文章编号: 1674-2974(2009)05-0077-04

基于双数组 Trie 树中文分词研究*

赵 欢†, 朱红权
(湖南大学 计算机与通信学院, 湖南 长沙 410082)

摘 要: 对双数组 Trie 树(Double-Array Trie)分词算法进行了优化: 在采用 Trie 树构造双数组 Trie 树的过程中, 优先处理分支节点多的结点, 以减少冲突; 构造一个空状态序列; 将冲突的结点放入 Hash 表中, 不需要重新分配结点. 然后, 利用这些方法构造了一个中文分词系统, 并与其他几种分词方法进行对比, 结果表明, 优化后的双数组 Trie 树插入速度和空间利用率得到了很大提高, 且分词查询效率也得到了提高.

关键词: 自然语言处理; 双数组; Trie 树; 词典; 分词
中图分类号: TU471 文献标识码: A

Research of Chinese Word Segmentation Based on Double-Array Trie

ZHAO Huan†, ZHU Hong-quan
(School of Computer and Communication, Hunan Univ, Changsha, Hunan 410082, China)

Abstract: This paper proposed some improved strategies for the algorithm of Double-Array Trie. Firstly, the priority was given to the node with most child nodes in order to avoid the collision; secondly, an empty-list was defined; Finally, the collision node was added to a hash table, which avoided re-allocation. Then, we implemented a program for a Chinese word segmentation system based on the improved Double-Array Trie and compared it with several other methods. From the results, it turns out that the insertion time and the space efficiency are achieved, and that search efficiency is improved.

Key words: natural language processing systems; double-array; trie; lexicon; word segmentation

中文信息处理存在着分词的问题, 但分词必须有一个足够大的词库, 词库技术对于搜索有很大影响. 理想情况下是包含所有的词语, 任意词串只要能在词库中查询到, 就认为是词语, 但势必存在大量数据的存储和搜索问题. 词库目前主要采用索引结构来实现, 常用的包括线性索引表、倒排表、散列(Hash)表以及搜索树.

线性索引和倒排表都是静态索引结构, 不利于更新, 只能按顺序或者折半搜索数据. 散列表则是根据设定的 Hash 函数 H(key)和处理冲突的方法将关键字映射到一个存储位置[1]. 搜索时只要对关键码进行函数计算, 得到存储位置, 搜索速度较快, 但冲突只能尽可能地少, 不可能完全避免, 另外还存在空间浪费问题.

搜索树包括 B 树和 Trie 树等. 它们的结构比较复杂, 设计好的话能提高检索效率. 另外还有一些基于这两种方法的变种[2-5].

本文首先介绍了双数组 Trie 树 (Double-Array Trie, DAT)的基本原理, 然后对其进行优化设计, 最后进行实验比较, 得出结论.

1 双数组 Trie 树基本原理

Trie 树用于确定词条的快速检索[1]. 对于给定

* 收稿日期: 2008-10-09
基金项目: 教育部科学技术研究重点项目资助(106458)
作者简介: 赵 欢(1967—), 女, 湖南长沙人, 湖南大学教授
† 通讯联系人, E-mail: hzhao@hnu.cn

的一个字符串 a_1, a_2, a_3, …a_n, 采用 Trie 树搜索最多经过 n 次匹配即可完成一次查找(即最坏是 $O(n)$), 而与词库中词条的数目无关, 缺点是空间空闲率高, 它是中文匹配分词算法中词典的一种常见实现.

双数组 Trie 树是 Trie 树的一种变形, 是在保证 Trie 树检索速度的前提下, 提高空间利用率而提出的一种数据结构. 其本质是一个确定有限状态自动机(Deterministic Finite Automaton, 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时完成查询. DAT 采用两个线性数组(*base* 和 *check*)对 Trie 树保存, *base* 和 *check* 数组拥有一致的下标, 即 DFA 中的每一个状态, 也即 Trie 树中所说的节点, *base* 数组用于确定状态的转移, *check* 数组用于检验转移的正确性, 检验该状态是否存在[6].

由文献[6]和文献[7], 有如下定义:

定义 从状态 s 输入 c 到状态 t 的一个转移必须满足:

$$t = base[s] + N(c);$$

$$check[t] = s.$$

式中: $N(c)$为字符 c 的序列码, 在词条插入时根据插入顺序建立序列码, 但不会有两个非空结点通过 $base[s]$ 映射到 *check* 数组中的同一位置. 当 $base[s]$ 和 $check[s]$ 都为 0 时表示该位置为空, 当 s 是可结束状态时, $base[s]$ 为负值[7].

2 双数组 Trie 树的优化

在传统的双数组 Trie 树中, 树中每个结点在数组中的位置, 都是由其父节点也就是上一状态的 *base* 值决定的. 而一个节点的 *base* 值取决于数组的当前空闲位置以及该节点的直接子结点. 如果某结点的子结点较多, 该节点在插入过程中就会遇到很多冲突. 为减少冲突, 文献[8]优先处理子结点数多的结点, 并将 Trie 树中的所有结点按子节点数进行降序排列. 这样做在一定程度上提高了速度, 但仍有可优化的空间, 且空间利用率不高.

为获得高的空间利用率, 本文对所有空状态构建一个序列, 在插入某节点确定 *base* 值时, 只需扫描空状态序列即可.

对双数组中空状态递增结点 r_1, r_2, …, r_m, 构建这样空空序列:

$$check[r_i] = -r_{i+1} (1 <= i <= m-1),$$

$$check[r_m] = -DA\_SIZE.$$

式中: $r_1 = E\_HEAD$, 为第一个空值状态对应的索引, DA_SIZE 为 Trie 树的结点数. 这种方法在空状态并不太多的情况下可以进一步大大提高插入速度. 判断 *check* 数组中的空元素非常简单, 只需判断 *check* 数组元素值是否为负数即可[5].

文献[8]在一定程度上减少了冲突. 本文在此基础上进行了进一步的优化: 在处理冲突的过程中, 不需要重新对结点进行分配, 只需将冲突结点对应数组索引放入 hash 表中即可, 这样可以大大提高速度.

2.1 优化双数组 Trie 树的建立过程

2.1.1 核心算法

1)寻找当前结点 *base* 数组下标值 $X\_CHECK(A, c, limit)$, A 为当前结点所有子结点字符集, c 为 A 中字符序列码最小值, $limit$ 为限制值.

步 **1** 变量 e_index 初始化为 E_HEAD;

步 **2** 让 $q = e\_index - c$. 若对于所有字符 $ch \in A$, 有 $check[q + N(ch)] < 0$ 并且 $q > limit$ 则返回 q; 否则 $e\_index = -check[e\_index]$, 进入步骤 3;

步 **3** 若 $e\_index < DA\_SIZE$, 进入步骤 2; 若 $e\_index >= DA\_SIZE$, 返回 $e\_index - c$.

2)设置 *check* 值 *setCheck*(idx, val).

步 **1** 若 $idx >= DA\_SIZE$, 返回;

步 **2** 若 $check[idx] < 0$, 进入步骤 3, 从空状态序列中删除该索引 idx;

步 **3** 若 $idx \neq E\_HEAD$, 进入步骤 4; 若 $idx = E\_HEAD$, 则 $E\_HEAD = -check[idx]$, $check[idx] = val$.

步 **4** 令 $pre\_index = E\_HEAD$, 扫描空状态序列, 找到满足 $idx = -check[pre\_index]$ 的 pre_index, 则 $check[pre\_index] = check[idx]$, $check[idx] = val$, 终止函数.

2.1.2 构造算法全过程

步 **1** 初始化

数组前 n 个位置留给根结点的直接子结点. 令 $E\_HEAD = 1$, 设置根节点的所有直接子结点 *check* 值为 0, 即调用 *setCheck*. 因为这些节点的索引就是它们的序列码, 减少空列表遍历次数.

步 **2** 初始化后 E_HEAD 将变为非根节点的第一个空节点, 将根结点的所有子结点加入新建的空队列中.

步 **3** 若队列非空, 在队列中选出子结点数最

多的结点为当前处理结点 *curNode*. 否则, 算法结束, 数组构造完成.

步 **4**　访问当前结点 *curNode*, 根据 *X_CHECK* 算法确定其数组下标 *s*, 及各直接子结点在数组中的位置 (要保证唯一性). 如节点遇到字符 *ch*, 若 $base[s]+N(ch)>=DA\_SIZE$, 则该子节点数组下标为 *E_HEAD*, 则 $check[E\_HEAD]=s$, 同时将该 *E_HEAD* 值保存在 *hash* 表中, 对应的键为下标 *s* 和字符 *ch* 生成的 *hash* 码 *hashCode* = *hash*(*s*, *ch*). 即(*hashCode*, *E_HEAD*)—>(键, 值对), 根据空序列将 *E_HEAD* 后移到空闲节点. 否则, $check[base[s]+N(ch)]=s$;

步 **5**　将当前结点的所有直接子结点加入到队列中, 重复步 3.

2.1.3　分词所需要的查询算法

步 **1**　假设当前状态为 *s*, 输入字符为 *c*, *N*(*c*)为序列码.

步 **2**　循环过程:

$t=base[s]+N(c)$;

If($t>=DA\_SIZE$)

then 根据 *s* 和字符 *c* 生成的 *hash* 码 *hash*(*s*, *ch*)得到 *t* 值.

If($check[t]=s$)then

$s=t$;

else fail;

endif

步 **3**　若 $base[t]$ 不为负, 重复步骤 1. 否则, *t* 为一个可结束状态.

2.2　优化双数组 **Trie** 树的具体应用

假定词表中只有"啊, 阿根廷, 阿胶, 阿拉伯, 阿拉伯人, 埃及"这几个词, 用 Trie 树表示如图 1 所示.

首先, 对词表中所出现的 10 个汉字进行编码: 啊——1, 阿——2, 唉——3, 根——4, 胶——5, 拉——6, 及——7, 廷——8, 伯——9, 人——10. 对每个汉字, 要确定一个 *base* 索引值, 使得所有以该字开头的词, 在双数组中都能放下. 例如, 要确定"阿"字的 *base* 值, 假设以"阿"开头的词的第二个字序列码依次为 a_1, a_2, …, a_n, 要确定 *q*, 使得 $check[q+a_1]$, $check[q+a_2]$, …, $check[q+a_n]$ 均为负数. 找到了 *q*, "阿"的 *base* 值就为 *q*. 用这种方法构建双数组 *Trie*, 需经过 4 次遍历, 将所有的词语放入双数组中. 本文用负 *base* 值表示该位置为词语, 若状态 *q* 对应某一个词, 且是叶子结点, 令 $base[q]=(-1)*q$, 若不是叶子结点, 令 $base[q]=(-1)*base[q]$. 最后得到双数组如下:

$base[]=\{0,-1,1,1,-4,1,-6,1,-8,-9,-1\}$,

$check[]=\{0,0,0,0,10,2,2,2,3,5,7\}$.

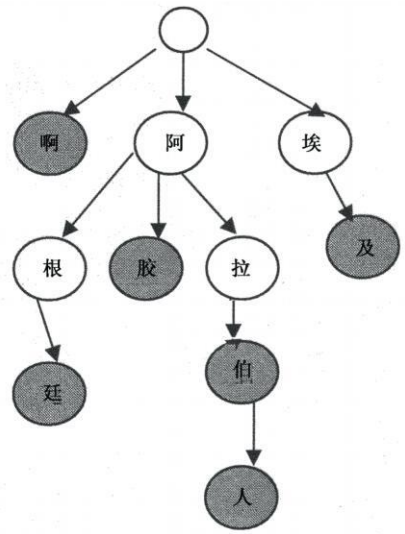

图 1　词表构造 Trie 树

Fig. 1　Established Trie base on wordlist

例如要搜索"阿拉伯人"是否为此表中一个词, 首先由状态"阿"的序列码为 2 得到 $base[2]=1$, 接下来输入变量是"拉", 序列码为 6, $base[2]+6=7$, $check[7]=2$, 所以"阿拉"是一个状态, 可以继续. $base[7]=1$, 接下来输入变量是"伯", 序列码是 9, $base[7]+9=10$, $check[10]=7$, 所以"阿拉伯"是一个状态, 可以继续. $base[10]=-1$, 接下来输入变量是"人", 序列码是 10, $|base[10]|+10=11$, 而 $11>=DA\_IZE$, 所以从前面的构造过程中保存的 *hash* 表得到对应下标为 4, 而 $check[4]=10$, 同时 $base[4]=-4$. 所以判定"阿拉伯人"是词表中的词.

由上面搜索过程可知, 优化后的双数组 Trie 树算法时间开销只与词长度有关, 算法时间复杂度为 $O(n)$, *n* 为词长度. 而且构造词典的过程中, 生成的数组长度为 Trie 树结点数.

3　实验比较及结果分析

本文对 3 种词典算法在相同环境下进行比较. 第 1 种是普通 Trie 树词典机制; 第 2 种是文献[8]的双数组 Trie 树词典机制; 第 3 种是本文优化后的 DAT 机制. 实验环境是 CPU 3.0GHZ(Intel Pentium 4), 内存 512M, 操作系统为 Windows XP, 所

用词典总共包括 55 501 个词条.

表 1 ~表 3 为实验结果.

表 1 构造双数组 Trie 树时间

Tab. 1 Time for establishing double-array

类别	t/s
文献[8]	3 366.328
本文	434.593

由表 1 可知, 构造本文优化双数组 Trie 树的时间远远少于构造文献[8] 双数组 Trie 树的时间.

表 2 占用空间

Tab. 2 Space occupied

类别	空间
文献[8]	数组长度 186 375
本文	数组长度 72 492, hash 表长 18

由表 2 可知, 本文优化的双数组 Trie 树空间占用率不到文献[8] 双数组 Trie 树的一半. 同时, 还可以预知随着词条数越多, 空间利用率也会越高.

表 3 给定语料分词速度

Tab. 3 Time for word segmentation

类别	t/s
普通算法	0.063
文献[8]	0.047
本文	0.031

该实验意在比较用于正向最大匹配分词的速度. 语料库文本为任意选择语料, 大小为 44 k. 由表 3 可知, 普通 Trie 树分词平均速度为 700 kB/s, 文献[8] 双数组 Trie 树分词平均速度为 936 kB/s, 而本文优化后的双数组 Trie 树分词平均速度则达 1.3 MB/s.

4 结 论

由理论分析和实验结果均可以看出, 优化后的双数组 Trie 树不仅在构造词典和搜索词语速度上, 而且在空间的利用率上都得到了很大提高, 因为数组长度始终是 Trie 树结点总数[5].

这种方法也有缺点, 由于本文在构造过程中数组长度是固定的, 不会自动增长, 当有结点发生冲突时, 则把该节点保存在空状态序列头, 同时保存实际数组下标到 hash 表中, 当插入词语数据量比较大时, 解决 hash 表的冲突将是下一步研究的关键.

参考文献

[1] 殷人昆. 数据结构(C++语言版)[M]. 北京: 清华大学出版社, 1999.
YIN Ren-kun. Data structures (C++)[M]. Beijing: Tsinghua University Press, 1999. (In Chinese)

[2] 杨文峰, 陈光英, 李星. 基于 PATRICIA Tree 的汉语自动分词词典机制[J]. 中文信息学报, 2001, 15(3): 44—49.
YANG Wen-fen, CHEN Guang-yin, LI Xing. Patricia-tree based dictionary mechanism for chinese word segmentation[J]. Journal of Chinese Information Processing, 2001, 15(3): 44—49. (In Chinese)

[3] 温滔, 朱巧明. 一种快速汉语分词算法[J]. 计算机工程, 2004, 30(19): 119—182.
WEN Tao, ZHU Qiao-ming. A fast algorithm for chinese word segmentation[J]. Computer Engineering, 2004, 30(19): 119—182. (In Chinese)

[4] 吴胜远. 一种汉语分词方法[J]. 计算机研究与发展, 1996, 33(4): 306—311.
WU Sheng-yuan. A new chinese phrase segmentation method[J]. Journal of Computer Research and Development, 1996, 33(4): 306—311. (In Chinese)

[5] KAZUHIRO M, EL—SAYED A, MASAO F. Fast and compact updating algorithms of a double-array structure[J]. Information Sciences, 2004, 159: 53—67.

[6] THEPPITAK K. An implementation of double-array trie[Z]. http://linux.thai.net/~thep/datrie/datrie.html, 2006.

[7] JUN-ICHI A, SEIGO Y, TAKASHI S. An efficient digital search algorithm by using a double-array structure[J]. IEEE Transactions on Software Engineering, 1989, 15(9): 1066—1077.

[8] 王思力, 张华平, 王斌. 双数组 Trie 树算法优化及其应用研究[J]. 中文信息学报, 2006, 20(5): 24—30.
WANG Si-li, ZHANG Hua-ping, WANG Bin. Research of optimization on double-array trie and its application[J]. Journal of Chinese Information Processing, 2006, 20(5): 24—30. (In Chinese)