北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

数据结构研学报告

AC 自动机的原理及应用

学　　号　1120231313

姓　　名　叶子宁

专　　业　软件工程

目　　录

1 引言

AC 自动机（Aho-Corasick Automaton）是一种用于多模式字符串匹配的高效算法结构，主要用于快速查找多个模式串在目标字符串中的出现位置。相比于单模式串匹配的 KMP 算法，AC 自动机在面对多模式串时表现出极高的效率，广泛应用于文本检索、词语过滤、DNA 序列分析等领域。

本文将详细介绍 AC 自动机的原理，并探讨其在实际场景中的应用。

2 AC 自动机的原理

2.1 有限自动机

这里给出非确定性有限自动机（Non-deterministic ·Finite Automaton, NFA）的形式化定义，非确定性有限自动机是 N 是一个五元组 $(Q, \Sigma, \delta, q_0, F)$，其中：

- Q是有限状态集合；
- Σ是有限输入字母表；
- $\delta : Q \times \Sigma_\varepsilon \to P(Q)$是状态转移函数；
- $q_0 \in Q$是初始状态；
- $F \subseteq Q$是接受状态集合。

有限自动机可以接受一个输入串，从初始状态开始，根据状态转移函数逐步转移到下一个状态，最终判断是否到达接受状态。

AC 自动机就是一个以多个模式串为语言的有限自动机，每个模式串对应一个接受状态。

AC 自动机的构建基于 **字典树** 和 **KMP 算法**，前者是确定 DFA 的基本结构以及成功匹配时的状态转移，后者用于处理失败匹配时的状态回退。

2.2 字典树

2.2.1 字典树的定义

字典树（Trie）是一种用于高效存储和检索字符串的树形数据结构，它在字符串的前缀匹配和查询方面具有显著优势。

本质上，字典树也是一种有限自动机。当遇到属于模式串的字符时，自动机会继续向下进行状态转移；如果遇到非模式串的字符，则会进入拒绝状态。

例如给出字符串集合

表 2-1 模式串集合

string_name
Reina
Rinai
Rin
Rena

对应的字典树如 图 1 所示：

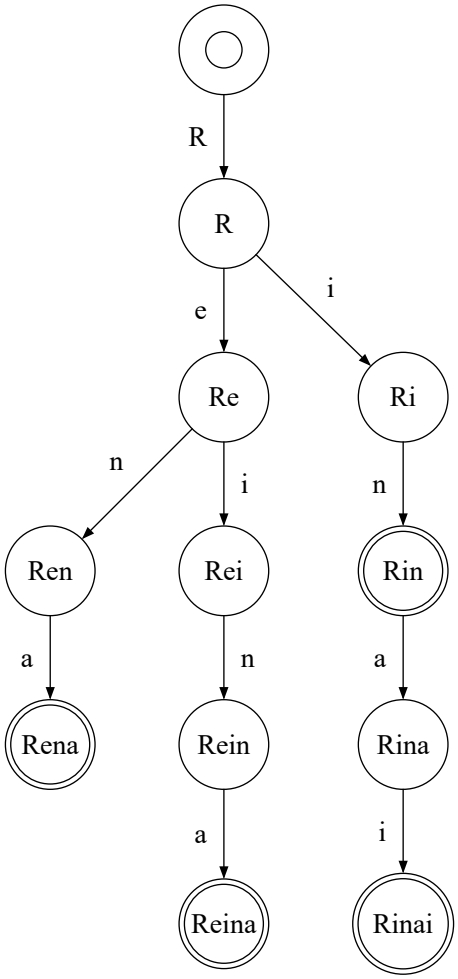

图 2-1　模式串集合的字典树

2.2.2　字典树的基本应用

每一个接受状态对应一个模式串，每一条根节点到其他节点的路径对应模式串的一个前缀，因为字典树的形态是固定的，所以可以通过遍历字典树的方式，判断目标串是否：

1. 是模式串的前缀；
2. 包含模式串。

如果采用二维数组存储转移函数，那么字典树的构建时间复杂度为 $O(\sum m_i)$，其中 m_i 为第 i 个模式串的长度，空间复杂度为 $O(\sum m_i * |\Sigma|)$，其中 $|\Sigma|$ 为字母表的大小。

查询目标串是否是模式串的前缀的时间复杂度为 $O(n)$，其中 n 为目标串的长度。

若采用红黑树（`std::map`）等数据结构存储转移函数，空间复杂度可降至 $O(\sum m_i)$，但是查询和构建的时间复杂度要乘上 $\log|\Sigma|$ 的因子。

由于字典树可以轻松的查找模式串的前缀，在算法竞赛中常使用 $0-1$ 字典树来维护最大异或值。（根据输入的数据，每次选取与当前异或值最大的路径）[1]

这里也能看到字典树的一个缺点，即只能判断与模式串的前缀关系匹配，无法判断其**子串**与模式串的关系。

为了解决这个问题，AC 自动机引入了 KMP 算法的思想，构建了一个更加强大的多模式匹配算法。

2.3　KMP 算法

KMP 算法（Knuth-Morris-Pratt Algorithm）是由 Kruth、Morris 和 Pratt 三位计算机科学家于 1977 年提出的一种高效的字符串匹配算法。[2] 用于判断一个字符串是否包含另一个字符串作为子串，以及在匹配时出现的位置、次数等信息。

KMP 算法的朴素思想就是在匹配失败时，利用已经匹配的信息，尽量减少重复匹配的次数。

而 KMP 在这方面的实现形式是**前缀函数**。

2.3.1 前缀函数的定义

对于字符串 s，其前缀函数定义为

$$\pi(i)=\begin{cases}0, i=0\\ \max\{k \mid k<i, s[0...k-1]=s[i-(k-1)...i]\}, 0<i<n\end{cases} \tag{2-1}$$

其中 $s[l...r]$ 表示字符串 s 的子串 $s[l], s[l+1], ..., s[r]$。

即 $s[0...k-1]$ 是 s 的前缀 $s[0...i]$ 的前 k 个字符，$s[i-(k-1)...i]$ 是 s 的后缀 $s[0...i]$ 的后 k 个字符。

也就是对于任意一个前缀，它前缀函数的值为**最长的**能使得前缀与后缀相等的长度。

例如对于字符串 `ababaca`，其前缀函数为

表 2-2　字符串 `ababaca` 的前缀函数

$s[0...i]$	a	ab	aba	abab	ababa	ababac	ababaca
$\pi(i)$	0	0	1	2	3	0	1

2.3.2 前缀函数的性质

1. $\pi(i+1) \le \pi(i)+1$，当且仅当 $s[\pi(i)]=s[i]$ 时等号成立；即前缀函数在相邻位置转移时，若增加，最多只能增加 1。
2. 若 $s[0...k-1]=s[i-(k-1)...i]$ 且 $\pi(k)>0$，则 $s[0...\pi(k)-1]=s[i-(\pi(k)-1)...i]$。也就是说相等的前后缀，其前缀函数对应的相等的前后缀也是原串中的前后缀。
   - 如 ababa 中的前缀 aba 的前后缀 a 同样也是 ababa 的前后缀。

根据这两个性质，我们可以快速的构造出一个字符串的前缀函数。

2.3.3 前缀函数的构造

算法的流程如下：

1. 初始化 $\pi(0)=0$；
2. 找到最大的 k 使得 $s[0...k-1]=s[i-(k-1)...i]$；
   - 即对 k 根据 $\pi(k-1)$ 递减，直到找到满足条件的 k（性质 2）；这样 k 是依次递减且不遗漏的。
3. 若 $s[k]=s[i]$，则 $\pi(i)=k+1$；否则 $\pi(i)=0$。

伪代码如 算法 1 所示：

```
1  function prefix_function(s : string) returns array_int
2      π[0] ← 0
3      for i ← 1 to n − 1 do
4          k ← π[i − 1]
5          while k > 0 and s[k] ≠ s[i] do
6              k ← π[k − 1]
7          end
8          if s[k] = s[i] then
9              k ← k + 1
10         end
```

```
11          |   π[i] ← k
12          end
13          return π
14      end
```

算法 2-1　前缀函数的构造算法

2.3.3.1　时间复杂度分析

构造前缀函数时的时间复杂度主要由指针变化引起，即以下两个部分：

1. $k \leftarrow \pi[k-1]$
2. $k \leftarrow k+1$

由于 k 恒不为负，而 1 操作最少使得 k 减少 1，因此 1 操作的次数不会超过 2 操作的次数。

而 2 操作的次数不会超过 n（每次匹配成功后 k 会增加 1，而匹配次数最多为 n），因此构造前缀函数的时间复杂度为 $O(n)$。

2.3.4　KMP 算法的实现

直接采用前缀函数构造的思路，我们可以实现 KMP 算法。

将目标串称为 s，模式串称为 p，我们可以通过构造 $p+s$ 的前缀函数，来判断 s 中是否包含 p。

当 $p+s$ 的某个前缀函数值等于 $|p|$ 时，说明 s 中存在 p。

当然可能出现前缀函数超过 $|p|$ 的情况，因此我们需要在连接字符串时插入一个特殊字符，以确保前缀函数的值不会超过 $|p|$。

伪代码如 算法 2 所示：

```
1   function kmp_search(s : string, p : string) returns array_int
2       n ← |s|, m ← |p|
3       π ← prefix_function(p + '#' + s)
4       ans ← []
5       for i ← 0 to n − 1 do
6           if π[m + i + 1] = m then
7               ans.push(i - m + 1)
8           end
9       end
10      return ans
11  end
```

,

算法 2-2　KMP 算法的实现

2.3.4.1　时间复杂度分析

KMP 字符串匹配的时间复杂度即为构造 $p+s$ 的前缀函数的时间复杂度，为 $O(n+m)$。

2.3.5　KMP 算法的思想总结

通过前缀函数的性质以及存下来已知的信息，我们只需要在**失败匹配**的时候找到前缀匹配的尽量多的部分，将指针移动到这个位置，而不需要重新匹配。

我们如果能在字典树中使用这个思想，就能够在匹配失败时，快速的找到下一个匹配的位置。不需要每次重新遍历整个字典树就能快速的找到目标串种是否包含某个模式串。

2.4　AC 自动机

AC 自动机是一个建立在字典树的基础之上的有限自动机，除了字典树中**成功匹配**的状态转移外，还引入了 KMP 算法中的**失败匹配**的状态回退，即在匹配失败时找到模式串中与当前已匹配状态的后缀相同的最长前缀，将状态回退到这个位置。

2.4.1 AC 自动机的结构

例如给出字符串集合：

表 2-3 模式串集合

string_name
he
she
his
hers
is

对应的 AC 自动机如 图 2 所示：

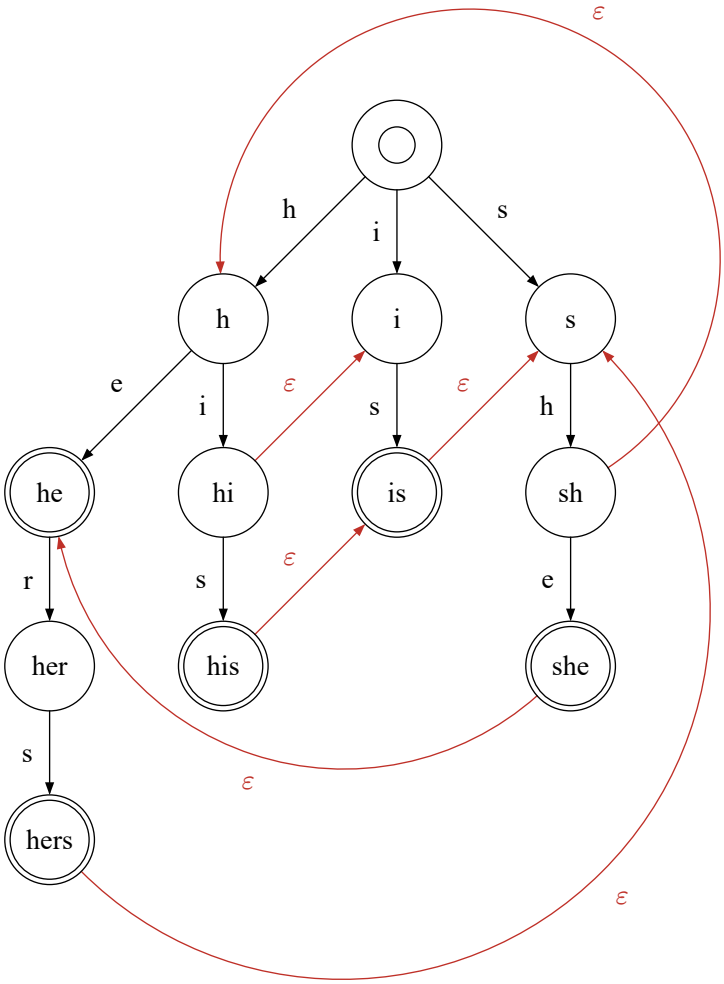

图 2-2 模式串集合的 AC 自动机

图中的 ε 表示非确定性有限自动机中的 ε 转移[3]，即不消耗任何输入，直接转移到下一个状态。在 AC 自动机中被称为 **fail** 指针。

图中未画出的 **fail** 指针均指向根节点。

可以清楚地看到，每个 **fail** 指针指向的状态都是当前状态的最长前缀，即在匹配失败时，可以直接跳转到这个状态，而不需要重新遍历整个字典树。

由 **fail** 指针的性质知道，只可能由深的节点指向浅的节点，因此在构建时，我们可以通过层序遍历的方式构建 **fail** 指针。

同时，**fail** 指针的连接构成了一棵内向树，根节点为根节点。

2.4.2 AC 自动机的构建

与构建字典树相比，AC 自动机的构建过程多了一个 **fail** 指针的构建。

1. 初始所有状态的 **fail** 指针指向根节点；
2. 层序遍历字典树，构建 **fail** 指针；
3. 若 $v = \delta(u, c)$，则 $\text{fail}(v) = \delta(\text{fail}(u), c)$；
4. 若 $\delta(\text{fail}(u), c)$ 不存在，则继续从 $\text{fail}(\text{fail}(u))$ 转移，以此类推，直到转移到根节点或者找到一个存在的转移。

可以发现这个过程与 KMP 算法中的前缀函数构造过程类似，只是在字典树中的状态转移上进行了扩展。

2.4.3 AC 自动机的匹配

与 AC 自动机的构建类似，匹配过程如下：

1. 初始化当前状态为根节点。
2. 逐个读入目标串中的字符：
   - 若下一个转移状态存在，则转移到该状态。
   - 若下一个转移状态不存在，则通过 **fail** 指针退回到直到一个存在的转移状态或根节点。
3. 读入字符后，当前状态表示以该字符为结尾的子串能在模式串中匹配的最长前缀。
4. 记录每个经过的状态，若状态为接受状态，则表示匹配成功。

由于某一个状态匹配成功了，那么它的 **fail** 指针指向的状态也一定匹配成功，因此我们可以通过 **fail** 指针的回溯来找到所有匹配成功的状态。

最后收集答案的时候可以使用类似于拓扑排序的方式，从 **fail** 树的叶节点开始，逐层向上回溯，找到所有匹配成功的状态。

2.4.4 字典图优化

在 AC 自动机的构建和匹配中，我们需要重复进行 **fail** 指针的回溯直到匹配成功。

类比 KMP 算法，的时间复杂度分析，匹配过程的时间复杂度为 $O(n + \sum m_i)$。

但是构建过程时无法保证 **fail** 指针的回溯次数，这是因为你并没有用到模式串中的有效信息，只知道 $\delta(\text{fail}(u), c)$ 不存在，但是不知道具体需要到哪一个 **fail** 指针才能保证 $\delta(\text{fail}(u), c)$ 存在。

我们构建的 AC 自动机中或字典树中，存在很多个空状态，而我们遇到这个状态的时候需要回溯到根节点，这是一个很大的开销。

一个朴素的想法就是，在构建的时候直接将这些空状态连接到应该回溯的状态上，这样在匹配的时候就不需要回溯到根节点，而是直接跳转到应该回溯的状态。

例如上面的例子，我们可以在 图 2 中加上如 图 3 所示的边，将原本需要回溯到根节点的状态直接连接到应该回溯的状态上。

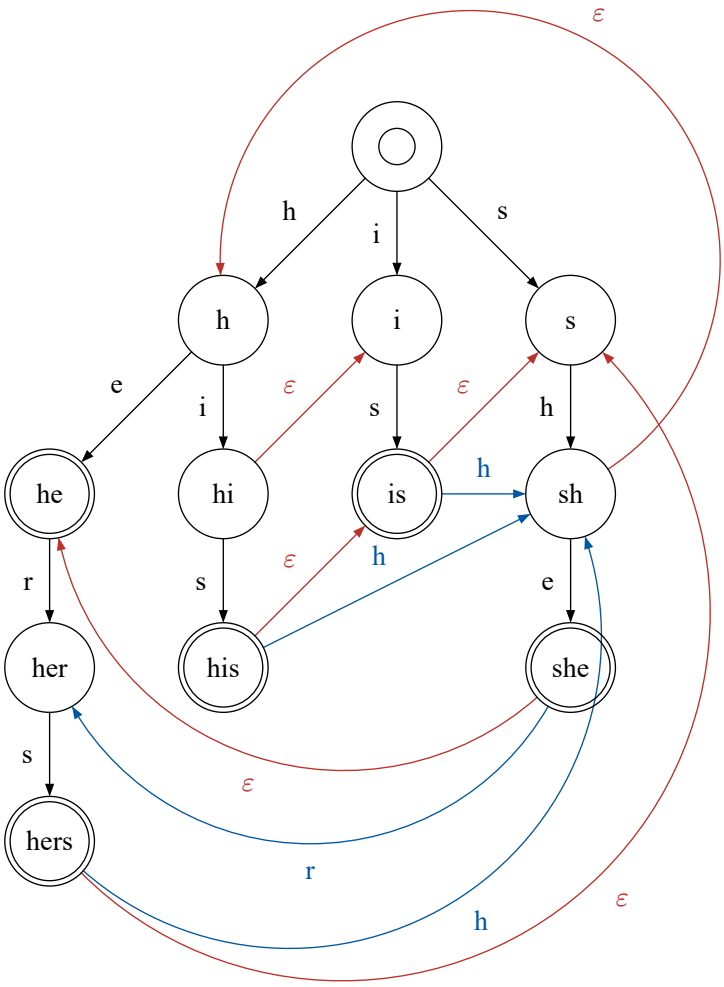

图 2-3　字典图优化后的 AC 自动机

图 3 中的蓝色边表示字典图优化后的边（省略了到根节点后转移的边），可以看到，我们将原本需要回溯到根节点的状态直接连接到了应该回溯的状态上。

例如状态 `she` 匹配失败时，若读入 `r`，则直接转移到状态 `her`，而不需要回溯到根节点。其他状态直接转移回根节点。

此时在匹配时只需要按照状态转移即可，不需要根据 **fail** 指针回溯，时间复杂度仍然为 $O(n+\sum m_i)$。

在构建时，每个节点只会用到它父亲与父亲的 **fail** 指针的状态，不会逐层回溯，因此构建的时间复杂度为 $O(|\Sigma| \times \sum m_i)$。其中 $|\Sigma|$ 为字母表大小，m_i 为模式串长度。

若用数组的方式存储状态转移，空间复杂度为 $O(|\Sigma| \times \sum m_i)$。

伪代码如下：

```
1  function ac_build(p : array_string) returns array_int
2      // 构建字典树
3      root ← node()
4      for i ← 0 to |p| − 1 do
5          cur ← root
6          for j ← 0 to |p[i]| − 1 do
7              if δ(cur, p[i][j]) then
8                  cur ← δ(cur, p[i][j])
9              else
10                 new_node ← node()
11                 δ(cur, p[i][j]) ← new_node
12                 fail(new_node) ← root
13                 cur ← new_node
14             end
```

```
        end
    end
    // 构建 fail 指针
    queue ← queue()
    for c ← Σ do
        if δ(root, c) then
            fail(δ(root, c)) ← root
            queue.push(δ(root, c))
        else
            δ(root, c) ← root
        end
    end
    while not queue.empty() do
        cur ← queue.front()
        queue.pop()
        for c ← Σ do
            if δ(cur, c) then
                fail(δ(cur, c)) ← δ(fail(cur), c)
                queue.push(δ(cur, c))
            else
                δ(cur, c) ← δ(fail(cur), c)
            end
        end
    end
    return root
end
```

算法 2-3　AC 自动机的构建算法

```
function ac_search(s : string, p : array_string) returns array_int
    root ← ac_build(p)
    cur ← root
    ans ← [0...m]
    count ← [0...size(root)]
    // 匹配
    for i ← 0 to |s| − 1 do
        // 直接转移
        cur ← δ(cur, s[i])
        count[cur] ← count[cur] + 1
    end
    queue ← queue()
    for v ← indegree_0(root) do
        queue.push(v)
    end
    // 拓扑排序
    while not queue.empty() do
        cur ← queue.front()
        queue.pop()
        if is_accepted(cur) then
            ans[index(cur)] ← count[cur]
        end
        count[fail(cur)] ← count[fail(cur)] + count[cur]
        remove_edge(cur, fail(cur))
        if indegree(fail(cur)) = 0 then
            queue.push(fail(cur))
        end
    // 返回的是每个模式串的匹配次数
    return ans
end
```

算法 2-4　AC 自动机的匹配算法

3 AC 自动机的应用

自 1975 年 Aho 和 Corasick 提出 AC 自动机以来，AC 自动机在字符串匹配领域有着广泛的应用。

3.1 多模式串匹配

作为 AC 自动机最基本的应用，多模式串匹配是 AC 自动机的一个重要应用。主要用于在一个目标串中匹配多个模式串。

1. 字符串匹配：搜索引擎中的关键词匹配；代码编辑器中的代码提示…
2. 文本过滤：游戏中过滤敏感词；过滤垃圾邮件[4]
3. 中文分词：使用动态规划概率推断的中文分词[5]…
4. 基因匹配：基因序列匹配[6]…
5. …

3.2 压缩字符串集合

AC 自动机通过构建多模式匹配树，优化字符串排序，帮助生成高重复性的 BWT 字符串，从而提升压缩效率。[7]。

3.3 各类有限自动机在算法竞赛中的应用

由 AC 自动机衍生出的各类有限自动机在算法竞赛中有着广泛的应用，例如：

1. KMP 自动机：类似于单串情况下的 AC 自动机，主要用于动态 / 可持久化字符串匹配等。[8]
2. 后缀自动机：又称 SAM（Suffix Automaton），功能最为强大，主要用于与单模式串的后缀匹配 / 后缀树 / k 大子串匹配 / LCP（Longest Common Prefix）等。[9]
3. 回文自动机：又称 PAM（Palindrome Automaton）或回文树，主要用于回文串匹配 / 回文子串个数统计 / 回文子串最长长度等。[10]

参考文献

[1] OI Wiki. Trie - OI Wiki2025. https://oi-wiki.org/string/trie/

[2] Donald E. Knuth, James H. Morris Jr., Vaughan R. Pratt. Fast Pattern Matching in StringsSIAM Journal on Computing, 1977, 6(2): 323-350

[3] Wikipedia. Nondeterministic finite automaton2025. https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

[4] 丁川芸, 兰全祥. 基于 AC 自动机和贝叶斯方法的垃圾内容识别 黑龙江工业学院学报（综合版）, 2019, 19(2): 36-39

[5] 徐懿彬. 基于 Aho-Corasick 自动机算法的概率模型中文分词 CPACA 算法 电子科技大学学报, 2017, 46(2): 426-433

[6] DRVLB Thambawita, Roshan G. Ragel, Dhammike Elkaduwe. An optimized Parallel Failure-less Aho-Corasick algorithm for DNA sequence matchingIn: 2016 IEEE International Conference on Information and Automation for Sustainability (ICIAfS), 2016: 1-6

[7] Bastien Cazaux, Eric Rivals. Strong link between BWT and XBW via Aho-Corasick automaton and applications to Run-Length EncodingarXiv preprint arXiv:1805.10070, 2018

[8] Leasier. KMP 自动机学习笔记 - 洛谷 2025. https://www.luogu.com.cn/article/e7u871vv

[9] OI Wiki. Suffix Automaton - OI Wiki2025. https://oi-wiki.org/string/sam/

[10] OI Wiki. PAM - OI Wiki2025. https://oi-wiki.org/string/pam/