Lab5 词典压缩

10235501419 李佳亮

本次实验采用Java语言，首先用**按块存储+前端编码**的方式对词典进行了压缩；后面又用了**前缀树+位图压缩**的方式对词典进行了压缩，压缩率更高，并给出了**如何将压缩后的词典用于倒排索引**，即实现词项指针+取词操作。

按块存储+前端编码

由于词的长度都小于256，那么我们可以只用1字节来记录长度（在Java中用`byte`类型）。

假设k=4，约定这个块压缩后的格式为：

```
[公共前缀的长度, 1B] [词1的长度, 1B] [词1, ?B] [词2后缀的长度, 1B] [词2后缀, ?B]
                                                [词3后缀的长度, 1B] [词3后缀, ?B]
                                                [词4后缀的长度, 1B] [词3后缀, ?B]
```

如，对块{automata, automate, automatic, automation}，可编码为：

```
[7] [8] [automata] [1] [e] [2] [ic] [3] [ion]
```

而对整个词典文件，在压缩后的文件的头部占用1个字节来写入k的值，便于解压时识别k的值。

```
[k的值, 1B] [块1, ?B] [块2, ?B] ...
```

压缩流程如下：

- 向文件写入1字节，表示k的值；
- 不断从词典中读取k个词（最后一块可能小于k）：
  - 计算k个词的公共前缀长度lcp；
  - 按格式写入文件。

解压函数只需要根据压缩后的格式来解码即可，在此不赘述，代码实现见附录。

具体在用Java语言实现中，写入文件我们使用的是`ByteArrayOutputStream`以字节为单位向文件写入的。`ByteArrayOutputStream.write()`仅接受`byte`类型的数组。int型数字会自动隐式转换为byte类型，而对于字符串（单词），需要调用`String.getBytes(StandardCharsets.UTF_8)`来把字符串转换为`byte`型数组。

压缩率：不考虑词项指针的开销，压缩后的词典文件大小502 KB，压缩率约为31%。

扩展：用位图表示的Trie来压缩词典，并实现“词项指针”

（1）压缩方法

前缀树是一种能够高效存储词典的数据结构，在Lab1中曾经用过该数据结构。Trie是一种多叉树，利用字符串的公共前缀来减少无谓的字符串比较以达到提高查询效率的目的。

如果只看结点占用的空间，那么它显然会比前端编码节省更多空间来存储字母。可是，如果用4字节的指针来表示边，边占用的空间是很大的。

为了优化指针占用的空间，如果是二叉树，我们可以用一个数组来表示这棵树，每个节点的孩子节点都可以用一公式算出。但是，用数组存储Trie这种多叉树又要额外耗费一定的空间来存储子节点在数组中的起始下标与孩子数量。

下面用到的Succinct数据结构是一种通过位图表达数据结构的技术，它能够很好地存储多叉树这一数据结构。下面我们用这一数据结构来压缩前缀树。

具体来说，该**压缩前缀树算法**输入一棵前缀树T，输出一个位图以及一个标签数组。其规则为：

- **输入**：前缀树T。
- **初始化**：位图b与标签数组l。
- 广度优先遍历树T，对于结点$V \in T$：
  - 设V有N个孩子结点，在b中追加$\underbrace{11\ldots10}_{N\text{ 个 }1}$来表示该结点；
  - 在l中追加结点V所存的字母。
- **返回**：位图b，标签数组l。

将位图b和标签数组l按下面的结构存入二进制文件，就可以得到压缩后的词典文件，其格式如下：

```
[位图b的长度，4B] [位图b，?B] [标签数组l的长度，4B] [标签数组l，?B]
```

（2）如何实现词项指针并取词？

下面需要思考的是，压缩后的Trie树怎么作为倒排索引的词典？也就是说，如何通过一个“指针”取到词典中的词？这一问题可分为两步走：①为词典中的每个词分配指针； ②通过指针在词典中取完整的词。

具体来说，我们需要用到下面的辅助函数：

```
// x表示比特位的位置，即位数组的下标，从0计
// y从1计
rank1(x)        // 返回在 位图[0, x] 里面 1 的个数
select1(y)      // 返回第 y 个 1 所在的位置
rank0(x)        // 返回在 位图[0, x] 里面 0 的个数
select0(y)      // 返回第 y 个 0 所在的位置
```

根据这些辅助函数，我们可以得出一些有用的公式帮我们查找这棵“树”（位串）。我们将每个结点赋予一个编号$\text{idx} \in \{0, 1, 2, \ldots\}$，按广度优先遍历的顺序。

```
// 1. 结点idx在位图中的起始索引（idx为0时，根结点的startBit = 0）
startBit = select0(idx)
// 2. 位图中起始索引为x所归属的结点编号（前提是该节点有孩子节点）
idx = rank0(startBit)
// 3. 若位图中的x处是1，那么rank1(x)就表示这个1所表示的孩子结点的编号
childIdx = rank1(x)
    // 3.1：结点idx的第i个子节点编号
    child_i_Idx = rank1(select0(idx) + i - 1)
// 4. 结点idx的父节点在位图中的起始索引
fatherStartBit = select1(idx)
    // 4.1：（结合2，4）结点idx的父节点的编号
    fatherIdx = rank0(fatherStartBit) = rank0(select1(idx))
// 结合上面的四条公式还可以导出其他复合公式
```

于是，借助上面的公式，我们可以得到计算词典指针的算法以及通过指针取词的算法。

计算词典“指针”的算法：输入原词典和压缩后的位图，输出词典压缩为位图后各个词的结尾字母在树中的结点的编号。通过这一编号，我们可以根据上面的公式（4.1）一路寻回完整的单词。（在这里，一个词的`指针`，其实是指各个词的结尾字母在树中的结点的编号。）

- **输入**：原词典D，压缩后得到的位图b。
- **初始化**：大小为$D.size$的整数数组ptr。
- 遍历词典D中的每个词w：
  - 初始化$curIdx$为根结点；
  - 遍历w中的每个字符c：
    - 从$curIdx$的孩子结点中，找到表示c的那个结点idx；（利用公式1，3）
    - $curIdx \leftarrow idx$；
    - continue。
  - $ptr[w] \leftarrow curIdx$。
- **输出**：ptr。

通过“指针”取词的算法：输入压缩后的位图、标签数组以及某一词的“指针”，返回该指针指向的完整单词。

- **输入**：压缩后的位图b，标签数组l，“指针”ptr。
- **初始化**：$\text{StringBuffer}\ sb$。
- $\text{while}\ ptr > 0:$
  - $\text{sb.append}(l[ptr])$;
  - $ptr \leftarrow ptr$ 父节点的编号；（利用公式4.1，$ptr \leftarrow \text{rank0}(\text{select1}(ptr))$）
- **输出**：$\text{sb.reverse}().\text{toString}()$。

（3）压缩率

位图b所占空间为57,851 bytes，标签数组l所占空间为231,401 bytes，指针数组ptr所占空间为351,920 bytes。如果不考虑词项指针的存储空间，合计289,252 bytes，约为283 KB。而原文件大小为726 KB，压缩率可以达到61%。

（4）再进一步优化的思路

可以看到，标签数组l占据了较大的空间。而标签数组所存的元素是26个字母，那么我们可以对这26个字母进行编码，如哈夫曼编码。

如果要优化词项指针的存储空间，可以让倒排索引表不按照字典序来排，而是按照单词结尾在前缀树中的编号来排，那么也可以用间隔编码，这样就可以用更少的字节来表示这个间隔。

附录（GitHub链接）

- FrontCoding.java, FrontCoding.bin - 按块存储+前端编码方式代码 及 压缩后文件；
- SuccinctTrie.java, SuccinctTrie.bin - 位图压缩后的Trie方式代码 及 压缩后文件。