U N I V E R S I T Y   O F   T A R T U
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Chair of Software Systems

Dmitri Melnikov

String Searching Algorithms

Research paper

Supervisor: Ain Isotamm, PhD

TARTU 2007

Table of Contents

1 Introduction

Representing and storing information as a sequence of characters is simple and natural. Sequences of characters, or texts, can be anything from a newspaper article to specially encoded binary files. Almost anything can be encoded, making texts a perfect solution for storing information.

Storing is not, however, the only operation that can be applied to texts. The problem that occurs often when working with sequences of characters is *pattern matching*. For example, a user is searching for a particular word in an electronic document. The pattern in this case is the word supplied by the user which needs to be matched against the whole document and the task is to find all of it's occurrences. Pattern matching is now a standard feature of any modern editor.

Pattern matching is not, however, limited to document editing. It's applications can be found it data compression, artificial intelligence, spell-checking software and bioinformatics. In bioinformatics DNA sequences are not always searched for exact patterns but can be used in combination with approximation algorithms.

Many different algorithms exist to find the exact pattern inside the text. The goal of this work is to present the most widely used algorithms, explain the ideas behind them and compare their running times on the same sets of input data. Algorithms are implemented in Java language and the input is generated randomly.

2 Base work review

The work by Hendrik Nigul gives a good overview of the problem of string searching. The task is clearly stated and defined. A total of nine algorithms are explained, implemented and tested. Logical conclusions are made from the testing results. Each algorithm is presented as an idea and a strategy for pattern matching that is not difficult to comprehend. The code that comes with the work has no compilation errors. The work is formatted well and uses good language. The work's goal is achieved and the author deserves the highest grade.

3 Problem definition

The formal definition of the pattern matching problem is as follows.

The text is represented as an array $T\,[1..\,n]$ and the pattern as an array $P\,[1..\,m]$ where $m \le n$. The arrays T and P consist of symbols from the alphabet Σ. We say that pattern P occurs in T if there is an s such that

$T\,[s + 1 \,..\, s + m] = P\,[1 \,..\, m]$, where $0 \le s \le n - m$

which means that for every $1 \le j \le m$, $T\,[\,s + j\,] = P\,[\,j\,]$.

The problem is to find every s in T that satisfy these conditions.

3.1 Used Definitions

T and P are often called *strings* and s is called a *shift*.
A *window,* or *current sliding window*, is a part of the text of length m starting at some shift.
We define v to be a *prefix* of the word w if there is such a word u that $w = vu$.
We define v to be a *suffix* of the word w if there is such a word u that $w = uv$.
A word z is a *factor* or a *substring* of a word w when there are words u and v such that $w = uzv$.

4 Algorithms

4.1 The naïve algorithm

This algorithm is simple to understand and implement. The idea is to shift the pattern P in the text T starting at position 0 and ending at $n - m$. At every shift the algorithm checks if the corresponding characters are equal in P and T. If the last check succeeds then a match has been found. This algorithm despite it's simplicity is not the best for this problem. One of it's drawbacks is that is doesn't use the information gained while searching the text although it may be useful later. The naïve algorithm does not do any preprocessing.

4.2 The Rabin-Karp algorithm

The idea behind the Rabin-Karp algorithm relies on the use of a hash function. If we could create such a hash function that would allow us to compare the hash of the pattern with the hashes of the parts of the text in constant time then we would have a linear algorithm (the time to compute hashes should also be linear) since comparing hashes takes constant time.

However, such hash functions are possible only for limited types of input and are impractical. The solution is to compute the hashes modulo q. The drawback is that this requires checking since collisions are possible while computing hashes, that is several sequences of characters may produce the same output when the hash function is applied to them. The checking is done character by character as in the naïve method to make sure that the pattern matches if and only if the hash of the pattern equals the hash of the part of the text. The collisions become less likely to happen when the constant q is chosen as a large prime number.

4.3 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm works by computing a prefix function for the pattern. The prefix function is defined as the length of the longest prefix of the pattern P that is a proper suffix (proper meaning strictly less than) of P_q where $1 \le q \le m$. This function is used to “step over” the characters that cannot possibly match.

The algorithm goes over every character in a text exactly once and keeps track of the current longest prefix. If the characters do not match then the prefix function is used and the algorithm proceeds with a shorter prefix. If they do then the prefix grows until it is equal to m when the pattern is found. Both the matching and preprocessing times of the Knuth-Morris-Pratt algorithm are linear.

4.4 The Boyer-Moore algorithm

This algorithm (along with it's modifications) is considered to be one of the most effective known pattern matching algorithm. The first difference one may observe when comparing it to other algorithms is that it compares the pattern with the text from right to left.

Two preprocessed functions are used: the *good-suffix shift* and *bad-character shift*. When a mismatch occurs at some point $P[j]$ and $T[i+j]$ the good-suffix shift function is used to find the rightmost occurrence of the currently matched suffix S in the pattern such that the character before it is not $P[j]$. If this is not possible then the longest prefix of P is shifted to match the suffix of S. The bad-character shift function is used to shift the window to the right by k so that $P[j-k]$ equals $T[i+j]$. If the pattern contains no occurrences of $T[i+j]$ then the window's new starting position is just after $T[i+j]$.

The algorithm always chooses the maximum possible shift. The Boyer-Moore has an interesting property: the longer the pattern the faster it will usually work.

4.5 The Boyer–Moore–Horspool algorithm

Sometimes simply referred to as *Horspool algorithm*, it is one of the variations of the Boyer-Moore algorithm. The idea is to only use the bad-character shift function to shift the window making it efficient for large alphabets. One of the advantages of Horspool is that it is simpler to implement compared to Boyer-Moore.

4.6 The Turbo-BM algorithm

Also based on Boyer-Moore, Turbo-BM main idea is to remember the factor of the text of the previously matched pattern's suffix. In the case when a mismatch occurs and this previously remembered factor is larger than the currently matched suffix, a turbo-shift can be made. It can be shown that the length of the turbo shift is given as previously remembered factor minus the currently matched suffix.

Turbo-BM requires the same preprocessing as Boyer-Moore does and only constant extra space. It's searching time is linear.

4.7 The Backward Oracle Matching algorithm

Backward Oracle Matching algorithm uses *suffix oracle* of the reverse pattern. Suffix oracle is an automaton built for some sequence that recognizes all the suffixes of that sequence. The advantage of suffix oracle is that it has a minimum number of states and a small number of transitions. During preprocessing the suffix oracle for the reverse pattern is computed. This operation takes linear time. The search phase goes over the characters from right to left with the automaton. When no more transitions are present for the given character the automaton stops. The algorithm then proceeds to compute the shift required using the information about the length of the longest prefix of the pattern and the number of passed states. Backward Oracle Matching is especially fast for long patterns and small alphabets.

4.8 The Shift Or algorithm

The Shift Or algorithm precomputes a bit mask for each character of the alphabet. It then passes each character of the pattern exactly once and uses bitwise OR on the current match mask and the precomputed current character mask. When the highest bit (at position *m*) of the current match mask changes it's state we can say that a match has been found. Because the bit masks are usually kept in a limited size variable the algorithm's performance it best on patterns smaller than some constant. It's also usually better on smaller alphabets.

4.9 The Sunday algorithm

Sometimes called Quick Search, this algorithm is an improvement of Boyer-Moore and Horspool. It uses the same bad-character shift function but when a mismatch happens instead of using the character from the current window it uses the character directly after it. It reaches it's best performance when characters from the text do not occur at all in the pattern making it effective for large alphabets. In practice Sunday works fast and is not difficult to implement.

4.10 java.lang.String.indexOf

To compare the described algorithms with the standard way of finding matches in a text we use indexOf method defined inside the String class of the Java API. This method is often used when a problem of pattern matching is encountered. Based on the information from the tests we shall able to answer the following questions: How fast indexOf really is? Is it better than the described algorithms? Can we guess what algorithm indexOf implements based on the comparison tables?

5 Theoretical comparison

In the study of algorithms the *O-notation* is used very often to describe the worst-case running time. The *O()* function gives an *asymptotic upper bound* for the growth of the algorithm depending on the input size. We take it's definition from [1].

$$O(g(n)) = \{ f(n) : \text{there exists positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}.$$

As shown in [2], the worst-case running time of almost all presented algorithms is *O(mn)*, except Knuth-Morris-Pratt *O(m + n)* and Turbo Boyer-Moore *O(n)*, where *n* is the length of the text and *m* is the length of the pattern. This does not give a good overview, however, since worst-cases do not occur often. The best performances can be very different from worst-cases. For example the best-case of Boyer-Moore is just *O(n / m)*. It is thus more important to compare the algorithms in practice to get a better understanding.

6 Practical comparison

It is possible to receive a lot more knowledge about different algorithms when comparing their running times with each other. Practical analysis gives an interesting overview: algorithms that are fast in theory can be relatively slow and vice versa. All algorithms described above are implemented in Java programming language and are available on the disk that is included with the current work.

The times were measured only for the search algorithms. Each of them finds all occurrences of a pattern inside the text. The time needed to generate input and read the input into memory is not taken into account. All algorithms implement a common interface and receive the same set of input parameters: a pattern and a text, both of type String. The output is saved along with the resulting time for each algorithm. This output is later checked to make sure that it is correct.

The running times depend on the length of the text. The rules and input sizes for the tests are taken from the base work [2]. There are a number of them.

1. We generate 100000000 (95.4 *MB*) symbols from the alphabet of length from the set {1, 2, 4, 8, 16, 32, 64, 128} and the pattern of length from the same set.
2. We are not generating the patterns larger that 8 in length for the alphabet of size 1. The reason for this comes from the fact that some algorithms require very long time to finish on such inputs.
3. Shift Or algorithm only operates on patterns of size less than or equal to 32. This restriction is due to the fact that the bit masks are kept in primitive variables with a limited amount of bits.
4. The input is generated randomly. This means that the input is generated with the given alphabet size and pattern length but the data may be in any order.

Tests were carried out on a computer with a Intel Centrino Duo dual core processor, each core with a clock rate of 1.60 *GHz,* and with a 1*GB* of RAM. The operating system was GNU/Linux and Java Runtime Environment version was 1.5. A total of (10 * 8 * 8 – 32 – 14) 594 test were made in about 28 minutes. The The test were conducted only once and thus there is a possibility that they are not entirely correct, but they do give a good overview and later tests are not likely to differ a great deal. The following table gives the best results for the alphabet size *s* and pattern of length *m*.

s	Pattern length *m*							
	1	2	4	8	16	32	64	128
1	Or	Or	Or	Or				
2	Or	Or	Or	BOM	BOM	BOM	BOM	BOM
4	indexOf	indexOf	indexOf	BOM	BOM	BOM	BOM	BOM
8	indexOf	indexOf	indexOf	Hors	BOM	BOM	BOM	BOM
16	indexOf	indexOf	indexOf	Hors	BOM	BOM	BOM	BOM
32	indexOf	indexOf	indexOf	Hors	Hors	Hors	Hors	BOM
64	indexOf	indexOf	indexOf	BOM	BOM	Hors	Hors	BOM
128	indexOf	indexOf	BOM	BOM	BOM	Hors	Hors	BOM

Table 1: The fastest algorithms depending on alphabet size *s* and pattern length *m*.

7 Conclusion

Table 1 gives interesting results.

It is clear that for very small alphabet sizes and small patterns **Shift Or** algorithm is perfect.

The **indexOf** method from the Java API is best for small patterns on almost all alphabets. It is also worth saying that judging from the tables in *Appendix A* **indexOf** running times differ just slightly as pattern's length grows. This can be a very useful property. However it is not possible to guess how this algorithm is implemented just by looking at the tables, since the the times differ from other algorithms.

Horspool turned out to be good in practice for medium sized alphabets and patterns, despite being a simplification of **Boyer-Moore**. It's modification, **Sunday** is not as fast though.

The **Backward Oracle Matching** algorithm is ideal for long patterns. An interesting observation can be made from the tables in *Appendix A*: the algorithm becomes twice as fast on average when the pattern's size doubles.

Knuth-Morris-Pratt algorithm was relatively slow, but it's running times changed little as pattern sizes grew.

Rabin-Karp and **Naïve** algorithms were the slowest of all.

Many more algorithms for pattern matching problem exist. Only some better known ones were considered in the scope of this work. All algorithms have their weaker and stronger sides and there is no such thing as a perfect string searching algorithm for every type of input. Thus knowledge about the problem and the restrictions posed is necessary for the optimal solution.

Appendix A: Test Tables

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	4075	3153	2311	5615	3615	3164	3713	3844	3959	2183
2	4540	4002	2376	7203	4038	5371	5178	4203	3689	2237
4	5641	6866	2304	9891	7063	6995	7779	4077	6120	2121
8	8422	13166	2357	16160	13371	12736	13582	4268	11277	2163

Table 1: Running times in *ms* using alphabet of size 1 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	2823	3427	2720	5240	4563	3625	2760	5100	3126	2727
2	2401	3823	3111	4940	3718	3066	3634	4361	3922	2401
4	2057	4682	3487	4529	3174	3654	3329	3796	3270	1998
8	2087	4899	3664	5183	2789	3803	4098	2888	1547	1869
16	2129	4894	3297	4889	1260	1790	4114	1396	772	1859
32	2126	4727	3444	4605	1095	2940	2435	1378	445	1855
64	2127	4930	3606	4363	1114	3961	3252	1368	289	
128	2184	5146	3513	4707	938	3832	3078	1178	183	

Table 2: Running times in *ms* using alphabet of size 2 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	1740	2832	2437	4481	4427	3296	2521	5218	2905	2476
2	1484	3031	2898	3937	3096	2125	2250	3783	2732	1986
4	1366	3203	2955	4814	2199	1519	1982	2665	1514	1869
8	1246	3085	2788	5395	1779	1511	1548	2225	832	1881
16	1434	3177	3038	5439	2008	1609	1818	2365	461	1901
32	1389	3149	3002	5210	921	1075	1034	1107	266	1833
64	1255	2989	2887	4628	991	1580	1615	1250	225	
128	1343	3161	2954	5215	568	784	1830	702	148	

Table 3: Running times in *ms* using alphabet of size 4 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	1151	2463	2295	4108	4145	2976	2101	5139	2634	2172
2	940	2432	2423	3611	2424	1690	1574	3041	1751	1872
4	882	2389	2316	5302	1358	977	1084	1739	1003	1879
8	896	2427	2347	5393	823	548	694	1009	644	1861
16	909	2514	2373	5545	590	429	501	781	317	1898
32	940	2473	2364	5286	345	288	344	435	181	1902
64	960	2472	2459	4663	731	608	626	907	173	
128	886	2523	2367	5529	520	395	423	668	157	

Table 4: Running times in *ms* using alphabet of size 8 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	853	2186	2023	3766	3919	2768	1844	5110	2421	1993
2	754	2168	2150	3597	2278	1539	1325	2773	1388	1840
4	708	2135	2153	5353	1113	832	874	1601	908	1910
8	764	2205	2184	5428	640	500	567	892	523	1903
16	732	2241	2150	5325	398	289	345	522	276	1900
32	752	2199	2166	5469	299	216	276	439	156	1914
64	736	2218	2169	5041	335	272	305	458	233	
128	709	2193	2118	5400	239	193	230	314	116	

Table 5: Running times in *ms* using alphabet of size 16 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	699	2022	1954	3541	3740	2615	1795	5051	2272	1991
2	670	2115	2004	3410	2030	1376	1255	2714	1250	1851
4	671	2094	2027	5317	1074	741	757	1366	679	1834
8	670	2034	2035	5379	603	391	479	738	431	1919
16	643	2110	2118	5361	372	228	303	428	248	1929
32	639	2131	2127	5404	203	149	183	269	154	1895
64	726	2061	2075	5340	183	153	204	221	218	
128	640	2089	2022	5363	171	140	168	216	118	

Table 6: Running times in *ms* using alphabet of size 32 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	664	2092	2076	3650	3855	2712	1743	5194	2273	1985
2	613	2048	1989	3573	1927	1359	1168	2670	1254	1874
4	609	2031	2029	5411	1173	696	711	1396	646	1900
8	604	2057	2067	5316	534	359	408	695	356	1921
16	679	2076	1990	5194	274	214	269	374	203	1899
32	640	2088	1994	5386	202	127	166	241	176	1923
64	663	2050	2022	5502	164	143	156	196	221	
128	682	2061	1998	5479	137	127	137	159	108	

Table 7: Running times in *ms* using alphabet of size 64 and text of size 95 *MB*

Pattern size	indexOf	Naive	KMP	RK	BM	Hors	Sunday	TBM	BOM	Or
1	617	1962	1964	3707	4008	2643	1759	5047	2306	1906
2	621	2060	2003	3485	1941	1344	1173	2618	1208	1902
4	607	2032	1928	5205	954	660	685	1291	586	1832
8	587	2219	1920	5188	492	344	402	664	312	1833
16	587	1941	1920	5183	255	190	220	344	174	1831
32	588	1940	1916	5177	143	111	130	189	125	1830
64	586	1936	1916	5146	170	159	162	204	204	
128	585	1935	1915	5170	126	111	120	133	108	

Table 8: Running times in *ms* using alphabet of size 128 and text of size 95 *MB*

Appendix B: Electronic Version

The electronic version of this work can be found at the following address:
http://uuslepo.it.da.ut.ee/~zx/stringsearch.tar.gz

The following files are included:

- paper/ - electronic version
  - strmatch.odt - Open Office document
  - strmatch.pdf - PDF document

- StringSearch/ - contains source code
  - src/aa/algorithms/ - interface for the algorithms and main class
  - src/aa/algorithms/pool - different algorithms
  - src/aa/tester - testing class
  - src/aa/util/generator - input generation class
  - src/aa/util/reader - class for reading input into memory
  - src/aa/util/results - classes for keeping and formating test results
  - src/aa/util/timer - timer class

Apache Ant is required to build the project. Ant is available at http://ant.apache.org

To compile the project with Ant type in the terminal in the StringSearch/ directory:

```
$ ant compile
```

To run the tests type in the terminal in the StringSearch/ directory:

```
$ ant TestRun
```

References

[1] Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithm (2nd edition).* MIT Press, 2001.

[2] Nigul, H. *Alamsõne otsimise algoritmid: praktiline analüüs.* Tartu, 2004

[3] http://en.wikipedia.org/wiki/String_searching_algorithm, November 2007

[4] http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/indexen.htm, November 2007

[5] http://www-igm.univ-mlv.fr/~lecroq/string/index.html, November 2007