String-searching algorithm

In computer science, **string-searching algorithms**, sometimes called **string-matching algorithms**, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) Σ. Σ may be a human language alphabet, for example, the letters *A* through *Z* and other applications may use a *binary alphabet* (Σ = {0,1}) or a *DNA alphabet* (Σ = {A,C,G,T}) in bioinformatics.

In practice, the method of feasible string-search algorithm may be affected by the string encoding. In particular, if a variable-width encoding is in use, then it may be slower to find the *N*th character, perhaps requiring time proportional to *N*. This may significantly slow some search algorithms. One of many possible solutions is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed to avoid it.

Contents

Kinds of searching

The most basic case of string searching involves one (often very long) string, sometimes called the *haystack*, and one (often very short) string, sometimes called the *needle*. The goal is to find one or more occurrences of the needle within the haystack. For example, one might search for *to* within:

```
Some books are to be tasted, others to be swallowed, and some few to be chewed and digested.
```

One might request the first occurrence of "to", which is the fourth word; or all occurrences, of which there are 3; or the last, which is the fifth word from the end.

Very commonly, however, various constraints are added. For example, one might want to match the "needle" only where it consists of one (or more) complete words—perhaps defined as *not* having other letters immediately adjacent on either side. In that case a search for "hew" or "low" should fail for the example sentence above, even though those literal strings do occur.

Another common example involves "normalization". For many purposes, a search for a phrase such as "to be" should succeed even in places where there is something else intervening between the "to" and the "be":

- More than one space
- Other "whitespace" characters such as tabs, non-breaking spaces, line-breaks, etc.
- Less commonly, a hyphen or soft hyphen
- In structured texts, tags or even arbitrarily large but "parenthetical" things such as footnotes, list-numbers or other markers, embedded images, and so on.

Many symbol systems include characters that are synonymous (at least for some purposes):

- Latin-based alphabets distinguish lower-case from upper-case, but for many purposes string search is expected to ignore the distinction.
- Many languages include ligatures, where one composite character is equivalent to two or more other characters.
- Many writing systems involve diacritical marks such as accents or vowel points, which may vary in their usage, or be of varying importance in matching.
- DNA sequences can involve non-coding segments which may be ignored for some purposes, or polymorphisms that lead to no change in the encoded proteins, which may not count as a true difference for some other purposes.
- Some languages have rules where a different character or form of character must be used at the start, middle, or end of words.

Finally, for strings that represent natural language, aspects of the language itself become involved. For example, one might wish to find all occurrences of a "word" despite it having alternate spellings, prefixes or suffixes, etc.

Another more complex type of search is regular expression searching, where the user constructs a pattern of characters or other symbols, and any match to the pattern should fulfill the search. For example, to catch both the American English word "color" and the British equivalent "colour", instead of searching for two different literal strings, one might use a regular expression such as:

```
colou?r
```

where the "?" conventionally makes the preceding character ("u") optional.

This article mainly discusses algorithms for the simpler kinds of string searching.

A similar problem introduced in the field of bioinformatics and genomics is the maximal exact matching (MEM).[1] Given two strings, MEMs are common substrings that cannot be extended left or right without causing a mismatch.[2]

Basic classification of search algorithms

The various algorithms can be classified by the number of patterns each uses.

Single-pattern algorithms

Let m be the length of the pattern, n be the length of the searchable text and $k = |\Sigma|$ be the size of the alphabet.

Algorithm	Preprocessing time	Matching time[1]	Space
Naive string-search algorithm	none	Θ(mn)	none
Rabin–Karp algorithm	Θ(m)	average Θ(n + m), worst Θ((n−m)m)	O(1)
Knuth–Morris–Pratt algorithm	Θ(m)	Θ(n)	Θ(m)
Boyer–Moore string-search algorithm	Θ(m + k)	best Ω(n/m), worst O(mn)	Θ(k)
Bitap algorithm (*shift-or*, *shift-and*, *Baeza–Yates–Gonnet*; fuzzy; agrep)	Θ(m + k)	O(mn)	
Two-way string-matching algorithm (glibc memmem/strstr)[3]	Θ(m)	O(n+m)	O(1)
BNDM (Backward Non-Deterministic DAWG Matching) (fuzzy + regex; nrgrep)[4]	O(m)	O(n)	
BOM (Backward Oracle Matching)[5]	O(m)	O(mn)	
FM-index	O(n)	O(m)	O(n)

1. ^ Asymptotic times are expressed using O, Ω, and Θ notation.

The **Boyer–Moore string-search algorithm** has been the standard benchmark for the practical string-search literature.[6]

Algorithms using a finite set of patterns

- Aho–Corasick string matching algorithm (extension of Knuth-Morris-Pratt)
- Commentz-Walter algorithm (extension of Boyer-Moore)
- Set-BOM (extension of Backward Oracle Matching)
- Rabin–Karp string search algorithm

Algorithms using an infinite number of patterns

Naturally, the patterns can not be enumerated finitely in this case. They are represented usually by a regular grammar or regular expression.

Other classification

Other classification approaches are possible. One of the most common uses preprocessing as main criteria.

Classes of string searching algorithms[7]

	Text not preprocessed	Text preprocessed
Patterns not preprocessed	Elementary algorithms	Index methods
Patterns preprocessed	Constructed search engines	Signature methods :[8]

Another one classifies the algorithms by their matching strategy:[9]

- Match the prefix first (Knuth-Morris-Pratt, Shift-And, Aho-Corasick)
- Match the suffix first (Boyer-Moore and variants, Commentz-Walter)
- Match the best factor first (BNDM, BOM, Set-BOM)
- Other strategy (Naive, Rabin-Karp)

Naïve string search

A simple and inefficient way to see where one string occurs inside another is to check each place it could be, one by one, to see if it's there. So first we see if there's a copy of the needle in the first character of the haystack; if not, we look to see if there's a copy of the needle starting at the second character of the haystack; if not, we look starting at the third character, and so forth. In the normal case, we only have to look at one or two characters for each wrong position to see that it is a wrong position, so in the average case, this takes $O(n + m)$ steps, where n is the length of the haystack and m is the length of the needle; but in the worst case, searching for a string like "aaaab" in a string like "aaaaaaaaab", it takes $O(nm)$

Finite-state-automaton-based search

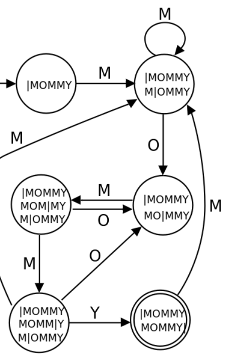
In this approach, we avoid backtracking by constructing a deterministic finite automaton (DFA) that recognizes stored search string. These are expensive to construct—they are usually created using the powerset construction—but are very quick to use. For example, the DFA shown to the right recognizes the word "MOMMY". This approach is frequently generalized in practice to search for arbitrary regular expressions.

Stubs

Knuth–Morris–Pratt computes a DFA that recognizes inputs with the string to search for as a suffix, Boyer–Moore starts searching from the end of the needle, so it can usually jump ahead a whole needle-length at each step. Baeza–Yates keeps track of whether the previous *j* characters were a prefix of the search string, and is therefore adaptable to fuzzy string searching. The bitap algorithm is an application of Baeza–Yates' approach.

Index methods

Faster search algorithms preprocess the text. After building a substring index, for example a suffix tree or suffix array, the occurrences of a pattern can be found quickly. As an example, a suffix tree can be built in $\Theta(n)$ time, and all z occurrences of a pattern can be found in $O(m)$ time under the assumption that the alphabet has a constant size and all inner nodes in the suffix tree know what leaves are underneath them. The latter can be accomplished by running a DFS algorithm from the root of the suffix tree.



Other variants

Some search methods, for instance trigram search, are intended to find a "closeness" score between the search string and the text rather than a "match/non-match". These are sometimes called "fuzzy" searches.

See also

- Sequence alignment
- Graph matching
- Pattern matching
- Compressed pattern matching
- Matching wildcards

References

1. Kurtz, Stefan; Phillippy, Adam; Delcher, Arthur L; Smoot, Michael; Shumway, Martin; Antonescu, Corina; Salzberg, Steven L (2004). "Versatile and open software for comparing large genomes" (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC395750). *Genome Biology*. **5** (2): R12. doi:10.1186/gb-2004-5-2-r12 (https://doi.org/10.1186%2Fgb-2004-5-2-r12). ISSN 1465-6906 (https://www.worldcat.org/issn/1465-6906). PMC 395750 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC395750). PMID 14759262 (https://pubmed.ncbi.nlm.nih.gov/14759262).
2. Khan, Zia; Bloom, Joshua S.; Kruglyak, Leonid; Singh, Mona (2009-07-01). "A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays" (https://academic.oup.com/bioinformatics/article/25/13/1609/196165). *Bioinformatics*. **25** (13): 1609–1616. doi:10.1093/bioinformatics/btp275 (https://doi.org/10.1093%2Fbioinformatics%2Fbtp275). PMC 2732316 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2732316). PMID 19389736 (https://pubmed.ncbi.nlm.nih.gov/19389736).
3. Crochemore, Maxime; Perrin, Dominique (1 July 1991). "Two-way string-matching" (http://monge.univ-mlv.fr/~mac/Articles-PDF/CP-1991-jacm.pdf) (PDF). *Journal of the ACM*. **38** (3): 650–674. doi:10.1145/116825.116845 (https://doi.org/10.1145%2F116825.116845).
4. Navarro, Gonzalo; Raffinot, Mathieu (1998). "A bit-parallel approach to suffix automata: Fast extended string matching" (https://users.dcc.uchile.cl/~gnavarro/ps/cpm98.pdf) (PDF). *Combinatorial Pattern Matching*. Lecture Notes in Computer Science. Springer Berlin Heidelberg. **1448**: 14–33. doi:10.1007/bfb0030778 (https://doi.org/10.1007%2Fbfb0030778). ISBN 978-3-540-64739-3.
5. Fan, H.; Yao, N.; Ma, H. (December 2009). "Fast Variants of the Backward-Oracle-Marching Algorithm" (https://pdfs.semanticscholar.org/8d81/94c293f8a81394ba545d09bd6ec711ad4c17.pdf) (PDF). *2009 Fourth International Conference on Internet Computing for Science and Engineering*: 56–59. doi:10.1109/ICICSE.2009.53 (https://doi.org/10.1109%2FICICSE.2009.53). ISBN 978-1-4244-6754-9.
6. Hume; Sunday (1991). "Fast String Searching" (https://semanticscholar.org/paper/d9912ea262986794e29e3f15e5f8930d42f2ced4). *Software: Practice and Experience*. **21** (11): 1221–1248. doi:10.1002/spe.4380211105 (https://doi.org/10.1002%2Fspe.4380211105).
7. Melichar, Borivoj, Jan Holub, and J. Polcar. Text Searching Algorithms. Volume I: Forward String Matching. Vol. 1. 2 vols., 2005. http://stringology.org/athens/TextSearchingAlgorithms/.
8. Riad Mokadem; Witold Litwin http://www.cse.scu.edu/~tschwarz/Papers/vldb07_final.pdf (2007), *Fast nGramBased String Search Over Data Encoded Using Algebraic Signatures*, 33rd International Conference on Very Large Data Bases (VLDB)
9. Gonzalo Navarro; Mathieu Raffinot (2008), *Flexible Pattern Matching Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*, ISBN 978-0-521-03993-2

- R. S. Boyer and J. S. Moore, *A fast string searching algorithm (http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf),* Carom. ACM 20, (10), 262–272(1977).
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press and McGraw-Hill, 2009. ISBN 0-262-03293-7. Chapter 32: String Matching, pp. 985–1013.

External links

- Huge list of pattern matching links (http://www.cs.ucr.edu/%7Estelo/pattern.html) Last updated: 12/27/2008 20:18:38
- Large (maintained) list of string-matching algorithms (https://www-igm.univ-mlv.fr/~lecroq/string/index.html)
- NIST list of string-matching algorithms (https://xlinux.nist.gov/dads/HTML/stringMatching.html)
- StringSearch – high-performance pattern matching algorithms in Java (http://johannburkard.de/software/stringsearch/) – Implementations of many String-Matching-Algorithms in Java (BNDM, Boyer-Moore-Horspool, Boyer-Moore-Horspool-Raita, Shift-Or)
- StringsAndChars (http://stringsandchars.amygdalum.net/) – Implementations of many String-Matching-Algorithms (for single and multiple patterns) in Java
- Exact String Matching Algorithms (http://www-igm.univ-mlv.fr/~lecroq/string/index.html) — Animation in Java, Detailed description and C implementation of many algorithms.
- (PDF) Improved Single and Multiple Approximate String Matching (http://www.cs.ucr.edu/~stelo/cpm/cpm04/35_Navarro.pdf)
- Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2647288/)