基于双数组 Trie 树算法的字典改进和实现

戴耿毅[1]，佘静涛[2]

（1. 浙江工业大学 信息学院，浙江 杭州 310032；2. 浙江工业大学 图书馆，浙江 杭州 310032）

摘 要：从汉字编码和构造过程两个方面对双数组 Trie 树算法进行改进和实现。在编码过程中，按照汉字的深度由浅入深依次编码；构造字典时，按照首字节点词条数目由大到小顺序构造。改进后的算法查找效率不变，但缩短了构造字典的时间；减少了数据稀疏，提高了空间利用率。针对改进前后的算法，从时间开支和空间开支两个角度分别进行对比，实验结果证明算法改进可行。

关键词：双数组；TRIE 字典；信息检索

中图分类号：TP312 **文献标识码**：A **文章编号**：1672-7800(2012)007-0017-03

0 引言

Trie 树，又称单词查找树或键树，来自英文单词“Retrieval”的简写，可以建立有效的数据检索组织结构，是一种高效的索引技术，适合字符串快速查找以及前缀匹配。中文分词往往需要使用字典查询，字典查询的速率直接影响中文分词的效率，因此 Trie 树常常被用来构建中文分词使用的字典；除此以外，Trie 树还可以用来实现用户输入的自动匹配，如当用户在输入框中输入“中”，自动匹配中国或者中华等。

Trie 树本质上是一个确定的有限状态自动机(DFA)，每个节点代表自动机的一个状态，根据变量的不同，进行状态转移，当到达结束状态或者无法转移的时候，完成查询。Trie 树的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。Trie 树查询的速率只和要查询的字符长度有关，查询结果最好的情况是 0(1)，即在第一层即可判断是否搜索到，最坏的情况是 0(n)，n 为查询词的长度。

以人民、人民大会堂、浙江、linux、like 为例，构造该 5 个词的 Trie 树如图 1 所示。

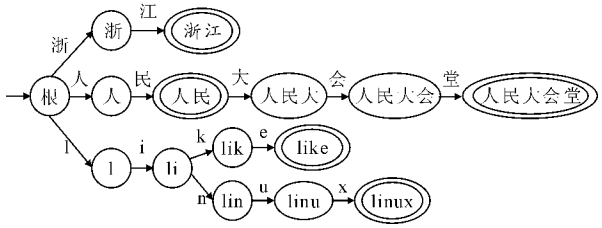

图 1 Trie 树结构

传统上的 DFA 一般用转换表方式来实现，表的列代表自动机的不同状态，行代表转换变量即输入字符，这种方式的查询效率很高，但是由于稀疏的现象严重，空间利用率较低。Trie 树的另一种实现方式是使用压缩的存储方式即链表来表示状态转移，但是由于数据结构复杂，查询效率较低，无法满足查询要求。为了让 Trie 树实现算法占用较少的空间，同时还要保证查询的效率，有学者提出了一种用 4 个线性数组表示 DFA 的方法，并进一步提出了用 3 个线性数组表示 Trie 树的方式。在此基础上，做出了进一步改进，用 2 个线性数组来进行 Trie 树的表示，即双数组 Trie(Double-Array Trie)。

1 双数组 Trie 树算法原理

双数组 Trie(Double-Array Trie)是 Trie 树的一个简单而有效的实现，由两个整数数组构成，称为 base 数组和 check 数组。base 数组的每个元素表示一个 Trie 节点，即一个状态；check 数组表示某个状态的前驱状态，初始状态设置 base 和 check 均为 0，在构建时如果对应状态的 base 和 check 值均为 0 标明该状态空闲。如果某状态时为一个完整的词，则 base 值设置为负数。如果某个状态为一个完整的词且该词不为其它词的前缀，其 base 值可以取其状态位置的负数。base 和 check 的关系满足下述条件：

$$base[s] + c = t$$

$$check[t] = s$$

其中，s 是当前状态的下标，t 是转移状态的下标，c 是输入字符的数值，如图 2 所示。

双数组 Trie 树在构造时，最复杂的是 base 值的确定，当前词对应状态 base 值的确定和所有下一个节点对应状态有关，base 值必须确保在数组中能放下所有的子节点。

作者简介：戴耿毅(1983－)，男，浙江宁波人，浙江工业大学信息学院硕士研究生，研究方向为计算机搜索引擎；佘静涛(1978－)，男，浙江杭州人，硕士，浙江工业大学图书馆馆员，研究方向为计算机网络、数据库系统。

如词中华、中国、中央、中间、中意 5 个词，假设“中”对应的状态为 i，华、国、央、间、中对应的输入为 code[华]、code[国]、code[央]、code[间]、code[意]。要确定 base[i]的值 b，b 必须要满足下面所有的条件：

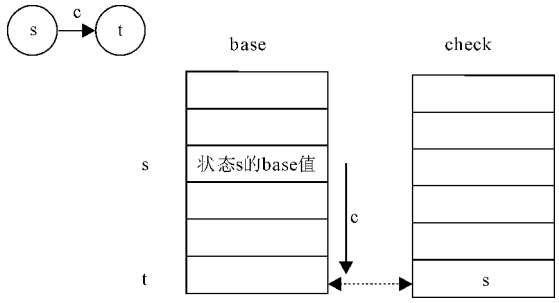

图 2 双数组 Trie 树原理

表示有新的位置可以存放元素

base[b+ code[华]]=0，check[b+ code[华]]=0；
base[b+ code[国]]=0，check[b+ code[国]]=0；
base[b+ code[央]]=0，check[b+ code[央]]=0；
base[b+ code[间]]=0，check[b+ code[间]]=0；
base[b+ code[意]]=0，check[b+ code[意]]=0；

寻找到满足上述所有条件的 b 值之后，就可以确定中华、中国、中央、中间、中意对应的状态，分别为 b+ code[华]、b+ code[国]、b+ code[央]、b+ code[间]、b+ code[意]。且可以设置这些状态的 check 值，为上一个状态 i。check[b+ code[华]]=i，check[b+ code[国]]=i，check[b+ code[央]]=i，check[b+ code[间]]=i，check[b+ code[意]]=i。其它多个汉字的词语以此类推。

双数组 Trie 树的查找相对构造来说，简单得多。比如查找“中世纪”前缀的词，查找“中”对应状态 i 的 base 值，检查 check[base[i]+code[世]] = i 是否成立，否则表明搜索失败，如果成立，继续判断 check[base[base[i]+code[世]]+code[纪]]= base[i]+code[世] 是否成立，成立则表明搜索成功，否则表明搜索失败。如果查找的是和“中世纪”完全匹配的词，还需要判断 base[[base[base[i]+code[世]]+code[纪]]]<0 是否成立，如果成立表明搜索成功。

2 双数组 Trie 树算法的改进和实现

双数组 Tire 树(Double-ArrayTire)算法有效地降低了 Tire 树结构的空间浪费，但是利用该算法生成的数组仍然还会存在较大的数据稀疏，其空间浪费仍可以再次优化。从双数组 Trie 树的构造过程中，可以分析出影响构造后数组大小的因素，主要是各个节点的 base 值和输入变量的大小，而各个 base 值的确定主要由数组当前的空闲位置及其子节点数目大小决定。一个节点的子节点越多，在确定其 base 值时产生冲突也就越多。因此主要从输入变量和首字子节点数目两个角度优化，减少输入变量的大小以及冲突的发生。汉字编码时，按照汉字在词中的深度由浅入深依次编码；构造字典时，按照首字子节点数目由大小顺序构造。汉字在词中的深度是指汉字是词中的第几个字，如词条“中国”、“中华”，深度为 1 级的有汉字“中”，深度为 2 级的汉字有“国”、“华”；1 级深度比 2 级深度要浅。

优化后的双数组 Trie 树算法构造过程如下：首先预处理需要构造的文件，将首字相同的词条排在一起，首字不同的词条按照相同首字词条数目由大到小排列，首字相同的词条按照次字 UTF 编码由小到大排序，依次类推，形成有序的词条集合。以人民、浙江、lie、民生、like 为例，排序后的顺序为 lie、like、人民、民生、浙江。

在英文中字符的 ACSII 编码比较小，可以直接使用 ACSII 编码来作为双数组 Trie 树的输入变量。但是汉字的字符编码比较大，直接使用汉字的字符编码会导致双数组 Trie 树严重稀疏。因此构建汉字或者汉字和英文混合组成的词条双数组 Trie 树字典时，应该首先对字符进行编码。编码时对所有词条从 1 级深度逐级增加编码，编码核心代码如下：

```
int coding(const char * * key, uint keynum, uint maxdeepth) {
    uint code = 0, UCS2 = 0;const char * string;uint i,j;
    for(j = 0; j < maxdeepth; j++){ for (i = 0; i < keynum; i++){
    string = key[i];
    unit wordnum = wordNum (string); // wordNum 函数返回字符串字符个数
    if(wordlen <= j) continue;
    UCS2 = wordUnit(string, j); // wordUnit 得到字符串 string 第 j 个字符的 utf 编码
    if (UCS2 == 0xffff) {continue;}
    if (code_[UCS2] == 0) {code++;code_[UCS2] = code;}}}
    code_max__= code; return code; }
```

以已经排序的词条 lie、like、人民、民生、浙江为例，按照深度级别由浅入深编码结果为：l－1，人－2，民－3，浙－4，i－5，生－6，江－7，e－8，k－9。

建立双数组 Trie 树字典时，初始化数组为 0。根节点对应 0 位置，其 base 值和 check 值都取初始化值 0。先确定所有词条 1 级深度字符(首字)对应位置的 base 值，确定 base 值时，确保其后的 2 级深度字符都能存入数组中。确定好 1 级深度字符对应位置的 base 值后，根据其后的 2 级深度字符编码，确定 2 级深度字符对应的位置及其相应的 check 值。依次类推可以确定 2 级深度字符对应的 base 值，直至字符结束。

还是以上述已经编码的词条 like、linux、人民、民生、浙江为例，首先按照词条顺序确定 l、人、民、浙 4 个首字的位置，l、人、民、浙 4 个字符对应的位置为 base[0]+code[l]=1、base[0]+code[人]=2、base[0]+code[民]=3、base[0]+code[浙]=4。其对应位置 check 值为根节点位

置即 check[1]=check[2]=check[3]= check[4]=0。确定位置 1 的 base 值，字符“l”后面接的字符有“i”，所以 base[1]的值 k 只要保证位置 k+code[i]=k+5 对应的 base 值和 check 值为 0 即可。可以取 base[1]=1，由此确定“li”对应位置的 check[6]=1。base[2]需要满足 base[base[2]+code[民]]=base[base[2]+3] =check[base[2]+3]=0，可以取 base[2]=2，则 check[2+3]=check[5]=2。令 base[1+code[生]]= base[1+6] =check[1+6]=0，可取 base[3]=1，则 check[7]=3。同理，令 base[4]=1，则“浙江”对应位置 的 check[8]=4。“li”对应的位置为 6，令 k=base[6] ，要使 base[k+9]=check[k+9] = base[k+8]=check[k+8]=0，取 base[6]=k=1，check[10]=check[9]=6。由于“人民”是一个完整的且不是其它词的前缀，因此取其对应位置的负数－5 为其 base 值，base[5]=－5。同理 base[7]=－7，base[8]=－8，base[9]=－9。“lik”对应位置的 base 值只要保证 base[10]+code[e]= base[10]+8 所在位置的 base 和 check 值为 0 即可，取 base[10]=3，则 check[11]=10。“like”是个完整的词，base 值取－11。最终构造成的双数组如表 1 所示。

表 1　构造后的双数组

	0	1	2	3	4	5	6	7	8	9	10	11
base	0	1	2	1	1	−5	1	−7	−8	−9	3	−11
check	0	0	0	0	0	2	1	3	4	6	6	10
字符	根	l	人	民	浙	人民	li	民生	浙江	lie	lik	like

双数组 Trie 树查询字符串时，从根节点开始，根据输入变量确定下一个位置，检查下一个位置的前一个位置是否为当前位置，如果是，则再根据下一个位置的 base 值和相应的输入变量确定下下个位置，依次类推。

我们以查询“浙江”这个词为例，首先确定下一个位置 t1=base[0]+code[浙]=4，check[4]=0 成立，因此继续寻找下一个位置 t2=base[4]+code[江]=8，check[8]=4 成立，再由于 base[8]=－8 小于零，所以“浙江”这个词存在双数组 Trie 树中。当查询字典中不存在词“江河”时，确定位置 t1= base[0]+code[江]=7，检查 check[7]不等于 0，因此查询失败。

双数组 Trie 树构造的字典查询操作所需要的时间很短，作几次加法就可以查到相应的词语。且查询复杂度跟被查询词语的长度相关，而跟词典规模的大小没有关系。

3　实验结果比较

为了更好地证明改进后的双数组 Trie 树算法的性能，我们对改进前后的算法进行了对比实验。实验环境为 32 位 linux 操作系统，内存 1G，选择大小不同的 3 个文件作为实验对象。文件 A 包含 1331 条词，有 1402 个不同的字符，大小为 55KB；文件 B 包含 12801 条词，有 2641 个不同的字符，大小为 549KB；文件 C 包含 40001 条词条，有 4135 个不同的字符，大小为 1.7MB。主要从双数组 Trie 树构造的时间以及构造后的数组大小两个角度比较。时间记录函数选择 clock()函数，构造的时间以秒来计算；构造后的双数组大小以数组元素总个数为参考。实验结果如表 2 和表 3 所示。

表 2　改进前后算法构造时间比较

	A	B	C
改进前	0.14	7.08	41.86
改进后	0.07	4.29	12.9

表 3　改进前后算法构造后数组大小比较

	A	B	C
改进前	19972	415917	1294674
改进后	19770	414329	1292972

从实验结果可以得出，对于同样的文件，改进后的算法比改进前的算法构造的时间要短，文件越大，构造时间的差距越大，效果越明显；从表 3 可以看出对于不同大小的文件，改进后的算法构建的双数组大小都比改进前的要小，改进后的算法缩小了构造后的数组大小，同样的词条内容，占用数组的位置相同，数组总大小越小，数组的稀疏程度越小。实验证明算法改进有效。

构建时间减少了，减少的幅度较大；但是对存储的稀疏性其实减少的并不多。

4　结语

本文从汉字编码和构造过程两个方面对双数组 Trie 树算法进行改进和实现。通过试验得出，改进后的算法与改进前的算法相比具有构造时间短、数组稀疏程度小的优点。

参考文献：

[1] FREDKIN E. Communication of the ACM[C]. Vol. 3:9 (Sep 1960).

[2] AHO A V, SETHI R, ULLMAN J D. Compilers:Principles, techniques, and tools[M]. Addison-Wesley. 1985.

[3] AOE J. An efficient digital search algorithm by using a double-array structure[C]. IEEE Transactions on Software Engineering. 1989.

[4] 赵欢，朱红权. 基于双数组 Trie 树中文分词研究[J]. 湖南大学学报，2009(5).

[5] 王思力，张华平，王斌. 双数组 T rie 树算法优化及其应用研究[J]. 中文信息学报，2006(5).

[6] 温滔，朱巧明. 一种快速汉语分词算法[J]. 计算机工程，2004(19).

（责任编辑：杜能钢）