机器词典的动态管理技术

孙健　胡海文　王启祥
（南京大学计算机科学与技术系）

摘 要：机器词典是自然语言处理系统中的重要组成部分，如何很好的管理词典，使系统具有动态修改、动态调试的能力，将对缩短整个系统的开发周期产生重要影响。本文提出了一种基于B树的机器词典的动态管理技术。该技术可用来实现基于词典的动态调试，为自然语言处理系统的开发带来方便。

The Dynamic Management Technology of Machine Dictionary

Sun jian　Hu haiwen　Wang qixiang
(Computer Science and Technology Department, Nanjing University)

Abstract: A machine dictionary is a important component of natural language processing system. A excellent management utility can not only make the dictionary get the power of dynamic modifying and debugging but also speed up the development process of the whole system. This paper discusses a kind of dynamic management technology of machine dictionary based B-tree, it can be applied to implement dynamic debugging on the dictionary and bring convenience into the development of NLP system.

1 引言

在自然语言处理的分析、生成阶段都必须与机器词典作交互。最初的词典一般是人工录入好的，难免会存在许多结构和意义上的错误。在转换作机器词典后，这些错误会直接影响到分析和生成的准确度。 就汉语词典来说 ， 一般采用首字索引Trie结构来加快词典的查找速度，这种静态索引适用于开发后的系统。但对处于开发中的系统来说，这种静态索引结构是不适用的。我们的设想是，用动态索引技术来管理开发中尚未定型的机器词典，在分析、生成的调试阶段，一旦发现了属于词典的错误，就可以动态随机的进行修改，然后在当前调试现场继续被中断的调试过程，这样既不损失效率又带来了方便。

B树是一种很好的动态索引结构由于有完善的理论基础，并且曾被成功的应用于操作系统及许多数据库系统，使我们自然的选择它来实现机器词典的动态管理，并在其基础上实现一个与词典相关的动态调试环境，为整个自然语言处理系统的开发过程带来方便，最

后它可单独作为用户词典管理工具被集成在整个系统中，与高效的静态索引并存。

2 B树索引结构

B树是一种大型外存索引结构，在理论上，B树的性能主要与访外次数有关，而访外次数一般与B树的高度成正比。所以，在实现时，应尽量增大每个结点的容量即B树的阶而压缩B树的高度。考虑到实际词典的大小及内存的限制，在我们的实现中，阶数为51而最大高度为3，并且把根结点常驻内存，这样访外次数最多为2，可以满足快速查找的需要。

（1） B树结点的数据结构

作为树结构，B树是由许多作为关键码集合的分支结点组成的，祖先结点和子孙结点由链连起来，这些链实际上是外存地址。举例说来，对一个100阶的B树，它每个结点包含99个查找关键码，99个指向数据记录的磁盘地址，100条链，以及一些必要的头信息。下面是这些结点的数据结构。

```
Type
  NodeHead = Record      (* 结点头信息 *)
    Order     :integer; (* B树的阶 *)
    FileLoc   :integer; (* 本结点在索引文件中的位置 *)
    ParentLoc :integer; (* 双亲结点在索引文件中的位置 *)
    MaxKeys   :integer; (* 每个结点所能容纳的最大关键码数 *)
    MinKeys   :integer; (* 每个结点所能容纳的最少关键码数 *)
    KeySize   :integer; (* 关键码最大长度 *)
    NoOfKeys  :integer  (* 本结点实际容纳的关键码数 *)
  end;

  Node = Record                              (* 结点类型 *)
    head :NodeHead;                          (* 标识本结点的头信息 *)
    key  :array [1..MAXKEY] of String;       (* 关键码集合 *)
    ptr  :array [1..MAXKEY] of DiskOffset;   (* 指向数据记录的指针集合 *)
    lnk  :array [1..ORDER]  of integer;      (* 指向所有子女结点的链集合 *)
                                             (* 值为负表示本结点是叶结点 *)
  end;
```

(2) B树索引的数据结构

把以上所定义的结点按照一定的规则链起来，就构成了B树，以下是B树类型。

```
Type
  BTreeHead = Record    (* B树索引的头信息 *)
    Order    :integer;  (* 阶 *)
    KeySize  :integer;  (* 关键码最大长度 *)
    RootLoc  :integer;  (* 根结点在索引中的位置 *)
    FirstDel :integer   (* 第一个被删除结点在索引中的位置 *)
  end;

  BTree = Record                          (* 索引文件结构 *)
    head  :BTreeHead;                     (* 标识本索引文件的头信息 *)
    index :array [1..MAXNODE] of Node     (* 结点集合 *)
  end;
```

这里有必要提一下FirstDel变量，这个变量指向第一个被删除结点的位置，而这个被删除结点的head.FileLoc变量又指向下一个被删除结点的位置，这样一个接一个，则索引文件中所有被删除的结点都被链接了起来。在写入下一个新结点时，我们可以覆盖这些被删除的结点从而节约磁盘空间。

3 B树的实现

(1) B树的检索算法search_key

search_key是一个递归函数，设所要查找的目标关键码为tkey，B树当前结点为current_node，初始值为根结点root，常驻内存，i为下标，初始值为0。

① tkey与current_node.key[i]比较，分三种情况:

② tkey = current_node.key[i],
则 成功，返回current_node.ptr[i]并记录当前位置;

③ tkey > current_node.key[i],
若 i = current_node.head.NoOfKeys 并且 current_node.lnk[1] = -1,
则 失败，记录当前位置;
否则 若 i = current_node.head.NoOfKeys,
则 执行
current_node := BTree.index[current_node.lnk[i+1]],
i:=0, 转①;
否则 执行i:=i+1, 转①;

④ tkey < current_node.key[i],
若 current_node.lnk[1] = -1,
则 失败，记录当前位置;

否则 执行

current_node := BTree.index[current_node.lnk[i]],

i:=0, 转①;

search_key成功后返回有关tkey在索引文件中的位置以及词条信息所在的磁盘偏移值，根据这个偏移值我们可以从磁盘上取出tkey的词条信息。

(2) B树的插入算法insert_key

① 由search_key失败后记录的当前位置作为tkey在索引文件中的插入位置，这个位置用current_node标记;

② 执行 current_node.head.NoOfKeys := current_node.head.NoOfkeys+1;

若 current_node.head.NoOfKeys <= current_node.head.MaxKeys,

则 插入tkey，并且在数据文件中写入有关tkey的词条信息，返回成功;

否则 转③

③ 将current_node分裂为left_node和right_node两个结点;

然后把tkey插入适当位置，重设left_node和right_node的head.NoOfkeys;

right_node.head.ParentLoc := current_node.head.ParentLoc;

若 current_node.lnk[1] > 0,

则 把current_node中属于right_node中关键码所链接的子结点的head.ParentLoc都标记为right node;

④ tkey := current node.key[current node.head.NoOfKeys/2];

current node = BTree.index[current node.head.ParentLoc];

转②

(3) B树的删除算法delete_key

① 用search key找到删除关键码tkey的位置. 这个位置在current node中;

② 删除tkey及其在数据文件的词条信息;

③ 执行 current_node.head.NoOfKeys := current_node.head.NoOfKeys-1;

若 current_node.head.NoOfKeys >= current node.head.MinKeys,

则 返回成功;

否则 转④

④ 判定current_node是否有这样一个兄弟结点，将其标记为brother_node,

若 满足brother_node.head.NoOfKeys > brother_node.head.MinKeys;

则 从brother_node中移出一个关键码到current node中,

同时这个关键码所连的在结点的head.ParentLoc置为brother node,

并返回成功;

否则 转⑤

⑤ 把current_node与brother_node合并为一个新结点，将其标记new_node，
若 brother_node为左兄弟，
则 将在其双亲结点中与brother_node相连的关键码下移至new_node中，
并把current_node的所有子结点的head.ParentLoc改为brother_node，
删除current_node，
current_node := BTree.index[brother_node.head.ParentLoc];
否则 将在其双亲结点中与current_node相连的关键码下移至new_node中，
并把brother_node的所有子结点的head.ParentLoc改为current_node，
删除brother_node，
current_node := BTree.index[current_node.head.ParentLoc];
转③

(4) B树的遍历算法traverse

B树的遍历算法采用深度优先的过程，得到的是排好序的关键码集合。

4 实现动态管理及调试环境

当整个索引构建好之后，我们的目标是在这个基础之上建立一个动态调试环境。当在调试过程中遇到确信是有关词典的错误需要人工干预的时候，我们可以动态的修改词典，继续被中断的调试过程，在当前断点处进行重试，直至满意或放弃为止。

我们在Windows窗口环境中实现了这个功能，并将它成功地应用于一个实验型的汉语句子分析系统。最初，先开发了一个单独的词典管理环境，它以汉语词汇作为关键码，有查询、新增、删除及修改等功能，在窗口中列出了有关的词法、语法、语义信息及相应的消歧规则；然后，我们在此基础上加入一层控制，使其连入动态的调试跟踪过程。具体做法是这样的，可以在一句句子中用鼠标标黑一个词，然后单击鼠标右键，便激活了汉语词典管理窗口，接着就可以在这个窗口中查询修改；最后，可以将修改过后的词进行重试。

在我们这个系统中，语料中的句子是成批处理的。在启动系统时，可以设定是否需要单步调试。若设定了单步调试功能，那么当处理完一句句子后，词法和句法分析的结果都会显示在屏幕上，研究者可以据此判定分析结果是否满意，然后决定是继续处理下一句句子还是激活动态调试环境(包扩规则库及词典)进行修改后重试该句。

下面举一个简单的例子来说明基于词典的动态调试流程。
如句子：他想的不是这样的。

① 设定系统处于跟踪状态；
② 经过词法与句法的分析，分词结果与形成的句法树显示在屏幕上；
③ 注意到分词结果为“他/想/的/不是/这样/的”，显然，这个结果把“不是”当作了名

词，而正确的结果应是把“不是”切成“不”和“是”两个词，虽然在句法分析的时候还是对此作出了正确的分析，但如果这个问题能够在分词阶段就能解决，那我们就决不把它留到以后的阶段去处理。于是用鼠标标黑“不是”这个关键词，按一下鼠标右键，激活词典管理窗口；

④ 词典管理窗口列出了有关“不是”这个词的所有信息，注意到在规则这一栏里显示为“pred(‘-1, saux’)→ ‘不是’ ”，这个规则说明“不是”的前一个词为结构助词时，“不是”为一个词，对本句例句而言，这个规则是不完善的，问题就出在这儿，于是修改这个规则为“pred(‘-1, saux’ and not ‘-2, v’) → ‘不是’ ”并且存盘，这个新规则表示当“不是”的前一个词为结构助词而前面第二个词不为动词的时候，“不是”为一个词，否则，将按最长匹配法切成两个词；

⑤ 关闭词典管理窗口，在分析窗口的按钮中选择“重试当前句”，那么系统将重新分析该句，得到正确的分词结果，然后继续处理下一句。另外需要注意的是，对机器词典采用动态管理技术，并不仅仅是为了调试方便，我们还可以把它集成入用户词典管理中，用户可以动态地登录、修改他的词典，扩充系统的处理能力。

5 结论

应用机器词典的动态管理技术，可以实时的调整词典，以符合分析处理的需要，而且并不带来查找速度上的损失。实践也证明，这对缩短整个系统的开发周期是很有效的。但是，由于查找速度对整个系统的效率都至关重要，所以在定型的系统中，还是应该采用快速的静态索引算法。

参 考 文 献

[1] D.E.Knuth 《计算机程序设计技巧》 Vol.3 《分类和检索》 国防工业出版社 1980

[2] 张永奎 《机器可读词典的快速查找技术》 中文信息学报 1994 Vol.8 No.2