**基于后缀数组索引模型的DNA序列匹配**

**摘要**

随着人类基因组计划(HumanGenomeProj eot, HGP)的顺利实施和信息技术的迅速发展,大量DNA序列数据被人们发掘出来。对这些DNA序列数据进行科学有效的分析和处理, 是生物信息学的重要研究方向。对基因序列进行有效且快速的比对,即基因序列的相似性分析是其热门课题之一。本文提出一种基于后缀数组索引模型的DNA序列快速查找算法。在100万个长度为100的DNA序列数据匹配中，有着查找准确，速度快，空间占用小的优势。

在大量DNA序列中建立k-mer索引，迅速定位目标k-mer，该问题可归结为模式匹配问题，即在一个较大的文本中，通过建立索引，查找模式字符串出现的位置。本文对此建立了基于倍增算法的后缀数组索引模型。在准确查找目标k-mer的前提下，利用后缀数组的结构特性，达到速度快，空间占用小的预期。通过实验验证模型，在计算机内存不超过500MB的条件下，可支持k的范围是2到100，占用内存的大小与k值成负相关。平均索引时间为21s，检索时间为0.9s。

建立索引阶段对每个DNA序列依据给定k值建立了基于倍增算法的后缀数组,长度为100-k+1,在100万次建立之后将离散化的后缀数组聚集，保存到一个index索引数组。时间复杂度为O(nlogn)，空间复杂度为O(n)。相比较哈希（散列）表，哈希树，Trie树，后缀树索引，本算法在保持检索速度和正确率的前提下，在实现难度和占用内存方面有着显著的优势。

检索阶段利用后缀数组相应后缀聚类的特性，即含有相同前缀的后缀在后缀数组中连续出现，我们选择二分检索算法，迅速命中目标子区间，时间复杂度为O(logn)，空间复杂度为O(n)。

本文借用KMP在线检索的方式检验模型结果的正确性，结果不重不漏。

最后，本文对后缀数组索引模型进行了相应的拓展。它们均可通过本文算法的改进去实现。

**关键字**：索引 后缀数组 倍增算法 基数排序 二分检索

**一．问题重述**

给定一个只含有4个字母ATCG的DNA序列，如 *S =*“CTGTACTGTAT”。*k-*mer：给定一个整数值*k*，从*S*的第一个位置开始，取一连续*k*个字母的短串，（如*k*= 5，则此短串为CTGTA）。然后从*S*的第二个位置，取另一*k-*mer（如*k*= 5，则此短串为TGTAC），这样直至*S*的末端，就得一个集合，包含全部*k-*mer。通过对这些*k-*mer建立一种数据索引，可以快速查找目标*k-*mer，返回其起始位置。

**问题及要求：**

现在以文件形式给定 100万个 DNA序列，序列编号为1-1000000，每个基因序列长度为100 。

（1）要求对给定k，用一种支持一个k值的索引方法，返回任意一个k-mer所在的DNA序列编号和相应序列中出现的位置。索引建立后，查询速度尽量快，所用内存尽量小。

（2）给出建立索引所用的计算复杂度，和空间复杂度分析。

（3）给出使用索引查询的计算复杂度，和空间复杂度分析。

（4）假设内存限制为8G，分析所设计索引方法所能支持的最大k值和相应数据查询效率。

（5）模型评价的重要性由高到低排列为，索引查询速度，索引内存使用，8G内存下所能支持的k值范围和建立索引时间

综上所述，该问题为模式匹配问题，即在一个较大的文本中，查找模式字符串的出现位置。文本中内容为AGCT四种字符，模式字符串的长度和内容由用户指定，检索时需要通过建立索引，每次建立索引只需要支持固定长度字符串的查询即可。

**二．模型的基本假设**

针对本问题，建立如下假设：

1. 默认用户输入一个k值，索引建立一次。
2. 索引建好后以文件形式给定的原 100万个 DNA序列不会被修改（如进行插入，删除操作）。
3. 用户计算机内存大于500M。

**三．符号说明**

|  |  |  |
| --- | --- | --- |
| 符号 | 表示意义 | 备注 |
| r[i..j] | 子串 | 表示r 串中从i 到j 这一段，也就是顺次排列r[i],r[i+1],...,r[j]形成的字符串 |
| Suffix(i) | 后缀是指从某个位置i 开始到整个串末尾结束的一个特殊子串 | 字符串r 的从第i 个字符开始的后缀  Suffix(i)=r[i..len(r)] |
| Sa[i] | 后缀数组 | SA 是一个一维数组，它保存1..n 的某个排列SA[1]，SA[2]，……，SA[n]，并且保证Suffix(SA[i]) <Suffix(SA[i+1])，1≤i<n。即将S 的n 个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入SA 中。 |
| Rank[i] | 名次数组 | Rank[i]存储的是Suffix(i)在所有后缀中从小到大排列的“名次”。 |
| DNA[i] | 百万DNA数组 | 存储文件中100万条DNA序列 |
| Index[i] | 索引数组 | 存储100万条DNA序列的索引，即后缀数组的集合 |
| binSearch() | 二分检索函数 | 建立索引后的检索函数，索引后通过对目标k-mer与索引中序列的大小比较实现二分查找 |
| [lt，rt] | k-mer区间 | 目标k-mer在后缀数组中的定位区间 |
| da() | 倍增算法函数 | 求出sa数组 |

注：简单的说，后缀数组是“排第几的是谁？”，名次数组是“你排第几？”。容

易看出，后缀数组和名次数组为互逆运算。如图2所示。

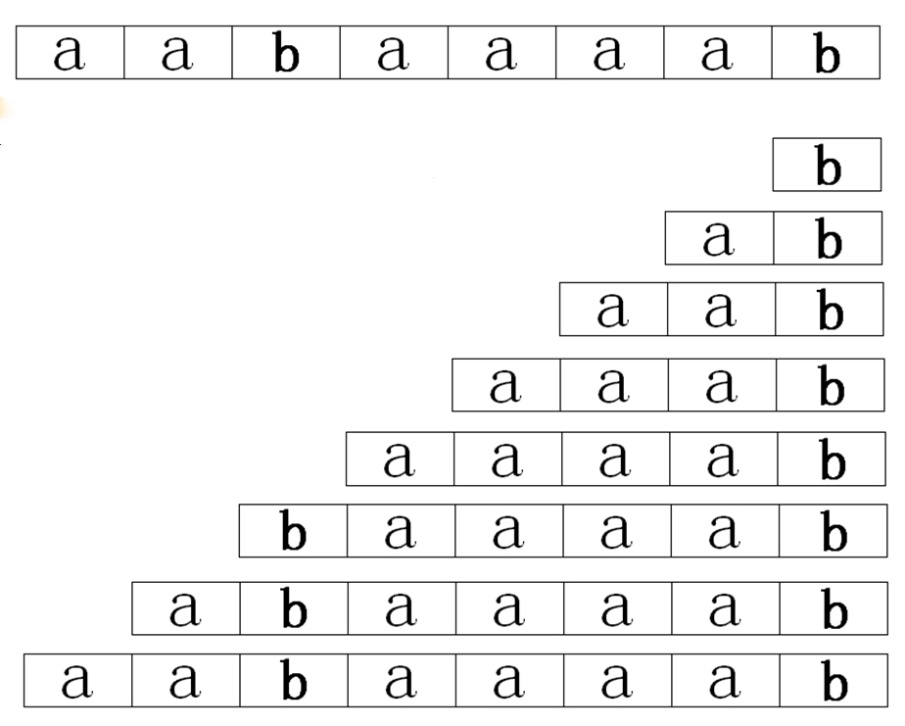


图1：后缀示例：字符串"aabaaaab"的Suffix(i)



图2：sa,rank数组示例

**四．问题的分析及模型的建立**

**4.1 问题的分析**

作字符串匹配，最直观的思想就是，枚举可能出现的子串，顺序扫描原文进行串匹配，匹配成功则输出该子串所在位置。该思想可能采用的算法包括，KMP算法，Boyer-Moore算法，Commentz-Walter等基于子串的算法，基于有限自动机的算法等，即都采用简单地顺序扫描匹配思想。使用这类算法，不需要预先对文本信息进行预处理。KMP算法的基本思想是通过一个窗口的字符串上滑动，利用先前检查的信息，从而跳过了一些不必要检查的位置，方法容易实现，可在O(m+n)时间内用O(m)的空间完成对文本的扫描。

这种基于子串的匹配算法缺点是每次匹配均需要对文本进行扫描，不适合文本非常巨大的情况，随着文本长度增加，查找效率降低。

为了提高查找速率和空间利用率，可以采用在文本中建立一种便于查找的数据结构——索引。如果文本规模较大且不更新的情况下，建立索引这种数据结构来代替顺序扫描过程，可以大大提高查找速度。

**4.2DNA索引分析**

题目要求建立数据索引后检索，减少以后多次搜索不同目标k-mer的时间，提高搜索速度，即进行所谓的线下检索。由此我们可以先否定采用KMP，BM等在线算法求解。

线下检索的索引结构种类很多，哈希（散列）表，哈希树，Trie树，后缀树，后缀数组，q-gram索引等等，哈希（散列）表，哈希树，Trie树占用内存较多，经常应用于生物基因研究的基于Ukkonen算法的后缀树实现较繁琐且空间开销较大，经过内存与复杂度分析后，我们选择内存占用较小，构造简单，效率较高，处理字符串有力工具----后缀数组。它将原始数据处理成一个高效的交叉引用的查找结构以便于快速查找。

**4.2.1后缀数组背景：**

后缀数组是一种较新的建立全文索引的方法。它由文本的所有后缀（起点在文本任意位置，终点在文本末尾的字符串）的字典排序而得，后缀的开头位置按照顺序放入后缀数组。后缀数组索引具有较高的检索效率并且更适合如范围查找，模糊查找等较复杂的查找方式，在词法扫描,模式匹配，Web搜索和文献检索等方面有着广泛的用途。[6]

**4.3用倍增算法实现后缀数组索引模型：**

构造后缀数组的方法分为倍增算法和DC3算法，本题采用后缀数组的倍增算法。

**倍增算法：**

1. 算法目标：

设字符串长度为n，在O(nlogn)时间复杂度内求得串的sa数组。

1. 大小比较：

按字典序比较。就是对于两个字符串u、v，令i 从1 开始顺次比较u[i]和v[i]，如果u[i]=v[i]则令i 加1，否则若u[i]<v[i]则认为u<v，u[i]>v[i]则认为u>v，比较结束。如果i>len(u)或者i>len(v)仍比较不出结果，那么若len(u)<len(v) 则认为u<v ，若len(u)=len(v) 则认为u=v ，若len(u)>len(v)则u>v。

从定义来看，字符串S 的两个开头位置不同的后缀u 和v 进行比较的结果不可能是相等，因为u=v 的必要条件len(u)=len(v)在这里不可能满足。

1. 前提：

已知待匹配的字符串长度k，故在100个长度的DNA字符串中Suffix(k+1)，Suffix(k+2)…Suffix(n)不可能与待匹配的字符串相匹配（长度不够），故sa数组长度length=100-k+1。

1. 算法思想：

用倍增的方法对每个字符开始的长度为的子字符串进行排序，求出排名，即rank 值。k 从0 开始，每次加1，当大于length后，以每个位置开始的长度为的子字符串便相当于所有的后缀。并且这些子字符串都一定已经比较出大小，即确保rank 值中没有相同的值，那么此时的rank 值就是最后的结果。

倍增长度H,利用之前排序H/2长度后得到的rank数组作为关键字，把后H/2部分作为第二关键字,把前H/2部分作为第一关键字,对H长度的子串作排序。即每一次排序都利用上次长度为的字符串的rank 值，那么长度为的字符串就可以用两个长度为的字符串的排名作为两个关键字表示，然后进行基数排序，便得出了长度为的字符串的rank 值。

以字符串“aabaaaab”为例，整个过程如图3所示。其中x、y 是表示长度为的字符串的两个关键字。[3][4][5][6]

****

图3：倍增算法的实现过程示例

1. 倍增算法的具体实现步骤：

算法开始：

Step1：将待排序的字符串放在r 数组中，从r[0]到r[n-1]，长度为n，且最大值小于m。为了函数操作的方便，约定除r[n-1]外所有的r[i]都大于0 r[n-1]=0（相当于哨兵）。

Step2：对长度为1 的字符串进行排序。在本题中，字符仅为ACGT，种类数目最

多只有4个，r 的最大值为T的ASCII码值，所以这里优先采用了基数排序，每

一轮的时间复杂度降为O(n)，总时间复杂度为O(nlogn)。

Step3：进行若干次基数排序，基数排序要分两次，第一次是对第二关键字排序，第二次是对第一关键字排序。直接利用上一次得到的sa数组求得第二关 键字。

Step4：对第一关键字进行基数排序。

Step5：在求出sa 后，计算rank值。必须比较两个字符串是否完全相同。

Step6：rank值保存在x 数组中，而变量p 的结果实际上就是不同的字符串的个数。如果p 等于n，那么函数可以结束。因为在当前长度的字符串中，已经没有相同的字符串，接下来的排序不会改变rank值。

Step7：索引放在sa 数组中，从sa[0]到sa[n-1]。算法结束

倍增算法流程图如图4所示：



图4：倍增算法实现流程图

本后缀数组模型着眼于100万条DNA序列个体，利用倍增算法在每条DNA序列上建立长度为n - k + 1的后缀数组索引，以“序列号\*索引长度+sa数组下标”为下标将sa数组值对应存入index数组的方式连续存入100万条DNA序列的后缀数组索引，创建index索引数组。

**4.4 DNA检索分析**

后缀数组对应的后缀是以字典序排序的，所以含有相同前缀的后缀在后缀数组中是连续出现的，形成相应序列的聚类，即后缀数组的字符串查询结果是后缀数组的一个子区间。通过二分查找可以得到以序列P为前缀的后缀数组区间[8]。本组解决此题的查询思想为先二分，成功定位到目标区间，利用落到目标区间的位置做左右扩展，直到（相应后缀的前缀与目标序列P不匹配），从而得到区间。设区间左边界lt，区间右边界rt。检索到区间为{sa[lt],sa[lt+1]…sa[rt]}

**4.5基于后缀数组索引的检索实现**

用户输入序列，依次从index数组中取出索引数组，对每一行DNA序列进行二分检索，循环100万次。

**二分检索算法：**

1. 算法思想：

为了提高检索效率，在一个DNA序列中匹配时，先将DNA后缀序列按照字典序升序排序,建立索引，将索引数组sa的中点设置为比较的对象，如果要找的序列的值小于该中点元素对应的字符串位置，则将待查序列缩小为sa数组左半部分，否则为sa数组右半部分。即假如DNA序列中某个位置开头的子序列是要检索的数据,则返回该位置，否则比较的结果排除掉sa序列（也是DNA序列）一半的元素，再在余下的一半sa数组中取中间的一个元素作为DNA序列起点进行比较，并根据比较的结果再次排除一半的sa数组元素，以此类推，直至最终找到或将检索的sa数组范围缩小到0（即未找到）。这就是二分查找（Binary Search），由于二分查找法每次都根据比较结果排除一半的数据，也称为“折半查找法”。

二分查找优点是比较次数少，查找速度快，平均性能好。其缺点是要求待查表为有序表，且插入删除困难。由于在上述模型中不需要插入且待查表已有序。所以采用二分查找具有优势。

1. 前提：

Sa数组按顺序存储了以字典序升序排列后缀数组后后每个后缀数组在本行DNA序列中出现的位置。

升序排序后DNA序列已经聚类，即只要匹配一个成功，则其左右均有可能是匹配序列。

二分查找的范围为0至100-k+1，需要匹配的字符串不会出现在大于100-k+1以后的位置。

循环100万次通过倍增算法求出sa数组，将sa数组序列依次存入index数组，建立索引，此时index数组保存了DNA序列每一行的索引。若k不变则只需建立一次索引。

1. 算法步骤：

Step1：首先是确定整个查找区间的中间位置

mid=(left+right)/2

sa[mid]定位为查找DNA序列r的起始位置

Step2：用待查DNA序列p与step1找到的位置的DNA序列r进行比较：若相等，则返回mid；若大于，则在后半区域继续进行二分查找；若小于，则在前半区域继续进行二分查找。重置 left和right。若最终left=right，仍未匹配，返回-1

Step3：检测二分查找返回值dot是否为-1，若不为-1，说明查找成功，再对返回值进行左右扩展，sa[dot-] sa[dot+]均可能为匹配序列。至此，一条100长度的DNA序列即检索完毕，再次循环检索下一条DNA序列

二分检索算法流程图如图5所示：



图5：二分检索算法流程图

1. 算法分析：

二分检索的时间复杂度为O(log n) n为区间长度，即100-k+1；需要循环执行调用1000000次二分检索函数。

**五．模型求解**

**5.1建模思想和模型建立：**

第一步：将两个文件DNA序列合并存储到DNA数组内。

第二步：输入k值

第三步：对100万条DNA序列基于倍增算法构建后缀数组。将离散的后缀数组聚集创建index索引数组，其下标隐含地记录了序列号，存储的数组元素即为位置信息。

第四步：输入目标k-mer

第五步：读取index索引数组，还原后缀数组，对其二分检索，找到目标自区间

输出序列号及其位置。

结束

模型建立的基本原理如图6表示：



图6：模型建立流程图

**5.2问题一**

**5.2.1问题分析**

要求对给定k，用一种支持一个k值的数据索引方法，返回任意一个k-mer所在的DNA序列编号和相应序列中出现的位置，为了达到查询速度快，所用内存小，我们联想到运用后缀数组的数据结构。

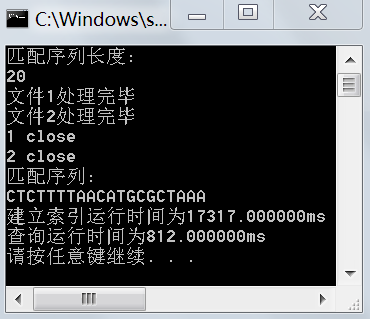
**5.2.2问题的解答**

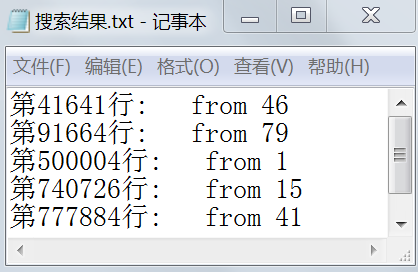
模型中主体分为后缀数组索引构建和二分检索两部分，索引以“序列号\*索引长度+sa数组下标”为下标将sa数组值对应存入index。其中index数组下标隐含后序列号信息，数组值即为位置信息。用二分检索迅速找到索引后形成的子区间[lt，rt]，即可返回k-mer所在的DNA序列编号和相应序列中出现的位置。

模型中的索引针对给定k值，剔除索引中后缀的长度小于k的元素，使得每条索引的长度为n-k+1，从而简化索引，达到随着k值增大，索引占用的内存越小的效果。

由于k过小时，返回的位置非常多不便于枚举，在此选择k=20 k-mer：CTCTTTTAACATGCGCTAAA

程序结果举例如下：





**5.3问题二**

**5.3.1问题分析**

建立索引所用的计算复杂度，和空间复杂度分析。

**5.3.2问题解答**

倍增算法的时间复杂度比较容易分析。每次基数排序的时间复杂度为O(n)，排序的次数决定于最长公共子串的长度，最坏情况下，排序次数为logn 次，所

以总的时间复杂度为O(nlogn)。倍增算法的空间复杂度是O(n)。

**5.4问题三**

**5.4.1问题分析**

使用索引查询的计算复杂度，和空间复杂度分析。

**5.4.2问题解答**

二分查询的时间复杂度O(logn)，空间复杂度为O(n)

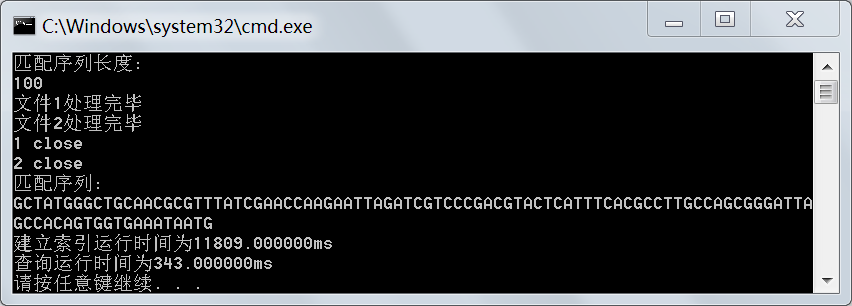
**5.5问题四**

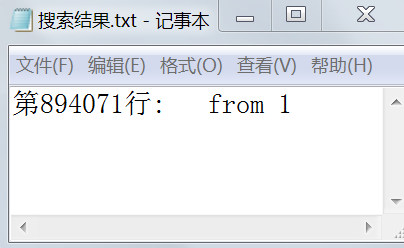
**5.5.1问题分析**

假设内存限制为8G，分析所设计索引方法所能支持的最大k值和相应数据查询效率。

**5.5.2问题解答**

最大k值为100，多次测试结果表明当k=100时，二分查询效率较高，平均用时约为0.35s。





**5.6问题五**

**5.6.1问题分析**

按重要性由高到低排列，依据索引查询速度，索引内存使用，8G内存下，所能支持的k值范围，建立索引时间，来评价索引方法性能。

**5.6.2问题解答**

实验结果：

|  |  |  |  |
| --- | --- | --- | --- |
| k | 建立索引时间（s） | 查询时间（s） | 占用内存（G） |
| 2 | 18.436 | 9.402 | 0.4634 |
| 3 | 18.084 | 3.055 | 0.4597 |
| 4 | 17.894 | 1.398 | 0.4559 |
| 5 | 19.381 | 1.018 | 0.4522 |
| 6 | 19.162 | 0.910 | 0.4485 |
| 7 | 18.470 | 0.904 | 0.4447 |
| 8 | 19.312 | 0.935 | 0.4408 |
| 9 | 20.701 | 0.888 | 0.4373 |
| 10 | 18.825 | 0.862 | 0.4335 |
| 15 | 18.002 | 0.839 | 0.4149 |
| 20 | 18.096 | 0.813 | 0.3962 |
| 30 | 18.548 | 0.773 | 0.3589 |
| 40 | 19.984 | 0.747 | 0.3216 |
| 50 | 22.652 | 0.737 | 0.2842 |
| 60 | 25.069 | 0.670 | 0.2469 |
| 70 | 27.987 | 0.608 | 0.2096 |
| 80 | 28.810 | 0.576 | 0.1723 |
| 90 | 29.563 | 0.449 | 0.1349 |
| 100 | 11.809 | 0.343 | 0.1094 |

注：

建立索引时间不包括从文件读入内存的时间

实验环境：

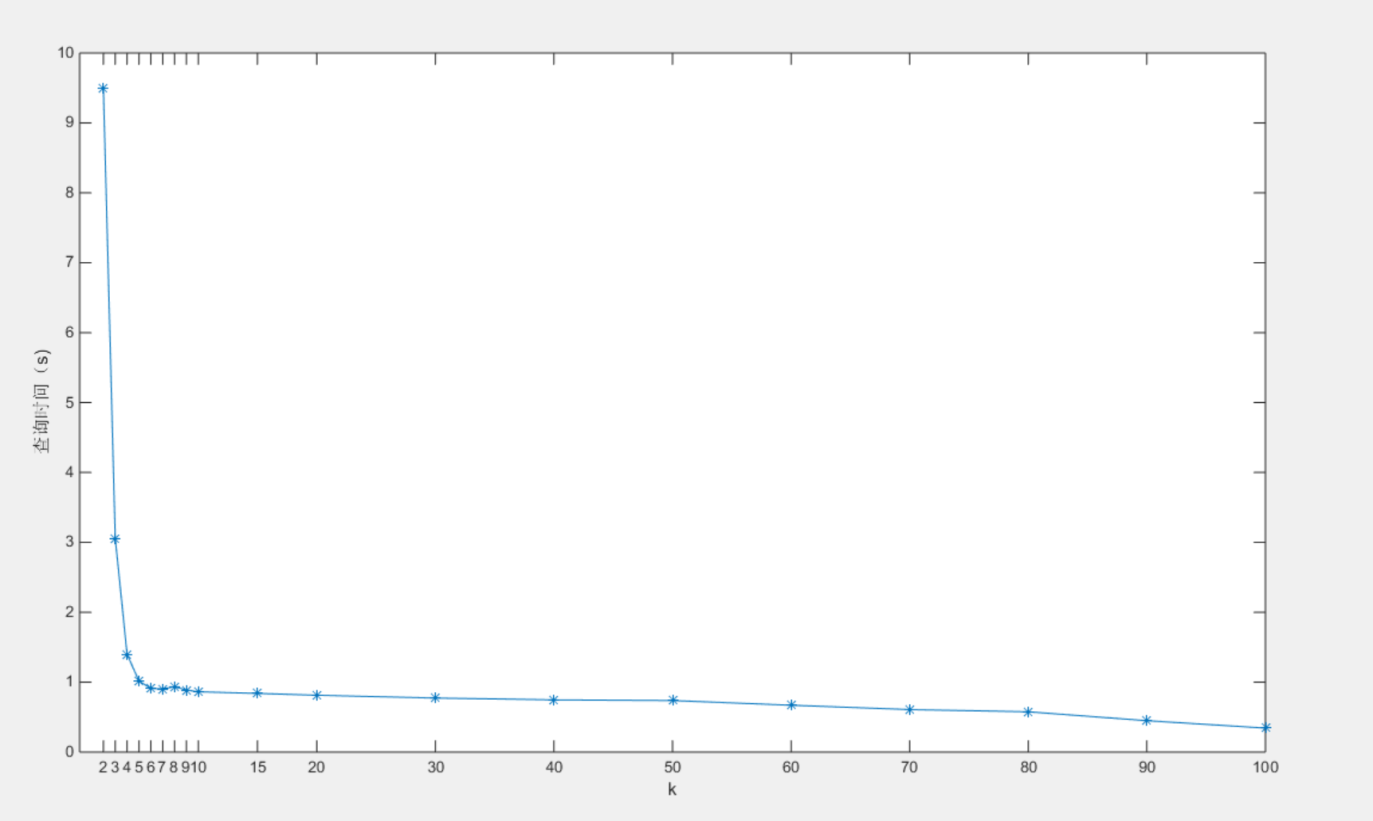
VS2013,处理器为lntel(R)Core(TM)i5-4258U CPU @2.40GHz 2.40GHz，内存大小为8GB，系统为64位win7操作系统。

检验：

对于数值较小的k，输出文件中查询结果数量庞大，对其抽样检验，与kmp在线算法结果比对，且比对成功。

对于数值较大的k，输出文件中查询结果数量较少，对其全体检验，与kmp在线算法结果比对，且比对成功。（kmp代码见附录3）

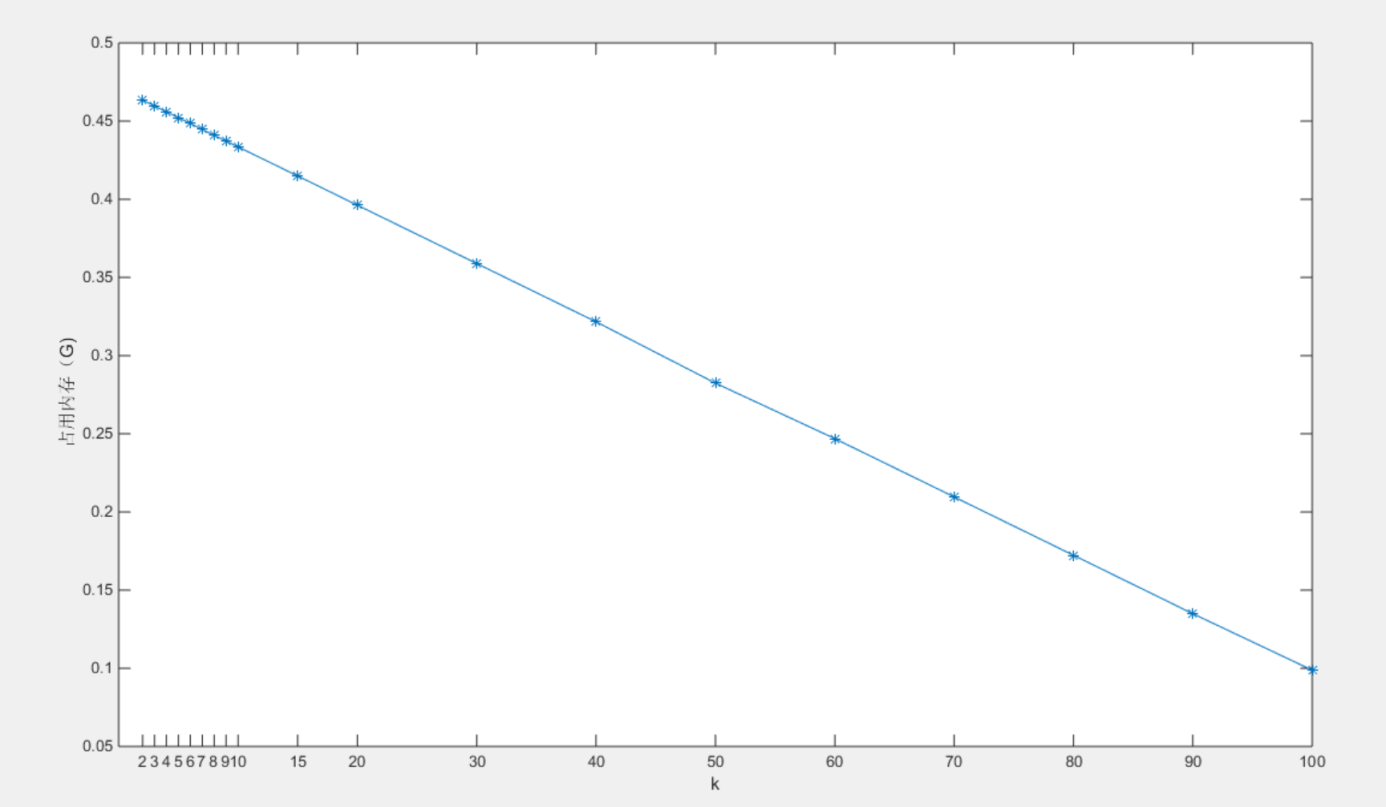
1. k值与查询时间的折线图：



索引查询速度

由求解表和折线图易知：当k值特别小时，查询时间较长，但对于绝大多数的k，查询时间在1s以内,且随k值的增大用时略有减少。

1. k值与占用内存的折线图：



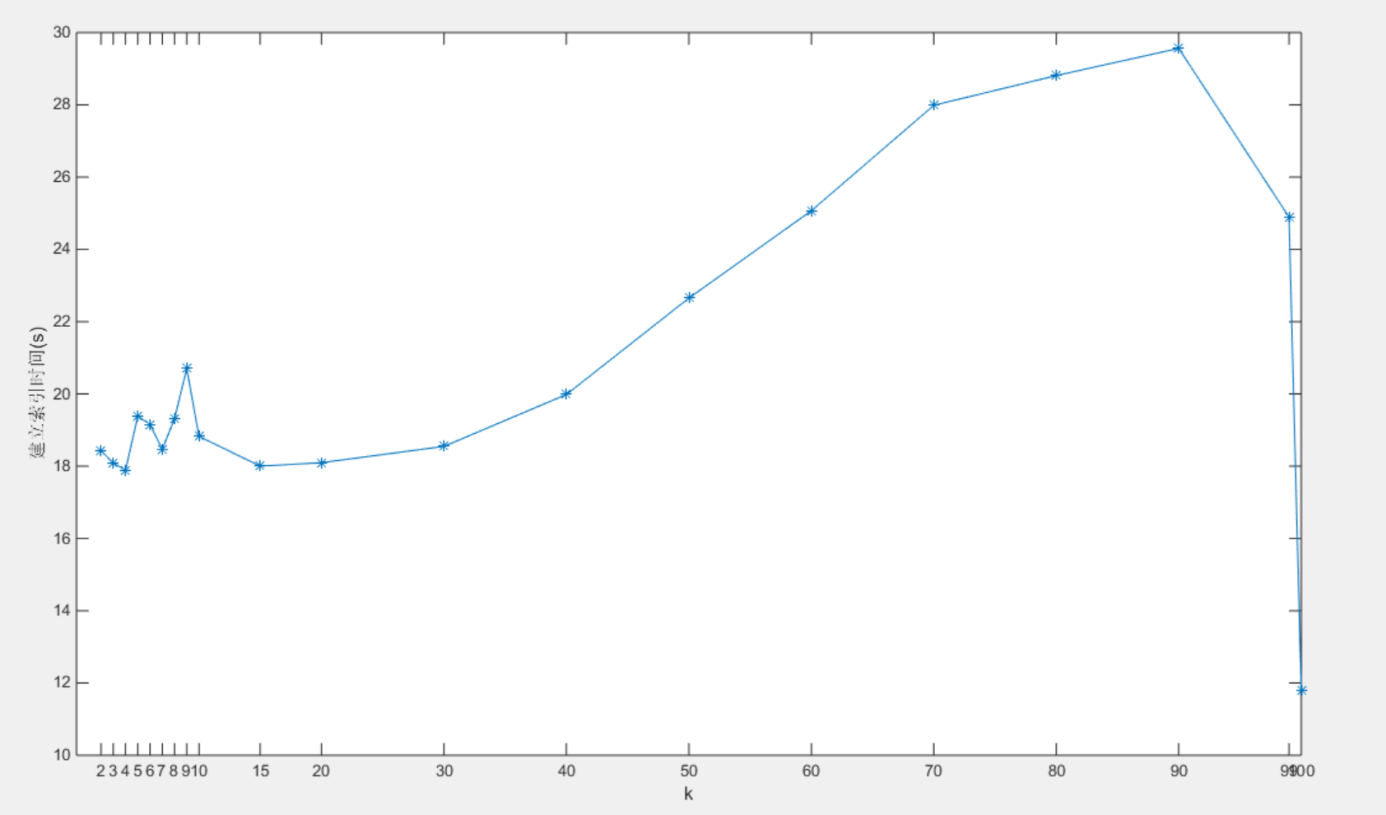
索引内存使用

索引占用内存小于500M，且随k值增大内存占用呈线性降低，当k为100时内存占用约为112M。

1. 8G内存下，所能支持的k值范围

2——100，处理较长的k-mer，模型的优势更加明显，查询效率升高，索引内存占用显著降低

1. k值与建立索引时间的折线图：



由折线图不难发现建立索引时间整体上逐渐增加,但始终维持在32s以内,k为100时发生突变。

六．模型的评价

**6.1优点**

1. 通过对问题的充分分析，选择新颖且功能强大的后缀数组作为索引的数组结构
2. 利用倍增思想，构建后缀数组。
3. 在保持检索速度较优的前提下，占用内存方面有着显著的优势
4. 建立索引速度快，整个查询时间短
5. 利用后缀数组的聚类特性，二分检索，检索效率较高
6. 不到500M的内存条件下最大可以支持的k为100
7. 模型的思想具有很强的实用性，在众多领域都具有优势

**6.2缺点**

1. 平均查询时间虽然维持在1秒左右,但相较于其他数据结构直接映射的O(1)查询复杂度，其查询时间可缩短几十倍。二分O(logn)的复杂度不占优势。
2. 用倍增算法内部的多关键字基数排序for循环很多，对于后缀数组的初步了解者不易理解其含义。
3. 建立索引为局部索引，即每个DNA序列的索引，仍需遍历每个DNA序列，时间开销过大。

七.模型的改进与推广

**7.1改进**

后缀数组最适合在海量信息中进行查找操作，因为处理的数据量异常庞大，必须充分考虑到时间复杂度和空间复杂度之间的平衡。在实际应用中，创建后缀数组的空间和时间开销仍然很大，必须对它加以改进，才能在实际应用中发挥更大的作用。

针对上文提到的一些缺点，我们可以对模型做出以下改进，使其更具备实用性。

* 空间改进：

压缩后缀数组，是后缀数组的压缩表示形式，由Grossi和Vitter在2000年提出，有效地解决了后缀数组空间开销大的问题，并且由于它支持对子串的查找使得应用范围更加广阔。缺点是在查询时需要将压缩后缀数组还原。

* 时间改进：

对后缀进行全排序，即将整个文件中DNA序列产生的后缀进行排序，再采用

调用一次二分检索函数查找。检索复杂度O(logn)，n最大为100000000。建立索引复杂度O(mlogn)，m最大为100000000，n最大为100。

* 索引建立改进：

对后缀进行排序时只对k长度的后缀进行比较。初步实现已附在附录二

**7.2推广**

简单的结构，高效的实现使得后缀数组的应用前景广阔，因此受到越来越多的关注。

* 模式匹配：

在信息检索，模式识别等众多领域均具有重要的价值，除了在生物信息学（基因数据库）领域应用，在文本数据库（电子式百科全书，网上词典），拼写检查，语言翻译，数据压缩，搜索引擎，入侵检测，内容过滤，计算机病毒特征码匹配等方面均有重要的作用

* Web搜索引擎的应用：

目前，对网上信息的高效、智能检索机制已成为计算机网络领域的研究热点。搜索引擎是帮助互联网用户查询信息的搜索工具，它搜集、发现信息，对信息进行理解、提取、组织和处理，为用户提供检索服务。由于网络信息量迅猛增加和网络信息组织的无序性，人们对信息服务的个性化，智能化的需求也越来越高，这就给搜索引擎的技术提出更高的要求。

目前,搜索引擎的核心模块(索引器)大多采用倒排文件结构，对短语查询的准确率较低。通过引入后缀数组技术进行全文索引，对搜索引擎建立索引的技术加以改进，大大提高了搜索引擎的查准率。同时为克服全文索引时占用空间大的缺点，再改进为压缩后缀数组技术，把后缀数组索引的大小压缩到了O(n)位。对比实验表明,基于压缩后缀数组的索引比传统倒排文件索引的短语查准率提高了近20%。

* 在计算机病毒特征码匹配的应用：

字符串匹配问题是计算机科学研究中最基础的问题之一，早期的研究多集中于精确字符串匹配领域，就精确字符串匹配提出了许多单模式和多模式匹配算法。然而在信息检索、计算机病毒特征码匹配等一些实际领域中有时更需要查找的是近似匹配字符串。因此研究高效的近似字符串匹配算法具有重要的理论价值和实际意义。

近似字符串的相关理论结合后缀数组索引结构，可以提出基于后缀数组的近似字符串匹配算法，。算法使用后缀数组加快动态规划矩阵上的迭代速度，用后缀数组构造算法可以预先排序文本串后缀，求解后缀数组中相邻后缀的最长公共前缀，通过加速对角线上扩展的方式降低了构造动态规划矩阵的时间消耗，减少占用内存。

* 数据压缩的应用：

滑动窗口压缩是一种依赖于两个滑动窗口来进行压缩的数据压缩方法。它包含两个窗口，一个窗口（DW）包含已输入数据流，另一个窗口（CW）包含待压缩编码的字符串。后缀数组由于其结构简单及空间效率较好的特点，已经结合到窗口数据压缩法中，使得算法通过查找在DW中与CW相同的字符数据的时间与内存减少。

**参考文献**

［1］刘汝佳，《算法艺术与信息学竞赛》，北京：清华大学出版社，2004

［2］许智磊，IOI2004 国家集训队论文《后缀数组》

［3］罗穗骞， IOI2009国家集训队论文《后缀数组——处理字符串的有力工具》

［4］刘汝佳，陈锋，《算法竞赛入门经典训练指南》，北京：清华大学出版社，2012，（219-227）

［5］王建德，吴永辉，《程序设计中实用的数据结构》，北京：人民邮电出版社,2012，（71-96）

［6］吴永辉，王建德，《算法设计编程实验》，机械工业出版社，2013，（273-287）

［7］黄影，《生物序列索引结构构造算法研究》，西安电子科技大学硕士学位论文，2009

［8］郭鹏飞，《一种基于后缀数组和倒排表的全文索引模型》，北京交通大学硕士学位论文，2014

［9］张任文，《生物序列索引结构的研究和实现》，哈尔滨工业大学硕士学位论文，2006

**附录：**

|  |
| --- |
| 附录1 |
| 模型全部程序(C/C++程序实现) |
| //Visual Studio 2013编译通过  #include"stdio.h"  #define maxn 100  #include<time.h>  #include<cstdlib>  #include<string.h>  int wa[maxn], wb[maxn], wv[maxn], ws[maxn];  int cmp(int \*r, int a, int b, int l)  {  return r[a] == r[b] && r[a + l] == r[b + l];  }  void da(int \*r, int \*sa, int n, int m, int length)  {  //r为字符串数组，sa为后缀数组，n=strlen(s)+1，m为max(r[i])+1。  int i, j, p, \*x = wa, \*y = wb, \*t;  //对长度为1的字符串基数排序。  for (i = 0; i<m; i++)  ws[i] = 0;//清零。  for (i = 0; i<n; i++)  ws[x[i] = r[i]]++;//统计各相同字符的个数。  for (i = 1; i<m; i++)  ws[i] += ws[i - 1];//统计小于等于i的字符共有多少个。  for (i = n-1; i >= 0; i--)  sa[--ws[x[i]]] = i;//小于等于r[i]共有ws[x[i]]个，因此r[i]排在第ws[x[i]]个。    for (j = 1, p = 1; p<length; j \*= 2, m = p)//p是第二关键字为0的个数，j是当前比较的字符串长度。  {  //直接利用sa数组排序第二关键字  for (p = 0, i = n - j; i<n; i++)  y[p++] = i;  for (i = 0; i < n; i++)  {  if (sa[i] >= j)  y[p++] = sa[i] - j;  }  //基数排序第一关键字  for (i = 0; i<n; i++)  wv[i] = x[y[i]];  for (i = 0; i<m; i++)  ws[i] = 0;  for (i = 0; i<n; i++)  ws[wv[i]]++;  for (i = 1; i<m; i++)  ws[i] += ws[i - 1];  for (i = n - 1; i >= 0; i--)  sa[--ws[wv[i]]] = y[i];    //根据sa和y数组计算新的x数组  for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i<n; i++)  x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;  }  //去掉多余项  for (i = 0; i < length; i++)  {  if (sa[i] >= length)  {  for (j = i; j < n-1;j++)  sa[j] = sa[j + 1];  i--;  }    }  return;  }  //二分检索函数  int binSearch(char \*r, char p[], int sa[], int k, int n)  {  int low = -1;  int high = n;  while (low + 1 != high)  {  int i = (low + high) / 2;  if (strncmp(p, &r[sa[i]], k)<0)  {  high = i;  }  if (strncmp(p, &r[sa[i]], k) == 0)  {  return i;  }  if (strncmp(p, &r[sa[i]], k) > 0)  {  low = i;  }  }  return -1;  }  char s[maxn];  char p[100];  int r[100], sa[100];  int index[100000000];  char DNA[100000000];  //读入文件到内存  void PreTreat(){  FILE \*fSource1, \*fSource2;  if ((fSource1 = fopen("solexa\_100\_170\_1.fa", "r")) == NULL)  {//文件1  printf("Can't open the solexa\_100\_170\_1.fa!\n");  getchar();  exit(-1);  }  if ((fSource2 = fopen("solexa\_100\_170\_2.fa", "r")) == NULL)  {//文件2  printf("Can't open the solexa\_100\_170\_1.fa!\n");  getchar();  exit(-1);  }  int i;  char ch;  char str[100];  //处理文件1 的 500000条  for (i = 0; i<500000; i++)  {  while ((ch = fgetc(fSource1)) && (ch<'A' || ch>'T'));  fgets(str, 100, fSource1);  DNA[100 \* i] = ch;  for (int j = 1; j < 100; j++)  {  DNA[100 \* i + j] = str[j - 1];  }  }  printf("文件1处理完毕\n");  //处理文件2 的 500000条  for (i = 500000; i<1000000; i++)  {  while ((ch = fgetc(fSource2)) && (ch<'A' || ch>'T'));  fgets(str, 100, fSource2);  DNA[100 \* i] = ch;  for (int j = 1; j < 100; j++)  {  DNA[100 \* i + j] = str[j - 1];  }  }  printf("文件2处理完毕\n");    fclose(fSource1);  printf("1 close\n");  fclose(fSource2);  printf("2 close\n");  }  int main()  {  clock\_t start, finish, middle,middle2;  double duration, duration1;  int i, length, n, k;  srand((unsigned)time(0));  printf("匹配序列长度:\n");  scanf("%d", &k);  n = 100;  length = n - k + 1;//索引长度  FILE \*fResult;  fResult = fopen("搜索结果.txt", "w");  PreTreat();//读两个文件  start = clock();  for (int line = 1; line <= 1000000; line++)//读1000000行  {  for (int j = 0; j < 100; j++)  {  r[j] = DNA[100 \* (line - 1) + j];  }    da(r, sa, n, 85,length);//求sa数组  for (int k = 0; k<length; k++)  {  index[(line - 1) \* length + k] = sa[k];//求index数组  }  }  middle = clock();  printf("匹配序列:\n");  scanf("%s", p);//输入目标序列  middle2 = clock();  for (int line = 1; line <= 1000000; line++)  {  for (i = 0; i < 100; i++)  {  s[i] = DNA[(line - 1) \* 100 + i];//取出DNA序列  }  for (i = 0; i < length; i++)  {  sa[i] = index[(line - 1) \* length + i];//取出索引  }  int dot = binSearch(s, p, sa, k, length);//二分检索  if (dot != -1)  {  int lt = dot, rt = dot;  while (strncmp(p, &s[sa[lt - 1]], k) == 0)//向左扩展  {  lt--;  }  while (strncmp(p, &s[sa[rt + 1]], k) == 0)//向右扩展  {  rt++;  }  while (lt<=rt)  {  fprintf(fResult, "第%d行: ", line);//输出到文件  fprintf(fResult, "from %d", sa[lt] + 1);  fprintf(fResult, "\n");  lt++;  }  }  }  finish = clock();  duration = (double)(middle - start);//建立索引时间  duration1 = (double)(finish - middle2);//检索时间  printf("建立索引运行时间为%fms\n", duration);  printf("查询运行时间为%fms\n", duration1);  return 0;  } |

|  |
| --- |
| 附录2 |
| 改进索引建立方法 |
| void da1(int \*r, int \*sa, int n, int m, int length)  {  int i, j, p, \*x = wa, \*y = wb, \*t;  for (i = 0; i<m; i++) ws[i] = 0;  for (i = 0; i<n; i++) ws[x[i] = r[i]]++;  for (i = 1; i<m; i++) ws[i] += ws[i - 1];  for (i = n-1; i >= 0; i--) sa[--ws[x[i]]] = i;  for (j = 1; j<n-length; j \*= 2)  {  int p = 0;  //直接利用sa数组排序第二关键字  for (i = n - j; i<n; i++) y[p++] = i;  for (i = 0; i<n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;  //基数排序第一关键字  for (i = 0; i<n; i++) wv[i] = x[y[i]];  for (i = 0; i<m; i++) ws[i] = 0;  for (i = 0; i<n; i++) ws[wv[i]]++;  for (i = 1; i<m; i++) ws[i] += ws[i - 1];  for (i = n - 1; i >= 0; i--) sa[--ws[wv[i]]] = y[i];  //根据sa和y数组计算新的x数组  for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i<n; i++)  x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;    if (p>n) break;  m = p;  }  return;  } |

|  |
| --- |
| 附录3 |
| 模型检验kmp算法核心程序 |
| //kmp函数  int KMPStrMatching(string T, string P, int \*N, int line, FILE \*fResult) {    int i = 0; // 模式的下标变量  int j = 0; // 目标的下标变量  int pLen = P.length(); // 模式的长度  int tLen = T.length(); // 目标的长度  if (tLen < pLen) // 如果目标比模式短，匹配无法成功  return -1;  while (j < tLen) { // 反复比较对应字符来开始匹配  if (i < pLen)  {  if (i == -1 || T[j] == P[i])  i++, j++;  else i = N[i];  }  else  {  fprintf(fResult, "第%d行 ", line);  fprintf(fResult, "from %d", (j - pLen + 1));  fprintf(fResult, "\n");  i = 0;  }  }  return -1;  }  //求next数组  int \*findNext(string P) {  int i = 0;  int k = -1;  int m = P.length(); // m为字符串P的长度  assert(m > 0); // 若m＝0，退出  int \*next = new int[m]; // 动态存储区开辟整数数组  assert(next != 0); // 若开辟存储区域失败，退出  next[0] = -1;  while (i < m) { // 计算i=1..m-1的next值  while (k >= 0 && P[i] != P[k]) // 求最大首尾子串  k = next[k];  i++;  k++;  if (i == m) break;  if (P[i] == P[k])  next[i] = next[k]; // P[i]和P[k]相等，优化  else next[i] = k; // 不需要优化，就是位置i的首尾子串长度  }  return next;  }  string t;//DNA  string p;//目标k-mer  int \*N;//next数组  cout << "检验k-mer:" << endl;  cin >> p;  FILE \*fSource;  fSource = fopen("PreIndex", "r");  FILE \*fResult;  fResult = fopen("搜索结果.txt", "w");  N = findNext(p);  for (int line = 1; line <= 100; line++)  {  char s[100];  fscanf(fSource, "%s", s);  t = s;  if (KMPStrMatching(t, p, N, line, fResult) == -1) fprintf(fResult, "\n");  }  delete[] N; |