RRR方法实现rank操作

# 简介

RRR方式是由R.Raman，V.Ramna以及S.Srinivasa Rao等人于2002年提出的一种静态的字典结构。通过这种结构可以实现对01二元序列的常数时间的rank&select操作，并且采用同一方法即扩展到对多符号序列的常数时间的*rank&selec*t操作。在二元01序列上，RRR方法实现*rank*操作只需要*nH*0 + *o*(*n*)的空间，而常用的Jacobson的*rank&select*方法则需要*n* + *O(n* loglog *n*/log *n)*的空间。可以看到在01序列中，如果0和1的几率相等时，二者的空间占用相差不大，而若0和1出现的几率并不相等，其中一个出现的几率远大于另一个时，RRR方法的空间占用将小于*n*比特。而在压缩后缀数组的多层结构实现中，需要维持一个由01序列构成的采样点的字典结构，该字典需要实现*rank&select*操作，且字典中的0远多于1，在这一应用场景中，采用RRR方法要优于Jacobson的方法。

本文介绍RRR方法的理论方法和具体实践方法，并给出RRR方法的实验结果和Jacobson方法的结果对比。

# RRR方法详解

## 2.1 RRR方法的rank操作实现

RRR方法和Jacobson的方法在目录结构上类似，都采用了分块的方法和两层目录结构。具体做法是首先把长为*n*的二元串*B*分长为*s* = log *n2*的大块*S*1, *S*2, …, *Sn/s*。之后每一个大块再分成长为*b =* log *n*/2的小块*Bi*(*j*)。这两种划分方法在RRR中和Jacobson的方法是一致的，所不同的是之后的处理。

对于每一个小块*Bi*(*j*)在Jacobson的方法中是显式直接存储的，而在RRR方法中则采用了一个(*c*, *o*)对替代，其中*c*表示*Bi*(*j*)这个小块中的1的个数，即这个数的类别，而*o*表示*Bi*(*j*)在所有的有*c*个1的长为*b*位的数中的名次。显然，对于第*c*类数，总共有个，而*c*的最大值为*b*，所以一个(*c*, *o*)对需要的存储空间是log(*c* + 1) + log位。从这里可以看出，相对于原来的*b*位的原始串，采用一个(*c*, *o*)对替代后，所需空间是随原始串中1的个数变化的，1的个数越少（即*c*越小），所需要的存储空间也会小于相应的变小，而且就整体而言，一个(*c*, *o*)对所占的空间也是小于*b*位的。这就是RRR方法的优势所在。对于每一类*c*中的每一个(*c*, *o*)对，都可以预先处理得到这个(*c*, *o*)对对应的长为*b*的01串的每一位的*rank*值，并保存为*Gc*表，总共需要*b*log(*c* + 1)位的空间。所有的*Gc*表组合起来就构成了我们预处理得到的表*G*，总共需要位的空间。

通过上面叙述的方法，可以把每一个小块*Bi*(*j*)变换成一个(*c*, *o*)对，表示为*Di*(*j*)，并且*Di*(*j*)总共需要log(*c* + 1) + log位的空间，累加所有的小块对应的(*c*, *o*)对所需要的空间，前面一项累加后为*O*(loglog *n*)位，后面一项累加起来为*nH*0位。把所有的变长的*Di*(*j*)连接起来构成一个单独的表，即*D*表，所需空间为*nH*0 + *O*(loglog *n*)位。

对于每一个大块*Si­ ,*对应存储一个指针*Pi*指向这个大块的第一个小块对应的(*c*, *o*)对在*D*表中的位置，即*Pi* = *Di*(0)，并且*Ri*存储这个大块对应的第一位的*rank*值，即*Ri* = *rank*((*i* −1)⋅*s*)*。P*表和*R*表总共需要的空间是*O*(*n*/log *n*)位。同样的对应每一个属于大块*Si*的小块*Bi*(*j*)，也存储一个指向其对应的(*c*, *o*)对在*D*表中的位置*Li*(*j*)，只是*Li*(*j*)是该位置相对于所在大块位置的相对位置，即*Li*(*j*) *= Di*(*j*) − *Pi*。同样的保存每一个小块*Bi*(*j*)的第一位的*rank*值*Qi*(*j*)，当然也是相对于所在大块的*rank*值，即*Qi*(*j*) *= rank*((*i* −1)⋅*s* + (*j* −1)⋅*b*) − *Ri*。由于这些相对量的最大值都是log *n*，所以*L*表和*Q*表总共需要*O*(*n* loglog *n*/log *n*)位的空间。

在求解任意的*rank*(*p*)时，首先计算第*p*位对应的大块的编号*i* *=* *p/s*,以及小块编号*j* = (*p* − (*i* − 1)⋅*s*)/*b*。之后，加上对应的大块对应的*rank*值*Ri*和小块对应的相对*rank*值*Qi*(*j*)。再根据*Pi*和*Li(j)*的值可得到这个小块对应的(*c*, *o*)对的值*Di*(*j*)，通过访问*D*表的*Di*(*j*)位置，即可得到第*p*位所在小块的每一位的*rank*值，加上前面得到的相对值即可得到最终的*rank*值。

上面即为RRR方法的实现，总共需要保存*D，P，R，L，Q*五个表，总的空间需求是*nH*0 + *O*(*n* loglog *n*/log *n)*位，可以在常数时间内实现了*rank*操作。

## 2.2 RRR方法的实现方法

上面的是RRR的原始论文中的实现方法，这种方法有比较好的理论基础，但实际上过于复杂，并不适合具体的实现。而且两层目录，四个表最终所占的空间会很大，实际的空间效率并不是很具优势。故在实现中一般可以做一些简化处理，从而得到更好的空间效率。具体的做法是，把小块的长度*b*固定为15，这样每一个*c*都恰好只需一个4位的整数来表示，*o*的保存方法不变，对于*D*表，只需要把所有的15位的整数重新按类别*c*排序，同一类的数放在一个桶内，所有的桶链接成一个表，这个表实际上存储了215个16位的整数，总共只需要64KB的空间。*D*表放弃了保存每一个*b*位长的整数的每一个*rank*值，这导致最终需要平均4次操作才能得到任意一个位的*rank*，相对于原始论文中的方法，时间效率有所下降。但空间效率得到了改善。

同样的为了减少空间占用，我们把两层目录改为单层目录，每一个大块的长度可为15的整数倍*k*，即每一个大块中包含*k*个小块，对每一个小块保存两个表，*R*和*S*。其中*R*表保存每一个小块的类别*c*，即这个小块有多少个1；*S*表保存每一个小块对应的*o*值，即这个小块在所述类别中的位置。明显的*R*表总共需要4⋅*n*/15 = 0.25*n*位的空间，而*S*表总共需要位。有了*R*和*S*表，只需顺序访问*R*表即可得到对应小块对应的*c*，经过累加计算访问*S*表可得到*o*，再根据(*c*, *o*)对访问*D*表即可得到对应小块的真实01序列。但要求得任意位置的*rank*值，则还需加上两个辅助结构，*sumR*和*sumS* 表。其中*sumR*[*i*]保存的是第*i*个大块的第一位所在位置的*rank*值，即*sumR*[*i*] *=* *rank*(*i⋅*15⋅*k* − 1)，*S*表保存每一个大块的第一个小块对应的(*c*, *o*)对的*o*值所在在*S*表中的偏移量。

对于要求的任意的*rank*[*p*]，首先计算其所在大块的编号*i* = *p*/(*k⋅*15)，再计算其所在小块的编号*j* = *p*/15。之后我们即可计算出*p*所在小块的第一位的*rank*值为，其中*R*[*t*]为第*t*个小块对应的*c*值，这个值可以在*R*表上常数时间内得到。有了这个相对值，借助于即可得到对应小块的*o*值在*S*表上的偏移量，访问*S*表即可得到*o*值，接着利用这个(*c*, *o*)对访问*D*表可以得到这个小块的原始01序列，借助*popcount*指令，可以在常数时间内获得这个小块内任意位置的*rank*值，和之前的相对*rank*值相加即为最终的*rank*值。总的时间开销会大于Jacobson的方法，比原始RRR论文中的方法也慢，但总的空间效率则得到了极大的提高。

# 实验结果

通过上述的理论分析，也可以看出RRR方法空间效率是其优势所在，故本实验着重测试了RRR方法的空间占用，并且和Jacobson的方法在同一测试平台上进行了实验结果对比。测试数据是自己构造的01序列，其中1的出现几率为10%。数据的规模最小的是64M到1G的01序列。结果如下所示。

RRR方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据规模(M) | rank结构大小(B) | Bit per symbol | Rank时间(us) | Get时间(us) |
| 64 | 5557352 | 0.6625 | 1.46 | 1.46 |
| 128 | 11064000 | 0.6595 | 1.47 | 1.45 |
| 256 | 22092224 | 0.6584 | 1.47 | 1.46 |
| 512 | 44178496 | 0.6583 | 1.52 | 1.46 |
| 1024 | 88410688 | 0.6583 | 1.52 | 1.46 |

上表中的rank结构大小指最终的RRR实现的rank数据结构的大小。bit per symbol表示经过RRR变换后，每个bit所占的位数，实际就是压缩率。Rank时间指在RRR结构上完成一次rank操作的平均时间，get指获取任意原始串的任意位的值的时间。//你这里的per symbol bit，是指01序列平均每位占的空间吗？rank结构如果用M为单位，为输入大小的多少？

Jacobson方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据规模(M) | rank结构大小(B) | Bit per symbol | Rank时间(us) | Get时间(us) |
| 64 | 1.927e+7 | 2.297 | 0.536 | 0 |
| 128 | 3.6148e+7 | 2.297 | 0.536 | 0 |
| 256 | 7.22991e+7 | 2.1546 | 0.543 | 0 |
| 512 | 1.446e+8 | 2.155 | 0.543 | 0 |

从以上两表中的实验结果可以看到随着数据规模的增大，两种方法的压缩率都趋于平缓，RRR方法保持在0.66左右，而Jacobson的方法则保持在2.155。对于*rank*而言，RRR的方法因为有过多的累加操作，平均求值时间几乎是Jacobson的3倍，但相对的，RRR方法的空间占用也比Jacobson的方法减少了2倍多，达到了真正的压缩效果。

# 总结

通过对RRR方法实现并和Jacobson的方法做实验对比，可知，在01序列并不均匀时，采用RRR方法具有明显的空间优势。而压缩后缀数组的实现中所用得到的支持*rank*操作的字典则都是01序列不均匀分布的，所以在CSA中采用RRR方法实现rank操作会有更好的空间效果。