raffinot M., "Factor oracle: a new structure for pattern matching*", in* Proceedings of SOFSEM'99 Theory and Practice of Informatics*, J. Pavelka, G. Tel and M. Bartosek ed., Milovy, Czech Republic, Lecture Notes in Computer Science* 1725:291-306, Springer-Verlag, 1999.

[10] Apostolico A., Giancarlo R., "The Boyer-Moore-Galil string searching strategies revisited", *SIAM Journal on Computing*, 15(1):98-105, 1986.

[11] Charras, C., Lecroq, T., Handbook of Exact String Matching Algorithms, *King's College London Publications*, 2004.