Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Español
فارسی
Français
Bahasa Indonesia
Italiano
Lietuvių
日本語
Polski
Русский
Српски / srpski
中文
Edit links

Boyer–Moore string search algorithm

From Wikipedia, the free encyclopedia

For the Boyer-Moore theorem prover, see Nqthm.

In computer science, the **Boyer–Moore string search algorithm** is an efficient string searching algorithm that is the standard benchmark for practical string search literature.[1] It was developed by Robert S. Boyer and J Strother Moore in 1977.[2] The algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. The Boyer-Moore algorithm uses information gathered during the preprocess step to skip sections of the text, resulting in a lower constant factor than many other string algorithms. In general, the algorithm runs faster as the pattern length increases. The key features of the algorithm are to match on the tail of the pattern rather than the head, and to skip along the text in jumps of multiple characters rather than searching every single character in the text.

Contents [hide]

Definitions [edit]

- ***S[i]*** denotes the character at index *i* of string *S*, counting from 1.
- ***S[i..j]*** denotes the substring of string *S* starting at index *i* and ending at *j*, inclusive.
- A *prefix* of *S* is a substring *S*[1..*i*] for some *i* in range [1, *n*], where *n* is the length of *S*.
- A *suffix* of *S* is a substring *S*[*i*..*n*] for some *i* in range [1, *n*], where *n* is the length of *S*.
- The string to be searched for is called the *pattern* and is denoted by ***P***. Its length is ***n***.
- The string being searched in is called the *text* and is denoted by ***T***. Its length is ***m***.
- An *alignment* of *P* to *T* is an index *k* in *T* such that the last character of *P* is aligned with index *k* of *T*.
- A *match* or *occurrence* of *P* occurs at an alignment if *P* is equivalent to *T*[(*k*-*n*+1)..*k*].

```
ANPANMAN-
PAN- - - - - -
- PAN- - - - -
- - PAN- - - -
- - - PAN- - -
- - - - PAN- -
- - - - - PAN-
```

Alignments of pattern **PAN** to text **ANPANMAN**, from **k=3** to **k=8**. A match occurs at **k=5**.

Description [edit]

The Boyer-Moore algorithm searches for occurrences of ***P*** in ***T*** by performing explicit character comparisons at different alignments. Instead of a brute-force search of all alignments (of which there are ***m - n + 1***), Boyer-Moore uses information gained by preprocessing ***P*** to skip as many alignments as possible.

Previous to the introduction of this algorithm, the usual way to search within text was to examine each character

of the text for the first character of the pattern. Once that was found the subsequent characters of the text would be compared to the characters of the pattern. If no match occurred then the text would again be checked character by character in an effort to find a match. Thus almost every character in the text needs to be examined.

The key insight in this algorithm is that if the end of the pattern is compared to the text then jumps along the text can be made rather than checking every character of the text. The reason that this works is that in lining up the pattern against the text, the last character of the pattern is compared to the character in the text. If the characters do not match there is no need to continue searching backwards along the pattern. If the character in the text does not match any of the characters in the pattern, then the next character to check in the text is located n characters farther along the text, where n is the length of the pattern. If the character is in the pattern then a partial shift of the pattern along the text is done to line up along the matching character and the process is repeated. The movement along the text in jumps to make comparisons rather than checking every character in the text decreases the number of comparisons that have to be made, which is the key to the increase of the efficiency of the algorithm.

More formally, the algorithm begins at alignment ***k = n***, so the start of ***P*** is aligned with the start of ***T***. Characters in ***P*** and ***T*** are then compared starting at index ***n*** in ***P*** and ***k*** in ***T***, moving backward: the strings are matched from the end of ***P*** to the start of ***P***. The comparisons continue until either the beginning of ***P*** is reached (which means there is a match) or a mismatch occurs upon which the alignment is shifted to the right according to the maximum value permitted by a number of rules. The comparisons are performed again at the new alignment, and the process repeats until the alignment is shifted past the end of ***T***, which means no further matches will be found.

The shift rules are implemented as constant-time table lookups, using tables generated during the preprocessing of ***P***.

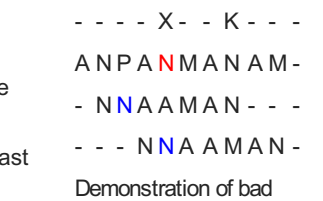
Shift Rules [edit]

A shift is calculated by applying two rules: the bad character rule and the good suffix rule. The actual shifting offset is the maximum of the shifts calculated by these rules.

The Bad Character Rule [edit]

Description [edit]

The bad-character rule considers the character in ***T*** at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in ***P*** is found, and a shift which brings that occurrence in line with the mismatched occurrence in ***T*** is proposed. If the mismatched character does not occur to the left in ***P***, a shift is proposed that moves the entirety of ***P*** past the point of mismatch.



Demonstration of bad character rule with pattern **NNAAMAN**.

Preprocessing [edit]

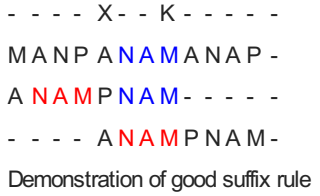
Methods vary on the exact form the table for the bad character rule should take, but a simple constant-time lookup solution is as follows: create a 2D table which is indexed first by the index of the character ***c*** in the alphabet and second by the index ***i*** in the pattern. This lookup will return the occurrence of ***c*** in ***P*** with the next-highest index ***j < i*** or ***-1*** if there is no such occurrence. The proposed shift will then be ***i - j***, with ***O(1)*** lookup time and ***O(kn)*** space, assuming a finite alphabet of length k.

The Good Suffix Rule [edit]

Description [edit]

The good suffix rule is markedly more complex in both concept and implementation than the bad character rule. It is the reason comparisons begin at the end of the pattern rather than the start, and is formally stated thus:[3]

> Suppose for a given alignment of ***P*** and ***T***, a substring ***t*** of ***T*** matches a suffix of ***P***, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy ***t'*** of ***t*** in ***P*** such that ***t'*** is not a suffix of ***P*** and the character to the left of ***t'*** in ***P*** differs from the character to the left of ***t*** in ***P***.



Demonstration of good suffix rule with pattern **ANAMPNAM**.

> Shift ***P*** to the right so that substring ***t'*** in ***P*** aligns with substring ***t*** in ***T***. If ***t'*** does not exist, then shift the left end of ***P*** past the left end of ***t*** in ***T*** by the least amount so that a prefix of the shifted pattern matches a suffix of ***t*** in ***T***. If no such shift is possible, then shift ***P*** by ***n*** places to the right. If an occurrence of ***P*** is found, then shift ***P*** by the least amount so that a *proper* prefix of the shifted ***P*** matches a suffix of the occurrence of ***P*** in ***T***. If no such shift is possible, then shift ***P*** by ***n*** places, that is, shift ***P*** past ***t***.

Preprocessing [edit]

The good suffix rule requires two tables: one for use in the general case, and another for use when either the general case returns no meaningful result or a match occurs. These tables will be designated ***L*** and ***H*** respectively. Their definitions are as follows:[3]

> For each ***i***, ***L[i]*** is the largest position less than ***n*** such that string ***P[i..n]*** matches a suffix of ***P[1..L[i]]*** and such that the character preceding that suffix is not equal to ***P[i-1]***. ***L[i]*** is defined to be zero if there is no position satisfying the condition.

> Let ***H[i]*** denote the length of the largest suffix of ***P[i..n]*** that is also a prefix of ***P***, if one exists. If none exists, let ***H[i]*** be zero.

Both of these tables are constructible in ***O(n)*** time and use ***O(n)*** space. The alignment shift for index ***i*** in ***P*** is given by ***n - L[i]*** or ***n - H[i]***. ***H*** should only be used if ***L[i]*** is zero or a match has been found.

The Galil Rule [edit]

A simple but important optimization of Boyer-Moore was put forth by Galil in 1979.[4] As opposed to shifting, the Galil rule deals with speeding up the actual comparisons done at each alignment by skipping sections that are known to match. Suppose that at an alignment k_1, ***P*** is compared with ***T*** down to character ***c*** of ***T***. Then if ***P*** is shifted to k_2 such that its left end is between ***c*** and k_1, in the next comparison phase a prefix of ***P*** must match the substring $T[(k_2 - n)..k_1]$. Thus if the comparisons get down to position k_1 of ***T***, an occurrence of ***P*** can be recorded without explicitly comparing past k_1. In addition to increasing the efficiency of Boyer-Moore, the Galil rule is required for proving linear-time execution in the worst case.

Performance [edit]

The Boyer-Moore algorithm as presented in the original paper has worst-case running time of ***O(n+m)*** only if the pattern does *not* appear in the text. This was first proved by Knuth, Morris, and Pratt in 1977,[5] followed by Guibas and Odlyzko in 1980[6] with an upper bound of ***5m*** comparisons in the worst case. Richard Cole gave a proof with an upper bound of ***3m*** comparisons in the worst case in 1991.[7]

When the pattern *does* occur in the text, running time of the original algorithm is ***O(nm)*** in the worst case. This is easy to see when both pattern and text consist solely of the same repeated character. However, inclusion of the Galil rule results in linear runtime across all cases.[4][7]

Implementations [edit]

Various implementations exist in different programming languages. In C++, Boost provides the generic Boyer–Moore search implementation under the *Algorithm* library. In Go (programming language) there is an implementation in search.go. D (programming language) uses a BoyerMooreFinder for predicate based matching within ranges as a part of the Phobos Runtime Library.

Below are a few simple implementations.

[Python implementation]	[show]
[C implementation]	[show]
[Java implementation]	[show]

Variants [edit]

The Boyer–Moore–Horspool algorithm is a simplification of the Boyer–Moore algorithm using only the bad character rule.

The Apostolico–Giancarlo algorithm speeds up the process of checking whether a match has occurred at the given alignment by skipping explicit character comparisons. This uses information gleaned during the pre-processing of the pattern in conjunction with suffix match lengths recorded at each match attempt. Storing suffix match lengths requires an additional table equal in size to the text being searched.

See also [edit]

- Knuth–Morris–Pratt string search algorithm
- Boyer–Moore–Horspool string search algorithm
- Apostolico–Giancarlo string search algorithm
- Aho–Corasick multi-pattern string search algorithm
- Rabin–Karp multi-pattern string search algorithm
- Suffix trees

References [edit]

1. ^ Hume; Sunday (November 1991). "Fast String Searching". *Software—Practice and Experience* **21** (11): 1221–1248.
2. ^ Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm.". *Comm. ACM* (New York, NY, USA: Association for Computing Machinery) **20** (10): 762–772. doi:10.1145/359842.359859. ISSN 0001-0782.
3. ^ [a] [b] 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
4. ^ [a] [b] Galil, Z (September 1979). "On improving the worst case running time of the Boyer-Moore string matching algorithm". *Comm. ACM* (New York, NY, USA: Association for Computing Machinery) **22** (9): 505–508. doi:10.1145/359146.359148. ISSN 0001-0782.
5. ^ Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". *SIAM Journal on Computing* **6** (2): 323–350. doi:10.1137/0206024.
6. ^ Guibas, Odlyzko; Odlyzko, Andrew (1977). "A new proof of the linearity of the Boyer-Moore string searching algorithm". *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA: IEEE Computer Society): 189–195. doi:10.1109/SFCS.1977.3.
7. ^ [a] [b] Cole, Richard (September 1991). "Tight bounds on the complexity of the Boyer-Moore string matching algorithm". *Proceedings of the 2nd annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA: Society for Industrial and Applied Mathematics): 224–233. ISBN 0-89791-376-0.

External links [edit]

- Original paper on the Boyer-Moore algorithm
- An example of the Boyer-Moore algorithm from the homepage of J Strother Moore, co-inventor of the algorithm
- Richard Cole's 1991 paper proving runtime linearity

Categories: String matching algorithms | Algorithms on strings