从单模式匹配到多模式匹配

安江涛

同济大学软件学院

摘要：我们经常用的字符串方法 indexOf，都是判定两个字符串的包含关系，底层使用类似 KMP，BM，Sunday 这样的算法。如果我们要判断一个长字符串是否包含多个短字符串呢？比如在一篇文章找几个敏感词，如果我们的模式串存在多个，单模式匹配就不合适了，我们需要用到一种多模式匹配算法，本文将利用单模式匹配算法来讲解多模式匹配算法，从而便于读者理解多模式匹配算法。本文的多模式匹配算法为 AC 自动机。

关键词：AC 自动机 前缀函数 KMP

From single pattern matching to multi pattern matching

AN Jiangtao

School of software,Tongji University

Abstract: The string method indexOf, which we often use, determines the inclusion of two strings, with algorithms such as KMP, BM, and Sunday at the bottom. What if we want to determine whether a long string contains more than one short string? For example, in an article to find a few sensitive words, if our pattern string exists more than one, single pattern matching is not appropriate, we need to use a multi-mode matching algorithm, this article will use a single-mode matching algorithm to explain the multi-pattern matching algorithm, so as to facilitate the reader to understand the multi-mode matching algorithm. The multi-mode matching algorithm in this paper is AC automaton.

Key words: AC automaton Prefix function KMP

1 引言

AC 自动机是以 Trie 的结构为基础，结合 KMP 的思想建立的。它是一种多模式匹配算法，典型应用是用于统计和排序大量的字符串（但不仅限于字符串），经常被搜索引擎系统用于文本词频统计。AC 自动机是每一个想学习字符串匹配的人必须要了解的一个算法，但相信很多人初识 AC 自动机的时候都是知其然二不知其所以然，对其失配指针更是一头雾水。本文将详细讲解 KMP 的思想以及 Trie 的结构，然后再进一步讲解 AC 自动机，相信有了这个过渡，读者会更清晰的认识 AC 自动机。

2 前缀函数

前缀函数是 KMP 算法的核心思想。也是 AC 自动机的前缀知识。

2.1 定义

给定一个长度为 n 的字符串 s，其前缀函数被定义为一个长度为 n 的数组 next，其中 next[i]的定义是：

1. 如果子串 s[0…i]有一对相等的真前缀与真后缀：s[0…k-1]和 s[i-(k-1)…i]，那么 next[i]就是这个

相等的真前缀（或者真后缀）的长度，也就是 next[i]=k；

2. 如果不止有一对相等的，那么 next[i]就是其中最长的那一对的长度；
3. 如果没有相等的，那么 next[i]=0。

简单来说 next[i]就是子串 s[0…i]最长的相等的真前缀与真后缀的长度。

用数学语言描述如下：

$$\pi[i] = \max_{k=0\ldots i}\{k: s[0\ldots k-1] = s[i-(k-1)\ldots i]\}$$

特别的，规定 next[0]=0。

2.2 计算前缀函数

2.2.1 暴力求解

算法流程：

- 如果 j=0 并且没有任何一次匹配成功，则 next[i]=0。i 自增 1。
- 如果此时的真前缀与真后缀相等，则 next[i]=j，否则 j 自减 1，继续匹配，直到 j=0。
- 为了降低计算时间，令变量 j 从最大的真前缀长度 i 开始尝试。
- 在一个循环中以 i=1->n-1 的顺序计算前缀数组 next[i]的值。

具体代码如下：

```
vector<int> prefix(string s) {
  int n = s.size();
  vector<int> next(n);
  for (int i = 1; i < n; i++)
    for (int j = i; j >= 0; j--)
      if (s.substr(0, j) ==
s.substr(i - j + 1, j)) {
        next[i] = j;
        break;
      }
  return next;
}
```

注：string substr (size_t pos = 0, size_t len = npos) const;

显然该算法的复杂度为$O(N^3)$。

2.2.2 优化一

第一个重要的点为相邻的前缀函数至多增加 1.那么只需考虑：当取一个尽可能大的 next[i]时，必然要求新增的 s[i+1]也与之对应的字符匹配，即 s[i+1]=s[next[i]]如图 1 所示：

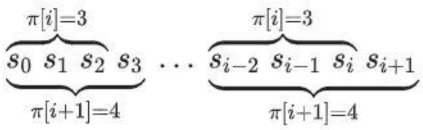

图 1

此时 next[i+1]=next[i]+1。所以 i 移动到下一个位置的时候，前缀数组要么加一，要么维持不变，要么减小。此时改进以后的代码为：

```
vector<int> prefix(string s) {
  int n = s.size();
  vector<int> next(n);
  for (int i = 1; i < n; i++)
    for (int j = next[i - 1] + 1; j
>= 0; j--)
      if (s.substr(0, j) ==
s.substr(i - j + 1, j)) {
        next[i] = j;
        break;
      }
  return next;
}
```

由于存在 j=next[i-1]+1 对于最大字符串比较次数的限制，可以看出每次只有在最好情况才会为字符串比较次数的上线积累 1，而每次超过一次的字符串比较消耗的是之后次数的增长空间。由此我们可以得到字符串比较次数的最多的一种情况：最少 1 次字符串比较次数的消耗和最多 n-2 次比较次数的积累，此时字符串比较次数为 n-1+n-2=2n-3。总体复杂度降到 $O(N^2)$。

2.2.3 优化二

在第一次优化中，我们讨论了当 s[i+1]=s[next[i]]的情况，我们再来讨论讨论不相等时改如何跳转。

失配时，我们希望找到对于子串 s[0…i]，仅次于 next[i]的第二长度 j，使得在位置 i 的前缀性质仍得以保持，也即 s[0……j-1]=s[i-j+1……i]如图 2 所示：

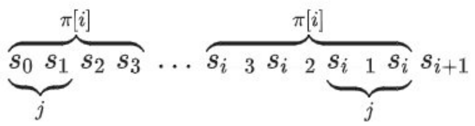

图 2

如果我们找到了这样的长度 j，那么仅需要再次比较 s[i+1]和 s[j]。如果它们相等，那么就有 next[i+1]=j+1。否则，我们需要找到子串 s[0……j]仅次于 j 的第二长度 $j^{(2)}$，使得前缀性质得以保持，如此反复，直到 j=0，如果 s[i+1]≠s[0]，则 next[i+1]=0。由此我们得到了一个关于 j 的状态转移方程：$j^{(n)}$=next[$j^{(n-1)}$-1)]，($j^{(n-1)}$>0)。

最终我们构建了一个不需要任何字符串比较，并且只进行$O(N)$次操作的算法：

```
vector<int> prefix(string s) {
  int n = s.size();
  vector<int> next(n);
  for (int i = 1; i < n; i++) {
    int j = next[i - 1];
    while (j > 0 && s[i] != s[j]) j
= next[j - 1];
    if (s[i] == s[j]) j++;
    next[i] = j;
  }
  return next;
}
```

3 KMP 算法

该算法由 Knuth、Pratt 和 Morris 在 1977 年共同发布。该算法是给定一个目标串 t 和模式串 s，找到 s 在 t 中的所有出现。为了简单起见，我们用 n 表示 s 的长度，用 m 表示 t 的长度。我们构造一个字符串 s+#+t，其中#为一个既不在 s 中出现也不在 t 中出现的分隔符。接下来计算该字符串的前缀函数。现在我们考虑该前缀函数 除去最开始的 n+1 个值（即属于字符串 s 和分隔符的函数值）后其余函数值的意义。根据定义，next[i]为右端点在 i 且同时为一个前缀的最长真子串的长度，具体到我们这种情况，其值为 s 的前缀相同且右端点位于 i 的最长字串的长度。由于分隔符的存在，该长度不可能超过 n。而如果 next[i]=n 成立，则意味着 s 完整地出现在该位置。因此如果在某一位置 i 有 next[i]=n 成立，则字符串 s 在字符串 t 的 i-(n-1)-(n+1)=i-2n 处出现。因此 Knuth-Morris-Pratt 算法（简称 KMP 算法）用$O(M+N)$的时间解决了该问题。

具体代码如下：

```
vector<int> kmp(string s, string t)
{
    int n = s.size();
    s += '#';
    s += t;
    vector<int> pre = prefix(s);
    vector<int> index;
    for (int i = n +
2;i<pre.size();i++) {
        if (pre[i] == n)
            index.push_back(i - 2 *
n);
    }
    return index;
}
```

4 字典树

AC 自动机是以字典树的结构建立的。字典树，英文名 Trie。顾名思义，就是一个像字典一样的树。用于保存字符串，与二叉搜索树不同，键不是直接保存在节点中，而是由节点在书中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应

的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

如图 3 所示，1->4->8->12 表示的就是字符串 caa：

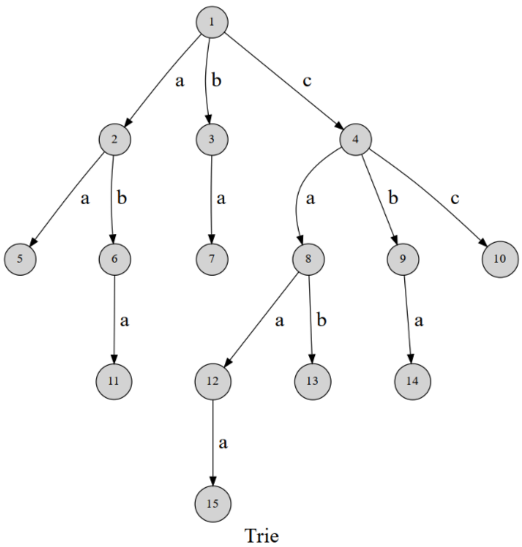

图 3

字典树的插入和查询都比较简单，在此用数组来存字典树。插入一个字符串时，在 trie 里依次插入字符串的字符,将字符串的最后一个字符节点做标记，表示结束。查询时，遍历查找字符串的字符，如果每个节点都存在,并且待查找字符串的最后一个字符对应的节点有标记，则表示该单词存在。

以下为一个封装好的模板：

```
const int N = 1000;
class trie {
private:
    int tree[N][26], cnt;
    bool exist[N];
    // 该结点结尾的字符串是否存在
public:
    void insert(string s) {
        // 插入字符串
        int p = 0;
        for (int i = 0; i <
s.size(); i++) {
            int c = s[i] - 'a';
            if (!tree[p][c])
                tree[p][c] = ++cnt;
            // 如果没有，就添加结点
            p = tree[p][c];
        }
        exist[p] = 1;
    }
    bool find(string s) {
        // 查找字符串
        int p = 0;
        for (int i = 0; i <
s.size(); i++) {
            int c = s[i] - 'a';
            if (!tree[p][c])
                return 0;
            p = tree[p][c];
        }
        return exist[p];
    }
};
```

5 AC 自动机

有了 KMP 算法和 Trie 的了解，接下来我们来讲解 AC 自动机。AC 自动机由贝尔实验室的两位研究人员 Alfred V. Aho 和 Margaret J. Corasick 于 1975 年发明，几乎与 KMP 算法同时问世。

AC 自动机的核心算法仍然是寻找模式串内部规律，达到在每次失配时的高效跳转。这一点与单模式匹配 KMP 算法是一致的。不同的是,AC 算法寻找的是模式串之间的相同前缀关系。

在 KMP 算法中，对于模式串”abcabcacab”，我们知道非前缀子串”abca”是模式串的一个前缀，而非前缀子串”cabca”不是模式串的前缀，根据此点，我们构造了 next 数组，实现在失配失败时的跳转。

而在多模式环境中,AC 自动机是使用前缀树来存放所有模式串的前缀，然后通过失配指针来处理失配的情况。它大概分为三个步骤：构建前缀树，添加失配指针，模式匹

配。

5.1 构建前缀树

AC 自动机在初始时会将所有模式串放到一个 Trie 中，然后在 Trie 上构建 AC 自动机。这个 Trie 就是上文中的字典树，构建方法相同。这里再说明一下字典树的结点的含义,Trie 中的结点表示的是某个模式串的前缀，我们在后文也称之为状态。一个结点表示一个状态,Trie 的边就是状态的转移。对于若干个模式串构成的字典树的所有状态的集合我们记作 Q。

具体代码如下：

```
int tree[N][26], cnt;
int exist[N], fail[N];
void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size();
i++) {
        if (!tree[u][s[i] - 'a'])
            tree[u][s[i] - 'a'] =
++cnt;
        u = tree[u][s[i] - 'a'];
    }
    exist[u]++;
}
```

5.2 添加失配指针

添加失配指针时，可以参考 KMP 中构建 next 指针的思想。

考虑字典树中当前节点u,u的父节点是p,p通过字符c的边指向u,即trie[p,c]=u。假设深度小于 u 的所有结点的 fail 指针都已求得：

1. 如果 tree[fail[p],c]存在，则让 u 的 fail 指针指向 tree[fail[p],c]。相当于在 p 和 fail[p]后面加一个字符 c，分别对应 u 和 fail[u]。
2. 如果 tree[fail[p],c]不存在,那么我们继续找到 tree[fail[fail[p]],c]。重复 1 的判断过次，一直跳 fail 指针直到根节点。
3. 如果真的没有，就让 fail 指针指向根节点。

如此即完成了 fail[u]的构建。

举个例子，对于字符串 i he his she hers 组成的字典树构建 fail 指针，我们重点分析结点6的fail指针构建如图4所示：

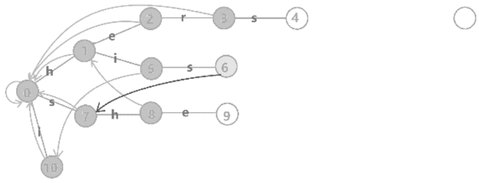

图 4

找到 6 的父节点 5，fail[5]=10。然而 10 结点没有字母 s 连出的边，继续跳到 10 的 fail 指针，fail[10]=0。发现 0 结点有字母 s 连出的边，指向 7 结点，所以 fail[6]=7。

具体代码如下，该函数的目标有两个，一个是构建fail指针,一个是构建自动机。

```
queue<int> q;
void build() {
    for (int i = 0; i < 26; i++)
        if (tree[0][i])
            q.push(tree[0][i]);
    while (q.size()) {
        int u = q.front();
        q.pop();
        for (int i = 0; i < 26; i++) {
            if (tree[u][i]) {
                fail[tree[u][i]] =
tree[fail[u]][i];
                q.push(tree[u][i]);
            }
            else
                tree[u][i] =
tree[fail[u]][i];
        }
    }
}
```

参数如下:

1. tree[u,c]:有两种理解方式。我们可以简单理解为字典树上的一条边，

也可以理解为从状态 u 后加一个字符 c 到达的状态，即一个状态转移函数。下文中我们将用第二种理解方式继续讲解。

2. 队列 q:用于 BFS 遍历字典树。
3. fail[u]:结点 u 的 fail 指针。

我们再根据上面的代码解释一下：build 函数将结点按 BFS 顺序入队，依次求 fail 指针。这里的字典树根节点为 0，我们将根节点的子结点一一入队。若将根节点入队，则在第一次 BFS 的时候，会根据根节点儿子的 fail 指针标记为自身。因此我们将根节点的儿子一一入队，而不是将根节点入队。

然后开始 BFS：每次取出队首的结点 u(fail[u]在之前的 BFS 过程中已求得)，然后遍历字符集（这里是 0-25，对应 a-z，即 u 的各个子节点）：

1. 如果 tree[u][i]存在，我们就将 tree[u][i]的 fail 指针赋值为 tree[fail[u]][i]。这里似乎有一个问题。根据之前的讲解，我们应该用 while 循环，不停的跳 fail 指针，判断是否存在字符 i 对应的结点，然后赋值，但是这里通过特殊化处理简化了这些代码。
2. 否则，令 tree[u][i] 指向 tree[fail[u]][i]的状态。

这里的处理是，通过 else 语句的代码修改字典树的结构。没错，它将不存在的字典树的状态链接到了失配指针的对应状态。在原字典树中，每一个结点代表一个字符串 S，是某个模式串的前缀，而在修改字典树后，尽管增加了许多转移关系，但结点所代表的字符串是不变的。

而 tree[S][c]相当于是在 S 后添加一个字符 c 变成另一个状态 S’。如果 S’存在，说明存在一个模式串的前缀是 S’，否则我们让 tree[S][c] 指向 tree[fail[S’]][c]。由于 fail[S’]对应的字符串是 S 的后缀，因此 tree[fail[S]][c]对应的字符串也是 S’的后缀。

换言之在 Trie 上跳转的时候，我们只会从 S 跳转到 S’，相当于匹配了一个 S’，但在 AC 自动机上跳转的时候，我们会从 S 跳转到 S’的后缀，也就是说我们匹配一个字符 c，然后舍弃 S 的部分前缀。舍弃前缀显然是能匹配的。那么 fail 指针呢？它也是在舍弃前缀，所以如果文本串能匹配 S，显然它也能匹配 S 的后缀。所谓的 fail 指针其实就是 S 的一个后缀集合。

tree 数组还有一种比较简单的理解方式：如果在位置 u 失配，我们会跳转到 fail[u]的位置。所以我们可能沿着 fail 数组跳转多次才能来到下一个能匹配的位置。所以我们可以用 tree 数组直接记录下一个能匹配的位置，这样就能省下很多时间。这样修改字典树的结构，使得匹配转移更加完善。同时它将 fail 指针跳转的路径做了压缩（就像并查集的路径压缩），使得本来需要条很多次 fail 指针变成跳一次。

这时候我们再来看上文中结点 6 的 fail 指针的构建如图 5 所示：

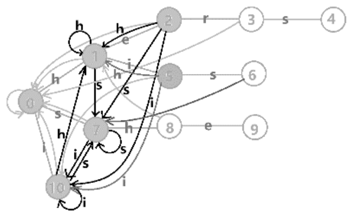

图 5

图中颜色较深的边是 AC 自动机修改字典树结构连出的边。本来的策略是找fail指针，于是我们跳到了 fail[5]=10 发现没有 s 连出的字典树的边，于是跳到 fail[10]=0，发现 tree[0][s]=7，于是 fail[6]=7;但是有了新边，我们跳到 fail[5]=10 之后直接走 tree[10][s]=7 就走到 7 号结点了。这就是 build 完成的两件事：构建 fail 指针和构建自动机。

5.3 模式匹配

接下来分析匹配函数 quetry()：

```
int query(string t) {
    int u = 0, res = 0;
    for (int i = 0; i < t.size();
i++) {
```

```
        u = tree[u][t[i] - 'a'];
// 转移
        for (int j = u; j &&
exist[j] != -1; j = fail[j]) {
            res += exist[j];
            exist[j] = -1;
        }
    }
    return res;
}
```

这里u作为字典树上当前匹配到的结点，res 即返回的答案。循环遍历匹配串，u 在字典树上跟踪当前字符。利用 fail 指针找出所有匹配的字符串，累加到答案中。然后清零。在上文中我们分析过，字典树的结构其实就是一个状态转移函数，而构建好这个函数后，在匹配字符串的过程中，我们会舍弃部分前缀达到最低限度的匹配。fail 指针则指向了更多的匹配状态。

5.4 复杂度分析

时间复杂度：定义|si|是模式串的长度，|S|是文本串的长度，|Σ|是字符集的大小（常数，一般为 26）。如果连了 trie 图，时间复杂度就是$O(\sum|s_i| + n|\sum| + |S|)$,其中 n 是 AC 自动机中结点的数目，并且最大可以达到$O(\sum|s_i|)$。如果不连 trie 图，并且在构建 fail 指针的时候避免遍历到空儿子，时间复杂度就是$O(\sum|s_i| + |S|)$。

6 结束语

综上所述，本文通过研究前缀数组，分析了 KMP 的算法思想，并解释了字典树的建立。通过这些前置知识对 AC 自动机进行了阐述，解答了失配指针的求解。希望通过这个过程能使读者对 AC 自动机有更深层次的理解。

参考文献

[1] 汤亚玲.KMP 算法中 next 数组的计算方法研究[J].计算机技术与发展,2009(6):98-101.

[2] 俞文洋,张连堂,段淑敏.KMP 模式匹配算法的研究[J].郑州轻工业学院学报：自然科学版,2007(5):64-66.

[3] 舒银东. 基于有限状态自动机的多模式匹配算法研究[D].合肥工业大学,2011.

[4] 朱俊. 多模式匹配算法研究[D].合肥工业大学,2010.