Finite State Machine Based String Matching Algorithm

Feng Zonghao[1]

Abstract

Finite state machine is a useful abstract model of computation. String matching is a classic problem with a variety of efficient algorithms. This essay gives a brief view of finite state machine based string matching algorithms, and describes one representative example: Aho-Corasick automaton in detail, by showing its history, main idea, construction method, implementation and applications.

1. Introduction

A **finite state machine** or **finite state automaton** is a mathematical model of computation. It usually consists of a finite number of states that describes the status of the machine at a given instant in time, and a transition function that offers the conditions for a change from one state to another. A large amount of problems can be modeled in finite state machines, which help people a lot in finding efficient algorithms to solve them.

String matching algorithms are used to find a place where one or several strings are found within a larger string or text. There are several mainstream string matching algorithms:

- Rabin-Karp algorithm
- finite-state machine based algorithms
- Knuth-Morris-Pratt algorithm (KMP)
- Boyer–Moore string search algorithm

Consider if we are to match two strings with length m and n. The most simple way to finish this task is by applying the brute force approach. The main idea is to enumerate every position in the longer string, and check if it's a possible occurrence position of the shorter string by compare characters one by one. In worst case it has to do $m \times n$ comparisons.

To accelerate the naive solution, we could try to avoid backtracking by constructing a deterministic finite automaton (DFA) that recognizes stored search string, which leads to the idea of **state machine based string matching algorithms**. The construction of the finite state machine may takes time, but ones the machine was built, it works much faster than the naive solution. What's more impressive is that it can be used repeatly.

The feature of automaton based algorithms is that it only depends on a look-up table to run, so that the problem can be dealt with just simply check the table over and over again. With slow construction time but fast verification time, the taste of automatons is kind of like one-way functions.

[1]Computer Science Class 2013, Taishan College, Shandong University. Student number: 201300130013

2. The Aho-Corasick Automaton

Aho-Corasick Algorithm, also known as Aho-Corasick automaton (AC automaton), was invented by Alfred V. Aho and Margaret J. Corasick from Bell Laboratories in 1975. [1] It works efficently on locating elements of a finite set of strings within an input text. It matches all patterns simultaneously, so the complexity of the algorithm is linear in the length of the patterns plus the length of the searched text.

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 pattern matches, to other branches of the trie that share a common prefix. This allows the automaton to transition between pattern matches without the need for backtracking. When the pattern dictionary is known in advance, the construction of the automaton can be performed once off-line and the compiled automaton stored for later use.

3. Idea

How does an AC automaton work? Let's analyse it by solving the following problem: given a set of strings $P = \{P_1, \ldots, P_k\}$ (usually called patterns or keywords), and a target string $T[1 \ldots m]$, count how many P_i occurs in T, that is P_i is a substring of T. This problem is also known as the **exact string set matching problem**.

Let $n = \sum_{i=1}^{k} |P_i|$. Exact string set matching can be solved in time $O(|P_1| + m + \ldots + |P_k| + m) = O(n + km)$ by applying any linear-time exact matching algorithm (such as Knuth-Morris-Pratt algorithm) for k times. But by using an AC automaton, we could gain a much more optimal time complexity of $O(n + m)$.

The first step of building an AC automaton is to constuct a keyword tree, or formally called a **trie**. A trie is a rooted tree K such that each edge of K is labeled by a a character. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. A trie is an efficient implementation of a dictionary of strings. The construction of a trie takes $O(|P_1| + \ldots + |P_k|) = O(n)$ time. A schematic diagram of a trie of $P = \{he, she, his, hers\}$ is shown in fig. 1.

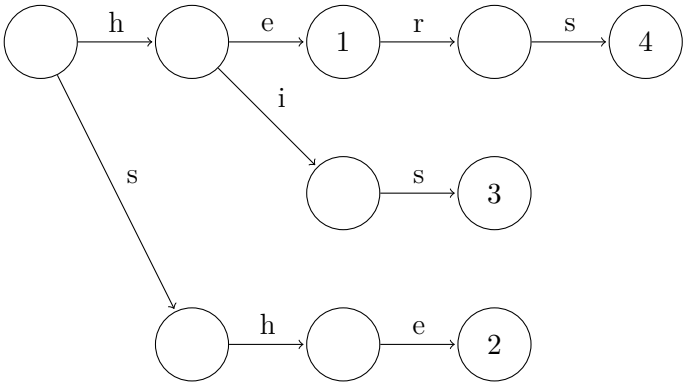

Figure 1. A Trie Example

After the trie was built, we could extend it into an automaton to support linear-time matching. The states of the AC automaton are the nodes of the trie, and the initial state is the root node. The transition funtcion is actually a set of three functions, which can be defined as below.

Definition 1. Define the **next** funtion $g(q, a)$ which gives the state entered from current state q by matching target char a. Next function can be built from trie. If edge (q, v) is labeled by a, then $g(q, a) = v$. $g(0, a) = 0$ for each a that does not label an edge out of the root. The automaton stays at the initial state while scanning non-matching characters. Otherwise $g(q, a) = \emptyset$.

Definition 2. Define the **failure** funtion $f(q)$ which gives the state entered at a mismatch. Define the label of a node v as the concatenation of edge labels on the path from the root to v, and denote it by $L(v)$. $f(q)$ is the node labeled by the longest proper suffix w of $L(q)$, such that w is a prefix of some pattern. A fail transition does not miss any potential occurrences.

Here $f(q)$ looks similar to the failure function defined in Knuth-Morris-Pratt algorithm. In fact, you can see an AC automaton as the combination of Trie and KMP.

Definition 3. Define the **ending** function $e(q)$, which gives the set of patterns recognized when entering state q.

To compute fail and ending functions, we could use a queue to maintain the nodes in a breadth-first order, which ensures nodes closer to the root to be processed earlier. Consider nodes r and $u = g(r, a)$, that is, r is the parent of u. $f(u)$ should be the deepest node labeled by a proper suffix of $L(u)$.

The time complexity of AC automaton's construction in all is $O(n)$. Full proof is contained in Aho's paper [1]. An AC automaton of $P = \{he, she, his, hers\}$ is shown in fig. 2. The next function is drawn as arrowed lines, the fail funtion is drawn as dashed lines, and the ending function is drawn as blocked texts.

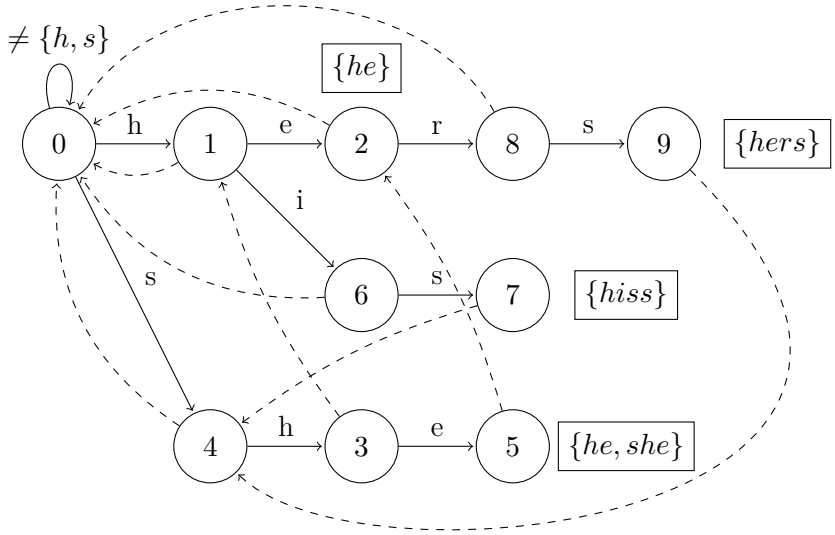

Figure 2. An AC Automaton Example

Now, as the construction is finished, we can use AC automaton to match strings. Starting from the initial state q, we enumerate the characters in string T. For each character, firstly move current state to $f(q)$ while $g(q, T[i]) = \emptyset$, then move to $g(q, T[i])$. If $e(q) \neq \emptyset$, it means that we've found a matching string. The time complexity is $O(m)$.

4. Usage

As a participant of competitive programming contests, I often use AC automaton to solve string-related problems. It is easy to implement, yet it works powerfully in a wide range of problems, by attaching additional informations on nodes and modifying the query function. There are also some interesting problems that combined AC automatons and dynamic programming[2].

AC automaton has important biological applications. [2] It can be used for matching against known DNA databases, what is significant in confirming the identity of new DNA sequences. It's an optimal choice as the search time is independent of the size of the DNA database.

AC automaton also formed the basis of the original Unix command-line utility **fgrep**, which is a variant of **grep** (**g**lobally search a **r**egular **e**xpression and **p**rint). It searches for any of a list of fixed strings.

5. Implementation

I implement AC automaton using C++, with main part shown below. Full code with test functions is available in my algorithm library[3].

```
#define SZ 500010
struct AC {
    int next[SZ][26], fail[SZ], end[SZ];
    int root, L;
    int newnode() {
        for (int i = 0; i < 26; i++) {
            next[L][i] = -1;
        }
        end[L++] = 0;
        return L - 1;
    }
    void init() {
        L = 0;
        root = newnode();
    }
    void insert(char buf[]) {
        int len = strlen(buf);
        int now = root;
        for (int i = 0; i < len; i++) {
            if (next[now][buf[i] - 'a'] == -1) {
                next[now][buf[i] - 'a'] = newnode();
            }
            now = next[now][buf[i] - 'a'];
        }
        end[now]++;
```

[2] http://acm.hdu.edu.cn/showproblem.php?pid=2243

[3] http://github.com/unisolate/acm-code

```
26      }
27      void build() {
28          queue<int> Q;
29          fail[root] = root;
30          for (int i = 0; i < 26; i++)
31              if (next[root][i] == -1) {
32                  next[root][i] = root;
33              } else {
34                  fail[next[root][i]] = root;
35                  Q.push(next[root][i]);
36              }
37          while (!Q.empty()) {
38              int now = Q.front();
39              Q.pop();
40              for (int i = 0; i < 26; i++) {
41                  if (next[now][i] == -1) {
42                      next[now][i] = next[fail[now]][i];
43                  } else {
44                      fail[next[now][i]] = next[fail[now]][i];
45                      Q.push(next[now][i]);
46                  }
47              }
48          }
49      }
50      int query(char buf[]) {
51          int len = strlen(buf);
52          int now = root;
53          int res = 0;
54          for (int i = 0; i < len; i++) {
55              now = next[now][buf[i] - 'a'];
56              int temp = now;
57              while (temp != root) {
58                  res += end[temp];
59                  end[temp] = 0;
60                  temp = fail[temp];
61              }
62          }
63          return res;
64      }
65  };
```

6. Summary

The applications of finite state machines covers almost every field one can imagine, while finite state machine based string matching algorithms just occupied a tiny part of them. Amazed by the elegancy of AC automaton, I wonder more about what finite state machines can do. So as for now, I'll continue my study on the theory of computation, and hope to see more scenes on the road of exploring the land of knowledges.

References

1. Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

2. Pekka Kilpelainen. Set matching and aho-corasick algorithm. *Biosequence Algorithms, Spring 2005, University of Kuopio*, 2005.