高阶熵压缩的全文自索引

霍红卫 陈晓阳 陈龙刚 于 强
(西安电子科技大学计算机学院 西安 710071)

摘 要 大数据集正在以前所未有的速度产生,研制大数据集的实用压缩全文自索引是目前的挑战问题之一.该文提出了一种高阶熵压缩的全文自索引.对于长为 n 的文本 T 以及任意 $k\leq c\log_\sigma n-1$ 和 $c<1$,该压缩索引占用 $2nH_k(T)+n+o(n)$ 位的空间,其中 $H_k(T)$ 表示文本 T 的 k 阶经验熵,σ 为字符表的大小.此外,该压缩索引可在线性时间 $O(n)$ 内构造.在此基础上,该文还给出了上述压缩索引的一种实用改进.这种改进引入了混合编码方法,额外的空间开销为 $o(n)$ 位.对于 *Pizza &Chili Corpus* 上的三类典型数据的实验表明:该文的压缩索引较之主流压缩索引在压缩率和查询时间上具有显著的优势.该文所述的压缩索引软件可在 GitHub 上访问.

关键词 大数据;压缩索引;自索引;高阶熵;混合编码
中图法分类号 TP18 **DOI 号** 10.11897/SP.J.1016.2016.02494

High-Order Entropy-Compressed Full-Text Self-Indexes

HUO Hong-Wei CHEN Xiao-Yang CHEN Long-Gang YU Qiang
(*School of Computer Science and Technology, Xidian University, Xi'an* 710071)

Abstract Big data sets are being produced at unprecedented rates. Developing a practical compressed full-text self-index for big data sets remains one of the most challenging problems. In this paper we present a high-order entropy-compressed full-text self-index. For a text of n characters, our compressed full-text self-index occuples $2nH_k(T)+n+o(n)$ bits of space for any $k\leq c\log_\sigma n-1$ and any constant $c<1$, where $H_k(T)$ denotes the kth order empirical entropy. In addition, our compressed indexes can be constructed in linear time. We also give a practical improvement on the compressed index described above. Our practical improvement introduces hybrid encoding methods. The extra cost needed is $o(n)$ bits. Experiments on three typical data sets on the *Pizza &Chili Corpus* show that our compressed indexes have significant advantages in terms of compression and query time over the state-of-the-art indexing technologies. The source code is available online.

Keywords big data; compressed indexes; self-indexes; high-order entropy; hybrid encoding

1 引 言

大数据正以前所未有的速度产生.现实世界的数据源如 WWW 数据、社交网络、基因组数据、音乐数据、数字图像和视频、流媒体等均产生数据,其中相当一部分结构化为字符串或文本序列.据 IDC 公司估计[1],到 2017 年后,全球互联网协议流量每年将突破 1.4 ZB(1 ZB=10^{21} Byte),到 2018 年全球移动数据流量每月将达到 15.9 EB(1 EB=10^{18} Byte).

收稿日期:2015-01-25;在线出版日期:2015-06-29.本课题得到国家自然科学基金(61173025,61373044)资助.**霍红卫**,女,1963 年生,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为算法设计与分析、大数据压缩索引与检索、压缩数据结构、外存算法、生物信息学算法,算法工程.E-mail: hwhuo@mail.xidian.edu.cn.**陈晓阳**,男,1991 年生,博士研究生,主要研究方向为大数据压缩索引与检索、图索引、外存算法.**陈龙刚**,男,1988 年生,硕士研究生,主要研究方向为压缩索引与检索.**于 强**,男,1983 年生,博士,讲师,主要研究方向为生物信息学算法、并行算法.

本文所述压缩索引网址:https://github.com/chenlonggang/Compressed-Suffix-Array 和 https://github.com/chenlonggang/Adaptive-CSA.

传感网和物联网的蓬勃发展是大数据的又一推动力，各个城市的视频监控每时每刻都在采集巨量的流媒体数据.大数据的不断激增为数据存储、管理、检索和从数据中搜索可用信息提出了新的挑战[2].

Gartner 提出了大数据的 3 个主要特征[1]：海量(Volume)、多样性(Variety)及速度(Velocity).海量是指数据量大，使用传统基础设施进行处理的难度变大.多样性是指数据类型、表现形式和语义解释的异质性.过去的数据更多的是结构化的数据.现在越来越多的数据是半结构数据，甚至是完全没有结构的数据.据 IBM 大数据的报告，2013 年世界上的数据量大约是 1.6ZB.在这 1.6ZB 的数据中，约 80% 的数据是无结构的数据，因而，存在很多数据源是传统技术不能管理或分析的.速度有两层含义：一是数据到达的速率；另一个是必须对数据处理的速率.

模式匹配是理论计算机科学最古老的研究领域之一[3-4]，是理论领域为许多应用领域提供切实可行解决方案的典范.从 KMP 算法[3](几乎所有算法教科书中最重要的内容之一)到 Boyer-Moore 算法[4](所有文本编辑器中搜索命令的核心)，由于其在信息检索、Web 数据挖掘、计算生物学和图像处理等领域的应用，一直得到广泛关注.

模式识别从一开始就是设法解决大型异质数据集的搜索问题.如果是在一个小型的文件中搜索某个特定的单词，是不需要什么特殊算法的.然而，如果要在人类基因组中查找某个基因，而基因组中没有单词分隔界限，而且数据规模和搜索的模式都很大，那么理解如何重用扫描过的数据就变得至关重要.因此，上述的 KMP 算法[3]和 Boyer-Moore 算法[4]的重要性就体现出来了.在出现更为复杂的问题时，例如，给定一个大型非文本数据库(如数值数据、音频数据或生物序列)，我们能够索引它并能快速回答搜索查询吗?正是因为这个问题，所以发明了后缀树[5-6]和后缀数组[7]这样的数据结构.

虽然后缀数组和后缀树均能在最优或几乎最优时间内支持模式匹配查询，但对于输入规模为 n 的文本 T，它们使用 $O(n)$ 字的空间，即 $O(n\log n)$[①]位.在空间复杂度方面，这个大小要比文本 T 自身的大小 $O(n\log\sigma)$ 位要大得多，其中 σ 为字符表的大小，且在实际中这些数据结构所占空间为原文本的 5～20 倍[2].

1.1　已有相关研究工作

压缩后缀数组[8-13](Compressed Suffix Array，CSA)和 FM-Index[14-16]利用了文本的可压缩性和规则性，克服了空间上的局限性，同时能支持后缀数组和后缀树的功能.压缩后缀数组和 FM-Index 是不需要原始文本的自索引(self-indexing)，也就是说，它们既能作为压缩的原始文本，也能作为索引，原始文本可以抛弃.从实际的角度来看，这是首次获得的模式匹配的空间高效的压缩索引方法.此外，这些索引技术在空间和时间上还能与搜索引擎中所使用的著名的倒排索引(inverted indexes)技术[17-18]相媲美.因为倒排索引是一种词索引技术，适合做关键字搜索，不适合短语搜索；如果用倒排索引进行短语搜索效率不高[19]，因而这些索引则提供了更为一般的搜索能力.

Grossi 和 Vitter[8-9]首次建立了压缩后缀数组理论(Compressed Suffix Array，CSA)，使用 $O(n\log\sigma)$ 位的空间，可在 $o(m/\log_\sigma n+occ\log_\sigma n)$ 时间内支持模式匹配查询，其中 m 为模式 P 的长度，occ 表示 P 在文本中出现的次数.在 Grossi 和 Vitter[8-9]的 CSA 中，后缀数组的值间接编码，以 Φ 函数(又称近邻函数[8-9])存储，其中 $\Phi(i)=SA^{-1}[SA[i]+1]$.Φ 函数可压缩为熵含义下的最优空间，每个 SA 的值可在 $O(\text{polylog}\,n)$ 时间使用 Φ 函数的一小部分值计算出.Sadakane[12-13]给出了将 Grossi 和 Vitter 的 CSA 变为自索引的方法，该索引使用 $(1/\epsilon)nH_0+O(n\log\log\sigma)+\sigma\log\sigma$ 位的空间，可在 $O(m\log n+occ\log^{\epsilon}n)$ 时间内支持模式匹配查询.Ferragina 和 Manzini[14-15]提出了基于 BWT(Burrows-Wheeler Transform)变换[20]的 FM-Index.其索引至多使用 $5nH_k+o(n\log\sigma)$ 位的空间，$k\leq\log_\sigma(n/\log n)-\omega(1)$，可在 $O(m+occ\log^{1+\epsilon}n)$ 时间内检索 P 在文本 T 中的出现次数 occ，其中 $0<\epsilon<1$，H_k 表示文本 T 的 k 阶经验熵，$k\geq 0$.

Grossi 等人[10]提出了理论上可证明达到渐近空间最优性的第 1 个自索引，即高阶项前的系数为常数 1.该自索引使用 $nH_k+o(n)$ 位的空间，达到了 $O(m\log\sigma+\text{polylog}\,n)$ 的查询时间.同样的分析也适用于 FM-Index，因而 FM-Indx 也达到了这样的最优性.Foschini 等人[21]给出了压缩后缀数组的一种新的存储结构，空间达到了熵界；他们的方法不仅保持了以往工作理论上的性能，且在实际中显示了良好的结果.Kärkkäinen 和 Puglisi[22]提出了一种固定块压缩提升技术，类似于上下文块提升技术，将 BWT 变换分成固定大小的块，而不是按照字符表

① 本文中不特别指明情况下，对数都以 2 为底.

的分块.虽然这种划分不是最优的,但他们表明了这种技术不会比最优划分差很多.这种对 BWT 变换 L 进行固定块划分的技术,其数据结构较为简洁、快速.Mäkinen 等人[23]提出了一种针对高度重复序列的压缩自索引,称为 RLCSA.他们提出了后缀数组采样的一种新策略.实验结果表明 RLCSA 对于高度重复 DNA 序列如酵母序列表现出了良好的性能.Huo 等人[24]提出了表示 CSA 的近邻函数 Φ 的一种新结构,理论上空间保持了高阶熵,设计了一种查找表支持快速解码.Ferragina 等人[25]从实践角度将已有索引进行了实现,并做了对比.Navarro 和 Mäkinen[26]综述了这方面的成果.Gog 和 Navarro[27]扩展了 CSA 的模式定位功能.Gog 和 Petri[28]实现了一个包含 GGV-CSA 和 FM-Index 的简明数据结构库.

1.2 本文的贡献

本文提出了一种基于压缩后缀数组(Compressed Suffix Array,CSA)的高阶熵压缩的全文自索引.对于长为 n 的文本 T,我们的压缩索引可在线性时间内构造.对于任意 $k\leq c\log_\sigma n-1$ 和 $c<1$,我们的压缩索引占用 $2nH_k+n+o(n)$ 位的空间,其中 H_k 表示文本 T 的 k 阶熵,σ 为字符表的大小.此外,本文还给出了上述描述的压缩索引的一种实用改进,称之为 ACSA(Adaptive CSA).这种实用改进引入了混合编码方法,额外的空间开销为 $o(n)$ 位.对于 *Pizza & Chili Corpus* 上的三类典型数据的实验表明,我们的压缩索引较之主流压缩索引在压缩率和查询时间上具有显著的优势.所开发的压缩索引可在 GitHub 上访问:https://github.com/chenlonggang/Compressed-Suffix-Array 和 https://github.com/chenlonggang/Adaptive-CSA.

2 预备知识

2.1 问题定义和符号含义

文本索引问题:给定大小为 n 的文本 T,其中字符取自大小为 σ 的字符表 Σ.文本索引的目标是构建 T 的一个索引,使得对于任一查询模式 P,能够高效地确定 P 是否在 T 中出现.对于具体应用问题,我们可能希望确定模式 P 在 T 中的出现次数 occ;或者定位模式 P 在 T 中出现的所有位置;或者展示文本 T 的子串 $T[s,s+len-1]$,给定子串起始位置 s 和长度 len.在压缩空间,本文索引的目标是构建空间占用逼近文本熵压缩大小的压缩索引,同时支持快速查询性能.

表 1 中给出了本文中使用的符号及含义.

表 1 文中使用的符号及含义

符号	含义
T	文本 T
n	文本 T 的长度
P	模式 P
m	模式 P 的长度
Σ	字符表$\{\alpha_1,\cdots,\alpha_\sigma\}$
σ	字符表的大小
Σ^k	定义在字符表 Σ 上长为 k 的串集
SA	T 的后缀数组(suffix array)
$T[i]$	T 中第 i 个字符
$T[i..n]$	T 的第 i 个后缀,$1\leq i\leq n$
$SA[i]$	第 i 个字典序最小后缀在 T 中的起始位置
$SA^{-1}[i]$	后缀 $T[i..n]$的排名.称 SA^{-1}为逆后缀数组
CSA	压缩后缀数组(Compressed Suffix Array)
$\Phi(i)$	近邻函数,满足$\Phi(i)=j$, if $SA[j]=(SA[i]+1)\bmod n$
x-list	以 x 开始的所有后缀位置的一个递增序列
$H_k(T)$	T 的 k 阶经验熵,简称 k 阶熵
n_i	字符 i 在 T 中出现的次数
w_s	连接 w 在 T 中每次出现之后的字符所形成的串,w 为长为 k 的子串
k-context	T 中后缀的长为 k 的前缀
$C[x]$	T 中所有小于 x 的字符出现次数的总和
occ	模式 P 在 T 中的出现次数
$count$	确定模式 P 在 T 中的出现次数
$locate$	确定模式 P 在 T 中出现的所有位置
$extract$	返回 $T[start..start+len-1]$,给定 $start$ 和 len
polylog n	n 的对数多项式

2.2 后缀数组

令 $T[1..n]$为长度为 n 的文本,其中字符取自大小为 σ 的字符表 Σ.形如 $T[i..n]$的 T 的子串($i=1,2,\cdots,n$)称为 T 的后缀.后缀数组(suffix array)$SA[1..n]$是 n 个元素的整型数组,$SA[i]=j$ 表示第 i 个字典序最小的后缀在 T 中的起始位置为 j.类似地,我们可以定义逆后缀数组:$SA^{-1}[i]=j$,表示后缀 $T[i..n]$的排名为 j,即 T 中由 i 起始的后缀的排名为 j.如果模式 P 出现在文本 T 中,那么存在整数 L 和 $R(L\leq R)$,满足 $SA[L],SA[L+1],\cdots,SA[R]$存储了 P 在 T 中出现的所有位置.

2.3 压缩后缀数组

Grossi & Vitter(GV-CSA)压缩后缀数组[8-10]解决了后缀数组作为索引时空间占用过大的问题,其核心是如何高效地表示近邻函数 Φ[8-10],定义如下:

$$\Phi(i)=j,\ \text{if } SA[j]=(SA[i]+1)\bmod n \quad (1)$$

Φ 函数将 SA 中的某个位置 i(满足 $SA[i]=p$)映射到另一个位置 j(满足 $SA[j]=p+1$),其核心在于把当前位置和下一个位置联系起来,不仅利用了上下文信息,且有利于压缩,又提供了通过 Φ 函数访问整个文本串的能力,从而提供了检索和数据

恢复的能力. Grossi 等人[10](GGV-CSA)所提出的达到渐近空间最优性的压缩自索引引入了 Σ-list 的概念(此后我们称 x-list),这些 x-list($x\in\Sigma$)可对文本中的所有后缀及其关联的 Φ 值按照其前缀进行划分. 通过对后缀指针按其长度为 2^k ($k=0,1,2,\cdots$)的前缀进行划分,可得这些 x-list. 这些 x-list 的简单连接恰好为近邻函数值. 每个 x-list 形成关于文本位置的一个递增序列,如图 1 所示. 其中标示出了 a-list 和 c-list. 因此,如果对递增序列的间隔长度进行编码,就能实现对文本的压缩.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
T	a	b	f	g	d	b	f	b	g	d	f	c	c	b	g	a	c	e	f	c	e	g	c	d	e	f	g	b	f	c	a	d	b	g	a	f
SA	0	15	30	34	5	27	1	13	32	7	29	12	11	22	16	19	4	31	23	9	17	24	20	35	6	28	10	18	25	2	14	33	26	21	3	8
Φ	6	14	17	23	24	25	29	30	31	35	2	7	11	18	20	22	4	8	21	26	27	28	33	0	9	10	12	15	32	34	1	3	5	13	16	19

a-list　　c-list

图 1　后缀数组 SA 和近邻函数 Φ

2.4　k 阶经验熵

令 T 为长度为 n 的文本串,其中字符取自大小为 σ 的字符表 Σ. 由信息论可得,使用经验熵(empirical entropy)可以界定存储文本 T 所需空间的下界. 经验熵类似于概率意义上所定义的熵,不同之处在于经验熵是根据所观察到的 T 中字符频率来定义的,而不是由字符概率来定义的. 可用经验熵来度量一个压缩算法的性能,它是文本串结构的一个函数,不对输入做任何假设. 文本串 T 的 0 阶经验熵定义为[29]

$$H_0(T)=-\sum_{i=1}^{\sigma}\frac{n_i}{n}\log\left(\frac{n_i}{n}\right) \tag{2}$$

其中:n_i 是字符 i 在 T 中出现的次数,$\sum_i n_i=n$. 值 $nH_0(T)$ 表示理想压缩器的输出大小,该压缩器使用 $-\log\left(\frac{n_i}{n}\right)$ 位对字符表 Σ 中的符号进行编码. 这是使用唯一可解码编码所能达到的最大压缩率,其中字符表中的每个字符被赋以一个固定的码字. 如果用于每个字符的码字依赖于其在 T 中的前 k 个字符,那么还可以达到更好的压缩率.

对于长为 k 的串 $w\in\Sigma^k$,令 w_s 是连接 w 在 T 中每次出现之后的字符所形成的串. $|w_s|$ 是该串的长度. T 的 k 阶经验熵定义为[29]

$$H_k(T)=\frac{1}{n}\sum_{w\in\Sigma^k}|w_s|H_0(w_s) \tag{3}$$

值 $nH_k(T)$ 表示如果使用的编码依赖于前 k 个最近所见的符号,所能达到压缩率的下界. 注意,对于任意串和 $k\geq 0$,有 $H_{k+1}(T)\leq H_k(T)\leq\log\sigma$.

例如,对于 $T=abcdabcdabcd$,其 0 阶经验熵和 1 阶经验熵如下计算:

$$\begin{aligned}H_0(T)&=-(1/4)\log(1/4)-(1/4)\log(1/4)-\\&\quad(1/4)\log(1/4)-(1/4)\log(1/4)=2,\end{aligned}$$

$$\begin{aligned}H_1(T)&=(1/12)(3H_0(w_a)+3H_0(w_b)+\\&\quad 3H_0(w_c)+2H_0(w_d))\\&=(1/4)H_0(bbb)+(1/4)H_0(ccc)+\\&\quad(1/4)H_0(ddd)+(1/6)H_0(aa)=0.\end{aligned}$$

上例 T 的所有高阶熵均为 0. 这表明如果我们随机均匀地从 T 中选择一个字符来猜测,其不确定性为 2. 如果猜测之前我们知道其前一个字符,那么可以肯定结果答案. 在一般情况下,对于给定的文本串 T,存在 N,满足对于 $k\geq N$,有 $H_k(T)=0$.

3　压缩后缀数组的存储结构

3.1　Φ 的简明表示

为了方便叙述起见,我们根据上下文交替使用 $\Phi()$ 函数和 $\Phi[]$ 数组这两种形式. GGV_CSA[10] 是一种多层递归结构,按照 k-context[10] 对 Φ 进行划分,达到了文本的高阶熵,且最高项系数为 1. 我们在这一理论的基础上,考虑实际,对此进行简化,并结合后向查找过程,提出了一种相对简明、易实现的单层 CSA 结构,同时空间上保持了理论上的高阶熵. 我们的结构采用两层模式:一层是超块($superblock$,SB),另一层是块($block$,B). 过程如下:

(1)首先把长为 n 的 Φ 数组分为长为 $a=(\log n)^2$ 的超块,故共有 n/a 个超块.

(2)其次把每个超块分成 a/b 个块,每个块包含 b 个整数,$b=(\log n)^2/\log\log n$.

(3)最后对每个块内的整数序列做差分,对差分后的每个差值进行 Elias Gamma 编码[30],连接起来即得到 Φ 数组的表示,记为 S.

但我们必须处理一种特殊情形. 假设 x 表示前一个 Φ 值,y 表示当前 Φ 值,当差值 $gap=y-x<0$ 时,Gamma 编码无法处理,此时令 $gap=y-x+n$,

再将该 gap 值进行 Gamma 编码. 这是由于$(x-z) \bmod n=(x+n-z) \bmod n$,其中 x 和 z 都是正整数. 在我们的结构中,需要保存超块的偏移量表 SB,块的偏移量表 B,Φ 的采样表 SAM 以及最终的 Gamma 编码序列 S,我们用这 4 个结构表示 Φ 数组.

图 2 给出了大小 $n=36$ 的 Φ 数组及其表示,其中超块长度 $a=9$,块长度 $b=3$,超块中的块数 $r=3$.

图 2 中每个 x-list($x \in \Sigma$)内对应的所有后缀都具有相同的首字符,如 a-list 内对应的所有后缀都具有相同的首字符 a. 每个 x-list 内的 Φ 值呈升序排列,但不同的 x-list 之间不存在这种关系. 表中的 gap 数组是各个块内求差分后的值,每个块内的第 1 个数会被采样,所以没有与之对应的"差值",表中这些位置标记为 0,实际编码的 gap 序列不包含块中的这些 0. 需要注意的是第 4、6、8 个块,因为这些块中的 3 个整数分属两个 x-list,所以出现了差分后为负值的情况,此时 $gap=\Phi[i]-\Phi[i-1]+n$,对于本例,第 4 个块的 $gap=2-35+36=3$,所以该 gap 编码为 011,对于第 6 个块,$gap=4-22+36=18$,因而对应的编码为 000010010.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Φ	6	14	17	23	24	25	29	30	31	35	2	7	11	18	20	22	4	8	21	26	27	28	33	0	9	10	12	15	32	34	1	3	5	13	16	19
gap	0	8	3	0	1	1	0	1	1	0	3	5	0	7	2	0	18	4	0	5	1	0	5	3	0	1	2	0	17	2	0	2	2	0	3	3
S	0001000011			11			11			01100101			00111010			00001001000100			001011			00101011			1010			000010001010			010010			011011		
SB	0									14									44									62								
B	0			10			12			0			8			16			0			6			14			0			12			18		
SAM	6			23			29			35			11			22			21			28			9			15			1			13		

图 2 Φ 的表示($n=36$,$a=9$,$r=3$,$b=3$)

Elias Gamma 编码[30]是一种前缀无关码,因此解码是唯一的. 理论上,对于一个大小为 x 的整数应用 Gamma 编码,需要 $2\lfloor \log x \rfloor+1$ 位表示这个整数. Gamma 编码由两部分组成,前一部分是 $\lfloor \log x \rfloor$ 个 0,后一部分是 x 的二进制表示,使用 $\lfloor \log x \rfloor+1$ 位. 解码时从左到右数出连续 0 的个数,不妨设为 c 个,然后解码 $c+1$ 位即得 x. 例如:$gap=8$,$\lfloor \log 8 \rfloor=3$,表示前一部分含有 3 个 0,由于 8 的二进制表示为 1000,即后一部分为 1000,因此 8 的 Gamma 编码为 0001000. 解码时,从左到右数出连续 0 的个数,为 3,然后解码 $c+1=3+1=4$ 位,即得 8.

SB 表示每个超块在编码序列 S 中的偏移位置,例如,上表中的第 2 个超块的第 2 个差值 3(因为每块中的第 1 个 Φ 值会被采样,不需要保存在 S 中)的编码在 S 中的偏移为 14,所以 $SB[1]=14$. B 中保存的是块中第 2 个差值(即每个块内之后的那个数)的偏移量,这个偏移量是相对于该块所在超块的相对偏移量,如第 5 个块是在第 2 个超块中,且相对偏移量为 8,所以 $B[4]=8$. 上表中的 SAM 数组保存的是每个块中的第 1 个数的值,即 Φ 的采样值.

如果要访问 $\Phi(i)$,就要进行解码. 解码过程是首先查询 SB 和 B 结构后确定所属块在 S 中的位置,然后解码 S 序列,加上相应的采样值 SAM,即可得到 Φ 值. $\Phi(i)$的表示如下:

$$\Phi(i)=(SAM[\lfloor i/b \rfloor]+decompress(S,SB[\lfloor i/a \rfloor]+B[\lfloor i/b \rfloor],i \bmod b)) \bmod n \quad (4)$$

其中 a 为超块长度,b 为块长度. $decompress$ 过程完成在 Gamma 编码序列 S 上的解码,解码的起始位置由 $SB[\lfloor i/a \rfloor]+B[\lfloor i/b \rfloor]$确定,即第 i 个值所属块的起始位置. $i \bmod b$ 为要解码的次数,最后将解码值与第 1 项 $SAM[\lfloor i/b \rfloor]$相加得到 $\Phi(i)$值.

对于图 2 示例,如果要求 $\Phi(11)$,首先求得 $SAM[11/b]=SAM[11/3]=35$,又由于 $SB[11/a]+B[11/b]=SB[11/9]+B[11/3]=SB[1]+B[3]=14$,即解码起始位置为 14,且 $11 \bmod 3=2$,所以从 S 表中的第 14 个位置开始解码,连续解码 2 次和 $SAM[3]$累加即可,S 表中第 14 个位置解码第 1 次得到 3,第 2 次解码得 5,因此 $\Phi(11)=(35+3+5) \bmod 36=7$.

如果要求 $\Phi(25)$,首先求得 $SAM[25/b]=SAM[25/3]=9$,又由于 $SB[25/a]+B[25/b]=SB[25/9]+B[25/3]=SB[2]+B[8]=58$,即解码起始位置为 58,且 $25 \bmod 3=1$,所以从 S 表中的第 58 个位置开始解码,解码 1 次和 $SAM[8]$累加即可,S 表中第 58 个位置解码 1 次得到 1,因此 $\Phi(25)=(9+1) \bmod 36=10$.

3.2 空间占用

令 g_j 为 x-list 内第 j 个 gap 的长度,n_x 为该表中的元素数. 由 x-list 的定义可知,n_x 恰好为字符 x

在文本 T 中出现的次数. 我们使用 Elias Gamma 对 $g_j=\Phi(j+1)-\Phi(j)$ 进行编码，该编码长度包含 $2\lfloor g_j\rfloor+1$ 位. 因此，该表内的 gap 总长度为

$$\sum_{j=1}^{n_x-1}|\Phi(j+1)-\Phi(j)|\leq n$$

所需要的总编码位数至多为 $\sum_{j=1}^{n_x-1}(2\log(\Phi(j+1)-\Phi(j))+1)$ 位，最坏情况下为 $2n'_x\log(n/n'_x)+n'_x$ 位，其中 $n'_x=n-1$. 对所有表上的 gap 编码位数求和，可得 $\sum_{x\in\Sigma}(2n'_x\log(n/n'_x)+n'_x)$，即 $2nH_1+n'$ 位，其中 $n'=n-\sigma$，σ 为字符表 Σ 的大小.

使用文献[21]中的方法，对表内 gap 编码进行更为细致的分析，可得更好的一个空间上界. 令 $\Phi(i)$ 的 k-context 表示 $T[SA[\Phi(i)]]$ 长为 k 的前缀. 按照 k-context 划分，每个 x-list 可被划分为至多 σ^k 个子表，由此可见，$\sum\log(\Phi(j+1)-\Phi(j))\leq nH_k$，其中求和取自同一表内及同一 k-context 内的所有 gap. 不在同一 k-context 内的 gap 数目至多为 σ^{k+1}. 同一 k-context 内的所有 gap 的编码总长度为 $2nH_k+n$，其他 gap 的编码总长度为 $2\sigma^{k+1}\log n$. 值得一提的是，我们的结果对于任何 k 都成立.

然而，我们还需要处理一种情况，那就是有些 gap(每块中的第 1 个)是不需要编码的，因为它们已被采样，这部分的编码大小可以去掉. 这些 gap 所占总空间至多为 $(n/b)(2\log g_i+1)=o(n)$，因为 $b=(\log n)^2/\log\log n$.

现在考虑结构 SB，B 和 SAM 所占空间. 每个超块的起始位置至多为 $|S|=O((n/b)(\log nH))$，其中 H 是编码每符号的平均位数；而每块在其超块内的相对位置至多为 $O((\log n)^2)$. 我们还需存储 $O(n/b)=O(n/((\log n)^2/\log\log n))$ 个 Φ 值采样，每个值需要 $\log n$ 位. 因此，结构 SB，B 和 SAM 所占总空间为 $O((n/a)\log|S|+(n/b)\log|b|+(n/b)\log n)$ 位. 将 $a=(\log n)^2$ 和 $b=(\log n)^2/\log\log n$ 代入，可得这 3 个结构所占总空间为

$$O\left(\frac{n}{(\log n)^2}\log(nH)+\frac{n(\log\log n)}{(\log n)^2}\log((\log n)^2)+\frac{n(\log\log n)}{(\log n)^2}\log n\right)=o(n).$$

将以上各部分所占空间加起来，可得总空间界为 $2nH_k+n+2\sigma^{k+1}\log n+o(n)$ 位.

定理 1.　对于任意 $k\leq c\log_\sigma n-1$ 及常数 $c<1$，Φ 的简明表示所占空间为 $2nH_k+n+o(n)$ 位，其中 H_k 表示文本 T 的 k 阶经验熵.

3.3　加速 Φ 值访问

给定编码序列 S 及解码位置 p. 对于大小为 $O(\log n)$ 的块，最坏情况下需要 $O(\log n)$ 时间访问 Φ. 通过维持一个宽度为 $W=O(\log n/2)$ 的表 R(如表 2 所示)，可在几乎常量时间内访问 Φ. 表 R 由四部分组成，分别记为 R_1、R_2、R_3 和 R_4. R_1 用来快速确定宽度为 W 的位串中从左端起连续出现的 0 的个数. R_2 表示宽度为 W 的位串中可以完整解码的 gap 数目. R_3 表示该宽度为 W 的位串可以正确解码的位数. R_4 表示可以正确解码(gap)值的累加和，这与我们用差值保存 Φ 是对应的，直接保存累加值，省掉了累加的过程.

例如，对于表 2 中的第 1 行，16 位全是 0，所以 R_1 列值为 16，该 16 位串一个 gap 都解码不出来，所以 R_2 列为 0，R_3、R_4 列也就都为 0. 该表中的第 6 行的位串 0010101110100110，可以完整解码的 gap 数目为 5，所以 $R_2=5$. 这些 gap 所对应的解码位数之和为 15，故 $R_3=15$，解码 gap 值总和 $R_4=14$，所剩 1 位是不可解码的.

表 2　R 表

index	R_1	R_2	R_3	R_4
0000000000000000	16	0	0	0
0000000000000001	15	0	0	0
…	…	…	…	…
0000000111111110	7	1	15	255
…				
0010101110100110	2	5	15	14
0010110010101110	2	5	15	15
…				
1010101001111001	0	7	13	11
…				
1111111111111111	0	16	16	16

可用 $2^W\log W$ 位存储每个 $R_i(i=1,2,3)$ 表，因为每个 R_i 表中的元素在 $[1,W]$ 中取整数值，因而每个元素可用 $\log W$ 位表示. 代入 $W=\log n/2$，可得这 3 个表所占总空间为 $3\sqrt{n}(\log\log n-1)$ 位. 类似分析可得 R_4 所占空间为 $(1/4)\sqrt{n}\log n$ 位，因为 R_4 至多有 $2^{W/2}$ 个元素，每个元素至多用 $\log n$ 表示. 因此，R 表所需总空间为 $3\sqrt{n}(\log\log n-1)+(1/4)\sqrt{n}\log n=o(n)$ 位.

利用 R 表，可以加速 $\Phi(i)$ 值的获取. 设 i 表示需要解码的次数，$temp$ 表示某次查找表可以解码的 Gamma 编码数(即可以解码的 gap 数)，num 表示已经解码的编码个数，则获取 Φ 的过程如下：

(1)读取 SAM、SB 和 B，得到采样值和偏移量.

(2)读取 W 位,用查找表解码,$temp$ 为本次可以解码的 Gamma 编码数,更新 $num=num+temp$,如果 $num<i$,重复该步,否则,后退一步,抛弃本次解码值,转到步(3).

(3)由步(2)的结束位置开始,顺序解码,每解码一次,num 自增,直至 $num=i$ 为止,此时返回解码值的累加和.

假设待解码序列为 S,当前位置为 p,需要连续解码 10 次,则利用查找表的解码过程如下:

获取 S 序列的前 $W(W=16)$ 位,不妨说 1 010 1 010 011 1 1 001,查表 R_2,发现该 16 位串可以解码 7 个数(gap),即该 16 位串包含 7 个完整的 Gamma 编码,查表 R_4,得到对应解码(gap)的累加值,即 11,查表 R_3 得到正确解码的位数,即这 7 个完整 Gamma 编码的解码位数总和为 13 位,所以本次查表解码 7 个数,累加值为 11,正确解码的位数为 13. 下一次从 $p+13$ 位置开始,再解码 $10-7=3$ 次,继续累加即可,新的 16 位串为 00101 011 1 010 011 0,查表 R_2,该串可以解码 5 个数,大于剩余解码次数 3,则结合 R_1 表,按照步 3,顺序解码 3 次,分别得到 5,3,1,与 gap 当前累加值累加,得到 $decompress(S,p,10)=20$,则一个完整的 $decompress$ 操作完成.

3.4 字符频数统计

统计字符频数是为了支持自索引. 按照字典序,$C[x]$表示原文本 T 中所有小于 x 的字符出现次数总和,$C[x]$的实际含义为:第 1 次以字符 x 打头的后缀的排名. 表 3 给出了图 1 中文本 T 的 C 值. 为了方便,我们在字符频数统计表的最后一个位置增添一项,填入 n 值.

表 3 字符频数统计表 C

a	b	c	d	e	f	g	
0	4	10	16	20	23	30	36

利用 C 表和 Φ 的表示结构,可以恢复 $SA[i]$对应的后缀 $T[SA[i]..n-1]$,因此 T 可以丢弃. 对于 $i=j,\Phi[j],\Phi[\Phi[j]]\cdots$,后缀 $T[SA[i]..n-1]$的第 1 个字符 $T[SA[i]]$在 SA 中是连续且字符表有序的. 因而 $T[SA[i]]$必定为满足 $C[c]<i\leq C[c+1]$ 的字符. 因此我们可以首先对 i 在 C 上做二分查找,确定 i 所属的字符区间,然后利用以下的 $incode$ 表,就可以恢复出首字符,然后改变 i 为 $\Phi[i]$,重复上述过程,即可恢复后缀 $T[SA[i]..n-1]$.

字符重映射的含义为:假设 T 中出现了 σ 个不同的字符,按照其 ASCII 编码排序这 σ 个字符,由小到大重映射为 $0\sim\sigma-1$,即编码为 $0\sim\sigma-1$. 在需要时,也需要把 $0\sim\sigma-1$ 之内的数字编码值再映射回原字符. 这两个映射可以使用大小分别为 256 和 σ 的数组完成. 以图 1 为例,假设 $code$ 表完成由字符到编码的映射,$incode$ 表完成由编码到字符的映射,表 4 和表 5 分别给出了这两个映射的实例.

表 4 code 表(size:256)

字符	0	…	a	b	c	d	e	f	g	…	255
编码	…	…	0	1	2	3	4	5	6	…	…

表 5 incode 表(size:σ)

编码	0	1	2	3	4	5	6
字符	a (97)	b (98)	c (99)	d (100)	e (101)	f (102)	g (103)

4 索引构造

我们可在 $O(n)$的时间内完成压缩后缀数组构造(不包括计算后缀数组的时间),主要由 4 步组成.

步 1(预处理). 读取原文件 T,计算字符频数统计表 C,计算 SA.

步 2(构造 Φ 数组). 利用表 C、SA、T 计算 Φ 值,删除 T.

步 3(采样 $SA\&SA^{-1}$). 采样 SA 和 SA^{-1} 数组,保存采样值,删除 SA.

步 4(编码 Φ). 采样、编码 Φ 数组,即用图 1 中的 S、SB、B 和 SAM 结构保存 Φ 数组,完成之后,删除 Φ 数组.

4.1 构造近邻函数 Φ

读取源文件,完成统计工作,必要时过滤非法字符,创建字符统计表 C、字符映射表 $code$、重映射表 $incode$,计算 SA 等都可以在这一步完成.

算法 $constructPhi$ 给出了近邻函数 Φ 的构造过程.

算法 1. $constructPhi$.

输入:C,SA,T

输出:Φ

```
1.  temp←C[lastchar]
2.  FOR i←0 to n−1 DO
3.      pos←SA[i]
4.      IF pos=0 THEN
5.          h←i
6.      ELSE
7.          c←T[pos−1]
8.          Φ[C[c]]←i
9.          C[c]←C[c]+1
10. Φ[temp]←h
11. RETURN Φ
```

算法计算公式(1)中定义的 Φ 值. 第 1 行记录后缀 $T[n-1]$($lastchar$ 是 T 的最后一个字符)的排名,保存在 $temp$ 中,即 $SA[temp]=n-1$. 算法的第 4、5 行记录 $SA[i]=0$ 时的 i 值,保存在 h 中. 根据式(1),$\Phi[temp]=h$,因为 $SA[h]=(SA[temp]+1) \bmod n=0$. 对于图 1 中的示例,$SA[0]=0$,所以 $h=0$,$C[lastchar]=C[f]=23$,故 $temp=23$,所以 $\Phi[23]=0$. 显然,该算法的时间复杂度为 $O(n)$. 该算法基于以下事实:当扫描整个后缀数组时,假设此时在处理后缀 $SA[i]=j$,如果 $c=T[j-1]$,且 c 是在某个后缀之前第 k 次出现,则 c-list$[k]=i$,即 $\Phi[C[c]]=i$. 算法第 9 行中 $C[c]$ 每次自增,所以 $C[c]$ 始终指向 c-list 表中第 1 个需要赋值的单元. 在算法伪代码的描述中,我们假设数组 $C[]$ 是一个局部变量. 当算法 $constructPhi$ 执行完时,C 表值复位.

4.2　采样 SA 及 SA^{-1}

被采样的后缀数组 SA 及逆后缀数组 SA^{-1} 中的点将作为锚点($anchor$)保留下来,用来恢复未采样点的值. 采样过程中将产生 SA_l 和 SA_l^{-1} 数组. 假设 SA 的采样步长为 c,SA^{-1} 的采样步长为 d,采样过程如下.

算法 2.　$sampling SA\&SA^{-1}$.

输入:SA,c,d

输出:SA_l,SA_l^{-1}

1. $k \leftarrow \lceil (n-1)/c \rceil$
2. FOR $i \leftarrow 0$ to $k-1$ DO
3.　$SA_l[i] \leftarrow SA[i \cdot c]$
4. FOR $i \leftarrow 0$ to $n-1$ DO
5.　IF $SA[i] \bmod d=0$ THEN
6.　$SA_l^{-1}[SA[i]/d] \leftarrow i$
7. RETURN SA_l and SA_l^{-1}

算法的第 2～3 行计算 SA_l,每隔 c 个 SA 值采样一个值,即保留那些下标为 c 的倍数的 $SA[i]$值,SA_l结构可用来支持 $getpos$ 过程(第 5 节中描述),用于返回 $SA[i]$的值. 第 4～6 行计算 SA_l^{-1}. 该结构是对逆后缀数组 SA^{-1} 进行采样(利用 SA 与 SA^{-1} 互逆的特点,所以采样时不需要 SA^{-1} 的存在),用来支持 $extract(start,len)$ 操作,即返回 $T[start..start+len-1]$. 我们知道,对于某个已知具体排名的后缀可以方便的恢复该后缀,所以如何将位置 $start$ 转换成排名 i 成为唯一的难点. 我们可以首先确定 $start/d$ 的排名 i,再重复执行 $start \bmod d$ 次 $i \leftarrow \Phi[i]$即可. 得到排名后,按照 3.4 节的方法,就可以恢复 $T[start..start+len-1]$,当然我们需要处理 $start+len$ 大于 n 的情况.

4.3　编码 Φ

该过程主要完成 S、SB、B 和 SAM 这 4 个结构的初始化工作. 假设超块长度为 a,块长度为 b,且 a 是 b 的倍数,并假设 S、SB、B 和 SAM 所需空间已经在该步的预处理阶段分配完成. 算法过程如下.

算法 3.　$codingPhi$.

输入:Φ

输出:S,SB,B,SAM

1. Initialize $index_1$, $index_2$, $index_3$, len_1, len_2 to be 0
2. FOR $i \leftarrow 0$ to $n-1$ DO
3.　IF $i \bmod a=0$ THEN
4.　$len_2 \leftarrow len_1$
5.　$SB[index_3] \leftarrow len_2$
6.　$index_3 \leftarrow index_3+1$
7.　IF $i \bmod b=0$ THEN
8.　$SAM[index_1] \leftarrow \Phi[i]$
9.　$index_1 \leftarrow index_1+1$
10.　$B[index_2] \leftarrow len_1-len_2$
11.　$index_2 \leftarrow index_2+1$
12.　$pre \leftarrow \Phi[i]$
13.　ELSE
14.　$gap \leftarrow \Phi[i]-pre$
15.　IF $gap<0$ THEN
16.　$gap \leftarrow gap+n$
17.　$pre \leftarrow \Phi[i]$
18.　$len_1 \leftarrow len_1+2bl(gap)-1$
19.　$append(gap,S)$
20. RETURN S,SB,B, and SAM

当第 3 行的 IF 条件成立时,该点就是超块的采样点,只保留采样点在 Gamma 编码串中的绝对偏移量,$index_3$是该结构(超块)的下标. 第 7 行的 IF 条件成立时,该点对应块的一个采样点,需要保留采样点的 Φ 值. 第 8～9 行完成这一工作,也需要保存该采样点相对于所属超块采样点的偏移量,所以第 10 行保存 $len_1 \sim len_2$,显然 len_2 为采样点所属超块的绝对偏移量,len_1 为当前 S 序列的长度. 第 12 行和 17 行更新 pre 值,表示前一个 Φ 值. 当前两个 IF 条件都不成立时,执行 14～19 行,计算差值 gap,如果发生 $gap<0$ 的情况,第 16 行修正. 第 18 行 $bl(gap)$表示 gap 的二进制码长,所以 $2bl(gap)-1$ 就是 gap 的 Gamma 编码长度. 第 19 行 $append$ 操作把 gap 值按照 Gamma 编码的方式,追加到 S 结构上. 显然,整个过程的时间复杂度为 $O(n)$.

因此,索引 CSA 的构造过程可以在线性时间内完成,最终保存的结构为 S、SB、B、SAM、SA_l 和

SA_l^{-1}结构，S、SB、B 和 SAM 加在一起在逻辑上充当 Φ 数组，$\Phi[i]$可由式(4)得到. 也可以用 3.3 节介绍的查找表加速计算.

5 模式匹配

5.1 查 找

查找过程可以分为计数查询和定位查询，分别用 *count* 和 *locate* 表示. *count* 返回模式 P 在 T 中的出现次数. 本质上说，*count* 查询确定 T 中以 P 为前缀的那些后缀的范围$[L,R]$. 一旦完成 *count* 查询，我们就能得到$[L,R]$中每个后缀在 T 中的起始位置. 算法采用后向查找技术，并结合 3.4 节的 C 表，可以实现自索引. 算法摆脱了基于字符串比较的查询框架，在查询的过程中，不断利用 Φ 数组，缩小结果范围$[L,R]$，当迭代到模式的首字符时，$SA[L,R]$内的所有后缀都以 P 为前缀，算法描述如下.

算法 4. *count*.

输入：P,Φ

输出：L,R

1. $c\leftarrow P[m-1]$
2. $L\leftarrow C[c]$
3. $R\leftarrow C[c+1]-1$
4. FOR $i\leftarrow m-2$ down to 0 DO
5. $c\leftarrow P[i]$
6. $LL\leftarrow C[c]$
7. $RR\leftarrow C[c+1]-1$
8. $newL\leftarrow\min\{j:\ j\in[LL,RR]\&\Phi[j]\in[L,R]\}$
9. $newR\leftarrow\max\{j:\ j\in[LL,RR]\&\Phi[j]\in[L,R]\}$
10. $L\leftarrow newL$
11. $R\leftarrow newR$
12. IF $L>R$ THEN
13. RETURN "pattern does not exist"
14. ELSE
15. RETURN L and R

其中 m 表示 P 的长度，第 8～9 行的含义为确定新的左右边界，该算法建立在 Φ 变换的实际含义和特点上. 该算法由模式的最后一个字符开始，循环到模式的第 1 个字符结束. 显然，该算法在执行过程中保持下列循环不变式：当算法执行完倒数第 k 个字符时，$[L,R]$区间之内的后缀以模式 P 的后 k 个字符为前缀，证明如下：

初始化. 算法执行 1～3 行，完成初始化，显然此时 L 对应 c-list 的第 1 个元素，R 对应 c-list 的最后一个元素，所以此时区间$[L,R]$就是 c-list 区间，该区间内的所有后缀均以字符 c 开头，得证.

保持. 假设算法已经执行到了倒数第 k 个字符 c，即此时区间$[L,R]$内的后缀以模式 P 最后面的 $k-1$ 个字符为前缀. 算法在第 6～7 行确定 c-list 的区间$[LL,RR]$，并且该区间内的 Φ 值都是升序的. $\Phi[i]$值的实际含义为该后缀"剔除"掉首字符后的后缀的排名，如果 T 中包含该模式，即会出现 $p_k p_{k-1} p_{k-2}\cdots p_{len-1}$，那么区间$[LL,RR]$中一定有一个连续片段$[newL,newR]$，其中所有的 Φ 值都在区间$[L,R]$内，因为$[L,R]$区间内的后缀都以 $p_{k-1} p_{k-2}\cdots p_{len-1}$ 为前缀，所以执行完该次循环，以$[newL,newR]$为新的$[L,R]$，即该区间内的后缀都以模式的最后 k 个字符为前缀.

结束. 算法有两个退出位置，第 13 行或第 14 行. 由第 13 行退出时，表示模式不存在，当由第 14 行退出时，区间$[L,R]$内的所有后缀都以模式 P 为前缀.

因此当该算法结束时，当 $L\leq R$ 时，模式出现 $R-L+1$ 次，当由第 13 行退出时，$L>R$，表示模式没有出现. 以 P="bga"为例，算法执行过程如下：

1～3 行初始化. 确定区间 $L=C[a]=0$，$R=C[a+1]-1=C[b]-1=3$，即区间$[L,R]=[0,3]$内的后缀都以字符 a 打头，对应 a-list.

第 1 次循环. 此时 $LL=C[g]=30$，$RR=C[g+1]-1=35$，即区间$[30,35]$内的后缀都以字符 g 打头，$[30,35]$内的 Φ 值为$\{1,3,5,13,16,19\}$，其中前两个的 Φ 值属于区间$[L,R]$，所以$[newL,newR]$为$[30,31]$，第 10、11 行将$[L,R]$更新为$[30,31]$，即该区间内的后缀以"ga"为前缀，可以从图 1 得到验证.

第 2 次循环. 此时$[L,R]$为$[30,31]$，$LL=C[b]=4$，$RR=C[b+1]-1=C[c]-1=9$，$[4,9]$内的 Φ 值为$\{24,25,29,30,31,35\}$，所以$[newL,newR]=[7,8]$，满足 $R\geq L$ 的条件，更新$[L,R]$为$[7,8]$，循环结束，返回.

所以最终$[7,8]$区间内的后缀以"bga"为前缀. 在具体确定 $newL$ 和 $newR$ 时，由于$[LL,RR]$区间内的 Φ 值都是升序的且采样点的值是直接存储的，所以先在采样点上执行二分查找，确定 $newL$、$newR$ 的目标区间 B_i，然后在目标区间 B_i 上利用查找表顺序解码，确定具体的位置. 采样点上二分过程的时间复杂度为 $O(\log(n/b))$，b 表示块大小. 假设 H 表示 Gamma 编码的平均长度，则一个 B 块对应 bH 位，一次查找表可以解码 W 位，W 表示查找表宽度，所以顺序解码过程的复杂度为 $O(bH/W)$. 所以算法

第 8～9 行的复杂度为 $O(\log(n/b)+bH/W)$. 对于块大小 $b=(\log n)^2/\log\log n$，计数查询算法的时间复杂度为 $O(m\log n/\log\log n)$，m 为模式 P 的长度. 算法的返回值为 L 和 R，表示区间 $SA[L,R]$ 内的后缀以模式 P 作为前缀，故 P 出现的次数为 $R-L+1$ 次. 如果查询的模式不存在，某次循环第 12 行的 IF 条件将成立，跳出循环，直接返回模式不存在.

定理 2. 给定长为 m 的模式 P，借助于查找表 R(3.3 节描述)，计数查询可在 $O(m\log n/\log\log n)$ 时间内完成，空间占用由定理 1 给出，额外 C 表开销为 $\sigma\log n$ 位，其中 σ 为字符表大小.

算法 *locate* 以 *count* 过程返回的 $[L,R]$ 作为输入，然后利用 *getpos*(i) 操作确定 $SA[i]$ 的值，$i\in[L,R]$.

算法 5. *locate*.

输入：P,L,R

输出：*ans*

1. $ans[0\cdots R-L]\leftarrow 0$
2. FOR $i\leftarrow L$ to R DO
3. $\quad ans[i-L]\leftarrow getpos(i)$
4. RETURN *ans*

getpos(i)过程返回 $SA[i]$，即第 i 个最小后缀在 T 中的起始位置. 算法过程如下.

过程 *getpos*(i).

1. $step\leftarrow 0$
2. WHILE $i \bmod c\neq 0$ DO
3. $\quad step\leftarrow step+1$
4. $\quad i\leftarrow\Phi[i]$
5. $i\leftarrow i/c$
6. RETURN $(SA_l[i]-step)\bmod n$

getpos 从某点 $i\in[L,R]$ 出发，沿着 $i'(\Phi[i])$，$i''(\Phi[i'])$，…，顺序后移，满足 $SA[i]+1=SA[i']$，$SA[i']+1=SA[i'']$，…，直至遇到被采样的某点 $SA_l[i]$. 令 *step* 表示后移的步数，则结果返回为 $SA_l[i]-step$，该结果在算法第 6 行返回. 因为对于 $b=(\log n)^2/\log\log n$，使用宽度为 $W=O(\log n/2)$ 的查找表，访问 Φ 的时间为 $O(\log n/\log\log n)$. 因此，*locate* 算法的时间复杂度为 $occ\cdot c\cdot O((\log n)^2/\log\log n)$，其中 c 为 SA 的采样步长，occ 为模式在 T 中出现的次数.

定理 3. 给定长为 m 的模式 P 及模式出现的范围 $[L,R]$，定位查询可在 $occ\cdot c\cdot O((\log n)^2/\log\log n)$ 时间内完成，其中 c 为 SA 的采样步长，occ 为模式出现的次数.

继续以模式“bga”为例，*count* 过程求出 $[7,8]$ 区间内的所有后缀具有“bga”前缀，所以 *locate* 过程用 *getpos* 分别求出 $SA[7]$，$SA[8]$ 的值. 以求 $SA[8]$ 为例，简单描述 *getpos* 过程($c=3$). $step=0$，$i=8$，WHILE 循环开始. $8 \bmod 3\neq 0$，step$=1$，更新 $i=\Phi[i]=\Phi[8]=31$；$31 \bmod 3\neq 0$，step$=2$，更新 $i=\Phi[31]=3$. $3 \bmod 3=0$，循环结束. 第 5 行 $i=i/3=3/3=1$，算法在第 6 行返回 $SA_l[i]-2=34-2=32$.

5.2 展示文本串

给定子串起始位置 *start* 和长度 *len*，*extract*$(start,len)$操作可以展示 T 中起始位置 *start* 处长度为 *len* 的子串，即 $T[start..start+len-1]$. 4.2 节采样 SA^{-1} 数组，产生 SA_l^{-1} 数组. 我们可以利用该数组实现 *extract*$(start,len)$操作，返回这个子串. 在 3.4 节中我们讨论了如何恢复后缀 $SA[i]$，即返回后缀 $T[SA[i]..n]$，该过程会用到 Φ 的表示结构和 C 表(字符统计表). 因此我们只需把某个后缀的起始位置 *start* 转换成对应的排名 i，*extract*$(start,len)$操作就能借助 3.4 节的方法完成.

由 4.2 节可知，对于排名为 i 的后缀，可以在 C 表上进行二分查找，确定 i 所属的区段 *index*，则排名为 i 的后缀的第 1 个字符就是 *incode*$[index]$，后续部分用 $f(i)$ 表示这一过程，即返回排名为 i 的后缀的首字符. 显然，过程 $f(i)$ 的时间复杂度由 C 表的二分过程决定，为 $O(\log\sigma)$，σ 表示 C 表大小，即字符表的大小. 更新 $i=\Phi[i]$，重复上述过程，可以确定整个后缀.

以下描述过程 *restore*(i,len)，该过程返回排名为 i 的后缀 $T[SA[i]..SA[i]+len-1]$.

过程 *restore*.

输入：i,len

输出：*substr*

1. FOR $j\leftarrow 0$ to $len-1$ DO
2. $\quad substr[j]\leftarrow f(i)$
3. $\quad i\leftarrow\Phi[i]$
4. RETURN *substr*

以下 *transform* 将后缀的起始位置 *start* 转换成排名 i.

过程 *transform*.

输入：*start*

输出：i

1. $i\leftarrow SA_l^{-1}[\lfloor start/d\rfloor]$
2. $step\leftarrow start \bmod d$
3. FOR $j\leftarrow 0$ to $step-1$ DO
4. $\quad i\leftarrow\Phi[i]$
5. RETURN i

d 表示 SA^{-1} 数组的采样步长，该过程先查找比 $start$ 小的最大的 SA^{-1} 数组采样点$\lfloor start/d \rfloor$，得到该点排名 i，然后重复执行 $start \bmod d$ 次 $i=\Phi[i]$，根据 $\Phi[i]$的物理含义，此时的 i 就是 $start$ 位置开始的后缀的排名 i.

算法 6. *extract*.

输入：*start*，*len*

输出：*substr*

1. $i \leftarrow transform(start)$
2. $substr \leftarrow restore(i, len)$
3. RETURN *substr*

算法的时间复杂度由两个过程 $f(i)$和 $\Phi[i]$决定. 由本节分析可知，$f(i)$的时间复杂度为 $O(\log\sigma)$. *Transform* 最坏情况下执行 d 次 $\Phi[i]$操作，*restore* 执行 *len* 次 $\Phi[i]$操作，因而 *extract* 操作的时间复杂度为 $O((d+len)(\log n/\log\log n)+len\log\sigma)$.

定理 4. 给定子串起始位置 *start* 和长度 *len*，展示子串查询可在 $O((d+len)(\log n/\log\log n)+len\log\sigma)$时间内完成，其中 d 为 SA^{-1}的采样步长.

图 3 给出了 $c=d=3$ 时对后缀数组 SA 和逆后缀数组 SA^{-1}的采样结果，分别用 SA_l和 SA_l^{-1}表示. 我们将 SA_l和 SA_l^{-1}存储在每个单元占用 $\log n$ 位的数组里. 结合图 3，执行 *extract*(14,4)的过程如下：

第 1 步. 确定位置 14 的排名，图 3 中 SA^{-1}的采样步长为 3，不超过 14 的最大采样点为 12，对应 SA_l^{-1}行中第 4 项(下标从 0 开始)，得 $i=11$，即位置 12 开始的后缀的排名为 11，14 mod 3=2，重复执行两次 $i=\Phi[i]$，最终 $i=30$，即位置 14 开始的后缀的排名为 30，该计算由 *transform* 过程完成.

第 2 步. $len=4$，循环 4 次. 首先，$i=30$，属于 *g*-list 范围，故第 1 个字符为‘*g*’. 其次，$i=\Phi[30]=1$，属于 *a*-list 范围，故第 2 个字符为‘*a*’. 然后，$i=\Phi[1]=14$，属于 *c*-list 范围，故第 3 个字符为‘*c*’. 最后，$i=\Phi[14]=20$，属于 *e*-list 范围，故第 4 个字符为‘*e*’. 因此 $T[14..17]$=“*gace*”.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
T	*a*	*b*	*f*	*g*	*d*	*b*	*f*	*b*	*g*	*d*	*f*	*c*	*c*	*b*	*g*	*a*	*c*	*e*	*f*	*c*	*e*	*g*	*c*	*d*	*e*	*f*	*g*	*b*	*f*	*c*	*a*	*d*	*b*	*g*	*a*	*f*
SA	0	15	30	34	5	27	1	13	32	7	29	12	11	22	16	19	4	31	23	9	17	24	20	35	6	28	10	18	25	2	14	33	26	21	3	8
SA_l	0	34	1	7	11	19	23	24	6	18	14	21																								
SA^{-1}	0	6	29	34	16	4	24	9	35	19	26	12	11	7	30	1	14	20	27	15	22	33	13	18	21	28	32	5	25	10	2	17	8	31	3	23
SA_l^{-1}	0	34	24	19	11	1	27	33	21	5	2	31																								

图 3 SA_l和 SA_l^{-1}示例

6 改 进

不同数据的分布特点是不同的，体现在 Φ 的 *gap* 序列中 1 的比例. 我们的统计实验表明，像 dna 这样的数据，其 *gap* 值为 1 的比例大约占到总数的 40%～60%左右；像 english 这样的数据，其 *gap* 值为 1 的比例大约占到总数的 50%～70%左右；像 influenza 这样高度重复的(highly-repetive)序列，其 *gap* 值为 1 的比例可高达 80%以上. 因此 *gap* 序列中势必存在大量连续的 1(也称长游程，long runs)，单纯一种 Gamma 编码不能很好的适应这种情况，需要结合 run-length 编码，而 run-length 编码可以很高效的处理这种具有长游程的数据. 对 Φ 的 *gap* 序列运用 run-length 编码之后，解码速度也会有所提升，因为一个 runs 可能会对应好多个 *gap*，不再需要逐个恢复，所以 *gap* 中 1 的比例越高，则倾向于应用 run-length 编码，块大小也可以适当的取的大些. 但并不是所有的 *gap* 片段都呈现很好的 runs 特性，因而我们仍然需要 Gamma 编码方法作为第 2 种编码方法，当 run-length 编码失效时，使用 Gamma编码.

最终我们的数据感知的、自适应的编码策略如下：候选编码方法有 Gamma 编码，run-length 编码，块大小根据 *gap* 中 1 的比例决定，块内编码方法在 Gamma 和 run-length 编码中择优选取. 最终的结果是：我们改进的 CSA 能根据数据分布的不同，自动的选择合适的编码方法和块大小，具有数据感知和自适应的特点.

如何把 run-length 编码结合进来是个很艺术的问题，区分一个值 y(表示待编码的值)是 *gap* 还是 1 的 runs，一种简单的策略如下：

$$h(y)=\begin{cases}2y-3, & \text{如果 } y \text{ 是一个 } gap\\ 2y, & \text{如果 } y \text{ 是 1 的 runs}\end{cases}.$$

这样解码时通过奇偶判断就能确定编码的是 *gap* 还是 1 的 runs 值. 这里可做进一步优化，将 y 映射为 $2y-1$ 时，所有被编码的值都大于 1，此时我们可以修改 Gamma 编码的规则，当 y 的二进制需要 x 位时，在 y 的二进制前面追加 $x-2$ 个 0，而不是原来的 $x-1$ 个 0. 此外，高度重复数据的 runs 值

或偶尔出现的非 1 gap 的值都较大，所以 Elias Delta 编码更合适，按照类似的方式修改 Delta 编码规则，“剔除”最左端的 0. 我们的改进版称为 ACSA (adaptive CSA)，改进版在结构上比图 2 多了记录编码方法的 methods 域，每个块对应一个 ceil，每个 ceil 需要 2 位，可表示 4 种编码方法(Gamma，RLG，RLD，All1). 其中 Gamma 表示对 y 使用 Gamma 编码，RLG 表示对 y 使用 run-length Gamma 编码，RLD 表示对 y 使用 run-length Delta 编码，All1 表示整个块为 1 的 runs，这样的块不编码. 这个 methods 域所用空间为 $2(n/b)=o(n)$，对于 $b=(\log n)^2/\log\log n$.

实现时，超块大小固定为块大小的 16 倍，块大小自适应，以 gap 序列中 1 的比例为依据. 如果数据是高度重复的，设为 512，如果数据是像 english 这样的，设为 256，否则设为 128，令 b 表示块大小，则在实际中 b 值取值如下：

$$b=\begin{cases}128, & \text{if } r\le l_1\\ 256, & \text{if } l_1<r\le l_2,\\ 512, & \text{if } r>l_2\end{cases}$$

其中 r 为 gap 序列中 1 的比率.

我们的实现中，提供了 $speedlevel$ 参数，该参数决定 3 个典型的阈值：

$speedlevel=0$ 对应阈值$(l_1,l_2)=(0.50,0.60)$，此时压缩率较好，$count$ 查询较慢.

$speedlevel=1$ 对应阈值$(l_1,l_2)=(0.60,0.75)$，该阈值是个比较好的权衡值.

$speedlevel=2$ 对应阈值$(l_1,l_2)=(0.65,0.80)$，此时压缩率较差，$count$ 查询较好.

这些阈值和 $speedlevel$ 的取值都只是经验值，可以根据需求进行权衡. $speedlevel$ 值偏小时，侧重于压缩率，$speedlevel$ 值偏大时，侧重于查询速度.

7　实验结果与分析

7.1　实验环境设置

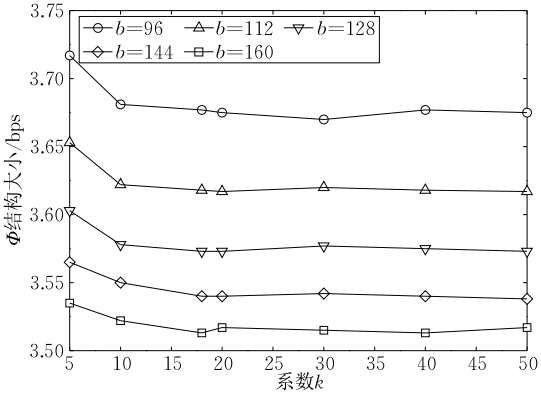
实验中没有运行其他大型程序，任何系统设置保持默认，运行 Ubuntu 12.04 LTS 32 位系统，g++ 4.4.1 编译，加-O3 优化参数. 机器情况：HP Z400，CPU：Inter(R)Xeon(R)2@2.53；RAM：4GB，L3：4MB.

我们使用源自 *Canterbury Corpus* (http://corpus.canterbury.ac.nz) 和 *Pizza & Chili Corpus* (http://pizzachili.dcc.uchile.cl/indexes.html) 的数据测试所提方法的性能. 我们的两个版本可在 https://github.com/chenlonggang/Adaptive-CSA 及 https://github.com/chenlonggang/Compressed-Suffix-Array 上获取.

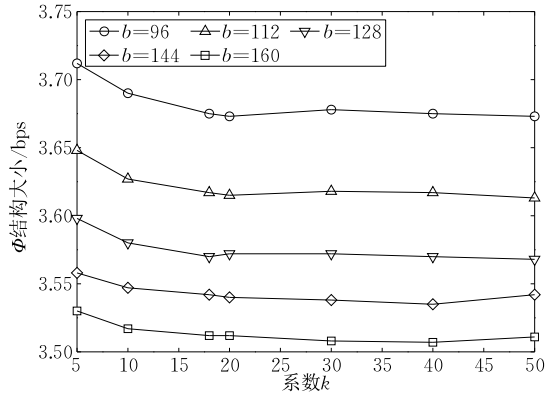
7.2　参数对性能的影响

我们的 CSA 结构是带参数的，参数的选取影响着压缩索引的空间和时间性能. 本节简要讨论这些参数对算法性能的影响. 假设 a 表示超块的大小，b 表示块的大小，c 表示 SA 的采样步长，d 表示 SA^{-1} 的采样步长，SAM、SB、B 按照紧凑方式存储. 为了降低空间，我们固定 $d=16c$. 本节我们将讨论 a 和 b 的最佳倍数关系，在此基础上确定参数 b、c 对算法实现性能的影响，最终确定一组最佳的参数.

图 4 表示 b 取不同的值，$a=kb$ 时，随着 k 的变化，Φ 结构大小(bps，bits per symbol)的变化趋势，横轴表示系数 k，纵轴表示 Φ 结构的大小. 从图 4 中可以看到，当 $n=50$ M、100 M 时，当 k 大于 18 左右之后，空间不再降低，甚至有变大的趋势，所以选取 $a=18b$ 是合适的. 后续的实验都是建立在 $a=18b$ 的基础上的.



(a) Φ结构大小随着系数k的变化趋势(n=50 M)



(b) Φ结构大小随着系数k的变化趋势(n=100 M)

图 4　Φ 结构大小

现在讨论参数 b、c 对性能的影响，实验中以 50 MB 大小 xml 数据为例.

第 1 组. 参数 b 对性能的影响

本组测试参数 b 对性能(压缩率、查询时间等)的影响，如图 5 所示. 实验中参数 $c=32$，参数 b 由 16 开始变化，以 16 为步幅，增加到 160.

参数 b 表示 Φ 块的大小，即 $\Phi[ib]$ 的点将被采样，直接保存，并且还会保存该点相对于所属超块的偏移量，随着参数 b 的增大，Φ 的辅助结构会越来越小，但随着 b 的增大，压缩率的降低趋势会放缓，图 5(a)表明了这一点.

参数 b 对压缩时间的影响不大，如果假设某个点被采样和被编码的时间代价相当，则不管参数 b 如何变化，压缩时间都不会有明显的变化，这一点可以从图 5(b)中看出.

图 5(c)和图 5(d)清楚的说明了 Φ 块大小 b 对计数查询、定位查询时间的影响，因为这两个操作都需要 Φ 结构的支持，增大 b 应该会显著的影响计数查找的时间，因为增大 b 会增大 Φ 获取过程中的跳跃次数，但是图 5(c)表明参数 b 的变化对计数查找的影响不大，这得益于对查找表的充分利用.

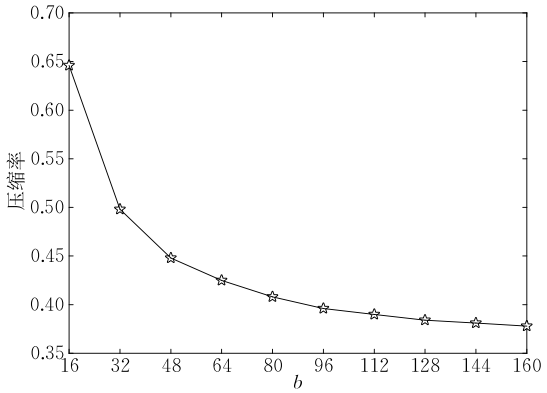
图 5(d)中参数 b 由 16 变化到 160 时，定位查询时间变为原来的 2 倍左右，图 5(a)表明压缩率变为原来的 0.6 倍左右，观察图 5(a)，图 5(d)的走势，不建议选取大于 128 的 b 值，因为 $b>128$ 后，对压缩率的贡献降低，但定位查找时间显著上升.

第 2 组. 参数 c 对性能的影响

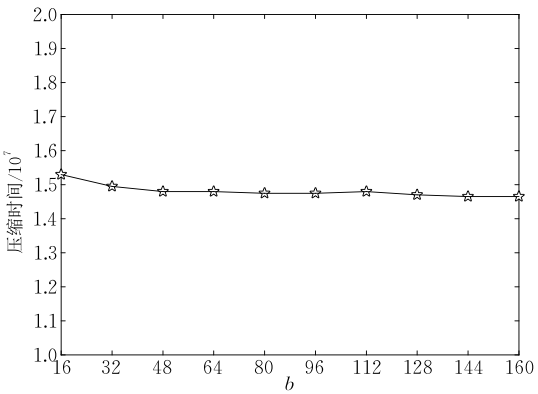
本组测试固定参数 $b=128$，参数 c 由 16 开始变化，以 16 为步幅，增加到 160. 在图 6(a)中，参数 c 由 16～160 变化时，压缩率逐渐降低，但随着 c 的增大，降幅逐渐减小，在整个过程中，压缩率由 0.49 左右降到 0.30 左右，变为原来的 0.6 倍左右. 压缩时间依然没有显著变化，调整 c 并不影响计数查找，因为计数查找只需要字符频数统计表 C 和 Φ 数组的支持. 图 6(d)表示定位查找时间，$c=16$、160 时的增幅高达 16 倍左右，而图 5(d)中的增幅只有 2 倍左右，这个结果是我们实际中将参数 b 选择较大，而将参数 c 选择较小的重要依据.

综合考虑上述实验，我们选取 $b=128$，$a=18b$，$c=32$，$d=16c$ 作为我们算法的默认值，该值既可以保证 Φ 数组的辅助空间较小，同时又保证了查询算法的速度，后续实验都基于该组参数值.

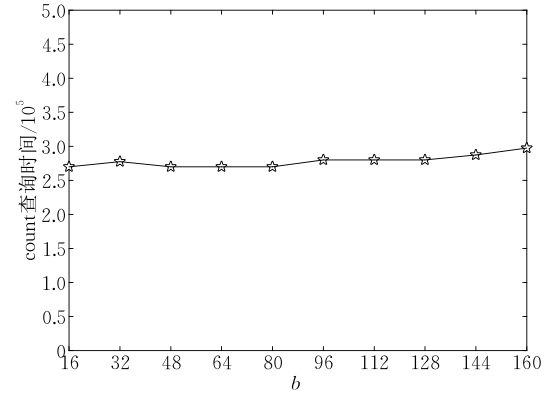
对于 ACSA，参数 b 取值分 3 档，128，256 和 512，*speedlevel* 可取值 0，1 和 2，对应的阈值分别为



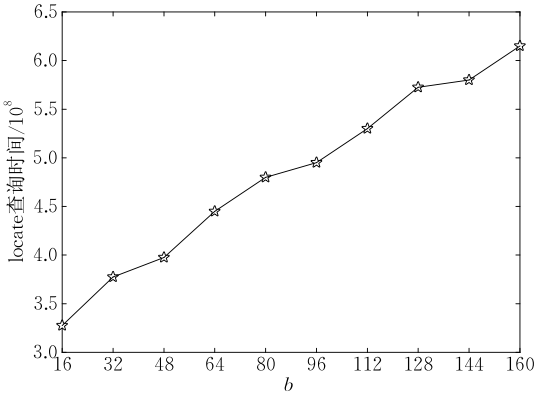
(a) 压缩率随着块大小b的变化趋势



(b) 压缩时间随着块大小b的变化趋势

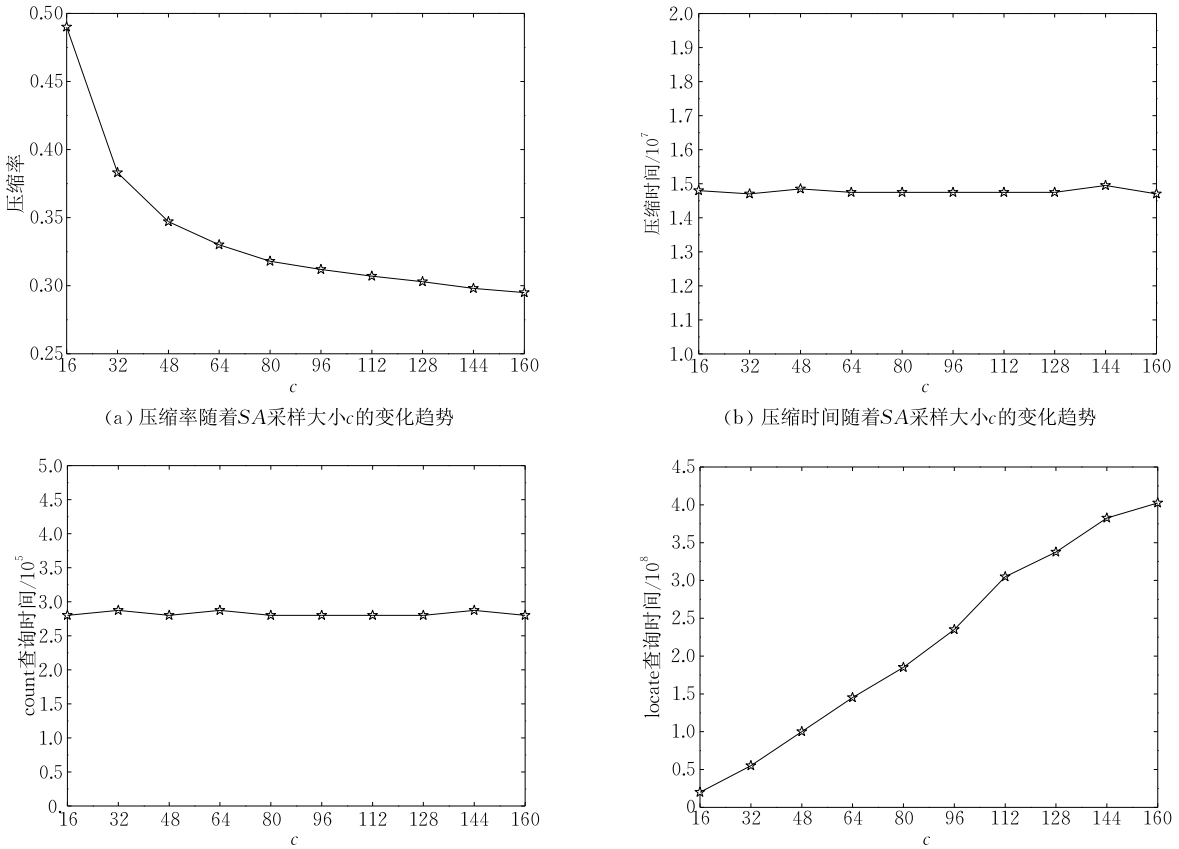


(c) count查询时间随着块大小b的变化趋势



(d) locate查询时间随着块大小b的变化趋势

图 5 参数 b 对性能的影响



(a) 压缩率随着SA采样大小c的变化趋势　(b) 压缩时间随着SA采样大小c的变化趋势

(c) count查询时间随着SA采样大小c的变化趋势　(d) locate查询时间随着SA采样大小c的变化趋势

图 6　参数 c 对性能的影响

(0.5,0.6),(0.6,0.75)和(0.65,0.80).此时阈值为 Φ 的 gap 序列中 1 的比例,$speedlevel$ 的默认值为 1,即当 1 的比例低于 0.6 时,b 取值 128,当 1 的比例在 0.6 到 0.75 之间时,b 取值 256,大于 0.75 时,b 取值 512.这些参数的取值都是在上述实验的基础上综合考虑的结果.

7.3　结果与分析

本节采用 *Pizza &Chili* 网站上的数据进行测试,与当前主流方法在压缩率和查询速度方面进行了比较.表 6 给出了有代表性的数据文件(100 M)的统计信息.它们包括像 dna 这样的数据(符号分布几乎均匀),像 influenza 这样的高度重复的数据以及介于这两者之间的数据,如像 english 文本这样的数据.

表 6　数据文件统计信息

(a) 一般序列

文件	dna	proteins	xml	source	english
字符集	16	25	96	227	215

(b) 高度重复序列

文件	influenza	kernel
字符集	16	161

待比较的方法包含:CSA(本文第 3 节的方法),ACSA(本文第 6 节的方法),$speedlevel=1$,FM-Index[14-15], RLCSA[23], Sad-CSA[12-13], SDSL-CSA[28].除了 ACSA 的块大小自适应之外,其他被比较的方法,块大小 $b=128$.我们在索引构造时间、压缩率和模式查询时间上与现有主流方法进行了比较.压缩率 bps(bits per symbol)定义为 CSA 结构大小与源文本文件大小的比值再乘以 8,表示文件压缩后每个字符平均所需要的比特数.对于每个文件,我们随机生成一万个长度为 20 的模式,获取平均的查询时间(单位为 us).每个算法查询的 10 000 个模式是一样的.表 7～10 中的黑体字说明的是这些方法在每一列不同数据上表现最好的前两个结果.

表 7 给出了上述的压缩索引方法在不同特征数据集上的构造时间.在所测试的 7 组数据中,CSA 的构造时间在其中 4 组数据上显著优于其他压缩索引的构造时间.FM-Index 在其余的 3 组数据上的构造时间表现最佳.但在这 3 组数据上,CSA 的构造时间与 FM-Index 的构造时间接近.

表 8 显示了对不同的数据采用不同的压缩索引

方法得到的压缩率.由表 8 可见,除了高度重复的数据 influenza 和 kernel 以及 sources,ACSA 的压缩率显著优于所比较的其他压缩索引方法的压缩率.而对于所测试的数据,CSA 的压缩率也有一定的优势.而对于高度重复的数据 influenza 和 kernel,RLCSA 的压缩率最佳,但我们 ACSA 的压缩率与之相当.FM-Index 的压缩率在所比较的压缩索引上表现最佳,而 ACSA 的压缩率也能与之相当.

随机选取数据文件中的长度为 20 的模式串,我们测试了不同压缩索引的 count 和 locate 查询时间,其结果如表 9 和表 10 所示.由表 9 可知,与其他所比较的压缩索引方法相比.CSA 的计数查询时间是最快的,这是由于我们设计了一个快速查找表,加速了 Φ 值的解码过程.改进后的 ACSA,在压缩率上进一步改善的同时,在查询时间上略逊于 CSA.但 ACSA 的查询效率在多数情况下仍然优于其他压缩索引的计数查询时间.由表 10 可知,在定位查询 locate 中,CSA 在 5 组测试数据上优于 Sad-CSA,在 4 组测试数据上优于 SDSL-CSA.改进后的 ACSA 增加了解码的复杂度,定位查询效率逊于 CSA,优于 RLCSA 和 FM-Index.

ACSA 和 CSA 相比,在压缩率上有很好的提升,特别是对于高度重复的序列数据,比如 influenza 数据,压缩效果提升 3.54 倍,块的增大并没有显著增大 count 查询时间,这归功于 run-length 编码.和 RLCSA 相比,CSA 和 ACSA 的 count 查询时间优势明显,压缩率在 dna 这样的数据和 english 这样的数据上优势也很明显,但在高度重复序列上略逊一筹.

表 7 索引构造时间 (单位:s)

Method	dna	proteins	english	sources	xml	influenza	kernel
ACSA	27.08	29.60	25.53	19.36	**20.32**	**23.02**	**19.77**
CSA	**25.10**	**27.16**	**23.97**	**17.84**	**19.51**	**22.48**	**18.94**
FM-Index	**22.15**	**23.33**	**22.96**	**18.33**	21.11	29.73	22.65
RLCSA	46.35	36.40	52.06	39.84	40.01	43.51	42.22
Sad-CSA	50.36	48.45	57.36	37.44	41.54	64.27	76.63
SDSL-CSA	33.59	36.26	32.04	26.19	27.19	30.97	27.18

表 8 压缩率比较(*bps*,*bits per symbol*)

Method	dna	proteins	english	sources	xml	influenza	kernel
ACSA	**3.54**	**5.52**	**2.97**	**2.25**	**1.22**	**0.37**	**0.57**
CSA	**3.56**	5.59	**3.52**	2.90	2.17	1.49	1.60
FM-Index	4.03	**5.55**	4.54	**2.24**	2.99	2.19	2.96
RLCSA	4.72	6.21	3.54	2.55	**1.39**	**0.29**	**0.49**
Sad-CSA	4.81	6.84	4.59	4.14	3.41	2.71	2.85
SDSL-CSA	5.31	6.92	4.78	4.24	3.56	2.82	2.93

表 9 Count 查询时间 (单位:μs)

Method	dna	proteins	english	sources	xml	influenza	kernel
ACSA	**35.962**	**36.238**	**35.166**	42.529	**25.530**	**37.423**	46.636
CSA	**30.447**	**29.224**	**34.487**	**27.960**	**22.445**	**31.097**	**28.861**
FM-Index	416.003	666.703	221.192	606.052	651.240	493.020	576.641
RLCSA	73.399	58.610	61.125	61.585	39.256	67.001	63.920
Sad-CSA	46.697	41.048	41.575	**38.896**	34.525	42.398	**38.534**
SDSL-CSA	95.706	61.832	96.099	70.265	49.893	81.780	73.042

表 10 Locate 定位时间 (单位:μs)

Method	dna	proteins	english	sources	xml	influenza	kernel
ACSA	1023.56	9186.42	624.44	102969.00	25831.30	2985.05	191733.00
CSA	**392.27**	**8163.84**	**495.07**	**33453.50**	**3626.76**	**240.93**	**7241.84**
FM-Index	3488.04	10745.61	1368.24	97021.69	115336.05	9160.00	425353.25
RLCSA	4302.62	24595.90	1304.99	198446.00	32463.90	8335.38	379402.00
Sad-CSA	617.53	9159.41	559.54	48222.97	**3041.82**	**314.59**	**6779.30**
SDSL-CSA	**455.08**	**1964.81**	**270.42**	**13406.39**	9178.77	701.12	23723.74

8　结论与进一步工作

本文提出了一种高阶熵压缩的全文自索引. 对于长为 n 的文本 T，我们的压缩索引可在线性时间内构造. 对于任意 $k \leq c\log_\sigma n - 1$ 和 $c < 1$，我们的压缩索引占用 $2nH_k + n + o(n)$ 位的空间，其中 H_k 表示文本 T 的 k 阶熵，σ 为字符表的大小. 此外，本文还给出了上述描述的压缩索引的一种实用改进. 这种实用改进引入了混合编码方法，能根据 1 在 gap 序列中的分布选择最佳的编码方法，其额外的空间开销为 $o(n)$ 位，可以忽略不计. 对于 *Pizza & Chili Corpus* 上的三类典型数据的实验表明，我们的压缩索引较之主流压缩索引在压缩率和查询时间上具有显著的优势.

虽然我们提出了一种高阶熵压缩的全文自索引，并开发出了相应的软件. 但还存在以下问题有待进一步深入研究. 首先，第一个问题是，运行时内存过大，严重影响其实用性. 这是由于在索引构建过程中，首先要建后缀数组，这就需要比原文本大 5 倍以上的空间. 其二，目前我们的方法只提供在压缩索引上支持精确模式匹配查询，下一步需要研究支持近似模式匹配查询的压缩索引. 这两个问题高效地解决后，我们的压缩索引才既能作为文档检索的基础，也能行使倒排索引的功能并可能应用于搜索引擎，且比倒排索引支持更为广泛的查询.

致　谢　各位评阅人给出了诸多建设性的意见，在此致谢！

参　考　文　献

[1] Jagadish H V, Gehrke J, Labrinidis A, et al. Big data and its technical challenges. Communications of the ACM, 2014, 57(7): 86-94

[2] Hon W-K, Shah R, Vitter J S. Compression, indexing, and retrieval for massive string data//Proceedings of the Annual Conference on Combinatorial Pattern Matching. New York, USA, 2010: 260-274

[3] Knuth D E, Morris J H, Pratt V R. Fast pattern matching in string. SIAM Journal on Computing, 1977, 6(2): 323-350

[4] Boyer R S, Moore J S. A fast string searching algorithm. Communications of the ACM, 1977, 20(10): 762-772

[5] McCreight E M. A space-economical suffix tree construction algorithm. Journal of the ACM, 1976, 23(2): 262-272

[6] Ukkonen E. On-line construction of suffix trees. Algorithmica, 1995, 14(3): 249-260

[7] Manber U, Myers G. Suffix arrays: A new method for on-line search. SIAM Journal on Computing, 1993, 22(5): 935-948

[8] Grossi R, Vitter J S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing, 2005, 35(2): 378-407

[9] Grossi R, Vitter J S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching//Proceedings of the ACM Annual Symposium on Theory of Computing. Portland, USA, 2000: 397-406

[10] Grossi R, Gupta A, Vitter J S. High-order entropy-compressed text indexes//Proceedings of the SIAM/ACM Annual Symposium Discrete Algorithms. Baltimore, USA, 2003: 841-850

[11] Rao S S. Time-space trade-offs for compressed suffix arrays. Information Processing Letters, 2002, 82(6): 307-311

[12] Sadakane K. Compressed text databases with efficient query algorithms based on the compressed suffix array//Proceedings of the International Symposium on Algorithms and Computation. Taipei, China, 2000: 410-421

[13] Sadakane K. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms, 2003, 48(2): 294-313

[14] Ferragina P, Manzini G. Opportunistic data structures with applications//Proceedings of the IEEE Annual Symposium on Foundations of Computer Science. Redondo Beach, USA, 2000: 390-398

[15] Ferragina P, Manzini G. Indexing compressed texts. Journal of the ACM, 2005, 52(4): 552-581

[16] Ferragina P, Manzini G. An experimental study of an opportunistic index//Proceedings of the SIAM/ACM Annual Symposium on Discrete Algorithm. Washington, USA, 2001: 269-278

[17] Mofat A, Zobel J. Self-indexing inverted files for fast text retrieval. ACM Transactions on Information Systems, 1996, 14(4): 349-379

[18] Zobel J, Mofat A. Inverted files for text search engines. ACM Computing Surveys, 2006, 38(2): Article 6

[19] Patil M, Thankachan S V, Shah R, et al. Inverted indexes for phrases and strings//Proceedings of the 34th Annual ACM SIGIR Conference. Beijing, China, 2011: 555-564

[20] Burrows M, Wheeler D J. A block sorting lossless data compression algorithm. Digital Equipment Corporation, Palo Alto: Technical Report SRC-RR-124, 1994

[21] Foschini L, Grossi R, Gupta A, Vitter J S. When indexing equals compression: Experiments with compressing suffix arrays and applications. ACM Transactions on Algorithms, 2006, 2(4): 611-639

[22] Kärkkäinen J, Puglisi S J. Fixed block compression boosting in FM-indexes//Proceedings of the International Conference on String Processing and Information Retrieval. Pisa, Italy 2011: 174-184

[23] Mäkinen V, Navarro G, Sirén J, Välimäki N. Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology, 2010, 17(3): 281-308

[24] Huo Hong-Wei, Chen Long-Gang, Vitter J S, Nekrich Y. A practical implementation of compressed suffix arrays with applications to self-indexing//Proceedings of the IEEE Data Compression Conference. Snowbird, USA, 2014: 292-301

[25] Ferragina P, González R, Navarro G, Venturini R. Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics, 2008, 13(12): 1-12

[26] Navarro G, Mäkinen V. Compressed full-text indexes. ACM Computing Surveys, 2007, 39(1): Article 2

[27] Gog S, Navarro G. Improved and extended locating functionality on compressed suffix arrays//Proceedings of the International Symposium on Experimental Algorithms Copenhagen. Denmark, 2014: 436-447

[28] Gog S, Petri M. Optimized succinct data structures for massive data. Software: Practice and Experience, 2014, 44(11): 1287-1416

[29] Manzini G. An analysis of the Burrows-Wheeler transform. Journal of the ACM, 2001, 48(3): 407-430

[30] Elias P. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 1975, 21(2): 194-203

HUO Hong-Wei, born in 1963, Ph. D., professor. Her research interests include design and analysis of algorithms, compressed indexs and retrievals for big data, compressed data structuress, external memory algorithms, bioinformatics algorithms, algorithm engineering.

CHEN Xiao-Yang, born in 1991, Ph. D. candidate. His research interests include compressed indexes and retrievals, graphing indexing, and external memory algorithms.

CHEN Long-Gang, born in 1988, M. S. candidate. His research interests include compressed indexes and retrievals.

YU Qiang, born in 1983, Ph. D., lecturer. His research interests include bioinformatics algorithms and parallel algorithms.

Background

This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61173025 and 61373044.

Massive data sets are being produced at unprecedented rates from sources like the World-Wide Web, social network, genome sequencing, XML, e-mail, satellite data, and business records. A larger part of the data consists of text in the form of a sequence of symbols representing not only natural language, but also music, program code, multimedia streams, biological sequences, and myriad forms of media. Proliferation of data as massive scales poses serious new challenges in terms of storing, managing, retrieving, and search for available information from the data.

From its inception, Pattern Matching grappled with understanding the nature of searching large corpora of heterogeneous data. When one is searching for a specific word in a small file, no special algorithms are needed. However, when seeking a gene in the human genome, where there is no separation to words, and where the data and the sought pattern are very large, understanding how to re-use data that had already been scanned, is critical. Thus, the importance of such algorithms as the aforementioned Boyer-Moore and KMP algorithms is reflected. When more complex questions arose, such as given a very large non-textual database (such as numerical data, audio data, or biological sequences) can one index it allowing for fast answers to search queries? This question led to understanding the nature of subwords and discovery of data structures as the suffix tree and suffix array.

The best-known full-text indexes are the suffix tree and suffix array, which support pattern matching queries in optimal or almost-optimal time. Both structures use $O(n)$ words of storage, which is $O(n\log n)$ bits, which is larger than the raw text size $n|\Sigma|$ bits, where Σ represents the text alphabet. In practice, the size of suffix trees and suffix arrays can be prohibitively large, often 5—20 times larger than the data they index. The exciting new field called compressed indexes and retrieval, which addresses the bloat exhibited by suffix trees and suffix arrays. There are two simultaneous goals: space-efficient compression and fast indexing.

The field of compressed or succinct data structures attempts to build a data structure whose space is provably close to the size of the data in compressed format and that still provides fast query functionality. Theoretical break-

throughs about 15 years ago led to the development of a new generation of space-efficient search indexes. The compressed suffix array and the FM-index have been developed to achieve this desired goal of compressed text indexing, and their query time is proportional to the query pattern size plus the product of the output size and a small polylog function of n. They provide random access to any part of the original text, and thus the text becomes redundant and can be discarded.

In this paper we present a high-order entropy-compressed full-text self-indexes. For a text of n characters, our compressed self-indexes need $2nH_k(T)+n+o(n)$ bits of space for any $k\leq c\log_\sigma n-1$ and any constant $c<1$, where $H_k(T)$ denotes the kth order empirical entropy.

In addition, the proposed compressed indexes can be constructed in linear time. We also give a practical improvement on the compressed indexes described above. Our practical improvement introduces hybrid encoding methods, which are chosen according to the distribution of one in the gap sequence. The extra cost needed is $o(n)$ bits. Experiments on three typical data on the *Pizza &Chili Corpus* show that our compressed indexes have significant advantages in terms of compression and query time over the state-of-the-art indexing technologies. The source code is available online.