Algorithm Analysis Project
String Matching

Andrei-Gabriel Mitran
andreigabrielmitran@gmail.com

Faculty of Automatic Control and Computer Science, Politehnica University of Bucharest

Abstract. A brief overview of the importance and usage of string matching algorithms, particularly the Knuth-Morris-Pratt (KMP) algorithm and the Boyer-Moore algorithm

Keywords: string · pattern · match · shift · KMP algorithm · Boyer-Moore algorithm · suffix · prefix · browser

1 Introduction

1.1 Description

String matching has always been incredibly important in computer science, having an essential role in various real-life problems. Generally, the problem comes down to finding occurrences of a substring (typically referred to as a "pattern") within a bigger string.

1.2 Real-life examples

When looking for such a pattern in a file, a database, or a browser, pattern searching algorithms are used in order to find said pattern. Another fairly interesting usage of string matching is in the field of bioinformatics, in order to find repeating sequences of DNA [1].

1.3 Chosen solution(s)

The algorithms that are going to be discussed below, i.e., KMP and Boyer-Moore, approach the problem of string matching very differently, however their goal is much the same: to not match each individual character of the substring to the string and reduce the amount of comparisons that are required.

1.4 Evaluation

A number of input tests are going to be used to generate outputs. An input consists of a pattern and a text in which the pattern is searched (a needle and a haystack, if you will).
An output is considered correct if and only if it contains the exact number of times that the pattern is found in the text, as well as every single starting position of said pattern. Further evaluation is done by looking at how fast the algorithms perform their task.

2 Solutions

2.1 Description

KMP Algorithm

Approach The basic idea of this algorithm is that when finding a character in the string that does not appear in the substring (known as a mismatch), after some matches have already occurred, we avoid doing unnecessary matching of characters that we already know would match anyway.

This takes advantage of the degenerative property: same sub-patterns appearing more than once in the pattern (take, for example, strings that are made up of only one repeating character).

The need for preprocessing arises, because we would need to know how many characters to avoid checking again. This requires constructing a vector that would hold this information, which would be of size m.

Steps [2]

1. Preprocessing: In the vector (v), at position i, we store the length of the maximum matching proper prefix, which is also a suffix of the sub-pattern that starts at position 0 and ends at position i.
2. After that we start matching the text and the pattern, using two different iterators (i and j) and incrementing both of them while the strings match.
3. When encountering a mismatch, we know that all character prior to the mismatch match (the characters starting from 0 and ending at j - 1 match the characters starting at i - j and ending at i - 1).
4. We can skip over attempting to match v[j - 1] characters with the string.

Improvements [3] By introducing a failure function for each character, an improved version of the algorithm can be constructed. If a mismatch occurs at the character **a** in the main string, then the failure function table is checked for this character at the index j in the pattern.

In short, with the added condition that the character after the prefix is **a**, the function returns the length of the longest substring ending at j, matching a prefix of the pattern.

As such, only a constant number of operations are performed between the processing of each index of the text, since character **a** does not need to be checked again in the next phase.

This makes the algorithm real-time optimized.

Boyer-Moore Algorithm

Approach Like KMP, preprocessing is required. However, this algorithm can best be described as two different approaches merged into one (the bad character rule and the good suffix rule). The key is to match the tail of the pattern and not the head, skipping along the main string in jumps of multiple characters.

Steps In summary, we keep checking if the pattern is matching and, if a mismatch occurs, the pattern is shifted to the right. The shift rules, which are implemented as constant-time table lookups, are dictated by the two approaches. Interestingly, the approaches can both be used separately for string searching, but they complement each other in order to create Boyer-Moore.

1. Bad character rule
   (a) Description: The bad character rule If a mismatch occurs at character **a** in the main string, the next occurrence (to the left) of **a** is found in the pattern and the said pattern gets shifted to the right so that the characters match. If no other occurrence of **a** is found, the shift is moved
   (b) Preprocessing: Many methods on how the table is constructed exist. A simple constant-time lookup solution is possible (a 2D table indexed by the character and by the index in the pattern). The table would return the next highest index at which the character appears again. If no valid index is returned then there is no other occurrence.
2. The good suffix rule
   (a) Description [4]: If a mismatch occurs:
      i. We find the rightmost copy of the sub-pattern **s** that matched with the main string (until the mismatch) in the pattern, making sure it is not a suffix of the given pattern and the following character after it is not the same one that caused the mismatch. We shift the pattern to the right so that the copy is now aligned.
      ii. If the copy does not exist, then we shift the left end of the given pattern past the left end of the sub-pattern by the least amount so that a prefix of the shifted pattern matches a suffix of **s** in the main string.
      iii. If the previous step is not possible, then shift by the length of the pattern.
      iv. If the entire pattern is matching after the previous step, shift again by the least amount so that a proper (not the entire string) prefix of the shifted pattern matches a suffix of the occurrence of the pattern in the main string.
      v. If the previous step is not possible, shift past **s**.

(b) Preprocessing: This rule requires two tables: one for the general case, and another for when the general case returns nothing of importance or a match occurs. The first would return the largest position less than the length of the pattern such that the sub-pattern matches a suffix of what comes before the character at the i-th position in the vector, and the second one would return the length of the largest suffix of the sub-pattern that is also a prefix of the pattern. Both tables would return 0, if nothing is found.

Improvements [5] Further improvements, such as the Galil rule, can be made. It is a simple, but effective change.

2.2 Complexities

Let n be the size of the string/text in which to search for the pattern, and m be the size of the substring/pattern that is being searched.
For KMP, it has a complexity of $\Theta(m)$ for preprocessing + $\Theta(n)$ for matching.
For Boyer-Moore, it has a complexity of $\Theta(m)$ for preprocessing + $O(mn)$ for matching (worst case) or $\Omega(\frac{n}{m})$ for matching (best case). The worst case occurs when both the pattern and text are made up of the same repeated character, although the Galil rule solves this issue. The best case occurs when all the characters of the text and pattern are different.

2.3 Advantages and disadvantages

If the alphabet (the characters that can appear in your strings) is relatively small, like the aforementioned DNA sequences, there is a higher chance that the search patterns contain reusable sub-patterns. As such, for strings such as binary strings (made up of only 1s and 0s), the KMP algorithm performs better, in theory.
The longer the search pattern is, the better the Boyer-Moore algorithm performs. It can be sub-linear for such cases. Natural text (such as English), makes this algorithm perform better as well, as big jumps can occur.
The preprocessing is a drawback for both algorithms. For small patterns and texts, the naive string searching algorithm may be preferred. The naive algorithm works by simply checking each character of the pattern against each character of the text.

3 Evaluation

3.1 Test construction

The tests are constructed by using an online string generator [6]. Random large strings are generated and the patterns are either inserted into them manually or they were already there enough number of times (like the English word 'the' in any English text).
There are a total of 25 tests, with every 5 serving a different purpose:

1. 5 tests in which the pattern found only a couple of times/not found
2. 5 tests in which the pattern is found many times
3. 5 tests to highlight the preprocessing
4. 5 tests to highlight KMP
5. 5 tests to highlight Boyer-Moore

3.2 System specification

The tests are run inside a virtual Ubuntu machine using Oracle VM VirtualBox Manager (Version 6.1.40 r154048 (Qt5.6.2)).

Hardware

1. CPU: AMD Ryzen 7 4800H with Radeon Graphics (2.90 GHz)
2. Available memory/Total memory (inside the VM): 1019/3924

Software

1. OS (VM): Ubuntu 22.04.1 LTS, jammy
2. Visual Studio Code (version 1.74.1)

4 Illustration and explanation

The Boyer-Moore algorithm was implemented using the two heuristics separately. The bad character implementation is referred to as BMBC and the good suffix implementation is referred to as BMGS. All results are in microseconds.

	KMP		BMBC		BMGS	
Test number	Search Time	Preproc Time	Search Time	Preproc Time	Search Time	Preproc Time
1	1	0	1	1	1	1
2	0	0	0	0	1	0
3	1	0	1	1	1	0
4	9	0	2	0	10	0
5	1	0	2	0	2	0
6	99	1	26	1	67	0
7	802	1	1197	0	409	0
8	744	0	653	1	534	1
9	56	0	45	0	41	0
10	539	0	454	0	472	0

Fig. 1. The results of the first 10 tests

When it comes to the alphabet size, it is specified exactly only when necessary.

If the text is simply in English, it is set to 52 or 26 (depending on whether both upper and lowercase letters are present or not). When multiple punctuation characters are included, it is set to 60.

Test number	Text length	Pattern length	Alphabet size (approx)	Total matches
1	61	4	52	2
2	53	1	1	53
3	91	3	52	0
4	3090	7	52	20
5	39	3	4	3
6	30430	10	26	43
7	253964	23	4	11040
8	211663	4	4	30287
9	16132	3	52	157
10	146170	3	60	2130

Fig. 2. The first 10 tests' structure

These tests are simply meant to check if the algorithms work. They work on big inputs such as tests 7, 8 and 10, as well as small special tests such as 2 (when the pattern matches the whole text) and 3, when the pattern does not match at all. Something to note is the fact that, the good suffix implementation of the Boyer-Moore algorithm beats KMP every time. In some instances, the bad character implementation performs better than the good suffix implementation, and in others, it performs worse than KMP.

	KMP		BMBC		BMGS	
Test number	Search Time	Preproc Time	Search Time	Preproc Time	Search Time	Preproc Time
11	29	21	21	13	14	81
12	187	100	105	62	68	408
13	200	288	213	123	192	804
14	1504	1009	1328	605	683	3892
15	2951	2022	2090	1227	1479	7981

Fig. 3. The results of tests 11 - 15

These tests were constructed in order to highlight the preprocessing time. As observed in the above table, BMGS takes a lot of time in order to preprocess the pattern. Surprisingly, KMP comes second when it come to both preprocessing time and total execution time. While BMGS is the fastest when it comes to searching, the one that has the best preprocessing time and overall execution time is BMBC.

Test number	Text length	Pattern length	Alphabet size (approx)	Total matches
11	10001	10000	26	1
12	50002	50000	26	1
13	100003	100000	26	1
14	500004	500000	26	1
15	999005	999000	26	1

Fig. 4. The structure of tests 11 - 15

As seen above, the pattern length is very big. It is almost equal to the text's. This was done to simply observe the fact that, while BMGS is generally the fastest, for extremely long patterns, BMBC is preferred.

	KMP		BMBC		BMGS	
Test number	Search Time	Preproc Time	Search Time	Preproc Time	Search Time	Preproc Time
16	4692	1	7790	0	1189	1
17	5414	0	1988	1	1159	1
18	6374	5	5444	1	806	8
19	7563	1	9088	1	1319	1
20	3742	562	3540	137	843	1146

Fig. 5. The results of tests 16 - 20

These tests were constructed in order to highlight KMP. However, as it can be easily observed, BMGS is faster. While KMP beats BMBC in most cases, it never manages to run faster than BMGS. This may be due to the test restrictions, as the strings cannot have a length greater than 1000000 characters.

Test number	Text length	Pattern length	Alphabet size (approx)	Total matches
16	999997	22	2	3
17	999892	45	4	6
18	996824	720	3	5
19	999999	98	2	1
20	933704	111840	4	4

Fig. 6. The structure of tests 16 - 20

The pattern's length relative to the alphabet's size is much smaller. This should have highlighted the fact that KMP runs better in such cases, as most tests are either DNA strings or binary strings. However, as mentioned before, this was not the case. Considering actual DNA sequences can be a lot larger, the preprocessing part of both BM algorithms may be the deciding factor or, perhaps, the fact that the two BM approaches are separate makes one of them faster.

	KMP		BMBC		BMGS	
Test number	Search Time	Preproc Time	Search Time	Preproc Time	Search Time	Preproc Time
21	2914	0	3237	1	1316	0
22	3635	20	20804417	13	13642594	59
23	2980	4	176	5	1444	18
24	3065	5	151	4	1163	22
25	2942	0	2185	0	2513	0

Fig. 7. The results of tests 21 - 25

These tests were meant to highlight the Boyer-Moore algorithms performance against KMP, especially BMGS. Test 22, however, to show the major flaw of either implementation. Test 22's main feature is the fact that the pattern is fairly large and it matches every single time. In other words, it is the algorithm's worst case and it shows, reaching the order of seconds instead of microseconds. In this case, KMP is preferred.

Test number	Text length	Pattern length	Alphabet size (approx)	Total matches
21	999810	2	52	385
22	999321	10000	1	989322
23	998958	2279	52	1
24	999086	2835	52	1
25	999437	3	62	2

Fig. 8. The structure of tests 21 - 25

The other tests, however, were designed to show that for long texts and small to medium patterns, BMGS works best. If a text is large enough, even a 2000-long string can resemble a word in the text. In such cases, when the alphabet is large as well, the Boyer-Moore algorithm shows its potential.

5 Conclusion

All in all, each algorithm has its advantages and disadvantages. Personally, I would opt for BMGS most of the time, as it is reliably the fastest. In cases when the pattern and text are relatively equal, I would go for BMBC instead. Finally, when both the pattern and text are made up of only 1 character, KMP is by far the most efficient.
KMP usage when it comes to small alphabets may not reflect in these tests, however, in practice, it should still be utilized over Boyer-Moore. This may be shown for bigger inputs, which may be tested again at a later date.

References

1. Gaston H. Gonnet S.: Some string matching problems from Bioinformatics which still need better solutions. Journal: Journal of Discrete Algorithms, Volume 2, Issue 1, pp. 3–15, (2004) https://doi.org/10.1016/S1570-8667(03)00062-5
2. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/.
   Last accessed 22 Dec 2022
3. https://www.geeksforgeeks.org/real-time-optimized-kmp-algorithm-for-pattern-searching/?ref=rp.
   Last accessed 22 Dec 2022
4. Gusfield, Dan (1999) [1997], "Chapter 2 - Exact Matching: Classical Comparison-Based Methods", Algorithms on Strings, Trees, and Sequences (1 ed.), Cambridge University Press, pp. 19–21. ISBN 0521585198
5. Galil, Z. (September 1979). "On improving the worst case running time of the Boyer–Moore string matching algorithm". Comm. ACM. New York: Association for Computing Machinery. 22 (9): 505–508. https://doi.org/10.1145/359146.359148
6. https://codebeautify.org/generate-random-string
   Last accessed 22 Dec 2022