基于跳跃式匹配的藏文多模式匹配算法

周磊超[1] 彭 展[1]
ZHOU Leichao PENG Zhan

摘 要 在计算机科学的研究领域中，字符串匹配算法是基础性问题的关键一环，依据查找过程中所涉及模式串的数量差异，可细分为单模式匹配算法与多模式匹配算法这两大类别。其中，AC 算法作为多模式匹配算法范畴内极具代表性且应用极为广泛的经典算法。文章通过分析 AC 算法及其相关改进算法，并结合藏文结构的特点，提出了一种基于跳跃式匹配的藏文多模式匹配算法—AC_BM2T 算法。该算法基于当前匹配窗口末尾的 2 个字符，设计了一种新的模式树移动规则，使得模式树可以实现跳跃式匹配并且跳跃距离相对较大，以此提高算法执行效率。实验结果表明，在处理藏文时，AC_BM2T 算法的执行时间是其他改进算法的 50%~60%；在模式串数量较少时，AC_BM2T 算法的执行时间是 AC 算法的 25%~90%。

关键词 多模式匹配算法；AC 算法；藏文结构；模式树；跳跃

doi：10.3969/j.issn.1672-9528.2025.01.003

0 引言

随着信息化技术的发展，互联网中的文本数据以指数级的趋势增长，如何快速从海量的文本数据中检索到想要的数据至关重要 [1]。字符串匹配算法即从文本中找到指定的字符串（将想要找到的字符串称为模式串）。经过多年研究与发展，字符串匹配算法已被广泛应用于敏感词检测、生物学、网络安全和人工智能等领域。

字符串匹配算法根据查找模式串的数量分为单模式匹配算法和多模式匹配算法 [2]。目前，模式匹配算法研究相对较多，但是为特定应用场景设计最合适的模式匹配算法仍是目前尚未解决的问题 [3]。多模式匹配算法通过扫描一次文本串即可找出多个模式串的出现位置。AC 算法 [4] 是实际表现优秀且应用十分广泛的多模式匹配算法 [5]。值得一提的是，AC 算法的改进算法有多种，本文分析了 AC_BM 算法、AC_BMH 算法和 AC_Sunday 算法。在此基础上，提出了一种高效的藏文多模式匹配算法。

1 藏文及其编码分析

藏文是一种拼音文字，共有 30 个辅音字母和 5 个元音字母 [6]。字形结构均以一个辅音字母为核心，其余字母以此为基础上下和左右拼写，组成一个完整的字。

藏文的字形结构如图 1 所示。藏文字形结构由 1~6 个辅音字母构成，元音字符在辅音结构的上方或下方。其中，核心字母为基字。音节是藏文的基本单位，不同的藏文音节使用音节符‘·’隔开，不同的藏文句子使用单垂符‘|’隔开 [7]。

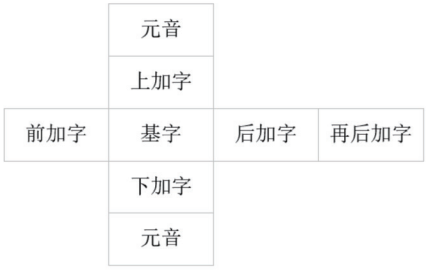

图 1 藏文字形结构图

目前，国际上主流的藏文编码方案为小字符集编码 [8]。小字符集编码是基于 ISO/IEC 10646 标准的编码方案，将藏文视作拼音文字，依据其结构特征，把一个藏文音节拆分成多个不同的部件，对每一个部件进行单独编码 [9]。计算机在处理藏文时，会将藏文拆分成多个部件，例如 ཀྲ 会拆分成 ཀ、ྲ 和 ་3 个组成部分。

2 AC 及其改进算法分析

2.1 AC 算法

AC 算法基于有限状态自动机的原理予以实现，在匹配前对模式串集合进行预处理。预处理过程中，根据模式串集合构造有限状态自动机，计算 goto 函数、failure 函数和 output 函数。在匹配过程中，根据预处理过程计算的 3 个函数，扫描一次文本串就可以找出所有模式串的出现位置。

1. 西藏民族大学信息工程学院 陕西咸阳 712000

2.2 AC 改进算法分析

AC 算法在匹配时，需要对文本串中的所有字符进行匹配，不可进行跳跃式匹配。针对此问题，后续有多种 AC 改进算法，可以实现跳跃式匹配，下面介绍几种 AC 算法的改进算法。

AC_BM 算法 [10] 是 AC 算法的一种改进算法。该算法在 AC 算法的基础上，结合 BM 算法 [11] 的移动规则，实现跳跃式匹配。BM 算法的移动规则基于坏字符规则和好后缀规则实现，取其中最大值作为失配时移动距离。坏字符规则是在匹配时，当某个字符不匹配时，称该字符为坏字符，根据该字符在模式串中是否出现决定模式串的移动距离；好后缀规则是利用已经匹配过的信息决定模式串的移动距离。和 AC 算法不同的是，AC_BM 算法在匹配前将模式串集合构建为 Trie 树 [12]（模式树），并且 AC_BM 算法只使用 goto 函数和 output 函数，在匹配时通过 BM 算法的移动规则进行跳跃。

文献 [13] 将 AC 算法与 BMH 算法结合，提出了 AC_BMH 算法。BMH 算法是 BM 算法的改进算法，由于好后缀规则相对复杂，该算法省去了好后缀规则，使得算法相比 BM 算法更加简单，并且效率比 BM 算法更高。同时，AC_BMH 算法相比 AC_BM 算法也更快。

文献 [14] 将 AC 算法与 Sunday 算法相结合，提出了 AC_Sunday 算法。AC_Sunday 算法借助 Sunday 算法的跳跃规则，在发生失配时，模式树根据文本串与模式串中最短字符串长度相对应的下一字符是否在模式串中出现进行移动。同时，AC_Sunday 算法使用了压缩存储，对算法的空间进行了优化。

上述改进算法都是基于跳跃式规则进行匹配的多模式匹配算法，算法的效率受到字符出现频率和字符出现位置的影响。在模式串数量较少时，单个字符的出现频率并不高，上述算法在效率上优于 AC 算法。但在模式串数量较多时，单个字符出现频率较高且出现在各个位置的概率也会提高，严重影响了这些算法模式树的移动距离，使得算法的效率较低甚至不如 AC 算法。

3 AC_BM2T 算法

针对上述 AC 改进算法所存在的问题，本文提出了 AC_BM2T 算法，该算法结合藏文结构的特点，设计了一种新的模式树的移动规则。模式树的移动规则基于当前匹配窗口末尾两个字符组成的字符块实现，在不发生漏检的情况下，使模式树尽可能多地移动，从而减少不必要的字符比较次数，同时使模式树的最大移动距离可达到最短模式串长度加 1，进而提高算法的执行效率。

3.1 算法设计思想

AC_BM2T 算法使用 1 个函数确定发生失配时模式树的移动规则。令文本串 *T*=དེ་རིང་ང་ཚོས་，模式串集合 *P*={རིང་、དགའ་}，*T* 和 *P* 的模式树以及模式树的移动方向如图 2 所示。模式树从右向左匹配，在发生失配时，先将模式树右移一个单位，再根据当前匹配窗口末两个字符组成的字符块在 *P* 中是否出现决定模式树的移动距离。由于 goto 函数的执行速度较慢，模式树在进行下一轮匹配前，继续根据移动函数检查模式树是否可以继续移动，直到模式树移动距离为 0 后再开始下一轮匹配，以此减少不必要的 goto 函数的执行，提高算法效率。

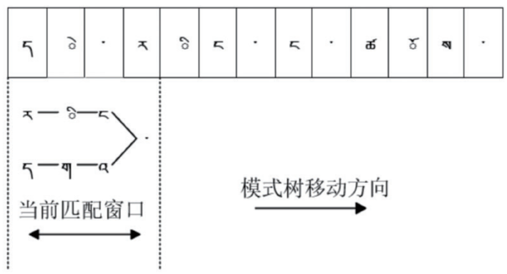

图 2 模式树及其移动方向

3.2 算法步骤

AC_BM2T 算法的实现分为预处理和匹配这两个过程。

（1）预处理过程

设待匹配模式串集合为 *P*，P_i 为 *P* 中的一个模式串，文本串为 *T*，*T*[*k*] 表示当前匹配窗口的末尾字符，*P* 中最短模式串的长度为 MinLength。

AC_BM2T 算法中 goto 函数和 output 函数的计算规则与 AC 算法的相同，都是先构造有限状态自动机，再根据有限状态自动机计算 goto 函数和 output 函数。但 AC_BM2T 算法和 AC 算法构造有限自动机的方式不同，AC 算法是从前向后构造的。

Skip 函数记录了发生失配时模式树的移动距离。Skip 函数的计算 3 个步骤为：① 将 Skip 数组所有值初始化为 MinLength-1。② 令 *L* 表示 P_i 中字符 *T*[*k*] 到 P_i 最后一个字符的距离，遍历 *P* 中所有的模式串，根据公式 Skip[*T*[*k*-1]][[*k*]]=min(*L*-1，Skip[*T*[*k*-1]][[*k*]]) 更新 Skip 数组的值。③ 遍历 Skip 数组，当 Skip[*i*][*j*]=MinLength-1 时（*i* 和 *j* 表示藏文字符），根据下方的公式更新 Skip 数组的值，计算出最终的 Skip 数组。

$$\text{Skip}[i][j]=\begin{cases}\text{MinLength}+1 & i\notin\{\text{་、།}\}\text{且}j\notin\{\text{་、།}\}\\ \text{MinLength} & j\in\{\text{་、།}\}\\ \text{MinLength}-1 & \text{其它}\end{cases}$$

（2）匹配过程

匹配时根据预处理过程计算出的 goto 函数、output 函数和 Skip 函数进行匹配，最终输出匹配成功的模式串的位置，匹配过程的核心代码为：

AC_BM2T 算法

```
i ← MinLen − 1            // 初始化
state ← 0
```

```
j ← i
while i < n
    if  j = 0
        i ← i + 1
        j ← i
        // 根据当前状态和字符获取下一个状态
        state ← goto ( state , T [ j ] )
    if  state > 0
        if  output ( state ) ≠ NULL
            print ( j ) // 成功匹配
            j ← j − 1
        else
            i ← i + 1
    while skip [ T [ i – 1 ] T [ i ] ] ≠ 0
        i ← i + skip [ T [i − 1 ] T [ i ] ]
    if  i > n − 1
        return -1
return -1 // 匹配结束
```

3.3 实例分析

令文本串 *T*=དེ་རིང་ང་ཚོས་，模式串集合 *P*={རིང་、དགའ་}。图 3 是根据 *P* 构建的有限状态自动机，图 3 中加粗的部分表示 output 函数的输出值，例如 output(4) = རིང་。表 1 是根据 *P* 计算的 Skip 数组。

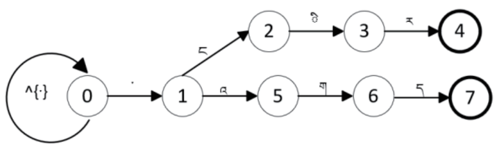

图 3 AC_BM2T 算法构建的有限状态自动机

表 1 Skip 数组

i	*j*	Skip[*i*][*j*]
ང	་	0
ི	ང	1
ར	ི	2
འ	་	0
ག	འ	1
ད	ག	2
都不为为་和།		5
其他	为་或།	4
其他		3

图 4 是 AC_BM2T 算法中模式树的移动过程。模式树共经历了 3 轮匹配和 2 轮移动。第 1 行是模式树的初始位置，最短模式串的长度为 4，模式树的根节点对齐文本串的第 4 个位置。算法的具体匹配过程为：模式树进行第 1 轮匹配，根据 goto 函数从右向左比较，goto(0,ར)=fail，匹配失败；模式树进行第 1 轮移动，先向右移动一个单位，再根据 Skip 数组移动，移动到第 2 行的位置；模式树进行第 2 轮匹配，goto(0, ་)=1，goto(1, ང)=2，goto(2, ི)=3，goto(3, ར)=4，output(4)=དགའ་，最终输出 དགའ་ 的值；模式树进行第 2 轮移动，移动到第 3 行的位置；模式树进行第 3 轮匹配，goto(0, ་)=1，goto(1, ས)=fail，最终匹配结束。goto 函数和 output 函数可参考图 3。

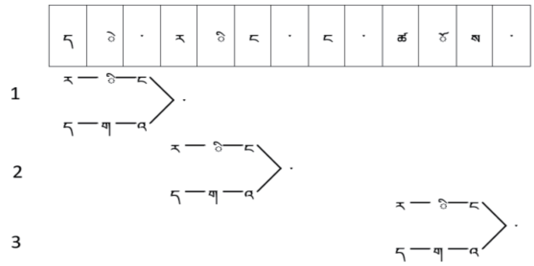

图 4 模式树的移动

4 实验分析

为了评估 AC_BM2T 算法的性能，本文选取 AC、AC_BM 和 AC_Sunday 算法进行对比。通过改变模式串数量或模式串长度来评估这几种算法在不同情况下的执行时间。实验测试环境为 Intel(R) Core(TM) i7-10700 CPU 2.90 GHz，内存为 16 GB，所有算法均使用 Python 语言实现，Python 版本为 3.9.13。实验中所用文本数据为西藏语言文字网、人民网藏文版网页和中国西藏网等网站上提取的藏文文本。模式串数据从藏文网页中随机抽取长度不同的藏文词语。

实验 1 中，文本串的大小约为 16 MB，模式串数量为 50 个，通过改变模式串长度，测试几种算法的执行时间。模式串长度的基本单位为音节，藏文中，一个音节相当于一个字。

图 5 和表 2 是实验 1 的运行结果。AC 算法的执行时间基本不受模式串长度的影响。其他 3 种算法的执行时间受到模式串长度的影响较大。由于这 3 种算法都是基于跳跃式规则进行匹配的，而模式串长度的增加会影响模式树的移动距离，模式树的移动距离又直接影响算法的运行效率。模式串长度越长，AC_BM2T 算法运行速度越快，但当模式串到达一定长度时，也会提高各个字符出现的频率，对算法执行时间产生反作用。

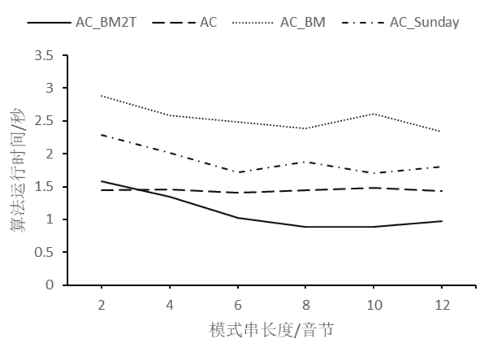

图 5 改变模式串长时算法的执行时间

表 2 改变模式串长时算法的执行时间

	AC_BM2T/s	AC/s	AC_BM/s	AC_Sunday/s
2	1.58	1.44	2.88	2.29
4	1.35	1.46	2.59	2.01
6	1.03	1.41	2.48	1.72
8	0.89	1.45	2.39	1.88
10	0.89	1.48	2.61	1.71
12	0.97	1.43	2.34	1.81

实验 2 中，文本串的大小约为 16 MB，固定模式串长度为 8 音节，改变模式串数量，测试几种算法的执行时间。

图 6 和表 3 是实验 2 的运行结果。AC 算法的执行时间基本不受模式串数量的影响。其他 3 种算法的执行时间受模式串数量的影响较大，由于模式串数量的增加，会提高原模式串集合中未出现过字符的出现概率以及字符在各个位置出现的概率，导致模式树的移动距离降低，进而影响算法的执行效率。AC_BM2T 算法的执行时间和模式串数量呈负相关。

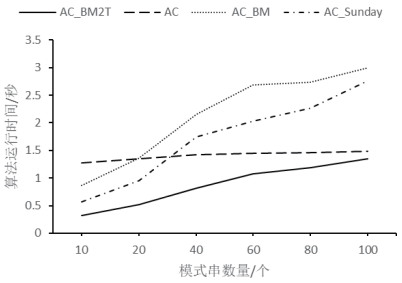

图 6 改变模式串数量时算法的执行时间

表 3 改变模式串数量时算法的执行时间

	AC_BM2T/s	AC/s	AC_BM/s	AC_Sunday/s
10	0.32	1.28	0.87	0.57
20	0.52	1.35	1.36	0.95
40	0.81	1.42	2.15	1.75
60	1.07	1.45	2.68	2.03
80	1.19	1.46	2.73	2.27
100	1.35	1.48	2.99	2.76

由于 AC_BM2T 算法的模式树移动规则是基于两个字符实现的，两个字符相比单个字符出现的概率更小，模式树有更大的移动距离，故 AC_BM2T 算法快于 AC_BM 算法和 AC_Sunday 算法。

综上所述，AC_BM2T 算法的执行时间受模式串数量影响较大，受模式串长度的影响相对较小。在处理藏文时，AC_BM2T 算法的执行时间是其他 AC 改进算法的 50%~60%；在模式串数量较少时，AC_BM2T 算法的执行时间是 AC 算法的 25%~90%。因此，AC_BM2T 相比其他算法，适用于处理藏文时模式串数量较少的应用场景中。

5 结语

本文介绍了 AC 算法以及 AC 的改进算法，并提出了 AC_BM2T 算法。该算法结合藏文结构的特点设计了新的模式树的移动规则。实验结果表明，在处理藏文时，该算法具有较高的效率。未来可进一步对该算法模式树的移动规则进行优化或改进。

参考文献：

[1] 陈永杰，吾守尔·斯拉木，于清 . 一种基于 Aho-Corasick 算法改进的多模式匹配算法 [J]. 现代电子技术，2019, 42(4): 89-93.

[2] HAKAK S L, KAMSIN A, SHIVAKUMARA P, et al. Exact string matching algorithms: survey, issues, and future research directions[J]. IEEE access, 2019,7: 69614-69637.

[3] 张学通，彭展 . 基于藏文音节结构的单模式匹配算法 [J]. 计算机仿真，2024, 41(8): 374-378.

[4] AHO A V, CORASICK M J. Efficient string matching: an aid to bibliographic search[J]. Communications of the ACM, 1975, 18(6): 333-340.

[5] 姜海洋，李雪菲，杨晔 . 基于距离比较的 AC 自动机并行匹配算法 [J]. 电子与信息学报，2022, 44(2):581-590.

[6] 尼玛扎西，完么扎西 . 藏语自然语言处理基本理论和方法 [M]. 北京：科学出版社，2020.

[7] 陈小莹 . 现代藏文音节结构分析研究 [J]. 智能计算机与应用，2019, 9(2):89-90+95.

[8] 陈小莹，艾金勇 . 基于小字符集藏文拉丁转写系统的设计与实现 [J]. 中文信息学报，2016,30(3):74-78.

[9] 春燕，曲珍 . 藏文文本编码识别方法研究 [J]. 计算机工程与应用，2013,49(1):141-144.

[10] COMMENTZ-WALTER B. A string matching algorithm fast on the average[C]//Proceedings of the 6th Colloquium.Berlin: Springer, 1979:762-772.

[11]BOYER R S，MOORE J S. A fast string searching algorithm[J]. Communications of the ACM, 1976, 20(10):762-772.

[12] 刘春晖，黄宇，宋琦 . 一种改进的 AC 多模式匹配算法 [J]. 计算机工程，2015,41(10):280-285.

[13] 陆琳琳，田野 . 基于确定有限状态自动机的改进多模式匹配算法研究 [J]. 计算机应用与软件，2013,30(7):321-323+330.

[14] 董世博，李训根，殷珍珍 . 一种改进的字符串多模式匹配算法 [J]. 计算机工程与应用，2013,49(8):133-137.

【作者简介】

周磊超（1999—），男，河南平顶山人，硕士研究生，研究方向：文本处理，email: 1417350360@qq.com。

彭展（1986—），男，陕西西安人，博士研究生，讲师，研究方向：文本处理、网络安全、网络舆情。

（收稿日期：2024-10-09）