**字符串匹配算法的综述与优化研究**

刘浩田1

（同济大学 软件学院，上海市 201804）

**摘要：** 本文综述了几种经典的字符串匹配算法，包括暴力匹配算法（Brute Force）、Boyer-Moore算法、Rabin-Karp算法、Aho-Corasick算法及其改进算法。通过分析各算法的原理、实现和性能，本文旨在探讨它们在不同应用场景下的适用性和效率。特别地，对Aho-Corasick算法的改进版本进行了详细讨论，并通过实验验证了其优化效果。本研究为字符串匹配领域提供了新的视角和优化策略。

**关键词：**字符串匹配；Brute Force；Boyer-Moore；Rabin-Karp；Aho-Corasick；算法优化

**中图分类号：**（本网站首页链接可检索，交叉学科可列多个） **文献标志码：**A（学术论文为A）

**A Survey and Optimization Study of String Matching Algorithms**

*LIU Haotian*1

(School of Software, Tongji University, Shanghai, 201804,China)

**Abstract:** This paper reviews several classic string matching algorithms, including the Brute Force algorithm, Boyer-Moore algorithm, Rabin-Karp algorithm, Aho-Corasick algorithm, and its improved version. By analyzing the principles, implementations, and performance of each algorithm, this paper aims to explore their applicability and efficiency in different scenarios. In particular, a detailed discussion on the improved version of the Aho-Corasick algorithm is provided, and its optimization effects are verified through experiments. This study offers new perspectives and optimization strategies in the field of string matching.

**Key words:** String Matching; Brute Force; Boyer-Moore; Rabin-Karp; Aho-Corasick; Algorithm Optimization

在当今信息技术飞速发展的时代，字符串匹配算法作为计算机科学中的一个基础而重要的问题，广泛应用于文本编辑、信息检索、生物信息学、网络安全等多个领域。字符串匹配问题的核心在于在一段文本（目标串）中查找一个或多个特定模式（模式串）的出现，其效率直接影响到这些应用的性能。随着数据量的爆炸性增长，对字符串匹配算法的效率要求也越来越高，这促使了众多经典算法的诞生和优化。

本文将对几种具有代表性的字符串匹配算法进行综述，包括暴力匹配算法（Brute Force）、Boyer-Moore算法、Rabin-Karp算法、Aho-Corasick算法以及对Aho-Corasick算法的改进。暴力匹配算法以其简单直观而著称，尽管效率不高，但仍是理解其他复杂算法的基础。Boyer-Moore算法以其坏字符规则和好后缀规则在实际应用中表现出色，特别是在长模式串匹配中。Rabin-Karp算法通过滚动哈希技术，提供了一种快速检测模式串存在性的方法。Aho-Corasick算法则是一种高效的多模式匹配算法，适用于同时搜索多个模式串的场景。本文还将探讨Aho-Corasick算法的改进版本AC-BM算法，该版本在原有基础上进行了优化，以适应更广泛的应用需求。

通过对这些算法的深入分析和实验验证，本文旨在揭示它们在不同应用场景下的表现和适用性，为字符串匹配算法的研究和应用提供参考。同时，本文也期望能够为算法优化提供新的思路，推动字符串匹配技术的发展。

# 1 字符串匹配算法基础

字符串匹配算法是计算机科学中用于解决特定子串在主串中出现位置的问题的一系列算法。这些算法的核心目标是在给定的文本中快速准确地找到模式串的匹配位置，或者确定模式串不存在于文本中。

## 1.1 字符串匹配问题的定义

字符串匹配问题可以定义为：给定两个字符串，一个较长的称为目标串（T），另一个较短的称为模式串（P）。问题的目标是在目标串T中找到模式串P出现的所有位置，或者确定P不出现在T中。这个问题可以分为两个子问题：单模式匹配和多模式匹配。单模式匹配关注于在目标串中查找单个模式串的所有出现，而多模式匹配则涉及在目标串中查找多个模式串的任意一个的出现。

## 1.2 算法性能的度量标准

算法性能的度量标准主要包括时间复杂度和空间复杂度。时间复杂度是衡量算法执行速度的关键指标，它反映了算法在最坏情况下所需的比较次数或操作次数。空间复杂度则描述了算法在执行过程中对存储资源的需求，包括算法所需的额外存储空间。在字符串匹配算法中，时间复杂度尤为重要，因为它直接影响到算法处理大规模数据的能力。理想的字符串匹配算法应具有较低的时间复杂度和空间复杂度，以适应各种应用场景的需求。

# 2 暴力匹配算法（Brute Force）

暴力匹配算法是字符串匹配领域中最基础的算法之一，其简单性和直观性使其成为理解其他复杂算法的基石。

## 2.1 算法描述和原理

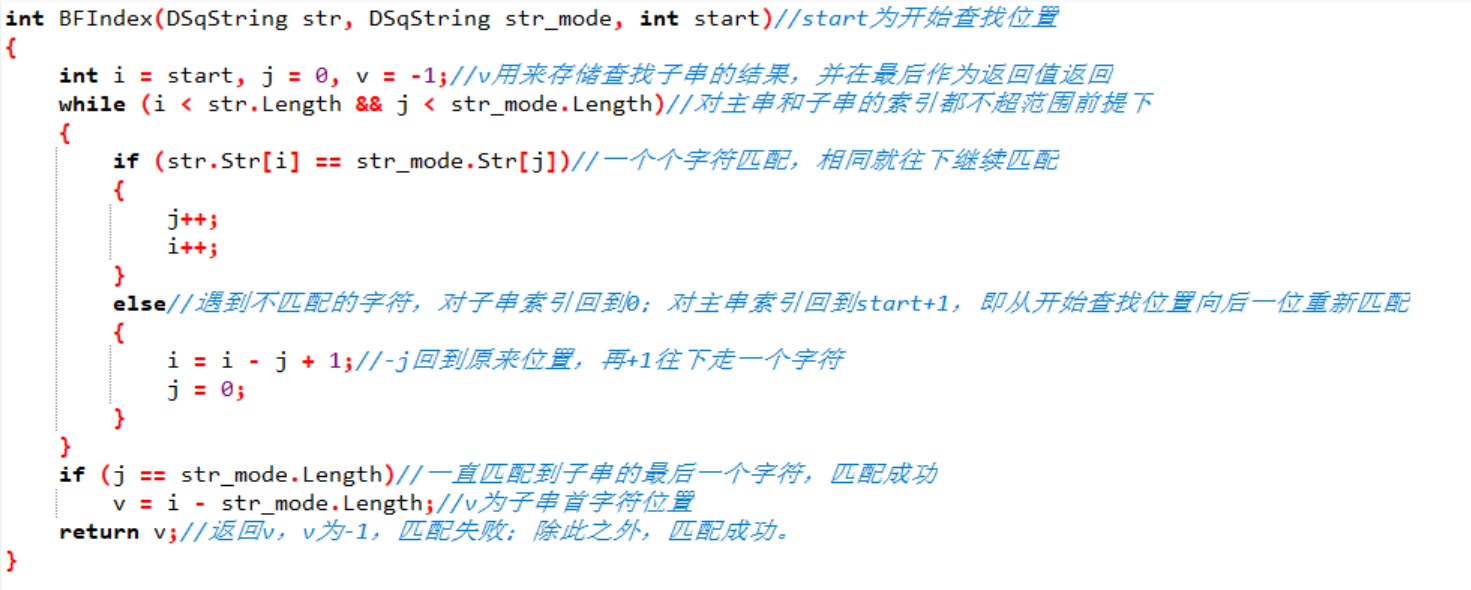


图1 BF算法核心代码

Fig.1 BF algorithm core code

暴力匹配算法的基本思想是将模式串与目标串的每个可能位置进行逐一比较，直到找到匹配或者确定不匹配为止。具体来说，算法从目标串的第一个字符开始，尝试与模式串的第一个字符进行匹配，如果匹配成功，则继续比较后续字符；如果失败，则模式串向右移动一个位置，继续尝试匹配。这个过程一直重复，直到模式串在目标串中找到匹配的位置或者完全遍历目标串。

## 2.2 算法实现和时间复杂度分析

暴力匹配算法的实现直接反映了其原理。在最坏情况下，算法需要对目标串中的每个位置都进行模式串长度的比较，因此其时间复杂度为O(N\*M)，其中N为目标串的长度，M为模式串的长度。这种算法在模式串较短时效率尚可，但随着模式串长度的增加，所需时间呈指数级增长，因此在处理大规模数据时效率低下。

## 2.3 应用场景和局限性

尽管暴力匹配算法在效率上存在局限性，但由于其实现简单，在一些特定场景下仍然有其应用价值。例如，在模式串非常短或者目标串非常短的情况下，暴力匹配算法可以快速得到结果。此外，对于一些对时间要求不高的小规模问题，暴力匹配算法也是一个可行的选择。然而，对于大规模文本或者长模式串的匹配问题，暴力匹配算法的效率问题使其不再适用，需要更高效的算法来解决。

# 3. Boyer-Moore算法

Boyer-Moore算法是一种高效的字符串匹配算法，以其优越的性能在实际应用中广受欢迎[2]。

## 3.1 算法描述和原理

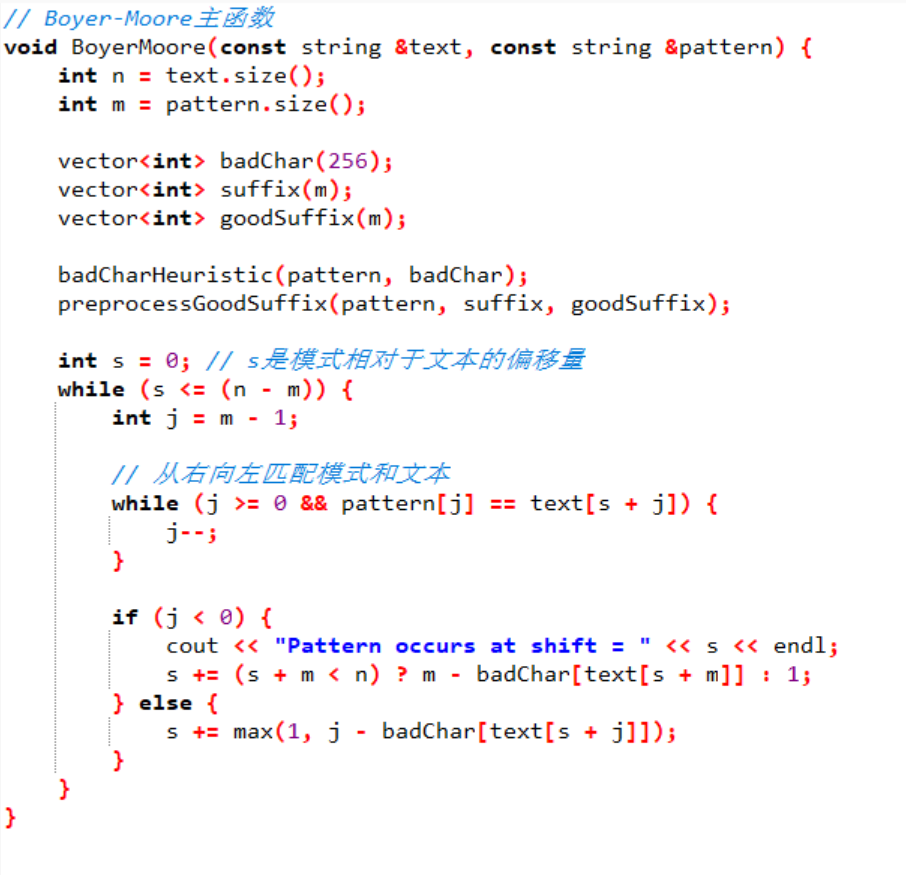


图2 BM 算法核心代码

Fig.2 BM algorithm core code

Boyer-Moore算法的核心在于减少模式串与目标串的字符比较次数。该算法利用模式串的已知信息，从模式串的末尾开始与目标串进行比较，一旦发现不匹配的字符，算法会根据两个规则将模式串向右移动，跳过已经比较过的位置，从而减少不必要的比较。这种方法使得Boyer-Moore算法在最佳情况下可以达到线性时间复杂度。

## 3.2 坏字符规则和好后缀规则的详细解释

Boyer-Moore算法的高效性主要归功于两个关键规则：坏字符规则和好后缀规则。坏字符规则是指，当比较过程中发现不匹配的字符时，算法会查看该字符在模式串中的位置，并将模式串移动到该字符在目标串中下一次出现的位置。如果该字符在模式串中不存在，则模式串移动到目标串的下一个位置。好后缀规则则是基于模式串中已知匹配的后缀和前缀，当发生不匹配时，算法会根据已匹配的后缀和前缀的长度来决定模式串的移动距离。

## 3.3 算法实现和时间复杂度分析

Boyer-Moore算法的实现涉及构建坏字符表和好后缀表。坏字符表记录了模式串中每个字符在目标串中的下一次出现位置，而好后缀表则记录了模式串中所有后缀的最长匹配前缀。算法的时间复杂度在最佳情况下是O(N/M)，其中N是目标串的长度，M是模式串的长度。然而，在最坏情况下，时间复杂度可能退化到O(N\*M)。

## 3.4 与Brute Force算法的比较

与Brute Force算法相比，Boyer-Moore算法在许多情况下都显示出显著的性能优势。Brute Force算法的时间复杂度为O(N\*M)，而Boyer-Moore算法在最佳情况下的时间复杂度远低于Brute Force算法。此外，Boyer-Moore算法不需要额外的存储空间来保存中间状态，这使得它在空间复杂度上也更为高效。尽管如此，Boyer-Moore算法在最坏情况下的性能可能不如Brute Force算法，这通常发生在模式串中的字符分布非常均匀时。

# 4. Rabin-Karp算法

Rabin-Karp算法是一种在字符串匹配领域中以高效率著称的算法，特别是在处理大型数据集时[2]。

## 4.1 算法描述和原理

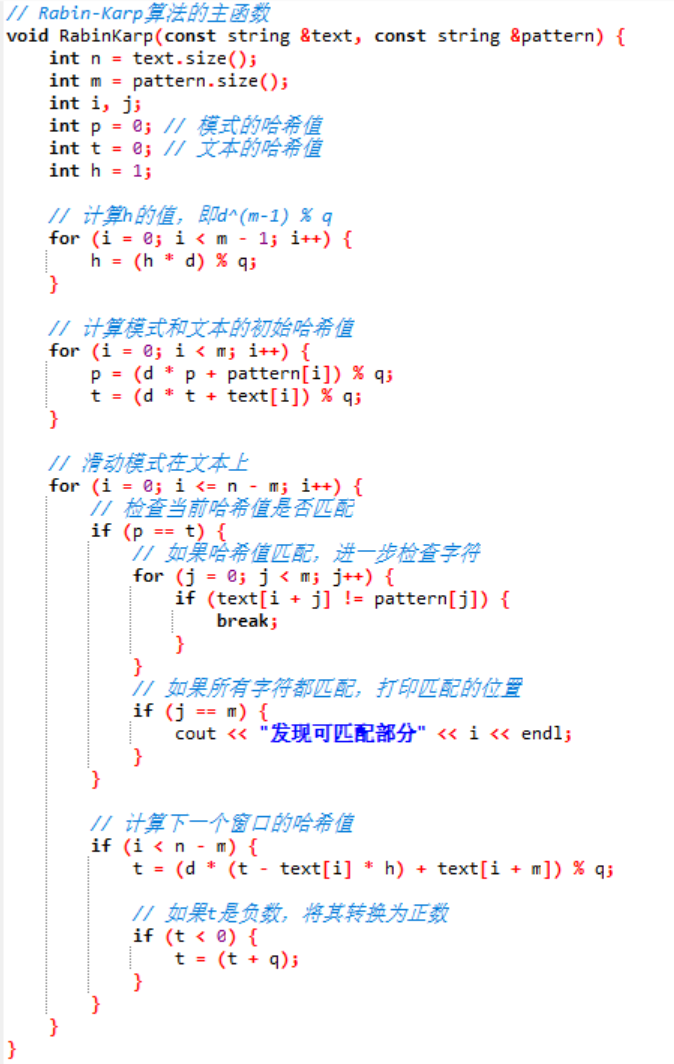


图3 Rabin-Karp 算法核心代码

Fig.3 Rabin-Karp algorithm core code

Rabin-Karp算法的核心思想是利用哈希函数来快速筛选出可能匹配的子串，然后通过字符比较来确认是否真的匹配。这种方法的优势在于它可以在O(1)时间内验证一个潜在的匹配，从而避免了对每个可能的匹配位置进行详尽的比较。算法首先计算模式串的哈希值，然后在目标串中滑动一个与模式串长度相等的窗口，计算窗口内子串的哈希值，并与模式串的哈希值进行比较。

## 4.2 滚动哈希的概念和应用

滚动哈希是Rabin-Karp算法中的关键技术，它允许算法在每次移动窗口时，不需要重新计算子串的哈希值，而是通过简单的数学运算更新哈希值。具体来说，当窗口向右移动一个位置时，新的哈希值可以通过以下公式计算得出：H(new) = H(old) \* base + char(new) - char(old) \* base^(len-1)，其中H(old)是旧子串的哈希值，char(new)是新加入窗口的字符的哈希值，char(old)是离开窗口的字符的哈希值，base是哈希函数的基数，len是子串的长度。这种技术显著提高了算法的处理速度，尤其是在长文本中搜索短模式串时。

## 4.3 算法实现和时间复杂度分析

Rabin-Karp算法的实现涉及以下几个步骤：选择一个合适的哈希函数，计算模式串的哈希值，然后在目标串中滑动窗口，计算每个子串的哈希值，并与模式串的哈希值进行比较。如果哈希值匹配，则进行详细的字符比较以确认是否真的匹配。算法的时间复杂度取决于哈希冲突的数量，最佳情况下为O(N + M)，其中N是目标串的长度，M是模式串的长度。在最坏情况下，如果所有的哈希值都发生了冲突，时间复杂度将退化为O(N \* M)。

## 4.4 对抗故意篡改的鲁棒性讨论

Rabin-Karp算法的鲁棒性主要体现在其对偶然变化的抵抗能力。由于算法依赖于哈希函数，它对单个字符的替换或小的扰动具有一定的容忍度。然而，如果攻击者知道哈希函数的具体细节，并故意制造哈希冲突，算法的鲁棒性可能会受到影响。因此，选择一个强哈希函数是提高Rabin-Karp算法鲁棒性的关键。强哈希函数应该具有低冲突率和均匀分布的特性，以减少误报和漏报的可能性。此外，可以采用多重哈希技术，即使用多个不同的哈希函数来计算同一个子串的哈希值，以进一步提高算法的准确性和鲁棒性。

# 5. Aho-Corasick算法

Aho-Corasick算法是由Alfred V. Aho和Margaret J. Corasick发明的，用于在一段文本中匹配一组模式串的字符串搜索算法。该算法通过构建一个有限状态机（类似于Trie树，并添加失配指针）来实现高效的多模式匹配[3]。

## 5.1 算法描述和原理

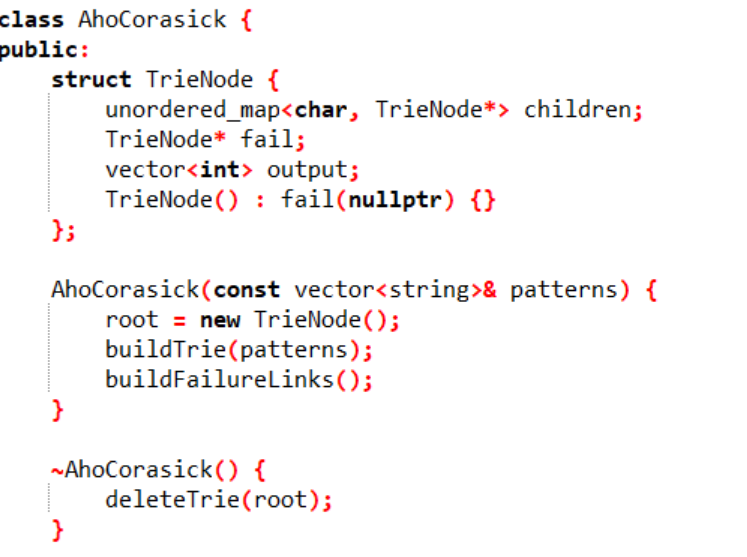


图4 Aho-Corasick 算法数据结构

Fig.4 Aho-Corasick algorithm data structure

Aho-Corasick算法的基本思想是将所有模式串构建成一棵Trie树，并在此基础上增加失败指针（failure pointers），以便在匹配过程中遇到不匹配的情况时能够快速回退到其他可能的匹配路径。这些失败指针允许算法在 Trie树中从一个状态转移到另一个状态，而不需要重新匹配已经匹配过的前缀，从而大大提高了匹配效率。

## 5.2 构建Trie树和失败指针的过程

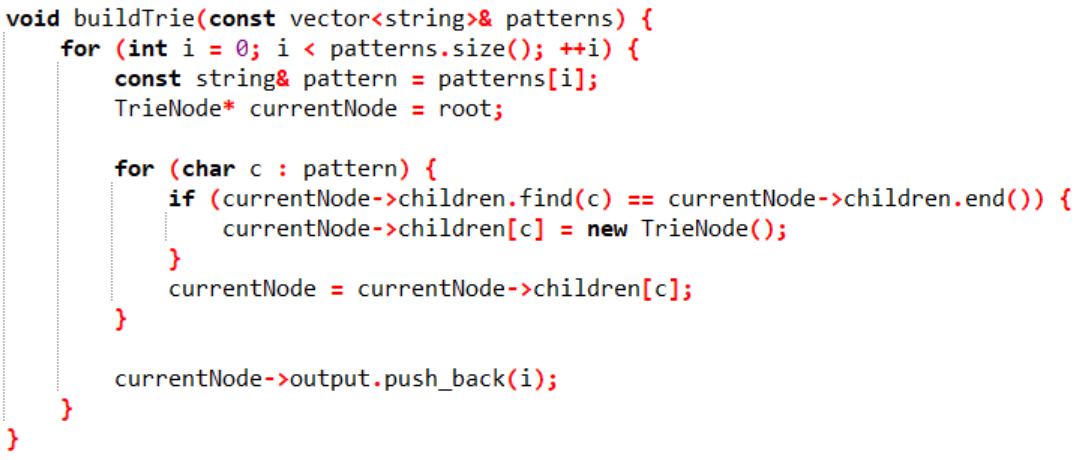


图5 构建Trie树

Fig.5 Build the trie tree

构建Trie树的过程是将每个模式串作为路径插入到树中。对于每个模式串的字符，如果字符在当前节点的子节点中不存在，则创建一个新节点并添加到子节点列表中；如果存在，则移动到相应的子节点。当一个模式串完全插入后，更新输出函数，标记该路径的最后一个节点为模式串的结束。

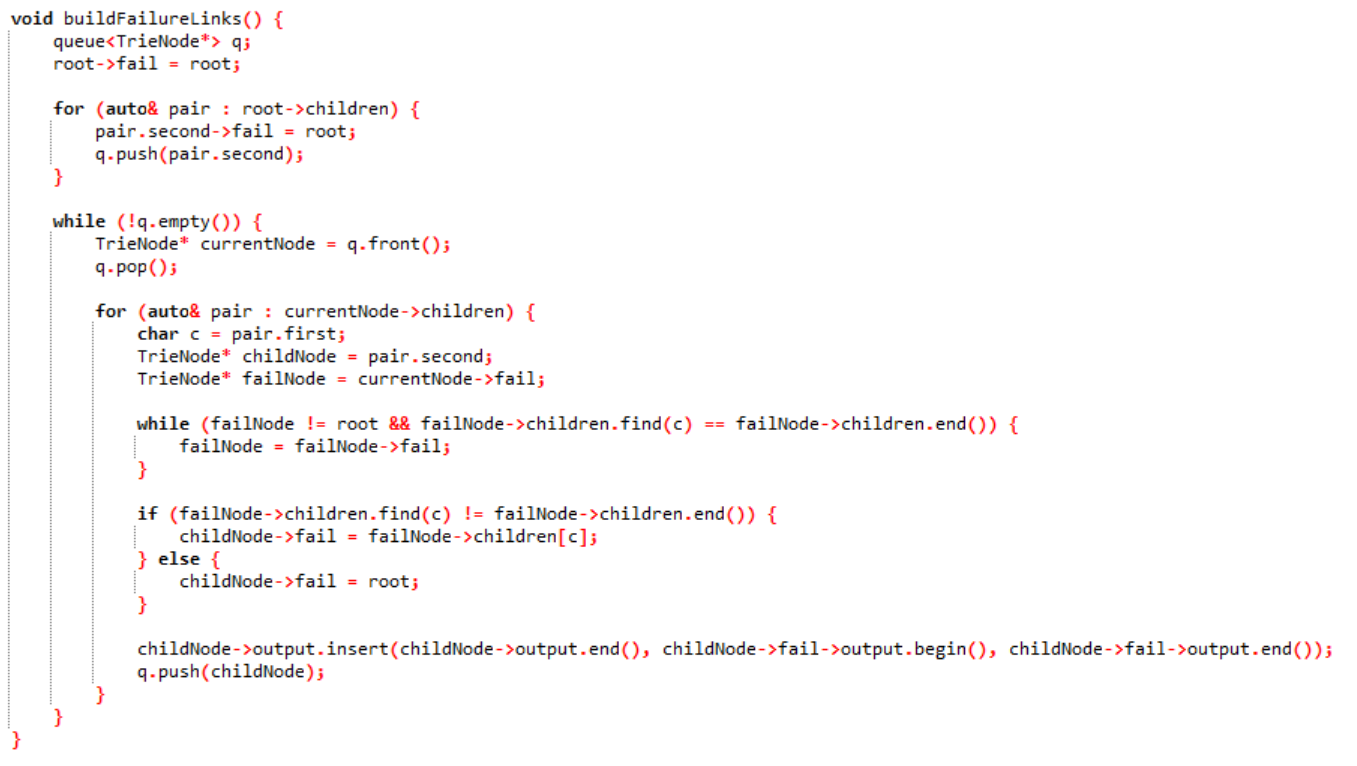


图6 构建失败指针

Fig.6 Build the failure links

失败指针的构建则是在Trie树构建完成后进行的，从Trie树的叶子节点开始，逐层向上回溯。对于每个节点，其失败指针指向在遇到不匹配字符时应该转移到的节点。这个转移节点通常是该节点的父节点的失败指针，或者是具有相同前缀的最近祖先节点的失败指针。如果一个节点的失败指针没有被设置，那么它的失败指针将指向其父节点的失败指针，除非父节点是根节点。如果一个节点的字符与另一个节点的失败指针匹配，那么当前节点的失败指针将指向那个匹配的节点。

## 5.3 算法实现和时间复杂度分析

Aho-Corasick算法的实现可以分为三个主要步骤：构建Trie树、构建失败指针和搜索过程。

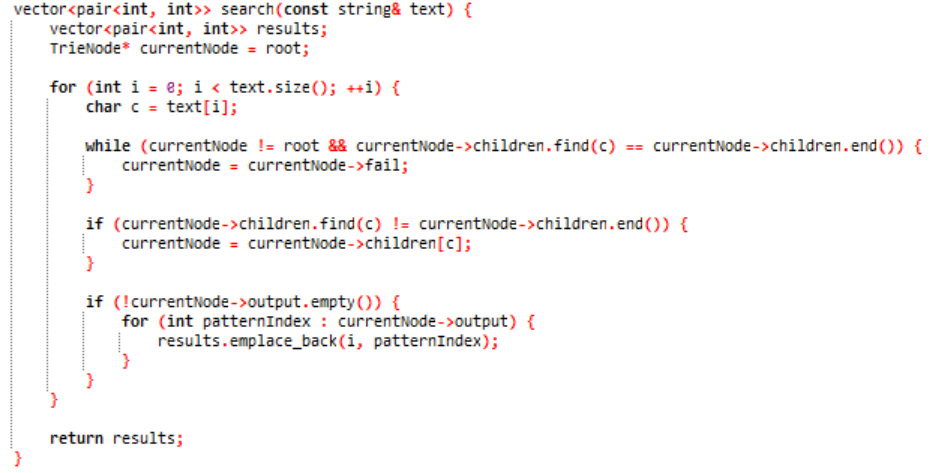


图7 Rabin-Karp 算法核心代码

Fig.7 Rabin-Karp algorithm core code

在搜索过程中，算法从Trie树的根节点开始，逐个处理目标串中的字符。对于目标串中的每个字符，算法首先尝试在当前节点的子节点中找到匹配的字符。如果找到匹配的字符，则移动到对应的子节点；如果没有找到，则通过失败指针跳转到另一个节点。这个过程一直持续，直到目标串被完全处理或者遇到一个模式串的结束节点。每当遇到一个模式串的结束节点时，就记录下该模式串在目标串中的位置。

构建Trie树的时间复杂度为O(M)，其中M是所有模式串的总长度。构建失败指针的时间复杂度也是O(M)。搜索过程的时间复杂度为O(N)，其中N是目标串的长度。因此，整个算法的时间复杂度为O(N + M)，在最坏情况下，如果每次搜索都需要回溯到根节点，则时间复杂度可能达到O(N \* M)。

## 5.4 多模式匹配的应用场景

Aho-Corasick算法在需要在一个文本中搜索多个模式串的场景中特别有用，例如拼写检查、基因序列分析、文本编辑器中的多关键字搜索等。由于其高效的搜索能力，Aho-Corasick算法在处理大量数据时比单一模式匹配算法更具优势。此外，算法的扩展性也使其能够适应不同的应用需求，如通过增加额外的匹配后处理来支持模式串的变体搜索。

# 6. 改进的Aho-Corasick算法

## 6.1 AC-BM算法



图8 AC-BM 算法核心搜索函数伪代码

AC-BM算法是由Aho-Corasick算法和Boyer-Moore算法结合而成的字符串匹配算法[4]。它通过构建一个树状有限状态自动机来处理多个模式串的匹配，并在匹配过程中采用自后向前的方法，同时借用BM算法的坏字符跳转(Bad Character Shift)和好前缀跳转(Good Prefix Shift)技术来优化匹配过程。这种结合使得AC-BM算法在处理多个模式串匹配时具有较高的效率。

Aho-Corasick算法是一种经典的多模式匹配算法，它通过构建一个Trie树来组织模式串，并利用失败指针来加速匹配过程。而Boyer-Moore算法则是一种高效的单模式匹配算法，它通过坏字符规则和好后缀规则来减少不必要的比较。AC-BM算法结合了这两种算法的优点，通过在Trie树中应用BM算法的跳转规则，提高了多模式匹配的效率。

在AC-BM算法中，每个模式串都被插入到Trie树中，形成一个前缀树。然后，算法为每个节点计算坏字符跳转和好前缀跳转参数。在匹配过程中，算法从文本的末尾开始，逐个字符向前比较。一旦发现不匹配，算法会根据坏字符跳转或好前缀跳转参数来调整匹配位置，从而避免了从头开始的重新匹配。

## 6.2 AC-BM算法的改进点和优化策略

改进的AC-BM算法在以下几个方面进行了优化：

### 取消好前缀跳转

传统的Boyer-Moore算法中包含好前缀跳转，但在多模式匹配的上下文中，好前缀跳转并不总是有效。改进的AC-BM算法取消了这一跳转，简化了算法的逻辑，减少了计算复杂度。

### 优化坏字符跳转

改进的AC-BM算法对坏字符跳转进行了优化。在多模式匹配中，坏字符跳转可以帮助算法快速跳过不可能匹配的位置，从而提高匹配效率。

### 大小写不敏感匹配

改进的AC-BM算法不再需要在扫描正文前将正文转换为小写，实现了对大小写不敏感的匹配。这使得算法能够适应更多的应用场景，特别是在文本内容中包含大小写混合的情况下。

### 计算跳转参数

在构建Trie树的过程中，算法为每个节点计算坏字符跳转参数。这些参数指示在发生不匹配时，算法应该跳转到哪个位置。计算这些参数的过程涉及对所有模式串的分析，以确定每个字符在模式串中的位置。改进的AC-BM算法构建Trie树的过程与传统的Aho-Corasick算法类似，但更注重于前缀的共享。这有助于减少树的大小，从而减少匹配过程中的比较次数。

## 6.3 AC-BM算法的实现和性能对比

在实现AC-BM算法时，我们首先需要构建一个Trie树，然后将每个模式串插入到树中。这个过程涉及到为每个节点计算坏字符跳转参数。

在性能对比方面，改进的AC-BM算法在处理大规模数据集时，特别是在模式串数量较多或文本内容较长的情况下，表现出比原始AC-BM算法更快的匹配速度。这是因为改进的算法减少了不必要的比较次数，并且能够更快地跳过不可能匹配的位置。

## 6.4 AC-BM算法的优化效果的实验举例

考虑文本串T="HERE IS A SIMPLE"和模式串P="SIMPLE"的匹配过程。

传统AC算法匹配过程：

|  |
| --- |
| 位置1，从文本串起始位置开始：  比较T[1]='H'与P[1]='S'，不匹配  移动到下一位置  位置2至位置4类似过程，直到：  位置5（IS中的S）：  比较T[5]='S'与P[1]='S'，匹配  比较T[6]='I'与P[2]='I'，不匹配  移动到下一位置  位置6至位置10类似过程，直到：  位置11（SIMPLE中的S）：  比较T[11]='S'与P[1]='S'，匹配  比较T[12]='I'与P[2]='I'，匹配  比较T[13]='M'与P[3]='M'，匹配  比较T[14]='P'与P[4]='P'，匹配  比较T[15]='L'与P[5]='L'，匹配  比较T[16]='E'与P[6]='E'，匹配  找到完整匹配 |

AC-BM算法匹配过程：

|  |
| --- |
| 首次对齐，从模式串最后一个字符开始比较：  比较T[6]='S'与P[6]='E'，不匹配  'S'在模式串中出现在首位置，使用坏字符跳转规则移动5位  第二次对齐（位置6）：  比较T[11]='S'与P[6]='E'，不匹配  同样使用坏字符跳转规则移动5位  第三次对齐（位置11）：  比较T[16]='E'与P[6]='E'，匹配  向左比较T[15]='L'与P[5]='L'，匹配  向左比较T[14]='P'与P[4]='P'，匹配  向左比较T[13]='M'与P[3]='M'，匹配  向左比较T[12]='I'与P[2]='I'，匹配  向左比较T[11]='S'与P[1]='S'，匹配  找到完整匹配 |

AC算法进行了(4×1 + 2 + 6)=12次字符比较：

4次初始失配比较（H,E,R,E）

IS处的S匹配后与I的比较（2次）

最终SIMPLE的完整匹配（6次）

AC-BM算法仍然是(2×1 + 6)=8次字符比较：

2次失配比较

1次完整匹配（6次比较）

实验数据表明，即使在这个简单的匹配场景中，AC-BM算法也实现了33%的比较次数减少。这种优化效果源于算法的两个关键改进：首先，从右向左的比较策略能够更快发现失配情况；其次，坏字符跳转机制允许算法跳过必然失配的位置，避免了大量无效比较。这种优化在处理大规模文本和多模式匹配时将产生更为显著的效果，特别是当待匹配文本中包含较多不在模式串中出现的字符时。

通过这种双重跳转机制和混合策略的优化设计，AC-BM算法在保持AC算法多模式匹配优势的同时，还能充分利用BM算法的跳转特性。这种改进使得算法特别适用于需要同时匹配多个模式串的应用场景，如网络入侵检测系统、病毒特征码扫描等领域。在这些应用中，AC-BM算法不仅能够保证匹配的准确性，还能提供显著的性能提升，为实际应用提供了更好的解决方案。

# 7 结语

本文对几种经典的字符串匹配算法进行了全面的综述与优化研究，探讨了暴力匹配算法、Boyer-Moore算法、Rabin-Karp算法、Aho-Corasick算法及其改进版本在不同应用场景下的适用性与效率。通过对各算法原理与实现的分析，我们发现，尽管暴力匹配算法因其简单性而被广泛理解，但在处理大规模数据时，其性能显著不如其他更为高效的算法。

Boyer-Moore算法凭借其独特的坏字符和好后缀规则，在实际应用中表现出色，尤其适用于长模式串的匹配。Rabin-Karp算法利用滚动哈希技术，在多模式匹配中展现出高效性，尽管其对哈希冲突的处理仍需加强。Aho-Corasick算法则在多模式搜索中展现了强大的优势，能够在单次遍历中有效匹配多个模式串。

此外，本文详细讨论了Aho-Corasick算法的改进版本——AC-BM算法，通过结合Boyer-Moore算法的跳转策略，显著提高了多模式匹配的效率，尤其在处理大规模文本时表现更加优越。这种新方法不仅提升了算法的性能，也为实际应用提供了更加灵活有效的解决方案。

字符串匹配算法在信息检索、文本处理和网络安全等领域中扮演着至关重要的角色。随着数据量的不断增加，对字符串匹配算法的性能要求也日益提高。因此，未来的研究可以继续探索更高效的算法和优化策略，以满足不断变化的应用需求。通过持续的创新和改进，字符串匹配技术必将迎来更广阔的发展空间。

**参考文献**

[1]巫喜红,凌捷.BM模式匹配算法剖析[J].计算机工程与设计,2007(01):29-31.DOI:10.16208/j.issn1000-7024.2007.01.010.

[2] Plaxton G. String Matching: Rabin-Karp Algorithm[J/OL]. University of Texas at Austin, Department of Computer Science, (2004-04-01).

h[ttps://www.cs.utexas.edu/~plaxton/c/337/04s/slides/StringMatching-1.pdf](https://www.cs.utexas.edu/~plaxton/c/337/04s/slides/StringMatching-1.pdf).

[3] Stanford University. Aho-Corasick Automata: String Data Structures [J/OL]. (2004-02).

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/02/Small02.pdf>.

[4] Jason C C, Staniford S, McAlemey J. Towards faster string for intrusion detection or exceeding the sped of snert[J/OL].

http://www.silicondefense.com/sotfware/acbm/speed\_of\_snort\_03\_16\_2001.padf, 2003-10-06