基于外存后缀树的 top-k 局部比对算法

王 斌 朱 睿 杨晓春 王国仁 于 戈
(东北大学信息科学与工程学院 沈阳 110004)

摘 要 局部比对是一种衡量字符串间相似程度的技术,它在生物信息学领域具有十分重要的作用.介于此,许多学者已对其进行了深入的研究.然而,随着数据规模的扩大,常规的内存算法已不适用于支持大规模文本数据的局部比对.为解决上述问题,该文研究了基于外存后缀树的 top-k 局部比对算法.它从根本上消除了内存空间对算法的束缚.为了提高算法的性能,该文首先将经典内存算法中的过滤策略引入该文.通过适当的修改,这些策略可以基于外存后缀树有效地降低计算开销.其次,该文提出一种巧妙的算法支持 top-k 局部比对查询.该算法通过引入启发式策略有效规避了 TA 算法的固有问题.具体地,它一方面可以提高算法的过滤能力,另一方面可以降低候选对象的维护代价.再次,该文对外存后缀树和磁盘的工作原理进行了研究.基于此,该文提出一种槽的结构支持查询.该结构既可以实现磁盘的顺序访问,又可以降低磁盘的访问次数.因此,它可以有效提高算法的查询效率.最后,大量的实验验证了该文所提出算法的有效性.

关键词 局部比对;top-k;外存后缀树;叉子区域
中图法分类号 TP301 **DOI 号** 10.11897/SP.J.1016.2016.02061

Top-k Local Alignment Algorithm Over External Suffix Tree

WANG Bin ZHU Rui YANG Xiao-Chun WANG Guo-Ren YU Ge
(*College of Information Science & Engineering, Northeastern University, Shenyang* 110004)

Abstract Local alignment is a common technique for finding a pair of highly similar substrings from two given sequences, which is very important in the biological information field. With the enlargement of data scale, the state of arts memory-based algorithms are not suitable for answering local alignment when handling long text data. In this paper, we study the problem of local alignment top-k query over external suffix tree. It could break the bottleneck limited by the memory space. In order to avoid unnecessary computing cost, we firstly employ a series of filtering strategies based on the classic memory-based algorithms. Via property amending them, these algorithms could effectively enhance the performance of our solution. We then propose a novel algorithm for answering top-k query local alignment over external suffix tree. It empolies the heuristic strategy for avoiding the defect of TA-algorithm. For one thing, it could provide a powerful threshold for filtering. For another, it could efficiently reduce the candidates maintainance cost. Then, we deeply study the operational principle of external suffix tree and disk. As the basis, we propose several techniques for optimizing external memory accessing. The results of the experiments on the real genetic data demonstrate the effectiveness of our algorithms.

Keywords local alignment; top-k; external suffix tree; fork area

收稿日期:2015-05-25;在线出版日期:2015-11-28.本课题得到国家"九七三"重点基础研究发展规划项目基金(2012CB316201)、国家自然科学基金(61572122,61173031,61129002,61532021,U1401256)、国家优秀青年科学基金(61322208)资助.**王 斌**,男,1972 年生,副教授,主要研究方向为大数据管理.E-mail:binwang@mail.neu.edu.cn.**朱 睿**,男,1982 年生,博士研究生,主要研究方向为传感器数据管理和不确定数据管理.**杨晓春**(通信作者),女,1973 年生,博士,教授,博士生导师,主要研究领域为字符串数据管理与并行计算.E-mail:yangxc@mail.neu.edu.cn.**王国仁**,男,1966 年生,博士,教授,博士生导师,主要研究领域为数据库、大数据管理.**于 戈**,男,1962 年生,博士,教授,博士生导师,主要研究领域为数据库、大数据管理.

1 引 言

局部比对是在两条文本串之间寻找近似子串的过程.它在生物信息学领域具有重要应用[1-2].具体来说,它有助于研究人员分析不同基因片段的功能,确定它们在遗传学上的意义[3-4].和常规文本数据相比,基因数据的规模较大.常规 PC 机的内存往往无法容纳这些数据[5-8],更无法容纳以这些字符串为基础构建的索引①.因此,在对基因数据执行局部比对时,传统的内存算法往往无法在单机环境下使用.针对内存不足等问题,分布式算法[9]和外存算法是两种流行的解决方案.对于前者而言,由于它对硬件的要求较高且不易维护,当实时性不是算法的第一考虑要素时②,它未必适用.相反,由于外存算法对硬件的要求较低且易于维护,它适合在生物序列比对技术中使用.因此,研究局部比对的外存算法具有十分重要的现实意义.

计算机的外存访问主要由两部分组成:磁道寻址和数据读写.由于外存的机械特性,前者的代价远远大于后者.换句话说,磁盘访问的高昂代价主要由磁道寻址引起.显然,如果外存算法能对磁盘执行顺序读取,它的访问代价就会急剧下降.研究结果表明,当磁盘执行顺序访问时,它的读取效率甚至高于内存的随机读取效率[9].由此可见,外存算法的研究焦点大多集中在:(1)减少磁盘访问次数;(2)尽可能地执行顺序访问.如果上述两点能够满足,外存算法不仅可以消除数据规模对算法的约束,而且可以打破 I/O 操作带来的瓶颈.

现如今,许多问题的外存算法都已被深入研究.在文本管理领域,外存后缀树作为一种高效索引结构,十分适用于管理大规模文本数据.其主要原因是后缀树是可分解的.这样一来,算法可以自适应加载不同子树.因此,基于外存后缀树实现字符串间的局部比对是一种可行的方法.

本文基于外存后缀树研究字符串间的 top-k 局部比对问题.给定原串 T,查询串 P 和评价字符串间相似度的打分函数 F,该问题要求返回 T,P 之间相似程度最高且彼此之间不存在支配关系的 k 对子串.

为了达到此目的,当查询串 P 到来时,查询算法依次从磁盘中读取后缀树 $\mathcal{I}$③的各子树 $\mathcal{I}_s$、深度优先遍历 $\mathcal{I}_s$、根据 $\mathcal{I}_s$ 维护的信息从内存和磁盘中分别找到参与局部比对的子串、计算比对结果,维护候选集中分值最高且不被支配的对子串.当遍历完成时,算法将候选集作为查询结果输出.为了提高算法的效率,本文需要面对以下挑战:

(1)找到适用于外存环境的过滤策略.近年来,基于内存的局部比对算法已被深入研究.然而,它们的过滤策略都是在内存环境下实现的.开发适用于外存环境的过滤算法是第 1 个需要客服的难点.

(2)选取合适的阈值.和基于阈值的局部比对算法相比,基于 top-k 的局部比对算法无法提供过滤能力强的阈值,这会导致算法的运算速度降低.因此,本文面临的第 2 个挑战是找到过滤能力强且保证不丢解的阈值从而提高算法性能.

(3)减少 I/O 代价.由文献[10-12]可知,由于局部比对算法需要频繁对原串 T 执行随机访问,它会导致算法多次访问磁盘.因此,本文面临的第 3 个挑战是找到迎合磁盘工作原理的算法从而降低 I/O 代价.

为克服上述挑战,本文贡献如下:

(1)高效的过滤策略.本文首先针对外存后缀树的特点进行分析.基于此,本文结合文献[8]中的过滤策略(例如:长度过滤、支配关系过滤等)加速局部比对计算.实验表明,这些过滤策略不仅降低了局部比对计算次数而且适合在外存环境下使用.

(2)高效的候选集维护算法.本文首先利用启发式策略找到合适的阈值.该阈值保证在不丢解的前提下提供较强的过滤能力,从而降低候选集的维护次数.当候选集需要被维护时,本文近一步利用散列表维护候选对象间的支配关系.它可以在常数时间内找到并移除候选集中的非候选对象.

(3)高效的外存算法.一方面,本文根据外存后缀树的工作原理挑选访问频率高的子串常驻内存.另一方面,本文提出一种批处理算法实现磁盘的顺序读取.显然,上述两种策略一方面降低了磁盘的访问次数,另一方面减少了随机寻址次数.它们均可以有效降低 I/O 代价.

本文第 2 节概述背景知识;第 3 节介绍基于外存后缀树的 top-k 局部比对算法;第 4 节进行有效性实验分析;第 5 节对全文进行总结.

---

① 以后缀树为例,给定字符串 T,如果根据 T 构建后缀树,后缀树的空间代价是 $|T|$ 的 24 倍.

② 在生物信息学领域,研究人员首要关注在不同基因串之间发现可用信息而不是如何提高算法效率.

③ 原串 T 可看成是已知串,研究人员通常将未知串与它进行比较从而寻找相似子串.因此,研究人员通常为建立索引以便加快查询.

2　准备工作

在介绍问题定义之前，表 1 首先定义一组符号.

表 1　符号列表

符号	名称
Σ	字符集合
P	查询串
T	被查询的字符串(也称原串)
$T[i]$(或 $P[i]$)	T(或 P)的第 i 个字符
$T[i,j]$	T 中位置 i 到位置 j 组成的子串
$P[i,j]$	P 中位置 i 到位置 j 组成的子串
s_a	匹配操作
s_b	替换操作
s_g	Gap 操作
s_s	Gap 扩展
ALC(t,p)	子串 $t\in T,p\in P$ 之间的最优局部比对
$\lvert$ALC$(t,p)\rvert$	$t\in T,p\in P$ 之间的最优比对分值
ALC(i_s,j_s,i_e,j_e)	基于 $T[i_s,i_e],P[j_s,j_e]$得到的局部比对
$s(a\rightarrow b)$	后缀树 $\mathcal{I}$ 中两节点 a,b 边上索引的字符串

2.1　背景知识

2.1.1　相似度评价函数

给定字符串 r 和 s，$F(r,s,\mathcal{O},\mathcal{S})$表示评价 r,s 之间相似程度的打分函数. 其中，$\mathcal{O}$表示将 r 转换为 s 所需的操作集合；$\mathcal{S}$表示这些操作对应的分值集合.

通常情况下，$\mathcal{O}$包含 4 种基本操作. 它们分别是匹配(简称 s_a)，替换(简称 s_b)，插入和删除. 给定字符串 r 和 s，如果 $r[i]=s[i]$，则称两者在位置 i 上匹配. 在一系列匹配或替换操作之后，本文将第 1 个插入或删除操作称为 Gap 开启(简称 s_g). 在 Gap 开启位置及其之后的连续插入或者删除操作称为 Gap 扩展(简称 s_s).

图 1 解释了字符串间相似度的计算方法. 其中，实线表示匹配操作；虚线表示替换操作；其余部分表示 Gap 开启或 Gap 扩展开启. 图 1 共有 10 次匹配、4 次替换、2 次 Gap 开启和 4 次 Gap 扩展开启. 假设匹配操作对应的分值为 1，其余操作对应的分值都为 -1，r 和 s 的相似度为 $10\times1+4\times(-1)+2\times(-1)+4\times(-1)=0$.

图 1　字符串间相似程度的评价方法

2.1.2　外存后缀树

给定字符串 T，它的后缀树$\mathcal{I}$是一棵索引 T 中所有后缀的树状结构[11-12]. 图 2(a)为字符串 $T=$“AGCCAGAT”对应的后缀树$\mathcal{I}$. 给定 T 中以任意字符为起点的后缀 S_{sur}，它都可以从根节点到某叶子节点组成的边上得到. 例如：字符串 T 包含两个以 G 为起始点的后缀串，它们分别是“GCCAGAT”和“GAT”.

为了降低后缀树的空间代价，Manber 等人[13-19]提出一种压缩算法. 它用字符串的概要信息对原串进行管理. 以节点 2 和 7 之间的边为例，在图 2(a)中，这条边上存储的字符串 $s=$“AGAT”. 为了对它进行压缩，在图 2(b)中，存储算法将边上的内容变为 A[4,7]. 它的含义为 s 的首字符为 A，它在原串中的起始和终止位置分别为 4 和 7.

由于后缀树的规模较大，许多学者对外存后缀树进行了研究. 主流算法通常把原串 T 的一部分加载到内存，而另一部分存放在磁盘上. 给定原串 T 和它对应的后缀树$\mathcal{I}$，如果$\mathcal{I}$中需要访问的子串保存在内存中，访问算法直接从内存读取子串. 否则，算法需要从磁盘读取原串的相应内容.

为方便叙述，本文用$\mathcal{M}$表示存储 T 子串的内存；$|\mathcal{M}|$表示$\mathcal{M}$空间的大小；$s(a\rightarrow b)$表示$\mathcal{I}$中 a,b 两节点边上对应的字符串. 图 2(b)所示，假设 $|\mathcal{M}|=4$、$\mathcal{M}$中初始的子串为“AGCC”. 在访问边 $s(5\rightarrow11)$时，因为它索引的子串不在内存中，所以这

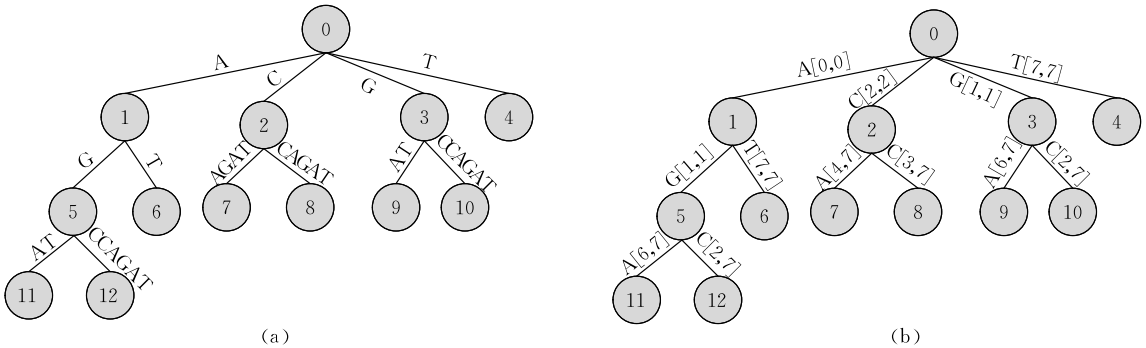


图 2　字符串 $T=$“AGCCAGAT”的后缀树

会引发一次磁盘操作.相似地,当算法完成遍历后,磁盘操作次数为 11.由上述观察可知,当 T 无法完全装载到内存时,遍历后缀树会频繁引发磁盘操作.其根本原因是后缀树相邻边上保存的子串在原串 T 中的位置往往较远.

2.2 问题定义

在讨论了背景知识后,本文介绍关于局部比对的相关概念.

定义 1(局部比对计算). 给定字符串 $r\in T$ 和 $s\in P$,局部比对计算(简称:ALC(Local Alignment Calculation))是利用评价函数 $F(r,s,\mathcal{O},\mathcal{S})$ 计算两者相似程度的过程.

定义 2(最优局部比对计算). 给定 $\mathcal{O}$ 和 $\mathcal{S}$,子串 $r\in T$ 和 $s\in P$ 之间的最优局部比对是使函数值 F 达到最大的局部比对操作序列.

为了方便描述,给定原串 T 和查询串 P,本文用 ALC(t,p) 表示子串 $t\in T$,$p\in P$ 之间的最优局部比对;$|\text{ALC}(t,p)|$ 表示最优比对结果对应的分值.

定义 3(局部比对的支配关系). 给定子串 p_1,$p_2\in P$,$t_1,t_2\in T$,如果它们满足:(1)$t_1[|t_1|-1]$,$t_2[|t_2|-1]$在 T 中的位置相同;(2)$p_1[|p_1|-1]$,$p_2[|p_2|-1]$在 P 中的位置相同;(3)$|\text{ALC}(t_1,p_1)|\geq|\text{ALC}(t_2,p_2)|$,本文称 ALC$(t_1,p_1)$ 支配 ALC(t_2,p_2)(记作:ALC$(t_2,p_2)\leq$ALC(t_1,p_1)).

问题定义 1(基于阈值的 ALC)(简称 T-ALC).给定原串 T、查询串 P 以及分数阈值 H,基于分数阈值的 ALC 返回 T,P 子串之间局部比对分值不小于 H 且彼此之间不存在支配关系的子串对.

问题定义 2(基于 top-k 的 ALC)(简称 K-ALC).给定原串 T、查询串 P 以及正整数 k,基于 top-k 的 ALC 返回 T,P 子串之间局部比对分数值最高且彼此之间不存在支配关系的 k 对子串.

2.3 相关工作

局部比对是一种衡量字符串间子串相似程度的方法[20-26].和全局比对相比,局部比对具有更重要的实际意义,因此它也受到学者的更多关注.

Smith-Waterman 算法[2]是基于分数阈值的局部比对算法.该算法的最大贡献是将局部比对问题转换为动态规划问题.具体地,给定原串 T 和查询串 P,该算法分别计算 T 和 P 中任意两个子串间的最优局部比对结果.由文献[2]可知,因为任意两个子串的最优局部比对结果依赖于它们子串间的最优局部比对结果,所以该问题的最优解依赖于更小规模子问题的最优解.因此,该问题可以使用动态规划算法进行解决.它的时间复杂度为 $\mathcal{O}(|T|\times|P|)$.显然,当文本数据的规模较大时,Smith-Waterman 算法的运行效率有待提高.

OASIS(Online and Accurate Search technique for Inferring local-alignments on Sequences)算法是 Meek 等人[27]于 2003 年提出的.与 Smith-Waterman 算法相比,它做出以下改进:(1)它为原串 T 建立后缀树从而避免了重复计算问题;(2)为每个搜索节点维护一个长度为 $\mathcal{O}(|P|)$ 分数向量从而加速了比对计算.它的缺点是只适合在查询串较短时使用.

GPU 算法是由 Salavert 等人[27]提出的.它通过使用 BWT(Burrows-Wheeler Transform)索引反向搜索原串中出现的子串.该方法可以模拟基于后缀树的局部比对计算过程.ALAE(Accelerating Local Alignment with Exactly Affine Gap)是 Yang 等人[8]提出的.它是以 GPU 算法为基础进行研究的.与之不同的是,该算法提出了多种过滤和复用策略以减少不必要的计算开销.

3 算法描述

本节讨论了基于外存后缀树的局部比对算法.首先,本文在 3.1 节提出了研究框架 ESF(Extend Suffix Framework).接下来,本文在 3.2 节提出算法 ESLA(Extend Suffix Local Alignment)解决基于外存后缀树的 T-ALC(基于阈值的 ALC)问题.以此为基础,本文在 3.3 节近一步研究了 top-k 局部比对算法.最后,本文在 3.4 节提出了一系列外存优化算法降低 I/O 开销.

3.1 基于外存后缀树的算法框架

本节提出框架 ESF(Extend Suffix Framework)支持基于外存后缀树的局部比对计算.本文使用后缀树作为文本索引的原因如下:(1)基于公共前缀的索引方式.后缀树可以将原串中不同的后缀按照其公共前缀进行索引.基于该性质,算法在执行 ALC 时,不同子串的公共前缀只需计算一次.因此,它可以减少冗余的计算量;(2)自适应于内存空间.虽然后缀树是一种全文索引结构,但是它可以被划分成许多子树.基于这个性质,ESF 可以根据内存容量自适应地选择某些子树读入内存.因此,后缀树适于在外存环境下支持字符串间的 ALC 操作.

如图 3 所示,ESF 主要由两部分组成.第 1 部分

是关于原串 T 和后缀树$\mathcal{I}$的维护策略. 如图所示，ESF 分别将后缀树$\mathcal{I}$的一个子树和原串 T 的多个子串放入内存(图中的黑色部分). 当查询串 P 到来时，ESF 根据查询类型和操作类型分别调用 ESLA 算法，ESLA^k(基于 top-k 的 ESLA)算法和 ESBR (Extend Suffix Batch Read)算法.

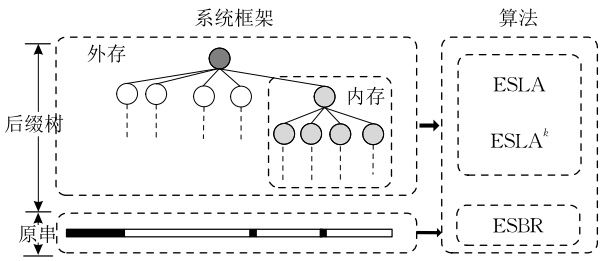


图 3　ESF 的结构示意

ESLA 算法可以在外存后缀树上解决 T-ALC 问题. 给定原串 T、查询串 P、后缀树$\mathcal{I}$和内存中的子树$\mathcal{I}_s$，ESLA 在遍历过程中根据$\mathcal{I}_s$索引的子串执行 ALC 操作并返回满足查询条件的结果. 需要强调的是：ESLA 克服了如下挑战提高计算效率：(1) 引入 ALAE 中的多种局部过滤策略，并使它们与后缀树有机结合；(2) 引入 ALAE 中提出的叉子区域. 以此为基础，ESLA 基于叉子区域的支配关系①进一步加强了算法的过滤能力.

以 ESLA 算法为基础，本文近一步提出 ESLA^k 算法解决 K-ALC 问题. 为了克服 TA(Threshold Algorithm)算法的固有问题和衍生问题②，本文引入启发式算法寻找合适的阈值. 基于该阈值，ESLA^k 算法可以快速地过滤中间结果. 此外，ESLA^k 提出一种巧妙的数据结构维护候选对象. 它的好处是可以高效维护候选结果间的支配关系，从而降低维护代价.

最后，ESF 提出算法 ESBR(Extend Suffix Batch Read)优化外存操作. 它是基于以下研究动机提出的：(1) 后缀树相邻边上保存的子串不满足局部性原理. 在很多情况下，它们在原串中的位置相距很远；(2) 后缀树边上只保存子串的概要信息且 ESF 只能在内存中保留 T 的一个子串，ESF 需要多次访问磁盘加载目标子串. 由于该操作代价较高，算法必须谨慎选择读入内存的原串从而避免频繁的 I/O 操作.

为了达到这个目标，算法根据后缀树的性质提出以下两个优化策略：(1) 选择原串中访问频率较高的子串常驻内存. 由大量观察可知：给定原串 T 和其对应的后缀树$\mathcal{I}$，子串的访问主要集中于 T 的前半部分. 因此，ESBR 尽可能将 T 的前部常驻内存；(2) 基于批处理的磁盘操作. 本文提出了一种延迟读取策略. 该策略的核心思想是缓存涉及 I/O 操作的子串. 当此类子串的数目到达上限时，算法根据这些子串在磁盘中的位置规划出最优的搜索顺序. 由前文可知，磁盘的顺序读写有着较高的效率. 本文的延迟读取策略恰好迎合了这一特性. 此外，由于排序后子串在原串中位置往往相近并且 ESBR 是以块③为单位读取数据，这些子串经常会出现在同一块中. 因此，这种批处理方法可以近一步地降低 I/O 访问次数.

3.2　计算局部比对的过滤策略

在外存环境下执行 T-ALC 时，一种简单的方法是使用 Smith-Waterman 算法. 它可以通过矩阵运算得到局部比对结果. 然而，该算法的时间代价较大. 为了提升效率，本文提出了 ESLA(Extend Suffix Local Alignment)算法加速计算. 一方面，该算法借助外存后缀树实现了 ALAE 算法中的几种局部过滤策略. 另一方面，ESLA 根据局部比对结果间的支配关系近一步加强了算法的过滤能力.

3.2.1　基于阈值的局部过滤策略

使用外存后缀树执行 ALC 操作时，ESLA 的整体思路是从外存中依次读取后缀子树，遍历这些子树，得到子树边上对应的子串，执行 ALC 操作. 对于任意后缀而言，因为它都可以通过一次深度优先遍历获得，所以提高 ESLA 性能的一个有效途径是缩短它在后缀树上的遍历范围.

本文首先通过引入 ALAE 中的几种局部过滤策略降低遍历代价. 它们包括长度过滤、前缀过滤和分数过滤. 本文首先讨论长度过滤.

引理 1.　给定查询串 P、原串 T、子串 $t\in T$ 和分数阈值 H，如果 t 是比对结果，它的长度需要满足不等式$\lceil H/s_a\rceil\leq\max\{m,n+\lfloor(H-s_a\times m+s_g)/s_s\rfloor\}$. 其中$|t|$为 t 的长度.

证明.　对于$|t|$的下限，因为分数阈值为 H，即使 t 中所有字符均与 P 中对应字符匹配，$|t|$也不能小于$\lceil H/s_a\rceil$. 对于$|t|$的上限，它的含义是 ALC 中有 m 个连续的匹配操作，其余的为 Gap 操作. 为了满足分数阈值 H 的要求，Gap 操作的次数需要控

① 叉子区域是 ALAE 中对局部比对计算区域的一种形象化描述. 其原因是局部比对的计算区域在利用了 ALAE 算法的过滤策略后形似于一个叉子.

② 固有问题是“小阈值问题”；衍生问题为“小阈值问题”引发的算法过滤能力差等问题.

③ 块是磁盘存储数据的基本单位. 它的默认 size 是 4096 字节.

制在$\lfloor(H-s_a\times m+s_g)/s_s\rfloor$以内.因此,$|t|$不能大于$\max\{m,m+\lfloor(H-s_a\times m+s_g)/s_s\rfloor\}$.

根据引理1,长度过滤可以减少一些不必要的局部比对计算.以此为基础,本文提出定理1.它帮助我们缩小深度优先遍历的范围.为了方便说明,给定根节点r、非叶子节点e_{in},本文用$|s(r\rightarrow e_{in})|$表示$s(r\rightarrow e_{in})$上维护的子串长度之和;用e表示$s(r\rightarrow e_{in})$上子串在原串中索引位置的终点值;用L_{min}表示长度过滤的下限值.

定理1. 给定后缀树$\mathcal{I}$和它的内节点e_{in},如果e_{in}满足不等式:$L_{min}\leq|T|-e+|s(r\rightarrow e_{in})|$,则它的子孙在深度优先遍历时可以被剪枝.

证明. 给定原串T和它对应的后缀树$\mathcal{I}$,$\mathcal{I}$边上索引的子串是它在T中出现最早的位置.给定e_{in}的父节点p_{in}和它们之间的边$s(p_{in}\rightarrow e_{in})$,如果$s(p_{in}\rightarrow e_{in})$上存储子串的终止位置为$e$,则从$e_{in}$到其任意子孙对应边上索引的子串长度不会超过$|T|-e$.根据长度过滤,用来比对的$T$的子串长度必须不小于$L_{min}$,假设从根节点与某内节点之间边上的子串长度为$|s(r\rightarrow e_{in})|$,则仍然需要$L_{min}-|s(r\rightarrow e_{in})|$个字符才符合要求,也就是说如果内节点不满足关系$|T|-e\geq L_{min}-|s(r\rightarrow e_{in})|$,继续访问其子孙节点是没有意义的. 证毕.

利用定理1,本文可以在后缀树上实现长度过滤.除此之外,本文使用分数过滤加速比对计算.具体地,给定子串$t\in T$,$p\in P$,如果$ALC(t[0,i],p[0,j])\leq 0$,计算$|ALC(t,p)|\leq 0$是无效的[8].由于这种过滤策略不依赖于后缀树,本文不再叙述它的使用方法.

接下来,本文讨论前缀过滤.给定子串$t\in T$和$p\in P$,前缀过滤的核心思想是根据t和p的前q个字符进行过滤.具体地,如果$T[0,q-1]\neq P[0,q-1]$,则$ALC(t,p)$可以提前结束.其中,$q=\lfloor\min\{|s_b|,|s_g+s_s|\}/s_a\rfloor+1$;$s_a$表示匹配操作;$s_b$表示替换操作.为了实现前缀过滤,本文为查询串$P$建立了$q$长子串的倒排索引$\mathcal{H}$.根据$\mathcal{H}$,ESLA可以定位$T$中子串的$q$长前缀在$P$中的起始位置.显然,ESLA从这些位置开始计算就可以得到比对结果.

综上所述,上述3种局部过滤策略易于在后缀树上实现.通过引入这些过滤策略,ESLA可以有效降低计算代价.接下来,本文介绍基于支配关系的过滤技术.

3.2.2 基于支配关系的全局过滤策略

由问题定义2.1可知,局部比对问题不仅要求比对结果的分值不小于阈值H,而且要求比对结果之间不存在支配关系.本节首先引入叉子区域的概念.其次,本文基于叉子区域提出基于支配关系的过滤算法.

叉子区域是根据前缀过滤和分数过滤形成的.给定局部比对$ALC(t,p)$,如果它不被过滤,它一定满足以下条件:(1)t,p之间的公共前缀长度大于$q-1$;(2)$ALC(t,p)\geq 0$.如图4所示,局部比对的计算区域在形状上可以看成是一个叉子[8].在这里,EMR代表准确匹配区域;NGR代表不存在Gap操作的区域.X表示一条子串.近一步,根据叉子区域,算法可以找到比对结果间的支配关系.为了便于描述,本文用叉子表示两子串间的局部比对计算区域.$ALC(i_s,j_s,i_e,j_e)$表示基于$T[i_s,i_e]$、$P[j_s,j_e]$得到的最优局部比对结果.

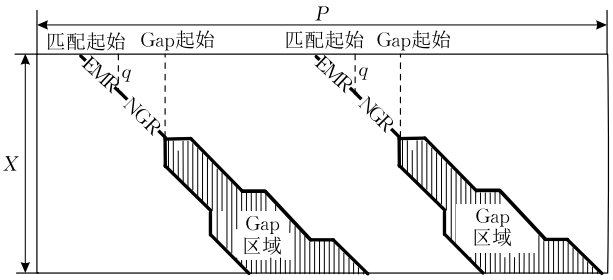


图4 叉子区域示意

定义4(叉子区域间的支配关系). 给定子串$T[i_s,i_e]\in T$,$P[i_s,i_e]\in P$和它们形成的叉子区域A,如果$ALC(i_s,j_s,i_e,j_e)>0$且$T[i_x,i_x+q-1]=P[j_y,j_y+q-1]$,则$A_x\leq A$,其中$i_x\in[i_s,i_e]$;$j_y\in[j_s,j_e]$;$A_x$表示起始点为$i_x$,$j_y$的叉子区域.

根据定义4可知,如果叉子区域A_x的起始位置包含于叉子区域A,则$A_x\leq A$.

实例1. 图5演示了两个叉子区域的支配关系.图中X和Y分别表示$T[i_s,i_e]$和$P[j_s,j_e]$.$T[i_s,i_e]$和$P[j_s,j_e]$的局部比对计算区域是叉子区域A,它的起始位置有q个匹配操作.在叉子区域A的位置(i_x,j_y)处,虽然$T[i_x,i_x+q-1]=P[j_y,j_y+q-1]$,ESLA可以在$(i_x,j_y)$处开辟新的叉子$A_x$,但$A_x$中参与局部比对的子串是$A$中参与局部比对的子串后缀.假设基于$A$和$A_x$得到的最终叉子区域分别为$A(i_s,j_s,i_e,j_e)$和$A_x(i_x,j_x,i_e,j_e)$,因为$A(i_s,j_s,i_e,j_e)=A(i_s,j_s,i_x,j_x)+A_x(i_x,j_x,i_e,j_e)$且$A(i_s,j_s,i_x,j_x)>0$,所以$A_x(i_x,j_x,i_e,j_e)<A(i_s,j_s,i_e,j_e)$.由此可知,计算$A_x$是没有意义的.

根据如上描述,如果ESLA在开启一个叉子区

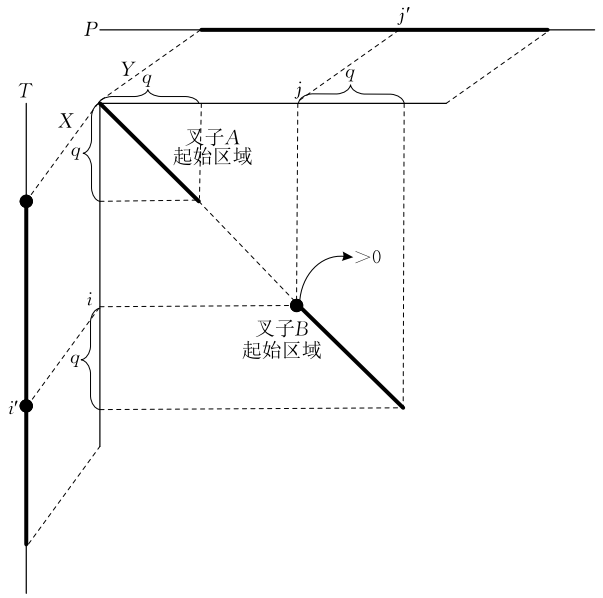


图 5　叉子区域的支配关系示意

域前判定它是否被支配，ESLA 可以减少很多不必要的计算. 由上文可知，判断一个叉子区域是否被支配的先决条件是判断其是否包括在另一个已计算的叉子区域中. 为了达到这个目的，ESLA 需要维护结果间的支配信息. 它的难点是随着结果数量的逐渐增多，ESLA 需要维护的叉子个数也会增多.

一个简单的策略是缓存所有的“q 匹配操作”. 它指的是在计算某个叉子区域的过程中，一旦碰到连续 q 个匹配操作，ESLA 就将匹配的起始位置记录下来. 当开始计算新的叉子区域时，ESLA 查询起始位置信息是否被包括. 如果是，此次 ALC 可以被过滤. 然而，该策略存在的问题是需要保存许多历史记录. 为了解决上述问题，本文首先提出 q-prefix 的概念. 随后，本文基于 q-prefix 的字典序提高计算效率. 具体地，给定 q-prefix q_1 和 q_2，如果 q_1 的字典序小于 q_2 的字典序，ESLA 优先计算与 q_1 相呼应的叉子区域.

定义 5(q-prefix).　给定叉子区域 A 和它们对应的子串 $T[i_s,i_e]\in T$，$P[i_s,i_e]\in P$，本文将 $T[i_s,i_e]$ 中前 q 个匹配操作称为 q-prefix.

具体地，在计算某个局部比对的过程中，当遇见连续 q 个匹配操作时(不包括起始位置的 q-prefix)，ESLA 比较该子串和其他 q-prefix 的字典序. 如果该子串具有更大的字典序，ESLA 则储存该位置信息. 基于这种方法，本文可以判定哪些叉子区域已被计算(或未被计算).

图 6 是一个 ALC 中遇到连续 q 次匹配操作的示意图. 根据图中的分数策略，$q=4$. 图中的 q-prefix 是字符串“ACGT”，在局部比对过程中，ESLA 遇到了其他 q-prefix“CAGT”和“AAAA”. 因为 C 的字典序大于 A，所以“CAGT”的字典序大于“ACGT”；ESLA 可以判定以“CAGT”开始的叉子区域未被计算. 因此，ESLA 将其对应在原串 T 和查询串 P 中的出现位置保存起来. 相反，因为“AAAA”的字典序小于“ACGT”，这意味着以“AAAA”开始的局部比对已被计算；再记录其出现位置是没有意义的. 因此，“AAAA”不被保存. 需要强调的是，出于空间效率考虑，ESLA 不保存 q-prefix 的位置信息. 因此，该策略可能会导致一些冗余计算.

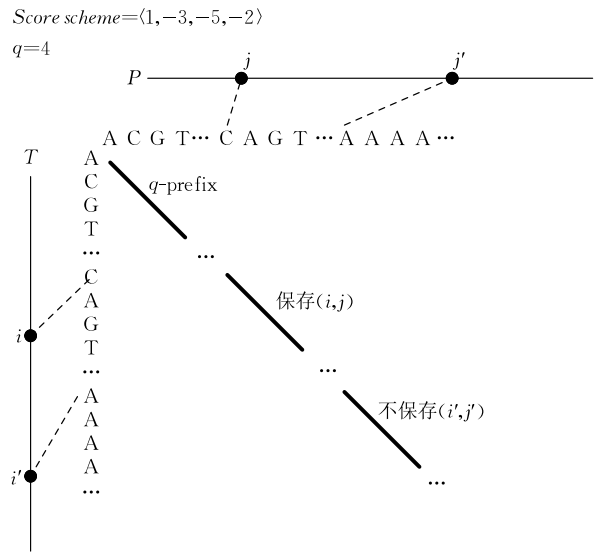


图 6　出现 q 次匹配的处理情况示意

当开始计算一个新的叉子区域时，ESLA 首先检查起始位置是否已经被保存. 如果是，ESLA 可以判定该叉子区域已被计算. 进而，ESLA 可以推断该区域已被支配. 在这种情况下，ESLA 过滤该叉子区域. 需要强调的是，那些保留的被支配的叉子区域的起始位置是按照其 q-prefix 进行组织的. 给定一个新的 q-prefix q_{new}，上述策略的好处是当开始计算以 q_{new} 为起始的叉子区域时，ESLA 可以释放字典序小于 q_{new} 的被支配叉子区域的起始位置，从而提升算法的存储空间利用率.

除此之外，本文在 3.3 节提出了一种巧妙的数据结构维护候选对象之间的支配关系. 这种算法也适于在本节使用. 最后本文对 ESLA 算法的时间复杂度进行分析.

这里，设数据构建在外存的索引大小为 I，内存块大小为 I_s，一次遍历外部索引的最坏代价是 $\mathcal{O}(I/I_s)$，每个块在内存中的计算代价可以借助文献[8]中定理 4 的分析结果，算法在内存中的计算代价是

$(k/k_2-1+k_1\sigma^2/\sigma-k_2)|P||T|^{\log k_2}$. 其中，$k_1=(1-1/s)(\sigma-1/\sigma-2)(s/\sqrt{2\prod(s-1)})$，$k_2=s\times(\sigma-1/(s-1)^{s-1})^{1/s}$，$s=1+|s_b|/|s_a|$，$\sigma$ 是字符集的个数. 从而，本文可以得出 ESLA 在整个数据集上时间复杂度等于$\mathcal{O}(I/I_s(k/k_2-1+k_1\sigma^2/\sigma-k_2)|P||T|^{\log k_2})$.

3.3 基于 top-k 的局部比对算法

在讨论了基于阈值的局部比对问题之后(T-ALC)，本节根据上节的研究结果讨论基于 top-k 的局部比对问题(K-ALC). 由问题定义可知，它返回分值最高且彼此间不存在支配关系的 k 个结果. 为了解决该问题，本文提出 ESLA^k(基于 top-k 的 ESLA 算法)算法.

该算法以 ESLA 中的过滤策略为基础，结合 TA 算法维护候选集合. 给定原串 T、查询串 P 和候选集$\mathcal{R}$，ESLA^k 首先将候选集初始化为空集. 接下来，ESLA^k 在访问 T 的过程中将满足查询条件的比对结果 r 插入$\mathcal{R}$. 其中，插入条件为：(1) $\min(\mathcal{R})<r$[①]；(2) r 不被$\mathcal{R}$中的其他元素支配. 如果 r 满足上述条件，ESLA^k 将 r 插入$\mathcal{R}$. 否则，ESLA^k 丢弃 r. 特殊地，在插入之后，如果$|\mathcal{R}|=k+1$，ESLA^{k}丢弃 $\min(\mathcal{R})$.

对于第一个插入条件，ESLA^k 面临的挑战是“小阈值”问题. 具体地，由于初始化状态下$\mathcal{R}=\varnothing$，算法只能将 $\min(\mathcal{R})$设置为 0. 因为此时的阈值较低，它会带来以下影响：(1) 前文讨论的过滤算法很难被使用；(2) 在算法运行初期，很多分值较小的比对结果也会加入候选集. 在大多数情况下，它们都会被后加入的比对结果代替. 显然，存储这样的对象是没有意义的. 总体来说，“小阈值”问题会消耗不必要的计算资源. 因此，提高 ESLA^k 性能的关键是找到合适的阈值 H 代替 $\min(\mathcal{R})$.

本文使用启发式算法初始化 H. 由前文可知，在评价函数 $F(P,T,\mathcal{O},\mathcal{S})$的操作集合$\mathcal{O}$中，只有匹配操作对 ALC 贡献正分数，也就是说只有通过匹配操作才能够提高 ALC 的分值. 根据上述观察，本文首先找到 k 个不相交且具有最长前缀的子串对. 接下来，本文基于这 k 个子串对执行比对计算. 近一步，本文用这些子串对间的比对结果初始化$\mathcal{R}$. 显然，此时的 $\min(\mathcal{R})$在初始化后具有较大的分值. 相应地，第 1 个问题可以被解决.

在初始化$\mathcal{R}$后，ESLA^k 使用 3.2 节中介绍的过滤策略加快局部比对. 然而，ESLA^k 需要面临第 2 个挑战：它需要额外判断新得到的比对结果是否被$\mathcal{R}$中元素支配. 其根本原因是$\mathcal{R}$中初始化结果分布在 T 中的各个位置，ESLA^k 无法像 ESLA 一样在扫描的过程中维护结果间的支配关系.

为了解决这个问题，ESLA^k 使用桶结构$\mathcal{B}$和哈希表$\mathcal{H}$维护$\mathcal{R}$中的元素. 图 7 演示了桶结构和哈希表的存储示例图. 图 7 的下半部分为哈希表$\mathcal{H}$. 表中每项元素 e 包含两部分内容. 第 1 部分是关键字，它存储某一局部比对结果的结束位置. 第 2 部分是二元组$\langle pt,F(e)\rangle$. 其中，pt 指向 e 在$\mathcal{B}$中的相应位置；$F(e)$表示 e 的局部比对分值. 图 7 的上半部分刻画了桶$\mathcal{B}$的结构. 它是按照局部比对的分值进行组织的. 其中，具有相同分值的局部比对结果会被存储在同一个桶中.

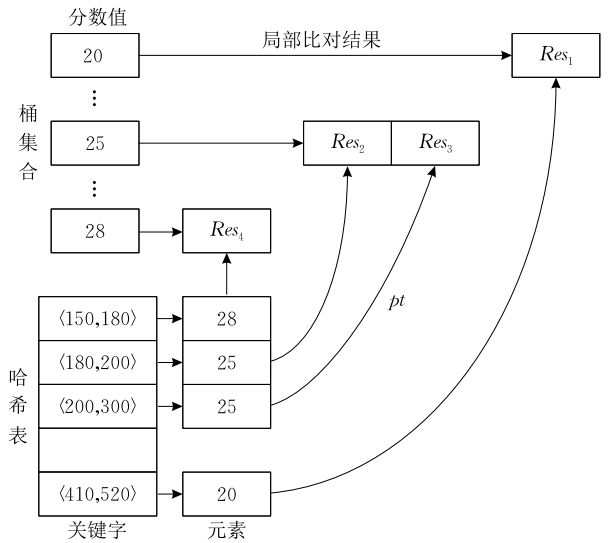


图 7 桶结构和哈希表的示例

ESLA^k 基于$\mathcal{B}$维护候选集$\mathcal{R}$. 具体地，当一个新的局部比对结果 r 到来时，ESLA^k 访问$\mathcal{H}$，并根据 r 的目标插入位置 I_r 执行如下操作.

(1) $I_r=\varnothing$. 这说明 r 既不支配$\mathcal{R}$中元素也不被$\mathcal{R}$中元素支配. 在这种情况下，ESLA^k 将 r 插入 I_r 和$\mathcal{B}$. 与此同时，ESLA^k 更新阈值 H.

(2) $I_r\neq\varnothing$且 I_r 中存在分值小于 $F(r)$的元素. 这说明 r 支配$\mathcal{R}$中的元素. 在这种情况下，ESLA^k 首先通过 I_r. pt 定位 r 支配的元素 r_d. 其次，ESLA^k 用 r 替换 r_d.

(3) $I_r\neq\varnothing$且 I_r 中不存在分值小于 $F(r)$的元素. 这说明 r 被 I_r 中的元素支配. 在这种情况下，ESLA^k 直接丢弃 r.

综上所述，ESLA^k 通过两方面的优化提高了计算效率. 第一，ESLA^k 使用启发式策略找到 k 个分

① $\min(\mathcal{R})$表示$\mathcal{R}$中分值最小的结果. 特殊地，当$|\mathcal{R}|<k$时，$\min(\mathcal{R})=0$.

值较大的比对结果. 通过这种方法，ESLA^k 可以有效解决“小阈值”问题. 第二，算法使用桶$\mathcal{B}$和哈希表$\mathcal{H}$维护候选对象. 借助它们，ESLA^k 可以快速识别新的比对结果是否被候选集中的元素支配.

3.4　外存访问优化策略

前文研究的焦点是在内存环境下优化基于后缀树的过滤算法. 然而，后缀树只索引子串的概要信息并且 ESF 只将 T 的部分子串保存在内存，当待访问的子串不在内存中时，算法需要从外存中读取数据. 此外，因为后缀树边之间存储的子串在 T 中不是连续的，所以磁盘读取操作将会频繁发生. 显然，这会导致算法的性能降低.

基于上述观察，本节提出 ESLA 算法优化外存的访问. 为了达到这个目的，本文从以下两个方面入手：(1) 对 T 中各部分子串被访问的频率进行研究. 基于研究结果，本文挑选一部分访问频率高的子串常驻内存；(2) 找到一种可以迎合磁盘工作特性的访问算法. 该方法可以通过实现磁盘的顺序访问提高磁盘的读取效率. 显然，上述两种方法可以有效降低外存访问代价.

3.4.1　后缀树访问特性的分析

给定原串 T 和它对应的后缀树 $\mathcal{I}$，本文将 $\mathcal{I}$ 上的边分为两种：(1) 内边；(2) 叶边. 其中，内边表示中间节点之间的边；叶边表示中间节点和叶子节点之间的边. 通过观察各类边上索引的子串特征，本文得到如下关于字符串 T 和其对应后缀树的特性：

特性 1. 大多数情况下，内边上索引的子串出现在 T 的前部；

特性 2. 大多数情况下，内节点间边上索引的字符串长度较短.

第 1 个特性是根据后缀树的构造过程得到的. 具体地，后缀树边上索引的内容为 T 中的最早出现的字符串[22]. 这种表示方法使得内边上索引的字符串大多出现在原串 T 的前半部分. 例如，给定长为 2 MB 老鼠基因和以此为基础构建的后缀树，超过 86% 内边上索引的子串出现在原串的前部. 因此，选择原串的前部常驻内存可以降低磁盘的访问频率.

第 2 个特性是根据后缀树的工作原理得到的. 因为后缀树根据子串的公共前缀索引 T 的各子串，所以后缀树上的大多数内边包含的子串长度较短. 原因如下：如定理 2 所示，给定随机生成的字符串 R 和它的两个后缀 r_1，r_2，因为 r_1，r_2 之间存在长公共前缀的概率较小，对于大部分子串而言，它们的公共前缀都是较短的.

定理 2.　给定随机生成的字符串 R 和它的两个后缀 r_1，r_2，$Pr(r_1[0,d]=r_2[0,d])=1/|\Sigma|^d$.

证明.　给定任意 $r_1[i]$和 $r_2[i]$，$Pr(r_1[i]=r_2[i]$等于 $1/|\Sigma|$. 因为 $Pr(r_1[i]=r_2[i])$与 $Pr(r_1[j]=r_2[j](i\neq j)$可以看成独立事件，所以 $Pr(r_1[0,d]=r_2[0,d]=1/|\Sigma|^d$.　证毕.

3.4.2　外存操作优化技术

由前文可知，当对后缀树进行遍历时，如果边上索引的子串不包含在内存中，ESLA 需要从磁盘中读出相应的部分. 显然，外存访问策略对于 ESF 的整体性能有很大的影响. 本文提出算法 ESBR. 它根据上节提到的两种性质优化 ESF.

根据特性 1，ESF 尽可能将原串的前部留在内存$\mathcal{M}$中. 除此之外，本文提出了一种批处理算法降低 I/O 代价. 首先本文给出溢出的概念.

定义 6(溢出).　给定后缀树$\mathcal{I}$、$\mathcal{I}$上的某条边 s 和它索引的子串 t，如果 t 不完全包含在$\mathcal{M}$中，本文将这种情况称为溢出.

回顾 2.2 节对后缀树的描述，因为后缀树相邻边之间索引的子串不具有局部性，所以相邻的后缀在原串中的位置可能相距很远. 因此，在遍历后缀树时，溢出事件会频繁发生. 显然，算法会因此消耗大量的磁盘寻址代价.

基于上述观察，本文提出一种批处理算法解决频繁溢出带来的问题. 其中，特性 2 保证了该算法的可行性. 具体地，当溢出发生时，ESLA 将当前进行的 ALC 中间结果存入缓冲区并推迟外存读取时间. 由文献[7-8]可知，因为需要保存的中间结果主要是动态规划矩阵的一行，所以保留这些中间结果需要花费较大的存储空间. 接下来，本文使用上节提到的特性 2 降低存储代价.

根据上节描述，大多数内节点索引的字符串长度相对较小. 如果溢出事件发生在内边，计算过的动态规划矩阵的行数往往较少. 在这种情况下，ESLA 如果访问到发生溢出的内边便停止访问，需要保存的字符串长度也会相应变小. 显然，同保存完整的中间结果相比，该策略可以有效节省存储空间. 显然，特性 2 可以帮助 ESF 花费相对较少的空间存储中间结果.

然而，如果溢出发生在叶边，ESF 需要采用相似的操作策略. 和内边相比，叶边发生溢出的频率更高且需要保存的中间结果也越多. 因此，这些中间结果需要更为合理的管理算法.

根据上述讨论，本文将内存$\mathcal{M}$近一步分成$\mathcal{M}_c$和

$\mathcal{M}_d$. 其中，$\mathcal{M}_c$ 存储 T 的头部子串(等于 $T[0,|\mathcal{M}_c|-1]$). 与 $\mathcal{M}_c$ 相比，$\mathcal{M}_d$ 中包含的字符串是离散的. 为了方便对这些零散的字符串进行管理，ESBR 提出了一种槽(shot)结构管理这些数据.

定义 7(槽). 给定散列表 $\mathcal{T}$ 和它的一组元素 $\{e_0,e_1,\cdots,e_n\}$，任意元素 e_i 对应于一个元组 $\langle key_i,s_i\rangle$. 其中，key_i 为键值，s_i 为一个数组. 通常，它被其称为槽.

具体地，ESBR 以槽为单位与磁盘中的数据进行交换，交换的规模等于槽的大小. 为决定哪些槽需要交换，ESBR 为每一个槽赋予一个优先级. 当需要进行数据交换时，ESBR 访问优先级低的槽并将它存储的内容替换为新读入的数据.

为了给所有槽设置合理的优先级，本文为所有槽附加了字段 at(accessing time). 它记录了槽内数据被访问的次数. 相应地，本文将槽的优先级设置为 $\frac{c}{at}$，其中 c 代表了一个常数. 它的作用是将访问不频繁的数据从槽中置换出去.

与槽结构相对应，算法使用小顶堆 $\mathcal{H}$ 决定优先读入 $\mathcal{M}_d$ 的子串. 具体地，$\mathcal{H}$ 维护溢出子串在 T 中的起始位置. 当 ESBR 需要读取磁盘的数据时，它首先将 $\mathcal{H}$ 的堆顶 h 弹出. 其次，算法根据 h 读取溢出子串并替换优先级最低槽中的数据.

根据上文描述，使用小顶堆维护数据的读入顺序具有以下好处：(1) 实现顺序访问. 因为 $\mathcal{H}$ 中的元素是基于溢出子串的起始位置维护的，ESBR 可以基于 $\mathcal{H}$ 实现磁盘的顺序读写. 显然，ESBR 迎合了磁盘的工作特性，避免了过多的磁道寻址；(2) 降低磁盘读写次数. 给定 $\mathcal{H}$ 中连续弹出的元素 h_1 和 h_2，由于它们对应的溢出子串位置相对接近，如果基于 h_1 读出的子串包含 h_2 对应的子串，ESBR 可以避免针对 h_2 的磁盘访问.

图 8 演示了 ESBR 的工作过程. 假设单个槽的容量为 1 MB、中间结果 IMR_i 需要的字符串起始位置为 1500 MB，由于当前的槽中不包括所需的字符串，ESBR 需要从外存中读入 $T[1500\text{MB},1501\text{MB}]$. 然后，ESBR 在槽中寻找插入位置. 如图所示，因为槽 3 的优先级最低(=0.3)，ESBR 将 $T[1500\text{MB},1501\text{MB}]$ 替换到该槽中. 最后，ESBR 更新该槽的优先级.

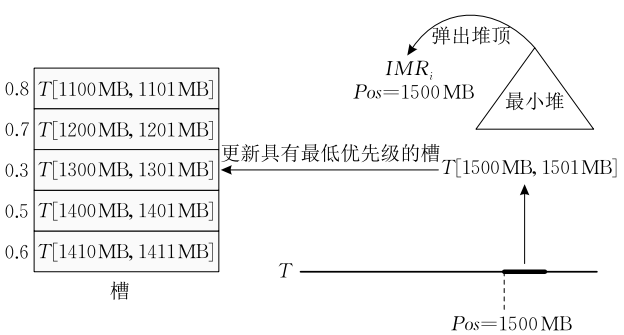


图 8 槽的交换操作的示意

为了便于理解，算法 1 给出了 ESBR 的基本流程. 它用最小堆 $\mathcal{H}$ 来保存局部比对的中间结果. 这些中间结果是在执行 ALC 时因溢出产生的. 当 $|\mathcal{H}|$ 大于阈值 θ 时，ESBR 集中对 $\mathcal{H}$ 中元素进行处理并更新具有最低优先级的槽. 当 $\mathcal{H}$ 为空时，算法结束.

算法 1. 计算中间结果.

输入：存储有中间结果集合的最小堆 $H_{\min}$

1. while 最小堆 $H_{\min}$ 非空 do
2. 从最小堆 $H_{\min}$ 中弹出堆顶元素 IMR_i；
3. if IMR_i 需要的字符串并没有包含在当前的槽中 then
4. 执行外存读取操作读取 IMR_i 需要的字符串 X；
5. 更新具有最低优先级的槽；
6. end if
7. 使用 X 来继续进行 IMR_i 的局部比对运算，若有新的中间结果产生则将其插入 $H_{\min}$ 中；
8. end while

4 实 验

本节描述了局部比对外存算法的相关实验. 本实验使用真实数据集作为测试集. 第 1 个数据集是人类基因序列①(GRCh37). 它包含 24 条染色体，染色体长度的变化范围为[48 MB,249 MB]. 它们的总长度为 2.98 GB，本文将它作为原串. 本文使用的第 2 个数据集是老鼠基因数据②(MGSCv37 chr1). 它的长度为 198 000 000 bp. 因为老鼠的基因与人类基因有相似之处，所以本文将其作为查询串. 为了测试不同查询串对算法的影响，本文从基因数据中随机选取不同的起始位置 s，并以此为起点生成不同长度 $|P|$ 的查询串. 其中，$|P|$ 的长度变化范围为[10 KB,2 MB].

4.1 算法的总体运行效率

本实验测试了局部比对外存算法的总体运行效率. 本实验使用约 2.98 GB 的人类基因序列(GRCh37)作为原串 T，并在其基础上建立外存后

① http://hgdownload.cse.ucsc.edu/goldenPath/hg18/chromosomes/

② http://hgdownload.cse.ucsc.edu/goldenPath/mm9/chromosomes/

缀树. 查询串是从鼠基因数据上抽取得来的. 本实验分别选取了 10 KB, 100 KB, 500 KB, 1 MB 和 2 MB 长的子串作为查询串. 表 2 说明了实验的硬件环境和软件环境.

表 2　局部比对外存算法实验环境

类别	描述	
硬件环境	CPU	i7@2.93 GHz
	内存	4 GB
	硬盘	500 GB
操作系统	Ubuntu(Linux) 10.04	
编程环境	GNU C++	

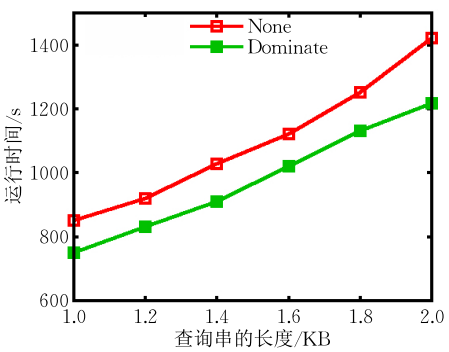
本节首先测试了 $|T|$ 不变时，查询串长度对算法性能的影响. 本组实验分别统计了内存计算时间 T_m，外存操作时间 T_e 和总时间 T^+. 由表 3 所示，随着 $|P|$ 的增长，T_m 和 T_e 都会增加. 具体地，因为 $|P|$ 的增加会导致 I/O 次数的增加，所以 T_e 随着 $|P|$ 的增加而增加；因为局部比对算法的时间复杂度与 $|P|$ 呈线性关系，所以 T_m 也随着 $|P|$ 的增加而增加. 此外，对于 T_m 而言，因为 ESLA 还需维护因溢出而保留的中间结果，所以 T_m 的增长速度更快一些. 例如，当 $|P|$ 增加到 2 MB 时，ELSA 需要花费大约 2700 ms.

表 3　当变化查询序列长度时的算法运行时间(s)比较(n=2.98 GB)

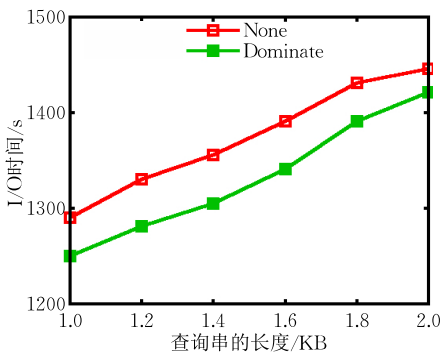
	m=10 K	m=100 K	m=500 K	m=1 MB	m=2 M
内存计算	89.532	209.090	441.453	748.475	1282.568
外存操作	1060.814	1097.016	1171.809	1248.155	1425.605
总和	1150.346	1306.106	1613.262	1996.630	2708.173

4.2　基于支配关系的过滤技术对算法的影响

本节测试了基于叉子区域支配关系的过滤能力. 其中，图 9(a)和图 9(b)分别说明了基于支配关系的过滤算法对内存计算和外存操作的影响. 如图 9(a)所示，与不使用基于支配关系的过滤算法相比(None 对应的曲线)，基于支配关系的过滤算法能节省大约 15%的内存计算时间. 然而，如图 9(b)所示，该技术对于外存操作的影响较小. 它只节省了大约 5%的操作时间. 其深入原因是这种过滤算法只能避免有限的外存操作.



(a) 支配关系过滤技术对内存计算的影响



(b) 支配关系过滤技术对外存操作的影响(n=2.98 GB)

图 9　支配关系过滤技术内存和外存的影响

4.3　top-k 算法的运行效率

本节对 top-k 算法的整体运行效率进行了测试. 其中，参数 k 分别设置为 2000, 4000, 6000 和 8000；原串 T 的长度分别为 100 MB, 200 MB 和 300 MB.

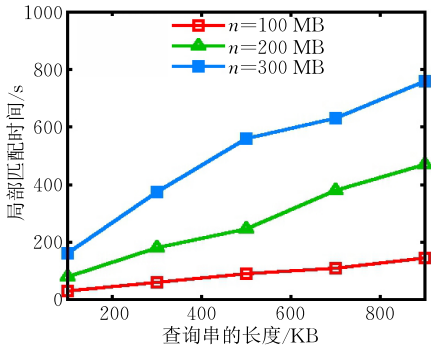
图 10 演示了不同的参数值 k 下，在不同长度的原串上寻找 top-k 局部比对的运行时间. 在固定的原串长度下，算法的运行时间基本和查询串的长度呈线性关系，而不同的原串长度对于算法的影响较大. 容易看出不同的参数 k 对算法的运行时间影响不大，原因是根据启发式策略得到的较大的分数. 因而在给定不同的参数 k 的情况下，算法的运算代价基本相同，运行时间也基本保持不变.

根据之前的描述可知维护局部比对结果是十分重要的，此处分别测试了在原串 T 的长度为 200 MB 和 300 MB 时为了获取 top-10 000 和 top-20 000 局部比对结果，维护计算过程中发现的局部比对结果所需要的时间，实验结果如图 11 所示.

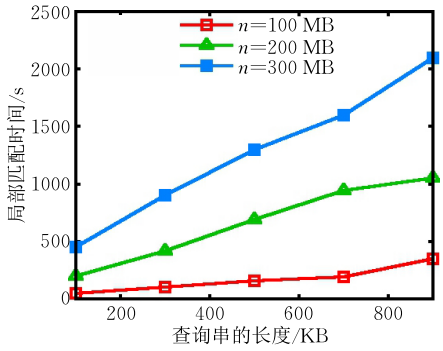
图 11 中参数 k 对运算时间影响不大，在图 11(a)和图 11(b)中两者的时间差距都不超过 5 s，造成这一结果的原因在于哈希表的容量较大，无论是 top-10 000 还是 top-20 000 对于哈希表的性能影响都不大. 比较图 11(a)和图 11(b)的结果可以发现，图 11(b)中用于维护局部比对结果的时间要比图 11(a)中的时间要长大约 10 余秒的时间，这是由于在原串长度较长时，新计算的局部比对结果支配已计算的局部比对结果的情况出现得更加频繁，因而用于更新局部比对结果的时间较长.

4.4　外存操作优化技术的影响

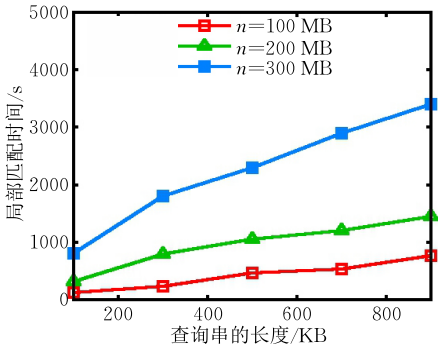
本节测试了外存优化策略对算法的影响. 算法的外存操作可以分为以下两种：(1) 读取外存后缀



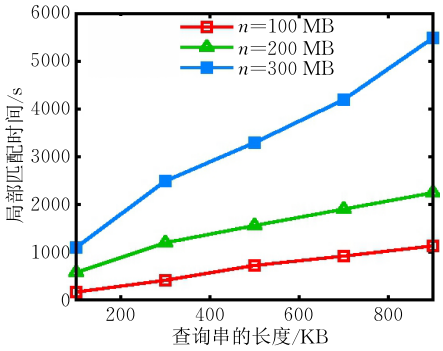
(a) 寻找top-2000局部比对的运行时间



(a) 寻找top-4000局部比对的运行时间

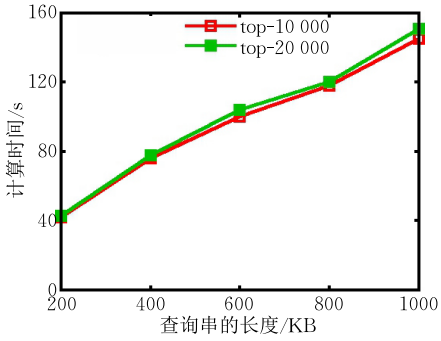


(c) 寻找top-5000局部比对的运行时间

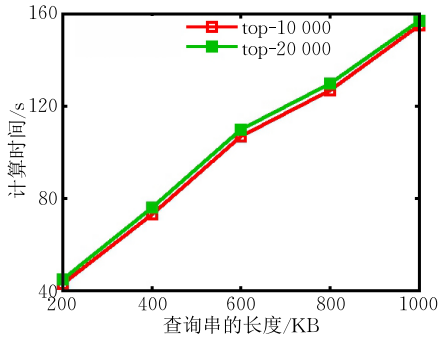


(d) 寻找top-6000局部比对的运行时间

图 10 top-k 查询时间



(a) n=200 MB时维护局部比对结果的时间



(b) n=300 MB时维护局部比对结果的时间

图 11 维护局部比对结果的时间

树. 由于后缀树索引的大小是固定的,所以这部分的时间是固定不变的;(2) 读取没有保存在内存中的原串. 这部分时间主要由外存操作频率决定.

由前文可知,算法共维护两类子串:(1) 常驻内存的子串$\mathcal{M}_c$;(2) 用于交换的子串$\mathcal{M}_d$. 因此,本组实验首先测试了常驻子串长度对算法的影响. 如图所示,常驻内存的子串长度对算法性能有着较大的影响. 当常驻内存子串的长度为 2 GB 时,算法的性能明显高于子串的长度为 1.6 GB 时算法的性能. 其主要原因是当常驻内存的子串较长时,I/O 次数也会随之减少. 然而,当内存总量一定时,常驻内存的子串过长时也会降低算法的性能.

如图 12 所示,当常驻内存的字符串大小为 2.4 GB 时,算法的总体性能低于$|\mathcal{M}_c|$=2 GB 时的情形. 造成这一结果的原因为:虽然$\mathcal{M}_c$可以保持更多的原串在内存中,但是由于$\mathcal{M}_d$较小,每次参与批处理的

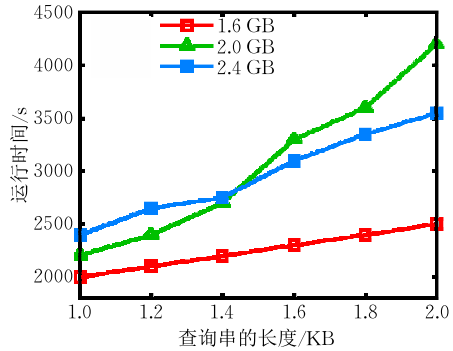


图 12 常驻内存的字符串的大小对算法运行时间的影响

子串较少，磁盘寻址次数也会增多. 因此，当$\mathcal{M}_c$过大时反而会影响算法的性能. 总之，$\mathcal{M}_c$大小的选取对算法的性能有着较大的影响. 由于原串和查询串中字符的分布都是未知的，本文无法给出一个理论上的拐点. 相反，本文可从大量实验出发找到相对最优的$|\mathcal{M}_c|$.

最后，本节测试了外存交换策略对算法的影响. 其中，本文对比了以下 3 种置换策略. 它们分别是：(1) MU 算法(优先置换使用次数最多的字符串)；(2) LU 算法(优先置换使用次数最少的字符串)；(3) FIFO 算法(先进先出的置换策略). 如图 13 所示，3 种置换策略对应的曲线十分接近，其中 MU 策略要略微占优，这从侧面反映出字符串被再次使用的概率较低的结论是正确的.

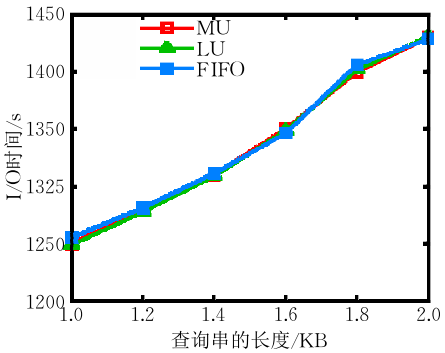


图 13　外存交换策略对 I/O 时间的影响

5　总结与展望

本文对 top-k 局部比对问题进行了研究. 和以往文献相比，本文研究的问题更为复杂. 首先，它需要考虑“小阈值”带来的问题. 其次，它需要突破磁盘 I/O 引发的瓶颈. 为了解决上述问题，本文首先将一些成熟的过滤策略与本文的研究环境有机结合. 其次，本文引入一种启发式算法解决“小阈值”问题. 最后，本文根据磁盘和后缀树的工作特性提出了一系列外存优化策略. 通过大量的实验验证了本文所提出算法的有效性.

然而，虽然本文提出的算法具有较高的运行效率，但是还有一些不足需要解决：

(1) 寻找 top-k 局部比对问题的算法没有充分利用高分数阈值，未来需要研究如何利用高分数阈值来过滤不必要的计算，进一步提升算法的运行效率.

(2) 使用外存后缀树来解决局部比对问题对于硬盘空间的要求太大，在硬盘上读入后缀树需要花费大量的时间，将来将研究是否有其他占据空间小并且兼顾运行效率的索引结构来代替后缀树的使用.

参　考　文　献

[1] Sellers H. The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms, 1980, 1(4): 359-373

[2] Smith F, Waterman S. Identification of common molecular subsequences. Journal of Molecular Biology, 1981, 147(1): 195-197

[3] Raphael B, Zhi D, Tang H. A novel method for multiple alignment of sequences with repeated and shuffled elements. Genome Research, 2004, 14(11): 2336-2346

[4] Paten B, Herrero J, Beal K. Sequence progressive alignment: A framework for pratical large-scale probabilistic consistency alignment. Bioinformatics, 2009, 25(3): 295-301

[5] Pearson R, Lipman J. Improved tools for biological sequence comparison. Journal of Computer and Communications, 1988, 85(8): 2444-2448

[6] Altschul F, Gish W, Miller W. Basic local alignment search tool. Journal of Molecular Biology, 1990, 215(3): 403-410

[7] Tak L, Wing S, Siu T. Compressed indexing and local alignment of DNA. Bioinformatics, 2008, 24(6): 791-797

[8] Yang Xiao-Chun, Liu Hong-Lei, Wang Bin. ALAE: Accelerating local alignment with affine gap exactly in biosequence databases//Proceedings of the Very Large Database(VLDB'12). Istanbul, Turkey, 2012: 1507-1518

[9] Adam J. The pathologies of big data. Communications of the ACM, 2009, 52(8): 36-44

[10] Gonzalo N, Veli M. Compressed full-text indexes. ACM Computing Surveys, 2007, 39(1): 1-61

[11] Barsky M, Ulrike S, Alex T. Full-text (substring) indexes in external memory. Synthesis Lectures on Data Management, 2011, 3(7): 1-92

[12] Barsky M, Ulrike S, Alex T. Suffix trees for inputs larger than main memory. Information Systems, 2011, 36(3): 644-654

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

[14] Kim K, Sim S, Park H. Linear-time construction of suffix arrays//Proceedings of the Combinatorial Pattern Matching (CPM'03). Morelia, Mexico, 2003: 186-199

[15] Ko P, Aluru S. Space efficient linear time construction of suffix arrays//Proceedings of the Combinatorial Pattern Matching(CPM'03). Morelia, Mexico, 2003: 200-210

[16] Karkkainen J, Sanders P, Burkhardt S. Linear work suffix array construction. Journal of the ACM, 2006, 53(6): 918-936

[17] Lam W, Sadakane K, Sung W. A space and time efficient algorithm for constructing compressed suffix arrays//Proceedings of the Computing and Combinatorics (COCOON'02). Singapore, 2002: 401-410

[18] Hon K, Lam W, Sadakane K. Constructing compressed suffix arrays with large alphabets//Proceedings of the Algorithms and Computation (ISAAC'03). Kyoto, Japan, 2003: 240-249

[19] Hon K, Lam W, Sadakane K. Compressed index for dynamic text//Proceedings of the Data Compression Conference (CPM'04). Snowbird, USA, 2004: 102-111

[20] Meek C, Patel M, Kasetty S. OASIS: An online and accurate technique for local-alignment searches on biological sequences//Proceedings of the Very Large Database(VLDB'03). Berlin, Germany, 2003: 910-921

[21] Shpaer E, Robinson M, Yee D, et al. Sensitivity and selectivity in protein similarity searches: A comparison of Smith-Waterman in hardware to BLAST and FASTA. Genomics, 1996, 38(2): 179-191

[22] Burrows M, Wheeler D. A Block Sorting Lossless Data Compression Algorithm. Middlesex: Digital Equipment Corporation, 1994

[23] Ferragina P, Manzini G. Opportunistic data structures with applications//Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'00). Redondo Beach, USA, 2000: 390-398

[24] Wong C, Zhuang D, Li L. Discovery of delta closed patterns and noninduced patterns from sequences. IEEE Transactions on Knowledge and Data Engineering, 2012, 24(8): 1408-1421

[25] Mansour E, Allam A, Skiadopoulos S. ERA: Efficient serial and parallel suffix tree construction for very long strings//Proceedings of the Very Large Database. Washington, USA, 2011: 49-60

[26] Knuth D, Morris H, Pratt R. Fast pattern matching in strings society for industrial and applied mathematics. Journal on Computing, 1977, 6(2): 323-350

[27] Salavert J, Blanquer I, Tomas A. Using GPUs for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2012, 9(4): 1245-1256

WANG Bin, born in 1972, associate professor. His main research interest is big data management.

ZHU Rui, born in 1982, Ph. D. candidate. His research interests include sensor data management and uncertain data management.

YANG Xiao-Chun, born in 1973, Ph. D., processor, Ph. D. supervisor. Her main research interests include string data, parallel computing.

WANG Guo-Ren, born in 1966, Ph.D., professor, Ph.D. supervisor. His research interests include database, big data management.

YU Ge, born in 1962, Ph. D., professor, Ph. D. supervisor. His major research interests include database, big data management.

Background

Recently, biosequence search has been given much attention. In this area, scientists often compare a biosequence against a series of known sequences. Generally, two biologically related sequences may contain subsequences that are highly similar. Local alignment is a common technique for finding a pair of highly similar substrings from two given sequences, and it has been fully studied. However, with the enlargement of biosequence scale, the conventional PC can not entirely load all the biosequence into the memery, and the state of arts approaches can not well work in this case.

A variety of computational algorithms have been developed for finding local alignments, among which BLAST (Basic Local Alignment Search Tool), the Smith-Waterman algorithm and the ALAE are typical ones. However, these algorithms are only doable in memory. If they are simply employed in the external environment, the algorithm performance may be sharply fall due to the frequently I/O operations.

In this paper, we study the problem of top-k local alignment over external suffix tree. In order to overcome the challenges in this work, we propose the ESF framework. It firstly employs a group of efficiently filtering strategies for accelerating the traversal of suffix tree. Based on these, we study the problem of answering top-k local alignment. And then, via studying the operational principle of disk and suffix tree, we propose several algorithms for reducing the I/O cost. Lastly, we evaluate the performance of our framework.

This work is supported by the National Basic Research 973 Program of China under Grant No. 2012CB316201, the National Natural Science Foundation of China under Grant Nos. 61572122, 61173031, 61129002, 61532021, and U1401256, and the National Natural Science Foundation of China for Outstanding Young Scholars (No. 61322208).