简介: [TOC]

Boyer-Moore算法介绍

概述

Boyer-Moore算法（BM算法）是由Robert S. Boyer和J Strother Moore在1977年提出的一种字符串匹配算法。该算法在查找一个子串在主串中首次出现的位置时表现非常出色，尤其在处理较长的文本时具有高效性。它之所以高效，是因为在匹配过程中可以跳过一些无意义的字符，从而减少比较次数。

BM算法的核心思想是通过分析文本和模式串之间的关系，提前预测某些字符的匹配结果，并在可能的情况下进行大步跳跃，以尽量避免不必要的字符比较。BM算法主要通过两个启发式规则来实现跳跃：**坏字符规则**和**好后缀规则**。

核心思想

1. 坏字符规则（Bad Character Rule）

坏字符规则主要是指在比较过程中，当模式串中的某个字符与主串中的字符不匹配时，可以根据主串中的不匹配字符跳过一些可能不会匹配的位置。其基本原理是将模式串向右移动，直到该不匹配字符在模式串中的下一个匹配位置出现，或者模式串完全移过该不匹配字符。坏字符规则可以避免逐一比较，提升算法效率。

2. 好后缀规则（Good Suffix Rule）

好后缀规则是当模式串的一部分在主串中匹配时，如果发现有一部分匹配，接下来发生了不匹配的情况，那么可以利用已经匹配的部分（即好后缀）来判断模式串应如何继续向右移动。好后缀规则的目的是确保尽可能大幅度地移动模式串，而不是简单地一位一位地滑动。

这两个规则的结合使得BM算法能够在大部分情况下避免逐字符比较，尤其是在长文本和较短模式串的情况下，效率尤为明显。

BM算法的复杂度

BM算法的时间复杂度依赖于文本和模式串的具体内容。它在最坏情况下的时间复杂度为O(mn)，其中m是模式串的长度，n是主串的长度。但在实际使用中，由于它的跳跃特性，BM算法的平均时间复杂度接近于O(n/m)，这使得它在大多数实际应用场景中非常高效。

应用场景

BM算法特别适用于文本搜索、DNA序列比对、编辑器中的查找替换功能等。其高效的匹配性能使它在处理大规模文本数据时，成为一种常用的字符串匹配算法之一。

代码样例(rust实现)

该代码实现随机生成两段DNA序列并在**找到可以匹配的部分**后输出

```
//@author    Allergy
//@workspace study/bm.rs
//@data      2024/10/10 19:15:05
```

```rust
use rand::Rng;
fn cin() -> String {
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).unwrap();
    input.trim().to_string()
}
fn main() {
    let t = 1; //let t = cin().parse::<i32>().unwrap();
    let _ = (0..t).for_each(|_| solve());
}
fn solve() {
    let (a, b) = cin()
        .split_whitespace()
        .fold((0, 0), |x, y| (x.1, y.parse::<usize>().unwrap()));
    let trans = |x: char| match x {
        'A' => 'T',
        'T' => 'A',
        'C' => 'G',
        'G' => 'C',
        _ => ' ',
    };
    let show = |qwq: &Vec<char>| {
        qwq.iter().for_each(|x| print!("{}", x));
        println!()
    };
    let dna_init = random_dna(a);
    let dna = dna_init.iter().map(|x| trans(*x)).collect::<Vec<char>>();
    let pattern = random_dna(b);

    show(&dna_init);
    println!("↓");
    show(&pattern);
    boyer_moore(&dna, &pattern);
}
fn random_dna(length: usize) -> Vec<char> {
    let nucleotides = ['A', 'T', 'C', 'G']; // DNA碱基
    let mut rng = rand::thread_rng(); // 创建随机数生成器

    // 随机生成指定长度的DNA序列
    let sequence: String = (0..length)
        .map(|_| nucleotides[rng.gen_range(0..4)]) // 随机选择ATCG中的一个
        .collect();

    sequence.chars().collect()
}

fn boyer_moore(text: &Vec<char>, pattern: &Vec<char>) {
    // 坏字符规则：生成坏字符表
    let bad_str = |pattern: &Vec<char>, qwq: &mut Vec<i32>| {
        qwq.iter_mut().for_each(|x| *x = -1);
        let m = pattern.len();
        for i in 0..m {
            qwq[pattern[i] as usize] = i as i32;
        }
```

```
    };
    // 好后缀规则
    let good_suffix = |pattern: &Vec<char>,qwq:&mut Vec<i32>|{

    }
    let n = text.len();
    let m = pattern.len();
    if m == 0 {
        println!("模式串为空，返回0");
        return;
    }
    // 坏字符表（ASCII字符集大小256）
    let mut bad_char = vec![-1; 256];
    bad_str(&pattern, &mut bad_char);
    // 开始匹配
    let mut i = 0;
    while i <= n - m {
        let mut j = m - 1;
        let mut check = false;
        //从末位开始判断
        while pattern[j] == text[i + j] {
            if j == 0 {
                check = true;
                break;
            }
            j -= 1;
        }
        if check {
            println!("成功匹配，索引位于{}", i + 1);
            i += if i + m < n {
                (m as i32 - bad_char[text[i + m] as usize]) as usize
            } else {
                1
            };
        } else {
            i += 1.max(j as i32 - bad_char[text[i + j] as usize]) as usize;
        }
    }
}
```