DNA 序列中基于适应性后缀树的重复体识别算法

霍红卫 王小武
(西安电子科技大学计算机学院 西安 710071)

摘 要 现有的在 DNA 序列中识别重复体的算法多数是基于比对的,对识别速度和吞吐量有很大的限制.针对这个问题文中根据一个平衡重复体的长度和频率的定义,提出了一种基于 Ukkonen 后缀树的快速识别重复体的 RepSeeker 算法.算法采用最低限制频率,最大程度地扩展了重复体的长度,同时为了进一步地提高 RepSeeker 算法的效率,对 Ukkonen 的后缀树构造算法进行了适应性改进,在构造时加入 RepSeeker 算法所需的结点信息并将叶子结点和分支结点加以区分,从而使得 RepSeeker 算法能通过直接读取结点信息来求得子串频率和子串位置.这种改进较大地提高了 RepSeeker 算法的性能,而且空间开销不大.实验中使用了 NCBI 中的 9 条典型 DNA 序列作为测试数据,并对后缀树改进前后的重复体识别算法做了比较分析.结果表明,RepSeeker 在没有损失精度的情况下缩短了算法的运行时间.实验结果与理论上的分析一致.

关键词 重复体识别;适应性后缀树;Ukkonen 算法;RepSeeker 算法
中图法分类号 TP18 **DOI 号**: 10.3724/SP.J.1016.2010.00747

An Adaptive Suffix Tree Based Algorithm for Repeats Identification in a DNA Sequence

HUO Hong-Wei WANG Xiao-Wu
(*School of Computer Science and Technology, Xidian University, Xi'an* 710071)

Abstract Many existing methods for repeats identification are based on alignments. Their speed and time significantly limit their applications. This paper presents the fast Rep(eats)Seeker algorithm for repeats identification based on the adaptive Ukkonen suffix tree construction algorithm. The RepSeeker algorithm uses the lowest frequency limit to maximize the extension of repeats. The adaptive improvements to the Ukkonen algorithm are made to increase the efficiency of the RepSeeker algorithm. The node information required by the RepSeeker algorithm is added during the suffix tree construction. Because information on leaves and branch nodes are different, the RepSeeker algorithm directly obtains the needed information from the nodes to find out the frequency and locate the positions of a substring. The improvement is considerable for the repeats identification at a little extra cost in space. Nine sequences from the National Center for Biotechnology Information (NCBI) are used to test the performance of the RepSeeker algorithm. Comparisons between before and after improvements of the suffix tree construction show that the running time of the RepSeeker algorithm is reduced without losing the accuracy. The experimental results agree with the theoretical expectations.

Keywords repeats identification; adaptive suffix tree; Ukkonen algorithm; RepSeeker algorithm

收稿日期:2008-11-19;最终修改稿收到日期:2010-03-16.本课题得到国家自然科学基金(69601003)和青年科学基金(60705004)资助.
霍红卫,女,博士,教授,主要研究领域包括大规模应用问题的算法及软件、并行算法和生物信息学算法.E-mail: hwhuo@mail.xidian.edu.cn.
王小武,男,硕士,工程师,主要研究方向包括大规模应用问题的算法及软件和生物信息学算法.

1 引 言

基因组中含有许多重复元素.例如,在人类基因组的约 3.2×10^9 个碱基对中超过 50%已被识别为各种重复元素[1-2].重复体识别对于分析新的基因组非常重要[3],这是因为:(1)重复体以各种方式引导着基因组的进化过程;(2)在进行同源查找之前需要对重复体进行掩模.而在实际中有各种各样的重复体,大多数重复体的功能并未完全被理解和定义[4].当前研究表明某些重复体在基因表达和转录调控方面起着重要作用[5].与重复体结合的碱基可能导致基因重组,使基因组发生重大变化.重复体种类繁多,它们包含着几个到数百的碱基对,有些可达上万碱基对.一些人类的遗传疾病诸如脆性 X 染色体综合症、亨延顿氏症以及弗里德共济失调都与重复体长度的不规则性有关[6].

一个重要的生物信息学问题是如何快速识别并有效地表示基因组中的重复体.目前,解决重复体识别的方法大致有两类:RepeatMasker[7]根据一个已经注释的重复体数据库对已知重复体进行查找.这个数据库很大程度上依赖于同源序列的相似性.这种方法不能用于处理新的基因组序列,是因为它不能为新测序的基因组构建所需的库信息.而且,对于新的基因组,它的重复体库需要手工编撰,因为这种方法是面向特定基因组的.对 RepeatMasker 数据库的从头识别仍然是生物信息学中的一个挑战问题[8]. REPuter[9]是另一种方法,它摘取具有最大长度的所有重复体的相似对,并把重复体定义为一组具有最大长度的相似字符串对.这两种方法都没有考虑重复体的出现次数.一般而言,在真实生物序列中,重复体会出现多次.例如,在人类基因组中 *Alu* 出现 10^6 次.在复杂的基因组中,转座子一般出现几十万次.因此,在识别方法中结合重复体的频率更合理.

具有生物意义的重复体的定义必须考虑重复体的长度和频率.一些研究表明准确地定义重复体是不容易的.一些方法只能找到短的重复序列或串联重复序列.它们难以找出长且散布的重复序列.最近的一些方法集中在识别重复体的边界上.Price 等人提出了 RepeatScout 算法[10],该方法使用高频 l-mer 种子来查找重复体的边界,且用贪心法扩展每个种子,使之成为更长的同源序列.Edgar 和 Myers 研制了 RILER 软件包[11],通过刻画重复体特征和局部比对来识别具有可靠边界的重复体.

意识到重复体出现频率的重要性,已有几种方法把长度和频率结合在重复体的定义中.Zheng 和 Lonardi[12]给出了一种基于后缀树来查找 DNA 序列中重复体的方法,且时间复杂度为 $O(n^2 f)$[13].Zheng 和 Lonardi 算法的效率不高,是因为对于数十万碱基对长度的 DNA 序列,算法仍然难以有效工作.

尽管在试图定义和识别一个序列中重复体已有大量成果,重复体查找仍然是一个挑战性的问题.文献[8]使用了局部序列比对策略和 A-Bruijn 图来解重复体查找问题.然而,基于 A-Bruijn 图的方法的分析非常复杂和困难.此外,这种方法需要对输入序列进行双序列局部比对.其性能主要依赖于局部比对的结果.Gu 等人[14]提出了一种基于精确字统计法来估计大型真核基因组中重复结构出现频率的方法,该方法对比期望出现次数多的寡核苷酸进行分类.

本文中按照文献[12]中对重复体的定义,对 Ukkonen 后缀树构造算法做了适应性的改进,提出一种快速识别重复体的算法 RepSeeker. RepSeeker 算法使用最低频率限制,并扩展重复体的长度使之达到最大.在后缀树的构造过程中,对叶子节点进行编号,并把叶子节点的信息加入到分支节点中,叶子节点和分支节点所包含的信息不同,以使 RepSeeker 算法能够直接从节点中获得子串的频率和位置信息.这种改进大大提高了 RepSeeker 算法的性能,而且空间开销不大. RepSeeker 算法使用来自 NCBI 中的 9 条序列进行了性能测试.并对改进前后算法的性能作了比较.实验结果表明,对其数据结构所做的改进大大降低了 RepSeeker 算法的运行时间,同时又保证了识别的精度.实验结果与理论上的分析一致.

2 RepSeeker 算法

2.1 表示、约定及基本定义

Σ 表示一个有限非空字母表.

字符串和字母用斜体字体.

$|S|$ 表示字符串 S 的长度.

$S[i]$ 表示字符串 S 中的第 i 个字符,$1\leq i\leq|S|$.

$S[i,j]$ 表示 S 的子串 $S[i]S[i+1]\cdots S[j]$,$1\leq$

$i \leq j \leq |S|$.

约定 $S[i,i]=S[i]$.

L-串(或 L-子串)是一个长为 L 的字符串(子串).

A_i 表示 A 在 S 中的第 i 个出现.

A_i 表示 A 的第 i 个拷贝.

有时我们会交替使用这些表示，因为根据上下文就可以明确它们是表示子串还是表示子串所在位置.

如果字符串 A 是字符串 S 的一个子串，且在 S 中出现多次，则称 A 是 S 的一个重复体. 在分子生物学中，重复体就是一个碱基序列在染色体中多次出现的拷贝.

阈值是指一个重复体在一个序列中重复出现次数所指定的最小值. 滑动窗口(见图 1)是某个定长的可视框.

图 1　频率数组 $f(L=4,F_m=3)$

2.2　定　义

定义 1.　设 A 是 S 的一个重复体，$(A_1,A_2,\cdots,A_m)$ 是 A 在 S 中出现的一个有序表，A_i 是 A 在 S 中的第 i 个出现，m 是 A 在 S 中的出现次数，且 $m \geq 2$. 设 B 是 A 的一个子串，$(B_1,B_2,\cdots,B_k)$ 是 B 在 S 中出现的一个有序表，k 是 B 在 S 中的出现次数.

如果 B 在 A 中从位置 s 开始，那么 $B=A[s,s-1+|B|]$，$0 \leq s \leq |A|-|B|$.

如果 $k=m$ 且每个 B_i 在 A_i 中出现的偏移量相同，则称 B 是 A 的一个子重复，$i=1,2,\cdots,m$.

定义 2.　A 是 S 的一个非平凡子串，当且仅当 A 是 S 的一个非空、真 L-子串，$0<|A|<|S|$，L 是重复体的最小长度.

定义 3.　如果 A 是 S 的一个具有最大长度的非平凡子串，且 A 在 S 中至少出现 F_m 次，A 的每个非平凡子串是 A 的一个子重复，其中 F_m 是指定的重复体最小出现频率，则称 A 是 S 的一个基本重复体.

性质 1.　如果 A 是 S 的一个基本重复体，且出现频率为 f，那么 A 的每个 L-子串出现频率均为 f.

证明.　略. 结论可由定义直接而得.

上述性质使我们在计算重复体的出现频率时，可以删除大多数非候选的重复体. 在计算中，使用后缀树作为基本数据结构，来统计具有给定阈值的其 L-子串的频率.

2.3　算法描述

RepSeeker 算法首先找出输入序列 S 中的所有基本重复体，然后输出重复体的一个有序表. 重复体表中的元素是一个数对，表示一个重复体在输入序列中的起始位置和结束位置. 因此，识别基本重复体的问题可以转换为寻找重复体的边界问题. 因而，RepSeeker 算法检查输入序列中每个位置，并确定一个位置是否是重复体的一个边界.

使用穷尽算法查找基本重复体是不切实际的，因为在 S 中有 $O(n^2)$ 个子串，对于每个长为 m 的子串，需要检查 $O(m^2)$ 个子串. 因而，我们构造输入序列 S 的一棵后缀树，帮助进行频率统计. 按照性质 1，可得：如果 A 是 S 的一个基本重复体，那么 A 的所有 L-子串出现频率相同，且至少为 F_m. 于是，RepSeeker 算法计算出所有 L-子串的出现频率，并根据频率数组把频率相等且至少为 F_m 的放在一块中. 由于这只是一个必要条件，RepSeeker 算法会检查这些块，并分裂那些包含非子重复的块. 进而，算法对所得重复体进行扩展，最终进行分类.

RepSeeker 算法由以下 5 步组成.

第 1 步. 计算子串在滑动窗口中的频率.

我们把重复体的最小长度 L 作为滑动窗口的宽度，并计算长度为 W 的子串在此窗口中出现的频率. 滑动窗口每次向右移动一个位置. 设频率数组为 f，$f[i]$ 表示在位置 i 开始的 L-子串的出现次数，即 $S[i,i+L-1]$ 的频率. 图 1 中显示了输入序列 S 的频率数组 f 的值. 在图 1 中，$f[0]=3$ 表示在位置 0 开始的长为 4 的子串 $S[0,3]$ 在 S 中出现 3 次，$S[0,3]=$"ABCD"；$f[5]=3$ 表示在位置 5 开始的长为 4 的子串 $S[5,8]$ 在 S 中出现 3 次.

第 2 步. 求出频率相等的块.

根据第 1 步中计算的频率数组，可以计算出出现次数至少为 F_m 的 L-子串的起始位置和结束位置. 在 RepSeeker 算法中，使用 l 和 r 分别记录它的左右边界(即起始位置和结束位置). 如果分别来自 l 数组和 r 数组的两个元素在输入序列 S 中的位置相同且位置为 i，那么它们表示同频率的候选重复块 $(l[i],r[i])$. 基于频率数组 f，我们把序列划分成同频率的块. 特别是，对于任何位置 i，如果 $f[i] \neq f[i-1]$，那么位置 i 是这个同频率块的起始位置；如果 $f[i] \neq f[i+1]$，说明同频率块的最后 L-子串在位置 i 开始，也就是说，$i+L-1$ 是这块的结束位置. 每当得到一个起始位置或结束位置时，就把这个位置插入到 l 数组或 r 数组中. 完成 l 数组或 r 数组的计算之后，接下来是对 l 数组和 r 数组中的元素进行配对. $(l[i],r[i])$ 是我们所找到的第 i 个相同频率块. 例如，对于 $l[0]=0$ 且 $r[0]=9$，图 1 中的块 $S[0,9]=$"ABCDEEBCAD"是一个频率相等块，它的长为 4 的 7 个子串出现频率一样. 对于图 1 中的示例，去掉频率小于 $F_m(=3)$

的子串后，最终频率相等块为 $S[0,9]$，$S[12,21]$，$S[24,28]$，$S[29,33]$和 $S[36,41]$.

第 3 步. 子重复检查.

对从第 2 步得到的块进行检查，设某一块 $D_i=S[i_1,i_2]$，块长度为 $length$，出现频率为 k，其块内所有长度为 L 的子串的频率为 m，若 $k<m$，则块 D_i 内含有不是子重复的子串. 从左到右依次求出块 D_i 中所有长度为 L 的子串在 S 中的出现位置，然后对开始位置相邻的两个子串的出现位置序列进行比较，若存在对应次序上的位置不相邻，则分裂块 D_i. 例如，图 1 中，子串 $S[0,9]$的发生频率为 2，但在第 2 步中被识别为重复体，通过子重复检查，发现 $S[1,4]$=“BCDE”的出现位置为{1,13,30}，$S[2,5]$=“CDEE”的出现位置为{2,14,36}，两子串的第 3 次出现位置不相邻，因此将 2 插入 l 数组，将 5 插入 r 数组. 对数组 l 和 r 的插入要求进行有序插入，即插入后数组仍保持有序. 所以 $S[0,9]$可分裂成 $S[0,4]$，$S[2,7]$，$S[5,9]$ 3 个重复体.

第 4 步. 重复体扩展.

为了尽可能得到更长的重复体，我们归并有重叠的重复体，如果归并后的重复体频率至少为 F_m. 具有较高频率的重复体仍然被保留下来，较低频率的重复体被扩展. 令重复体 A 和 B 含有重叠块，重叠块称为 C. 归并满足的条件如下：

$$merop(A,B)=\frac{C}{A\cup B}\times 100\%\geq OP$$

且

$$frequency(A\cup B)\geq F_m,$$

其中 $merop$ 为重叠率，OP 为指定最小重叠率，$A\cup B$ 为归并 A 与 B 之后的结果，$frequency$ 为归并后子串的出现频率.

例如，在图 2 中，假设重复体 A 和 B 分别在输入序列中出现 3 次和 2 次. 由图 2 可得 A_1 和 B_1 有公共重叠子块. 如果 A 和 B 的重叠率至少为 OP，且归并之后的结果块 M_1 至少出现 F_m 次，那么我们进行归并，也即实施了扩展. 同样，归并 A_3 和 B_2 得到 M_2. 最终，B_1 和 B_2 分别被扩展至 M_1 和 M_2.

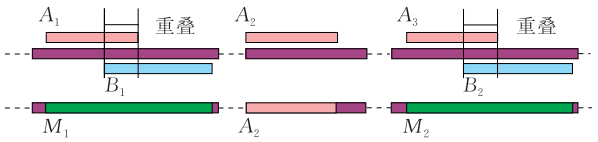

图 2 重复体扩展，$F_m=2$

第 5 步. 重复体归类.

将相同的重复体归为一类.

2.4 RepSeeker 算法

RepSeeker 算法如下.

RepSeeker(S,L,F_m)

Input: string S with length n, minimum length L of repeat, minimum frequency F_m

Output: classification list of all repeats in S that appears at least F_m times

```
0. create a suffix tree for the string S
1. for i←0 to n−L do
2.    f[i]←frequency of the ith L-substring in S
3. create D: an array of repeat blocks in S
4. if f[0]≥F_m then add 0 in l array
5. for i←1 to n−L−1 do
6.    if f[i]≥F_m then
7.       if f[i]≠f[i−1] then add i in l array as a
           starting position
8.       if f[i]≠f[i+1] then add i+L−1 in r array
           as an ending position
9. for i←0 to |l|−1 do
10.   D[i]←(l[i], r[i])
11. D←{D[0],D[1],…,D[k−1]}, sum←|D|
12. Check(D)
13. Extend(D)
14. Classify(D)
```

Check(D: an array of blocks of equal frequency)

```
1. for i←0 to sum−1 do
2.    if f[l[i]]≠f(D[i]) then
3.       for j←0 to |D[i]|−L do
4.          P[j]←sorted list of positions of occurrences of
                 j-th L-substring in D[i]
5.       for k←0 to |D[i]|−L−1 do
6.          for m←0 to f[l[i]]−1 do
7.             if P[k+1,m]≠P[k,m]+1 then
8.                insert l[i]+k+1 into l and keep its order
9.                insert l[i]+k+L−1 into r and keep its order
10.               sum←sum+1
```

Extend(D: an array of elementary repeats)

```
1. for i←1 to sum−1 do
2.    if merop(i,i+1)≥OP and frequency of merged
        sequence≥ F_m
3.       then
4.          D[i]←(l[i], r[i+1])
5.          i←i−1, sum←sum−1
```

Classify(D: an array of extended repeats)

```
1. for i←0 to sum−1 do
2.    class[i]←repeats equal to D[i]
```

RepSeeker 算法工作如下. 第 1～2 行计算 S 的所有 L-子串的频率. 第 4～8 行找出频率至少为 F_m 的所有重复块的起始位置和结束位置. 第 9～10 行对 l 数组和 r 数组中元素配对. 子例程 Check 检查块 D_i 是否包含非子重复. 子例程 Extend 归并满足条件的重复体，以达到扩展的目的. 子例程 Classify 对重复体归类输出.

2.5　后缀树在 RepSeeker 算法中的作用

后缀树数据结构对于 RepSeeker 重复体识别算法的有效实现起着至关重要的作用. 在图 3 中，我们从 T 出发到叶子结点，可得到该序列的一个后缀. 设序列 P 是 k 个后缀的公共前缀，而对于整个序列来说，P 出现了 k 次. 这样我们可以通过遍历同一“主干”下的叶子数目，来确定该“主干”在序列中的出现次数，也即频率. RepSeeker 算法的主过程第 2～3 行用后缀树求得频率数组 f.

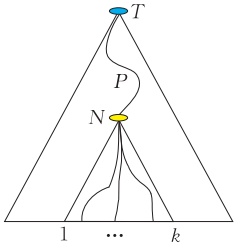

图 3　后缀树 T

存在的问题：

(1) 为了得到子串 P 的频率，我们必须遍历 N 下所有叶子结点，来计算叶子数目. 对于 DNA 序列来说，通过遍历来获取所需信息的时间需求是无法忍受的.

(2) Ukkonen 构建的后缀树，结点是按顺序分配的，没有对叶子结点和分支结点的区分. 我们不能通过叶子的序号来得到重复体的出现位置. 在第 3 步的子重复检查中，要通过 KMP[15] 来解决. 在第 5 步的重复体归类中，要通过甚至时间复杂度为 $O(n^2)$ 的算法来完成.

3　对 Ukkonen 算法的适应性改进

3.1　Ukkonen 后缀树构造过程

Ukkonen 后缀树构造算法[16-17] 的基本思想是对于字符串 S 的每个前缀 $S[1..i]$ 构造一个隐后缀树 T_i，从 T_1 开始，逐步增加 i，直到完成 T_n 的构造，$i=1..n$. 若字符串 S 的长度为 n，则构造算法分成 n 步，对于每一步 $i+1$，又分成 $i+1$ 个扩展，每个扩展代表了 $S[1..i+1]$ 的 $i+1$ 个后缀中的一个后缀，在第 $i+1$ 步的第 j 个扩展中，算法首先找到从根节点开始标记子串 $S[j,\cdots,i]$ 的路径的结束位置，然后将字符 $S(i+1)$ 加入该子串尾对它进行扩展，除非 $S(i+1)$ 已经存在. 因此，在第 $i+1$ 步，字符串 $S[1,\cdots,i+1]$ 首先被插入树中，然后插入 $S[2,\cdots,i+1]$，$S[3,\cdots,i+1]$，以此类推. 在 $i+1$ 步的第 $i+1$ 个扩展中对 $S[1..i]$ 的空后缀进行扩展，确保向树中插入了单个字符 $S(i+1)$（除非 $S(i+1)$ 已经存在）. 树 T_1 是标记为 $S(1)$ 的单条边. 在后缀树算法构造过程中，引入了扩展后缀树的 3 个规则以及后缀链使算法的时间复杂度为 $O(n)$. 算法的详细描述见文献[16-17].

3.2　Ukkonen 算法特征分析

这个算法有几个显著的特点：一旦一个结点被作为叶子结点创建，那么它始终都是叶子结点，将不会有子孙结点. 更重要的是，在每次将一个前缀插入到树中，就机械地用一个相同的字符去扩展每一个指向叶子结点的边，而这个字符必然是新后缀的最后一个字符.

3.3　构建后缀树时的改进

后缀树对 RepSeeker 算法的主要贡献在于，它能快速地求出定长子串 P 在 S 中的出现频率，也就是求出有相同前缀 P 的后缀个数.

如图 3，利用后缀树计算子串频率传统的方法是：(1) 从树根找起，先找到与 P 相匹配的树的主干；(2) 遍历以结点 N 为根的所有叶子结点，所得的叶子结点数就是要求的 P 的频率. 第 1 步可以在线性时间内找到 N，第 2 步则要遍历以 N 为根的树，对于 RepSeeker 算法中，用移动窗口求频率数组，总共要遍历 $|S|-L$ 个这样的树.

而查找重复体的副本位置，Ukkonen 构建的后缀树无所适从.

为了高效实现 RepSeeker 算法，我们对 Ukkonen 构建后缀树的算法进行了适应性改进. 改进主要是在构造时加入了结点信息，包括两点：

(1) 在构建后缀树时，对叶子结点和分支结点进行区分，做不同的编号. 结点总个数初始化为 $2\times|S|$，给叶子编号是从 0 开始，每次加 1，给分支结点编号从尾部开始，每次减 1.

(2) 在分支结点处加入结点信息，包括当前结点下的叶子数以及叶子序号. 以 2.3 节中的算法为据，首先给结点数组中每一个成员加入 parent 变量，保存父结点编号. 通过 parent，给新叶子结点的所有父结点加入新的结点信息，即叶子数加 1，并在结点信息数组中加入新叶子编号.

图 4 显示了字符串 agagagcagagt 所对应的改进后的适应性的后缀树. 结点 5 上的信息 3 1 3 8 表明，该结点下有 3 个叶子结点，它的后缀在字符串中出现的位置为 1，3 和 8.

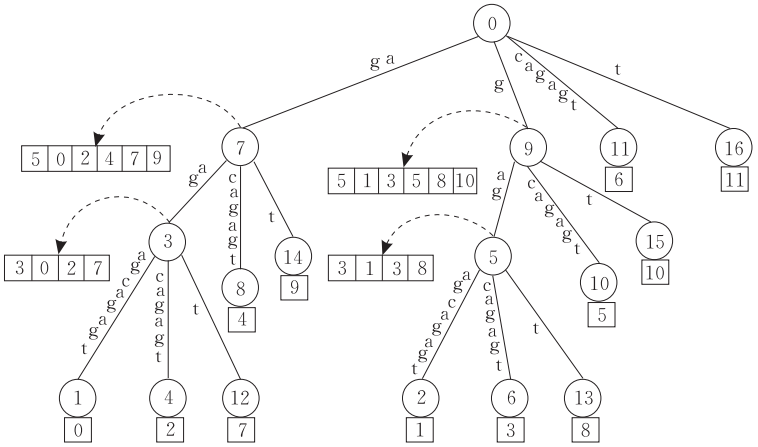

图 4 字符串 agagagcagagt 所对应的适应性的后缀树

3.4 改进对算法的支持

对 Ukkonen 后缀树改进后，带来了算法速度上质的提高. 优点主要表现在两方面：

(1) 用移动窗口求频率数组时，求每一窗口内子串的频率可在 $O(L)$ 时间内完成，L 为窗口宽度，即如图 3 中，求 P 的频率就等于是在找到主干 P 的终止结点 N，找到后读取信息即可. 而匹配 P 的过程是在 HASH 表中进行的，只需进行主干 P 路径上的结点数次比较，便可找到 N. 时间复杂度由 $O(n\log(n))$ 降为 $O(n)$.

(2) 在子过程 Classify 中，无需进行二重循环式的比较，和求频率数组一样，只需找到当前重复体的结束结点，读取结点信息即可，结点中的后缀位置信息就是该重复体的副本分布位置. 时间复杂度由 $O(n^2)$ 降为 $O(n)$.

(3) 更重要的是，加入结点信息所用的时间只是在找新叶子的父结点，parent 变量是结点数组的索引号，找父结点的时间复杂度为 $O(1)\times$父结点个数. 因此改进后的 Ukkonen 算法在构造速度方面几乎没有受到影响.

4 时间和空间复杂度分析

在对后缀树构造算法改进之后，算法主过程 RepSeeker(S,L,F_m) 的第 1～3 步，在后缀树上查找长度为 L 的子串是在 Hash 表上进行的，所以计算频率数组时间复杂度为 $O(n)$. 算法主过程 RepSeeker(S,L,F_m) 的时间复杂度依赖于子过程 Check 的时间复杂度. 子过程 Check 是一个多重循环过程，第 1 层循环遍历所有频率相等且大于限定频率 F_m 的所有块. 第 2 层循环用宽度为 L 的移动窗口对每个块进行分析. 第 3 层循环的循环次数为当前移动窗口内子串的发生位置的频率. 设频率相等块的个数为 N，移动窗口可移动次数为 M，窗口内子串的平均发生频率为 f，那么可以看出检查过程运算次数为 $N\times M\times f$，时间复杂度为 $O(N\times M\times f)$. 分析出算法的时间复杂度为 $O(MN)$.

算法在运行过程中存储边的信息到散列表中，这个表的大小一般不超过 $|S|\times 2.1$. 结点存储在顺序数组中，大小为 $|S|+1$. 由于对 Ukkonen 后缀树的适应性改进，增加了结点信息，从而加大了算法对于空间的要求. 设新增结点信息平均空间需求为 $|X|$，由于只是在分支结点上加入了结点信息，叶子结点所需空间为 $|M|$，所以结点总的空间应为 $(|S|+1)\times|X|+|M|\times(|S|+1)/2$.

5 实 验

5.1 实验参数设置

程序 RepSeeker 在 Microsoft Visual C++6.0 环境上通过 C++ 语言实现. 算法取移动窗口宽度 L 为 20，最低限制频率为 3，合并重叠序列时重叠限制比例为 25%. 本实验测试机器为 Intel 3GHz，1GB 内存的计算机.

5.2 实验结果

实验使用大小不等的 DNA 序列作为测试对象，以识别出的重复体个数、最大重复体及其长度、重复体的归类表作为测试的结果. 并与改进前的指标进行了对比. 表 1 列出了实验结果.

表 1　重复体识别算法在后缀树改进前后的性能

序列名	长度	限制频率 F_m	移动窗口长度	重叠比例/%	改进前运行时间/s	改进后运行时间/s	发现重复体个数	最长重复体长度
X14112	152261	3	20	25	125.473	10.297	45	299
AL593853	223276	3	20	25	231.506	20.172	739	147
AC008583	122493	3	20	25	72.323	6.937	78	41
CU210914	31433	3	20	25	28.921	0.921	8	37
NC_007410	366354	3	20	25	201.937	71.578	26	1547
DOG	1308479	3	20	25	10214.687	2880.469	3999	207
HUMAN3M	2900010	3	20	25	35007.182	15609.390	32759	513

5.3　结果分析

基于表 1 的实验结果，分别从重复体识别结果和运行时间两个方面进行分析. 对于重复体识别结果，从实验结果来看，改进后的后缀树对 RepSeeker 算法提供了强有力的支持，使其在计算速度上得到了很大程度的提高. 对于长度较小的序列时间性能的提高意义不大，如 DNA 序列文件 AC008583，运行时间都是在 100s 以内；而对于大序列，如 DNA 序列文件 HUMAN3M，运行时间减少了 5 个小时左右. 从运行时间上看，RepSeeker 算法在改进了后缀树后优于改进前的算法，这也证明了算法在时间复杂度上有所改进的分析. 下一步算法将在如何减少算法的空间需求上做改进，将在引入新的数据结构上做进一步的探索.

6　结　论

本文根据当前重复体识别算法中存在的问题提出了一种基于 Ukkonen 后缀树算法的准确重复体的快速识别算法 RepSeeker. RepSeeker 算法将对近似重复体和重复结构的识别提供强有力的支持，算法在快速识别重复体的同时定位了每一个重复体序列的左右边界，并把每一个重复体归入相应的类. 算法无论是在计算频率、检查子重复、重复体合并以及最后的重复归类都充分运用了改进后的适应性后缀树构造算法，从而较大地提高了运行速度. 实验结果表明 RepSeeker 算法的运算速度相对改进 Ukkonen 算法前的算法得到了很大的提高，是一种有效的重复体识别算法.

参　考　文　献

[1] Lander E S, Linton L M, Birren B et al. Initial sequencing and analysis of the human genome. Nature, 2001, 409 (6822): 860-921

[2] Huo Hong-Wei, Bai Fan. An algorithm for identification of repeats with accurate boundaries. Chinese Journal of Computers, 2008, 31(2): 214-219(in Chinese)
(霍红卫，白帆. 一种具有精确边界的重复体识别算法. 计算机学报，2008，31(2)：214-219)

[3] Saha Surya, Bridges Susan, Magbanua Zenaida V, Peterson Daniel G. Empirical comparison of ab initio repeat finding programs. Nucleic Acids Research, 2008, 36(7): 2284-2294

[4] Lefebvre A, Lecroq T, Dauchel H, Alexandre J. FORRepeats: Detects repeats on entire chromosomes and between genomes. Bioinformatics, 2003, 19(3): 319-326

[5] Jones Neil C, Pevzner Pavel A. Introduction to Bioinformatics Algorithms. Cambridge, Massachusetts: MIT Press, 2004

[6] Huntington's Disease Collaborative Research Group. A novel gene containing a trinucleotide repeat that is expanded an unstable on Huntington's disease chromosomes. Cell, 1993, 72(6): 971-983

[7] Bergman Casey M, Quesneville Hadi. Discovering and detecting transposable elements in genome sequences. Briefings in Bioinformatics, 2007, 8(6): 382-392

[8] Pevzner P A, Tang H, Tesler G. De novo repeat classification and fragment assembly. Genome Research, 2004, 14 (9): 1786-1796

[9] Kurtz S, Schleiermacher C. REPuter: Fast computation of maximal repeats in complete genomes. Bioinformatics, 1999, 15(5): 426-427

[10] Price A L, Jones N C, Pevzner P A. De novo identification of repeat families in large genomes. Bioinformatics, 2005, 21 (Supplement): i351-i358

[11] Edgar R, Myers E. Piler: Identification and classification of genomic repeats. Bioinformatics, 2005, 21 (Supplement): i152-i158

[12] Zheng Jie, Lonardi S. Discovery of repetitive patterns in DNA with accurate boundaries//Proceedings of the 5th IEEE Symposium on Bioinformatics and Bioengineering (BIBE). Minneapolis, MN, USA, 2005: 105-112

[13] He Dan. Using suffix tree to discover complex repetitive patterns in DNA sequences//Proceedings of the 28th IEEE EMBS Annual International Conference. New York City, USA, 2006: 3474-3477

[14] Gu Wanjun, Castoe Todd A, Hedges Dale J, Batzer Mark A, Pollock David D. Identification of repeat structure in large genomes using repeat probability clouds. Analytical Biochemistry, 2008, 380(1): 77-83

[15] Knuth Donald E, Morris James H, Pratt Vaughan R. Fast pattern matching in strings. SIAM Journal on Computing, 1977, 6(2): 323-350

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

[17] Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. New York: Cambridge University Press, 1997

HUO Hong-Wei, Ph. D., professor. Her research interests include algorithms and software for large-scale applications, bioinformatics algorithms and parallel algorithms.

WANG Xiao-Wu, M. S., engineer. His research interests include algorithms and software for large-scale applications and bioinformatics algorithms.

Background

A genome is an organism's complete set of DNA. Genomes vary widely in size: the smallest known genome for a free-living organism (a bacterium) contains about 600000 DNA base pairs (bps), while mouse and human genomes have about 3 billions base pairs. A genome has a lot of repeats. For example, more than 50% of a human genome is various kinds of repeats. Repeats identification play a critical part in the analysis of a newly sequenced genome, because both repeats drive genome evolution in various ways and because of pragmatic needs for thorough repeat mask before doing homology searches. There are various kinds of repeats. Unfortunately, many functions of the repeats are not yet well understood and defined. Current studies show that some repeats play an important role in the gene expression and transcription regulations. The base composition of regions with repeats may lead to genome recombination which causes great changes of the genome. These repeats fall into different types, which contain from a few to several hundreds of base pairs, some up to ten thousands of base pairs. Some human genetic diseases such as the fragile-X chromosome syndrome, Huntington's disease, and Friedreich's ataxia are all related to irregularities of the length of repeats.

An important bioinformatics problem is how to recognize fast and represent efficiently repeats in a genome. There are two major approaches to solve the repeats identification problems. RepeatMasker uses an annotated library of repeats, which largely depends on similarity to consensus sequences for known repeat classes. This approach is not capable of treating new genomes whose libraries of regions are not available, because RepeatMasker does not build such libraries for new sequenced genomes. Moreover, the repeat libraries have to be manually compiled for any new genome because they are genome-specific. De novo compilation of the RepeatMasker libraries remains a challenging bioinformatics problem. The other approach, such as REPuter, is to extract all pairs of similar repeated regions with maximal length. Recently, more approaches are concerned on the detection of repeat boundaries. Price et al. presented the RepeatScout algorithm to find repeat boundaries by using high-frequency l-mer as seeds and greedily extends each seed to a progressively longer consensus sequence. Edgar and Myers developed the RILER package to identify repeats with reliable boundaries by considering characteristics repeats and tandem arrays of local alignments.

Some work uses the local sequence alignment strategy and the A-Bruijn graph to solve the problem. Gu et al. proposed an approach based on exact word counts to evaluate, de novo, the presents of a repeat structure within large eukaryotic genomes. In this method, clusters of related oligonucleotides that occur more often than expected by chance are grouped. Zheng and Lonardi give a suffix tree construction algorithm based algorithm for finding the repeats in DNA sequences with time complexity $O(n^2 f)$, where n is the length of the sequence and f is the minimum frequency requirement.

This paper presents the RepSeeker algorithm for repeats identification based on the adaptive Ukkonen algorithm for a suffix tree construction. Experimental results show that the improvements reduce the running time of the RepSeeker algorithm without losing the accuracy. The experimental results coincide with the theoretical expectations.