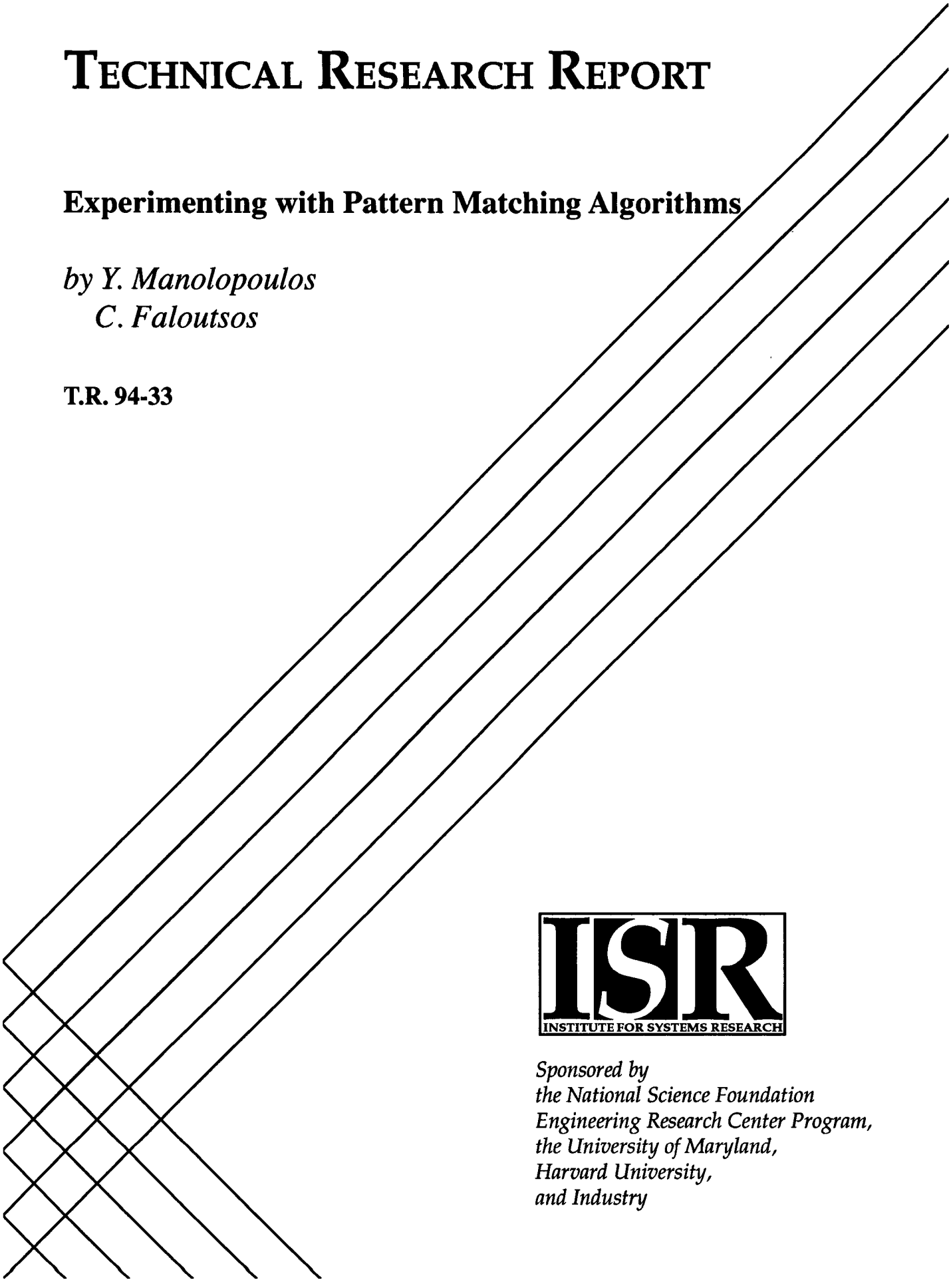
TECHNICAL RESEARCH REPORT

Experimenting with Pattern Matching Algorithms

by Y. Manolopoulos
C. Faloutsos

T.R. 94-33



Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

EXPERIMENTING WITH PATTERN MATCHING ALGORITHMS *

Yannis Manolopoulos † Christos Faloutsos ‡
Computer Science Department &
Institute for Systems Research
University of Maryland
College Park, MD 20742
manolopo,christos@cs.umd.edu

Abstract

Two new pattern matching algorithms based on the Boyer-Moore algorithm are presented. Their performance is compared to that of earlier relevant variants in terms of the number of character comparisons and the required running time by exhaustive simulation. Experimental results show the efficiency of both these two new algorithms.

CR Categories and subject descriptors: E.5., F.2.2, H.3.3
General Terms: Algorithms, Performance

*A preliminary version of this paper has appeared in the Proceedings of the 4th Panhellenic Computer Conference, held in Patras, Greece, December 1993.

†On sabbatical leave from the Department of Informatics, Aristotle University, Thessaloniki, Greece 54006.

‡This research was partially funded by the Institute for Systems Research (ISR), by the National Science Foundation under Grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc.

1 Introduction

Text searching methods have been divided in three categories: (a) *full text scanning*, (b) *text inversion*, and (c) *signature files* [5]. The first category includes the highly honored research topic of *pattern matching*. Milestones in this area are the *BM algorithm* by Boyer and Moore [4] and the *KMP algorithm* by Knuth, Morris and Pratt [8], which appeared in the late 70's. Since then a lot of efforts have been reported; references [1, 2] give pointers towards this rich literature. The reason that the topic remains open until now is its importance in a number of applications, such as in text editors, word processors, lexical analyzers, information retrieval systems, or even in vision for two-dimensional image recognition or in biology for molecular sequence analysis.

New methods have appeared more recently. For example, Sunday reported three new enhancements of the BM algorithm [11]. Then, Smith elaborated on a variant proposed in the latter citation [10]. Finally, a synthetic and exhaustive experimentation on a score of BM variants was reported by Hume and Sunday [7]. In the present paper, we experimentally test and compare the efficiency of some known BM variants as well as two new ones. In the next section we will introduce very briefly some BM variations so that we can explain the two new ones in the third section. Experimental results comparing the algorithms' performance in terms of the number of character comparisons and the required running time are included in the fourth section.

2 Algorithms presentation

The simplest *naive* algorithm uses two pointers which are initialized to point to the first pattern and the first text character. If these characters match, then the pointers are advanced by one position and the comparison of succeeding pattern and text characters continues the apparent way. In case of mismatch, the pattern pointer is initialized, whereas the text pointer is advanced one position over its previous initialization. Thus, it seems as if after a mismatch the pattern slides only one position on top of the text. The authors of references [4, 8] recognized the fact that trying to search for a pattern in the text by shifting the pattern only one position after a mismatch is a memoryless procedure, e.g. the information obtained from the matches and the final mismatch is lost.

Boyer-Moore method

The Boyer-Moore algorithm is based on: (a) the *occurrence heuristic*, and (b) the *match heuristic* (terminology according to Baeza-Yates [2]). These heuristic techniques are used to reposition the pattern after a mismatch. We notice also that a basic characteristic of the BM algorithm is that comparisons start from the rightmost pattern character proceeding towards the leftmost ones.

According to the former heuristic, the pattern has to be shifted after a mismatch, so that on top of the text character where the mismatch occured, a same pattern character will be positioned. For this reason, an *occurrence table* with length equal to the alphabet size is built in a preprocessing phase. Entries of the table corresponding to alphabet characters which do not exist in the pattern, are assigned a value equal to the pattern length. Entries corresponding to characters appearing in the pattern are assigned a value equal to the shorter distance between this character and the last one. Thus, the table entry for the last pattern character is equal to 0.

According to the latter heuristic, we have to take in consideration any existing subpatterns. Thus, after a mismatch the pattern should be shifted so that there are matches for all the characters that matched before, and, in addition, a different pattern character is positioned on top of the text character where the mismatch occured. Again, a *match table* with length equal to that of the pattern is built before actual processing starts. Each table entry takes a value equal to the distance of the specific character x to a different character y such that the characters succeeding x to the pattern end are the same with the characters succeeding y.

Thus, whenever a mismatch is encountered, the pattern pointer is initialized to point to the pattern end, whereas the text pointer is advanced according to the greatest value of the two relevant entries of the occurrence and the match table. It is important, also, to notice that Rytter proposed a more efficient way to calculate the match table [9]. More specifically, to each match table entry we have to add the distance of the specific character from the last pattern character.

Simplified Boyer-Moore algorithm

This method (in short *SBM method*) has been studied experimentally and analytically by Baeza-Yates [3]. It does not use a match table based on the conjecture that in practice patterns are not periodic. A minor detail is that the occurrence table entry for the last pattern character is equal to 1 (instead of 0), to prevent the case of an infinite loop.

Horspool method

This method (*BMH method*) does not use a match table either [6]. In case of mismatch, the shifting size is maximized by using the occurrence table entry for the text character corresponding to the rightmost pattern character (and not for the text character where the mismatch occured). In order to prevent an infinite loop from happening, the occurrence table entry for the last pattern character is equal to the smallest distance from a same character in the pattern. Thus, if the last pattern character is met only once in the pattern, then this value is equal to the pattern length.

Quick Search method

This method (*QS method*) has been proposed recently by Sunday [11]. It is another simple (and therefore quick) method which does not use a match table but has the following two characteristics. First, comparisons start from the leftmost pattern character proceeding to the rightmost one. Second, in case of mismatch, the shifting size is equal to the occurrence table entry for the first text character after the last pattern character by the time of mismatch. For this reason, all the occurrence table values are incremented by a unit.

Maximal Shift method

This method (*MS method*), which too has been proposed by Sunday [11], uses both the occurrence and the match tables. The occurrence table is built and used in the same way as for the QS method. However, the match table is constructed in a much more complicated manner. More specifically, a temporary structure (with length equal to that of the pattern) gives for each pattern character the distance of an identical proceeding character in the pattern. If such a character does not exist, then the value is equal to the order of the character in the pattern. Then the pattern characters are ordered decreasingly according to this temporary table (ties are resolved in favor of the leftmost pattern character). Thus, finally we come up with a transformed pattern, for which the match table is built. The reasoning underneath this method is that by changing the order of the pattern scan, the shift sizes are maximized when the new match table is used.

During searching, the text pointer is initialized to point to the first text character, whereas the pattern pointer points to the first position of the transformed pattern. Comparisons are performed by examining the text characters according to the order of characters of the transformed pattern. After a mismatch, the pattern pointer is initialized again, whereas the text pointer is advanced according to the maximum value of the relevant occurrence and match tables entries.

Optimal Mismatch method

This is another method proposed by Sunday [11] (*OM method*), which uses both the occurrence and match tables. The occurrence table is constructed and used in the same way as for the previous two methods (QS and MS methods). The match table is built for a transformed pattern having its characters ordered ascendingly according to the frequency of appearance in the text. The reasoning behind the method is: make a mismatch as soon as possible and then make a shift as large as possible. Thus, during comparisons priority is given to the most rare text characters. It is evident that this method uses a temporary table during a preprocessing step too. It is noted, that during the same period Baeza-Yates observed independently that the pattern can be scanned in any order, and in reverse letter frequence order in particular [3].

Smith method

This is a method proposed very recently [10] (*BMS method*) and is based on the previous one. In particular, it enhances the OM method in two ways. First, it is language-independent and initially it uses a match table with arbibtrary character ordering. If during searching a pattern character results in a mismatch, it is moved to the front of the match table so that in the next alignment, this character is tried first. Second, in case of a mismatch we choose as shifting size the greatest possible by inspecting all the occurrence table entries, which correspond to the pattern characters.

3 The new methods

In the previous section we have described briefly the BM algorithm and six of its variations. Their main characteristics are summarized in Table 1. As explained all the methods use an occurrence table, although not in an identical way. Also, it is evident that it is necessary to construct an additional auxiliary table for the methods which use a transformed pattern in place of the original. We have crafted two new pattern matching algorithms based on some of the reported techniques. The first one combines characteristics of the OM method and the BMH method; therefore, we call it in short *OMH method.* The second one is based on the OMH and BMS methods; for this reason we call it *OMHS* in short. In the following subsections we examine the new algorithms in more detail.

Method code	Match table	Pattern search direction
BM	Yes	from end to start
SBM	No	from end to start
BMH	No	from end to start
QS	No	from start to end
MS	Yes	character ordering
OM	Yes	character ordering
BMS	No	character ordering
OMH	No	character ordering
OMHS	No	character ordering

Table 1: Characteristics of pattern matching methods based on BM algorithm.

OMH method

The code in 'C' for this method may be found in the Appendix. The method resembles the BMH method since:

• it does not use a match table.
• the occurrence table entry for the last pattern character is equal to the smallest distance from a same character in the pattern, whereas if this character is met only once in the pattern, then this value is equal to the pattern length.
In addition, it has common characteristics with the OM method since:
• it uses an auxiliary table for the text character frequencies to order the pattern characters.
• it builds an auxiliary structure with the pattern characters sorted ascendingly according to the appearance frequences and the distances from the last pattern character.
• it makes comparisons considering the end of the pattern.
The following example will demonstrate the details of the method.

Example. Suppose we seek for the 11-character pattern 'abracadabra' in the text 'abracababra-cadabra'. The occurrence table is illustrated in the Table 2:

...	a	b	c	d	e	...	q	r	s	...
...	3	2	6	4	11	...	11	1	11	...

Table 2: Occurrence table for the OMH method.

Table 3 gives the frequencies of appearance for characters in an english text ([11]). The pattern characters appearing in the first row of Table 4 are ordered according to ascending frequencies as shown in Table 3. Integer numbers in the second row of Table 4 give the distance of the specific character from the last pattern character.

character	a	b	c	d	e	f	g	h	i	j	k	l	m
frequency	8.9	2.3	4.5	3.2	11.1	1.5	2.4	2.9	7.8	0.2	1.1	5.5	3.2
character	n	o	p	q	r	s	t	u	v	w	x	y	z
frequency	6.8	6.9	3.1	0.2	7.4	5.6	7.1	3.6	1.0	1.1	0.3	2.0	0.2

Table 3: Appearance frequencies of characters in english text (%).

b	b	d	c	r	r	a	a	a	a	a
2	9	4	6	1	8	0	3	5	7	10

Table 4: Auxiliary structure with ordered pattern characters.

Searching starts by setting the text pointer to the 11-th character in the text ('a'). The first two pattern characters of the auxiliary structure (the two 'b's) match with their corresponding text characters in the (11-2=) 9th and the (11-9=) 2nd positions. However, there is a mismatch

between the third character of the structure ('d') and the relevant (11-4=) 7th text character ('b'). Thus, we have to shift the text pointer a number of positions shown by the occurrence table entry for the character 'a', which is the text character corresponding to the last pattern character. Now the text pointer shows to the (11+3=) 14th character ('d'). This time, we recognize a mismatch at the first comparison (pattern's 'b' vs. text's 'c'). So, we move the text pointer by four positions as the occurrence table entry for 'd' shows. At this point, we have a perfect match of the pattern and the corresponding portion of the text. □

OMHS method

This new method:
• inherits the characteristics of the previous one, and in addition,
• uses the technique proposed by Smith [10], that is, in case of mismatch we may choose the greatest shifting size by inspecting all the occurrence table entries (which correspond to the pattern characters).
The latter technique has its own processing cost and may result in excess searching time for patterns with many different characters. After experimentation we decided to partially adopt it by considering only two occurrence table entries: the ones for the two text characters which correspond to the last two pattern characters. The following example demonstrates this new algorithm, whereas the 'C' code may be found in the Appendix.

Example. Suppose, again, that we search for the 6-character pattern 'abacab' in the text 'bacabadabacab'. The occurrence table is illustrated in Table 5, whereas the additional auxiliary structure is depicted in Table 6.

...	a	b	c	d	...
...	1	4	2	6	...

Table 5: Occurrence table for the OMHS method.

b	b	c	a	a	a
0	3	2	1	4	5

Table 6: Auxiliary structure with the ordered pattern characters.

Searching starts by initializing the text pointer to point to the 6-th character in the text, whereas the pattern pointer points to the beginning of the auxiliary structure. The first character of the auxiliary structure ('b') does not match to the 6-th text character ('a'). The shifting size is equal to the greatest of the occurrence table entries for the two text characters ('b' and 'a')

corresponding to the two last pattern characters. The two entries are 4 and 1 respectively, thus, we have to advance the text pointer (4-1=) 3 positions as shown by the occurrence table entry for the character 'b'. After one match between the text's 'b' and the pattern's 'b', we have a mismatch between the pattern character 'b' and the text character 'a'. This time, the shift size is equal to the greatest of the two numbers (1-1=) 0 and 4, which correspond to the two text characters ('a' and 'b' respectively). After we advance the text pointer by four positions we have a perfect match of the pattern and the corresponding portion of the text. □

4 Results and Discussion

To compare our new methods to the previous ones, we run an experiment with a methodology similar to that of reference [11]. More specifically, we created an english text of approximate size 65 Kbytes with approximately 7500 distinct ordered lexicographically patterns categorized in 15 classes of length from 1 to 15 characters. We implemented all the methods using the 'C' language on a personal computer with a 68000 microprocessor. We greatly used the code presented in [2, 11] for the known methods. Our cost metrics were the required number of comparisons as well as the required execution time. More specificly, to understand better the merits and pitfalls of each algorithm we counted:

- the number of *direct comparisons*,
- the number of *indirect comparisons*,
- the elapsed *searching time*, and
- the elapsed *preprocessing time*.

The number of direct comparisons (which occur by an 'if ... then ... else' statement) is a classical criterion for analyzing algorithms. However, the total number of comparisons is also a crucial measure, since the number of indirect comparisons (which are performed when accessing the occurrence table) is an important overhead. It is interesting, also, to have a measure of the preprocessing cost which varies significantly between simple and complicated methods. However, if the text size is large (i.e. >10 Kb), then the preprocessing time should be ignored since the searching time is considerably larger.

Tables 7 through 10 depict our results. The values for the number of comparisons (either direct or indirect) and the elapsed time (either for processing or for preprocessing) are produced in the following way: for each pattern class and metric we have divided the measured value by the population of the class and the size of the text. Thus, all values have been normalized and denote the cost per character.

Closer inspection of the results obtained in the next four tables results in the following remarks.

- **Direct comparisons**: in general, the Smith method is the best regardless of the pattern class. Therefore, an interesting problem is to analyze its complexity. More specifically, for small patterns

the ranking is: BMS, OM, QS, and MS, whereas for larger patterns the ranking is: BMS, OMHS, and OM. This means that embedding the technique proposed by Smith in the OMH method, improves the latter substantially for long patterns.

• **Indirect comparisons**: The methods QS, MS, OM, and BMS perform indirect comparisons, whereas the remaining methods (BM, SBM, BMH, OMH, and OMHS) do not. In other words, the relative weight of the advantage of the Smith method is decreased. Thus, the ranking according to the total number of comparisons is: OMHS, OMH, and BM. Let us notice that this ranking does not change with the pattern length. It is interesting, also, to note that the new methods OMH and OMHS outperform the BMH, OM, and BMS methods, although they are chemical unions of the latter ones.

• **Searching time**: For small patterns (up to three characters), the ranking is: QS, OMH, BMH, and SBM, whereas for large patterns the ranking is: OMH, BMH, SBM, and OMHS. That is, the simple and 'quick' QS method becomes slow with increasing pattern length. The new OMH method presents a stable performance outperforming its ancestors BMH and OM regardless of the pattern class.

• **Preprocessing time**: It is evident that it increases with pattern length increasing. The ranking sequence is: BMH, SBM, QS, and BM. We remark, also, that the cost increase for the patterns of length 15 in comparison to the cost for patterns of length 1 is small (ratio 1:2). This observation does not hold for the other methods. For example, for the OM and BMS methods this ratio is 1:35, whereas the performance of the new methods OMH and OMHS is in between.

Concluding, in this paper we have presented the results of an extensive experiment of the most well known pattern matching algorithms based on the Boyer-Moore method. We have described two new variations (the OMH and OMHS methods) by building blocks of previous variations. Summarizing the previous comments with regards to the new methods we have:

• the OMHS method is always the best method if the total number of comparisons is the most important metric. In case we are interested in the number of direct comparisons the OMHS method is second best after the BMS method (for length > 6).

• the OMH method is asymptotically (for length > 3) the best method if searching time is considered to be the most important metric.

Length	Freq	BM	SBM	BMH	QS	MS	OM	BMS	OMH	OMHS
1	26	1.000	1.000	1.000	0.509	0.509	0.509	0.509	1.000	1.000
2	38	0.538	0.552	0.538	0.382	0.384	0.373	0.363	0.536	0.536
3	343	0.370	0.382	0.371	0.294	0.294	0.284	0.271	0.367	0.359
4	880	0.289	0.302	0.290	0.241	0.247	0.234	0.219	0.283	0.271
5	1031	0.239	0.251	0.240	0.209	0.213	0.201	0.184	0.233	0.218
6	1204	0.207	0.218	0.208	0.185	0.190	0.177	0.160	0.201	0.184
7	1193	0.183	0.193	0.183	0.166	0.170	0.159	0.141	0.177	0.159
8	964	0.164	0.173	0.165	0.152	0.154	0.144	0.127	0.158	0.140
9	780	0.150	0.159	0.151	0.141	0.142	0.134	0.115	0.145	0.126
10	533	0.140	0.148	0.141	0.132	0.133	0.125	0.107	0.134	0.115
11	336	0.130	0.138	0.132	0.125	0.124	0.118	0.099	0.126	0.106
12	166	0.122	0.130	0.123	0.117	0.117	0.111	0.093	0.117	0.098
13	90	0.117	0.124	0.118	0.112	0.112	0.107	0.088	0.112	0.093
14	39	0.111	0.117	0.113	0.106	0.105	0.100	0.083	0.106	0.087
15	12	0.104	0.109	0.106	0.101	0.099	0.094	0.077	0.100	0.082

Table 7: Number of direct comparisons per character.

Length	Freq	BM	SBM	BMH	QS	MS	OM	BMS	OMH	OMHS
1	26	1.000	1.000	1.000	1.017	1.017	1.017	1.017	1.000	1.000
2	38	0.538	0.552	0.538	0.732	0.734	0.723	0.708	0.536	0.536
3	343	0.371	0.382	0.371	0.563	0.563	0.553	0.532	0.367	0.359
4	880	0.289	0.302	0.290	0.464	0.469	0.456	0.430	0.283	0.271
5	1031	0.239	0.251	0.240	0.400	0.403	0.392	0.362	0.233	0.218
6	1204	0.207	0.218	0.208	0.354	0.357	0.346	0.314	0.201	0.184
7	1193	0.183	0.193	0.183	0.318	0.320	0.311	0.277	0.177	0.159
8	964	0.164	0.173	0.165	0.290	0.290	0.283	0.249	0.158	0.140
9	780	0.150	0.159	0.151	0.269	0.268	0.261	0.227	0.145	0.126
10	533	0.140	0.148	0.141	0.252	0.251	0.245	0.209	0.134	0.115
11	336	0.130	0.138	0.132	0.238	0.235	0.231	0.195	0.126	0.106
12	166	0.122	0.130	0.123	0.223	0.221	0.217	0.182	0.117	0.098
13	90	0.117	0.124	0.118	0.215	0.212	0.209	0.173	0.112	0.093
14	39	0.111	0.117	0.113	0.202	0.199	0.196	0.163	0.106	0.087
15	12	0.104	0.109	0.106	0.193	0.188	0.186	0.152	0.100	0.082

Table 8: Number of comparisons per character.

Length	Freq	BM	SBM	BMH	QS	MS	OM	BMS	OMH	OMHS
1	26	1.865	1.474	1.473	0.899	2.422	2.423	1.374	1.383	1.853
2	38	0.943	0.789	0.768	0.629	1.682	1.677	0.937	0.719	0.964
3	343	0.650	0.545	0.529	0.483	1.292	1.287	0.705	0.493	0.654
4	880	0.502	0.428	0.410	0.399	1.067	1.062	0.569	0.380	0.497
5	1031	0.414	0.355	0.338	0.344	0.916	0.914	0.479	0.313	0.402
6	1204	0.357	0.307	0.292	0.304	0.807	0.806	0.415	0.269	0.341
7	1193	0.314	0.271	0.258	0.273	0.724	0.725	0.366	0.238	0.296
8	964	0.282	0.243	0.232	0.249	0.658	0.659	0.329	0.213	0.262
9	780	0.258	0.223	0.212	0.231	0.607	0.610	0.299	0.195	0.237
10	533	0.239	0.207	0.197	0.216	0.568	0.572	0.276	0.180	0.216
11	336	0.223	0.194	0.184	0.204	0.533	0.539	0.257	0.169	0.200
12	166	0.209	0.182	0.172	0.191	0.501	0.507	0.240	0.158	0.185
13	90	0.200	0.174	0.165	0.184	0.480	0.488	0.228	0.151	0.175
14	39	0.189	0.164	0.157	0.173	0.451	0.458	0.215	0.144	0.166
15	12	0.178	0.153	0.149	0.165	0.429	0.436	0.201	0.135	0.156

Table 9: Searching time per pattern character (in secs).

Length	Freq	BM	SBM	BMH	QS	MS	OM	BMS	OMH	OMHS
1	26	0.842	0.690	0.687	0.838	1.167	1.056	1.031	0.923	0.923
2	38	0.889	0.700	0.697	0.839	1.345	1.636	1.530	1.128	1.121
3	343	0.947	0.711	0.707	0.849	1.599	2.542	2.354	1.495	1.494
4	880	1.001	0.719	0.714	0.859	2.016	5.575	5.299	2.033	2.033
5	1031	1.056	0.729	0.725	0.869	2.280	7.339	6.979	2.742	2.741
6	1204	1.112	0.738	0.733	0.878	2.666	9.748	9.296	3.624	3.624
7	1193	1.167	0.747	0.742	0.889	3.131	12.276	11.742	4.678	4.677
8	964	1.220	0.757	0.752	0.898	3.636	14.949	14.338	5.903	5.904
9	780	1.275	0.767	0.763	0.907	4.005	17.777	17.084	7.299	7.300
10	533	1.328	0.776	0.773	0.917	4.465	20.530	19.743	8.864	8.864
11	336	1.382	0.785	0.780	0.927	4.918	23.904	23.041	10.600	10.600
12	166	1.429	0.794	0.786	0.935	5.439	26.758	25.816	12.514	12.514
13	90	1.482	0.805	0.797	0.947	5.900	29.508	28.488	14.585	14.588
14	39	1.538	0.813	0.810	0.954	6.367	31.809	30.722	16.856	16.850
15	12	1.588	0.817	0.817	0.967	6.846	36.967	35.862	19.292	19.283

Table 10: Preprocessing time per pattern character (in msec).

Appendix

The names of the variables used in the following fragments of code are summarized in Table 11.

Variable	Definition	Variable	Definition
text	text pointer	ASIZE	alphabet size
tlen	text length	occ	occurrence table
pattern	pattern pointer	freq	frequency table
plen	pattern length	aux	auxiliary structure
MAXPAT	maximum pattern length	auxptr	auxiliary table

Table 11: Names of variables used and definitions.

OMH Method

```
typedef struct pattern_scan_element
        {
        int loc; char c;
        }
        aux;
static aux auxptr[MAXPAT];

void prep(unsigned char *pattern, unsigned char plen)
{
   unsigned char h; register int i;
   register unsigned char *c, *ce; register aux *p, *p1, *ps;
   for (i=0; i<ASIZE; i++) occ[i]=plen;
   for (c=pattern, ce=pattern+plen-1; c<ce; c++) occ[*c]=ce-c;
   for (i=plen-1, p=auxptr; i>=0; i--, pattern++, p++)
   {
      p->loc=i; p->c=*pattern;
   }
   p->c=0; p->loc=0;
   for (p=auxptr+plen-1, ps=auxptr+1; p>=ps; p--)
   {
      for (p1=p-1; p1>=auxptr; p1--)
      {
         if (freq[p1->c-97]<freq[p->c-97]) continue;
         h=p->c; i=p->loc; p->c=p1->c;
         p->loc=p1->loc; p1->c=h; p1->loc=i;
      }
   }
}
```

```
long OMHsearch(unsigned char *text, unsigned int tlen)
{
   register aux *p; register unsigned char *tx;
   register unsigned int found;
   tx=text+plen-1; found=0;
   while (tx<=text+tlen)
   {
      for (p=auxptr; p->c; p++) if ( p->c != *(tx-p->loc) ) goto mismatch;
      found++;
      mismatch: tx+=occ[*tx];
   }
   return found;
}
```

OMHS Method

```
long OMHSsearch(unsigned char *text, unsigned int tlen)
{
   register aux *p; register unsigned char *tx;
   register unsigned int found;
   tx=text+plen-1; found=0;
   while (tx<=text+tlen)
   {
      for (p=auxptr; p->c; p++) if ( p->c != *(tx-p->loc) ) goto mismatch;
      found++;
      mismatch: tx+=(occ[*tx]>occ[*(tx-1)]-1 ? occ[*tx]:occ[*(tx-1)]-1);
   }
   return found;
}
```

Acknowledgement

Thanks are due to Mr. Michael Timoleon for his valuable assistance in the course of this effort.

References

[1] Aho A.V.: "Algorithms for Finding Patterns in Strings", in book by VanLeeuwen J. (editor): *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pp.255-300. Elsevier, 1990.

[2] Baeza-Yates R.A.: "Algorithms for String Searching - a Survey", *ACM SIGIR Forum*, Vol.23, No.3-4, pp.34-58, 1989.

[3] Baeza-Yates R.A.:"Improved String Searching", *Software - Practice and Experience*, Vol.19, No.3, pp.257-271, 1989.

[4] Boyer R.S. and Moore J.S.: "A Fast String Searching Algorithm", *Communications of the ACM*, Vol.20, No.10, pp.762-772, 1977.

[5] Faloutsos C.: "Access Methods for Text", *ACM Computing Surveys*, Vol.17, No.1, pp.49-74, 1985.

[6] Horspool R.N.: "Practical Fast Searching in Strings", *Software - Practice and Experience*, Vol.10, pp.501-506, 1980.

[7] Hume A. and Sunday D.: "Fast String Searching", *Software - Practice and Experience*, Vol.21, No.11, pp.1221-1248, 1991.

[8] Knuth D.E., Morris J.H. and Pratt V.R.: "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, Vol.6, No.2, pp.323-349, 1977.

[9] Rytter W.: "A Correct Preprocessing Algorithm for the Knuth-Morris-Pratt Algorithm", *SIAM Journal on Computing*, Vol.9, pp.509-512, 1980.

[10] Smith P.D.: "Experiments with a Very Fast Substring Search", *Software - Practice and Experience*, Vol.21, No.10, pp.1065-1074, 1991.

[11] Sunday D.: "A very Fast Substring Search Algorithm", *Communications of the ACM*, Vol.33, No.4, pp.132-142, 1990. Also, see technical correspondence, Vol.35, No.4, pp.132-137, 1992.