Aho-Corasick Algorithm for Pattern Searching - GeeksforGeeks

Given an input text and an array of k words, arr[], find all occurrences of all words in the input text. Let **n** be the length of text and **m** be the total number characters in all words, i.e. m = length(arr[0]) + length(arr[1]) + .. + O(n + length(arr[k-1]). Here **k** is total numbers of input words.

Example:

```
Input: text = "ahishers"
       arr[] = {"he", "she", "hers", "his"}

Output:
   Word his appears from 1 to 3
   Word he appears from 4 to 5
   Word she appears from 3 to 5
   Word hers appears from 4 to 7
```

If we use a linear time searching algorithm like **KMP**, then we need to one by one search all words in text[]. This gives us total time complexity as O(n + length(word[0]) + O(n + length(word[1]) + O(n + length(word[2]) + … O(n + length(word[k-1]). This time complexity can be written as ***O(n*k + m)***. **Aho-Corasick Algorithm** finds all words in ***O(n + m + z)*** time where **z** is total number of occurrences of words in text. The Aho–Corasick string matching algorithm formed the basis of the original Unix command fgrep.

1. **Prepocessing :** Build an automaton of all words in arr[] The automaton has mainly three functions:

   ```
   Go To :   This function simply follows edges
             of Trie of all words in arr[]. It is
             represented as 2D array g[][] where
             we store next state for current state
             and character.

   Failure : This function stores all edges that are
             followed when current character doesn't
             have edge in Trie.  It is represented as
             1D array f[] where we store next state for
             current state.

   Output :  Stores indexes of all words that end at
             current state. It is represented as 1D
             array o[] where we store indexes
   ```

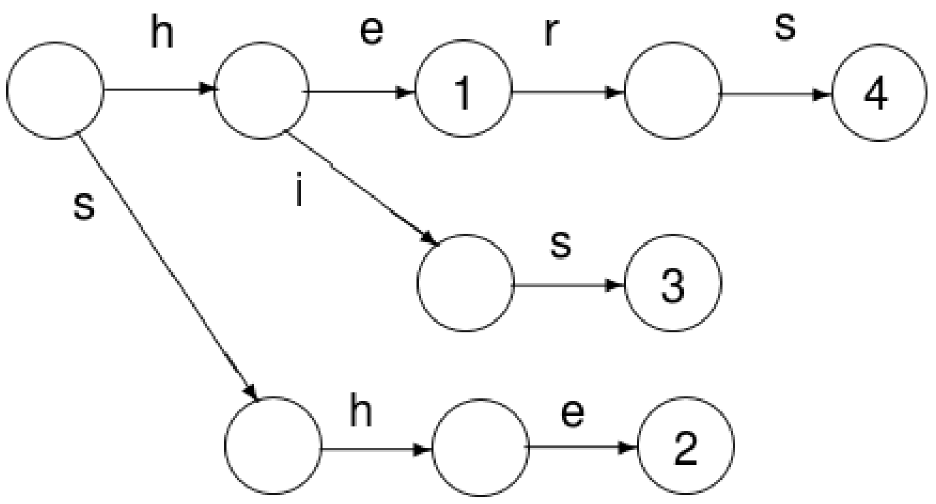
```
            of all matching words as a bitmap for
            current state.
```

2. **Matching :** Traverse the given text over built automaton to find all matching words.

Preprocessing:

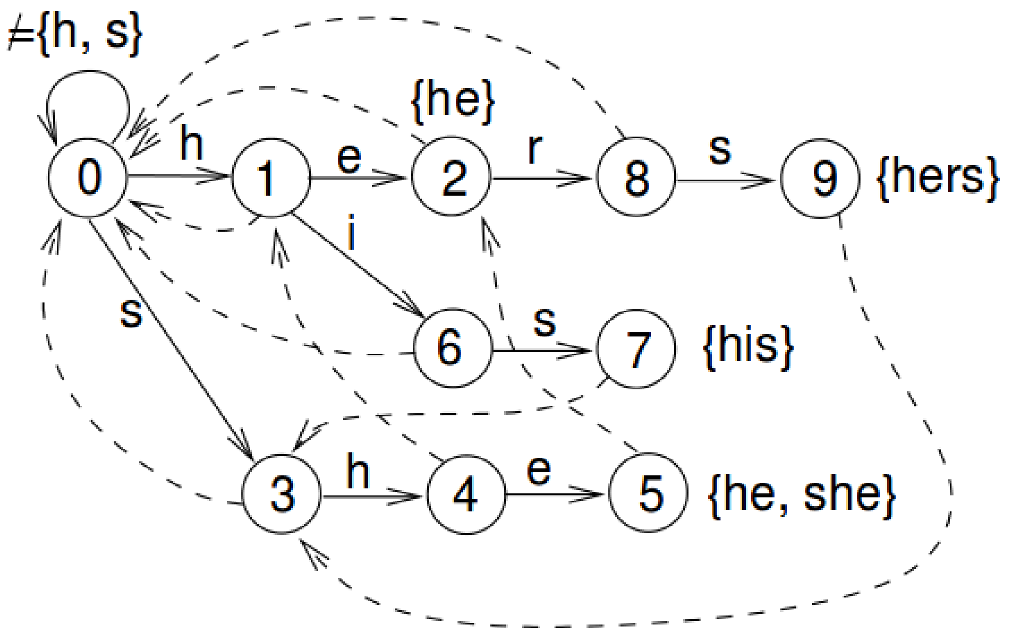
1. We first Build a Trie (or Keyword Tree) of all words.

Trie for arr[] = {he, she, his, hers}



   This part fills entries in goto g[][] and output o[].

2. Next we extend Trie into an automaton to support linear time matching.



Dashed arrows are failed transactions.

Normal arrows are Goto (or Trie) transactions

   This part fills entries in failure f[] and output o[].

Go to :
We build Trie. And for all characters which don’t have an edge at root, we add an edge back to root.

Failure :
For a state s, we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie.

Output :
For a state s, indexes of all words ending at s are stored. These indexes are stored as bitwise map (by doing bitwise OR of values). This is also computing using Breadth First Traversal with Failure.

Below is C++ implementation of Aho-Corasick Algorithm

```
// C++ program for implementation of Aho Corasick algorithm
// for string matching
using namespace std;
#include <bits/stdc++.h>

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 500;

// Maximum number of characters in input alphabet
const int MAXC = 26;

// OUTPUT FUNCTION IS IMPLEMENTED USING out[]
// Bit i in this mask is one if the word with index i
// appears when the machine enters this state.
int out[MAXS];

// FAILURE FUNCTION IS IMPLEMENTED USING f[]
int f[MAXS];

// GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
int g[MAXS][MAXC];

// Builds the string matching machine.
// arr -   array of words. The index of each keyword is important:
//         "out[state] & (1 << i)" is > 0 if we just found word[i]
//         in the text.
// Returns the number of states that the built machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(string arr[], int k)
{
    // Initialize all values in output function as 0.
    memset(out, 0, sizeof out);

    // Initialize all values in goto function as -1.
    memset(g, -1, sizeof g);

    // Initially, we just have the 0 state
    int states = 1;

    // Construct values for goto function, i.e., fill g[][]
    // This is same as building a Trie for arr[]
    for (int i = 0; i < k; ++i)
    {
        const string &word = arr[i];
        int currentState = 0;

        // Insert all characters of current word in arr[]
        for (int j = 0; j < word.size(); ++j)
        {
            int ch = word[j] - 'a';

            // Allocate a new node (create a new state) if a
            // node for ch doesn't exist.
            if (g[currentState][ch] == -1)
                g[currentState][ch] = states++;

            currentState = g[currentState][ch];
        }

        // Add current word in output function
        out[currentState] |= (1 << i);
    }

    // For all characters which don't have an edge from
    // root (or state 0) in Trie, add a goto edge to state
    // 0 itself
    for (int ch = 0; ch < MAXC; ++ch)
        if (g[0][ch] == -1)
            g[0][ch] = 0;

    // Now, let's build the failure function

    // Initialize values in fail function
    memset(f, -1, sizeof f);
    // Failure function is computed in breadth first order
    // using a queue
    queue<int> q;

     // Iterate over every possible input
    for (int ch = 0; ch < MAXC; ++ch)
    {
        // All nodes of depth 1 have failure function value
        // as 0. For example, in above diagram we move to 0
        // from states 1 and 3.
        if (g[0][ch] != 0)
        {
            f[g[0][ch]] = 0;
            q.push(g[0][ch]);
        }
    }

    // Now queue has states 1 and 3
    while (q.size())
    {
        // Remove the front state from queue
        int state = q.front();
        q.pop();

        // For the removed state, find failure function for
        // all those characters for which goto function is
        // not defined.
        for (int ch = 0; ch <= MAXC; ++ch)
        {
            // If goto function is defined for character 'ch'
            // and 'state'
            if (g[state][ch] != -1)
            {
                // Find failure state of removed state
                int failure = f[state];

                // Find the deepest node labeled by proper
                // suffix of string from root to current
                // state.
                while (g[failure][ch] == -1)
                      failure = f[failure];

                failure = g[failure][ch];
                f[g[state][ch]] = failure;

                // Merge output values
                out[g[state][ch]] |= out[failure];

                // Insert the next level node (of Trie) in Queue
                q.push(g[state][ch]);
            }
        }
    }

    return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.
// currentState - The current state of the machine. Must be between
//                0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine.
int findNextState(int currentState, char nextInput)
{
    int answer = currentState;
    int ch = nextInput - 'a';

    // If goto is not defined, use failure function
    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

// This function finds all occurrences of all array words
// in text.
void searchWords(string arr[], int k, string text)
{
    // Preprocess patterns.
    // Build machine with goto, failure and output functions
    buildMatchingMachine(arr, k);

    // Initialize current state
    int currentState = 0;

    // Traverse the text through the nuilt machine to find
    // all occurrences of words in arr[]
    for (int i = 0; i < text.size(); ++i)
    {
        currentState = findNextState(currentState, text[i]);

        // If match not found, move to next state
        if (out[currentState] == 0)
             continue;

        // Match found, print all matching words of arr[]
        // using output function.
        for (int j = 0; j < k; ++j)
        {
            if (out[currentState] & (1 << j))
            {
                cout << "Word " << arr[j] << " appears from "
                     << i - arr[j].size() + 1 << " to " << i << endl;
            }
        }
    }
}

// Driver program to test above
int main()
{
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int k = sizeof(arr)/sizeof(arr[0]);

    searchWords(arr, k, text);

    return 0;
}
```

Output:

```
Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5
Word hers appears from 4 to 7
```

Source:
http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf

This article is contributed by **Ayush Govil**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above