Aho-Corasick algorithm - Algorithms for Competitive Programming

Source: https://cp-algorithms.com/string/aho_corasick.html

Last update: April 18, 2025 Translated From: e-maxx.ru Aho-Corasick algorithm ¶ The Aho-Corasick algorithm allows us to quickly search for multiple patterns in a text. The set of pattern strings is also called a dictionary . We will denote the total length of its constituent strings by m and the size of the alphabet by k . The algorithm constructs a finite state automaton based on a trie in $O(m k)$ time and then uses it to process the text. The algorithm was proposed by Alfred Aho and Margaret Corasick in 1975. Construction of the trie ¶ A trie based on words "Java", "Rad", "Rand", "Rau", "Raum" and "Rose". The image by [nd](https://de.wikipedia.org/wiki/Benutzer:Nd) is distributed under CC BY-SA 3.0 license. Formally, a trie is a rooted tree, where each edge of the tree is labeled with some letter and outgoing edges of a vertex have distinct labels. We will identify each vertex in the trie with the string formed by the labels on the path from the root to that vertex. Each vertex will also have a flag output which will be set if the vertex corresponds to a pattern in the dictionary. Accordingly, a trie for a set of strings is a trie such that each output vertex corresponds to one string from the set, and conversely, each string of the set corresponds to one output vertex. We now describe how to construct a trie for a given set of strings in linear time with respect to their total length. We introduce a structure for the vertices of the tree: const int K = 26 ; struct Vertex { int next [K]; bool output = false ; Vertex () { fill (begin (next), end (next), -1); } }; vector < Vertex > trie (1); Here, we store the trie as an array of Vertex . Each Vertex contains the flag output and the edges in the form of an array $\text{next}[]$, where $\text{next}[i]$ is the index of the vertex that we reach by following the character i , or -1 if there is no such edge. Initially, the trie consists of only one vertex - the root - with the index 0 . Now we implement a function that will add a string s to the trie. The implementation is simple: we start at the root node, and as long as there are edges corresponding to the characters of s we follow them. If there is no edge for one character, we generate a new vertex and connect it with an edge. At the end of the process we mark the last vertex with the flag output . void add_string (string const & s) { int v = 0 ; for (char ch : s) { int c = ch - 'a' ; if (trie [v]. next [c] == -1) { trie [v]. next [c] = trie . size (); trie . emplace_back (); } v = trie [v]. next [c]; } trie [v]. output = true ; } This implementation obviously runs in linear time, and since every vertex stores k links, it will use $O(m k)$ memory. It is possible to decrease the memory consumption to $O(m)$ by using a map instead of an array in each vertex. However, this will increase the time complexity to $O(m \log k)$. Construction of an automaton ¶ Suppose we have built a trie for the given set of strings. Now let's look at it from a different side. If we look at any vertex, the string that corresponds to it is a prefix of one or more strings in the set, thus each vertex of the trie can be interpreted as a position in one or more strings from the set. In fact, the trie vertices can be interpreted as states in a finite deterministic automaton . From any state we can transition - using some input letter - to other states, i.e., to another position in the set of strings. For example, if there is only one string abc in the dictionary, and we are standing at vertex ab , then using the letter c we can go to the vertex abc . Thus we can understand the edges of the trie as transitions in an automaton according to the corresponding letter. However, in an automaton we need to have transitions for each combination of a state and a letter. If we try to perform a transition using a letter, and there is no corresponding edge in the trie, then we nevertheless must go into some state. More precisely, suppose we are in a state corresponding to a string t , and we want to transition to a different state using the character c . If there is an edge labeled with this letter c , then we can simply go over this edge, and get the vertex corresponding to $t + c$. If there is no such edge, since we want to maintain the invariant that the current state is the longest partial match in the processed string, we must find the longest string in the trie that's a proper suffix of the string t , and try to perform a transition from there. For example, let the trie be constructed by the strings ab and bc , and we are currently at the vertex corresponding to ab , which is also an output vertex. To transition with the letter c , we are forced to go to the state corresponding to the string b , and from there follow the edge with the letter c . An Aho-Corasick automaton based on words "a", "ab", "bc", "bca", "c" and "caa". Blue arrows are suffix links, green arrows are terminal links. A suffix link for a vertex p is an edge that points to the longest proper suffix of the string corresponding to the vertex p . The only special case is the root of the trie, whose suffix link will point to itself. Now we can reformulate the statement about the transitions in the automaton like this: while there is no transition from the current vertex of the trie using the current letter (or until we reach the root), we follow the suffix link. Thus we reduced the problem of constructing an automaton to the problem of finding suffix links for all vertices of the trie. However, we will build these suffix links, oddly enough, using the transitions constructed in the automaton. The suffix links of the root vertex and all its immediate children point to the root vertex. For any vertex v deeper in the tree, we can calculate the suffix link as follows: if p is the ancestor of v with c being the letter labeling the edge from p to v , go to p , then follow its suffix link, and perform the transition with the letter c from there. Thus, the problem of finding the transitions has been reduced to the problem of finding suffix links, and the problem of finding suffix links has been reduced to the problem of finding a suffix link and a transition, except for vertices closer to the root. So we have a recursive dependence that we can resolve in linear time. Let's move to the implementation. Note that we now will store the ancestor p and the character pch of the edge from p to v for each vertex v . Also, at each vertex we will store the suffix link link (or -1 if it hasn't been calculated yet), and in the array $\text{go}[k]$ the transitions in the machine for each symbol (again -1 if it hasn't been calculated yet). const int K = 26 ; struct Vertex { int next [K]; bool output = false ; int p = -1 ; char pch ; int link = -1 ; int go [K]; Vertex (int p = -1 , char ch = '$') : p (p), pch (ch) { fill (begin (next), end (next), -1); fill (begin (go), end (go), -1); } }; vector < Vertex > t (1); void add_string (string const & s) { int v = 0 ; for (char ch : s) { int c = ch - 'a' ; if (t [v]. next [c] == -1) { t [v]. next [c] = t . size (); t . emplace_back (v , ch); } v = t [v]. next [c]; } t [v]. output = true ; } int go (int v , char ch); int get_link (int v) { if (t [v]. link == -1) { if (v == 0 || t [v]. p == 0) t [v]. link = 0 ; else t [v]. link = go (get_link (t [v]. p), t [v]. pch); } return t [v]. link ; } int go (int v , char ch) { int c = ch - 'a' ; if (t [v]. go [c] == -1) { if (t [v]. next [c] != -1) t [v]. go [c] = t [v]. next [c]; else t [v]. go [c] = v == 0 ? 0 : go (get_link (v), ch); } return t [v]. go [c]; } It is easy to see that thanks to memoization of the suffix links and transitions, the total time for finding all suffix links and transitions will be linear. For an illustration of the concept refer to slide number 103 of the Stanford slides . BFS-based construction ¶ Instead of computing transitions and suffix links with recursive calls to go and get_link , it is possible to compute them bottom-up starting from the root. (In fact, when the dictionary consists of only one string, we obtain the familiar Knuth-Morris-Pratt algorithm.) This approach will have some advantages over the one described above as, instead of the total length m , its running time depends only on the number of vertices n in the trie. Moreover, it is possible to adapt it for large alphabets using a persistent array data structure, thus making the construction time $O(n \log k)$ instead of $O(mk)$, which is a significant improvement granted that m may go up to n^2 . We can reason inductively using the fact that BFS from the root traverses vertices in order of increasing length. We may assume that when we're in a vertex v , its suffix link $u = link[v]$ is already successfully computed, and for all vertices with shorter length transitions from them are also fully computed. Assume that at the moment we stand in a vertex v and consider a character c . We essentially have two cases: $go[v][c] = -1$. In this case, we may assign $go[v][c] = go[u][c]$, which is already known by the induction hypothesis; $go[v][c] = w \neq -1$. In this case, we may assign $link[w] = go[u][c]$. In this way, we spend $O(1)$ time per each pair of a vertex and a character, making the running time $O(nk)$. The major overhead here is that we copy a lot of transitions from u in the first case, while the transitions of the second case form the trie and sum up to n over all vertices. To avoid the copying of $go[u][c]$, we may use a persistent array data structure, using which we initially copy $go[u]$ into $go[v]$ and then only update values for characters in which the transition would differ. This leads to the $O(n \log k)$ algorithm. Applications ¶ Find all strings from a given set in a text ¶ We are given a set of strings and a text. We have to print all occurrences of all strings from the set in the given text in $O(\text{len} + \text{ans})$, where len is the length of the text and ans is the size of the answer. We construct an automaton for this set of strings. We will now process the text letter by letter using the automaton, starting at the root of the trie. If we are at any time at state v , and the next letter is c , then we transition to the next state with $\text{go}(v, c)$, thereby either increasing the length of the current match substring by 1 , or decreasing it by following a suffix link. How can we find out for a state v , if there are any matches with strings for the set? First, it is clear that if we stand on a output vertex, then the string corresponding to the vertex ends at this position in the text. However this is by no means the only possible case of achieving a match: if we can reach one or more output vertices by moving along the suffix links, then there will be also a match corresponding to each found output vertex. A simple example demonstrating this situation can be created using the set of strings $\{dabce, abc, bc\}$ and the text $dabc$. Thus if we store in each output vertex the index of the string corresponding to it (or the list of indices if duplicate strings appear in the set), then we can find in $O(n)$ time the indices of all strings which match the current state, by simply following the suffix links from the current vertex to the root. This is not the most efficient solution, since this results in $O(n ~ \text{len})$ complexity overall. However, this can be optimized by computing and storing the nearest output vertex that is reachable using suffix links (this is sometimes called the exit link). This value we can compute lazily in linear time. Thus for each vertex we can advance in $O(1)$ time to the next marked vertex in the suffix link path, i.e. to the next match. Thus for each match we spend $O(1)$ time, and therefore we reach the complexity $O(\text{len} + \text{ans})$. If you only want to count the occurrences and not find the indices themselves, you can calculate the number of marked vertices in the suffix link path for each vertex v . This can be calculated in $O(n)$ time in total. Thus we can sum up all matches in $O(\text{len})$. Finding the lexicographically smallest string of a given length that doesn't match any given strings ¶ A set of strings and a length L is given. We have to find a string of length L , which does not contain any of the strings, and derive the lexicographically smallest of such strings. We can construct the automaton for the set of strings. Recall that output vertices are the states where we have a match with a string from the set. Since in this task we have to avoid matches, we are not allowed to enter such states. On the other hand we can enter all other vertices. Thus we delete all "bad" vertices from the machine, and in the remaining graph of the automaton we find the lexicographically smallest path of length L . This task can be solved in $O(L)$ for example by depth first search . Finding the shortest string containing all given strings ¶ Here we use the same ideas. For each vertex we store a mask that denotes the strings which match at this state. Then the problem can be reformulated as follows: initially being in the state $(v = \text{root},~ \text{mask} = 0)$, we want to reach the state $(v,~ \text{mask} = 2^n - 1)$, where n is the number of strings in the set. When we transition from one state to another using a letter, we update the mask accordingly. By running a breadth first search we can find a path to the state $(v,~ \text{mask} = 2^n - 1)$ with the smallest length. Finding the lexicographically smallest string of length L containing k strings ¶ As in the previous problem, we calculate for each vertex the number of matches that correspond to it (that is the number of marked vertices reachable using suffix links). We reformulate the problem: the current state is determined by a triple of numbers $(v,~ \text{len},~ \text{cnt})$, and we want to reach from the state $(\text{root},~ 0,~ 0)$ the state $(v,~ L,~ k)$, where v can be any vertex. Thus we can find such a path using depth first search (and if the search looks at the edges in their natural order, then the found path will automatically be the lexicographically smallest). Problems ¶ UVA #11590 - Prefix Lookup UVA #11171 - SMS UVA #10679 - I Love Strings!! Codeforces - x-prime Substrings Codeforces - Frequency of String CodeChef - TWOSTRS References ¶ Stanford's CS166 - Aho-Corasick Automata (Condensed) Contributors: jakobkogler (61.07%) wikku (23.93%) adamant-pwn (11.78%) madhur4127 (1.79%) iamlockon (0.36%) gafeol (0.36%) amanCoder110599 (0.36%) GaurangTandon (0.36%)