A FAST MULTIPLE STRING-PATTERN MATCHING ALGORITHM

Sun Kim*
Central Research & Development
DuPont Experimental Station
Wilmington, DE 19880
Sun.Kim@usa.dupont.com

Yanggon Kim
Department of Computer and Information Sciences
Towson University
Towson, MD 21252
ykim@towson.edu

ABSTRACT

In this paper, we propose a simple but efficient multiple string pattern matching algorithm based on a compact encoding scheme. This algorithm scans text from left to right while encoding characters in the text based on the alphabet that occurs in the input patterns. The simple scanning algorithm demonstrates the ability to handle a very large number of input patterns simultaneously. Our experiments demonstrate that our algorithm runs faster than state-of-the-art multiple pattern matching programs including grep and agrep in many cases.

INTRODUCTION

The multiple string pattern matching problem is to find all occurrences of multiple input patterns, $P_1, .., P_n$ in text T. A naive way to solve this problem is to scan T for each pattern P_i separately, but requires scanning T as many times as the number of input patterns. Clearly a better solution to this problem can be devised. Desirable solutions would be to find all pattern occurrences by scanning T only once regardless of the number of input patterns. A lot of research on the multiple string pattern matching can be found in the literature [1, 5, 6, 13]. The state of the art algorithms can quickly find all pattern occurrences by scanning T only once even for a very large number of patterns; for example, Wu and Manber's algorithm [13] can find all occurrences of 10,000 input patterns in 15.8M text about 10 seconds on a SUN SPARC 10. Due to this efficiency, multiple string pattern matching techniques have many applications. Wu and Manber proposed an efficient approximate pattern matching software, AGREP [12] and a text searching tool, GLIMPSE [7] based on their multiple string matching algorithm [13]. Kim and Segre proposed a fast, reliable DNA sequence assembly algorithm using their multiple string pattern matching technique [9].

We propose an efficient multiple string pattern matching algorithm based on a compact encoding and a hashing scheme. When searching for pattern occurrences, the most intuitive way is to compare text characters against pattern characters. Well known string pattern matching algorithms like KMP [10] and the Boyer-Moore algorithm [4] are based on character comparisons. However, other techniques for finding pattern occurrences called *seminumerical* techniques [3, 11] also exists. Algorithms in this category are based on bit operations or on arithmetic by representing patterns and the text in bit strings. The bit representation results in comparing many pattern and text characters simultaneously and can lead to efficient techniques for finding multiple patterns simultaneously. We propose a multiple string pattern matching algorithm based on a bit representation of patterns and text using a compact encoding scheme.

We start by describing the compact encoding scheme. Then we propose a multiple string pattern matching algorithm using the compact encoding scheme together with a hashing technique in Section 3. We present some experimental results in Section 4, followed by discussion in Section 5.

*Partially supported by the University of Illinois Critical Research Initiatives

COMPACT ENCODING OF PATTERNS AND TEXT

In this section, we explain how to encode patterns and the text in a compact way. One advantage of using a compact encoding scheme is that multiple characters can be compared simultaneously with a single bit operation.

First, we start with the simplest case, DNA sequences where the alphabet has only four characters, `a, t, g` and `c`. By exploiting the small alphabet size of four, we can encode each character in two bits; 00 for `a`, 01 for `t`, 10 for `g`, and 11 for `c`. The following example illustrates how a DNA pattern and a DNA sequence can be encoded with this encoding scheme.

Example 1 Consider a DNA pattern P = `gatca` and a DNA sequence S = `gtacgatcagac`. Then we can encode P as 10 00 01 11 00 and S as 10 01 00 11 10 00 01 11 00 10 00 11 in a bit representation.[1]

Now we extend this compact encoding scheme to the general case. For a given pattern P and text T, we first count the number of distinct symbols in P. Let D be the number of distinct symbols in P and E be the smallest integer such that $2^E \geq (D+1)$. Then we can encode any symbol in P and T with E bits by assigning a distinct E bits for each character in P and assigning a distinct E bits for any character that does not occur in P but occurs in T. The following example illustrates this scheme.

Example 2 Consider a pattern P = `encoding` and T = "`Compact encoding can`". Since we have 7 distinct symbols in P, each character can be encoded in 3 bits ($2^3 = 8 \geq (7+1)$). Let's introduce a function ENCODE for encoding characters: ENCODE(e) = 001, ENCODE(n) = 010, ENCODE(c) = 011, ENCODE(o) = 100, ENCODE(d) = 101, ENCODE(i) = 110, ENCODE(g) = 111, and ENCODE(-) = 000 for any character - that does not occur in P. Then, P is encoded as 001 010 011 100 101 110 010 111 and T as 000 100 000 000 000 011 000 000 001 010 011 100 101 110 010 111 000 011 010.[2]

[1] A space between bit representations of characters is provided for readability.

[2] A space between bit representations of characters is provided for readability.

The compact encoding scheme can be summarized as follows.

1. Scan input pattern P and determine how many bits E are needed for compact encoding.
2. Define the encoding function ENCODE for each symbol in P and any symbol that does not occur in P.
3. Encode each symbol in P and T by function ENCODE.

A NEW MULTIPLE STRING PATTERN MATCHING ALGORITHM

Given the compact representation of patterns and text, the remaining questions are how to store the bit representation of patterns and text in registers or words and how to find pattern occurrences. For simplicity, we assume that the bit representation of a pattern can be stored in a single computer word.

Finding Occurrences Of A Single Pattern A pattern can be stored in words simply by repeatedly shifting and logical-ORing. The following example illustrates how we can store P in a variable of a word size.

Example 3 To store the bit representation of the pattern `encoding`, we first initialize a variable W with all 0's. Remember that each character in the pattern is encoded in 3 bits (see Example 2). Then
1. Perform logical-OR W and ENCODE(e) = 001,
2. Shift W left 3 bits and perform logical OR between W and ENCODE(n) = 010,
3. Continue the step 2 until the last pattern character is processed.
Then the word W will be 000000000010100111001011100101 11 where the lower 24 bits are the bit representation of the input pattern `encoding`.

To scan the text in the bit representation, we prepare another variable T and do exactly the same thing for each text character as for the pattern character while scanning each text character from left

to right. To determine the pattern occurrences in T, we need a mask PMASK. PMASK contains 1's for lower bit positions of the pattern length in bits and 0's for all other bit positions. For example, PMASK for pattern P=`encoding` in Example 3 is 00000000011111111111111111111111. PMASK is used for masking out non-pattern part of T. Given a text word T, pattern word P, and PMASK for P, two bit operations can determine the pattern occurrence:

1. Perform logical-AND T and PMASK.
2. Perform logical-XOR the result from Step 1 and P.
3. Test if the above result is all 0's; if so, the pattern occurs, otherwise, it dose not.

The next example illustrates how the pattern can be found in the text by a series of bit operations.

Example 4 Let us use the same pattern P = `encoding` and T = `Compact encoding can ...` as in Example 2. Then the bit representation of P is 00000000001010011100101110010111 and its PMASK is 00000000111111111111111111111111. Let variable T be used for scanning the text.

CASE 1. At the time of the first blank character (the one right after the character `t`) in the text being scanned.

T = 00000000000100000000000011000000
(T AND PMASK)
= 00000000000100000000000011000000.
((T AND PMASK) XOR P)
= 00000000001110011100101101010111

Since the result is not all 0's, the pattern does not occur at this position.

CASE 2. At the time of the character `g` in the text being scanned.

T = 11000000001010011100101110010111
(T AND PMASK)
= 00000000001010011100101110010111
((T AND PMASK) XOR P)
= 00000000000000000000000000000000

Here a pattern occurrence is found since the result is all 0's.

Finding Occurrences Of Multiple Patterns

Now we consider multiple input patterns $P_1, ..., P_n$. We can easily extend the single pattern search algorithm by preparing a separate PMASK$_i$ for each input pattern P_i. To find occurrences of all P_i's, we perform the pattern testing procedure (explained in the earlier section for finding occurrences of a single pattern) for each input pattern P_i. Unfortunately, it will take significantly more time as the number of input patterns increases. To avoid unnecessarily pattern testings, we prepare a hashing table of size H bits and distribute the input pattern to the hash table entry that corresponds to the lower H bits of the bit representation of the pattern. We perform the pattern testing only when the hash entry corresponding to the lower H bits of the text variable T is not empty.

Let S be the bit length of the shortest input pattern. We can set H to any value up to S, depending on the memory availability. Note that H should not be larger than S. Since we are using the compact encoding scheme, any reasonably large value H can screen out unnecessary pattern testings effectively. For example, suppose that E (the number of bits for a character) is 3 and H is 15. Then 32KB of memory ($=2^{15}$) for hashing is used, which is affordable. Since five characters (15 bits = 3 bits $\times 5$) are used for prefiltering unnecessary pattern testings, we expect that most unnecessary pattern testings can be avoided.

To utilize the hashing technique, we need to introduce another mask HMASK for hashing whose lower H bits are all 1's and the remaining bits are all 0's. Given a text variable T, we perform a logical AND between HMASK and T and then check if the hash entry corresponding to the resulting value is empty. If it is empty, we can skip pattern testings. Otherwise pattern testings for all patterns in the hash entry are performed.

The multiple string pattern matching algorithm can be summarized as below. A sample C code is given in Figure 1.

1. Scan the input patterns to determine the number of bits, E, for each character encoding and define the encoding function ENCODE (see Section 2).
2. Encode each pattern P_i and set the associated

mask PMASK_i.

3. Set the hash mask HMASK according to the hash table size.

4. Initialize the text scanning variable T to 0.

5. While scanning the text character by shifting T E bits left, perform the pattern testing procedure for all patterns at the hash entry position computed by logically ANDing T and H. If the hash entry at the position is empty, skip the pattern testing procedure (see Section 3.1) and scan the next text character.

EXPERIMENTS

To evaluate the performance of our two algorithms, MULTI1, we compare our algorithms to `grep` and `agrep` [12] on a SUN Ultra 10 workstation (333MHz processor) with 128MB memory. MULTI1 used a hash table of size up to 1M entries. All times reported are user times measured by a Unix command `time`. We performed two different kinds of experiments with two different texts; one text is the King James Bible (4,441,849 characters) obtained from the project Gutenberg[3] and the other is a large file of DNA sequences (18,617,116 characters) from 7 different completed genomes in organisms, *Archaeoglobus fulgidus, Aquifex aeolicus, Bacillus subtilis, Borrelia burgdorferi, Chlamydia trachomatis, Escherichia coli* and *Haemophilus influenzae Rd* obtained from GenBank[4].

An Experiment With English Pattern Searching We prepare a set of patterns by randomly selecting N patterns from the Unix dictionary `/usr/dict/words` and measured runtimes of grep, agrep, and MULTI1. The actual text we used is the one inflated by concatenating the King James Bile 3 times, so the text is of 13,325,547 characters. This is because the string matching algorithms are so fast that it was hard to measure the runtimes reliably with a text of about 4MB. The result is shown in Table 1.

[3](`ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/`)
[4](`ftp://ncbi.nlm.nih.gov`)

```
/* Searching for P[1],..,P[n] in text[0..Tlen] */
/* H: bits size for the hash table             */
/* E: \# of bits for encoding a character */
/* S: the bit length of the shortest pattern */

struct hash_entry {
   PAT                 p;
   PATMASK             pmask;
   struct hash_entry * next
};

/* HASH_SZ = exp(2,H) */
struct hash_entry HTBL[HASH_SZ];

PAT   HMASK;    /* a mask for hashing */

for (i=1; i<=n; i++)
   insert_pattern_into_hash_table(P[i]);

T = encode_ncharacters(text, S);
i = S+1;
while (i <= Tlen) {
  if (HTBL[T&HMASK] != NULL) {
    candidate = HTBL[T&HMASK];
    while(candidate) {
      if (( (T & candidate->pmask)
         ^ candidate->p) == 0)
            report_pattern_match(candidate);
      candidate = candidate->next;
    }
  }
  T = T<<E | ENCODE(text[i]);
  i ++;
}
```

Figure 1: A partial code for multiple string pattern matching algorithm in C (MULTI1)

#Patterns	MPL	grep	agrep	MULTI1
10	3	12.8	0.5	0.8
50	4	64.7	0.6	0.8
100	3	155.7	1.3	0.9
200	3	-	1.3	0.9
500	3	-	1.8	1.1
1000	3	-	2.3	1.2
2000	3	-	3.3	1.6
5000	3	-	5.5	2.9
10000	3	-	7.6	4.8
20000	3	-	12.3	11.3

Table 1: Runtimes for English pattern searching in seconds. (MPL stands for the minumum pattern length)

#Patterns	MPL	grep	agrep	MULTI1
10	10	4.2	1.6	3.4
50	10	21.8	2.1	3.5
100	10	47.7	4.0	3.5
200	10	108.1	5.6	3.5
500	10	-	9.2	3.4
1000	10	-	15.8	3.6
2000	10	-	29.6	3.8
5000	10	-	73.2	4.6
10000	10	-	151.9	5.9

Table 2: Runtimes for DNA pattern searching in seconds. (MPL stands for the minumum pattern length)

We could not measure runitmes for grep for cases of 200 patterns or larger since it took too much time. As you can see, MULTI1 runs faster than agrep for cases of 100 patterns or larger. However, this result should not be misled that our algorithms always run faster than state-of-art algorithms like agrep for all cases as the speed of pattern searching is affected by the structure of the input patterns and the text. One of the factors we should consider in the analysis of pattern searching speed is the minimum pattern length as the three algorithms, grep and agrep use text skipping scheme (the Boyer-Moore shifting scheme) while MULTI1 use no text skipping scheme scheme. For example, agrep takes 0.3 seconds but MULTI1 takes 1.6 seconds for the case of 10 patterns with the minimum pattern length of 6 as we expect the text skipping scheme more effective for longer patterns. In addition to the minimum pattern length, there are others factors that affect the pattern searching speed; for example, the alphabet size of the patterns and the text, etc. We will present our experience with variable input pattern structures in a separte report.

An Experiment With DNA Pattern Searching In this section, we present the results from an experiment with the DNA file from 7 microorganisms. The main difference between the DNA data and the English data is the size of alphabet; DNA has the alphabet of size 4 while the English data has most of alphanumeric characters. This experiment will show how the string matching algorithms behave for texts of different alphabet size. We randomly generate N DNA patterns of length ranging from 10 to 32 and measured runtimes of grep, agrep, and MULTI1. The result is shown in Table 2.

Again, we could not measure runitmes for grep for cases of 500 patterns or larger since it took too much time. As shown in the table, MULTI1 runs faster than grep for many cases. One interesting observation is that runtimes of MULTI1 does not vary much with respect to the number of input patterns; it just took 5.9 seconds for 10,000 patterns while it took 3.4 seconds for 10 patterns. Another interesting observation is that MULTI1 runs much faster than agrep as the the number of input patterns increases. This is beacuse our hashing scheme utilizes the small alphabet size of four automatically using the compact encoding scheme. These show the effectiveness of our compact encoding and hashing scheme for searching multiple patterns simultaneously.

DISCUSSION

In this paper, we proposed a new multiple string-pattern matching algorithm based on compact encoding and hashing scheme. The algorithm demonstrates the ability of handling a very large number of patterns simultaneously and runs faster than

`grep` and `agrep` in many cases. The hashing techniques are used in other multiple-string matching algorithms to handle a large number of patterns. For example, Wu and Manber's algorithm [13] uses three hash tables, SHIFT, HASH, and PREFIX to filter out unnecessary pattern occurrence testings. The hashing scheme proposed in this paper exploits the compact character encoding scheme that dynamically adjusts to the alphabet size to achieve better runtime performance.

As our algorithms rely on hashing techniques, it is essential to reduce the number of collisions in the hash entries to speed up the pattern searching. One thing we did not explored in this paper is that we can reduce the number of collisions in the hash entries by using different compact encoding scheme for characters depending on the input pattern structure. We call this technique *adaptive string pattern matching* to emphasize that the string matching algorithm *adapts* to the input pattern structure to optimize its performance. This adaptive string matching technique has been successfully applied to various algorithms including a new string-pattern matching algorithm by text partitioing [8], which will be reported in a forthcoming paper.

References

[1] Aho, A. V. and Corasick, M. J., "Efficient string matching: an aid to bibliographic search," *Communications of the ACM* **18**, (1975)

[2] Beaza-Yates, R. A., "Improved string searching" *Software – Practice and Experience*, **19**, (1989), 257-271

[3] Beaza-Yates, R. A. and Gonnet, G. H, "A new approach to text searching" *Communications of the ACM* **35**, 10 (1992), 74-82

[4] Boyer, R. S. and Moore, J. S., "A fast string searching algorithm," *Communications of the ACM* **20**, 10 (1977), 762-772

[5] Commentz-Walter, B., "A string matching algorithm fast on the average," *Proc. 6th International Colloquium on Automata, Languages, and Programming*, 1979

[6] Haertel, M., "Gnugrep-2.0,", Usenet archive *comp.sources.reviewed* Vol. 3, 1993

[7] Manber, U. and Wu, S., "GLIMPSE: A Tool to Search Through Entire File Systems," *Usenix Winter Technical Conference*, 1994

[8] Kim, S., "A New String Pattern Matching Algorithm Using Partitioning and Hashing Efficiently," Submitted to *ACM Journal of Experimental Algorithmics*, March 1998

[9] Kim, S. and Segre, A. M., "AMASS: A Structured Pattern Matching Approach to Shotgun Sequence Assembly," *Journal of Computational Biology*, Vol 6(2) 1999

[10] Knuth, D. E., Morris, J. H. and Pratt, V. R., "Fast pattern matching in strings," *SIAM Journal on Computing*, **6**, 2 (1977), 323-350

[11] Karp, R. M. and Rabin, M. O, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, **31**, 249-260

[12] Wu, S. and Manber, U., "Agrep – A Fast Approximate Pattern-matching Tool," *Usenix Winter Technical Conference*, 1992

[13] Wu, S. and Manber, U., "A Fast Algorithm for Multi-pattern Searching," Technical Report, The Computer Science Department, The University of Arizona, 1994