A Twitter Search Algorithm using Aho-Corasic Algorithm

Chenbo Zhang
Department of ECE
Boston University
zhangcb@bu.edu

Haoxiang Sun
Department of ECE
Boston University
shx95@bu.edu

Yiwei Zhang
Department of ECE
Boston University
zhang598@bu.edu

Shizhe Yang
Department of ECE
Boston University
szyang78@bu.edu

Abstract

Twitter search problem is basically a word matching problem. However, ordinary word matching algorithm like Knuth–Morris–Pratt (KMP) or Boyer-Moore algorithms has to go over data-set multiple times when scanning multiple patterns. The Aho-Corasic method provide a way to match all patterns with only one time scan. In this report, I will introduce Aho-Corasic algorithm from theory to practice.

1 Introduction

This section includes the basic description of Aho-Corasic algorithm, the comparison with KMP algorithm, and the analysis of their time complexity.

1.1 Description

Aho Corasic algorithm is based on Trie tree and is an extension of KMP pattern matching algorithm. So here we can from two aspects as a starting point, a detailed description of Aho Corasic algorithm formed. The core point of Trie tree is state transition, and the core point of KMP pattern matching is to reduce duplicate matching. Let's start with the Trie tree. The node is the character itself, Each node is both the character itself and the state of the node. For example, there are currently two dictionary strings: T1: "abcde" and T2: "abdef", when we walk through "abcd" and stop at the 'd' character. At this point, we can assume that we are currently on the string T1. Because the current node can represent its state. In T1 and T2, the states of the two 'd' nodes are different. The state transition of Trie tree can be understood as: when traversing to node D, we dynamically determine the next state of node D, namely node E.

1.2 Comparison with KMP Algorithm

Let's talk about KMP pattern matching. In the KMP pattern matching process, we used a next function. The next function allows us to slide the pattern string forward n characters when a match fails, saving n comparisons. Using the Trie tree matching, we would still iterate over D, just like naive pattern matching. The Trie tree reduces only the overlap in the Trie tree, so the time complexity is quite high. What about the KMP algorithm? The KMP algorithm generates the next function for the pattern string. In the case of multiple patterns, we need to generate many next functions and match each pattern. This is clearly not ideal. Based on the above two points, our Aho Corasic algorithm was born.

1.3 Time complexity Comparison

Compared to the KMP algorithm, Given an input text and an array of k words, array[], 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(array[0]) + length(array[1]) + . . . + length(array[k-1]). Here k is total numbers of input words. 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 Corasic Algorithm finds all words in O(n + m + z) time where z is total number of occurrences of words in text.

2 Data Structure

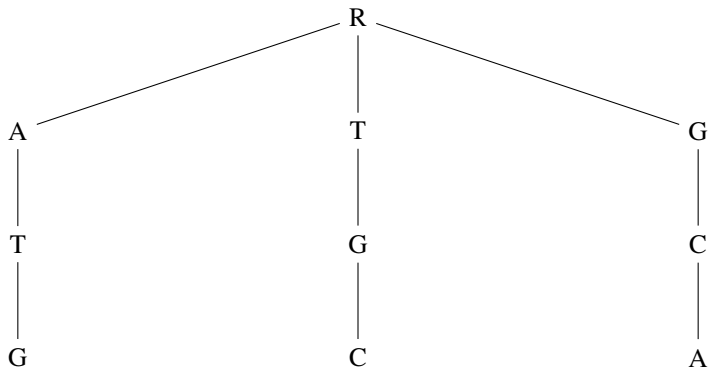
In this chapter, I would introduce the data structure we use to implement Aho-Corasick algorithm.

2.1 The Aho-Trie

Aho Corasick algorithm basically build a tree, where every node has a link to parent, a map to multiple children, a single character value, and a "redirect link", we will explain it later.

The core Idea is that, when matching failed at some character, instead of going over, it use "redirect link" to get to the longest possible substring (count from the last character of the pattern) , go that substring branch and continue.

For example, suppose we have pattern ATG, TGC, and GCA, we can construct a tree like this:



That R stand for Root. If we failed at ATX, that is, T(3,1) has no child X, we will follow its redirect link, which in this situation, will link to T(2,2). Also, if we fail at TGX, we go to G(1,3).

If any attempt gets to the bottom node of the tree then a pattern is matched.

So, by going over this tree, we can find matched pattern by only scanning the data for one time.

2.2 Structure and Function

The class CAhoTree in is defined in AhoCorasickAlg.hpp, with properties and functions below:

- Class CAhoTree
  - Private Structure SNode
    - nId
      int, the id of the node;
    - nValue
      char, the value for matching;
    - nHeight
      int, the height of node on the tree;
    - pParent
      pointer, point to parent node;
    - pRedirect
      pointer, point to the redirect node, if the matching fails, go this way for next match;

    - mapChild
      map<char, SNode*> link parent to their children;
    - bEndNode
      bool, decide if the node is a end of a matching pattern;
  - Public Functions
    - CAhoTree
      Construct method
    - CAhoTree
      Destroy method
    - AddPattern
      From root, walk the tree by the character of the pattern, if walk into a dead end, build a branch using what is left in input string, the last character of the string is matched as bEndNode;
    - Redirecting
      Go through every exist node with same key (every new node will be pushed into a local map, key is char nValue , value is the set of nodes with char nValue);
      Try connect them, Call function **ConnectRedirectNode**;
    - MatchPattern
      Return if there exist any match, stops at first match, faster than searching method;
    - SearchPattern
      Search all matches, find matched patterns, call **GenerateOutput** to return a output string, insert it in a map, return this map, key is matched pattern, value is this pattern's matched count;
    - SearchCount
      Return the sum of matched times of all pattern;
  - Private Functions
    - NewNode
      Called by function **AddPattern**;
      Create a new node pointer with initial value and push it into node map and node vector;
    - ConnectRedirectNode
      Called by function **Redirecting**;
      evaluate this 2 nodes and connect them if passed evaluation;
    - CaseConvert
      In case-not-sensitive situation, transfer upper case to lower case;
    - GenerateOutput
      Send a endnode in, go from it to the root and get the whole matched word, return it as a string;

Much detailed version can be viewed from the code comment.

3 Conclusion

Although Aho-Corasick algorithm is theoretically faster than multi-time KMP algorithm, the large overhead building Aho-Trie makes it practically slower than actual KMP Algorithm when the dataset is not very large and there is not too many patterns. But Aho-Corasick Algorithm showed a excellent behavior when scanning the extremely-large data-set provided with a large amount of pattern.

4 More to Go

If the time is more sufficient, we could have more time finishing Boyer-Moore algorithm and we can compare the time difference, and even generate data-set to plot some data-scale/ pattern number/ time consume graph. Also we could finish our UI as promised, to make this program more user friendly.

5 Divide of the Work

Name	Job	result
Chenbo Zhang	Data structure design and Aho-Tree implementation	AhoCorasickAlg.hpp
Haoxiang Sun	Main function design and IO design	searchstr.cpp
Yiwei Zhang	UI design and Integration, math part	GUI ERROR Unsolved
Shizhe Yang	Information collection and research, project management	Divided work load

Note: Though Yiwei failed building UI for this project, he contributed a great time and effort, he did his best fighting with GUI bugs that the whole internet can not solve, he is the hero of our team.