KYBERNETIKA — VOLUME 38 (2002), NUMBER 1, PAGES 67 – 80

TUNING THE ZHU–TAKAOKA STRING MATCHING ALGORITHM AND EXPERIMENTAL RESULTS

THOMAS BERRY AND SOMASUNDARAM RAVINDRAN

In this paper we present experimental results for string matching algorithms which have a competitive theoretical worst case run time complexity. Of these algorithms a few are already famous for their speed in practice, such as the Boyer–Moore and its derivatives. We chose to evaluate the algorithms by counting the number of comparisons made and by timing how long they took to complete a given search. Using the experimental results we were able to introduce a new string matching algorithm and compared it with the existing algorithms by experimentation. These experimental results clearly show that the new algorithm is more efficient than the existing algorithms for our chosen data sets. Using the chosen data sets over 1,500,000 separate tests were conducted to determine the most efficient algorithm.

1. INTRODUCTION

Many promising data structures and algorithms discovered by the theoretical community are never implemented or tested at all. Moreover, theoretical analysis (asymptotic worst-case running time) will show only how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance. In this paper we show that by considerable experimentation and fine-tuning of the algorithms we can get the most out of a theoretical idea.

The string matching problem [8] has attracted a lot of interest throughout the history of computer science, and is crucial to the computing industry. String matching is finding an occurrence of a pattern string in a larger string of text. This problem arises in many computer packages in the form of spell checkers, search engines on the internet, find utilities on various machines, matching of DNA strands and so on.

Existing string matching algorithms which are known to be fast are described in the next section. Experimental results for these algorithms are given the following section. From the findings of the experimental results we identify two fast algorithms. We combine these two algorithms and introduce a new algorithm. We compare the new algorithm with the existing algorithms by experimentation.

2. THE STRING MATCHING ALGORITHMS

String matching algorithms work as follows. First the pattern of length m, P[1...m], is aligned with the extreme left of the text of length n, T[1...n]. Then the pattern characters are compared with the text characters. The algorithms can vary in the order in which the comparisons are made. After a mismatch is found the pattern is shifted to the right and the distance the pattern can be shifted is determined by the algorithm that is being used. It is this shifting procedure and the speed at which a mismatch is found which is the main difference between the string matching algorithms.

In the Naive or Brute Force (BF) algorithm, the pattern is aligned with the extreme left of the text characters and corresponding pairs of characters are compared from left to right. This process continues until either the pattern is exhausted or a mismatch is found. Then the pattern is shifted one place to the right and the pattern characters are again compared with the corresponding text characters from left to right until either the text is exhausted or a full match is obtained. This algorithm can be very slow. Consider the worst case when both pattern and text are all a's followed by a b. The total number of comparisons in the worst case is $O(nm)$. However, this worst case example is not one that occurs often in natural language text.

The number of comparisons can be reduced by moving the pattern to the right by more than one position when a mismatch is found. This is the idea behind the Knuth–Morris–Pratt (KMP) algorithm [12]. The KMP algorithm starts and compares the characters from left to right the same as the BF algorithm. When a mismatch occurs the KMP algorithm moves the pattern to the right by maintaining the longest overlap of a prefix of the pattern with a suffix of the part of the text that has matched the pattern so far. After a shift, the pattern character compared against the mismatched text character has to be different from the pattern character that mismatched. The KMP algorithm takes at most $2n$ character comparisons. The KMP algorithm does $O(m+n)$ operations in the worst case.

The Boyer–Moore (BM) algorithm [3, 19] differs in one main feature from the algorithms already discussed. Instead of the characters being compared from left to right, in the BM algorithm the characters are compared from right to left starting with the rightmost character of the pattern. In a case of mismatch it uses two functions, last occurrence function and good suffix function and shifts the pattern by the maximum number of positions computed by these functions. The good suffix function returns the number of positions for moving the pattern to the right by the least amount, so as to align the already matched characters with any other substring in the pattern that are identical. The number of positions returned by the last occurrence function determines the rightmost occurrence of the mismatched text character in the pattern. If the text character does not appear in the pattern then the last occurrence function returns m. The worst case time complexity of the BM algorithm is $O(nm)$.

The Turbo Boyer–Moore (TBM) algorithm [6] and the Apostolico–Giancarlo (AG) algorithm [7] are amelioration's of the BM algorithm. When a partial match is made between the pattern and the text these algorithms remember the characters

that matched and do not compare them again with the text. The TBM algorithm and the Apostolico–Giancarlo algorithm perform in the worst case at most $2n$ and $1.5n$ character comparisons respectively.

The Horspool (HOR) algorithm [10] is a simplification of the BM algorithm. It does not use the good suffix function, but uses a modified version of the last occurrence function. The modified last occurrence function determines the right most occurrence of the $(k + m)$th text character, $T[k + m]$ in the pattern, if a mismatch occurs when a pattern is aligned with $T[k \ldots k + m]$. The comparison order is not described in [10]. We assumed that the order is from right to left as in the BM algorithm.

The Raita (RAI) algorithm [15] uses variables to represent the first, middle and last characters of the pattern. The process used is to compare the rightmost character of the pattern, then the leftmost character, then the middle character and then the rest of the characters from the second to the $(m - 1)$th position. Using variables is more efficient than looking up the characters in the pattern array. The use of variables to represent characters in the array is known as 'Raita's trick'. This optimization trick is only used in the RAI algorithm. If at any time during the procedure a mismatch occurs then it performs the shift as in the HOR algorithm.

The Quicksearch (QS) algorithm [18] is similar to the HOR algorithm and the RAI algorithm. It does not use the good suffix function to compute the shifts. It uses a modified version of the last occurrence function. Assume that a pattern is aligned with the text characters $T[k \ldots k + m]$. After a mismatch the length of the shift is at least one. So, the character at the next position in the text after the alignment ($T[k + m + 1]$) is necessarily involved in the next attempt. The last occurrence function determines the right most occurrence of $T[k + m + 1]$ in the pattern. If $T[k + m + 1]$ is not in the pattern the pattern can be shifted by $m + 1$ positions. The comparisons between text and pattern characters during each attempt can be done in any order.

The Maximal Shift (MS) algorithm [18] is another variant of the QS algorithm. The algorithm is designed in such a way that the pattern characters are compared in the order which will give the maximum shift if a mismatch occurs.

The Liu, Du and Ishi (LDI) algorithm [13] is a variant of the QS algorithm. The algorithm uses the same shifting function as the QS but changes the order in which the pattern characters are compared to the text. The characters are compared in a circular method starting at the first character in the pattern and finishing at the last. If a mismatch occurs then the pattern is shifted and searching restarts with the pattern character that mismatched. For example, if the pattern was *'string'* and the pattern mismatched the text at the '*r*' then we would search in the order *'ringst'*.

The Smith (SMI) algorithm [16] uses HOR and Quick Search last occurrence functions. When a mismatch occurs, it takes the maximum values between these functions. The characters are compared from left to right.

The Zhu and Takaoka (ZT) algorithm [20] is another variant of the BM algorithm. The comparisons are done in the same way as BM (i. e. from right to left) and it uses the good suffix function. If a mismatch occurs at $T[i]$, the last occurrence function

determines the right most occurrence of T[$i-1 \ldots i$] in the pattern. If the substring is in the pattern, the pattern and text are aligned at these two characters for the next attempt. The shift is m, if the two character substring is not in the pattern. The shift table is a two dimensional array of size alphabet size by alphabet size.

The Baeza–Yates (BY) algorithm is similar to the ZT algorithm. It calculates the shift according to the last k characters of the pattern aligned with the text. When k=2 the shifts are the same as ZT but without the good suffix function. The main differences are constructing and storing the shift table. The shift table is a one dimensional array of length σ^2, where σ is the size of the alphabet. The table is constructed by bit shifting the two characters to form a 16 bit number and storing the value of the shift at this location in the array.

Searching can be done in $O(n)$ time using a minimal Deterministic Finite Automaton (DFA) [9, 14]. This algorithm uses $O(\sigma m)$ space and $O(m+\sigma)$ pre-processing time.

The pre-processing is needed for the algorithm to calculate the relevant shifts upon a mismatch/match except for the BF algorithm, which has no pre-processing. The pre-processing cost of the algorithms is important factor in the speed of the algorithm with regard to the number of operations required and the amount of memory required. This will be most noticeable when we are searching in smaller texts.

3. EXPERIMENTAL RESULTS OF THE EXISTING ALGORITHMS

Monitoring the number of comparisons performed by each algorithm was chosen as a way to compare the algorithms. All the algorithms were coded in C and their C code are taken from [4] and animation's of the algorithms can be found at [5]. This collection of string matching algorithms were easy to implement as functions into our main control program. The algorithms were coded as their authors had devised them in their papers. The main control program read in the text and pattern and had one of the algorithms to be tested inserted into it for the searching process. The main control program was the same for each algorithm and so did not affect the performance of the algorithms. Each algorithm had an integer counter inserted into it, to count the number of comparisons made between the pattern and the text. The counter was incremented by one each time a comparison was made. Note that text characters compared when calculating the valid shift are not included.

A random text of 200,000 words from the UNIX English dictionary was used for the first set of experiments. We decided to number each of the words in UNIX dictionary from 1 to 25,000. Then we used a pseudo random number generator to pick words from the UNIX dictionary and place them in the random text. Separating each word by a space character. Punctuation was also removed as we were concerned with finding words and the punctuation would not effect the results obtained. We selected a word (pattern) from the UNIX dictionary and searched the text for the first occurrence of the word.

The text was searched for each word in the UNIX dictionary and the results are given in Table 1. The first column in Table 1 is the length of the pattern.

The second column is the number of words of that length in the UNIX English dictionary. For example, for a pattern length of 7, 4042 test cases were carried out and the average number of character comparisons made by the KMP algorithm was 197,000 (to the nearest 1000). The average was calculated by taking the total number of comparisons performed to find all 4042 cases and dividing this number by 4042. The figure given is the total number of comparisons taken divided by the number of words for the pattern length and then divided by 1000.These columns are arranged in descending order of the average of the total number of comparisons of the algorithms. An interesting observation is that for (almost) each row the values are in descending order except for the last two columns.

Table 1. The number of comparisons in 1000's for searching a text of 200,000 words (1670005 characters).

p. len	num.	BF	KMP	DFA	BY	BM	AG	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI
2	133	7	7	7	7	3	3	3	3	3	2	2	2	3	2
3	765	38	38	37	19	13	13	13	13	13	11	10	10	13	10
4	2178	82	82	80	28	23	23	23	23	22	19	19	19	22	18
5	3146	151	150	145	39	34	34	34	34	34	30	30	30	32	28
6	3852	186	185	179	38	36	36	36	36	36	33	33	32	33	30
7	4042	198	197	191	34	34	34	34	34	34	32	31	31	30	28
8	3607	205	204	197	30	32	32	31	32	31	30	29	29	27	26
9	3088	212	211	204	28	30	30	30	30	30	29	28	28	25	24
10	1971	220	219	212	26	29	29	29	29	29	28	27	27	24	23
11	1120	209	207	201	22	26	26	26	26	25	25	24	24	21	21
12	593	218	217	210	21	25	25	25	25	25	24	24	24	21	20
13	279	224	222	215	20	24	24	24	24	24	23	24	23	19	19
14	116	228	227	220	19	23	23	23	23	23	23	23	23	19	19
15	44	151	150	144	11	15	15	15	15	14	14	14	14	11	12
16	17	227	225	217	16	20	21	21	21	20	20	21	20	18	16
17	7	233	231	222	16	20	20	20	20	19	19	20	20	15	16
18	4	236	234	225	15	19	20	20	20	19	19	20	20	14	16
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	1	132	131	122	7	10	10	10	10	10	10	10	10	7	8
21	2	311	309	295	16	24	24	25	25	23	23	24	24	15	18
22	1	491	486	455	23	33	33	33	33	33	31	34	34	22	27
Total	24966	180	179	174	31	31	31	30	30	30	28	28	28	27	25

The algorithm with the largest number of comparisons is the BF algorithm. This is because the algorithm shifts the pattern by one place to the right when a mismatch occurs, no matter how much of a partial/full match has been made. This algorithm has a quadratic worst case time complexity. However, the KMP algorithm, which has a linear worst case time complexity, does roughly the same number of comparisons as the BF algorithm. The reason for this is that in a natural language a multiple occurrence of a substring in a word is not common. Other linear time algorithms, DFA, also have roughly the same number of comparisons as the BF algorithm. We will see below that the other quadratic worst case time complexity algorithms perform much better than these linear worst case time algorithms. This is a good example showing that asymptotic worst-case running time analysis can be indicative of how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance.

The BM algorithm uses the good suffix function to calculate the shift which depends on a reoccurrence of a substring in a word. But, it also uses the last occurrence function. It is this last occurrence function that reduces the number of comparisons significantly. In practice, on an English text, the BM algorithm is three or more times faster than the KMP algorithm [17]. From Table 1, one can see that the KMP algorithm takes six times as many comparisons than the BM algorithm on average. The other algorithms, BY, TBM, AG, HOR, RAI, LDI, QS, MS, SMI and ZT, are variants of the BM algorithm. The number of comparisons for these algorithms is roughly the same number as in the BM algorithm.

The SMI algorithm and the ZT algorithm do the least number of comparisons for pattern lengths less than or equal to twelve and greater than twelve respectively.

4. THE NEW ALGORITHM (BR)

From the findings of the experimental results discussed in Section 3, it is clear that the SMI and ZT algorithms have the lowest number of comparisons among the others. We combined the calculations of a valid shift in QS and ZT algorithms to produce a more efficient algorithm, the BR algorithm [21]. If a mismatch occurs when the pattern P[1 ... m] is aligned with the text T[$k..,k+m$], the shift is calculated by the rightmost occurrence of the substring T[$k+m+1 \ldots k+m+2$] in the pattern. If the substring is in the pattern then the pattern and text are aligned at this substring for the next attempt. This can be done shifting the pattern as shown in the table below. Let $*$ be a wildcard character that is any character in the ASCII set. Note that if T[$k+m+1 \ldots k+m+2$] is not in the pattern, the pattern is shifted by $m+2$ positions.

T[$k+m+1$]	T[$k+m+2$]	*Shift*
$*$	P[1]	$m+1$
P[i]	P[$i+1$]	$m-i+1$, $1 \leq i \leq m-1$
P[m]	$*$	1
Otherwise		$m+2$

For example, the following shifts would be associated with the pattern, 'onion'.

T[$k+m+1$]	T[$k+m+2$]	*Shift*
$*$	o	6
o	n	5
n	i	4
i	o	3
o	n	2
n	$*$	1
Otherwise		7

After a mismatch the calculation of a shift in our new algorithm takes $O(1)$ time. Note that for the substrings 'ni' and 'n$*$' have a value of 4 and 1 respectively. This ambiguity can be solved by the higher shift value being overwritten with the lower value. We will explain this later in this section. For a given pattern P[1 ... m]

the preprocessing is done as follows, and it takes $O(m^2 + \sigma)$ time, where σ is the size of the alphabet. The two dimensional array, ST (Shift Table), of size at most $m + 1 \times m + 1$ will store the shift values for all pairs of characters. The ST will be initialised as $m+2$. As the index of the ST is of type integer, we need to convert the pairs of characters into pairs of integers. This is done by defining an array of ASCII character set size called CON with each entry initialised to 1. For each character in the pattern the right most position (numbering from the right, starting with 2) is entered in the corresponding location in CON. For example, the relative position of the character 'a' in the ASCII set is 97. Assume that the character 'a' is in the pattern. The right most position of 'a' in the pattern is entered in CON[97].

If the pattern was the word 'onion' then the rightmost positions of n, o and i are 2, 3 and 4 respectively. The CON for 'onion' would look like this:

Character:	...	a	b	...	h	i	j	...	n	o	p	...
ASCII value:	...	97	98	...	104	105	106	...	110	111	112	...
CON:	...	1	1	...	1	4	1	...	2	3	1	...

The value of a shift for the pair T[k+m+1] and T[k+m+2] is ST(CON[T[k+m+1]], CON[T[$k + m + 1$]]).

All the entries in the ST will be initialised as 7, and the above shift values will be entered as follows.

[*wildcard*] [o] = 6	[i] [o] = 3
[o] [n] = 5	[o] [n] = 2
[n] [i] = 4	[n] [*wildcard*] = 1

The ST for the pattern 'onion' would look like this after the complete insertion of all the values.

	1	2	3	4
1	7	7	6	7
2	1	1	1	1
3	7	2	6	7
4	7	7	3	7

The order of performing the steps is important in ensuring the correct values appear in ST. Note that the higher values have been over written by the lower values. We search for the pattern starting at P[m] and searching from right to left finish at P[1]

We now give an example of our new algorithm in action to find the pattern 'onion'. The tables above, ST and CON for the pattern 'onion' were used to calculate the shift after a mismatch.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				#																					
o	n	i	o	n																					

mismatch shift on ST(CON[n], CON[t]) = ST(2, 1) = 1

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				#	=																				
	o	n	i	o	n																				

mismatch shift on ST(Con [t]], CON[ ]) = ST(1, 1) = 7.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
												≠													
							o	n	i	o	n														

mismatch shift on ST(CON[s], CON[t]) = ST(1, 1) = 7

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																			≠						
															o	n	i	o	n						

mismatch shift on ST(CON[ ], CON[o]) = ST(1, 3) = 6

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																					=	=	=	=
																					1	2	3	4
																				o	n	i	o	n

So the word onion is found in 10 comparisons in a text of length 26. On the above full match, the order in which the comparisons are conducted is shown on the third row.

5. EXPERIMENTAL RESULTS AND COMPARISONS WITH THE BR ALGORITHM

We select the best eight algorithms from the results in Table 1, the BM algorithm and the KMP algorithm, and compare with our BR algorithm. Experiments were carried out for different random texts as described in Section 3. There were 2 different texts of 10,000 words (Texts A and B), a text of 50,000 words and a text of 100,000 words. The results are described in Tables A1 – A4 (see Appendix) respectively. Tables 3 – 6 (which can be found in the appendix at the back of this paper) show the average number of comparisons required for a search for the given pattern length. They are based on taking the total number of comparisons for the search for all the patterns of a length and dividing the number by the number of patterns of that size to give the average. So for example, in Table 3 the BM algorithm takes 12,000 comparisons (to the nearest thousand) on average if the pattern length is seven. From these tables one can observe that the relative order of their performance is the same as in Table 1. The main observation is that the BR algorithm performs better than the other algorithms for all pattern lengths and for all texts used in the experiments.

Table 2 summarises the results of Tables A1 – A4. The entries in Table 2 are in percentage form and describe how many more comparisons existing algorithms did than our BR algorithm. The figures are an average of the four different texts used. To

calculate the difference as a percentage between our BR algorithm and the existing algorithms we used the following formula. The average number of comparisons was taken from the relevant cell in Tables A1 – A4 and divided by the value for that pattern length for our BR algorithm. This value was then deducted by 1 and multiplied by 100 to give the percentage difference between the two algorithms. An interesting observation of the existing algorithms when compared with the BR algorithm, is that for each individual text the percentages were within 1 % for each specific algorithm. Each value in Table 2 is calculated by taking the difference as a percentage between each algorithm and our BR algorithm for each pattern length, adding them together and dividing by 4. For example, for a pattern length of 4 the BM algorithm takes on average 51.11 % more comparisons than our BR algorithm. Note that the figures only include direct comparisons between the text and the pattern and not any text comparisons made during the calculation of a shift.

Table 2. The average (of Tables A1 – A4) percentage difference in the number of comparisons between existing algorithms and the BR algorithm.

pat. len.	num.	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI
2	133	199.98	93.96	94.00	93.96	93.89	35.94	37.23	32.92	93.96	31.48
3	765	366.02	64.18	64.20	64.19	63.70	28.78	32.90	28.21	60.03	24.93
4	2178	449.02	51.11	50.86	50.90	50.77	28.25	31.09	25.77	43.19	19.73
5	3146	540.11	45.02	44.58	44.46	44.72	28.33	31.57	26.47	33.91	18.13
6	3852	626.30	42.42	41.83	41.68	41.91	30.02	32.38	27.32	27.71	16.42
7	4042	719.01	41.38	40.92	41.00	40.72	31.49	33.58	28.83	24.94	16.08
8	3607	807.61	40.58	40.28	40.35	39.95	32.27	34.99	30.10	21.67	15.49
9	3088	896.18	41.52	40.92	40.84	40.69	34.75	37.13	32.19	19.29	15.45
10	1971	982.63	42.19	41.69	41.79	41.16	36.62	39.31	34.37	17.75	15.64
11	1120	1067.87	44.14	43.67	43.79	42.97	38.57	42.14	37.18	17.06	16.32
12	593	1164.14	45.28	44.58	44.68	44.20	40.06	44.38	39.28	16.14	17.34
13	279	1245.53	47.88	47.22	47.32	46.36	42.26	46.69	41.61	12.65	17.54
14	116	1322.70	46.74	46.46	46.60	45.16	42.62	48.68	42.26	11.32	17.03
15	44	1426.02	51.20	51.51	51.59	49.23	44.73	52.89	45.29	8.72	19.00
16	17	1527.28	49.34	50.44	50.60	47.37	46.60	52.95	49.06	24.80	20.02
17	7	1598.50	45.29	44.51	44.58	43.42	40.22	50.20	45.01	6.72	16.95
18	4	1700.81	50.58	53.96	54.06	48.54	50.12	59.06	53.59	6.09	22.21
19	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	1	1948.74	58.37	58.12	58.07	58.37	52.25	72.62	63.51	3.01	29.43
21	2	1947.96	57.38	63.98	63.99	56.32	57.59	64.09	57.50	2.22	21.84
22	1	2129.14	50.97	49.87	49.89	50.97	45.07	66.54	55.43	1.04	25.09
Total	24992	737.56	43.29	42.83	42.82	42.65	32.00	34.59	29.72	26.09	16.66

The result of a full search for the dictionary over all four texts is given in the last row of Table 2. From this we can see that the BM algorithm is took on average 43.54 % more comparisons than our BR algorithm (see 5th column, last row) for a complete search for all the words in the dictionary.

We also measure the user time for these algorithms as the saving in the number of comparisons may be paid for by some extra overhead operations. We timed the search of book1 of for all occurrences of 500 words from the UNIX dictionary. The words are of length 2 to 11 and there are 50 words of each length. The words were chosen at random from the UNIX dictionary. We show the average length of a shift performed by each algorithm in the second column. The percentage difference

between the existing algorithms and the BR algorithm is shown in the third column. We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. The user time includes the time taken for any pre-processing and the reading of the text into memory. Each algorithm was evaluated ten times for and the average time taken is given in Table 3. The timing was accurate to 1/100 of a second but was rounded to the nearest second. The difference between the slowest and fastest time for each test for an algorithm was less than 0.2 of a second. The last column shows the percentage difference of the user time between existing algorithms and the BR algorithm.

Table 3. The average shift and the user time in seconds.

Algorithm	average shift	% difference	time in secs.	% difference
BF	1.00	708.00	3402	315.89
KMP	1.00	708.00	4727	477.87
DFA	1.00	708.00	3057	273.72
BY	5.61	44.03	987	20.66
BM	5.76	40.28	1518	85.57
AG	5.65	43.01	4396	437.41
HOR	5.72	41.26	1042	27.38
RAI	5.72	41.26	865	5.75
MS	6.40	26.25	1237	51.22
LDI	6.34	27.44	1115	36.31
QS	6.49	24.50	1094	33.74
ZT	6.38	26.65	1874	129.10
TBM	5.57	45.06	2240	173.84
SMI	7.11	13.64	1186	44.99
BR	8.08	N/A	818	N/A

If we list the algorithms in order of the average shift that they take from the highest to the lowest starting at the BM, we will get: BM, LDI, ZT, QS, MS, SMI and the BR. But, if we do the same for the timings we get BM, MS, SMI, LDI, QS, HOR, BY, RAI and the BR. The reason for the difference in the lists is due to overheads in traversing the data structures which are present in the algorithms for the calculation of the correct shift value. Therefore, we can not assume that because an algorithm has a higher average shift that it will be more efficient than another.

We then considered eight other texts, 'Book2', 'news' and the six papers from the Calgary corpus [22]. The number of words and the number of characters of these texts are shown in Table 5. We searched for the same 500 random words from the UNIX dictionary for the BM, BR, BY, HOR, LDI, QS, RAI, and SMI algorithms. The reason for using different texts of different sizes was to check that the pre-processing of the BR didn't become too expensive as the text became smaller in size. We also needed to check that the distribution of the characters in the text didn't effect the speed of the BR algorithm.

The results documented in Table 4 show that the BR algorithm is the faster than the existing algorithms for when the text is large. The RAI algorithm is the fastest algorithm for texts 'paper 4' and 'paper 5'. This is due to the time for the pre-processing in BR which is not as dominant in the RAI algorithm. The tests were carried out for a wide range of text sizes as shown in Table 5. The main reason for the speed of our BR algorithm is the improved maximum shift of $m+2$.

Table 4. User times in seconds for the eight chosen texts.

	BM	BR	BY	HOR	LDI	QS	RAI	SMI	BNR	ZHU
Paper 1	103.7	56.0	68.2	71.3	76.6	74.7	59.3	81.3	56.3	169.9
Paper 2	161.8	86.8	106.2	111.2	120.1	116.9	92.4	126.5	87.3	247.1
Paper 3	93.2	50.1	61.2	64.0	69.2	67.4	53.3	72.8	50.4	164.9
Paper 4	26.7	15.5	17.6	18.2	19.8	19.2	15.1	20.9	15.6	85.5
Paper 5	23.3	13.9	15.7	16.2	17.8	17.1	13.5	18.7	14.0	82.2
Paper 6	74.2	40.2	48.7	51.0	54.5	53.2	42.4	58.2	40.4	143.3
Book 2	1195.0	639.0	784.0	820.0	884.0	862.0	681.0	934.0	642.0	1485.0
News	727.0	391.0	476.0	498.0	533.0	520.0	414.0	570.5	393.0	862.0

Table 5. The number of words and characters of the texts used in Table 4.

	number of words	number of characters
Paper1	8512	53162
Paper2	13830	82205
Paper3	7220	47139
Paper4	2167	13292
Paper5	2100	11960
Paper6	6754	38111
Book1	139994	773635
Book2	101221	610856
News	53940	37711

6. CONCLUSIONS

The experimental results show that the BR algorithm is more efficient than the existing algorithms in practice for most of our chosen data sets. Over our 4 random texts and 9 real texts where the BR algorithm is compared to the existing algorithms, our algorithm is more efficient for all but two of the texts. With the addition of punctuation and capital letters it does not affect the BR algorithm. So in the real world we would expect our savings to remain and make our BR algorithm competitive with the existing algorithms. It is also possible to apply some of our findings to what makes a fast algorithm to the existing algorithms. This may make them faster but we were concerned with the original algorithms that were devised by their authors.

ACKNOWLEDGEMENTS

We wish to thank Carl Bamford for comments and suggestions made to us during the writing of this paper.

(Received May 16, 2000.)

APPENDIX

The figure given in each table is the total number of comparisons taken divided by the number of words for the pattern length and then divided by 1000.

Table A1. The number of comparisons in 1000's for searching Text A of 10,000 words (83360 characters).

p. len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	6	3	3	3	3	2	2	2	3	2	2
3	765	20	7	7	7	7	6	6	6	7	5	4
4	2178	41	11	11	11	11	10	10	10	11	9	7
5	3146	60	14	13	13	13	12	12	12	12	11	9
6	3852	67	13	13	13	13	12	12	12	12	11	9
7	4042	68	12	12	12	12	11	11	11	10	10	8
8	3607	69	11	11	11	11	10	10	10	9	9	7
9	3088	70	10	10	10	10	9	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	6
11	1120	70	9	9	9	9	8	8	8	7	7	6
12	593	70	8	8	8	8	8	8	8	6	7	5
13	279	72	8	8	8	8	8	8	8	6	6	5
14	116	69	7	7	7	7	7	7	7	5	6	5
15	44	72	7	7	7	7	7	7	7	5	6	5
16	17	70	6	6	6	6	6	6	6	5	5	4
17	7	75	7	7	7	6	6	6	6	5	5	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	89	7	7	7	7	7	7	7	4	5	4
21	2	88	7	7	7	7	6	7	7	4	5	4
22	1	89	6	6	6	6	6	6	6	4	5	4
Total	24966	64	11	11	11	11	10	10	10	10	9	7

Table A2. The number of comparisons in 1000's for searching Text B of 10,000 words (83425 characters).

p. len	Num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	6	3	3	3	3	2	2	2	3	2	2
3	765	21	7	7	7	7	6	6	6	7	6	4
4	2178	42	12	12	12	12	10	10	10	11	9	7
5	3146	59	13	13	13	13	12	12	12	12	11	9
6	3852	66	13	13	13	13	12	12	12	11	11	9
7	4042	68	12	12	12	12	11	11	11	10	10	8
8	3607	69	11	11	11	11	10	10	10	9	9	7
9	3088	70	10	10	10	10	9	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	6
11	1120	70	9	9	9	9	8	8	8	7	7	6
12	593	71	8	8	8	8	8	8	8	6	7	5
13	279	71	8	8	8	8	8	8	7	6	6	5
14	116	70	7	7	7	7	7	7	7	6	6	5
15	44	64	6	6	6	6	6	6	6	5	5	4
16	17	74	7	7	7	7	7	7	7	5	5	4
17	7	64	6	6	6	6	5	6	6	4	4	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	72	5	6	6	5	5	6	5	4	4	3
22	1	89	6	6	6	6	6	6	6	4	5	4
Total	24966	63	11	11	11	11	10	10	10	10	9	7

Table A3. The number of comparisons in 1000's for searching a text of 50,000 words (417923 characters).

p. len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	9	6	6	6	6	4	4	4	6	4	3
3	765	37	13	13	13	13	10	10	10	13	10	8
4	2178	77	21	21	21	21	18	18	18	20	17	14
5	3146	133	30	30	30	30	27	26	26	28	25	21
6	3852	159	31	31	31	31	29	28	28	28	26	22
7	4042	170	29	29	29	29	27	27	27	26	24	21
8	3607	176	27	27	27	27	26	25	25	24	22	19
9	3088	181	26	26	26	26	25	24	24	22	21	18
10	1971	185	24	24	24	24	23	23	23	20	20	17
11	1120	184	23	23	23	23	22	22	22	18	18	16
12	593	186	21	21	21	21	21	21	20	17	17	15
13	279	183	20	20	20	20	19	19	19	15	16	14
14	116	194	20	20	20	20	19	20	19	15	16	14
15	44	164	16	16	16	16	16	16	16	12	13	11
16	17	217	20	20	20	20	20	20	20	17	16	13
17	7	172	15	15	15	14	14	15	15	11	12	10
18	4	147	12	13	13	12	12	13	13	9	10	8
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	221	17	18	18	17	17	17	17	11	13	10
22	1	397	27	27	27	27	26	28	28	18	22	18
Total	24966	155	27	26	26	26	24	24	24	23	22	18

Table A4. The number of comparisons in 1000's for searching a text of 100,000 words (834381 characters).

p. len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	13	7	7	7	7	5	5	5	7	5	3
3	765	37	13	13	13	13	10	10	10	13	10	8
4	2178	80	22	22	22	22	19	18	18	21	17	14
5	3146	149	34	34	34	34	30	30	29	31	28	22
6	3852	182	36	36	36	36	33	32	32	33	29	24
7	4042	193	33	33	33	33	31	30	30	29	27	23
8	3607	201	31	31	31	31	29	29	29	27	26	21
9	3088	198	28	28	28	28	27	26	26	24	23	19
10	1971	198	26	26	26	26	25	25	25	22	21	18
11	1120	199	25	25	24	24	24	23	23	20	20	17
12	593	217	25	25	25	25	24	24	24	20	20	17
13	279	207	23	23	23	22	22	22	22	18	18	15
14	116	180	19	19	19	19	18	18	18	14	15	12
15	44	218	22	22	22	21	21	21	21	17	17	14
16	17	162	15	15	15	15	15	15	15	12	12	10
17	7	220	20	20	20	19	19	19	19	14	15	13
18	4	208	17	17	17	17	17	18	18	12	14	11
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	157	12	12	12	12	12	13	13	8	10	7
21	2	89	7	7	7	7	7	7	7	11	5	4
22	1	315	21	21	21	21	20	22	22	14	18	13
Total	24966	173	30	30	30	29	27	27	27	26	24	20

REFERENCES

[1] A. Apostolico and R. Giancarlo: The Boyer–Moore–Galil string strategies revisited. SIAM J. Comput. *15* (1986), 1, 98–105.
[2] R. A. Baeza–Yates: Improved string searching. Software – Practice and Experience *19* (1989), 3, 257–271.
[3] R. S. Boyer and J. S. Moore: A fast string searching algorithm. Comm. ACM *23* (1977), 5, 1075–1091.
[4] C. Charras and T. Lecroq: Exact string matching, available at: http://www.dir.univ-rouen.fr/~ lecroq/string.ps (1998).
[5] C. Charras and T. Lecroq: Exact string matching animation in JAVA available at: http://www. dir.univ-rouen.fr/~ charras/string/ (1998).
[6] M. Crochemore, A. Czumaj, L. Gasieniec, L. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: Speeding up two string matching algorithms. Algorithmica *12* (1994), 4, 247–267.
[7] M. Crochemore and T. Lecroq: Tight bounds on the complexity of the Apostolico–Giancarlo algorithm. Inform. Process. Lett. *63* (1997), 4, 195–203.
[8] M. Crochemore and W. Rytter: Text Algorithms. Oxford University Press 1994.
[9] C. Hancart: Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte. These de doctorat de l'Universite de Paris 7, 1993.
[10] R. N. Horspool: Practical fast searching in strings. Software – Practice and Experience *10* (1980), 6, 501–506.
[11] A. Hume and D. Sunday: Fast string searching. Software – Practice and Experience *21* (1991), 11, 1221–1248.
[12] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: Fast pattern matching in strings. SIAM J. of Comput. *6* (1980), 1, 323–350.
[13] Z. Liu, X. Du and N. Ishi: An improved adaptive string searching algorithm. Software – Practice and Experience *28* (1998), 2, 191–198.
[14] B. Melichar: Approximate string matching by finite automata. In: Computer Analysis of Images and Patterns (Lecture Notes in Computer Scince 970), Springer–Verlag, Berlin 1995, pp. 342–349.
[15] T. Raita: Tuning the Boyer–Moore–Horspool string searching algorithm. Software – Practice and Experience *22* (1992), 10, 879–884.
[16] P. D. Smith: Experiments with a very fast substring search algorithm. Software – Practice and Experience *21* (1991), 10, 1065–1074.
[17] G. V. de Smit: A comparison of three string matching algorithms. Software – Practice and Experience *12* (1982), 57–66.
[18] D. M. Sunday: A very fast substring search algorithm. Comm. ACM *33* (1990), 8, 132–142.
[19] W. Rytter: A correct preprocessing algorithm for Boyer–Moore string searching. SIAM J. Comput. *9* (1980), 509–512.
[20] R. F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm. J. Inform. Process. *10* (1987), 3, 173–177.
[21] The code for the BR algorithm is available at: http://java.cms.livjm.ac.uk/homepage/research/cmstberr/
[22] The Calgary Corpus available at: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/

Dr. Thomas Berry, Dr. Somasundaram Ravindran, School of Computing and Mathematical Sciences, Liverpool John Moores University, Byrom Street, Liverpool, L3 3AF. U. K.

e-mails: t.berry@livjm.ac.uk, s.ravindran@livjm.ac.uk