Aho–Corasick algorithm

In computer science, the **Aho–Corasick algorithm** is a string-searching algorithm invented by Alfred V. Aho and Margaret J. Corasick.[1] It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all strings simultaneously. The complexity of the algorithm is linear in the length of the strings plus the length of the searched text plus the number of output matches. Note that because all matches are found, there can be a quadratic number of matches if every substring matches (e.g. dictionary = `a`, `aa`, `aaa`, `aaaa` and input string is `aaaa`).

Informally, the algorithm constructs a finite-state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed string matches (e.g. a search for `cat` in a trie that does not contain `cat`, but contains `cart`, and thus would fail at the node prefixed by `ca`), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for `attribute` might be the best lateral transition). This allows the automaton to transition between string matches without the need for backtracking.

When the string dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear in the length of the input plus the number of matched entries.

The Aho–Corasick string-matching algorithm formed the basis of the original Unix command fgrep.

Contents

Example

In this example, we will consider a dictionary consisting of the following words: {a, ab, bab, bc, bca, c, caa}.

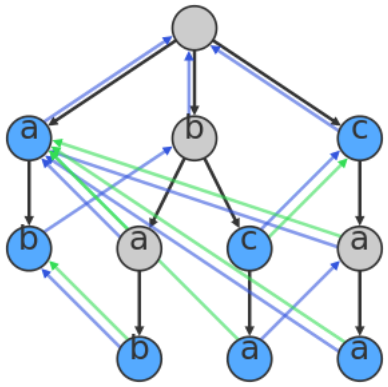
The graph below is the Aho–Corasick data structure constructed from the specified dictionary, with each row in the table representing a node in the trie, with the column path indicating the (unique) sequence of characters from the root to the node.

The data structure has one node for every prefix of every string in the dictionary. So if (bca) is in the dictionary, then there will be nodes for (bca), (bc), (b), and (). If a node is in the dictionary then it is a blue node. Otherwise it is a grey node.

There is a black directed "child" arc from each node to a node whose name is found by appending one character. So there is a black arc from (bc) to (bca).

There is a blue directed "suffix" arc from each node to the node that is the longest possible strict suffix of it in the graph. For example, for node (caa), its strict suffixes are (aa) and (a) and (). The longest of these that exists in the graph is (a). So there is a blue arc from (caa) to (a). The blue arcs can be computed in linear time by performing a breadth-first search starting from the root. The target for the blue arc of a visited node can be found by following its parent's blue arc to its longest suffix node and searching for a child of the suffix node whose character matches that of the visited node. If the character does not exist as a child, we can find the next longest suffix (following the blue arc again) and then search for the character. We can do this until we either find the character (as child of a node) or we reach the root (which will always be a suffix of every string).

There is a green "dictionary suffix" arc from each node to the next node in the dictionary that can be reached by following blue arcs. For example, there is a green arc from (bca) to (a) because (a) is the first node in the dictionary (i.e. a blue node) that is reached when following the blue arcs to (ca) and then on to (a). The green arcs can be computed in linear time by repeatedly traversing blue arcs until a blue node is found, and memoizing this information.



A visualization of the trie for the dictionary on the right. Suffix links are in blue; dictionary suffix links in green. Nodes corresponding to dictionary entries are highlighted in blue.

Dictionary {a, ab, bab, bc, bca, c, caa}

Path	In dictionary	Suffix link	Dict suffix link
()	–		
(a)	+	()	
(ab)	+	(b)	
(b)	–	()	
(ba)	–	(a)	(a)
(bab)	+	(ab)	(ab)
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	–	(a)	(a)
(caa)	+	(a)	(a)

At each step, the current node is extended by finding its child, and if that doesn't exist, finding its suffix's child, and if that doesn't work, finding its suffix's suffix's child, and so on, finally ending in the root node if nothing's seen before.

When the algorithm reaches a node, it outputs all the dictionary entries that end at the current character position in the input text. This is done by printing every node reached by following the dictionary suffix links, starting from that node, and continuing until it reaches a node with no dictionary suffix link. In addition, the node itself is printed, if it is a dictionary entry.

Execution on input string `abccab` yields the following steps:

Analysis of input string `abccab`

Node	Remaining string	Output:end position	Transition	Output
()	abccab		start at root	
(a)	bccab	a:1	() to child (a)	Current node
(ab)	ccab	ab:2	(a) to child (ab)	Current node
(bc)	cab	bc:3, c:3	(ab) to suffix (b) to child (bc)	Current Node, Dict suffix node
(c)	ab	c:4	(bc) to suffix (c) to suffix () to child (c)	Current node
(ca)	b	a:5	(c) to child (ca)	Dict suffix node
(ab)		ab:6	(ca) to suffix (a) to child (ab)	Current node

Dynamic search list

The original Aho-Corasick algorithm assumes that the set of search strings is fixed. It does not directly apply to applications in which new search strings are added during application of the algorithm. An example is an interactive indexing program, in which the user goes through the text and highlights new words or phrases to index as he or she sees them. Bertrand Meyer introduced an incremental version of the algorithm in which the search string set can be incrementally extended during the search, retaining the algorithmic complexity of the original.[2]

See also

- Commentz-Walter algorithm

References

1. Aho, Alfred V.; Corasick, Margaret J. (June 1975). "Efficient string matching: An aid to bibliographic search". *Communications of the ACM*. **18** (6): 333–340. doi:10.1145/360825.360855 (https://doi.org/10.1145%2F360825.360855). MR 0371172 (https://www.ams.org/mathscinet-getitem?mr=0371172).
2. Meyer, Bertrand (1985). "Incremental string matching" (http://se.ethz.ch/~meyer/publications/string/string_matching.pdf) (PDF). *Information Processing Letters*. **21**: 219–227. doi:10.1016/0020-0190(85)90088-2 (https://doi.org/10.1016%2F0020-0190%2885%2990088-2).

External links

- Aho-Corasick (https://xlinux.nist.gov/dads/HTML/ahoCorasick.html) in NIST's Dictionary of Algorithms and Data Structures (2019-07-15)
- Aho-Corasick visualisation (https://wiomoc.de/aho-corasick-viz/)
- Aho-Corasick Implementation (C++) (https://github.com/cjgdev/aho_corasick) on GitHub
- A Golang implementation of the Aho-Corasick string matching algorithm (https://github.com/cloudflare/ahocorasick) on GitHub
- A fast implementation of Aho-Corasick in Rust (https://github.com/BurntSushi/aho-corasick) on GitHub