# String Algorihtms II

AC Automata

## Introduction

We will study the multi-pattern string matching problem. Given a set of $n$ strings $K = \{P_1, P_2, \dots, P_n\}$ as patterns, we are to test whether a string $S$ has a substring in $K$.

We will introduce the AC automata that is designed specially for this problem. We will first talk about the construction of AC automata, and then analyze the time complexity. Finally, we will show some typical ICPC problems that can be solved using AC automata.

## Limitation of Trie

Recall that Trie is a dictionary structure for strings. Given a string $S$, a Trie built on $K$ can be used to test wether a *prefix* of $S$ is in $K$ efficiently.

However, in the string matching problem, we are asked to test the existence of *substrings* of $S$ in $K$, which cannot be efficiently done using Trie.

## Extending Trie

We shall extend Trie to support the desired function.

Suppose $T$ is the Trie constructed by adding all strings in $K$. For any state $q$ of $T$, we define the string corresponding to the path from the start state (root) to $q$ to be $s(q)$. Let $S \subset_p T$ denote string $S$ is a prefix of string $T$, and $S \subset_s T$ denotes string $S$ is a suffix of string $T$.

## Fail Transition

We define a function $f$ that takes a state and returns another state, such that

$$f(q) = \begin{cases} \arg\max_{q' \neq q : s(q') \subset_s s(q)} \{|s(q')|\}, & \text{for } q \neq q_0 \\ \text{NULL}, & \text{for } q = q_0 \end{cases}$$

That is, $f(q)$ is the state $q'$ has longest $s(q')$ such that $s(q')$ is a suffix of $s(q)$. We call the transition defined by $f$ to be *fail transitions*.

## Accept State

We futher extend the accept states.

If $q$ is an accept state in the Trie, then it is still an accept state. Otherwise, if $\exists q'$ that is an accept state and can be reached by following the fail transition from $q$, then $q$ is an accept state.

We note that this is well defined, since whether $q$ is an accept state depends only on the the state $q'$ that is with smaller height in $T$ (so this is not a circular definition).

## Matching Using AC Automata

Before we talk about the construction of fail transition, we first show that if we are given an AC automata, how we do the string matching efficiently.

As we shall see, the mathcing proceudre is different from a traditional automata matching, since it may use the fail transition in some cases.

```
Define $q = q_0$.
for $c \in S$ do
    if $\delta(q, c) \neq NULL$ then
        $q = \delta(q, c)$.
    end
    else
        while $q \neq q_0$ do
            if $\delta(q, c) \neq NULL$ then
                $q = \delta(q, c)$, break the loop.
            end
            else
                $q = f(q)$.
            end
            /* Report a match if $q$ is an accept state.
                */
        end
    end
end
```

**Algorithm 1:** Matching

## Matching: Correctness

We first show the correctness of the algorithm. That is, if and only if the algorithm reports a match, the string $S$ has a substring in $K$.

By the definition of accept states, we know that if a state $q$ is an accept state, then there exists $q'$ that is origianlly an accept state in the Trie such that $s(q') \in_s s(q)$. This implies the "if" part. The other direction is that if a string $S$ contains a string in $K$ then it will be accepted. Define the state corresponding to $S[1, i]$ to be $q^{(i)}$. We can prove by induction on $i$ that $q^{(i)} = \arg\max_{q : s(q) \in_s S[1,i]} \{|s(q)|\}$. The detail is omitted.

## Matching: Time Complexity

Claim

*The algorithm runs in $O(|S|)$ time.*

Proof.

It is sufficient to bound the number of transitions made. Define the transitions made using the transition in the original Trie as normal transition, and others fail transition.

Define $\alpha$ as the number of normal transitions, and $\beta$ as the number of fail transitions. We shall upper bound $\alpha + \beta$. We observe that if one normal transition is taken, then exactly one character in $S$ is consumed. Therefore, $\alpha = |S|$.

On the other hand, each fail transition will decrease the height of the state in the tree. However, the matching starts at the start state that is of height 0, and to increase the height by 1, we must take a normal transtion. Therefore, $\beta \leq \alpha$. In conclusion, $\alpha + \beta \leq 2\alpha \leq 2|S|$. □

## Constructing the Fail Transition

Now we turn to the construction of the fail transitions. We describe a procedure that takes a Trie, and returns a function $f$ that denotes the fail transitions.

## Constructing the Fail Transition

```
Let $Q$ be the sequence of states organized in the BFS order.
Let $f(q_0) = \text{NULL}$.
for $q \in Q$ do
    for $c \in \Sigma$ and $\delta(q, c) \neq NULL$ do
        Let $q' = f(q)$.
        while True do
            if $q' = NULL$ then
                Set $f(\delta(q, c)) = q_0$, and break the loop.
            end
            if $\delta(q', c) \neq NULL$ then
                Set $f(\delta(q, c)) = \delta(q', c)$, and break the loop.
            end
            Let $q' = f(q')$.
        end
    end
end
```

**Algorithm 2:** Fail Transition Construction

## Explanation

The algorithm calculates $f(\delta(q, c))$, for all $q$ and $c$ such that $\delta(q, c) \neq$ NULL.

Fix $q$ and $c$. The algorithm follows the fail transitions starting from $q$. If it finds some $q'$ such that $\delta(q', c) \neq$ NULL, then it sets $f(\delta(q, c)) := \delta(q', c)$. Or eventually, $q'$ becomes NULL, then the fail transition for $\delta(q, c)$ has to be $q_0$.

The reason that we construct the fail transition in BFS order is to ensure the correctness.

## Analysis

The correctness can be shown by induction on the iteration of the algorithm. We omit the proof.

We turn to the analysis of the time complexity. We consider each $P_i$ that is inserted in Trie. Suppose $|P_i| = l$ and we denote the $l+1$ nodes on the path of the tree as $u_0, u_1, \ldots, u_l$. We consider the calculation of their fail transitions. For a state $q$, we define the height of $q$ to be $h(q)$. Then we observe that $h(f(u_{i+1})) \leq h(f(u_i)) + 1$.

This implies that the total number of "while" iterations for calculating the fail transitions for all $u_0, u_1, \ldots, u_l$'s is at most $l$. Therefore, we conclude that the total running time is $O(|T|)$.

## Marking the Accept States

We have finished describing the construction of the transition function. It remains to calculate the new accept states.

The accept states can be calculated by its recursive definition. In particular, we may evaluate the accept states during the BFS procedure, and apply the rule for the state $q$

- If $q$ is an accept state in Trie, then $q$ is still an accept state.
- If $f(q)$ is an accept state, then $q$ is an accept state.

## Application: Counting Illegal Words

Problem statement is at `http://acm.timus.ru/problem.aspx?space=1&num=1158`.

In general, if we have an automata that recognizes the language of illegal words, then we can use an automata DP to count the number of illegal words.
However, the AC automata on the patterns is not an automata recognizes the language we desire. The matching procedure makes use of the fail transition for acceleration, instead of using the pre-built transition function.
Therefore, what we need to do is to remove the fail transition, and encode the behaviour of fail transition into the transition function $\delta$. Note that this proceudre may take quadratic time, but is sufficient to solve the problem.

Question: what if the length is large, say $10^9$, but we need the answer mod $10^9 + 7$?

## Application: Bit Mask

The problem is at
`http://acm.hdu.edu.cn/showproblem.php?pid=4057`.
We are given a set of $n$ pettern DNAs, and each one has a value (can be negative). For a DNA string $S$, the value of $S$ is the sum of the value of the pettern DNAs that have appeared at least once in $S$ (overlapping is allowed). Note that the multiple appearances only counts once.
Given an integer $l$ ($l \leq 100$), our goal is to calculate the maximum value of DNA of length $l$.

## Application: Bit Mask

We apply the automata DP technique.

We let $f[q][S][l]$ denote the maximum value of the string of length $l$, with patterns $S$ appear, and when input to the automata the state is $q$. Here $S$ is a $n$ dimensional binary vector, indicating the appearance of the patterns. We note that we first convert the AC automata to a normal automata.

The recursion is left as exercise.

## Application: Monkey at The Keyboard[1]

We have a keyboard of 26 lower case letters. A monkey types at the keyboard uniformly at random (each time it will press 1 key only). We are given a string $S$ ($|S| \leq 30000$), and we let the monkey stop typing exactly when we see the first occurence of $S$. We are to calculate the expected length of string the monkey must type until stop.

[1] `http://acm.timus.ru/problem.aspx?space=1&num=1677`

## Monkey at The Keyboard

We still use the DP technique, and we build the AC automata on the single string $S$.[2]

Note that the Trie now is a chain. We label the states corresponding to $S_i$ as $q_i$. Let $f[q]$ denote the expected length until stop from state $q$. Then, $f[q_i] = \frac{1}{26}(1 + f[q_{i+1}]) + \frac{25}{26} f[\mathrm{fail}(q_i)]$.

Although this is an linear equation system, we can still discover an ordering to solve it efficiently. This is left as exercise.

[2]Now the AC automata has exactly the same structure as in the KMP.

## Application: Fail Graph

Problem statement:
`http://acm.hdu.edu.cn/showproblem.php?pid=4117`.

We use $W_i$ to denote the $i$-th word, and $v_i$ to denote its score. The naive approach is a dynamic programming. Let $f[i]$ denote the maximum score among the first $i$ words such that $W_i$ must be selected. Then $f[i] = \max_{j:j<i, W_j \text{ is a substring of } W_i}\{f[j]\} + v_i$, and the final answer is $\max_i\{f[i]\}$.

## Application: Fail Graph

We can use AC automata and Segment Tree for acceleration. We first observe the following fact.

**Fact**

*Suppose $q$ is an accept state in Trie, and the corresponding string is $S$. Denote $L(q)$ to be the accept state set of the strings that contains $S$ as a substring. Then for all $q' \in L$, $q'$ will reach $q$ by following the fail transition.*

Moreover, we have the following.

**Fact**

*Removing the Trie edges in AC automata, we get a DAG which only has fail transition edges. If we interpret each fail transition $(u, v)$ as an undirected edge and define $v$ to be the parent of $u$, then we get a tree rooted at $q_0$. We call this rooted tree fail graph.*

## Fail Graph

Combining the two facts together, we know that the strings that has some string $S$ as a substring corresponds to the accept states in the subtree rooted at the accept state of $S$, where the subtrees are defined in terms of the fail graph.

Therefore, for some word $W_i$, the words that has $W_i$ as its substring can be organized in a subtree of the fail graph. To accelerate the dynamic programming, it is sufficient to have a data structure for some node weighted tree that supports:

- Update(q, v). Define the subtree rooted at $q$ to be $T_q$. This operation requires for each $q' \in T_q$, $v(q') = \max\{v(q'), v\}$, where $v(q)$ denotes the node weight.
- Query(q). Return the weight of node $q$.

This can be done by using Segment Tree on the DFS sequence of the tree.