Algorithm Homework 12

Daoyu Wang

November 30

Contents

1 32.1-2

1.1 Analysis

Assume the text is T and the pattern is P and every character in P is different. Different from the **NAIVE-STRING-MATCHER** that if P and the substring of T are unmatched, we start new from next index of T, the improved **STRING-MATCHER** compares every character not substring and starts new from the first index that unmatched or next index if all characters in pattern are matched. Be care that this *index* is the index of T not the index of P.

For example, if $T = abcabfab$ and $P = abc$, firstly match the $T[0]$, $T[1]$ and $T[2]$, then we start from $T[3]$ but not $T[1]$, then match $T[3]$ and $T[4]$ but not $T[5]$, then start from $T[5]$ but $T[4]$. Since every character in P is different, we can easily find index jumping is reasonable because distance of the first index of two different matches will never be less than the length of P.

We can find that the index of T will never decrease and the addition of increase of index in T and P is n, so the running time of improved **STRING-MATCHER** is $O(n)$.

1.2 Algorithm

Algorithm 1 Improved String Matcher

function STRING-MATCHER($string\ text \quad string\ pattern$)
    $n \leftarrow text.size$
    $m \leftarrow pattern.size$
    $ans \leftarrow \emptyset$
    for $i \leftarrow 0$ to $n - m$ **do**
        $j \leftarrow 0$
        while $text[i] = pattern[j]$ **do**
            $j \leftarrow j + 1$
            $i \leftarrow i + 1$
        end while
        if $j = m$ **then**
            $ans.push(i - m + 1)$
        end if
    end for
    $return\ ans$
end function

2 32.1-4

2.1 Analysis

Assume the pattern is divided into k parts with length $\{m_1, m_2, \cdots, m_k\}$ and the pattern is $s_1\diamondsuit s_2\diamondsuit\cdots\diamondsuit s_k$.

Since we just need to find whether the pattern P exists in text P, we only need to find the first occurence(be care that the occurence means the last index of T matching each part) of the first part s_1 in T and then find the first occurence of the second part s_2 in T from the index of the last character of s_1 in T and so on. If we can find all parts in T, then we can find P in T. And if we can't find one part in T, we can stop the algorithm and don't need to find any second occurence because if we start from any second occurence and finally find a part that match, starting from the corresponding first occurence can also find it.

Let's analyse its running time: since we find next part from the index of the last character of the previous part, the index of T will never decrease. Consider a case that we finally find a match and the first occurence of each part is $\{x_1, x_2, \cdots, x_k\}$: call function **NAIVE-STRING-MATCHER**, the first part needs $m_1(x_1 - m_1 + 1)$, the second part needs $m_2(x_2 - x_1 - m_2 + 1)$, and so on. So if we let $x_0 = 0$, the total running time is as follows:

$$T = \sum_{i=1}^{k} m_i(x_i - x_{i-1} - m_i + 1) \tag{1}$$

Assume $\min_{i\in\{1,2,\cdots,k\}} m_i = min_m$ while $\max_{i\in\{1,2,\cdots,k\}} m_i = max_m$, then we can get that $min_m \le m_i \le max_m$ for all $i \in \{1, 2, \cdots, k\}$. Assume $m = \sum_{i=1}^{k} m_i$ which represents the length of pattern, so we have:

$$\begin{aligned} T &= \sum_{i=1}^{k} m_i(x_i - x_{i-1} - m_i + 1) \\ &\le \sum_{i=1}^{k} max_m(x_i - x_{i-1} - min_m + 1) \\ &= max_m \sum_{i=1}^{k}(x_i - x_{i-1} - min_m + 1) \\ &= max_m(x_k - x_0 - k \cdot min_m + k) \end{aligned} \tag{2}$$

Consider that $x_k \le n$, $x_0 \ge 0$, $max_m \le m$ and $min_m \ge 1$, so we have:

$$\begin{aligned} T &\le max_m(x_k - x_0 - k \cdot min_m + k) \\ &\le m(n - 0 - k \cdot 1 + k) \\ &= m(n - k + k) \\ &= mn \end{aligned} \tag{3}$$

So the running time of this algorithm is $O(mn)$.

2.2 Algorithm

Algorithm 2 Multi-part String Matcher

```
function MULTI-PART-MATCHER(string text, vector pattern)
    n ← text.size
    k ← pattern.size
    vector m ← ∅
    ans ← true
    for i ← 0 to k − 1 do
        m.push(pattern[i].size)
    end for
    for i ← 0 to k − 1 do
        isfind ← FIND − FITST − INDEX(text, pattern[i])
        ans ← ans and (isfind ≠ −1)
        if ans = false then
            break
        end if
    end for
end function
```

where function **FIND-FITST-INDEX(string, string)** is to find the first occurence of the second string in the first string and return the index of its occurence if it exists and -1 if not which is similar to **NAIVE-STRING-MATCHER**.

3 32.2-3

Firstly, let's consider the pattern matrix $M(m,m)$: if we regard each column in m columns as a whole, then each of them is a one-dimensional string and we can calculate the **hash value**, so the pattern matrix becomes a one-dimensional string with length m.

Then for the first m rows of text matrix $N(n,n)$, we can get a string with length n using the same method above.

Then matching the pattern matrix in the text matrix can be transfered to one-dimensional string matching, and we can use **NAIVE-STRING-MATCHER** algorithm to solve it.

Finally, using the same method, from the second to $m+1$-th row, the third to $m+2$-th row and so on, we can find all the matches. The key of this is that in each process of matching, the transfered one-dimensional string can be calculated in $O(1)$ time from the previous one, using the same method in **Rabin-Karp** algorithm that spending $O(1)$ time to calculate t_{s+1} from t_s.

When we move the pattern matrix **horizontally**, the problem is transfered to one-dimensional string matching, so the running time is $O(m(n-m+1))$.

When we move the pattern matrix **vertically**, the hash value can be calculated in $O(1)$ time from the previous one. Be care that the numbers of vertically moving is $n-m+1$, so the running time is $O((n-m+1)\cdot m(n-m+1)) = O(m(n-m+1)^2)$.

4 32.3-5

Assume the pattern is divided into k parts with length $\{m_1, m_2, \cdots, m_k\}$ and the pattern is $s_1\diamondsuit s_2\diamondsuit\cdots\diamondsuit s_k$.

Separately, suppose that $(Q_i, q_{i,0}, A_i, \Sigma_i, \delta_i)$ is the DFA corresponding to the pattern s_i.

Then, suppose the concrete DFA is $Q, q_0, A, \Sigma, \delta$ where $Q = \bigcup_{i=1}^{k} Q_i$, $q_0 = q_{0,0}$, $A = A_k$, $\Sigma = \bigcup_{i=1}^{k} \Sigma_i$ and δ is as follows:

If we are at state $q \in Q_i$ and see character a, if $q \notin A_i$, we just go to the state $\delta(q,a) = \delta_i(q,a)$. However, if $q \in A_i$, $\delta(q,a) = \delta_{i+1}(q_{i+1}, 0)$. Be care that q_{i+1} is the initial state of Q_{i+1}.

Then we can get the concrete DFA of matching the pattern P.

5 32.4-1

5.1 Algorithm

The Algorithm $GetPrefix$ is as follows:

Algorithm 3 Get Prefix

function PREFIX_FUNCTION(*string pattern*)
    $n \leftarrow pattern.size$
    $\pi[n] \leftarrow 0$
    for $i \leftarrow 1$ to $n-1$ **do**
        $j \leftarrow \pi[i-1]$
        while $j > 0$ and $pattern[i] \neq pattern[j]$ **do**
            $j \leftarrow \pi[j-1]$
        end while
        if $pattern[i] = pattern[j]$ **then**
            $j \leftarrow j + 1$
        end if
        $\pi[i] \leftarrow j$
    end for
    $return\ \pi$
end function

5.2 Solution

For pattern *ababbabbabbababbabb*, call $prefix\_function$ above, the answer is $\{0, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 4, 5, 6, 7, 8\}$

6 32.4-5

6.1 Algorithm

The Algorithm KMP is as follows:

Algorithm 4 KMP

```
1: function KMP(string text, string pattern)
2:     n ← text.size
3:     m ← pattern.size
4:     π ← PREFIX_FUNCTION(pattern)
5:     ans ← ∅
6:     q ← 0
7:     for i ← 0 to n − 1 do
8:         while q > 0 and pattern[q] ≠ text[i] do
9:             q ← π[q − 1]
10:        end while
11:        if pattern[q] = text[i] then
12:            q ← q + 1
13:        end if
14:        if q = m then
15:            ans.push(i − m + 1)
16:            q ← π[q − 1]
17:        end if
18:    end for
19:    return ans
20: end function
```

6.2 Solution

Consider two different cases to operate the pointer q:

- A: $pattern[q] = text[i]$, then $q \leftarrow q + 1$
- B: $pattern[q] \neq text[i]$, then $q \leftarrow \pi[q - 1]$

And according to the definition of $prefix\_function$, we can get that $\pi[q-1]$ will never be greater than $q - 1$. Let the potential function be the position of q. This means that when we execute line 12, we pay a constant amount to raise up the potential function. And when we execute line 9 and line 16, we decrease the potential function which reduces the amortized cost of an iteration of the while loop to a zero amortized cost. As a result, we can easily find that numbers of executions on operator A will never be less than numbers of the executions on operator B.

Obviously, the running time of single A and B is $O(1+1)$(one is its original running time and another is the change of potential function), the running time of the whole A is $O(n)$ because the maximum loops of for in line 7 is n. So we have $O(n) = T(A) \geq T(B)$. The whole running time of matching is $T(A+B) = O(n)$. Consider that the running time of $PREFIX\_FUNCTION$ is $O(m)$, so the running time of KMP is $O(n+m)$.