Aho–Corasick

Neil Thistlethwaite

4th June 2017

1 Introduction

Aho–Corasick is a string searching algorithm that simultaneously finds all strings within an input text, and runs in O(n + m + z) time, where n is the length of the input text, m is the total length of strings to search for, and z is the total number of matches, which makes it faster than algorithms just as KMP, which would require you to run it once on each input string.

Aho–Corasick works by constructing a Finite State Machine from a trie of the input strings, adding additional connections for the fail-case, which provides the linear computational efficiency.

2 Contsruction of the FSM

First, we make a trie (similar to a tree) of the input strings, which simplifies the input into one traversable tree. Here, connections between nodes represent successful matches.

For example, the trie on the right represents the input strings “aba”, “abc”, “ad”, and “babac”.

Next, we need to add the correct state transitions for when a string mismatch occurs – if, for example, we were at b-a-b in the trie and encountered a mismatch, we would want to jump over to the a-b state, avoiding having to backtrack in the text.

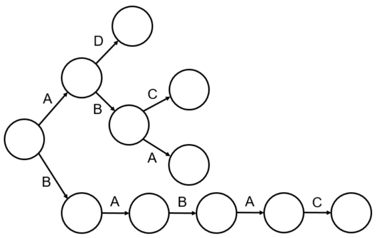

Figure 1: Sample Trie

To do this, we compute ”suffix links”: every state should have a fail connection that takes it to a new state of the longest possible suffix of the current state. Specifically, we will conduct a Breadth-First Search on the trie, and at every node we visit do the following steps:

- Step back up to your parent
- Follow your parents suffix links
- Search for value of original state in suffix links' children
- Add a new suffix link

By doing this in a BFS, it's guaranteed that when a state at depth d is visited, all suffix links of depth d-1 will have already been generated. Essentially, the suffix links are built up from the root node.

It is left as an exercise to the reader to perform this on the provided trie.[1]

3 Searching

Once the FSM is constructed, searching is simple. We start at the root state, and then repeat the following steps until the entire search text has been processed.

- Read in the next character c
- If the current state has a child state that matches character c, go there
- Otherwise, follow the current state's suffix
    - Repeat until find node with child c, or at root node

Whenever we reach the end of a search string, we have found a match and can record it. However, because it is possible that we've also matched another string that is a suffix of the current match, we must check all suffix links and see if they complete search strings as well, and record those matches too.

[1] Just kidding, but diagrams are hard to make so I'll do this on the board