研究生课程考试成绩单

（试卷封面）

院　　系	计算机科学与工程学院	专业	电子信息		
学生姓名	茅伟龙	学号	212122		
课程名称	Web 数据管理技术				
授课时间	2022 年　3 月至　2022 年　6 月	周学时	2	学分	2
简 要 评 语					
考核论题					
总评成绩（含平时成绩）					
备注					

任课教师签名：

日期：

注：1. 以论文或大作业为考核方式的课程必须填此表，综合考试可不填。“简要评语”栏缺填无效。

2. 任课教师填写后与试卷一起送院系研究生秘书处。

3. 学位课总评成绩以百分制计分。

目录

1 Problem Definition

Pattern matching in computer science is the process of determining if the elements of a given pattern are present in a given sequence of tokens. Strings of characters are used in pattern matching, which is by far the most popular type. An important class of string algorithms called string-matching or string-searching attempts to locate the location where one or more strings (also known as patterns) are found inside a larger string or text. A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) $\Sigma.\Sigma$ may be a human language alphabet, for example, the letters A through Z and other applications may use a binary alphabet

$(\Sigma = \{0,1\})$ or a DNA alphabet $(\Sigma = \{A, C, G, T\})$ in bioinformatics.

2 Basic Idea

The Boyer-Moore method employs explicit character comparisons at various alignments to look for instances of P in T . Boyer-Moore leverages data obtained from preprocessing P to choose which alignments to skip as many times as possible rather than performing a brute-force search of all alignments (there are $n - m + 1$).

The traditional method of searching within text before the invention of this algorithm involved looking at each character of the text to find the first character of the pattern. Once it was discovered, the text's succeeding characters would be checked against the pattern's characters. If there was no match, each character of the text was searched once more to see if there was a match. Thus, it is necessary to scrutinize practically every character in the text.

The essential finding of this algorithm is that rather than examining each character of the text, leaps through the text can be performed if the end of the pattern is compared to the text. This works because the last character of the pattern is compared to the character in the text when the pattern and text are lined up against each other. There is no need to look further back along the text if the characters don't match. The next character in the text to check is found mm characters further along the text, where m is the length of the pattern, if none of the characters in the pattern match the character in the text. If the pattern contains the character in the text, the pattern is partially shifted along the text to line up with the matching character, and the process is then repeated. The key to the algorithm's effectiveness is reducing the number of comparisons that must be made by jumping along the text rather than verifying each character individually.

More formally, the algorithm begins at alignment $k = m$, so the start of P is aligned with the start of T. Characters in P and T are then compared starting at index m in P and k in T, moving backward. The strings are matched from the end of P to the start of P. When a mismatch occurs, the alignment is shifted forward (to the right) by the greatest value allowed by a number of rules, or until the beginning of P is reached (which indicates that there is a match). The process is repeated until the alignment is

shifted past the end of T, at which point no further matches will be discovered. The comparisons are then carried out at the new alignment.
The shift rules are implemented as constant-time table lookups, using tables generated during the preprocessing of P.

3 Method

3.1 Algorithm Description

- T denotes the input text to be searched. Its length is $\mathbf{n}$.
- P denotes the string to be searched for, called the pattern. Its length is $\mathbf{m}$.
- $\mathbf{P}$ denotes one proposition.
- S_i denotes the character at index i of string S, counting from 1.
- $S_{i,j}$ denotes the substring of string S starting at index i and ending at j, inclusive.

3.1.1 Basic Rule

$$T = P \Leftrightarrow m = n, \forall i \in [\mathbf{1}, \mathbf{m}], T_i = P_i$$

$$\exists i \in [\mathbf{1}, \mathbf{m}], T_i \neq P_i \Leftrightarrow T \neq P$$

$$T_k \neq P_i \Rightarrow T_{1-i+k,n-i+k} \neq P_{i,n}$$

3.1.2 The Bad Character Rule

1) Description of the bad character rule

Suppose that P_1 is aligned to T_s now, and we perform a pair-wise comparing between text T and pattern P from right to left. Assume that the first mismatch occurs when

comparing T_{s+j-1} with P_j.

Since $T_{s+j-1} \neq P_j$, we move the pattern P to the right such that the largest position c in the left of P_j is equal to T_{s+j-1}. We can shift the pattern at least $(j-c)$ positions right.

2) Proof of the bad character rule

Consider the proposition, P : "We can shift the pattern at least $(j-c)$ positions right. ". In a proof by contradiction, we start by assuming the opposite, $\neg P$: We can shift the pattern less than $(j-c)$ positions right, say, $r < j-c$.

Now, P_{j-r} is going to compare with T_{s+j-1}. But that contradicts the assumption that $j-c$ was the smallest shifted position that $P_{j-c} = T_{s+j-1}$. According to basic rule2, when P_r is aligned to T_{s+j-1}, thus two patterns don't match. That is, that "We can shift the pattern at least $(j-c)$ positions right.".

3.1.3 The Good Character Rule1

1) Description of the good suffix rule1

If a mismatch occurs in T_{s+j-1}, we match T_{s+j-1} with $P_{j-m+j'}$, where j' $(m-j+1 \leqq j' < m)$ is the largest position such that

(1) $P_{j+1,m}$ is a suffix of $P_{1,j}$,

(2) $P_{j'-(m-j)} \neq P_j$.

We can move the window at least $(m-j')$ position(s).

2) Proof of The Good Suffix Rule1

Consider the proposition, **P** :

If a mismatch occurs in T_{s+j-1} , we match T_{s+j-1} with P_{j-m+j} , where $j'(m-j+1 \leqq j' < m)$ is the largest position such that

(1) $P_{j+1,m}$ is a suffix of $P_{1,j}$,

(2) $P_{j'-(m-j)} \neq P_j$.

We can move the window at least $(m-j$ ') position(s).

In a proof by contradiction, we start by assuming the opposite, **¬P** :

$Pj'-(m-j)=Pj \vee Pj+1,m$ is not a suffix of $P1,j$

This problem will be discussed on two cases.

In the case of $P_j \neq T_{s+j-1}$, thus that $P_{j'-(m-j)} \neq T_{s+j-1}$. Therefore, we can't move the window $(m-j$ ') position(s).

In the case of $P_{j+1,m}$ is not a suffix of $P_{1,j}$, there must be one character doesn't match when move the window with $(m-j')$ position(s) according to basic rule2, in another word , $P_{j'-m+j+1,j'} \neq P_{j+1,m}$. However, $T_{s+j,m} = P_{j+1,m}$, therefore $T_{s+j,m} \neq P_{j'-m+j+1,j'}$. Therefore, we can't move the window $(m-j$ ') position(s).

3.1.4 The Good Character Rule2

1) Description of the good suffix rule2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 can not be used. That is, t does not appear in $P(1,j)$. Thus, t is unique in P.

If a mismatch occurs in T_{s+j-1}, we match $T_{s+m-j'}$ with P_1, where $j'(1 \leqq j' \leqq m-j)$ is the largest position such that $P_{1,j'}$, is a suffix of $P_{j+1,m}$.

2) Proof of the good suffix rule2

Consider the proposition, **P**:

If a mismatch occurs in T_{s+j-1}, we match T_{s+m-j} ' with P_1, where $j'(1 \leqq j' \leqq m-j)$ is the largest position such that $P_{1,j}$, is a suffix of $P_{j+1,m}$.We can move the window at least $(m-j$ ') position(s).

In a proof by contradiction, we start by assuming the opposite, **¬P**:

$j''(1 \leqq j' < j'' \leqq m-j)$ is larger than the largest position such that $P_{1,j'}$, is a suffix of $P_{j+1,m}$.

In another word $P_{1,j''}$, is not a suffix of $P_{j+1,m}$. Therefore, $P_{1,j''} \neq P_{m-j''+1,m}$. However, $P_{m-j''+1,m} = T_{s+m-j'',s+m-1}$. Thus $P_{1,j''} \neq T_{s+m-j'',s+m-1}$. That is, that "If a mismatch occurs in T_{s+j-1}, we match T_{s+m-j} ' with P_1, where $j'(1 \leqq j' \leqq m-j)$ is the largest position such that $P_{1,j}$, is a suffix of $P_{j+1,m}$.We can move the window at least $(m-j$ ') position(s).".

3.2 Implementation Consideration

此部分为邹刘文同学撰写

3.2.0 Implementation Overview

3.2.1 Construction of the Suffix Function

I. Method One
II. Method Two

III. Method Three

3.2.2 The purpose and method of using suffix function

3.2.3 The Use of the Suffix Function for Good Suffix Rule 1

3.2.5 The Overall Pipeline

4 Performance Analysis & Experiments

此部分为邹刘文同学撰写

4.1 Implementations By Java

4.2 Experiments

4.3 Performance Analysis

5 Conclusions

此部分为邹刘文同学撰写

6 References

[1]Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, *20*(10), 762-772.

[2]wzhang1117. KMP 算法求 next 数组 [EB/OL]. [2022-07-20]. https://www.zybuluo.com/wzhang1117/note/27431.

[3]tcfellow. C++字符串与 KMP 算法和 BM 算法时间复杂度 [EB/OL]. [2022-07-20]. https://juejin.cn/post/6844903635101417485.

[4]igm. Turbo-BM algorithm [EB/OL]. [2022-07-20]. http://www-igm.univ-mlv.fr/~lecroq/string/node15.html#SECTION00150