分类号 TP301 密级 公开
UDC 编号

# 雲南大學

# 硕士研究生学位论文

题 目 LM-Suffix:基于后缀树的基因序列索引结构研究

学院（所、中心） 软件学院

专 业 名 称 系统分析与集成

研究生姓名 皇甫永伟 学号 12011001327

导 师 姓 名 姚绍文 职称 教 授

2014 年 5 月

# 独创性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人或集体已经发表或撰写过的研究成果，对本文的研究做出贡献的集体和个人均已在论文中作了明确的说明并表示了谢意。

研究生签名：皇甫永伟　日　期：2014年6月

# 论文使用和授权说明

本人完全了解云南大学有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交学位论文和论文电子版；允许论文被查阅或借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

**（保密的论文在解密后应遵循此规定）**

研究生签名：皇甫永伟　导师签名：[illegible]　日　期：2014年6月

.....

本人及导师同意将学位论文提交至清华大学“中国学术期刊（光盘版）电子杂志社”进行电子和网络出版，并编入 CNKI 系列数据库，传播本学位论文的全部或部分内容，同意按《中国优秀博硕士学位论文全文数据库出版章程》规定享受相关权益。

研究生签名：皇甫永伟　导师签名：[illegible]　日　期：2014年6月

# 摘要

随着生物信息学研究的不断深入，大量的生物信息已经产生并仍在源源不断的产生。如果能加快对这些信息的处理，就能更好的揭示生命的奥秘。正是基于这样的原因，研究人员利用索引技术来改进序列的处理。对于较小的序列来说，后缀树无疑是个不错的选择，但是存在“内存瓶颈”的问题导致不适合用于长基因序列；后缀数组是另一个不错的选择，它占用的存储空间比后缀树占用的存储空间更少，但是后缀数组的性能要远远低于后缀树；基于 q-gram 和 q-sample 的索引结构虽然可用于快速匹配，但是对于相似度低的基因序列却无能为力。

本文通过对经典的后缀树构造算法进行深入的研究分析，在此基础上提出了一种基于后缀树索引结构的 LM-Suffix 算法。LM-Suffix 算法将长基因序列划分为若干个短基因序列为解决传统的后缀树构造算法不支持长基因序列的问题奠定了基础。通过对短基因序列按照字母表顺序排序并根据两两相邻的短基因序列的最大公共前缀来确定分支点从而实现后缀树的构造。由于 LM-Suffix 算法在构造后缀树的过程中对后缀进行了分割从而产生了多个子后缀树，因此传统的后缀树搜索技术并不适合于该算法。本文在传统的后缀树搜索方案的基础上设计了一种适合于 LM-Suffix 算法的搜索方案。该方案的主要改进在于使用哈希表来记录子后缀树的访问次数，通过遍历这个哈希表将经常被访问的子后缀树优先调入内存进行搜索。

最后使用本文提出的 LM-Suffix 算法来为长基因序列构造后缀树，实验结果表明了 LM-Suffix 算法解决了传统的后缀树构造算法不支持长基因序列的问题。

**关键词：**基因序列；后缀树；索引结构；LM-Suffix；

# Abstract

With the deepening of the bioinformatics, a large number of biological information has been produced and is still being produced in a steady stream. If we can quicken to deal with this information, we can better reveal the mysteries of life.For these reasons, the researchers make use of index to improve the processing of sequences.For smaller sequences, Suffix treeis a good index structure, but there exists the so-called "memory bottleneck", so it not suit large sequences. The suffix array is another good structure,as it needs less store space than a suffix tree, but its performance is far lower than a suffix tree; Approaches based on q-gram or q-sample can be used for fast matching, but for the biologysequences of low similarity, it can do nothing.

In this paper, based on the deep study of the classical suffix tree, we present LM-Suffix based on a suffix tree index structure. LM-Suffix divides the long biologysequences into some short biologysequences, which solve the "memory bottleneck" that the classical suffix tree exists, and then in order to construct a suffix tree, we sort the short biologysequences by alphabetical order and according to the biggest publicprefix of the two adjacentbiologysequences confirm the branch point. During the constructing a suffix tree, because LM-Suffix is parting a suffix tree, so the classical suffix treesearching technology is not suitable for the algorithm.In this paper, based on the classical suffix treesearching technology, we present a search scheme which is suitable for LM-Suffix, which increase child suffix tree traversal.

Finally, we show experimentally that the LM-Suffix can be built effectively with sequences whose index size exceeds the available RAM, and its performance is better, which solve the so-called "memory bottleneck". In addition, We also implement the searching technology using the LM-Suffix.

**Keywords:** biology sequences;suffix tree;index structure; LM-Suffix

# 目录

# 插图列表



# 插表列表

# 第1章 绪论

## 1.1 研究背景及意义

### 1.1.1 研究背景

生物信息学[1]不仅是当今生命科学的重要前沿领域之一，而且也是 21 世纪自然科学的核心领域之一。随着生物信息学研究的不断深入以及人类基因组蓝图的完成，生物信息量陡增特别是基因序列的数量成几何级增长，如何在大量的基因序列中快速、高效地检索出所需的信息显得尤为重要。

基因序列[2]数据库最为典型的特征就是具有极其庞大的数据规模。例如，美国国家生物技术信息中心建立的 GenBank[3]数据库的数据规模正以指数形式递增，核酸碱基的数目每 14 个月就翻一倍。GenBank 数据库拥有 70,000 个物种的约 1000 亿个碱基[3]。在过去的几年中，用途不尽相同的各种类型的基因序列数据库在世界各地陆续建立起来，例如：DDBJ、dbSTS、FLYBASE、HAEMA、HAEMB 等。这些数据库的访问量在飞速增长，仅对 GenBank 数据库的查询请求每天就达百万次之多。

目前基因序列数据库存储数据的方式主要有两种[4,5]，一类是用数据库管理系统中的文本数据类型，另一类是直接使用外部文本文件存放基因序列数据，在数据库中只存放指向该文件的指针。不管采用哪种存储方式都不能有效地对基因序列进行操作，那么如何才能高效的检索出所需的信息呢？除了对计算机的性能进行改进之外，还需要从程序的角度来考虑。数据检索领域的众多研究结果表明，索引技术可以加快数据检索的速度。该技术已经成功地应用在众多数据处理领域中。例如，B-trees 索引结构已成为了数据库系统的标准[5]。因此，我们完全可以将索引技术引入到生物信息领域中。

对基因序列数据库中的海量信息建立索引面对的主要挑战是为如此庞大的基因序列建立索引需要的时间开销过于高昂。在对基因序列进行检索的过程中每次检索都要建立索引结构，这显然是事倍功半。根据基因序列更新周期缓慢这一显著特点，一次建立多次检索[6,7]的思想应运而生。该方案的核心是首先为待搜索的基因序列建立索引，然后将已建立的索引存储到磁盘中，如此以来在下次检索时直接从磁盘读入即可。

### 1.1.2 研究意义

在海量的基因序列数据中快速找到所需信息是一项十分重要的研究工作，对基因序列进行快速、高效的查询已经成为当前的研究热点[8]。为了加速基因序列数据库中的序列查询需要一种高效的方法查询基因序列数据库。索引结构用一定的存储空间作为代价来换取查询时的快速响应，良好的索引结构能够有效的组织基因序列数据，显著提高查询速度，相反不使用索引的搜索算法需要遍历整个数据库，其效率并不理想。相对于非索引情况下逐条序列比对的搜索方法，使用索引能快速的进行比对，从而减少搜索时间。

由于基因序列具有缺少结构信息这样的特性，因此传统的索引技术并不适用。目前广泛使用的索引结构有后缀树[9,10]、后缀数组[11]和 q-gram[12]索引。基于 q-gram 和 q-sample 的索引结构具有最快的运行速度，但是其最大的缺陷在于不能用于检索相似度低的基因序列。后缀数组索引结构可以认为是后缀树中的所有叶子结点按照一定的顺序排列后放入一个整型数组中。由于后缀数组仅仅存储了索引号，这就大大的限制了其用途，而且如何提高后缀数组的搜索速度又是一个难题。后缀树是一种比较合理的索引结构，该结构最大的优势在于具有很好的搜索效率，但是在处理长基因序列时会产生“内存瓶颈”问题，因此只适合于短基因序列。

## 1.3 论文的主要工作及创新点

本文对经典的线性时间内构造后缀树的 Ukkonen 算法[10]进行了深入的研究分析，该算法虽然可以在线性时间内构造后缀树，但是由于引入了后缀链[10]这一概念造成了所谓的“内存瓶颈”问题，正是由于该问题导致了不能为长基因序列建立索引。本文在此基础上提出一种新的后缀树构造方案——LM-Suffix 算法。该方案在保持后缀树索引结构优点的同时尽量消除各种不良因素。一方面，该方案要考虑具有尽可能小的时间复杂度。另一方面，该方案要尽量消除“内存瓶颈”问题，从而使得后缀树索引结构可以处理超长的基因序列。传统的后缀树搜索技术并不适用于 LM-Suffix 算法，这是因为传统的搜索技术都是针对单一后缀树而言的，并不适合于多个子树的情况。本文通过对传统的搜索技术进行改进，使之可以搜索 LM-Suffix 算法构造的后缀树。论文的主要创新点有：

（1）LM-Suffix 算法将一个长基因序列划分为若干个短基因序列，然后再对

这若干个短基因序列进行后缀树构造。通过这样的划分为解决传统的后缀树构造算法不支持长基因序列的问题提供了可能。

（2）为了解决频繁的调入调出操作所带来的效率低下问题和 I/O 过多问题，LM-Suffix 算法根据内存的可用情况将划分后的某些短基因序列进行合并，然后一起调入内存进行处理，从而解决 I/O 频繁引起的效率低下问题。

（3）由于 LM-Suffix 算法放弃使用后缀链，因此 LM-Suffix 算法在构造后缀树之前需要对待处理的后缀按照字母表顺序进行排序，以此来尽量消除放弃后缀链带来的效率骤降问题。

（4）为了解决传统的搜索技术不适合 LM-Suffix 算法的问题，在原有的搜索方案的基础上使用哈希表来记录子后缀树的访问次数，然后通过遍历这个哈希表将经常被访问的子后缀树优先调入内存进行搜索。

最后通过实验来验证本文提出的 LM-Suffix 算法与现存的后缀树构造算法在为不同长度的基因序列建立索引时的性能情况，同时对其搜索性能进行测试。

## 1.4 论文的组织结构

全文共分 5 章：

第 1 章绪论。简单介绍了基因序列索引结构的研究背景及意义，并给出了论文的主要工作及创新点，最后给出了论文的组织结构。

第 2 章相关工作。对基因序列及其搜索技术进行了简单的介绍，并对主流的基因序列索引结构进行了阐述，最后对后缀树索引结构进行了详细的介绍。

第 3 章基于 LM-Suffix 算法的后缀树索引结构。本章提出了 LM-Suffix 算法，该算法可以解决现存的后缀树构造算法普遍存在的不支持长基因序列的问题，进而设计了一种适合于 LM-Suffix 算法的搜索方案。

第 4 章实验验证及分析。通过实验来验证本文提出的 LM-Suffix 算法是否达到预期效果。

第 5 章总结与展望。全文工作的总结及对未来工作的展望。

# 第2章 相关工作

## 2.1 基因序列及其搜索技术

本节首先对基因序列的相关情况进行了简单的介绍，在此基础上介绍了基因序列的搜索技术。本节内容为下一步的研究工作奠定了基础。

### 2.1.1 基因序列相关介绍

从信息学的角度出发，生物分子是生物信息的载体，基因序列对蛋白质氨基酸序列进行编码，蛋白质序列决定蛋白质结构，而蛋白质结构又决定了蛋白质的功能[3,4]。归根结底，DNA 包含了最基本的生物信息。

DNA 序列可以看成是由 A、C、G、T 四个固定的字符所组成的字符串序列，其中 A、C、G、T 为四个碱基[3,4]。这四个碱基分别对应于四个不同的核苷酸：A——腺嘌呤、C——胞嘧啶、G——鸟嘌呤、T——胸腺嘧啶。由于 DNA 序列呈双螺旋结构，而这种双链结构又是依靠嘌呤和嘧啶之间的氢键连在一起，因此碱基 A 和碱基 T 配对，碱基 C 和碱基 G 配对，按照这样的规则配对形成的一对碱基称之为一个碱基对（bp）。碱基对是用来衡量基因序列长度的单位而不是衡量基因序列的大小的单位[3,4]。例如，人类基因组的长度大概为 3Gbp。如果不采用编码的方式，而是直接采用碱基的字母来进行存储，那么一个碱基占用一个字节（一个字符的长度）。显然，对于长度为 3Gbp 的 DNA 序列而言，其占用的存储空间为 6GB。碱基之间的配对特性是 DNA 复制的基础。在复制的过程中，首先将双螺旋结构断开，然后把断开的这两条链当作模板，同时按照碱基配对的规则来产生新链，至此复制结束[3,4]。通过这样的方式产生的两条新链就像双生子一样。DNA 序列的复制揭示了遗传信息的传递规则，互补的双链结构则确保了信息复制的正确性。除了 DNA 这种双链结构之外，细胞分子中还存在一种单链结构，这种单链结构称之为 RNA。该结构使用尿嘧啶 U 代替了 DNA 中的胸腺嘧啶 T，也就是说在 RNA 中 A 和 U 配对，C 和 G 配对。

基因序列不仅包含了合成蛋白质所必需的信息，而且还包含了所有细胞结构以及生命活动的基本指令。由紧密缠绕的脱氧核糖核酸以及蛋白质共同组成的染色体上的全套基因就称为基因组。基因组是信息传递的结构和功能单位，而基因

则是信息表达的结构和功能单位。

### 2.1.2 基因序列搜索技术

基因序列的搜索技术主要是指字符串匹配技术[13]。简而言之，就是在长度为 n 的文本 S 中查找长度为 m 的模式 P。研究者感兴趣的检索主要有三种：是否存在、计数、枚举。是否存在是判断模式 P 是否出现在文本 S 中。计数是计算模式 P 在文本 S 中出现的次数。枚举是枚举出与模式 P 匹配的所有子字符串在文本 S 中的位置。

目前研究者已经提出了许多字符串匹配算法。按照是否需要对文本 S 进行预处理将字符串匹配分为 On-line 模式和 Off-line 模式[13]。On-line 模式只需要对模式 P 进行预处理操作，而对文本 S 则不需要进行预处理操作。Off-line 模式不仅可以对文本 S 进行预处理操作，而且还可以通过建立索引结构来提高匹配速度。当前经典的字符串匹配算法是由 Knuth, Morris 和 Pratt 提出的KMP 算法[14,15]。R.S.Boyer 和 J.S.Moore 在文献[14]中共同提出一种新的字符串匹配算法——BM 算法，该算法是目前使用最为广泛、搜索效率很高的算法之一。在此基础上研究者又提出了 Tuned Boyer-Moore 算法[14]、Boyer-Moore-Horspool 算法[14]等基于 BM 算法的改进算法。

对于基因序列而言，通常是基于 Off-line 模式的搜索，这是因为该模式可以允许文本 S 建立索引结构。其主要的应用场景为文本 S（待匹配的基因序列）是已知的，且具有稳定的特性，在对文本 S 进行过预处理后，输入一个或多个模式 P（进行匹配的基因序列），要求在文本 S 中找出所有可能的模式 P。现假设模式 P 的长度为 m，该模式 P 在文本 S 中出现的次数为 k。通过对文本 S 进行后缀树的构造（即进行预处理操作），可以在$O(m+k)$[15]时间内找出所有在文本 S 中出现过的模式 P，该时间与文本 S 的长度无关。也就是说从文本 S 中找出任何一个模式所耗费的时间仅与该模式的长度成正比。

由于基因序列具有稳定的特性，而且一般在进行搜索之前基因序列是已知的，因此在大多数情况下都需要先对基因序列建立索引结构，然后再进行搜索。这样做的目的是可以大大的提高搜索效率。

## 2.2 基因序列索引结构

基因序列的相似性往往导致其功能上的相似性。因此，研究者完全可以通过搜索相似序列来预测新基因序列的功能。基因序列的相似性查询[16]是指在基因序列数据库中查找与查询序列相似的基因序列的技术，该技术可以分为基于索引的查询和非索引的查询两类。非索引的查询需要检索整个数据库，因此其性能并不理想。基于索引的查询不需要检索整个数据库，因此性能要优于非索引的查询。索引的作用相当于图书的目录，根据目录可以快速找到所需的内容。

目前，基因序列数据库可采用的索引结构多种多样，其中广泛使用的有后缀树、后缀数组和 q-gram 索引。

### 2.2.1 后缀树

在具体介绍后缀树之前，需要对什么是字符串 S 的后缀有所了解，下面给出字符串 S 及其后缀的详细定义。

定义 2.1 ∑表示字符串 S 的取值集合，我们定义∑的取值范围是$\sum \in \{\sum | 1 \le \sum \le 26 \cup 33 \le \sum \le 58\}$，其中 1、26、33、58 分别是字母 A、Z、a、z 相应的 ASCII 码的十进制表示形式[9,10]。

定义2.2 长度为n+1的字符串S表示为$S = s_0 s_1 \dots s_{n-1}\$(s_i \in \sum, 0 \le i \le n+1, \$ \ni \sum)$。$是特殊字符，该特殊字符不属于∑集合，只是用来表示字符串到这里结束[9,10]。

定义 2.3 字符串 S 的后缀表示为$S_i = s_i s_{i+1} \dots \$(s_i \in \sum, 0 \le i \le n, \$ \ni \sum)$[9,10]。

有了以上知识作为铺垫，现给出字符串 S 相应的后缀树的定义：长度为 n+1 的字符串 S 对应的后缀树是一颗有向树，该有向树恰好有 n+1 个编号为 0 至 n 的叶子结点。后缀树中结点的标识被定义为从根结点到该结点的所有的边标识的串联。下面给出后缀树的相关性质：

性质 2.1 对于任意一个叶子结点 i，从根结点到该叶子结点所经历的边的所有标识串联起来恰好组成字符串 S 的一个后缀，该后缀的起始位置是 i，即 $S[i \dots n]$[10]。

性质 2.2 每个内部结点的每条分支的首字符一定不相同[10]。

性质 2.3 除去根结点，每个内部结点都至少有两个孩子并且每条边都被相应

字符所标记。如果某个内部结点只有一个孩子，那么就将其与父结点合并[10]。

为了加深对后缀树的认识，现通过一个具体的例子来做进一步的介绍。例如，长度为 10 的字符串 ATTAGTACA$（$为结束符）的后缀树如图 2.1 所示。从图 2.1 中可以看出该后缀树共有 10 个叶子结点，也就是说该字符串有 10 个后缀，这是因为从根结点出发到达每个叶子结点所经历的边标识串联起来后都是该字符串的一个后缀。从根结点到第 5 个叶子结点串连出的后缀是 TTAGTACA$，从根结点到第 10 个叶子结点串连出的后缀是$，这里需要强调的是结束符$也是一个后缀。

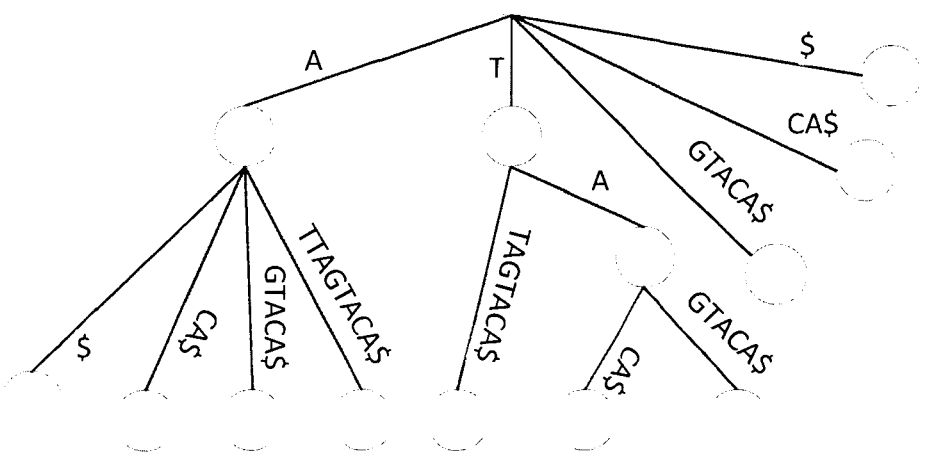


**图 2.1 字符串 ATTAGTACA$的后缀树**

### 2.2.2 后缀数组

为超长的基因序列构建后缀树并不是一个好的选择，这是因为与该基因序列相对应的后缀树所占用的内存空间是极其庞大的，所以空间问题是后缀树的一大瓶颈。相对于后缀树，后缀数组采用数组作为存储结构，因此占用的存储空间更小，也更加简单易懂[11]。

定义长度为 n 的后缀数组是一个长度为 n 的整型数组，该整型数组仅仅保存了按照字母表顺序排列的后缀的索引号而不保存字符串中的字符信息。正是由于该特性使得后缀数组占用的内存空间比后缀树要少的多。

比较经典的后缀数组构造算法有 R.M.Karp，R.E.Miller 和 A.L.Rosenberg 提出的 KMR 算法[11]，N.J.Larsson 和 K.Sadakane 提出的 LS 算法[11]，另外 D.K.Kim，J.S.Sim，H.Park 和 K.Park 在文献[11]提出了一种构造后缀数组的线性

算法。

### 2.2.3 其他索引结构

除去已经介绍的后缀树和后缀数组这两种索引结构之外，还有一类索引结构要不就是针对某些特定数据设计的，要不就是针对后缀树、后缀数组进行的改进。这类索引结构的典型代表就是 q-gram 索引[12]。

q-gram 是一种松散的索引结构其允许搜索的基因序列的长度最长为 q，该结构按照递增的顺序存储 q-gram 在基因序列中的所有位置。虽然该结构所需的内存空间更少，但是这也限制了从索引中搜索到的基因序列的长度。

q-sample 是一种占用内存空间更少的索引结构，该结构只是存储了一些 q-gram 而这些 q-gram 被称为 q-sample[12]。相对于 q-gram 而言，q-sample 不仅对内存空间需求更低，而且还可以检索更长的基因序列。

## 2.3 后缀树索引结构的建立

由于后缀树的查询时间仅与要查找的字符串的长度成正比，因此后缀树可以实现字符串的快速检索[9]。既然后缀树在进行搜索时拥有如此高的效率，那么如何快速实现后缀树的构造就显得尤为重要。

经典的后缀树构造算法有：Weiner 算法[9,17]、MCC 算法[10,17]和 Ukkonen 算法[17,18]。虽然这三种算法的时间复杂度都是$O(n)$（n为字符串的长度），但是 Weiner 算法和 MCC 算法占用的内存空间更多，因此使用的并不广泛[17]。Ukkonen 算法则不然，该算法不仅保留了 Weiner 算法和 MCC 算法的优点，而且占用的内存空间更少。下面针对 Ukkoenen 算法进行详细介绍，在此之前，需要储备以下知识。

### 2.3.1Ukkonen 算法相关术语

首先给出 Ukkonen 算法中涉及到的相关术语：

定义 2.1 隐式后缀树是指后缀可能终止于叶子结点，也可能隐藏在内部结点中。如果输入串中最后一个字符不同于其他字符，那么所有的后缀都终止于叶子结点，不会有后缀隐藏在内部结点中[17,18]。这就是为什么要求最后一个字符必须是特殊字符的原因。

定义 2.2 阶段 i+1 是指将$S[i+1]$考虑进来并将$S[0 \ldots i+1]$的所有后缀加入到

上一个阶段 i 生成的隐式后缀树中，从而形成一个新的隐式后缀树[17,18]。

定义 2.3 扩展是指在每个阶段中，需要将每一个后缀加入到上一个阶段的隐式后缀树中，每个后缀加入操作叫做扩展 j[17,18]。

定义 2.4 后缀链是一个指针，用来简化下一次扩展位置的查找。只有内部结点才有后缀链，叶子结点无需后缀链[17,18]。

后缀链[19,20]是 Ukkonen 算法的核心。下面对其做进一步的介绍。后缀链本质上是一个指针，用来简化下一次扩展位置的查找。比如用xα代表一个任意字符串，其中 x 代表单个字符，α代表一个可能为空的子字符串。对后缀树中的一个路径标签为xα的结点$v$，如果存在一个路径标签为α的结点μ，则存在一个从$v$到μ的指针，这个指针就是后缀链。例如，字符串 CACAT 的后缀链，如图 2.2 所示。从图 2.2 可以看出后缀链有(1,5)和(5,root)。由于编号为 1 的结点和编号为 5 的结点之间存在后缀链，因此在对编号为 5 的结点进行边 CAT 的构造时，并不需要完全遍历已经构造出来的部分后缀树，只需要通过这个后缀链就可以直接找到编号为 5 的结点。

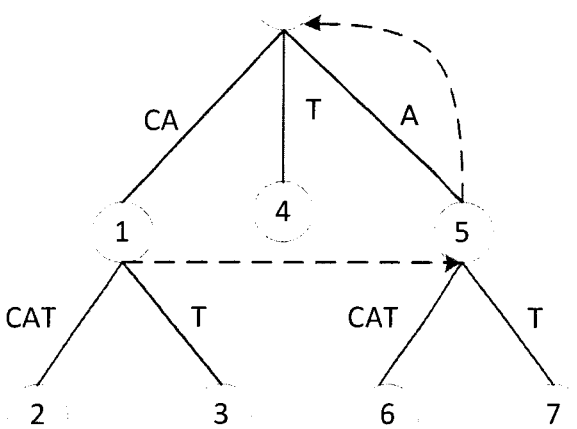


**图 2.2 字符串 CACAT 的后缀链**

### 2.3.2Ukkonen 算法设计

Ukkonen 算法是一种经典的后缀树构造算法，该算法引入了路径压缩技术和后缀链的概念[17,19]。这两大技巧不仅使得 Ukkonen 算法在运行时占用更少的内存空间，而且还将 Ukkonen 算法的时间复杂度降为线性。路径压缩技术是指每条边不只包含一个字符，也有可能包含若干个字符，该技术可以大大的减少内存空间的开销。后缀链是用来简化下一次扩展位置的查找，该技术可以将算法的时间复

杂度降为线性。

Ukkonen 算法的基本设计思想是假设 S 是一个非空字符串，c 是一个字符，那么将字符 c 追加到非空字符串 S 的每个后缀（包括空后缀）上，我们就可以得到字符串 Sc 的所有后缀[17,18]。下面通过一个简单的例子对 Ukkonen 算法做进一步的介绍。

定义字符串 S=ACAGT，对该字符串使用 Ukkonen 算法进行后缀树的构造过程如图 2.3 所示。首先对字符串 A 进行后缀树的构造，结果如图 2.3 中 a 所示；然后在字符串 A 后追加字符 C，从而对字符串 AC 进行后缀树的构造，结果如图 2.3 中 b 所示；紧接着在字符串 AC 后追加字符 A，从而对字符串 ACA 进行后缀树的构造，结果如图 2.3 中 c 所示，因为字符 A 是字符串 ACA 的前缀，即字符串的当前结尾字符在前面的位置出现过，因此需要做进一步的处理，结果如图 2.3 中 d 所示；之后在字符串 ACA 后追加字符 G，从而对字符串 ACAG 进行后缀树的构造，结果如图 2.3 中 e 所示。

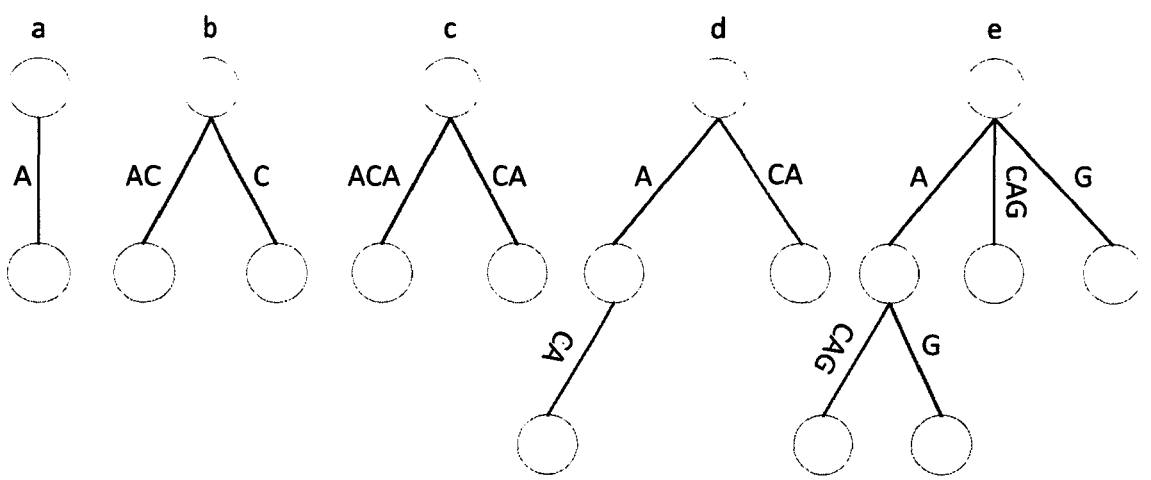


**图 2.3 字符串 ACAGT 的后缀树构造过程**

上述例子模拟了最直接、最简单的 Ukkoenen 算法构造后缀树的过程，下面给出该算法的代码描述。

输入：字符串 S；

输出：字符串 S 的后缀树；

算法描述：

```
Algorithm: Ukkonen
1   n= S.length;
2   strcat(S,'$'); //S 后追加$
3   for (i=0;i<n;i++){        //n+1 个 prefix phase 阶段
4       for(j=0;j<i-1;j++){      //每个 suffix phase 阶段 i 次扩展
5           v=Find_end_path_from_root(); //根结点顺着T[j ... i − 1]走到结点 v
6           Apply_suffix_extension_rules(v){
7               if(get(v)== S[i]){
8                   continue;
9               } else if(v.flag==0){ //0 表示是隐式结点
10                  v.flag=1; //将结点 v 设为显式结点;
11                  add leaf to v;
12                  add(S[i]);
13              }else if(v.flag==1){ //内部结点
14                  add leaf to v;
15                  add(S[i]);
16              }else{   //只能是叶子结点
17                  add(S[i]);
18              }
19          }
20      }
21  }
```

虽然该算法简单、易懂，但是其时间复杂度达到$O(n^3)$（n 为字符串 S 的长度）[17,18]。显然如此低的效率是不具有任何工业价值的，因此需要对这个算法进行如下改进，从而使得时间复杂度降为线性。

（1）对该算法的 Apply_suffix_extension_rules()函数应用以下三条规则来减少遍历次数，从而达到提高算法效率的目的。

a. 如果$S[j \dots i]$的最后一个字符在叶子结点中，那么直接将$S[i+1]$添加到

$S[j \ldots i]$后面[19,20]。

b. 如果$S[j \ldots i]$的最后一个字符在不在叶子结点中，而且当前的隐式后缀树中该路径上的下一个字符 c 不等于$S[i+1]$，那么需要生成一个新的结点[19,20]。

c. 在规则 b 中，如果下一个字符 c 等于$S[i+1]$，则不需要做任何操作。因为当前后缀$S[j \ldots i+1]$已经存在于隐式后缀树中[19,20]。

（2）对该算法的 Find_end_path_from_root()函数而言每次都要从根结点开始进行逐一搜索直到找到待处理的结点为止，显然这样处理的效率并不高。通过引入后缀链可以避免每次都从根结点开始搜索，从而实现结点的快速定位，如此以来可以大大的提高处理效率[17,19]。

虽然后缀链可以将 Ukkonen 算法的时间复杂度降为$O(n)$（$n$为字符串 S 的长度），但是却也带来了“内存瓶颈”问题。这是因为对后缀树的遍历可以采用后缀链遍历和父子链遍历这两种方式[21,22]。在使用 Ukkoenen 算法构造后缀树的过程中不可避免的会出现以下情况：至少有一种遍历方式需要对内存进行随机访问。由于计算机都采用多级存储体系结构，因此在对内存进行随机访问时不可避免的会产生命中率低的情况。如果不采用适当的手段，后缀树的大小将是原字符串大小的 12 倍左右[22]，这将导致无法为长基因序列构造后缀树。

## 2.4 本章小结

本章主要介绍了基因序列索引技术的相关工作，包括对基因序列的相关介绍，基因序列搜索技术的介绍，基因序列索引结构的介绍，最后针对基于后缀树的基因序列索引结构进行了深入的研究分析，其中主要介绍了经典的后缀树构造算法——Ukkonen 算法。这些相关工作的讨论为本文的进一步研究提供了可能。

# 第3章 基于LM-Suffix算法的后缀树索引结构

经典的 Ukkonen 算法虽然可以在线性时间内构造后缀树，但是由于引入了后缀链，因此产生了“内存瓶颈”问题，从而导致不支持长字符串的处理。本文提出了一种分步构造后缀树的方法——LM-Suffix 算法。该算法在尽量保证构造效率的同时还消除了传统的后缀树构造算法不支持长字符串处理的问题。最后设计了一种适合于 LM-Suffix 算法的搜索方案。

## 3.1LM-Suffix 算法思想

### 3.1.1 定义介绍

首先给出 LM-Suffix 算法需要用到的定义：

定义 3.1　字符串 S 的后缀的前缀表示为 S-Prefix。

定义 3.2　将字符串 S 的后缀树进行划分：$T_P$表示以 p 开始的子树，此时 S-Prefix=p；$F_P$表示以 p 开始的所有后缀的个数，此时 S-Prefix=p。

定义 3.3　MTS 表示系统分配给$T_P$的内存空间。

定义 3.4 后缀树的结点用 u 表示，边用 e 表示，边标识用label($e$)表示。

定义 3.5 字符深度为从根结点到该结点的字符长度。

### 3.1.2 基本思想

公式 3.1 是 LM-Suffix 算法实现划分功能的基础。在给出该公式的具体推导过程之前需要引入定理：对一颗$T_P$而言，每条分支的内部结点数目等于其叶子结点数目[23]。该定理在文献[23]中进行了详细的阐述。

$$F_P \leq \frac{MTS}{2\times \text{sizeof(node)}} \qquad \text{（公式 3.1）}$$

推导：定义 Ln 为叶子结点的数目。对于结点数为 N 的一颗树而言，其占用的内存空间为N × sizeof(node)。根据文献[23]中的定理可知，一颗后缀树$T_P$占用的内存空间为2Ln × sizeof(node)，又因为在后缀树中，从根结点到叶子结点的一条路径就是一个后缀，因此 Ln=$F_P$，也就是说后缀树$T_P$占用的内存空间为$2F_P$ × sizeof(node)。MTS 为系统分配给$T_P$的内存空间，如果要保证程序的正常运行，那么必然满足该式子MTS ≥ $2F_P$ × sizeof(node)。通过变形即可得到公式

3.1。下面给出由公式 3.1 得出的几个结论。

（1）在sizeof(node)一定的条件下，$F_P$与$T_P$占用的内存空间成正比关系。

（2）$F_P$与$T_P$成反比关系。即如果$F_P$变小，那么$T_P$就会变大；如果$F_P$变大，那么$T_P$就会变小。

（3）当$2F_P \times$ sizeof(node) $\leq$ MTS时表示系统分配的内存空间有剩余，反之则表示系统分配的内存空间不足。

在此基础上给出 LM-Suffix 算法的基本思想：首先将长字符串划分为若干个短字符串，这时根据内存可用空间的大小来确定调入其中的短字符串的个数，紧接着对已调入内存中的若干短字符串按照字母表顺序进行排序，然后根据两两相邻的短字符串的最大公共前缀来确定分支点，最后根据求得的分支点来构造相应的子后缀树。LM-Suffix 算法通过将一个长字符串划分为若干个短字符串为解决传统的后缀树构造算法不支持长字符串的问题提供了可能。该算法将划分后的短字符串按照字母表顺序进行排序，然后根据两两相邻的短字符串的最大公共前缀来确定分支点，这样做的目的是尽量消除放弃使用后缀链带来的效率骤降问题。

本文提出的 LM-Suffix 构造算法不仅可以很好的支持长字符串的处理，而且还保留了传统的后缀树构造算法高效的特点。由于 LM-Suffix 构造算法在将短字符串调入内存时会尽可能多的将其调入，因此可以在一定程度上减少 I/O 开销。

## 3.2LM-Suffix 算法设计

### 3.2.1 长字符串划分为短字符串

本小节的所有研究都是以公式 3.1 为基础的。这是因为公式 3.1 表明了$F_P$与内存之间的隐含关系，通过这个隐含关系来决定调入内存中的短字符串的个数。本小节为解决传统的后缀树构造算法不支持长字符串的问题奠定了基础。

为了便于理解本小节的内容可以将划分看作是在逻辑上把一颗完整的后缀树划分为若干个子后缀树，下面结合具体的例子来加深对该部分内容的理解：为了与基因序列相对应，定义字符串的取值集合$\sum = \{A, C, G, T\}$，其中 A、C、G、T 为基因序列中的四个碱基。在此基础上定义一个字符串 S=TGGTGGTGGTGCGGTGATGGTGC$，$为一个结束符，用来表示字符串的结束，如图 3.1 所示。

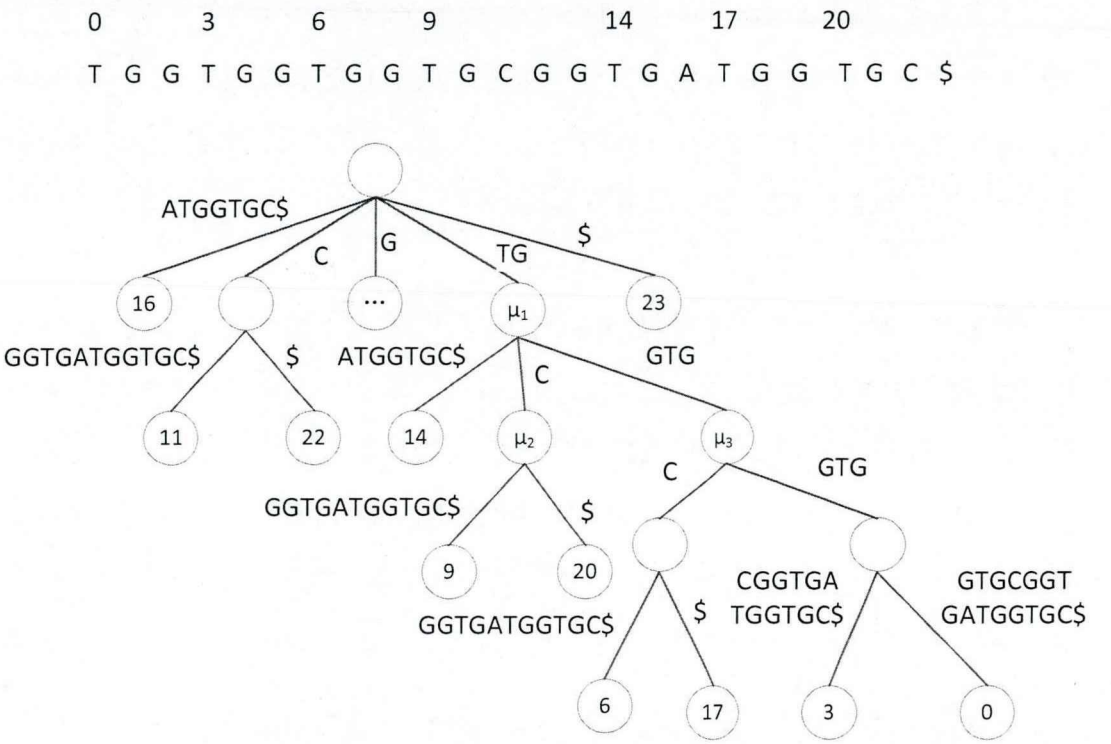


**图 3.1 字符串 S 及其相应的后缀树**

首先可以考虑将这个字符串 S 对应的一颗完整的后缀树（如图 3.1 所示）划分为五颗子树：$T_A$、$T_C$、$T_G$、$T_{TG}$、$T_\$$。$T_A$就表示以字符 A（S-Prefix=A）开始的子树，$T_{\mathrm{TG}}$表示以字符 TG（S-Prefix=TG）开始的子树。$F_{\mathrm{TG}}$表示以字符 TG（S-Prefix=TG）开始的所有后缀的个数，显然$F_{\mathrm{TG}} = 7$。如表 3.1 所示。

**表 3.1 字符串 S 中以 TG 开始的子串**

| i | $S_i$ | Suffix |
|---|---|---|
| 0 | $S_0$ | TGGTGGTGGTGCGGTGATGGTGC$ |
| 3 | $S_3$ | TGGTGGTGCGGTGATGGTGC$ |
| 6 | $S_6$ | TGGTGCGGTGATGGTGC$ |
| 9 | $S_9$ | TGCGGTGATGGTGC$ |
| 14 | $S_{14}$ | TGATGGTGC$ |
| 17 | $S_{17}$ | TGGTGC$ |
| 20 | $S_{20}$ | TGC$ |

由公式 3.1 可知，$F_P$与$T_P$成反比关系。对于上述例子而言，如果在字符 TG 后依次追加集合$\sum = \{A\ C\ G\ T\}$中的每个元素从而使得字符 TG 增长为 TGA、TGC、TGG、TGT，这样以来本来是一个以 TG 开始的后缀，现在被调整为四个分别以 TGA、TGC、TGG、TGT 开始的后缀，且$\mathrm{F}_{TGA} = 1, \mathrm{F}_{TGC} = 2, \mathrm{F}_{TGG} = 4, \mathrm{F}_{TGT} = 0$。从这个小例子可以看出通过增加$\mathrm{T_P}$的个数可以达到减少$\mathrm{F_P}$的目的。

根据表 3.1 可知，$F_{\mathrm{TG}}=7$，现假设$\frac{MTS}{2\times\mathrm{sizeof(node)}}=5$，即内存可容纳的后缀的个数为 5 个。显然在这种情况下，如果采用传统的后缀树构造算法——Ukkonen 算法来处理该问题会产生“内存瓶颈”问题。文献[23]提出的划分算法虽然可以解决上述问题，但是该算法在对长字符串进行划分的过程中会产生不平衡的子树从而导致不必要的 I/O 开销，而且对于多核 CPU 而言，会导致某个 CPU 长时间处于闲置状态。正是由于上述原因，该算法在最坏情况下的时间复杂度为$O(n^2)$[23]（$n$为字符串 S 的长度）。

通过上面的例子可知，在字符 TG 后依次加入集合$\sum=\{A\ C\ G\ T\}$中的元素可以将以字符 TG 开始的后缀划分为三个（因为$F_{TGT}=0$）分别以字符 TGA、字符 TGC、字符 TGG 开始的后缀，且相应的后缀的个数分别为$F_{TGA}=1$，$F_{TGC}=2$，$F_{TGG}=4$。通过进一步的研究分析可知$F_{TGA}+F_{TGG}=5=\frac{MTS}{2\times\mathrm{sizeof(tree-node)}}$，那么可以考虑将这两个后缀一起调入内存中进行处理，即以字符 TGA 和字符 TGG 开始的后缀将被合并到同一个分组中，以字符 TGC 开始的后缀成为另一个分组。这样一来不仅可以减少 I/O 开销，而且还可以提高 CPU 的利用率。显然对于该例子而言，如果不进行合并，那么需要三次 I/O 操作。现在通过合并操作，只需要两次 I/O 操作。下面给出 LM-Suffix 算法对于文献[23]中提出的划分算法的具体改进方案。

首先对存放 S-Prefix 的链表按照$F_{\mathrm{P}}$递减的顺序进行排序，然后将链表中的第一个结点移走，之后遍历该链表。如果被移走的第一个结点的$F_{\mathrm{P}}$值与现在遍历的结点的$F_{\mathrm{P}}$的值之和小于等于$\frac{MTS}{2\times\mathrm{sizeof(node)}}$，那么将这两个 S-Prefix 归并为同一个分组，其实就是将子树合并。如果它们之和大于$\frac{MTS}{2\times\mathrm{sizeof(node)}}$，那么什么也不做。如此循环直至存放 S-Prefix 的链表为空。下面给出 LM-Suffix 算法关于划分部分的代码描述。

输入：字符串 S；集合$\sum=\{A\ C\ G\ T\}$；$\mathrm{R}=\frac{MTS}{2\times\mathrm{sizeof(node)}}$；

输出：划分后的若干短字符串；

算法描述：

Algorithm: Divide

```
1   P=null;//S-Prefix 的链表
2   Q={every symbol s do generate S-Prefix p_i}
3   repeat
4       scan S;
5       count F_{p_i};
6       while(p_i!=null){
7           if (0 < F_{p_i} ≤ R)
8               add p_i to P;
9           else{
10              every symbol s do add p_is to Q;
11              remove p_i from Q;
12      }
13      }
14  until(Q=null)
15  Sort(P);
16  while(P!=null){
17      G=null;       //存放分组
18      add head P to G;
19      remove the element from P;
20      curr=next element in P;
21          while(p!=null){
22              if(sum+F_curr ≤ R){    //sum 为 G 中的元素之和
23                  add curr to G;
24                  remove the element from P;
25              }
26              curr=next element in P;
27          }
28  }
```

该方案的第 3 步至第 14 步实现了划分功能，通过第 7 步中的判断条件（即公式 3.1）来决定到底如何进行划分。第 15 步至第 28 步实现了后缀的合并，通过第 22 步中的判断条件（即公式 3.1）来决定到底如何进行合并。这再一次证明了公式 3.1 的重要性。

该改进方案将原本不平衡的子树合并到同一个分组中，这样不仅可以大大的减少 I/O 开销，而且还可以提高 CPU 的利用率。这是因为 LM-Suffix 算法中的每个分组就是一个独立的处理单元，所以当需要将子树调入内存时，根据内存可用空间的大小，如果内存的可用空间足够，那么就会将同一个分组中的所有子树全部调入内存中，这样就减少了 I/O 开销，提高了处理效率。

### 3.2.2 确定分支结点

将一个长字符串划分为若干短字符串之后，下面要做的工作就是如何构造出相应的后缀树。虽然使用后缀链可以在线性时间内构造后缀树，但是会产生“内存瓶颈”问题[24]。本文提出的 LM-Suffix 算法通过将划分后的短字符串按照字母表顺序进行排序，然后根据两两相邻的短字符的最大公共前缀来确定分支点。这样做的目的是尽量消除放弃使用后缀链带来的效率骤降问题。

首先以图 3.1 中的以字符 TG（S-Prefix=TG）开始的子树$T_{TG}$为例对本文提出的 LM-Suffix 构造算法进行简单的介绍。在介绍之前，首先定义一个整型 Array L 用于记录 S-Prefix p 在字符串 S 中的起始位置，然后再定义一个三元组 B($b_1$, $b_2$,comm)用于记录分支结点之间的关系，$b_1$和$b_2$分别记录了两个相邻的短字符串出现分支时的首字符，comm 记录了这两个相邻的短字符串的公共前缀的长度。如表 3.2 所示，给出了以字符 TG（S-Prefix=TG）开始的子树$T_{TG}$的所有结点的分支关系。

**表 3.2 以字符 TG 开始的后缀树结点的分支关系**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| L | 14 | 9 | 20 | 6 | 17 | 3 | 0 |
| B | | (A,C,2) | (G,$,3) | (C,G,2) | (G,$,6) | (C,G,5) | (C,G,8) |

比如，当 L[3]=6 时，从表可知，B[3]=(C,G,2)。这就表示图 3.1 中的以字符 TG（S-Prefix=TG）开始的子树$T_{TG}$的第 3 个后缀与第 2 个后缀的分支首字符为 C、G，且它们的公共前缀的长度为 2。为了实现上述功能，首先需要定义一些相关

变量。下面对这些变量进行详细的介绍。

Array I 是一个索引数组，通过遍历该数组可以实现短字符串的快速定位。Array A 是一个标记数组，通过遍历该数组可以知道哪些短字符串已经被处理过了，哪些还没被处理过。Array L 是一个位置数组，用来记录短字符串在字符串 S 中的位置。Array P 用于记录各个后缀初始状态时的顺序。通过遍历 Array P 就可以知道后缀的初始顺序。Array R 用于存储待处理的字符。Array B 是一个三元组，用来记录分支结点之间的关系。

现结合图 3.1 中以字符 TG（S-Prefix=TG）开始的子树$T_{TG}$为例对该算法进行详细的介绍。首先，分别对 Array P、Array L、Array I、Array A 和 Array R 进行初始化，其结果如表 3.3 所示。比如，当i = 3时，首先通过索引数组 I[3]=3 来查找到位置数组 L[I[i]]=L[3]=9，然后从字符串 S 的第 11 号（|TG|=2，9+2=11）位置开始连续读取之后的 4 个字符，最后将读取的字符写入数组 R[3]中。

**表 3.3LM-Suffix 算法初始化情况**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| L | 0 | 3 | 6 | 9 | 14 | 17 | 20 |
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R | GTGG | GTGG | GTGC | CGGT | ATGG | GTGC | C$ |

然后，对 Array R 按照字母表顺序进行排序并且根据 comm（公共子串）来计算数组 B 的值，其结果如表 3.4 所示。比如，当 i=1 时，R[0]=ATGG、R[1]=CGGT，显然 R[0]和 R[1]是没有公共前缀的，即 comm=0，此时将 B[1]的值设为 B[1]=(R[0][0],R[1][0],2+0)=(A,C,2)，与此同时设置 I[P[0]]=I[4]=已处理、A[0]=已处理。

**表 3.4LM-Suffix 算法确定分支结点情况**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| P | 4 | 3 | 6 | 2 | 5 | 0 | 1 |
| L | 14 | 9 | 20 | 6 | 17 | 0 | 3 |
| I | 5 | 6 | 3 | 已处理 | 已处理 | 4 | 已处理 |
| A | 已处理 | 已处理 | 已处理 | 1 | 1 | 2 | 2 |
| R | ATGG | CGGT | C$ | GTGC | GTGC | GTGG | GTGG |
| B | | (A,C,2) | (G,$,3) | (C,G,2) | | (C,G,5) | |

这时，如果 Array B 中仍然存在为空的元素，那么就需要针对这些未被处理的后缀继续读取字符到 Array R 中，然后重复上述比较过程，直到 Array B 中的

值确定为止，其结果如表 3.5 所示。比如，通过遍历 Array I 可知，该数组中的 I[0]、I[1]、I[2]、I[5]对应的后缀需要做进一步的处理。

**表 3.5LM-Suffix 算法确定分支结点情况**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| P | 4 | 3 | 6 | 2 | 5 | 0 | 1 |
| L | 14 | 9 | 20 | 6 | 17 | 0 | 3 |
| I | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 |
| A | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 | 已处理 |
| R | | | | GGTG | $ | TGCG | TGGT |
| B | | (A,C,2) | (G,$,3) | (C,G,2) | (G,$,6) | (C,G,5) | (C,G,8) |

最后，当 Array B 已经全部被处理，也就是说该算法执行完成，此时以字符 TG（S-Prefix=TG）开始的子树$T_{TG}$的全部分支信息被保存在 Array B 中。有了以上储备知识，现给出 LM-Suffix 算法关于确定分支结点部分的代码描述。

输入：字符串 S；S-Prefix p

输出：Array L；Array B

算法描述：

```
Algorithm: SuffixRelation
1   Array L=( ) ; //保存 S-Prefix p 在 S 中的起始位置
2   Array B=( ) ; //保存分支关系
3   Array I=(0,1,...,|L| − 1); //索引数组，|L|是后缀的个数
4   Array A=(0,0,...,0); //标记分支情况
5   Array R=( ); //存储边标识
6   Array P=(0,1,...,|L| − 1); //原始顺序
7   start=p;
8   while(B[i]==null){
9       range=4; //读取 range 个字符
10      for( i=0; i≤ |L| − 1;i++){ //初始化 R
11              if(I[i]!=done)
12                  R[I[i]]=ReadString(S,L[I[i]]+start,range); //找出 range 字符
13      }
14      Sort (R);   //按照字母表顺序排序
15      While(B[i]!=done){
16          comm is the common  S − prefixes  of R[i-1] and R[i]; //公共子串
17          if(comm<range){
18              B[i]=(R[i-1][comm],R[i][comm],start+comm);//分支信息和公共子串
19              if(B[i-1]==done || i==1){
20                  I[p[i-1]]=done;
21                  A[i-1]=done;
22              }
23              if(B[i+1]==done || i==|L| − 1){
24                  I[p[i]]=done;
25                  A[i]=done;
26              }
27          }
28      }
29      start=start+range;
30      i++;
31  }
```

该算法的第 8 步至第 13 步用来完成对 Array R 的初始化，在这个过程中根据 range 的值来决定每次读取的字符的个数。第 14 步实现了按照字母表顺序进行排序的功能，通过排序可以简化下次查找的位置，从而实现快速定位。第 15 步至第 28 步实现了对 Array B 的赋值，也就是说实现了确定分支信息的功能，通过第 17 步的判断条件来确定分支结点到底在哪里。第 29 步至第 31 步准备进入下次循环。

如果对待处理的后缀不进行排序的话，那么对于已经确定的部分后缀而言，它们在内存中的存储并不是有序的，这就导致了在添加新边的时候需要完全遍历这部分后缀才能确定在什么位置添加新边，显然这样做的效率并不高。正是由于上述原因该算法引入了排序功能，该功能的作用就是用来简化下次查找新边的位置。

### 3.2.3 构造子后缀树

3.2.2 小节完成了分支关系的计算，本小节以此为基础来完成相应的后缀树的构造。在给出具体的算法设计之前，通过下面的例子来对后缀树的构造过程进行介绍。现结合表 3.5 的处理结果来模拟 LM-Suffix 算法中关于构造子后缀树的执行流程，如图 3.2 所示，给出了以字符 TG 开始的子后缀树$T_{TG}$的构造过程。

首先将边 TGAT....$压入堆栈内并计算堆栈的高度 length=10（图 3.2 中 a 所示），然后对于存储边关系的 Array B 而言，B[1]=(A,C,2)=($c_1$, $c_2$, comm)，显然 length=10>2=comm，这时可以将边 TGAT...$分割为两条边，分别为 TG 和 AT...$，其中边 TG 连接的是分支结点与其父结点，边 AT...$连接的是分支结点与其左孩子，之后从 Array L 中取出下一个后缀（即 CG...$）作为该分支结点的另外一条边，也就是说边 CG...$连接的是该分支结点与其右孩子，最后将边 TG 和 CG...$一并压入堆栈内并计算堆栈的高度 length=15（图 3.2 中 b 所示）。进入下次循环后待处理的就是堆栈顶的边 CG...$，如此循环下去直至 Array B 中的元素全部被处理，那么一颗以字符 TG 开始的子后缀树$T_{TG}$就构造完成了。

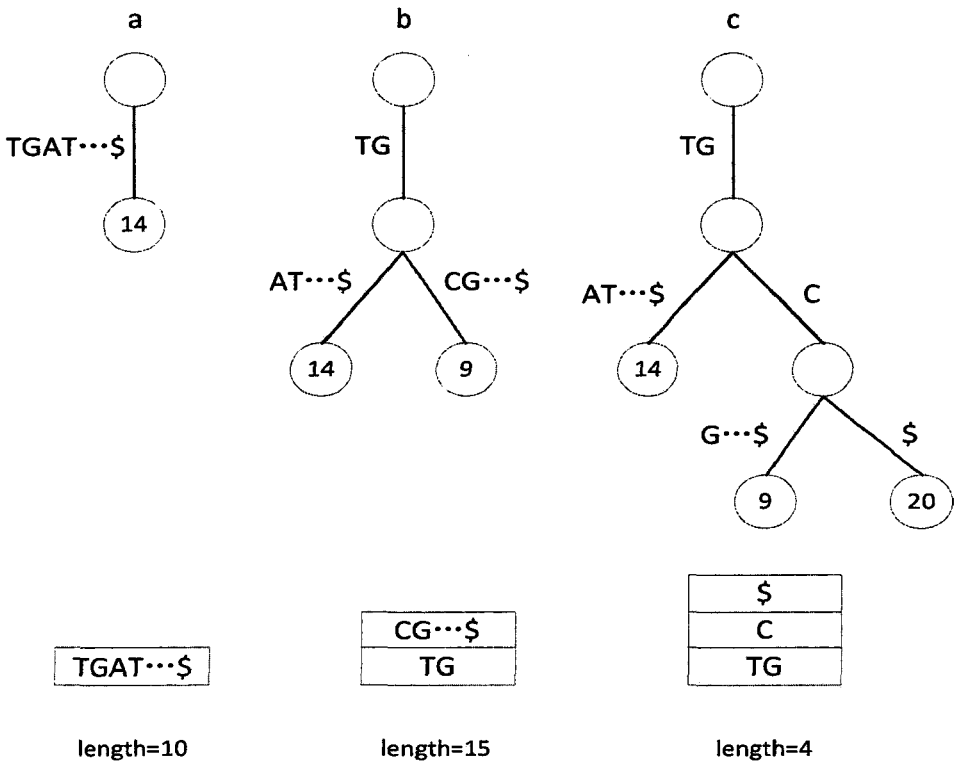


**图 3.2 以字符 TG 开始的后缀树构造过程**

上面的例子简单的介绍了 LM-Suffix 算法是如何实现子后缀树的构造的，简而言之，首先使用堆栈来保存边标识信息，然后通过进堆栈、出堆栈的操作配合记录分支关系的数组（即 3.2.2 小节中的数组 Array B）来完成子后缀树的构造。在此基础上，给出 LM-Suffix 算法关于构造子后缀树的代码描述。

输入：Array L；Array B

输出：$T_P$

算法描述：

```
Algorithm: SuffixBulid
1   root=new Node;
2   u′=new Node;
3   e′=new Edge(root,u′);
4   label e′=S-Prefix L[0];   //后缀位置
5   push(e′);   //入栈
6   length=|label(e′)|;
7   for(i=1; i< |B| − 1; i++){
8       (c1, c2, comm)=B[i];
9       while(length>comm){
10          pop(se); //出栈
11          length=length-|label(se)|;
12      }
13      if(length==comm) //不需要分割
14          u=v1;
15      else{
16          break  se(v1, v2)  to  se1(v1,vt)  and  se2(vt,v2);
17          label  se1=SubString(label se, comm);
18          label  se2= SubString(label se, length-comm);
19          u=vt;      //u 为分支结点
20          push(se1);
21          length=length+|label(se1)|;
22      }
23      u′=new Node;
24      ne=new Edge(u,u′);
25      label ne=S-Prefix L[i];   //分支结点的右孩子的边标识
26      push(ne);
27      length=length+|label(ne)|;
28  }
```

该算法的第 4 步至第 6 步借助堆栈来存储边的标识信息，第 9 步至第 13 步通过与 comm（公共子串）进行数值比较来确定在什么位置产生分支结点，第 15 步至第 22 步对边标识进行分割处理，第 23 步至第 27 步产生新的分支结点，并准备进入下次循环，如此反复直至构造出子后缀树。

## 3.3 内存分配

LM-Suffix 算法在构造后缀树的过程中实现了内存的动态调整，即将某些不再使用的数据所占用的内存空间释放掉，以便其他正在运行的程序使用。这样不仅可以最大程度的利用内存，而且还可以提高算法的处理效率。在介绍内存的动态调整之前，需要了解 LM-Suffix 算法是如何对内存进行划分的。LM-Suffix 算法将内存划分为三个部分，如图 3.3 所示。

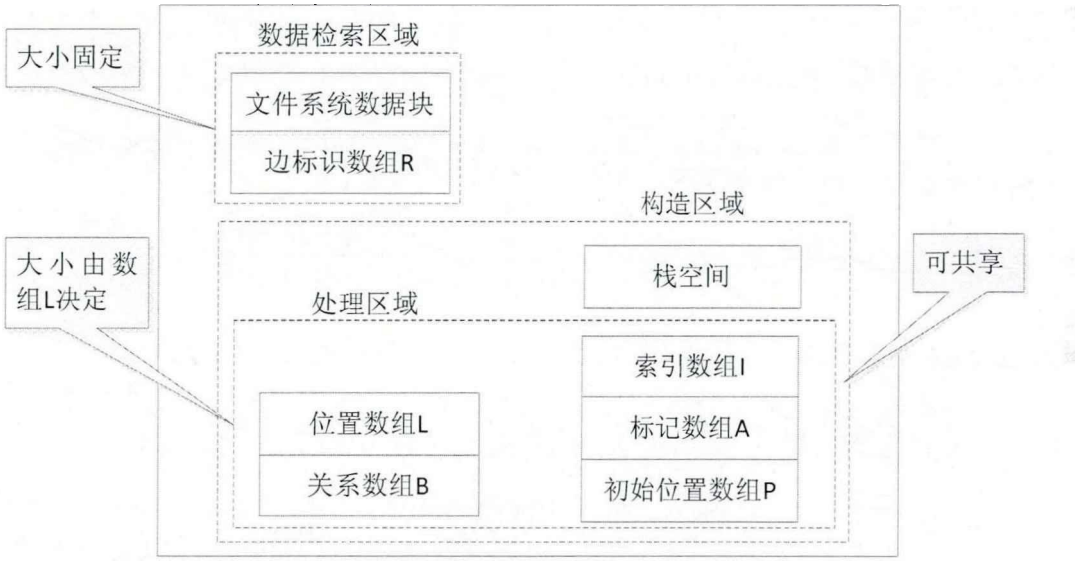


**图 3.3LM-Suffix 算法内存分配情况**

数据检索区域是系统分配的大小固定的一块内存空间。该区域包含有文件系统数据块和存储边标识的 Array R，其中文件系统数据块的大小由系统本身决定，Array R 的大小由程序本身确定。

处理区域是确定分支结点算法（SuffixRelation）的运行区域。该区域包含有 Array L、Array B、Array I、Array A 和 Array P。处理区域占用的内存空间大小由 Array L 确定。这是因为 Array L 的长度就是后缀的个数，显然后缀越多所需的内存空间就越大。

构造区域是构造子后缀树算法（SuffixBulid）的运行区域。该区域包含有自

定义的堆栈结构以及处理区域中的 Array L 和 Array B。构造区域占据了剩下的可用内存空间。

之所以将 Array R 放入数据检索区域而不是放入处理区域是因为这样做可以减少 I/O 开销，从而提高算法的处理效率。当某个后缀已经被处理就会在 Array A 中的相应位置标记为已处理，那么相应的这个后缀在 Array R 中占用的空间就会被释放掉。在下次循环时，由于已经被处理过的后缀不再占用内存空间，因此可以将更多的待处理后缀调入内存。这样一来就达到了减少 I/O 的目的，因为一次遍历能够读入更多的待处理后缀到 Array R 中。

从图 3.3 中可以看出处理区域和构造区域是可以重叠的，即可以共享内存空间。正是由于这个小技巧使得内存的动态调整成为了可能。简单来说，确定分支结点算法（SuffixRelation）中的 Array L 和 Array B 将会作为构造子后缀树算法（SuffixBulid）的输入被保存在内存中，但是 Array I、Array A 和 Array P 在构造子后缀树算法（SuffixBulid）中并没有被使用，因此可以将这三个数组占用的内存空间释放掉。如此以来，构造子后缀树算法（SuffixBulid）可以使用的内存空间就会大大的增加。通过这样的调整不仅可以最大限度的利用内存，而且还可以提高算法的处理效率。

## 3.4LM-Suffix 算法的搜索技术

本文之所以研究基因序列索引结构其目的在于可以更加高效、快速的在基因序列数据库中检索信息。本小节就来研究适合于 LM-Suffix 构造算法的搜索技术。在此之前，需要对后缀树索引结构的搜索流程有所了解，对后缀树索引结构进行搜索的主要步骤[25,26]是：

（1）从后缀树 T 的根结点出发，遍历根结点的所有孩子结点；

（2）如果所有孩子结点中的第一个字符都和模式 P 的第一个字符不匹配，那么搜索失败，算法结束；

（3）如果存在一个孩子结点的第一个字符和模式 P 的第一个字符相同；那么会遇到以下两种情况：

a. 如果该孩子结点的边的长度大于等于模式 P 的长度且该孩子结点的边上的每一个字符都和模式 P 的字符匹配成功，那么搜索成功，

算法结束；

b. 如果该孩子结点的边的长度小于模式 P 的长度且该孩子结点的边上的每一个字符都和模式 P 的字符匹配成功，那么将匹配成功的子字符串从模式 P 中删除，然后将该孩子结点作为根结点继续重复（1）至（3），直至模式 P 中的字符匹配完；

虽然该方案可以实现后缀树索引结构的搜索，但是并不适用于由 LM-Suffix 构造算法所构造出来的后缀树。这是因为本文提出的 LM-Suffix 构造算法将一颗完整的后缀树划分为若干个子后缀树，因此在进行搜索时需要考虑多个子树的情况。

本文通过对该方案进行深入的研究分析同时结合 LM-Suffix 构造算法的特点提出了一种适合于 LM-Suffix 算法的搜索方案。本文提出的搜索方案的主要改进在于使用一个哈希表来记录子后缀树被访问的次数。在进行搜索之前，首先通过遍历这个哈希表将经常被访问的子后缀树调入内存，然后进行搜索。如果匹配失败，则将哈希表中的下一个子后缀树调入内存进行搜索，如此下去，直到匹配成功为止。下面给出该方案的具体步骤。

输入：子树的根结点；模式 P

输出：否是存在，计数，枚举

算法描述：

（1）确定模式 P 当前的字符与内存中的子树的当前结点的哪条边对应，在这个过程中会出现以下两种情况：

a. 如果找到对应的边，直接进入（2）；

b. 如果找不到对应的边，那么从磁盘中选取下一颗子树读入内存中，然后再执（1）；

（2）然后从该边的起始位置开始对边标识与模式 P 中的字符进行逐个比对直至出现下面的情况：

a. 匹配失败，也就是说边标识上的某个字符与模式 P 的某个字符不匹配，直接跳转至（3）；

b. 当前的边标识匹配成功，需要对下一个结点（该结点是存在孩子的）的边标识进行匹配，直接跳转至（1）；

c. 匹配到模式 P 的最后位置，直接跳转至（4）；

（3）搜索失败，算法结束。

（4）搜索成功，根据需求输出相应的结果：

a. 是否存在：存在。

b. 计数：递归访问以当前边标签所到达的结点为根的子后缀树，计算所有的叶子结点数，并输出该数目。

c. 枚举：递归访问以当前边标签所到达的结点为根的子后缀树，输出所有叶子结点对应的后缀编号。

在步骤（1）中，子后缀树调入内存的先后顺序是通过遍历之前介绍的哈希表来实现的。容易看出，该算法的时间复杂度与其输出的结果是有着直接的关系的[27,28]。假设存在模式 P 其长度为 n，下面对该算法的时间复杂度进行简单的分析。

如果输出结果仅仅判断是否匹配成功，那么该算法的时间复杂度为$O(n)$。这是因为整个过程一共匹配了 n 次。如果输出结果是计数或者枚举，那么该算法的时间复杂度为$O(n+k)$，其中 k 为子树的叶子结点的个数。不管是计数还是枚举它们所耗费的时间均为$O(k)$，这是因为对于计数其结果就是叶子结点的数量，对于枚举其结果就是叶子结点相应的后缀编号。

## 3.5 本章小结

本章详细介绍了后缀树索引结构的 LM-Suffix 构造算法。LM-Suffix 算法通过将一个长字符串划分为若干个短字符串为解决传统的后缀树构造算法不支持长字符串的问题奠定了基础；具体到子后缀树的构造，通过对划分后的短字符串按照字母表顺序进行排序来尽量消除放弃后缀链带来的不利影响。最后介绍了一种适合于 LM-Suffix 构造算法的搜索技术。

# 第4章 实验验证及分析

本章主要从索引结构的建立和索引结构的搜索两方面进行实验验证，然后根据实验结果进行性能分析和比较。

## 4.1 实验数据及实验环境

### 4.1.1 实验数据

本章节的实验都是以人类基因序列作为实验数据。字母表∑={A，C，G，T}，其中 A、C、G、T 四个字母为人类基因序列中的四个碱基，同时定义一个结束符$。本次实验所使用到的数据均来自于 Entrez。Entrez 是由 NCBI（美国国家生物技术信息中心）所主持的一个数据库系统。Entrez 将 GenBank 序列与其原始出处链接在一起，从而方便研究者使用[29]。

在进行实验之前，首先从该 URL——ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/上下载编号分别为 CHR_01、CHR_21 和 CHR_22 的人类基因序列数据集，其总长度为 1.4Gbp。编号 CHR_01 表示位于人类的第 1 号染色体上的基因序列；编号 CHR_21 表示位于人类的第 21 号染色体上的基因序列；编号 CHR_22 表示位于人类的第 22 号染色体上的基因序列。

### 4.1.2 实验环境

本章节的实验所使用的计算机的硬件环境如表 4.1 所示，软件环境如表 4.2 所示。

**表 4.1 硬件环境**

| 硬件 | 配置 |
|---|---|
| CPU | 1.2GHZ |
| 内存 | 2GB |
| 硬盘 | 120GB |

**表 4.2 软件环境**

| | |
|---|---|
| 操作系统 | Windows7 |
| 编程语言 | Java 语言 |
| JDK | 1.6u6 |

## 4.2 实验验证

### 4.2.1 索引建立

本次实验的目的是分析验证对于不同数据规模的人类基因序列数据集，经典的 Ukkonen 构造算法和本文提出的 LM-Suffix 构造算法在构造后缀树的过程中所展现出的不同的性能状况。

为了保证实验结果的准确性限定计算机的内存为 2GB。本次实验所选取的数据均来自于编号为 CHR_21 和 CHR_22 的人类基因序列，其长度分别为 20Mbp、40Mbp、60Mbp、80Mbp、100Mbp、120Mbp、160Mbp 和 200Mbp。对上述八个人类基因序列数据集分别采用经典的 Ukkonen 构造算法和本文提出的 LM-Suffix 构造算法来构造后缀树。本次实验的验证结果如图 4.1 所示。

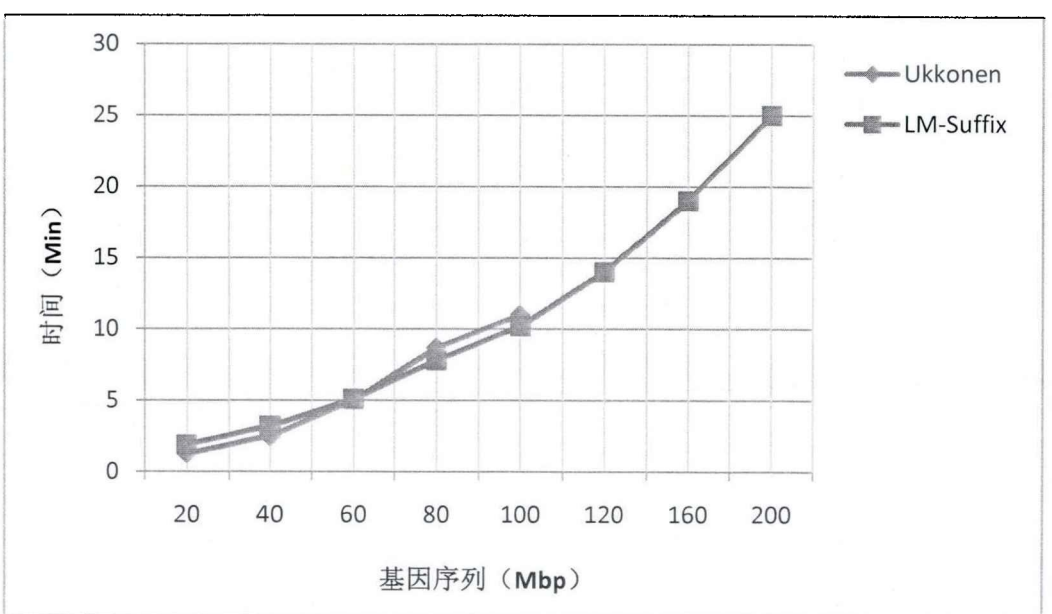


**图 4.1 构造后缀树的时间与基因序列长度的关系**

下面针对本次实验的结果进行理论分析，从图 4.1 可以得出以下四个结论。

（1）在内存不变的条件下，经典的 Ukkonen 构造算法和本文提出的

LM-Suffix 构造算法在构造后缀树的过程中所耗费的时间都随着人类基因序列的增长而增加。

（2）当人类基因序列数据集的长度小于 60Mbp 的时候，经典的 Ukkonen 构造算法所耗费的时间要比本文提出的 LM-Suffix 构造算法所耗费的时间少。

（3）当人类基因序列数据集的长度大于 60Mbp 且小于 100Mbp 的时候，经典的 Ukkonen 构造算法所耗费的时间要比本文提出的 LM-Suffix 构造算法所耗费的时间多。

（4）当人类基因序列数据集的长度大于 100Mbp 的时候，经典的 Ukkonen 构造算法已经无能为力了，这是因为产生了“内存瓶颈”问题，而本文提出的 LM-Suffix 构造算法依然可以处理基因序列数据集大于 100Mbp 的情况。

首先对结论（2）进行分析，当人类基因序列数据集的长度小于 60Mbp 的时候，经典的 Ukkonen 构造算法的性能优于 LM-Suffix 构造算法是因为经典的 Ukkonen 算法引入了后缀链的概念。后缀链可以减少定位次数从而将构造后缀树的时间降为线性。LM-Suffix 构造算法需要先对基因序列进行划分，然后再进行后缀树的构造。这在基因序列数据集不大的情况下反而增加了计算开销，因此效率不如经典的 Ukkonen 算法。

下面对结论（3）进行分析，当人类基因序列数据集的长度大于 60Mbp 且小于 100Mbp 的时候，经典的 Ukkonen 构造算法的性能劣于 LM-Suffix 构造算法是因为对于经典的 Ukkonen 构造算法而言，系统提供的可用内存空间已经不足，这时 Ukkonen 构造算法已经开始占用虚拟内存来进行后缀树的构造，同时后缀链的弊端开始逐渐的显现出来，由于计算机都采用多级存储体系结构，因此在对内存进行随机访问时不可避免的会产生命中率低的情况，因此其性能开始降低。反观 LM-Suffix 算法，由于该算法放弃使用后缀链，因此其优势就开始逐渐的显现出来了。

最后对结论（4）进行分析，当人类基因序列数据集的长度大于 100Mbp 的时候，对于经典的 Ukkonen 构造算法而言，即使继续使用虚拟内存也不可能完成后缀树的构造，至此经典的 Ukkonen 构造算法已经产生了“内存瓶颈”问题。但是对于 LM-Suffix 构造算法而言，并没有受到内存空间的限制，这是因为 LM-Suffix 构造算法先对基因序列进行划分使之满足内存的需求，然后再对划分后的基因序

列构造后缀树。

### 4.2.2 搜索方法

本次实验的目的是分析验证 LM-Suffix 算法构造出的后缀树的搜索性能。为了保证实验结果的准确性，本次实验选取的文本来自于编号 CHR_21 的人类基因序列，其长度为 60Mbp，选取的模式来自于编号 CHR_01 的人类基因序列，其长度分别为 2Mbp、3Mbp、4Mbp 和 5Mbp。采用 3.4 节中介绍的搜索方法对 LM-Suffix 算法构造的后缀树进行搜索。采用文献[11]中介绍的 LS 算法进行后缀数组的构造，然后再对其进行搜索。本次实验的验证结果如图 4.2 所示。

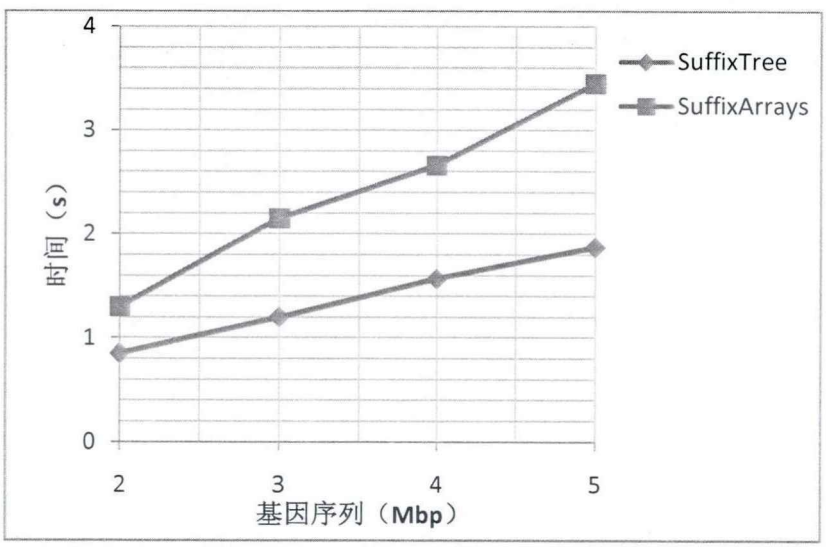


**图 4.2 搜索时间与基因序列长度的关系**

为了保证实验结果的准确性，本次实验限定了文本是不可变的，因此只需要考虑模式对搜索性能的影响。图 4.2 的结果反映出，在文本一定的条件下，虽然搜索所耗费的时间都随着人类基因序列（即模式）的增长而增加，但是 LM-Suffix 算法构造出的后缀树的搜索性能要明显优于后缀数组索引结构的搜索性能。

## 4.3 本章小结

本章从实验的角度对索引结构的建立和索引结构的搜索进行了验证分析。在建立索引的实验中，分别使用 LM-Suffix 算法和 Ukkonen 算法对不同大小的基因序列建立了索引。实验结果反映出，当为长基因序列建立索引时，LM-Suffix 算法具有更好的性能。在验证搜索性能的实验中，LM-Suffix 算法构造的后缀树保持了该结构一贯高效的特点。本章的实验结果对 LM-Suffix 算法的进一步发展起

到了很好的推进作用。

# 第5章 总结与展望

## 5.1 工作总结

基因序列索引技术的发展对生物信息学的发展起到了巨大的推进作用。可惜的是现存的各种索引技术都有一定的不足。比如，最常见的基于后缀树的索引结构在建立索引时容易产生“内存瓶颈”问题且不支持长字符串的处理；基于后缀数组的索引结构在处理速度方面效率较低；基于 q-gram 和 q-sample 的索引结构对于相似度低的基因序列却无能为力。

本文通过对基因序列索引结构的构造技术进行深入的研究分析，提出了一种基于后缀树索引结构的 LM-Suffix 算法。该算法首先对长基因序列进行划分使之成为若干个短基因序列，然后根据内存可用空间的大小来确定调入内存的短基因序列的个数。通过这样的划分为解决传统的后缀树构造算法不支持长字符串的问题奠定了基础。紧接着对调入内存的短基因序列按照字母表顺序进行排序，然后根据两两相邻的短基因序列的最大公共前缀来确定分支点，最后根据分支结点来构造后缀树。本文提出的 LM-Suffix 构造算法不仅对划分后的若干后缀进行了必要的合并，而且还实现了内存的动态调整。正是由于这两点才使得 LM-Suffix 算法在保证处理效率的同时大大减少了 I/O 开销。由于本文提出的 LM-Suffix 构造算法需要将长基因序列划分为若干个短基因序列，然后再对这些短基因序列进行子后缀树的构造，因此传统的后缀树搜索技术并不适用该情况。本文结合 LM-Suffix 算法的实际情况，在传统搜索方案的基础上设计了一种适合于 LM-Suffix 算法的搜索方案。本文提出的搜索方案的主要改进在于使用一个哈希表来记录子后缀树被访问的次数，通过遍历这个哈希表将经常被访问的子后缀树优先调入内存进行搜索，如此执行下去，直到匹配成功为止。

最后通过实验来验证本文提出的 LM-Suffix 构造算法与经典的 Ukkoenen 算法在性能上的优劣。本次实验结果反映出，LM-Suffix 算法在处理短基因序列时的效率要比经典的 Ukkonen 算法差些，这是因为 LM-Suffix 算法对短基因序列也进行了划分，但是 LM-Suffix 算法对于长基因序列依然有效，而经典的 Ukkonen 算法则产生了“内存瓶颈”问题。

## 5.2 工作展望

由于种种因素的限制，本文提出的 LM-Suffix 算法还存在一些不足之处。

（1）由于该算法实现了内存的动态调整，因此其必然使用了内存的动态申请和释放技术。当为大的基因序列建立索引时不断的申请、释放内存空间会在无形中大大的增加内存的开销，从而影响算法的性能。

（2）在确定分支结点的过程中，每次读入内存的待处理的字符的长度（即 Array R 的大小）对该算法的性能有直接的影响，这将是下一步工作需要重点解决的问题。

（3）对该算法构造出的后缀树进行搜索时，由于该算法对后缀树进行了划分，因此需要确定搜索子后缀树的先后顺序，在这个过程中会存在命中率低的情况。也就是说直至最后才找到那颗子后缀树。

对于上述不足需要做进一步的研究、分析，从中找出改进方案。

# 参考文献

[1]夏其昌，曾嵘.蛋白质化学与蛋白质组学.北京:科学出版社,2004.

[2] 谢惠民,基因序列分析中的若干数学方法,高校应用数学学报 A 辑(中文版) 2005 年 04 期.

[3] 靳德明，彭卫东，史红梅. 现代生物学基础. 北京: 高等教育出版社, 2000.

[4] 阎隆飞. 分子生物学. 北京: 中国农业大学出版社, 1997.

[5] 赵国屏. 生物信息学. 北京: 科学出版社, 2002.

[6]D.A.Benson, M.S.Boguski, D.J.Lipman,etal.GenBankNuclAeidsRes.25,1997.1-6.

[7] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, et al. Genbank update Nucleic Acids Res. Vol 32, Database issue, 2004. D23-D26.

[8] E. Hunt, M. P. Atkinson, and R. W. Irving. Batabase indexing for large DNA and protein sequene collections. The VLDB journal, 11:256-271,2002.

[9] R.Grossi andGFItaliano. SuffixTrees and Their Applications in String Algorithms. In Proc. 1st South AmericanWorkshop on String Processing (WSP 1993),pages 57--76, September 1993.

[10] McCreight EM. A space-economical suffix tree construction algorithm. Jouranl of the ACM. 1976.

[11] U. Manber and E. W. Myers. Suffix Arrays: A new method for on-line string searches.SIAM Journal on Computing, 1993,22(5), Oct. 935-948.

[12] N.Holsti and E.Sutinen. Approximate String Matching Using Q-gram Places. In proc. 7th Finnish Symp. On Computer Science, Pages 23-32. Univ. of Joensuu, 1994.

[13] C.Charras and T. Lecroq. Handbook of Exact String Matching Algorithms. King's College London Publications, 2004.

[14] R.S.Boyer,J.S.Moore. A fast string searching algorithm. Communication of the ACM, 1977, 20(10):763—772.

[15] A.Hume, D.M.Sunday.Fast string searching. Software Practice & Experience, 1991, 21(11): 1221—1248.

[16] E.Myers.ASublinear Algorithm for Approximate Keyword Searching. Algorithmica, 12(4/5):345-374, 1994. Earlier version in Tech. report TR-90-25, Dept. of CS, Univ. of Arizona, 1990.

[17] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 1995.

[18] E. Ukkonen. Approximate string matching over suffix tree. In: CPM93, Lecture Notes in Computer Science, vol. 684. Springer, Berlin Heidelberg New York, 1993.

[19] J. Tarhio, E. Ukkonen. Boyer-Moore approach to approximate string matching. In: J. R. Gilbert, R. Karlsson Proc. Springer, Berlin Heidelberg New York, 1990.

[20] E. Ukkonen. Approximate string matching with q-grams and maximal matches. Theoret Comput Sci 92:191-212,1992.

[21]R.Giegerich,S.Kurtz,andJ.Stoye.EfficientImplementationofLazySuffix Trees.InProceedingoftheThirdWorkshoponAlgorithmEngineering(WAE'99), 1999.

[22] Dr A.L.Brown. Constructing Chromosome Scale Suffix Trees. In the 2nd Asia-Pacific Bioinformatics Conference. 2004.

[23] E. Hunt, M. P. Atkinson, and R. W. Irving. Batabase indexing for large DNA and protein sequene collections. The VLDB journal, 11:256-271,2002.

[24]Manisealeo,M.A.andPuglisi,5.J.Aneffieient,versatileapProaehtosuffixsorting.ACMJ .ExP.Algor.12,Artiele1.2 2007.

[25] G. Myers A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM. 1999, 46. 395-415.

[26] EskoUkkonen.Approximate String-Matching over Suffix trees. Proc. CPM93, Lecture Notes in Computer Science 684, Springer 1993, 228—242.

[27] Ela Hunt, Robert W. Irving, Malcolm Atkinson. Persistent Suffix Trees and Suffix BinarySearch Trees asDNA Sequence Indexes. Technical Report TR-2000-63,Oct 4,2000.

[28] S. F. Altschul, T. L. Madden, A. A. Schaeffer, J. Zhang, W. Miller, D. J. Lipman. Gapped BLAST and PSI-BLAST:A New Generation of Protein Database Search programs. Nucleic Acids Res 25,1997.

[29] S Altschul, W Gish, W Miller, et al. Basic local alignment search tool. Journal of Molecular Biology, 1990, 215: 403-410.

# 致谢

时光如白驹过隙，三年的研究生生活就要结束了。我顺利地修完学业，并完成毕业论文的写作。一路走来，颇多体会，我谨向所有关心、支持、帮助我的师长，亲友呈上我最诚挚的感谢与最美好的祝愿。

在此衷心地感谢我的导师姚绍文教授。正所谓："师者，所以传道授业解惑也"，在云南大学求学的三年里，正是在您的悉心指导下，我掌握了分析问题解决问题的研究方法，也是您的谆谆教诲让我养成了勤奋务实，不断进取的学术作风，是您给我指明了方向，是您给我前进的动力。您严谨的治学态度，深厚的专业知识、正直的为人作风和认真负责的敬业精神令我钦佩不已，是我以后工作、生活中学习的楷模，您传授给我的不仅仅是鱼，还有"渔"。在此，深深感谢姚绍文老师三年来的教导与关怀。

本论文是在导师周维副教授的亲切关怀和悉心指导下完成的。从论文的选题、研究、撰稿到成稿，周老师始终给予我细心的指导和不懈的支持。三年的研究生生活，无论是在学习上，还是在生活上，周老师都给予了我无微不至的指导和关怀。他是我真正意义上的良师益友。

我还要感谢实验室这个温暖的、蓬勃向上的集体，感谢薛岗老师、刘璟老师、张云春老师对我无私的帮助，是你们的包容和鼓励让我的研究生生活充实而有意义。感谢李芬老师，是你的默默付出让每一个加入实验室这个大家庭的同学倍感温暖。

感谢软件学院的每一位老师，你们给予我的是我一辈子受用不完的宝贵财富。研究生的三年时光，除了感谢老师们的教诲之外，还要感谢我的同学、实验室的师兄师姐以及学弟学妹们，我们在融洽的气氛中共同学习、共同进步，彼此结下了深厚的友谊。

感谢我的父母，在我的求学路上，他们给予我无条件的支持和关心，使我能够全身心的投入到学习和科研中去。

"书山有路勤为径，学海无涯苦作舟"，2011 年的金秋，我的研究生生活从九五阶梯开始，一步一步登上求知的殿堂。"会泽百家，致公天下"。如今已是 2014 年的初夏，我贡献社会，报效祖国的凌云壮志也将从云南大学起步。

最后，衷心的感谢为评审本次论文而付出辛勤劳动的教授和专家们。