**中山大学博士学位论文**

**中文题目：大规模前缀和后缀索引的若干算法研究**

**英文题目：Research on Algorithms for Large-scale Prefix and Suffix Indices**

**专业： 计算机科学与技术**

**学位申请人： 吴裔**

**导师姓名：** **农革**

论文答辩委员会主席：

成员：

年 月**论文原创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

学位论文使用授权声明

本人完全了解中山大学有关保留、使用学位论文的规定，即：学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版，有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆、院系资料室被查阅，有权将学位论文的内容编入有关数据库进行检索，可以采用复印、缩印或其他方法保存学位论文。

学位论文作者签名： 导师签名：

日期： 年 月 日

# 论文题目：大规模前缀和后缀索引的若干算法研究

专业：计算机科学与技术

博士生：吴裔

指导教师：农革

# 摘 要

索引可大幅提高匹配、聚类和对齐等字符串处理操作的执行效率。随着待处理数据规模的增长，需研究可处理海量数据的索引技术。本文针对大规模前缀和后缀索引，提出若干查找、构造和正确性验证算法，各算法部署于内存、外存或分布式模型，具体包括：

1. 在多存储块内存模型上设计一个最长前缀匹配算法。算法改造已知的指针型前缀树构造算法，将输入字符串集索引为多棵前缀树并将各前缀树结点随机存储于复数个存储块。算法可同时调度多个查找请求访问位于不同存储块中的前缀树结点。算法将查找请求与目标存储块之间的“请求-授权-访问”关系建模为二部图匹配问题，通过寻找二部图中的“最大匹配集”来最大化查找并行度。我们将运用该算法构建一个高吞吐量的流水路由查找系统。
2. 在外存模型上设计一个后缀数组构造算法。算法改造已知的内存后缀数组构造算法，采用归纳排序方法将输入字符串的所有后缀按其字典序排列，核心思想是从有序后缀的字典序推导无序后缀的字典序。算法的时、空复杂度与外存整数排序相当。
3. 设计一个有限阶最长公共前缀数组构造算法。算法采用递归折半法计算输入字符串的有限阶最长公共前缀数组，核心思想是先比较两个字符串的前半部分，若相等则比较后半的前半部分，否则比较前半的前半部分，以此类推。已知字符串的长度为n，若在内存模型上计算K阶最长公共前缀数组，则算法的时间复杂度线性正比，空间复杂度线性正比于n。算法易于扩展，可在外存和对称型分布式模型上实施。
4. 设计两个后缀数组和最长公共前缀数组的事后验证算法。算法一判断输入的后缀数组和最长公共前缀数组是否与定义相符，该算法的通用性好，可检验完整或稀疏后缀数组及其最长公共前缀数组。算法二首先验证输入的后缀数组和最长公共前缀数组的一个子集，然后采用归纳排序方法从已验证子集推导出完整的后缀数组和最长公共前缀数组并将其与输入比较来验证后者的正确性。目前，尚无文献记载最长公共前缀数组的事后验证算法，本工作填补了该研究空白。两算法为概率性算法，可部署于单机内存和外存模型。
5. 设计两个后缀数组事中验证算法。两算法需与采用归纳排序方法设计的后缀数组构造算法联用，核心思想是将后缀数组的验证问题转化为其子数组的验证问题并在后缀数组的构造过程中验证子数组的正确性。算法一为概率性算法，可在整数排序的时、空复杂度内验证构造中的后缀数组。可改造算法一并将其与采用归纳排序方法设计的最长公共前缀数组构造算法联用，实现同步构造和验证最长公共前缀数组的目的。算法二为确定性算法，在线性时间和常数空间内验证构造中的后缀数组。两算法可部署于单机内存和外存模型。

**关键词：前缀和后缀索引；查找、构造和验证；内存、外存和分布式模型**

Title: Research on Algorithms for Large-scale Prefix and Suffix Indices

Major: Computer Science and Technology

Name: Yi Wu

Supervisor: Ge Nong

# Abstract

Indexing techniques can be applied to accelerating a plethora of string processing tasks, such as matching, clustering and aligning. With the rapid increase in the size of datasets, there comes a new research challenge on text indexing. This paper proposed several algorithms specific for large-scale prefix and suffix indices, with the purposes of conducting searching, building and checking tasks. Our main contributions are concluded as below:

We proposed a searching algorithm for longest matching prefix on multi-block internal memory model. This algorithm adapts the existing prefix tree construction algorithms to organize a set of strings as multiple prefix trees and adopts a random allocation scheme to evenly scatter nodes of prefix trees onto a number of memory blocks. The algorithm schedules multiple searching tasks to visit prefix nodes residing on different memory blocks at the same time. To avoid memory access contentions among tasks aiming at the same block, it casts the scheduling problem as bipartite matching and attempts to find a maximum matching set during each scheduling instance for maximizing throughput. In the paper, we exploited the use of this algorithm to build a high-throughput pipelined routing system.

We proposed a suffix array (SA) construction algorithm on external memory model. This algorithm is adapted from an existing suffix sorting algorithm designed on internal memory model. The main idea behind our proposal is to determine the lexical order of unsorted suffixes from those already sorted using the induced sorting (IS) method. The time and space complexities of this algorithm is comparable to that of disk-based integer sorts.

We proposed a finite-order longest common prefix (LCP) array construction algorithm. This algorithm computes the LCP-value of two suffixes as following: (1) conduct a checking process to determine whether the left halves of two suffixes are equal; (2) if equal, then recursively conduct the checking process with the right halves of two suffixes as input; otherwise, recursively conduct the checking process with the left halves of two suffixes as input. Suppose the input string contains n characters, the time and space complexities to compute a K-order LCP array by this algorithm are linear proportional to and n, respectively, where K is typically no more than a hundred. Compared to the prior arts, this algorithm is rather scalable, it can be deployed on internal, external and distributed models.

We proposed two algorithms for checking constructed suffix and LCP arrays. The first algorithm determines whether or not the given suffix and LCP arrays are correct following the definitions of two arrays, it is universal for checking both sparse or full suffix and LCP arrays. The second algorithm first checks a subset of the given suffix and LCP arrays by calling the first algorithm, then it uses the IS method to induce the full suffix and LCP arrays from the verified subset and compares the induced two arrays with their input counterparts to ensure the correctness of the latter. At the time of writing this paper, there exists no research on checking LCP array, our work fills the gap in this field. Both algorithms can be applied on internal and external memory models.

We proposed two algorithms for checking an SA when it is being constructed. Both algorithms can be integrated into any IS suffix sorting algorithm to make the sorter capable of building and checking an SA at the same time. The main idea is to transform the problem of checking an SA to the problem of checking a subset of the SA, where the latter can be solved within much less and space. The first algorithm is of sorting time and space complexities, it can be also enhanced to check an LCP array built by the IS method. The second algorithm is much more lightweight, it only takes linear time and constant space to run. Both algorithms can be applied on internal and external memory models.

**Key Words：Prefix and Suffix Indices; Searching, Building and Checking; Internal, External and Distributed Models**

目 录

[论文题目：大规模前缀和后缀索引的若干算法研究 I](#_Toc494479326)

[摘 要 I](#_Toc494479327)

[Abstract III](#_Toc494479328)

[第一章 绪论 1](#_Toc494479329)

[1.1研究背景 1](#_Toc494479330)

[1.2研究内容 3](#_Toc494479331)

[1.2.1负载均衡的流水路由查找算法 3](#_Toc494479332)

[1.2.2空间节省的后缀数组外存构造算法 6](#_Toc494479333)

[1.2.3可扩展性好的有限阶最长公共前缀数组构造算法 7](#_Toc494479334)

[1.2.4新型后缀数组和最长公共前缀数组验证算法 8](#_Toc494479335)

[第二章 流水路由查找算法 9](#_Toc494479336)

[2.1预备知识 9](#_Toc494479337)

[2.2索引结构 11](#_Toc494479338)

[2.3体系架构 13](#_Toc494479339)

[2.4排队模型和调度算法 14](#_Toc494479340)

[2.4.1排队模型 14](#_Toc494479341)

[2.4.2调度算法 14](#_Toc494479342)

[2.4.3理论分析 16](#_Toc494479343)

[2.5实验分析 17](#_Toc494479344)

[2.5.1实验环境 17](#_Toc494479345)

[2.5.2存储需求评估 18](#_Toc494479346)

[2.5.3吞吐量评估 24](#_Toc494479347)

[2.6硬件设计 29](#_Toc494479348)

[2.7本章小结 30](#_Toc494479349)

[第三章 后缀数组构造算法 32](#_Toc494479350)

[3.1预备知识 32](#_Toc494479351)

[3.2 DSA-IS算法原理和设计 34](#_Toc494479352)

[3.2.1数据结构 34](#_Toc494479353)

[3.2.2收缩阶段 34](#_Toc494479354)

[3.2.3解推导阶段 39](#_Toc494479355)

[3.3设计优化 41](#_Toc494479356)

[3.3.1同步排序和命名子串 41](#_Toc494479357)

[3.3.2分治排序和命名子串 41](#_Toc494479358)

[3.4实现优化 42](#_Toc494479359)

[3.5实验分析 43](#_Toc494479360)

[3.5.1实验环境 43](#_Toc494479361)

[3.5.2实验结果 43](#_Toc494479362)

[3.6本章总结 48](#_Toc494479363)

[第四章 有限阶最长公共前缀数组构造算法 49](#_Toc494479364)

[4.1预备知识 49](#_Toc494479365)

[4.2算法原理 49](#_Toc494479366)

[4.3算法设计 50](#_Toc494479367)

[4.3.1内存 50](#_Toc494479368)

[4.3.2外存 51](#_Toc494479369)

[4.3.3分布式 52](#_Toc494479370)

[4.4算法优化 53](#_Toc494479371)

[4.5实验分析 54](#_Toc494479372)

[4.6本章小结 56](#_Toc494479373)

[第五章 后缀数组和最长公共前缀数组事后验证算法 57](#_Toc494479374)

[5.1 预备知识 57](#_Toc494479375)

[5.2 算法一 57](#_Toc494479376)

[5.2.1原理 57](#_Toc494479377)

[5.2.2设计 59](#_Toc494479378)

[5.2.3复杂度 60](#_Toc494479379)

[5.3 算法二 61](#_Toc494479380)

[5.3.1 原理 61](#_Toc494479381)

[5.3.2 设计 62](#_Toc494479382)

[5.3.3复杂度 62](#_Toc494479383)

[5.4 算法优化 63](#_Toc494479384)

[5.5 实验分析 63](#_Toc494479385)

[5.6 本章总结 69](#_Toc494479386)

[第六章 后缀数组事中验证算法 70](#_Toc494479387)

[6.1预备知识 70](#_Toc494479388)

[6.2算法一 71](#_Toc494479389)

[6.3算法二 71](#_Toc494479390)

[6.4 复杂度 72](#_Toc494479391)

[6.5 实验分析 72](#_Toc494479392)

[6.6 本章总结 74](#_Toc494479393)

[参考文献 76](#_Toc494479394)

[结 论 75](#_Toc494479395)

[攻读博士（硕士）学位期间发表学术论文情况 80](#_Toc494479396)

[致 谢 81](#_Toc494479397)

[中山大学学位论文版权使用授权书 82](#_Toc494479398)

# 第一章 绪论

## 1.1研究背景

索引技术广泛运用于文本聚合、数据挖掘和基因对齐等字符串处理程序。在此类程序中，模式匹配是常见任务，任务要求在给定的文本中查找并返回与输入模式相匹配的段落。可预处理文本，对其分词并构建倒排索引，该索引支持增量更新且查找速度快。倒排索引的缺点在于：难以返回未包含在词汇表中的模式在给定文本中的（所有）出现。与倒排索引相比，前缀和后缀索引为全文索引，可在其上快速查找任意模式在本文中的所有出现。本文研究大规模前缀和后缀索引的构造、查找和正确性验证算法，相关数据结构的定义和示例如下所述。

|  |  |  |  |
| --- | --- | --- | --- |
| 表1-1 字符串集合示例 | | | |
| 编号 | 字符串 | 编号 | 字符串 |
| 1 | ab | 4 | bcc |
| 2 | abc | 5 | bcd |
| 3 | abd | 6 | bce |

|  |
| --- |
| 图1-1 单字符前缀树示例.png |
| 图1-1 单字符前缀树示例 |

前缀树是一种经典数据结构，可索引单个或多个字符串。前缀树可解决词频统计和最长前缀匹配等问题。图1-1给出了表1-1中的字符串集合的单字符前缀树。各结点至少包含一个存储字符串的数据域和一个存储孩子指针的指针数组。各边用单个字符标记，该字符隐含延该边下行到达的孩子结点所存储的字符串信息。在单字符前缀树上执行一个模式的最长前缀匹配任务的一般过程可归纳为：从根结点出发，从上至下依次访问各层结点，直到到达某个叶子结点或满足匹配条件。匹配任务的关键步骤是：当访问完第i层的某个结点时，从待匹配字符串中获取左侧第i个字符并从该字符标记的分支下行至第i+1层的孩子结点。例如：已知待匹配模式为ab，则首先访问前缀树的根结点并从ab中取出下标为0的字符，然后从根结点出发从字符a标记的分支下行至根结点的最左孩子结点。由此可知，一个模式匹配任务的时间复杂度为,其中m为模式的长度。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位置 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
| 字符 | m | m | i | i | s | s | i | i | S | s | i | i | p | p | i | i |
| 表1-2 字符串示例 | | | | | | | | | | | | | | | | |

|  |  |  |
| --- | --- | --- |
| 起始位置 | 字典序排序的后缀 | 最长公共前缀 |
| 15 | i | 0 |
| 14 | ii | 1 |
| 10 | iippii | 2 |
| 06 | iissiippii | 2 |
| 02 | iissiissiippii | 6 |
| 11 | ippii | 1 |
| 07 | issiippii | 1 |
| 03 | issiissiippii | 5 |
| 01 | miissiissiippii | 0 |
| 00 | mmiissiissiippii | 1 |
| 13 | pii | 0 |
| 12 | ppii | 1 |
| 09 | siippii | 0 |
| 05 | siissiippii | 3 |
| 08 | ssiippii | 1 |
| 04 | ssiissiippii | 4 |
| 表1-3 后缀数组和最长公共前缀数组示例 | | |

后缀树按字典序组织字符串中的所有后缀，可在后缀树上查找待匹配模式在给定文本中的所有出现，后缀树的缺点是空间开销大。后缀数组是后缀树的简洁替代，根据定义，后缀数组是所有后缀的起始位置的一个全排列，排列顺序与后缀的字典序一致。例如：表1-3展示了表1-2中字符串的所有后缀，其中的起始位置列构成了字符串的后缀数组。最长公共前缀数组记录了后缀数组中相邻后缀的最长公共前缀的长度，该数据结构可辅助后缀数组模拟后缀树的遍历操作。如图1-3所示，最长公共前缀列构成了表1-2中字符串的最长公共前缀数组。在后缀数组和最长公共前缀数组上执行一个模式匹配任务的一般过程是一个递归折半查找过程，时间复杂度为O(m)，其中m为模式长度。

## 1.2研究内容

### 1.2.1负载均衡的流水路由查找算法

路由表包含一个或多个尾缀通配符的字符串，字符集为。可将路由查找过程形式化描述为最长前缀匹配问题：已知字符串集S和待匹配模式P，在S中查找P的前缀出现并返回最长前缀。为提高查找速度，可为路由表中的字符串集构建前缀树索引。现有binary tree、prefix tree[1]、fixed-stride tree[2]和multi-prefix tree[3]等多个用于路由查找的前缀树构造算法，各类前缀树的主要区别在于树中结点的数据结构。例如：binary tree中的单个结点存储1条路由和2个孩子指针，而fixed-stride tree中的单个结点存储条路由和孩子指针，k为跨度。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编号 | 前缀 | 路由信息 | 编号 | 前缀 | 路由信息 |
| 1 | 0\* | E1 | 6 | 011\* | E6 |
| 2 | 00\* | E2 | 7 | 110\* | E7 |
| 3 | 01\* | E3 | 8 | 1101\* | E8 |
| 4 | 000\* | E4 | 9 | 01001\* | E9 |
| 5 | 010\* | E5 | 10 | 01010\* | E10 |
| 表1-4 简化路由表示例 | | | | | |

为进一步说明，图1-2和1-3分别给出了表1-4中的简单路由表的binary tree和fixed-stride tree。从结构上来看，binary tree与单字符前缀树类似，其数据域存储了对应前缀的路由信息。例如：图1-2中的第1层有两个结点，分别代表前缀0\*和1\*。因为0\*属于路由表，所以对应结点的数据域中存储了路由信息E1。假设当前到达的数据包的目的地址为000000，则查找任务首先访问根结点得到前缀\*的默认路由，然后依次访问代表前缀0\*，00\*和000\*的结点并从中取出路由信息E1、E2和E4。因为000\*为路由表中与目的地址匹配的最长前缀，所以数据包的路由信息为E4。另一方面，跨度为2的fixed-stride tree中的每个结点包含4个数据域和4个孩子指针。为将路由表中的前缀插入到树中，需采用前缀扩展技术，将表中前缀就近扩展为一个或多个长度为2的整数倍的前缀。例如：0\*的长度为1，故将其扩展为00\*和01\*；010\*的长度为3，故将其扩展为0100\*和0101\*；00\*的长度为2，故保持不变。扩展后的前缀集合中可能存在多个字面值相同但路由信息不同的前缀。例如：由0\*扩展得到的前缀00\*和01\*的路由信息为E1，而原本就包含在路由表中的00\*和01\*的路由信息分别为E2和E3。此时，E1和E2均会被插入到根结点的同一个数据域中。根据匹配原则，应当用E2覆盖E1。为此，可先将扩展前缀按其原始长度从小到大排序，然后按序依次插入到前缀树中。在fixed-stride tree上的查找过程与在binary tree上的查找过程近似，区别在于：每次从当前访问的结点下行至孩子结点时，需查看目的地址中两个相邻比特。例如：假设当前到达的数据包的目的地址为010000，查找任务首先从目的地址中提取最左两比特01，然后从前缀01\*对应的数据域中提取路由信息E3，最后从标记为01的分支下行至孩子结点。在孩子结点，查找任务首先从目的地址中提取次左两比特00，然后从前缀0100\*对应的数据域中提取路由信息E5，最后从标记为0100的分支下行至孩子结点。查找任务重复上述过程直到到达叶子结点。在此过程中，E3和E5分别为与目的地址匹配的01\*和0100\*对应的路由信息。因为0100\*较长，所以数据包的路由信息为E5。

|  |  |
| --- | --- |
| 图1-2 binary tree示例.png | 图1-3 fixed-stride tree示例.png |
| 图1-2 binary tree示例 | 图1-3 跨度为2的fixed-stride tree示例 |

现有研究工作表明：可将前缀树映射到由复数个存储块组成的流水路由查找系统中，当多个查找任务同时访问位于不同存储块中的结点时，这些访问操作可并行执行，从而显著提高查找速度。目前常见的流水路由查找系统可分为线型和环型两大类，两类系统均采用按层存储结点的方式将前缀树映射到各存储块中。图1-4和1-5分别展示了图1-2中binary tree在线型和环型流水路由查找系统中的存储方式。需要注意的是：线型流水路由查找系统中的存储块个数不能小于前缀树的高度，而环型流水路由查找系统不受该约束的限制。具体而言，在图1-4的线型流水系统中，binary tree的第4和5层结点分别位于存储块5和6中；而在图1-5的环型流水系统中，这两层结点分别与第0和1层结点共用存储块1和2。每个查找任务从左向右依次访问线型/环型流水系统的存储块来遍历前缀树结点，从而多个查找请求可错时进入流水系统并同时执行。假设系统时间被切分为连续的时间片且每个存储块在单个时间片内仅被允许访问一次，则线型和环型路由查找系统的最大吞吐量为每时间片完成一个查找任务。

|  |
| --- |
| 图1-4 线型流水路由系统示例.png |
| 图1-4 线型流水路由系统示例 |

|  |
| --- |
| 图1-5 环型流水路由系统示例.png |
| 图1-5 环型流水路由系统示例 |

线型和环型流水路由系统的缺点包括以下两个方面：（1）各存储块的空间需求不平衡；（2）各存储块的访问负载不平衡。这主要是由于以下两个原因：（1）IPv4/IPv6路由表中的前缀的长度分布不均匀，例如：IPv4路由表中的大半前缀为16和24比特；（2）前缀树自身的结构特征导致各层结点的数量差异较大。这引起以下现象：（1）因为线型和环型流水均按层存储结点，所以相同层的结点位于同一个存储块且存储24比特前缀的路由信息的存储块所需空间远大于其他存储块；（2）一个查找任务一般从前缀树的根结点出发至叶子结点结束，而前缀树中的大部分叶子存储24比特前缀的路由信息，这表明存储更长前缀的路由信息的存储块所承受的访问频率远小于其他存储块。第二章提出一种负载均衡的流水路由查找算法，其改造前缀树索引并采用新的结点分布方案来平衡流水系统中各存储块的存储负载和工作负载。

### 1.2.2空间节省的后缀数组外存构造算法

|  |
| --- |
| 图1-6 块链技术实施示例.png |
| 图1-6 文件块链技术示例 |

后缀数组构是重要的索引数据结构，其构造问题长期得到研究关注，已有多个研究工作在不同计算模型上设计后缀数组构造算法。目前性能最优的内存算法是采用归纳排序方法设计的SA-IS，算法的时、空复杂度均为线性且常数因子小。该算法在外存模型上的变种有eSAIS、DSA-IS和SAIS-PQ，三者的时、空复杂度均优于DC3、bwt-disk、SAscan和pSAscan等其它外存算法。然而，采用IS方法设计的外存算法的当前实现存在磁盘空间开销过大的问题：在计算40比特整数编码的后缀数组时，eSAIS和DSA-IS的算法程序的磁盘峰值分别为24n和18n，而pSAscan的磁盘峰值约为8n。另一方面，SAIS-PQ在相同条件下的磁盘峰值为15n，但其运行速度是eSAIS和DSA-IS的1.5倍以上。分析eSAIS、DSA-IS和SAIS-PQ的算法流程可知：采用IS方法设计的后缀数组构造算法在执行过程中会生成大量临时数据，这些临时数据按访问顺序集中存放在数个文件中，每个文件仅在其上所有数据过期后方能被删除。可使用文件块链技术降低空间复杂度。如图1-5所示，文件块链技术将一个存储临时数据的文件划分为多个小文件，当一个小文件中的所有数据过期时，可删除小文件来快速回收不再使用的磁盘空间。近期的研究工作表明：文件块链技术可大幅提升采用IS方法设计的后缀数组外存构造算法。第三章介绍DSA-IS算法及其优化方法。

### 1.2.3可扩展性好的有限阶最长公共前缀数组构造算法

最长公共前缀数组可作为后缀数组的辅助，加快在后缀数组上的模式匹配速度。可按输入将最长公共前缀数组构造算法分为两类：第一类算法的输入为文本字符串，可同时计算后缀数组和最长公共前缀数组，第二类算法还要求输入其他辅助数据（例如：Burrows-Wheeler变换数组或后缀数组等）。对于第二类算法，Kasai等人首先在内存模型上提出了一个线性时间复杂度的构造算法，该算法要求输入文本字符串、后缀数组和逆后缀数组。其中，逆后缀数组是将后缀数组中的元素按其在文本字符串中的起始位置排序所得到的数组。Mazini等人改进了该算法的空间性能，但优化后的空间复杂度仍然是非线性的。其后，Kärkkäinen等人提出一种二步计算最长公共前缀数组的方法，该方法首先计算一个称为“转置最长公共前缀数组”的中间量，然后将其在线性时间内转换为最长公共前缀数组。其中，转置最长公共前缀数组记录了文本中位置相邻的后缀之间的最长公共前缀。Kärkkäinen等人在该内存构造算法的基础上又提出一种外存构造算法LCPscan，其空间复杂度与文本长度呈线性正比，但时间复杂度与可用内存的大小呈反比。从实验数据可知，当文本与可用内存之比变大时，LCPscan的性能衰减很快。对于第一类算法，Fischer采用归纳排序法提出了一种新的内存算法，该算法在计算后缀数组的过程中同步推导其最长公共前缀数组，时、空复杂度均为线性。Bingmann等人将该算法扩展到外存模型，时、空复杂度与外存整数排序相当，优于其他算法。现有多个研究工作致力于设计可在共享内存模型上运行的并行构造算法。这些算法充分利用多核CPU/GPU的计算能力，取得了较为理想的性能。

在实际应用中，输入文本一般由复数个较短的字符串相连而成。可设计算法构造广义后缀数组及其最长公共前缀数组。Baul等人设计的eGSA算法采用分支策略和多路合并思想计算一个字符串集的广义后缀数组和最长公共前缀数组。Bille等人设计的exLCP算法在外存模型上依次计算一个字符串集的广义Burrows-Wheeler变换数组和广义最长公共前缀数组。这两个算法的主要缺点是设计复杂，难以扩展到分布式计算模型上。一般而言，待匹配模式的长度不超过百个字符，这表明：可按需构造有限阶后缀数组和最长公共前缀数组，满足具体应用上下文中的模式匹配要求即可。第四章提出一种具有较好扩展性的有限阶最长公共前缀数组构造算法，可部署于内存、外存和对称型分布式模型。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表1-5 后缀数组构造算法研究现状 | | | | |
| 计算模型 | 经典算法 | 文本规模  （GB） | 构造速度  （MB/s） | 实现复杂度  （代码行） |
| 内存 | SA-IS |  | 8 |  |
| 外存 | DSA-IS |  | 0.5 |  |
| 共享内存 | DC3-GPU |  | 30 |  |
|  | | | | |

### 1.2.4新型后缀数组和最长公共前缀数组验证算法

待处理数据规模的爆发式增长推动了索引构造技术的进步。以后缀数组构造算法为例，如表1-5所示，最早的单机内存算法可处理的文本规模为GB级，构造速度接近10MB/s。与之相比，单机外存算法可处理的文本规模是单机内存算法的百倍以上，但构造速度较后者慢数十倍；而采用GPU技术的共享内存算法在处理速度上较单机内存算法更有优势，但前者的可处理文本规模较小。从实现复杂度来看，单机内存算法的程序较为简单，只需百行代码；而外存和共享内存算法的程序较为复杂，均包含数千行代码。目前，SA-IS、DSA-IS和DC3-GPU的作者虽然向第三方提供开源的算法包，但均不保证内附程序的正确性。因此，这些算法包中均附带一个验证程序。除暴力验证外，Burkhardt 等人提出的算法是目前唯一已知的后缀数组验证算法。该算法是一种事后验证算法，可检测任意后缀数组构造算法的输出。第五章提出两个同时验证后缀数组和最长公共前缀数组的算法，算法可检测任意后缀数组构造算法和最长公共前缀数组构造算法的输出，验证结果大概率正确，在外存模型上的时、空复杂度与整数排序相当。目前，尚无文献记载最长公共前缀数组验证算法，此工作填补了该研究空白。

事后验证算法可验证任意构造算法的计算结果，通用性好。然而，若能在构造过程中同时验证构造结果，则可大幅提高验证速度。第六章提出两个后缀数组事中验证算法，算法需与采用IS方法设计的后缀数组构造算法联用，同步构造和验证后缀数组。其中，第一个算法是概率性算法，可将其进一步扩展来验证构造中的最长公共前缀数组；第二个算法是确定性算法，仅可验证后缀数组。

# 第二章 流水路由查找算法

## 2.1预备知识

当数据包到达路由的一个输入端口时，其依次经历两个阶段：路由阶段和交换阶段。在路由阶段，路由系统首先从数据包中提取目的地址并在本地路由表中查找与该地址匹配的最长前缀，然后路由系统获取最长前缀对应的路由信息并从路由信息中提取输出端口。在交换阶段，路由系统将数据包从输入端口调度到输出端口并尽快安排数据包从输出端口转发出去。自互联网诞生以来，网络规模和传输的数据类型变化很大，骨干网的结点负载呈几何级数上升。为保证服务质量，要求设计高效的路由系统，满足日益增长的吞吐量需求。

过去20年，前缀树索引是解决路由查找问题的关键数据结构。可将路由表中的IP地址前缀视为一个字符串集合并使用一个前缀树构造算法来构造该集合的前缀树索引，树中各结点包含0、1或多条路由信息，结点之间以指针相连。路由系统为每个数据包创建一个查找请求，该请求按如下过程在前缀树索引上查找与包内目的地址匹配的最长IP地址前缀：将目的地址作为分支判定条件从当前访问的前缀树结点下行至某个孩子结点；从根结点开始递归执行该过程直到到达某个叶子结点或满足特定条件。已有多个研究工作提出了采用前缀树索引的路由查找算法。作为示例，第一章的图1-2和1-3分别给出了表1-4中的简单路由表的binary tree和fixed-stride tree。

在前缀树索引上执行并行查找可以大幅提升查找效率。例如：可将前缀树索引存储到由多个存储块构成的流水系统中，当多个查找请求访问位于不同存储块中的结点时，这些访问操作可以并行执行。理论上，目下流行的线型和环型流水路由查找系统可在每个系统时间片完成一个查找请求。文献提出了采用fixed-stride tree的线型流水路由查找系统，该系统将前缀树的根结点保存于初始存储块并将后继结点按层保存于后继存储块，使得同层结点位于同一个存储块且相邻层结点位于相邻存储块。该线型流水路由查找系统的缺点在于存储块的存储负载和工作负载不均衡，主要原因是路由表的IP地址前缀的长度不均。为此，文献提出了一种空间节省的环型流水路由查找系统，该系统首先将前缀树划分为一棵主树和多棵辅树，主树包含前缀树的上层结点，每棵辅树包含前缀树的部分下层结点且辅树的根结点为主树的一个叶子结点。如图2-1所示，图1-3中的binary tree被划分为一棵主树和三棵辅树，主树包括原树的第0-2层结点，而辅树包括原树的第2-5层结点。所述环型流水路由查找系统将图2-1中的前缀树索引按如下方式映射到存储块中：

* 主树的存储方式：将主树中的结点存储到一张哈希表中，表中各项以IP地址前缀为键并以路由信息和指向一棵辅树的指针为值，指针可以为空。
* 辅树的存储方式：以最小化存储块的空间需求为目标函数，采用动态规划方法确定各辅树的根结点所在的存储块，根结点的子孙结点按分层存储的方式依次存储到后续的存储块中。

在环型流水路由系统中，一个查找请求首先访问哈希表来获取其中的最长前缀匹配，若查找得到的表项中的辅树指针不为空则继续访问指针指向的辅树来获取其中的最长前缀匹配。在建立索引时，环型流水路由查找系统中的各存储块的存储负载和工作负载可达到平衡。但在长期路由更新后，系统可能发生失衡。

除上述工作外，近年的一些研究成果首先将路由表中的前缀划分为多个不相关子集，然后为各子集分别构建前缀树索引并将前缀树中的结点映射到一个线型或环型流水路由查找系统中。因为不同子集上的路由查找请求可并行执行，所以系统的吞吐量得到了进一步的提升。

|  |
| --- |
| 图2-1 主树与辅树的划分示例.png |
| 图2-1 主树和辅树的划分示例 |

采用前缀树索引的多存储块路由查找系统需支持两个基本功能：索引查找和索引动态更新。如前文所述，路由表中的前缀分布不均衡和长期动态更新均可能导致各存储块的工作、存储负载不均衡。为解决该问题，本章提出一种新型流水路由查找系统，该系统采用随机分布策略将前缀树中的结点映射到各存储块中。实验结果表明，在系统初始化和长期动态更新后，整个系统均保持在平衡状态。

## 2.2索引结构

所述路由查找系统改造任意前缀树索引来组织路由表中的IP地址前缀，改造后的索引包含一个快表和一个前缀森林。具体而言，将路由表中的IP地址前缀按长度分为长、短两类：若长度超过L个比特则为长前缀，否则为短前缀。将所有长前缀按前L比特划分为个不相交的子集，采用一种前缀树构造算法将每个子集转化为一棵前缀树；将所有短前缀插入到一个包含个表项的快表中，表中各项的数据域包括：

* 扩展前缀：每个短前缀对应一个长度为L比特的扩展前缀。扩展前缀可由短前缀右侧补零得到。例如：若，则01\*和1\*的扩展前缀分别为010\*和100\*，而0\*和00\*的扩展前缀均为000\*。
* 掩码数组：至多L个不同的短前缀具有相同的扩展前缀。例如：若，则0\*、00\*和000\*的扩展前缀均为000\*且没有其他长度不为0的前缀的扩展前缀为000\*。使用一个长度为L的比特数组来记录这些前缀是否在路由表中出现。例如：若扩展前缀为000\*且其在快表中的掩码数组的前两比特均为1，则表示0\*和00\*均包括在路由表中而000\*不包括在路由表中。
* 路由信息数组：与掩码数组联用。数组中第i个元素记录掩码数组第i个短前缀所对应的路由信息。
* 前缀树指针：按划分规则可知，同一个子集中的长前缀的前L比特相等。前缀树指针指向前L比特的字面值等于扩展前缀的所有长前缀所构成的前缀树。

在索引中添加长度为*l*的前缀p的过程如下所述：若p为短前缀，则将其转化为扩展前缀并以扩展前缀为键查询快表得到表项，将该表项中的掩码数组和路由信息数组中第*l*个元素分别设置为1和p的路由信息；若p为长前缀，则以p的前*l*比特为键查询快表得到表项，获取该表项中存储的前缀树指针并调用前缀树插入函数将p插入到前缀树中。例如：若，表1-4中的简单路由表的索引快表和前缀森林分别如表2-1和图2-2所示。观察可知，0\*、00\*和000\*为短前缀，其扩展前缀均为000\*，所以它们被插入到以000\*为键的快表表项中，该表项的掩码数组设置为111，路由信息数组设置为(E1,E2,E4)。另一方面，1101\*为长前缀，其前3比特为110，所以它被插入到tree B中。

在索引中查找目的地址的最长前缀匹配的过程为：

1. 从目的地址中提取前L比特，查询快表中扩展前缀与提取的L比特相等的表项。
2. 查看表项的掩码数组并获知数组中值为1的最右元素的位置i。若数组中的元素均为0，则无匹配的短前缀，执行步骤4；否则，执行步骤3。
3. 查看表项的路由信息数组并提取位置为i的数组元素，该元素为与目的地址的最长短前缀所对应的路由信息。
4. 查看表项的前缀树指针。若指针非空，则执行步骤5；否则，返回步骤2的查找结果。
5. 调用前缀树查找函数在树中查找与目的地址匹配的最长长前缀。若存在，则返回匹配的长前缀所对应的路由信息；否则，返回步骤2的查找结果。

|  |  |  |  |
| --- | --- | --- | --- |
| 表2-1 快表示例 | | | |
| 扩展前缀 | 掩码数组 | 路由信息数组 | 前缀树指针 |
| 000\* | 111 | null | (E1,E2,E4) |
| 001\* | 000 | null | (-,-,-) |
| 010\* | 011 | to trie A | (-,E3,E5) |
| 011\* | 001 | null | (-,-,E6) |
| 100\* | 000 | null | (-,-,-) |
| 101\* | 000 | null | (-,-,-) |
| 110\* | 000 | to trie B | (-,-,E7) |
| 111\* | 000 | null | (-,-,-) |

|  |
| --- |
| 图2-2 前缀森林示例.png |
| 图2-2 前缀森林示例 |

## 2.3体系架构

如图2-3所示，所述路由系统的体系结构主要包括以下部件：（1）多个存储单元，用于存储第2.2节描述的索引；（2）一个缓冲单元，用于缓存处于路由阶段的数据包；（3）一个调度单元，用于控制数据包的查找请求的执行过程。其中，索引的快表被放置于一个独立的存储单元中，而前缀森林中的结点被放置于由组成的流水线中。为使的存储需求达到平衡，系统按如下方式将前缀森林中的结点映射到流水线中：对前缀森林中的每个结点，使用随机数生成装置从中等概率选取一个数并将结点存储到以该随机数作为下标的存储块中。分析可知，若结点总数为N，则每个存储块分配到的结点个数随着N的增大而无限趋近于。

|  |
| --- |
| 图2-3 体系结构.png |
| 图2-3 体系结构 |

在图2-3的体系结构中，系统首先在快表中查找到包的最长前缀匹配并判断是否需要在前缀森林中继续查找。若否，则系统根据快表的查询结果将数据包交换至输出端口；否则，系统创建一个查找请求并将数据包连同查找请求缓存于缓冲单元，查找请求附带快表的查询结果和待访问前缀树的根指针信息。然后，调度单元授权查找请求遍历位于中的前缀树，多个查找请求可并行访问不同存储单元。区别于线型和环型流水路由查找系统，本章提出的流水路由查找系统将前缀树中的结点随机映射到存储单元中，导致父子结点可能位于不相邻的存储单元。在此情况下，一个查找请求需要跳跃地访问各存储单元来执行整个查找过程。此外，多个查找请求可能同时访问同一个存储单元。假设系统时间被切分为等跨度的时间片且每个存储块在单个时间片内仅允许被访问一次，则系统需预先为查找请求赋予优先级并由调度单元按优先级安排目标存储块相同的多个查找请求串行地执行访问操作，从而避免访问冲突。分析可知，路由系统的吞吐量取决于每个时间片内可无冲突地并行访问存储块的查找请求个数。我们在下一节中给出该问题的排队模型和解决方案。

## 2.4排队模型和调度算法

### 2.4.1排队模型

为了建立排队模型，我们对路由系统做出如下假设：

* 系统时间被切分为等跨度的时间片。
* 缓冲单元容量无限。
* 在每个时间片的开始，数据包到达路由的输入端口。
* 在每个时间片的末尾，调度单元从缓冲单元中选取一个可无冲突访问存储块的数据包集合并授权各数据包访问目标存储块。

为描述方便，我们在下文中引入参数N来表示缓冲单元中的数据包个数。图2-4展示了所提出的路由查找系统的排队模型。需要指出的是：每个数据包在前缀树中查找最长前缀匹配时需遍历一个或多个结点，这意味着每个数据包需要执行一次或多次访问操作。在排队模型中，若一个数据包在执行完一次访问操作后找到了最长前缀匹配，则系统将数据包从缓冲单元移至输出端口。

|  |
| --- |
| 图2-4 排队模型.png |
| 图2-4 排队模型 |

### 2.4.2调度算法

可将调度问题转化为二部图匹配问题：定义缓冲单元中的每个数据包和存储单元为二部图中的一个顶点，从数据包顶点到存储单元之间的一条边表示数据包请求访问存储块。调度的目标是在二部图中寻找一个最大边集，使得数据包结点或存储单元结点最多只有一条边与其相连。在二部图中，若同时有多条边与一个存储单元结点相连，调度单元需要根据与各边相连的数据包的优先级从中选取一条。常用的优先级策略包括：先来先服务和轮询等。因为每个数据包在寻找最长前缀匹配时执行一个或多个查找步且每个查找步需访问一次存储单元，我们将一个查找请求抽象为一个t元组，t为查找步数，si为第i次查找步访问的存储单元。例如：五元组表示数据包需在第1、2、3、4和5查找步依次访问m1、m2、m5、m4和m3。据此，我们给出一个三步调度算法框架：

1. 缓冲单元中的每个数据包发送一个访问请求至目标存储单元。
2. 每个存储单元按数据包的优先级选取接收到的一个访问请求并通知调度单元授权对应的数据包访问自身。
3. 每个得到授权的数据包对目存储单元执行一次访问操作。

|  |  |  |
| --- | --- | --- |
| 表2-2 先来先服务策略下的调度实例 | | |
| 初始状态( | | |
| 数据包 | 到达时间片 | 查找元组 |
| P1 | 0 |  |
| P2 | 1 |  |
| P3 | 2 |  |
| P4 | 3 |  |
| P5 | 4 |  |
| P6 | 5 |  |
| P7 | 6 |  |
| P8 | 7 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 表2-3 先来先服务策略下各时间片内的最大边集 | | | |
| 时间片 | 最大边集 | 时间片 | 最大边集 |
| 0 |  | 7 |  |
| 1 |  | 8 |  |
| 2 |  | 9 |  |
| 3 |  | 10 |  |
| 4 |  | 11 |  |
| 5 |  | 12 |  |
| 6 |  |  |  |

表2-2给出了一个采用先来先服务优先级策略的调度实例，调度单元在各时间片得到的最大边集如表2-3所示。在该实例中，系统根据到达时间赋予数据包优先级且先到的数据包优先级高。假设流水系统中的存储块个数为4且缓冲单元的初始状态为空，数据包P1 - P8分别在时间片0-7到达路由且各数据包的查找元组分别如R1-R8所示。在时间片0，R1请求访问存储单元m1。因为缓冲单元中仅有一个数据包，R1获得授权。从而，时间片0的最大边集为。在时间片1，R2参与到调度中来，其请求访问存储单元m1。同一时刻，R1请求访问m2。因为R1和R2的目标存储单元不同，所以它们都得到授权。从而，时间片1的最大边集为。在时间片2，R1和R2同时访问m3而R3访问m1。根据先来先服务的优先级策略，R1具有优先访问的权利并因此得到授权。从而，时间片2的最大边集为。按上述过程继续执行可得时间片3-12的调度结果。

### 2.4.3理论分析

在每个时间片，调度单元构造最大边集并执行调度作业。流水路由查找系统的吞吐量与最大边集的平均值成正比。令表示N个查找请求访问某个存储单元的概率，我们分析应变量与自变量和之间的变化关系。因为前缀森林中的所有结点被等概率分布到各存储单元中，所以任意数据包访问任意存储单元的概率。根据事件独立性，我们可以得到。图2-5展示了的变化趋势图，其中。观察可知，虽然的值随着M的增大而减小，但系统的总吞吐量仍然在不断上升。更具体地，由系统吞吐量的计算公式可知，我们能够通过增大M和/或N来提高系统吞吐量。

|  |
| --- |
| 图2-5 P(M,N)的变化趋势图.png |
| 图2-5 的变化趋势图 |

## 2.5实验分析

### 2.5.1实验环境

在所述流水路由查找系统中，影响系统吞吐量的关键因素是在每个时间片尽可能调度更多的数据包访问不同的存储块。在本节的仿真实验中，我们假设缓冲单元中的数据包按到达顺序排列且所有数据包的查找请求均被允许在每个时间片竞争存储单元的访问授权。在每个时间片，调度单元根据先来先服务的优先级策略在数据包和存储单元所构成的“请求-访问-授权”二部图中寻找最大边集，得到授权的数据包从目标存储单元中获取前缀树结点的信息。假设所有数据包和存储单元在每个时间片的开始均处于未匹配状态，我们采用先来先服务的优先级策略实例化2.3节中的调度算法框架：

* 请求：每个未匹配的数据包发送一个访问请求至目标存储单元。
* 授权：每个未匹配的存储单元从接收到的访问请求中选取“年龄最大的”数据包并通知调度单元授权访问。
* 访问：若一个未匹配的数据包获得授权，则其访问目标存储单元。

为便于描述，我们将本章提出的新型流水路由查找系统称为随机型。下文通过仿真实验评估随机型的性能，选取的指标为单个存储单元的存储需求和工作负载。为进一步分析，我们将随机型的仿真结果与线型和环型的仿真结果进行对比。其中，线型、环型和随机型均采用第2.2节中的索引结构组织路由表，但它们采用不同的映射方式将索引结构中的前缀森林存储到多个存储单元中。线型、环型和随机型的特点如下：

* 线型：简记为LP（linear pipeline）。每棵前缀树的根结点位于初始存储单元，子孙结点按层存储到后继存储单元。每个存储单元放置一层结点，因此存储单元的个数不少于前缀树的高度。数据包的查找请求从初始存储单元开始向右依次访问各存储单元。
* 环型：简记为CP（circular pipeline）。CP是已知工作的改进，其使用动态规划方法确定每棵前缀树的根结点所在存储单元并按层存储子孙结点到后继存储单元。动态规划的目标是平衡存储单元的存储需求。数据包的查找请求从根结点所在的存储单元开始（循环）向右访问各存储单元。
* 随机型：简记为RP（random pipeline）。采用随机策略将每个结点等概率存储到一个存储单元中。数据包的查找请求与存储单元之间的“请求-访问-授权”过程被转化为二部图匹配。数据包的查找请求跳跃地访问各存储单元。

常见的前缀树路由查找算法的查找过程是从根结点出发至叶子结点结束。这类算法包括BT、PT和FST等，可在LP、CP和RP中使用。然而，类似MPT的算法，其上的查找过程可能在访问子结点后回访父结点，因此只能在RP中使用。这体现了RP具有更好的通用性。在下文中，我们分别改造BT、PT、FST和MPT算法得到BT-based、PT-based、FST-based和MPT-based算法来构造第2.2节中的索引。

### 2.5.2存储需求评估

我们首先评估流水路由查找系统在初始构造索引时的存储开销。不失一般性，假设每个存储单元的容量相等，流水路由查找系统的总空间需求等于存储单元的容量与个数的乘积。实验首先测量系统构造初始索引时的存储开销。令长、短前缀的分类阀值，表2-4和2-5分别展示了LP、CP和RP存储IPv4和IPv6路由表的开销，路由表均从http://www.ris.ripe.net下载。表中存储块个数M由公式确定，C为IP地址长度。观察可知：

* 当索引IPv4路由表时，LP的存储需求不平衡，CP和RP的存储需求保持平衡。LP的单个存储块的最大和最小开销相差千倍。例如：当存储rrc0的IPv4路由表时，单个存储块的开销在5.32E-04到2.41E-01之间波动。另一方面，CP和RP的单个存储块的开销在4.35E-02上小幅波动。
* 当索引IPv6路由表时，LP和CP的存储需求不平衡，CP较LP表现更好。与CP和LP相比，RP的存储需求较为平衡。例如：当存储rrc0的IPv6路由表时，CP的单个存储块的开销在3.16E-05和7.27E-02之间波动，而RP的单个存储块的开销在7.89E-03和8.97E-03之间波动。

上述实验结果可解释如下：IPv6路由表的前缀长度分布不均，构造得到的索引中的前缀树大小和结构相差较大。因为LP和CP采用“层到块”的映射方案，两者均面临存储需求失衡的问题。另一方面，RP采用随机分布方案存储前缀树的结点，这使得其对前缀长度的分布和前缀树结构不敏感，从而取得较LP和CP更好的性能。

这里再次指出：LP中存储块的个数不能少于索引中前缀树的最大高度，而CP和RP不受此约束的限制。有鉴于此，我们进一步评估16阶CP和RP在的情况下的存储空间利用率，实验采用BT-based、PT-based和FST-based算法构建索引结构，测量每个存储块的存储开销占总开销的比例，结果如图2-6至2-8所示。该实验结果与表2-4和2-5中的测试数据相符。

BT、PT和FST上的最长前缀匹配过程相似：查找请求从根结点开始遍历前缀树直到遇到一个叶子结点结束，位于根结点到叶子结点路径上的所有结点均被访问一次。MPT上的最长前缀匹配过程与之不同：查找请求在访问一个内部结点的子孙结点后可能再次回访该内部结点。从结点存储方式和在其上执行查找请求的过程可知，LP和CP均不支持MPT-based算法构造的索引，而RP支持。图2-9展示了16阶RP存储MPT-based算法构建的索引的空间利用率，路由表前缀按分类。观察可知，RP的单个存储块的空间开销依然保持平衡。

索引结构可能在长期增量更新后发生变化。为评估增量更新对LP、CP和RP的存储需求的影响，我们对采用BT-based算法构造的IPv4和IPv6路由表执行一个月的增量更新操作。实验结果如表2-6和2-7所示，总结如下：

* 当执行完IPv4路由更新后，LP、CP和RP的存储开销未发生显著变化。这是因为初始的IPv4路由表相对稳定，后续的更新操作通常是由于短期内的链接的反复失效和恢复所引起，这导致在前缀树上不断地添加和删除相同结点。
* 当执行完IPv6路由更新后，RP的存储开销进一步趋于平衡。从长期统计的角度来看，当前缀树结点的数量增加时，每个存储块分配到的结点个数愈发趋近于结点总数与存储块个数的商。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表2-4 LP、CP和RP存储IPv4路由表的BT-based索引的存储开销 | | | | |
| LP | IPv4 (M=23) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 5.32E-04 | 2.41E-01 | 5.32E-04 | 2.41E-01 |
| rrc01 | 1.99E-05 | 2.43E-01 | 1.91E-05 | 2.43E-01 |
| rrc03 | 5.52E-04 | 2.41E-01 | 6.67E-04 | 2.19E-01 |
| rrc04 | 5.51E-04 | 2.42E-01 | 5.47E-04 | 2.42E-01 |
| rrc05 | 2.17E-04 | 2.42E-01 | 2.29E-04 | 2.42E-01 |
| rrc06 | 0 | 2.43E-01 | 0 | 2.44E-01 |
| rrc07 | 2.68E-04 | 2.43E-01 | 3.15E-04 | 2.43E-01 |
| rrc10 | 5.48E-04 | 2.42E-01 | 5.44E-04 | 2.41E-01 |
| rrc11 | 3.34E-04 | 2.42E-01 | 2.82E-04 | 2.42E-01 |
| RP | IPv4 (M=23) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 4.32E-02 | 4.39E-02 | 4.31E-02 | 4.38E-02 |
| rrc01 | 4.32E-02 | 4.37E-02 | 4.32E-02 | 4.37E-02 |
| rrc03 | 4.31E-02 | 4.39E-02 | 4.32E-02 | 4.38E-02 |
| rrc04 | 4.31E-02 | 4.40E-02 | 4.31E-02 | 4.40E-02 |
| rrc05 | 4.32E-02 | 4.38E-02 | 4.32E-02 | 4.37E-02 |
| rrc06 | 4.30E-02 | 4.39E-02 | 4.31E-02 | 4.38E-02 |
| rrc07 | 4.31E-02 | 4.39E-02 | 4.32E-02 | 4.39E-02 |
| rrc10 | 4.31E-02 | 4.38E-02 | 4.31E-02 | 4.38E-02 |
| rrc11 | 4.32E-02 | 4.37E-02 | 4.32E-02 | 4.38E-02 |
| CP | IPv4 (M=23) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 4.35E-02 | 4.35E-02 | 4.34E-02 | 4.36E-02 |
| rrc01 | 4.35E-02 | 4.35E-02 | 4.32E-02 | 4.38E-02 |
| rrc03 | 4.35E-02 | 4.35E-02 | 4.13E-02 | 4.74E-02 |
| rrc04 | 4.35E-02 | 4.35E-02 | 4.27E-02 | 4.41E-02 |
| rrc05 | 4.35E-02 | 4.35E-02 | 4.32E-02 | 4.37E-02 |
| rrc06 | 4.35E-02 | 4.35E-02 | 4.32E-02 | 4.39E-02 |
| rrc07 | 4.35E-02 | 4.35E-02 | 4.33E-02 | 4.37E-02 |
| rrc10 | 4.35E-02 | 4.35E-02 | 4.29E-02 | 4.38E-02 |
| rrc11 | 4.35E-02 | 4.35E-02 | 4.28E-02 | 4.39E-02 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表2-5 LP、CP和RP存储IPv6路由表的BT-based索引的存储开销 | | | | |
| LP | IPv6(M=119) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 3.16E-05 | 7.27E-02 | 3.63E-05 | 7.32E-02 |
| rrc01 | 3.22E-05 | 7.37E-02 | 3.25E-05 | 7.58E-02 |
| rrc03 | 2.96E-05 | 6.73E-02 | 3.66E-05 | 6.36E-02 |
| rrc04 | 2.98E-05 | 6.78E-02 | 3.25E-05 | 7.50E-02 |
| rrc05 | 3.25E-05 | 7.43E-02 | 3.26E-05 | 7.64E-02 |
| rrc06 | 0 | 7.83E-02 | 0 | 7.92E-02 |
| rrc07 | 3.31E-05 | 7.53E-02 | 3.28E-05 | 7.65E-02 |
| rrc10 | 1.47E-05 | 7.67E-02 | 1.46E-05 | 7.81E-02 |
| rrc11 | 3.28E-05 | 7.37E-02 | 3.31E-05 | 7.59E-02 |
| RP | IPv6(M=119) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 7.89E-03 | 8.97E-03 | 7.94E-03 | 9.01E-03 |
| rrc01 | 7.93E-03 | 9.05E-03 | 7.89E-03 | 8.85E-03 |
| rrc03 | 7.83E-03 | 8.94E-03 | 7.98E-03 | 8.91E-03 |
| rrc04 | 7.86E-03 | 8.89E-03 | 7.97E-03 | 8.85E-03 |
| rrc05 | 7.82E-03 | 8.84E-03 | 7.81E-03 | 8.97E-03 |
| rrc06 | 7.95E-03 | 8.98E-03 | 7.96E-03 | 8.90E-03 |
| rrc07 | 7.79E-03 | 8.99E-03 | 7.83E-03 | 8.90E-03 |
| rrc10 | 7.92E-03 | 8.87E-03 | 7.92E-03 | 8.87E-0 |
| rrc11 | 8.01E-03 | 8.90E-03 | 7.70E-03 | 8.86E-03 |
| CP | IPv6(M=119) | | | |
| 表 | 更新前 | | 更新后 | |
|  | 最小值 | 最大值 | 最小值 | 最大值 |
| rrc00 | 1.19E-03 | 2.36E-02 | 1.18E-03 | 2.36E-02 |
| rrc01 | 1.13E-03 | 2.49E-02 | 1.06E-03 | 2.61E-02 |
| rrc03 | 1.22E-03 | 2.25E-02 | 1.21E-03 | 2.06E-02 |
| rrc04 | 1.16E-03 | 2.27E-02 | 1.00E-03 | 2.45E-02 |
| rrc05 | 1.36E-03 | 2.42E-02 | 1.21E-03 | 2.44E-02 |
| rrc06 | 9.72E-05 | 2.61E-02 | 9.59E-05 | 2.60E-02 |
| rrc07 | 1.10E-03 | 2.52E-02 | 1.05E-03 | 2.62E-02 |
| rrc10 | 3.82E-04 | 2.53E-02 | 3.71E-04 | 2.56E-02 |
| rrc11 | 1.20E-03 | 2.52E-02 | 9.36E-04 | 2.65E-02 |

|  |
| --- |
|  |
| 图2-6 CP和RP存储路由表的单个存储块存储开销，索引由BT-based算法构造 |

|  |
| --- |
|  |
| 图2-7 CP和RP存储IPv6路由表的单个存储块存储开销，索引由PT-based算法构造 |

|  |
| --- |
|  |
| 图2-8 CP和RP存储IPv6路由表的单个存储块存储开销，索引由FST-based算法构造 |

|  |
| --- |
| 图2-9 CP和RP的初始空间利用率_MPT.png |
| 图2-9 CP和RP存储IPv6路由表的单个存储块存储开销，索引由MPT-based算法构造 |

### 2.5.3吞吐量评估

在每个时间片，LP允许最多一个查找请求进入流水，而CP和RP允许多个查找请求进入流水。在LP和CP中，进入流水的查找请求使用一个或多个连续的时间片访问位置相邻的一个或多个存储块。在RP中，进入流水的查找请求首先使用一个时间片访问一个存储块，然后退出流水并等待下次允入时机的到来。由于LP的空间利用率较低，本节仅考察CP和RP在IPv4和IPv6数据流下的查找吞吐量，实验引入如下参数符号：

* S：单个数据包的查找请求执行的查找步数，平均值。每步访问一次存储块，每次访问读取一个结点信息。
* B：单个时间片内到达路由输入端口的数据包个数，平均值。到包的概率服从参数为λ的伯努利分布。
* W：单个流水阶的工作负载，平均值。计算公式为。
* Q：单个时间片内缓冲单元中数据包的最大个数。CP和RP均采用缓冲单元缓存尚未找到最长前缀匹配的数据包。其中，CP为每个存储块配备一个缓冲单元，RP为所有存储块提供一个共享的缓冲单元。实验监视所有缓冲单元中的数据包总数来计算CP的Q值。

实验评估的性能指标为Q与W的变化关系，每个实验数据为执行两次相同实验所得结果的平均值，每次实验的输入为100万个数据包，每个数据包的目的地址从rrc00的IPv4/IPv6路由表中随机抽取。已知且，图2-10至2-12和2-13至2-15分别展示了CP和RP处理IPv4和IPv6数据流的查找性能，结论如下：

* 在W较小时，各曲线随W的增长而平滑上扬；当W接近临界点时，系统发生过载，各曲线上扬趋势陡增。观察图2-10可知，当时，CP和RP处理IPv4数据流的工作负载的临界值W’均为0.85。
* 令W固定不变，当CP和RP处理IPv4和IPv6数据流时，Q随B的增大而增大。与此同时，工作负载临界值W’未随B的变化而变化。
* 当处理IPv4数据流时，RP和CP的查找吞吐量相近，但当处理IPv6数据流时，后者的最大工作负载较前者高出约10个百分点。观察图2-13可知，CP和RP的负载临界值W’分别为0.85和0.75。类似现象也出现在图2-14和2-15中。

为了实验的完整性，我们在图2-16和2-17中给出了RP采用MPT-based算法构造索引时的查找性能，结果与图2-10至2-15一致。

本节剩余部分讨论增量更新对系统的查找吞吐量的影响。一个增量更新操作可能添加、修改或删除一个IP地址前缀及其路由信息。当在索引中插入或删除一条短前缀时，可在常量时间内修改快表中的对应表项。例如：当从表2-1中删除000\*时，读取000\*对应的表项，将表项中的掩码数组和路由信息数组中的第三个元素清零。当在索引中插入或删除一条长前缀时，系统需要调用前缀树路由算法的更新函数在前缀森林中添加或删除结点。现有文献提供了一个可用于LP和CP的更新方法，该方法使用一个预处理装置预先计算每个更新操作对前缀森林结构的修改，然后将多个可并行执行的更新操作打包为一个“写泡”，每个写泡等待系统授权并在授权后进入流水。在执行过程中，写泡依次访问相邻的存储块并完成其中的更新操作。可将写泡视为一个特殊的查找请求，当访问的存储块不同时，写泡与查找请求可并行执行，从而实现在线更新。

RP同样支持在线更新操作。可将用于更新的写操作划分为多个批处理作业，每个作业中的写操作所访问的目标存储单元两两不同，从而单个批处理作业可在一个时间片内完成。系统为每个批处理作业创建一个更新请求并将其与系统内的查找请求一同处理。为保证数据一致性，每个更新请求的优先级低于当前执行中的查找请求并高于尚未执行的查找请求。因为更新请求的数量远小于查找请求的数量，我们可以合理地假设：更新操作不会对系统的查找吞吐量产生显著的副作用。

|  |
| --- |
| 图2-10 CP和RP的查找吞吐量_BT_IPV4.png |
| 图2-10 CP和RP处理IPv4数据流时的查找吞吐量，索引由BT-based算法构造 |

|  |
| --- |
| 图2-11 CP和RP的查找吞吐量_PT_IPV4.png |
| 图2-11 CP和RP处理IPv4数据流时的查找吞吐量，索引由PT-based算法构造 |

|  |
| --- |
| 图2-12 CP和RP的查找吞吐量_FST_IPV4.png |
| 图2-12 CP和RP处理IPv4数据流时的查找吞吐量，索引由FST-based算法构造 |

|  |
| --- |
| 图2-14 CP和RP的查找吞吐量_BT_IPV6.png |
| 图2-13 CP和RP处理IPv6数据流时的查找吞吐量，索引由BT-based算法构造 |

|  |
| --- |
| 图2-14 CP和RP的查找吞吐量_PT_IPV6.png |
| 图2-14 CP和RP处理IPv6数据流时的查找吞吐量，索引由PT-based算法构造 |

|  |
| --- |
| 图2-15 CP和RP的查找吞吐量_FST_IPV6.png |
| 图2-15 CP和RP处理IPv6数据流时的查找吞吐量，索引由FST-based算法构造 |

|  |
| --- |
| 图2-16 RP的查找吞吐量_MPT_IPV4.png |
| 图2-16 RP处理IPv4数据流时的查找吞吐量，索引由MPT-based算法构造 |

|  |
| --- |
| 图2-17 RP的查找吞吐量_MPT_IPV6.png |
| 图2-17 RP处理IPv6数据流时的查找吞吐量，索引由MPT-based算法构造 |

## 2.6硬件设计

采用先来先服务的优先级策略的调度算法框架已被用于解决数据包在交换阶段的输出端口冲突问题。这体现了所提出的流水路由查找系统的路由阶段和交换阶段的共性特征。本节参考交换机的仲裁单元的硬件实现来设计所提出的流水路由查找系统的调度单元。根据相关文献的描述，我们在图2-18展示了调度单元的参考硬件设计，调度单元采用先来先服务的优先级策略控制存储块上的访问操作。该调度单元包括M个授权仲裁单元、H个接受仲裁单元和一个用于缓存仲裁结果的决策寄存器。每个授权仲裁单元对应一个存储块，每个接受仲裁单元对应一个缓冲单元中的数据包。可用H个向量表示缓冲单元E的状态，每个向量包含M个比特且各比特的值表示缓冲单元中的数据包是否要求访问特定的存储块。例如：已知，则表示缓冲单元中的第i个数据包请求访问。令和分别为授权仲裁单元和接受仲裁单元，则系统在每个时间片执行以下三步来调度数据包访问存储块：

* 请求：若，则数据包i尚未匹配。若，则将其发送至授权仲裁单元。
* 授权：每个尚未匹配的授权仲裁单元可能接收到最多H个来自不同数据包的请求比特，其采用先来先服务的优先级策略从中选取一个。若来自的请求被选中，则将访问授权发送给接受仲裁单元。
* 接受：每个尚未匹配的接受仲裁单元可能接受到零个或者一个授权比特。其接受授权并将结果存入决策寄存器。

该调度单元的硬件复杂度为。从本章的实验结果可知，若，则系统在时达到临界工作负载。

|  |
| --- |
|  |
| 图2-18 调度单元的参考硬件设计 |

## 2.7本章小结

本章提出一种基于前缀树索引的流水路由查找系统。系统通用性好，可采用任意前缀树路由算法组织路由表中的IP地址前缀。系统中包含复数个存储块并允许多个查找请求并行访问不同的存储块。系统通过一个调度单元控制查找请求与存储块之间的“请求-授权-访问”操作。由理论分析和实验结果，我们总结RP、LP和CP各自的优劣如下：

* + RP提供一个通用的、可扩展的路由查找方案，方案可流水并行化在任意前缀树上执行的多个查找请求。
  + RP中的调度问题被建模为二部图匹配问题，每个查找请求在执行过程中被多次调度，每次调度时执行一个查找步，每个查找步访问一次存储块，连续执行的查找步可能访问不相邻的存储块。这导致数据包的处理延迟难以预知。与之相比，LP和CP中的每个查找请求仅被调度一次，数据包的处理延迟可以预知。
  + 一般而言，流水路由系统的电量开销与存储块的容量成正比。因为RP的存储需求在初始和长期增量更新后均保持平衡，每个存储块的预估电量开销较优。

目前，我们正以本章的研究工作为基础，开发一个实际可用的高吞吐量流水路由查找系统。

# 第三章 后缀数组构造算法

## 3.1预备知识

已知字符串的后缀树，可在的时间内确定给定模式是否出现在字符串中，m为模式长度。后缀树的空间开销线性正比于字符串的长度，每个结点维持一个数据域、多个指针域和后缀链，常数因子较大。后缀数组记录字符串中所有后缀按字典序排序的结果，是后缀树的简洁替代。结合最长公共前缀数组，后缀数组可模拟后缀树上的（自顶向下或自底向上）遍历过程。目前，多个算法可在内存模型上使用线性时、空构造后缀数组。其中，采用归纳排序（Induce Sorting, IS）方法设计的SA-IS算法常数因子最小、实际性能最优。部分研究工作在外存模型上改造SA-IS算法，使其能够处理更大规模的输入文本。已知的采用IS方法设计的外存后缀数组构造算法包括eSAIS、DSA-IS和SAIS-PQ。这些由IS方法设计的外存算法的时、空复杂度优于其他外存算法，但三者的程序的磁盘峰值较大。例如：eSAIS、DSA-IS和SAIS-PQ的磁盘峰值分别为23n、18n和15n，而pSAscan的磁盘峰值为8n。该现象的主要原因在于工程实现不佳，可通过优化程序设计来改进性能。近期的研究成果表明：采用文件块链技术可大幅提升上述三个算法的磁盘峰值。本章介绍DSA-IS算法的设计原理并给出一系列优化方法来改善原算法的设计与实现。为描述方便，我们引入以下名词术语和参数符号：

* 字符、后缀和子串。已知长度为n的字符串x，将后缀记作，将子串记作。
* L/S/L\*/S\*类型的字符、后缀和子串。将字符串x中的所有字符分为L和S两类。其中，当满足以下任一条件时，为L类型：
  + ；
  + ；
  + 且为L类型。

特别地，当和分别为S和L类型时，同时也是一个S\*类型字符。类似地，当和分别为L和S类型时，同时也是一个L\*类型字符。一个子串或后缀的类型与其首字符的类型相同。若无特别声明，后文所指的L/S/S\*类型子串仅包含首字符为L/S/S\*类型、尾字符为S\*类型且中间字符无S\*类型的子串。

* 后缀桶和最长公共前缀桶。令sa为x的后缀数组，sa中的后缀可被划分为多个后缀桶，每个桶占据数组的一个连续区间且桶中的所有后缀的首字符相同。进一步地，每个桶可被分为左、右两个子桶，左、右子桶分别包含L和S类型后缀。记以字符c为首字符的所有后缀所在的后缀桶为并记其左、右子桶为和。令lcp为x的最长公共前缀数组，按sa的划分将lcp划分为多个最长公共前缀桶。记所对应的最长公共前缀桶为，记录了中的后缀与其在sa中的前继后缀之间的最长公共前缀。类似地，我们定义中的左、右子桶和。
* S\*类型后缀的后缀数组和最长公共前缀数组。假设x中的S\*类型后缀的个数为n1，记录了所有S\*类型后缀按字典序排序的结果，记录了中相邻后缀的最长公共前缀。
* 类型数组。将x中所有字符的类型信息记录在数组t中，或1表示为L或S类型。
* 收缩字符串。可将x视为多个S\*类型子串的拼接，按字典序排序这些S\*类型子串并按排序结果对其从小到大命名。将x中的S\*类型子串替换为它们的名字可得收缩子串x1。
* 字符、子串和后缀的前继与后继。字符在x中的前继和后继分别为和。子串在x中的前继和后继分别为和。后缀在x中的前继和后继分别为和。

|  |  |
| --- | --- |
| 算法3-1 采用IS方法设计的后缀数组构造算法的框架 | |
|  | 输入：x |
|  | 输出：sa |
| 1 | /\* 收缩阶段 \*/ |
| 2 | 采用IS方法排序S\*类型子串。 |
| 3 | 按排序结果从小到大命名S\*类型子串，得到收缩子串。 |
| 4 |  |
| 5 | /\* 检测递归条件 \*/ |
| 6 | 若中包含重复出现的字符，则递归地对执行收缩阶段；否则，中的后缀的首字符两两不同，按首字符的字典序排序后缀得到。 |
| 7 |  |
| 8 | /\* 解推导阶段 \*/ |
| 9 | 采用IS方法排序所有后缀。 |

如算法3-1所示，采用IS方法设计的后缀数组构造算法主要包含两个阶段：收缩阶段和解推导阶段。在收缩阶段，排序和命名x中所有的S\*类型子串来计算x的收缩子串x1。若x1中存在同名字符，则递归执行收缩阶段，输入为x1；否则，直接计算x1的后缀数组sa1。在解推导阶段，首先从sa1计算得到所有S\*后缀的字典序，然后从S\*后缀的排序结果推出所有后缀的字典序。收缩阶段/解推导阶段的关键操作是从一个有序的子串/后缀集合推导出所有无序的子串/后缀的字典序，核心思想是通过两个子串或后缀的首字符和后继来确定它们的大小关系。

## 3.2 DSA-IS算法原理和设计

本节首先介绍DSA-IS算法的设计原理，然后提出多个改进设计与实现的方法。

### 3.2.1数据结构

DSA-IS算法是SA-IS算法的外存变种，其继承了SA-IS的线性空间复杂度，时间复杂度与外存整数排序相当。与eSAIS和SAIS-PQ相比，DSA-IS的算法设计最为简单。在构造后缀数组的过程中，SA-IS算法在执行收缩阶段和解推导阶段时需要非顺序地读写x和sa中的元素。在外存模型上，非顺序的读写操作的执行效率很低，若将SA-IS移植到外存，需将非顺序的内存访问操作转换为顺序的I/O读写操作，这就要求设计新的数据结构。回顾IS方法的核心思想：从子串/后缀的首字符和其有序后缀的字典序推出自身的字典序。换言之，在推导两个子串/后缀的字典序时，需要访问它们在x中的首字符和在sa中的有序后继的字典序信息。若能够预先将数据按其访问顺序保存在外存中，我们就能在需要时通过顺序I/O将其读入内存。为此，DSA-IS算法引入一个称为DSAItem的数据结构，该数据结构是一个包含以下数据域的元组：

* p：或的起始位置，为右侧最近S\*类型的字符的位置。
* c：或的首字符，即。
* r：或在所有后缀或子串中的相对字典序。
* t：的前继字符的类型，即。

每个DSAItem对应一个以p起始的子串或后缀，DSA-IS算法的输出为由n个DSAItem组成的数组DSA，数组中的元素按其对应的后缀的字典序排列。与SA-IS相同，DSA-IS主要包含收缩阶段和解推导阶段，后续两节分别描述两个阶段的设计细节。

### 3.2.2收缩阶段

在收缩阶段，SA-IS算法依次执行如下四步来推导所有S\*类型子串的字典序：

1. 从右向左遍历x并判断各字符的类型，将S\*类型字符插入到对应后缀桶的最右空位。
2. 清空各后缀桶的左子桶。从左向右遍历sa中的有序子串。若当前访问的子串在x中的前继为L类型，则将该L类型子串插入到对应后缀桶的最左空位。
3. 清空各后缀桶的右子桶。从右向左遍历sa中的有序子串。若当前访问的子串在x中的前继为S类型，则将该S类型子串插入到对应后缀桶的最右空位。
4. 从左向右遍历sa中的有序子串，按字面值判定相邻子串是否相同来对其命名。所得名字序列构成x的收缩子串。

图3-1展示了SA-IS算法的一个收缩阶段实例。

第2-6行执行步骤1。其中，第2-3行遍历x得到类型数组t并找出x中的所有S\*类型字符，第5-6行将S\*类型字符插入到对应的后缀桶的末尾空位。因为、和的首字符均为i，故将其依次插入到中。特别地，末尾的L类型字符没有后继字符，需在收缩阶段开始时将其插入到对应的后缀桶的起始空位。

第7-21行执行步骤2。具体而言，第8-9行找到所有后缀桶的最左位置（用符号标记）。第10-11行访问并查看其前继子串的首字符是否为L类型。因为且，所以将前继子串插入到的最左空位。第12-13行访问，因为元素为空，所以直接跳过。类似地，第14-17行依次跳过和。第18-19行访问并查看其前继子串的首字符是否为L类型。因为且，所以将前继子串插入到的最左空位。重复上述过程继续向右遍历x，最终可得第21行的排序结果。此时，所有L类型子串在sa中按字典序排列。

第22-30行执行步骤3。具体而言，第23-24行找到所有后缀桶的最右位置（用符号标记）。第25-26行访问并查看其前继子串的首字符是否为S类型。因为且，所以将前继子串插入到的最右空位。第27-28行访问并查看其前继子串的首字符是否为S类型。因为且，所以将前继子串插入到的最右空位。重复上述过程直到遍历完x中的字符，最终可得第30行的排序结果。此时，所有子串在sa中按字典序排列。

需要指出的是：第3行的按字典序存储所有S\*类型字符，每个字符作为一个特殊的S\*类型子串被用于推导L类型子串的字典序。第31行的按字典序存储S\*类型子串，步骤4依次遍历这些子串并比较它们的字面值来计算x的收缩子串。

|  |
| --- |
| 图3-1 SA-IS的收缩阶段实例.png |
| 图3-1 SA-IS的收缩阶段实例 |

DSA-IS算法按如下方法改造SA-IS的收缩阶段：

**第一步，划分x为多个块并在内存中分治排序各块的S\*类型子串。**

假设内存容量为M，将x划分为多个分块，每个分块包含一个或多个头尾相连的S\*类型子串且单个分块最多容纳m个字符，。按此方法得到的分块个数k不超过且每个分块包含一个或多个S\*类型子串。我们按S\*类型子串的个数将分块分为两类：

* 单子串块：块内仅含一个S\*类型子串。子串与左侧或右侧相邻的S\*类型子串的总长超过m，因此只能独立成块。特别地，块内子串的长度可能超过m。
* 多子串块：块内包含两个或多个位置相邻的S\*类型子串，块长不超过m。

对于多子串块，执行SA-IS的收缩阶段来排序块内S\*类型子串并按访问顺序将排序过程中读取的字符和已排序子串的字典序信息存储到外存文件中。同时，将排序后的S\*类型子串按排序结果保存到外存文件中。 对于单子串块，块内S\*子串自然有序，无需执行任何操作。

**第二步，合并分块的排序结果。**

该步骤依次执行以下三个子步骤：

1. 创建外存小顶堆，堆中元素类型为DSAItem，排序关键字为<c,t,r,p>。从右向左遍历x中的字符并判定字符类型，将S\*类型字符的DSAItem插入到小顶堆中。
2. 顺序取出小顶堆的堆顶元素并推导其前继子串。通过顺序I/O从外存中读取所需前继子串的首字符并判断首字符的类型。若首字符为L类型，则将该L类型子串的DSAItem插入到小顶堆中。在此过程中，将所有从小顶堆中取出的L类型子串的DSAItem插入到外存大顶堆中。当小顶堆为空时，所有L类型子串的DSAItem均在大顶堆中按字典序排列。
3. 顺序取出大顶堆的堆顶元素并推导其前继子串。通过顺序I/O从外存中读取所需前继子串的首字符并判断首字符的类型。若首字符为S类型，则将该S类型子串的DSAItem插入到大顶堆中。在此过程中，将所有从大顶堆中取出的子串的DSAItem顺序输出到一个外存数组中。当大顶堆为空时，所有子串的DSAItem在数组中按字典序排列。

在子步骤2，为获得待排序子串的首字符，首先根据DSAItem的数据域p确定子串所属分块，然后通过顺序I/O读入在第一步分块排序时保存至外存的首字符信息。

**第三步，命名排序后的子串。**

从左向右遍历外存数组中的S\*类型子串的DSAItem，按字面值判定相邻DSAItem所对应的子串是否相等以对子串命名。将子串的名字顺序存储到一个外存数组中，该数组即为字符串x的收缩子串。在此步骤，为获得待命名子串的字面值，首先根据DSAItem的数据域p确定子串所属分块，然后通过顺序I/O读入在第一步分块排序时保存至外存的子串字面值信息。

|  |
| --- |
| 图3-3 DSA-IS的收缩阶段实例_1.png |
| 图3-3 DSA-IS的收缩阶段中推导L类型子串的实例 |

图3-2展示了DSA-IS算法的收缩阶段实例，输入字符串x如图3-1所示。其中，和分别为大顶堆和小顶堆，排序关键字为。第1行将所有S\*类型子串的DSAItem插入到中，各DSAItem的数据域r初始化为定义域中的最大值max。第2-9行依次取出的堆顶元素并推导元素对应的子串的L类型前继。在第2行，当前堆顶元素为<且其对应的子串的前继为L类型，因此将前继的DSAItem插入pqmin并将该DSAItem的数据域r赋值为0，0表示该DSAItem是除S\*类型子串的DSAItem外首个插入到pqmin中元素。在第3行，当前堆顶元素为且其对应的子串的前继为L类型，因此将前继的DSAItem插入pqmin。在第6行，当前堆顶元素为且对应子串没有L类型前继，故无需插入新的元素至pqmin。又因为是一个L类型子串，所以将插入到中。重复上述过程直到pqmin为空，则所有L类型子串在中排列有序。

因为推导S类型子串的过程与推导L类型子串的过程相同，为避免重复，我们不再另附实例。

### 3.2.3解推导阶段

在解推导阶段，SA-IS算法依次执行如下四步来推导所有后缀的字典序：

1. 按的元素值排序元素下标得到x中S\*类型后缀的字典序。
2. 清空各后缀桶。按字典序从大到小遍历S\*类型后缀并将其依次插入到对应后缀桶的最右空位。
3. 清空各后缀桶的左子桶。从左向右遍历sa中的有序后缀。若当前访问的后缀在x中的前继为L类型，则将该L类型后缀插入到对应后缀桶的最左空位。
4. 清空各后缀桶的右子桶。从右向左遍历sa中的有序后缀。若当前访问的后缀在x中的前继为S类型，则将该S类型后缀插入到对应后缀桶的最右空位。

比对可知，解推导阶段的步骤2-4与收缩阶段的步骤1-3的逻辑相同，均是采用IS方法从一个有序的子集推导出全集的字典序。图3-3展示了解推导阶段步骤1的执行过程。其中，为收缩子串的后缀数组，为的逆且记录了字符串中各S\*类型后缀的字典序。可从计算得到x中各S\*类型后缀的后缀数组sa\*并执行步骤2-4来计算x的后缀数组。特别地，步骤2-4的执行过程与图3-1中的第4-30行相同。

|  |
| --- |
| 图3-2 SA-IS的解推导阶段实例.png |
| 图3-3 SA-IS的解推导阶段的实例 |

DSA-IS算法按如下方法改造SA-IS的解推导阶段：

**第一步，计算各分块中S\*类型后缀的字典序。**

记录x中所有S\*类型后缀的名字，按S\*类型后缀在x中的位置在外存中排序其名字，排序结果保存在外存文件中。从后向前遍历x中的后缀并判断其类型，将S\*类型后缀的名字按后缀所在的分块分组并保存在外存文件中，S\*类型后缀的名字可通过顺序I/O从外存文件中获取。

**第二步，在内存中分治排序各分块的所有后缀。**

对于多子串块，从外存读入其中S\*类型后缀的名字，调用SA-IS的解推导阶段从已排序的S\*类型后缀推导块内所有后缀的字典序。对于单子串块，块内后缀自然有序，无需执行任何操作。

**第三步，合并分块的排序结果。**

该步骤依次执行以下三个子步骤：

1. 创建外存小顶堆，堆中元素类型为DSAItem，排序关键字为<c,t,r,p>。从右向左遍历x中的后缀并判断后缀类型，将S\*类型后缀的DSAItem插入到小顶堆中。各DSAItem的数据域r记录了对应S\*类型后缀的名字，通过顺序I/O从外存中读取后缀的名字。
2. 顺序取出小顶堆的堆顶元素并推导其前继后缀。通过顺序I/O从外存中读取所需前继后缀的首字符并判断首字符的类型。若首字符为L类型，则将该L类型后缀的DSAItem插入到小顶堆中。在此过程中，将所有从小顶堆中取出的L类型后缀的DSAItem插入到外存大顶堆中。当小顶堆为空时，所有L类型后缀的DSAItem均在大顶堆中且按字典序排列。
3. 顺序取出大顶堆的堆顶元素并推导其前继后缀。通过顺序I/O从外存中读取所需前继后缀的首字符并判断首字符的类型。若首字符为S类型，则将该S类型后缀的DSAItem插入到大顶堆中。在此过程中，将所有从大顶堆中取出的后缀的DSAItem顺序输出到一个外存数组中。当大顶堆为空时，所有后缀的DSAItem在数组中按字典序排列，可按序输出DSAItem中的数据域p得到x的后缀数组。

在子步骤1，为获得S\*类型后缀的名字，首先根据DSAItem的数据域p确定后缀所属分块，然后通过顺序I/O读入在第一步按块分组存储的S\*类型后缀的名字。

DSA-IS的解推导阶段与收缩阶段的执行过程相似，为避免重复，我们不再给出解推导阶段的实例。下一节从设计和实现两个方面提出优化DSA-IS的方法。

## 3.3设计优化

### 3.3.1同步排序和命名子串

收缩阶段顺序执行排序和命名两个过程。为排序S\*类型子串，DSA-IS首先排序所有子串的DSAItem并记录其中S\*类型子串的DSAItem的排序结果。为命名S\*类型子串，DSA-IS首先按排序结果从小到大遍历S\*类型子串的DSAItem并从中获取数据域p的值，然后根据p的值从外存中读入子串的字面值并将其与前一个子串的字面值比较来确定自身的名字：若字面值相等，则当前读入的子串与前一个子串同名；否则，当前读入的子串的名字比前一个子串大1。

我们提出一种同步排序和命名子串的方法，将命名过程嵌入排序过程。该方法无需比较子串的字面值即可判断它们是否相等，其核心思想是：若两个子串的字典序相同，则它们的字面值也相同。在排序过程中，DSA-IS算法使用一个外存堆来排序子串的DSAItem，排序关键字为，r记录元素插入到堆中的顺序。

需要指出的是：元素插入到堆中的顺序代表该元素的对应子串的后继的字典序。因为子串的名字也代表其字典序，所以可用后继子串的名字来赋值DSAItem中的数据域r。这里的问题是：如何在插入一个DSAItem之前计算出该DSAItem所对应的子串的后继的名字。我们的解决方案如下：已知堆中任意两个子串的DSAItem为和，则当时，两个子串的字典序相同。换言之，若，则两个子串的名字也相同。因为字典序相同的子串必然在堆中相邻，算法只需比较当前堆顶元素与前一个取出的堆顶元素的即可计算当前堆顶元素对应的子串的名字。当名字确定后，算法推导堆顶元素对应的子串的前继。

### 3.3.2分治排序和命名子串

外存算法是I/O密集型算法，DSA-IS的I/O开销主要用于在外存中读、写和排序DSAItem。在真实数据集中，大部分S\*类型子串只包含几个字符，但其DSAItem的结构却较为臃肿，这导致读写DSAItem的开销远大于直接读写子串的开销。本节提出一种分类、排序和命名长、短子串的方法，该方法可降低收缩阶段的I/O量和磁盘峰值。

我们按预定义阀值D将字符串x中的所有子串划分为长（）、短（）两类并依次执行如下三步来排序和命名所有子串：

1. 重用DSA-IS算法的子串排序和命名方法来排序和命名所有长子串。可使用上一小节中提出的方法合并排序和命名过程。
2. 在分块后，以分块为单位调用SA-IS算法排序块内的所有短子串，按排序结果将短子串保存在外存文件中。
3. 在内存中创建一个小顶堆，堆中的每个元素是一个短子串或长子串的前D个字符，堆的排序关键字为子串字面值。用小顶堆合并步骤1得到的有序长子串和步骤2得到的各分块中的有序短子串，从而得到所有子串的有序序列。在合并的过程中对各子串命名。

在步骤3，小顶堆在任意时刻最多包含k个短子串和1个长子串的前缀，k为分块个数。因此，小顶堆所需内存空间为。在步骤3的开始，从长子串的有序序列和各分块的短子串的有序序列中分别读取最小子串并插入到小顶堆中，堆中子串按字面值比较。需要指出的是：一个长子串的前D个字符可能与一个短子串相等，在此情况下，前者的字典序小于后者。

在步骤3，假设a和b是两个从小顶堆中相继弹出的子串，则按如下方法在弹出b时确定b的名字：

* 若a和b不全为长子串，则按字面值比较a和b。若相等，则a和b同名；否则，b的名字比a的名字大1。
* 若a和b均为长子串，则比较它们在步骤1中计算得到的名字。若相等，则a和b同名；否则，b的名字比a的名字大1。

## 3.4实现优化

本节提出多个工程方法优化DSA-IS算法的实现，部分方法已固化在最新版的DSA-IS程序中。

第一个改进方法是采用文件块链技术降低程序的磁盘峰值。文件块链的核心思想是降低临时数据的最小存储粒度，以便更早地释放磁盘上的过期数据。

第二个改进方法是一种空间换时间策略。当内存和外存空间充足时，可并行排序多个分块的子串/后缀。

第三个改进方法是采用GPU基数排序算法设计更快的外存排序器。我们在当前的程序版本中使用CPU+外存I/O的方式来排序DSAItem，速度较慢。因为DSAItem的排序关键字为整数，可使用吞吐量较高的GPU基数排序算法来加快外存排序的过程。目前，我们正设计并实现专用的GPU外存排序器。

## 3.5实验分析

### 3.5.1实验环境

我们实现本章算法并在一台塔式工作站上运行算法程序，工作站的硬件配置为1个Intel Xeon E3-1220 V2 CPU、1块4 GB内存和一块500 GB硬盘。所有代码均由gcc/g++ 4.8.4编译器编译，编译优化级别为O3，程序的运行环境是64位ubuntu14.04操作系统。我们在试验中考察以下性能指标：

* 构造时间（CT）：程序运行时间，单位是微秒/字符。
* 磁盘峰值（PDU）：最大磁盘空间需求，单位是字节/字符。
* I/O量（IOV）：如字面所述，单位是字节/字符。

表3-1列出了四个数据集的信息，所有数据集均定义在常数字符集上，单字符可由单字节表示。

|  |  |  |  |
| --- | --- | --- | --- |
| 表3-1 测试集信息 | | | |
| 数据集 | 数据集大小  （GB） | 字符集大小  （个） | 数据集来源 |
| guten | 22.5 | 256 | http://algo2.iti.kit.edu/bingmann/esais-corpus |
| enwiki | 74.7 | 256 | https://dumps.wikimedia.org/enwiki |
| Proteins | 1.1 | 27 | http://pizzachilli.dcc.uchile.cl/texts/protein |
| uniprot | 2.5 | 96 | ftp://ftp.expasy.org/databases/.../complete |

### 3.5.2实验结果

本节比较eSAIS、DSA-IS和DSA-IS+的性能，其中，DSA-IS算法程序在实现原算法的基础上引入了同步排序和命名子串方法，DSA-IS+算法程序则在DSA-IS算法程序的基础上进一步引入了分治排序和命名子串方法。图3-4至3-9展示了三个算法程序在运行时间、最大磁盘空间和I/O量上的性能比较。观察可知，当处理由enwiki和guten数据集截取得到的测试集时，DSA-IS的算法程序所需磁盘空间少于eSAIS。具体而言，DSA-IS和eSAIS的磁盘峰值分别为18n和24n。另一方面，eSAIS的运行速度较DSA-IS更快，这主要是因为两者产生的I/O量上存在差异。我们在表3-2中给出了各算法程序在收缩和解推导阶段的I/O量。观察可知，虽然DSA-IS和eSAIS在解推导阶段产生的I/O相近，但后者在收缩阶段产生的I/O量远小于前者。具体而言，eSAIS和DSA-IS在收缩阶段和解推导阶段产生的I/O量之比分别为0.23和 0.71。继续观察可知：DSA-IS+在各性能指标上均优于DSA-IS，DSA-IS+的算法程序的运行速度快于eSAIS且其磁盘峰值仅为eSAIS的一半。速度上的提升主要是因为DSA-IS+在收缩阶段产生的I/O量仅为DSA-IS的二分之一。理论上，当长、短子串的分类阀值D较小且大多数S\*类型子串为短子串时，DSA-IS+的性能较好。我们在实验中发现，enwiki和guten等现实数据集中的S\*类型子串的平均长度很短，这意味着DSA-IS+在实际中能够取得较好的性能。

|  |
| --- |
| 图3-4 磁盘峰值比较_enwiki.png |
| 图3-4 eSAIS、DSA-IS和DSA-IS+的磁盘峰值比较，输入为enwiki数据集的前缀 |

|  |
| --- |
| 图3-5 IO量比较_enwiki.png |
| 图3-5 eSAIS、DSA-IS和DSA-IS+的I/O量比较，输入为enwiki数据集的前缀 |

|  |
| --- |
| 图3-6 运行时间比较_enwiki.png |
| 图3-6 eSAIS、DSA-IS和DSA-IS+的运行时间比较，输入为enwiki数据集的前缀 |

|  |
| --- |
| 图3-7 磁盘峰值比较_guten.png |
| 图3-7 eSAIS、DSA-IS和DSA-IS+的磁盘峰值比较，输入为guten数据集的前缀 |

|  |
| --- |
| 图3-8 IO量比较_guten.png |
| 图3-8 eSAIS、DSA-IS和DSA-IS+的I/O量比较，输入为guten数据集的前缀 |

|  |
| --- |
| 图3-9 运行时间比较_guten.png |
| 图3-9 eSAIS、DSA-IS和DSA-IS+的运行时间比较，输入为guten数据集的前缀 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表3-2 eSAIS、DSA-IS和DSA-IS+的I/O性能统计，测试集取自enwiki | | | | |
| eSAIS | | | | |
| 测试集大小 | 收缩阶段 | 解推导阶段 | 总量 | 比率 |
| 1G | 36.6 | 132.8 | 169.4 | 0.27 |
| 2G | 36.0 | 141.9 | 177.9 | 0.25 |
| 4G | 35.6 | 152.1 | 187.7 | 0.23 |
| 8G | 35.2 | 165.7 | 200.9 | 0.21 |
| 16G | 35.0 | 172.1 | 207.1 | 0.20 |
| DSA-IS | | | | |
| 测试集大小 | 收缩阶段 | 解推导阶段 | 总量 | 比率 |
| 1G | 81.3 | 109.6 | 190.9 | 0.74 |
| 2G | 83.5 | 111.6 | 195.1 | 0.75 |
| 4G | 94.3 | 144.1 | 238.4 | 0.65 |
| 8G | 107.8 | 159.6 | 267.4 | 0.68 |
| 16G | 121.9 | 166.1 | 288.0 | 0.73 |
| DSA-IS+ | | | | |
| 测试集大小 | 收缩阶段 | 解推导阶段 | 总量 | 比率 |
| 1G | 45.4 | 91.7 | 137.1 | 0.33 |
| 2G | 47.2 | 93.4 | 140.6 | 0.34 |
| 4G | 54.1 | 111.5 | 165.6 | 0.33 |
| 8G | 60.1 | 122.1 | 182.2 | 0.33 |
| 16G | 62.7 | 128.7 | 191.4 | 0.33 |

## 3.6本章总结

本章介绍了DSA-IS算法及其优化方法。在实验环节，我们采用本章第3节中的两个设计优化方法来改造DSA-IS。实验结果表明：分类排序和命名长、短子串方法能够显著降低程序的磁盘峰值和I/O量。

目前我们计划重构DSA-IS算法在收缩阶段的设计，思路如下：DSA-IS算法使用通用的外存堆来排序DSAItem。可设计一个专用的外存堆，该外存堆由两个内存小顶堆和多个外存数组组成。其中一个内存小顶堆H1中的元素类型为DSAItem，排序关键字为，可存储元素的个数。另一个内存小顶堆H2中的元素类型为（记为DSAItem’），排序关键字为，存储的元素个数为k，k为当前外存数组的个数。每个外存数组中的元素类型为DSAItem’，数组中的元素已按从小到大排列有序，所有外存数组中的元素通过H2来合并。该外存堆的工作原理是：将已经排序完成的子串序列划分为多个有序序列，每个序列包含由字面值相等的子串推出的所有前继子串，从而可通过比较首字符和类型来确定位于不同序列中的子串的大小关系。现阶段的测试数据表明：该外存堆可显著降低最大磁盘空间，当与文件块链结合使用时，算法程序的磁盘峰值不超过7.25n，接近理论最优值。

# 第四章 有限阶最长公共前缀数组构造算法

## 4.1预备知识

已知输入字符串x和后缀数组sa，x的最长公共前缀数组记录了sa中每对相邻后缀的最长公共前缀的长度。具体而言，记录了和的最长公共前缀的长度。最长公共前缀数组能够辅助后缀数组模拟后缀树上的遍历操作。在内存和外存模型上，可改造使用IS方法设计的后缀数组构造算法来同时计算后缀数组和最长公共前缀数组。IS方法推导最长公共前缀数组的原理可简述如下：令和是插入到sa中相邻位置的两个后缀，它们的后继在sa中的位置分别是p和q。若和的首字符不相等，则两者的最长公共前缀的长度为0；否则，两者的最长公共前缀的长度为。求解的最小值是一个区间最小值查询问题，平均查询时间为。

在实际应用场景中，待匹配模式一般仅含几个或十数个字符，无需为输入字符串构造全阶后缀数组和最长公共前缀数组。本章提出一种有限阶最长公共前缀数组构造算法，算法可部署在单机内存和外存模型上，也可部署在对称型分布式模型上。在描述算法原理之前，我们先给出K阶最长公共前缀数组的定义。已知字符串x和其后缀数组sa，x的K阶最长公共前缀数组满足：对任意，若和的最长公共前缀的长度*l*大于K，则；否则，。

## 4.2算法原理

若字符串x的长度为n，则x中任意两个后缀的最长公共前缀的长度小于n。已知文献提出了一种递归计算任意多对后缀的最长公共前缀的算法。令是一个包含任意多对二元组的集合，每个二元组记录一对后缀的起始地址，则该算法执行次递归，在第k次递归时的执行过程为：

1. 对中的每个二元组，比较和。若相等，则令且；否则，令且。
2. 将放入。

当执行完所有递归时，任意在中有对应的二元组。此时，以和为起始地址的两个后缀的最长公共前缀的长度值可由计算得到。

该算法的核心思想是在第k次递归时判断中每对二元组所对应的后缀对的前半是否相等，若是则在第k+1次递归时继续判断后续的四分之一是否相等，否则在第k+1次递归时转而判断前四分之一是否相等。在步骤1，若按字面值比较和，则上述算法的时间复杂度为，其中m为中二元组的个数。另一种方法是首先计算子串的哈希值，然后比较哈希值来判断子串是否相等。若使用支持递归计算的Karp-Rabin指纹函数技术，可在步骤1使用公式4-1至4-3计算所有待比较子串的哈希值。在公式4-1至4-3中，记录的哈希值，参数L和均为质数且。

|  |
| --- |
| 公式4-1： |
| 公式4-2： |
| 公式4-3： |

根据上述讨论，首先从左向右遍历x并使用公式4-1和4-2递归计算x中所有前缀的哈希值，计算得到的哈希值保存在一个哈希表中。然后，执行次递归且在第k次递归的步骤1中执行以下子步骤来计算待比较子串的哈希值：

1. 从哈希表中取出和并使用公式4-3计算。类似地，计算。
2. 比较和。

## 4.3算法设计

### 4.3.1内存

我们将构造K阶最长公共前缀数组问题转化为计算对后缀的K阶最长公共前缀问题。令，设计内存算法4-1来计算中各二元组指代的后缀对的最长公共前缀：

1. 遍历x并使用公式4-1和4-2递归计算x中所有前缀的哈希值，计算得到的哈希值存储在内存哈希表中。
2. 执行次递归并在第k次递归执行以下子步骤：
   1. 依次取出中的元组。若当前取出的元组为，则从哈希表中取出和并使用公式4-3计算。类似地，计算。
   2. 比较和。若相等，则将插入；否则，将插入。
3. 依次取出中的元组及其在中的对应元组。若当前取出的元组为和，则和的最长公共前缀的长度为。将计算得到的值记录到中。

### 4.3.2外存

可将算法4-1改造为外存算法，改造过程需要解决两个问题。其一、在外存模型上无法假设内存容量足够存储算法4-1的步骤1中生成的哈希表；其二、公式4-3计算时需要快速计算，这需要在内存中保存，而外存模型上无法假设内存容量足够存储。

注意到对任意，可将分解为以2为底的多项式的积，其中。从而，我们只需在内存中保存就能在的时间内快速得到的值。这解决了第二个问题。此外，我们可借鉴DSA-IS算法，将哈希值按其访问顺序保存在外存文件中并在需要时通过顺序I/O读入内存。这解决了第一个问题。按前述思路，我们提出算法4-2，该算法引入一个三元组数据结构LCPItem，每个LCPItem存储一个数组下标idx、一个位置信息pos和一个哈希值fp。算法4-2执行次递归并在第k次递归时依次执行以下步骤：

1. 为中的每对元组创建四个LCPItem。前两个LCPItem的pos值分别为和，将pos值为的LCPItem添加到数组CPk的末尾并更新idx值为其在CPk中的数组下标，将pos值为的LCPItem添加到数组PPk的末尾并更新idx为其在PPk中的数组下标。后两个LCPItem的pos值分别为和，类似地将它们添加到CPk和PPk的末尾。
2. 使用外存堆按pos值排序CPk和PPk中的元素，得到两个数组的逆数组ICPk和IPPk。
3. 从左向右遍历x计算所有前缀的哈希值。若当前计算得到，则从ICPk和IPPk中取出pos值等于的元素并将保存到它们的fp数据域中。
4. 使用外存堆按idx值排序ICPk和IPPk中的元素，重新得到CPk和PPk。
5. 从左向右遍历CPk和PPk中的元素并依次执行以下三个子步骤：
   1. 从CPk中取出最左未访问的两个LCPItem。假设两个LCPItem的pos数据域的值为a和b，则它们的fp数据域分别保存和。使用公式4-3计算。
   2. 从PPk中取出最左未访问的两个LCPItem。假设两个LCPItem的pos数据域的值为c和d，则仿照上个子步骤计算。
   3. 比较和。若相等，则将插入；否则，将插入。
6. 依次取出中的元组及其在中的对应元组。若当前取出的元组为和，则和的最长公共前缀的长度为。将计算得到的值记录到中。

在算法4-2中，每次递归的时、空复杂度与外存整数排序的复杂度相当。

### 4.3.3分布式

本章将算法4-2改造为分布式算法4-3，新算法可部署在对称型分布式系统中。如图4-1所示，对称型分布式系统由计算节点通过局域网交换机互联而成，每个结点配有一个容量为M的内存单元和一个容量为E的外存单元。算法4-3要求输入的字符串x和后缀数组sa按如下方式分布在各计算节点：

* 将x切分为均等的d个分块，节点在外存中存储分块。
* 将sa切分为均等的d个分块，节点在外存中存储分块。

算法4-3执行次递归，我们引入以下参数符号和名词术语来描述第k次递归的执行过程：

* Pi,k：存储在节点外存上的二元组集合，为Pk的一部分。特别地，。
* 和：存储在节点外存上的LCPItem数组，为和的一部分。其中，，。类似地，我们定义和。
* 和：在节点的内存中常驻的两个数据发送缓冲区。和缓存等待发送至节点的。
* 和：在节点的内存中常驻的两个数据接收缓冲区。和缓存接收自节点的。

算法4-3在第k次递归时在节点上依次执行以下步骤：

1. 参照算法4-2的步骤1，为Pi,k中的每对元组创建四个LCPItem并将创建的LCPItem添加到和的末尾
2. 按pos值外存排序和中的LCPItem，得到和。将和中pos值在范围内的LCPItem通过和发送至。同时，从和中读取接受自的。
3. 参照算法4-2的步骤3，在遍历时计算其所有前继的哈希值并将计算得到的哈希值保存在步骤2接收到的LCPItem中。将idx值在范围内的LCPItem分别通过和发送至。同时，从和中读取接收自的。
4. 按idx值外存排序从各计算节点接收到的LCPItem，得到和。
5. 参照算法4-2的步骤5，遍历和中的元素来计算。
6. 参照算法4-2的步骤6，依次取出中的元组及其在中的对应元组来计算。

|  |
| --- |
|  |
| 图4-1 由d个计算节点构成的对称型分布系统 |

在算法4-3中，各结点上的内存容量被划分为两部分，一部分用于创建收发数据的缓冲区，另一部分用于创建外存操作的I/O缓冲区。在执行完所有递归后，的磁盘上存储了的一个连续区间。

## 4.4算法优化

直观上，本章算法的运行时间与递归次数呈正比。在执行第k次递归时，各算法的目标是从计算。可修改算法流程，跳过Pk+1，从直接计算。下面举例说明如何改造算法4-1的步骤2的子步骤来减少递归次数。

1. 依次取出中的元组。若当前取出的元组为，则从哈希表中取出和并使用公式4-3计算。类似地，计算。
2. 比较和。若相等，则执行子步骤c）；否则，执行子步骤e）。
3. 从哈希表中取出和并使用公式4-3计算。类似地，计算。
4. 比较和。若相等，则将插入；否则，将插入。结束当前递归。
5. 从哈希表中取出和并使用公式4-3计算。类似地，计算。
6. 比较和。若相等，则将插入；否则，将插入。结束当前递归。

理论上，该方法可将任意多次递归合并成一次。

## 4.5实验分析

本节测试算法4-2和4-3的性能，数据集与第三章实验部分相同，所使用的性能指标如下所示：

* S：测试数据集大小。
* ST：总的运行时间，单位为微妙。
* MT：处理单字符的运行时间，单位为微秒/字符。
* PD：磁盘峰值，单位为字节/字符。
* LR：递归次数，单位为次。
* W：采用第4.4节的优化方法将W次递归合并为一次递归。

我们首先分析外存算法的性能，下文使用lcp-disk指代算法4-2的程序实现。表4-1展示了lcp-disk的性能统计，其中有限阶参数。观察可知：lcp-disk在时的运行时间大幅少于在时的运行时间，但是前一种情况的单轮递归执行所需磁盘峰值和时间多于后一种情况。此外，lcp-disk在时的运行速度随着输入测试集的增大而快速增大并最终劣于和时的运行速度。这些现象可解释如下：lcp-disk在执行每次递归过程时需重新计算x中所有前缀的哈希值并遍历x和LCPItem的数组，这一部分的CPU和I/O开销与递归次数呈线性正比。若将多次递归合并为一次来执行，虽然可以减少前述的顺序I/O读写和CPU计算的开销，但每次递归所需排序的LCPItem成倍增加，这一部分的I/O复杂度是超线性的。从实验数据分析可知，若将两次递归合并为一次，则外存排序的开销的增幅不及顺序I/O和CPU计算的开销的降幅；若将三次递合并为一次，则外存排序的开销的增幅超出了顺序I/O和CPU计算的开销的降幅。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表4-1 lcp-disk的性能统计， | | | | | | | | |
| S  (GB) | lcp-disk () | | | | lcp-disk () | | | |
| LR | MT | ST | PD | LR | MT | ST | PD |
| 0.2 | 13 | 1.03 | 2803 | 30.94 | 7 | 1.33 | 1950 | 47.36 |
| 0.4 | 13 | 1.11 | 6034 | 31.50 | 7 | 1.40 | 4108 | 47.36 |
| 0.6 | 13 | 1.18 | 9650 | 31.58 | 7 | 1.48 | 6928 | 47.36 |
| 0.8 | 13 | 1.34 | 14557 | 31.52 | 7 | 1.80 | 10547 | 47.36 |
| 2.0 | 13 | 1.72 | 48021 | 46.04 | 7 | 2.47 | 37029 | 60.48 |
| S  (GB) | lcp-disk () | | | |  | | | |
| LR | MT | ST | PD |  | | | |
| 0.2 | 5 | 2.43 | 2541 | 77.52 |  | | | |
| 0.4 | 5 | 2.20 | 4605 | 78.58 |  | | | |
| 0.6 | 5 | 3.14 | 9860 | 130.86 |  | | | |
| 0.8 | 5 | 3.06 | 12827 | 103.50 |  | | | |
| 2.0 | 5 | 4.59 | 49280 | 94.40 |  | | | |

我们接着分析分布式算法的性能，下文使用lcp-ds指代算法4-3的程序实现。分析算法4-2可知，各结点工作负载平衡且单个结点上的通信量为。根据Amdahl定律，lcp-ds的理论速度应该是lcp-disk的d分之一。为验证该预期，我们在表4-2中给出了lcp-ds和lcp-disk的性能统计和对比。观察可知，当时，lcp-ds的运行时间是lcp-disk的二分之一而其单节点上的磁盘峰值为后者的四分之一。显然，运行时间的比率与我们的预期不符，这主要是因为分布式程序中额外的数据通信开销。目前，我们的程序运行在同步工作方式下，仅当目标数据从其他结点完整搬运至本地后才执行数据处理作业，停等时间较多。我们的下一步工作是改进算法流程，减少无效的停等时间。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表4-2 lcp-disk和lcp-ds的性能统计和对比，， | | | | | | | | |
| S  (GB) | LR | lcp-ds | | | lcp-disk | | | ST Ratio  (%) |
| *MT* | *ST* | *PD* | *MT* | *ST* | *PD* |
| 200 | 7 | 0.71 | 1038 | 13.50 | 1.33 | 1950 | 47.36 | 0.54 |
| 400 | 7 | 0.72 | 2113 | 13.50 | 1.40 | 4108 | 47.36 | 0.52 |
| 600 | 7 | 0.74 | 3235 | 13.75 | 1.48 | 6928 | 47.36 | 0.50 |
| 800 | 7 | 0.78 | 4547 | 13.82 | 1.80 | 10547 | 47.38 | 0.43 |
| 2048 | 7 | 1.24 | 18617 | 16.21 | 2.47 | 37029 | 60.48 | 0.50 |

## 4.6本章小结

本章提出有限阶最长公共前缀数组构造算法，可部署于内存、外存和对称型分布式模型。算法的外存版本的C++实现不到600行，易于维护和改进。已知有限阶的值为K，算法的分布式版本在规模为d的分布式系统上的时、空复杂度为和，且单节点上的时、空复杂度为总量的d分之一。目前，尚无研究提出可在分布式环境下运行的（有限阶）最长公共前缀数组构造算法，我们的工作填补了该研究空白。

# 第五章 后缀数组和最长公共前缀数组事后验证算法

## 5.1 预备知识

后缀数组和最长公共前缀数组的外存构造算法可构建TB级索引，算法程序包含数千行代码。目前，eSAIS、bwt-disk和DSA-IS等算法的作者均向第三方免费提供开源程序包，但其不保证程序的正确性。因此，需要提供简单、高效的验证程序，检测构造结果的正确性。除本文工作外，目前已知的唯一一个后缀数组验证算法根据引理5-1验证后缀数组。我们在本章的实验部分实现了该算法的外存版本，算法流程主要包括两轮外存整数排序。

引理5-1：已知长度为n的字符串x，当且仅当以下条件成立时，是x的后缀数组：

1. sa是的一个全排列。
2. 对任意且，。其中，和表示和的字典序。

证明：略。

本章提出两个后缀数组和最长公共前缀数组的事后验证算法。第一个算法从数组的定义出发，同时检测后缀数组和最长公共前缀数组的正确性。第二个算法首先调用第一个算法来验证输入字符串x中S\*类型后缀的后缀数组和最长公共前缀数组，然后采用IS方法从和推出x的后缀数组和最长公共前缀数组，最后比较计算得到的和输入的后缀数组和最长公共前缀数组来验证输入是否正确。

## 5.2 算法一

### 5.2.1原理

下文中的引理5-2从后缀数组和最长公共前缀数组的定义出发，给出了一组验证两个数组是否正确的充要条件：

引理5-2：已知长度为n的字符串x，当且仅当以下条件成立时和分别为x的后缀数组和最长公共前缀数组，其中：

1. sa是的一个全排列。
2. 。

证明：条件1）保证sa包含x中所有后缀，条件2）和3）保证对任意成立，于是由定义可知：sa正确。假设引理5-2中的条件成立，则由前述分析可知sa正确，条件2）和3）又保证对任意均有且成立，于是由定义可知：lcp正确。证毕。

为检测引理5-2的条件2）是否成立，可按字面值比较子串来判定它们是否相等，时间复杂度为。为提高速度，可首先使用支持递归计算的哈希函数得到这些子串的哈希值，然后比较哈希值来判定子串是否相等，时间复杂度为。需要指出的是：不同子串的哈希值可能相同，从而导致哈希冲突。我们选用Karp-Rabin指纹技术计算子串的哈希值，冲突概率可忽略不计。令表示子串的哈希值，我们得到以下结论：

推论5-1：已知长度为n的字符串x，当以下条件成立时和大概率为x的后缀数组和最长公共前缀数组，其中：

1. sa是的一个全排列。
2. 。
3. 。

图5-1用示例展示了如何使用Karp-Rabin指纹技术验证推论5-1的条件2）。首先，第4-8行递归计算x中所有前缀的哈希值。其后，第10-12行计算并比较与的哈希值来判断条件2）对是否成立，而第13-14行计算并比较与的哈希值来判断条件2）对是否成立。以此类推。

|  |
| --- |
| 图5-1 Karp-Rabin验证后缀数组和最长公共数组实例.png |
| 图5-1 使用Karp-Rabin指纹技术验证推论5-1的条件2） |

### 5.2.2设计

我们首先在内存模型上设计验证推论5-1的算法5-1，该算法初始化两个数组和。中下标为i的元素存储的哈希值，中下标为i的元素标记是否在sa中出现。算法依次执行以下步骤：

1. 从左向右遍历x中的字符，使用Karp-Rabin指纹技术递归计算x中前缀的哈希值并将计算结果存储在中。
2. 从左向右遍历sa和lcp中的元素。若当前访问和，则执行以下子步骤验证且两后缀的最长公共前缀的长度等于：
   1. 记，从读取和来计算，令。
   2. 从读取和来计算。
   3. 判断且是否成立。
3. 令=1。判断=1是否对任意成立。

算法5-1在执行过程中非顺序地访问和中的元素。若将算法移植到外存模型上，需将非顺序的访问操作转化为顺序的访问操作，可借鉴DSA-IS算法，首先将和中的元素按其访问顺序排序后保存在外存文件中，然后在需要时通过顺序的I/O操作读入内存。于是我们得到算法5-1的外存版本，该外存算法的执行过程如下所述：

1. 从左向右遍历sa和lcp来创建数组、和。三个数组的定义为：  
   ，，。
2. 按左侧第一个数据域的值从小到大排序、和中的元素。
3. 从左向右遍历x并使用Karp-Rabin指纹技术计算x中各前缀的哈希值。若当前访问的字符为，则在排序后的、和中查找最左数据域的值为i的所有元素并将和分别记录在这些元素的左侧第三和第四个数据域。该步骤需从左向右遍历、和中的元素。注意到中的元素的左侧第一数据域应当构成sa的一个拷贝，若在遍历排序后的中的元素时发现其左侧第一数据域为的有序序列，则可判断推论5-1中的条件1）是成立的。
4. 按左侧第二个数据域的值从小到大排序、和中的元素。
5. 我们从和的元素中记录的哈希值计算得到的哈希值并从和的元素中记录的哈希值计算得到的哈希值，比较两个哈希值来判断推论5-1中的条件2）是否成立。我们从和的元素中获取和，比较两个字符来判断推论5-1中的条件3）是否成立。

### 5.2.3复杂度

第5.2.2小节中的外存算法执行多次数组遍历和整数排序操作。假设内存容量为M、外存容量为D且外存读写块的大小为B（M、D、B均以字节为单位），一次外存数组遍历的时间和I/O复杂度分别为和，而一次外存整数排序的时间和I/O复杂度分别为和。算法的磁盘峰值出现在外存整数排序的执行过程中，假设字符串x和其后缀数组/最长公共前缀数组的单个元素分别占用和个字节，而单个哈希值占用个字节，则排序所需的磁盘空间为，而排序和的磁盘空间为。为节省空间，可串行排序、和中的元素。

## 5.3 算法二

### 5.3.1 原理

本节提出一种采用IS方法设计的验证算法，该算法的外存版本的磁盘空间需求远小于算法5-1的外存版本。回忆一下，采用IS方法设计的后缀数组和最长公共前缀数组的构造算法主要包括收缩和解推导两个阶段，收缩阶段计算字符串x中所有S\*类型后缀的后缀数组sa\*和最长公共前缀数组lcp\*，解推导阶段从sa\*和lcp\*推导出所有后缀的后缀数组sa和最长公共前缀数组lcp。假设sa\*和lcp\*已知，可先检测sa\*和lcp\*是否正确，然后从已验证的sa\*和lcp\*推导sa和lcp。若能保证推导过程被正确地实施，则推导出的sa和lcp是正确的。此时，可将推导出的和输入的后缀数组与最长公共前缀数组比较来判断输入是否正确。此验证算法的合理性在于：只需正确地实现解推导阶段，就能够从已验证的sa\*和lcp\*推导出正确的后缀数组和最长公共前缀数组。在实际编程中，解推导阶段的代码只在百行规模。按前述思路，我们在引理5-3中给出一组验证后缀数组和最长公共前缀数组的充要条件。

引理5-3：已知字符串x、后缀数组sa、最长公共前缀数组lcp和任意一种采用IS方法设计的后缀数组和最长公共前缀数组构造算法A，当且仅当满足以下条件时，sa和lcp是正确的：

1. sa\*和lcp\*是正确的。
2. 且，其中和是执行A的解推导阶段从sa\*和lcp\*计算得到的。
3. A的解推导阶段的实现是正确的。

可使用算法5-1来判断引理5-3中的条件1）是否成立。关于条件2），可按字面值比较和来判断它们是否相等，也可使用Karp-Rabin指纹技术递归计算和的哈希值并比较哈希值来判断它们是否相等。我们在推论5-2中给出引理5-3的概率性版本，本章算法验证推论5-2中的条件来概率性检测后缀数组和最长公共前缀数组是否正确。

推论5-2：给定字符串x、后缀数组sa、最长公共前缀数组lcp和任意一种采用IS方法设计的后缀数组和最长公共前缀数组构造算法A，当满足以下条件时sa和lcp大概率是正确的，其中，，且：

1. 是S\*类型字符且。
2. fp(。
3. 。
4. 且。其中，和是执行A的解推导阶段从sa\*和lcp\*计算得到的。
5. A的解推导阶段的实现是正确的。

### 5.3.2 设计

我们首先在内存模型上设计算法5-2，算法依次执行以下步骤来检测推论5-2中的条件是否成立：

1. 按sa中的元素值排序sa中的元素下标，得到sa的逆isa。此过程需遍历sa中的元素，在遍历时计算sa的哈希值。
2. 从左向右遍历x和isa，判断x中各后缀的类型并从isa中取出各S\*类型后缀在sa中的字典序，将S\*类型后缀的字典序记录在数组isa\*中。
3. 按isa\*中的元素值排序isa\*的元素下标，得到sa\*。
4. 从左向右遍历sa、lcp和sa\*来计算sa\*中相邻后缀的最长公共前缀的长度值，得到lcp\*。在遍历时计算lcp的哈希值。
5. 调用算法5-1检测sa\*和lcp\*的正确性。
6. 执行SA-IS算法的解推导阶段从sa\*和lcp\*推导出sa’和lcp’。在推导过程中需遍历sa’和lcp’中的元素，在遍历时计算sa’和lcp’的哈希值。
7. 比较sa/lcp和sa’/lcp’的哈希值来判断两者是否相等。

步骤4的关键操作是计算在sa\*中相邻的S\*类型后缀的最长公共前缀。假设和是sa\*中的任意两个相邻后缀且它们在sa中的位置为p和q，则这两个后缀的最长公共前缀是中的最小值。

可按如下方式改造算法5-2得到外存版本：将内存顺序读写替换为外存顺序读写，将内存整数排序替换为外存整数排序，将步骤5中调用的算法5-1替换为相应的外存版本，将步骤6中执行的SA-IS算法的解推导阶段替换为DSA-IS算法的解推导阶段。

### 5.3.3复杂度

算法5-2主要包含两步，第一步验证sa\*和lcp\*，第二步验证sa和lcp。第一步的性能瓶颈在于调用算法5-1验证sa\*和lcp\*的正确性，第二步的性能瓶颈在于采用IS方法从sa\*和lcp\*解推导sa和lcp，两步的时、空复杂度与外存整数排序相当。

算法5-2的开销远远小于构造后缀数组和最长公共前缀数组的开销。这主要是因为两个原因：其一，任意字符串x中的S\*类型后缀个数与总的后缀个数之比不超过二分之一，该比例在多数现实数据集中不超过三分之一。其二、从sa\*和lcp\*推导sa和lcp的过程只需执行一次解推导阶段，而从x推导sa和lcp的过程需递归执行多次收缩阶段和解推导阶段。

## 5.4 算法优化

算法5-1的性能瓶颈在于外存排序、和。因为、和之间没有数据依赖，可并行执行三个排序过程，用空间换取时间。

当验证完sa\*和lcp\*后，算法5-2从sa\*和lcp\*解推导sa’和lcp’。在解推导过程中，需要将推导出的sa’和lcp’中的元素按字典序保存在外存以便之后推导它们的前继。这意味着需要在外存中同时维持两个后缀数组和最长公共前缀数组的副本，一个是输入的sa和lcp，另一个是计算中的sa’和lcp’。假设字符串x定义在常数字符集上，可访问sa和lcp中的元素来推导sa’和lcp’中的元素，从而无需在外存中保存sa’和lcp’。方法的核心思想是：为sa/lcp中的每个桶维持一个指针，指针指向下一个将被推导入sa’/lcp’中同一个桶的元素位置；当推导出sa’/lcp’的一个元素时，首先确定该元素所插入的桶，然后通过指针取出sa/lcp中同一个桶的对应元素；若从sa/lcp中取出的元素与待插入sa’/lcp’的元素相等，则说明sa/lcp中的元素是正确的，从而可访问sa/lcp中已验证的元素来继续推导尚未插入sa’/lcp’中待排元素。简而言之，该方法将sa/lcp中已验证的元素视为sa’/lcp’中对应元素的镜像。我们在本章实验环节测试该方法的效果。

除上述方法外，文件块链技术和GPU基数排序是提高两个算法的时、空性能的通用方法。

## 5.5 实验分析

本章评估算法5-1和5-2在外存模型上的性能，性能指标和数据集与第三章相同。为描述方便，我们简记算法5-1和5-2的外存版本为ProgA和ProgB。

图5-2至5-4展示了ProgA和ProgB在不同测试集上的性能比较。观察可知，ProgB的运行速度较ProgA慢五分之一。速度差异主要是因为两者的I/O开销不同，ProgA在不同测试集上的I/O量均保持在155n，而ProgB在相同条件下的I/O量达到了200n。此外，ProgA和ProgB的磁盘峰值分别为40n和26n，后者是前者的0.65倍。

算法2调用算法1来验证S\*类型后缀的后缀数组sa\*和最长公共前缀数组lcp\*。因为任意字符串中的S\*类型后缀的个数不超过后缀总数的二分之一，ProgB用于验证sa\*和lcp\*的开销应当不超过ProgA的开销的二分之一。为证实该假设，我们在ProgB和ProgA中分别统计用于验证sa\*&lcp\*和sa&lcp的开销，统计结果如表5-1所示。观察可知，测试数据集中的S\*类型后缀的个数与后缀总数之比约为0.30，而验证sa\*和lcp\*的时间、空间和I/O量分别是验证sa和lcp的时间、空间和I/O量的0.38、0.57和0.60倍。

因为所有数据集的字符集为常数，我们在ProgB中实现第5.4节提出的优化方法并测试改造后的程序ProgB+的性能。观察图5-2至5-4可知，ProgB+的最大磁盘空间为21n，该值小于ProgB且仅为ProgA的二分之一。此外，ProgB+在时间和I/O性能上也优于ProgB。接着，我们在图5-5至5-7中分析ProgA、ProgB和ProgB+的性能与输入测试集大小的关系。观察可知，各程序的磁盘空间不随测试集规模的变化而变化，但它们的运行速度随测试集规模的增大而减小。分析表5-1可知，当测试集规模变大时，ProgB+的I/O量始终保持在90n上下而其运行时间从1.05攀升到了1.33。这表明：程序使用的外存排序器在处理大规模数据集时出现性能衰减。

|  |
| --- |
| 图5-1 磁盘峰值量比较.png |
| 图5-2 ProgA、ProgB、ProgB+和ProgC在多个测试集上的磁盘峰值比较 |
| 图5-1 IO量比较.png |
| 图5-3 ProgA、ProgB、ProgB+和ProgC在多个测试集上的I/O量比较 |

|  |
| --- |
| 图5-3 运行时间比较.png |
| 图5-4 ProgA、ProgB、ProgB+和ProgC在多个测试集上的运行时间比较 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 表5-1 算法5-1的外存版本验证sa\*&lcp\*和sa&lcp的开销 | | | | | | |
| 测试集 | 后缀数 | | | 磁盘峰值 | | |
| S\*类型 | 总数 | 比例 | S\*类型 | 总数 | 比例 |
| enwiki\_1g | 329,810,376 | 1,073,741,824 | 0.31 | 15.67 | 40 | 0.39 |
| enwiki\_2g | 650,901,939 | 2,147,483,648 | 0.30 | 15.41 | 40 | 0.39 |
| enwiki\_4g | 1,301,327,878 | 4,294,967,296 | 0.30 | 15.45 | 40 | 0.39 |
| enwiki\_8g | 2,586,471,839 | 8,589,934,592 | 0.30 | 15.35 | 40 | 0.38 |
| uniprot | 829,262,945 | 3,028,811,776 | 0.27 | 13.94 | 40 | 0.35 |
| proteins | 379,092,002 | 1,184,366,592 | 0.32 | 16.21 | 40 | 0.41 |
| mean | 1,012,811,163 | 3,189,156,522 | 0.30 | 15.34 | 40 | 0.38 |
| 测试集 | I/O量 | | | 运行时间 | | |
| S\*类型 | 总数 | 比例 | S\*类型 | 总数 | 比例 |
| enwiki\_1g | 89.94 | 155 | 0.58 | 1.05 | 1.70 | 0.62 |
| enwiki\_2g | 89.18 | 155 | 0.58 | 1.22 | 1.85 | 0.66 |
| enwiki\_4g | 89.14 | 155 | 0.58 | 1.19 | 1.89 | 0.63 |
| enwiki\_8g | 88.80 | 155 | 0.57 | 1.33 | 2.14 | 0.62 |
| uniprot | 83.80 | 155 | 0.54 | 1.04 | 2.26 | 0.46 |
| proteins | 92.29 | 155 | 0.60 | 1.14 | 1.85 | 0.62 |
| mean | 88.86 | 155 | 0.57 | 1.16 | 1.95 | 0.60 |

|  |
| --- |
| 图5-5 磁盘空间比较_enwiki.png |
| 图5-6 ProgA、ProgB、ProgB+和ProgC在单个测试集上的磁盘峰值比较 |

|  |
| --- |
| 图5-6 IO量比较_enwiki.png |
| 图5-7 ProgA、ProgB、ProgB+和ProgC在单个测试集上的I/O量比较 |

|  |
| --- |
| 图5-7 运行时间比较_enwiki.png |
| 图5-8 ProgA、ProgB、ProgB+和ProgC在单个测试集上的运行时间比较 |

在本文执笔阶段，尚无文献报告最长公共前缀数组的验证方法，而文献的工作是仅知的一个后缀数组验证算法。为进一步评估本章提出的算法性能，我们实现了文献的工作并将程序命名为ProgC。图5-2至5-8比较了ProgC与ProgA、ProgB和ProgB+的性能。观察可知，ProgC的运行时间分别比ProgA和ProgB快两倍和三倍，这一结果与各程序的I/O量保持一致。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表5-2 ProgA、ProgB+与两个参考构造算法的程序之间的性能比较 | | | | | | | | |
| 测试集 | 第一个构造算法的程序 | | | 第二个构造算法的程序 | | | ProgA | ProgB+ |
| eSAIS | Sparse-  串行版 | 总和 | pSAscan | Sparse-  并行版 | 总和 |
| enwiki\_1g | 2.21 | 0.61 | 2.82 | 0.39 | 0.59 | 0.98 | 1.70 | 2.54 |
| enwiki\_2g | 2.63 | 0.53 | 3.16 | 0.47 | 0.53 | 1.00 | 1.84 | 2.51 |
| enwiki\_4g | 2.90 | 0.63 | 3.53 | 0.59 | 0.40 | 0.99 | 1.89 | 2.56 |
| enwiki\_8g | 3.02 | 0.63 | 3.65 | 0.83 | 0.45 | 1.28 | 2.13 | 2.79 |

最后，我们将ProgA和ProgB+与当前最快的两个后缀数组和最长公共前缀数组的构造算法的程序进行比较。第一个构造算法由后缀数组构造算法eSAIS和最长公共前缀数组构造算法Sparse-的串行版组成，第二个构造算法由后缀数组构造算法pSAscan和最长公共前缀数组构造算法Sparse-的并行版组成。实验结果如表5-2所示。分析可知，当输入为enwiki\_8g时，第二个构造算法的程序比第一个构造算法的程序快且其运行时间是ProgA的二分之一。然而，pSAscan和Sparse-的时间复杂度均正比于，这远比eSAIS和本章提出的算法高。当字符串x的长度n远大于可用内存容量M时，pSAscan和Sparse-算法的程序均出现显著的性能衰减。在本实验中，pSAscan在构造enwiki\_1g和enwiki\_8g时的运行速度之比为2。

## 5.6 本章总结

本章提出两个后缀数组和最长公共前缀数组的事后验证算法，可大概率检测任意构造算法输出的后缀数组和最长公共前缀数组的正确性。理论上，两个算法的时、空复杂度优于当前最快的后缀数组和最长公共前缀数组的构造算法。算法5-1通用性好，可验证完整或稀疏后缀数组和最长公共前缀数组。算法5-2专用于验证完整后缀数组和最长公共前缀数组。实验结果表明：在验证完整后缀数组和最长公共前缀数组时，算法5-2的运行速度慢于算法5-1，但前者所需的磁盘空间小于后者。算法5-1和算法5-2的设计与实现仍有较大的改进空间，我们现阶段的研究目标是通过简化算法5-2的执行流程来降低时、空开销。

# 第六章 后缀数组事中验证算法

## 6.1预备知识

第五章提出了两个后缀数组和最长公共前缀数组的事后验证算法。其中，算法5-2首先验证S\*类型后缀的后缀数组sa\*和最长公共前缀数组lcp\*，然后使用IS方法从sa\*和lcp\*推出一个后缀数组和最长公共前缀数组的副本，最后将该副本与输入的副本比较来判断输入的副本是否正确。为了确保从sa\*和lcp\*推导出的后缀数组和最长公共前缀数组是正确的，算法5-2假设解推导阶段在执行过程中是无误的。换言之，如果解推导阶段的实现是正确的，那么我们可以将验证sa/lcp的问题转化为验证sa\*/lcp\*的问题，由此得到引理6-1中的结论：

引理6-1： 已知长度为n的字符串x和采用IS方法设计的任意后缀数组构造算法A，当以下条件成立时，算法A的程序所输出的是x的后缀数组：

1. 算法A的程序在最初递归层的收缩阶段计算出的sa\*是正确的。
2. 算法A的程序在最初递归层的解推导阶段的实现是正确的。

证明：我们证明引理6-1和5-1互为充要。显然，必要性成立。令和分别为和在最初递归层推出的前继，我们如下证明充分性。已知引理6-1的条件2）成立，则。于是可知引理5-1的条件1）成立。已知引理6-1的条件2）成立，若且，则且。将替换为并重复该推理过程直到为S\*类型，此时可知sa\*中包含重复后缀。又已知引理6-1的条件1）成立，则sa\*中必不包含重复后缀，出现矛盾。于是可知引理5-1的条件2）成立。证毕。

我们指出：虽然采用IS方法设计的后缀数组构造算法比较复杂，但其最初递归层的解推导阶段的逻辑较为简单。可单独实现最初递归层的解推导阶段以确保该段代码的正确性，从而将sa的验证问题转化为sa\*的验证问题。根据这一思路，本章提出两个后缀数组事中验证算法，算法采用不同的方法验证sa\*。两算法均需与采用IS方法设计的任意后缀数组构造算法联用，同步构造并验证后缀数组。事中验证算法虽仅能验证IS方法设计的后缀数组构造算法的输出，但同步执行构造和验证的开销小于依次执行构造和验证的开销。

## 6.2算法一

本节提出的事中验证算法类似算法5-2。具体而言，首先验证所有S\*类型后缀的后缀数组sa\*和最长公共前缀数组lcp\*，然后执行由IS方法设计的任意构造算法的最初递归层的解推导阶段从sa\*和lcp\*推导出输入字符串x的后缀数组和最长公共前缀数组。若解推导阶段的程序实现是正确的，则推导出的结果也是正确的。

推论6-1：已知长度为n的字符串x和采用IS方法设计的任意后缀数组和最长公共前缀数组构造算法A，当满足以下条件时，算法A的程序所输出的和大概率为x的后缀数组和最长公共前缀数组：

1. 使用算法5-1的程序验证和，验证结果为真。
2. A的程序的最初递归层的解推导阶段的实现是正确的。

可从推论6-1得到同步构造和验证后缀数组和最长公共前缀数组的算法6-1：

1. 使用算法A的程序计算最初递归层的sa\*和lcp\*。
2. 使用算法5-1的程序验证sa\*和lcp\*。
3. 执行算法A的程序的最初递归层的解推导阶段来计算sa和lcp。

## 6.3算法二

算法6-1 是一个概率性验证算法，其使用算法5-1验证x中所有S\*后缀的后缀数组sa\*和最长公共前缀数组lcp\*。本节提出一个专用于验证后缀数组的确定性验证算法。在描述算法细节之前，我们重新定义并引入一个新的参数符号。

给定使用IS方法设计的任意后缀数组构造算法A，sa\*和均是由A在解推导阶段计算得到的数组，两者均记录了所有S\*类型后缀的字典序。其中，sa\*是A从计算得到的，是A在推导S类型后缀时计算得到的。推论6-2给出了一组验证sa\*是否正确的充分条件：

推论6-2：已知长度为n的字符串x和使用IS方法设计的后缀数组构造算法A，当满足以下条件时，算法A的程序所输出的是x的后缀数组：

1. 是所有S\*类型后缀的一个全排列。
2. 对任意，。其中是x中S\*类型后缀的个数。
3. 算法A的最初递归层的解推导阶段的实现是正确的。

证明：分两种情况证明。情况1：假设x中的所有S\*类型后缀均在中出现并按字典序排列，则根据引理6-1可知A的程序所输出的是正确的。情况2：假设中任意两个S\*后缀的排序不正确，即：且。根据条件2)，。若和分别由和推出，则和为两个S\*类型子串，于是有且。已知条件3）成立，将替换为()并重复前述推理过程，则我们最终可根据条件1）得到且，其中且。然而，若，则我们将在解推导过程中得到，于是由条件2）可知，出现矛盾。证毕。

可从推论6-2得到同步构造和验证后缀数组的算法6-2：

1. 使用算法A的程序计算最初递归层的sa\*。
2. 执行算法A的程序的最初递归层的解推导阶段来推导L和S类型后缀。此过程需按字典序遍历所有S类型后缀，在遍历时将所有S\*类型后缀记录在中。
3. 比较sa\*和中的元素。

算法6-2分别在步骤1和2将sa\*和保存在外存文件中，磁盘空间需求较大。因为步骤2需从后向前遍历sa\*和中的元素，可首先使用Karp-Rabin指纹技术在遍历的过程中计算sa\*和的指纹，然后在解推导阶段执行完毕后比较指纹来判断sa\*和是否相等，从而无需在外存中保存这两个数组。

## 6.4 复杂度

算法6-1中的验证开销主要为使用算法5-1验证sa\*和lcp\*，时、空和I/O复杂度分析请参阅第五章。算法6-2中的验证开销主要为读、写和比较和中的元素，时、空和I/O复杂度均线性正比于字符串x的长度n。在算法6-2中，若使用Karp-Rabin指纹技术计算和的指纹，则用于验证的时间为，而空间和I/O量可忽略不计。

## 6.5 实验分析

本节中的实验环境与第三章相同，不再复述。我们比较以下两个构造和验证后缀数组的外存解决方案：

1. 在算法6-2中使用DSA-IS+算法构造后缀数组。
2. 使用eSAIS算法构造后缀数组并使用已知的后缀数组事后验证算法验证eSAIS算法的计算结果。

图6-1至6-3给出了两个方案在磁盘峰值、I/O量和运行时间上的统计数据。观察可知：方案1中的验证开销与其构造开销相比可忽略不计，而方案2中的验证开销占总开销的比例较大。我们在表6-1中详细给出了方案2在构造过程和验证过程中的开销。观察可知：方案2的验证时间是构造时间的五分之一且其在验证过程中达到磁盘峰值。

|  |
| --- |
| 图6-1 验证解决方案_磁盘峰值.png |
| 图6-1 方案1和2在多个测试数据集上的磁盘峰值 |

|  |
| --- |
| 图6-2 验证解决方案_IO量.png |
| 图6-1 方案1和2在多个测试数据集上的IO量 |

|  |
| --- |
| 图6-3 验证解决方案_运行时间.png |
| 图6-1 方案1和2在多个测试数据集上的运行时间 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 表6-1 方案2在验证和构造过程中的性能统计 | | | | | | |
| 测试集 | 验证过程 | | | 构造过程 | | |
| PDU | IOV | CT | PDU | IOV | CT |
| enwiki\_16g | 26.0 | 53.0 | 0.71 | 23.5 | 205.6 | 3.49 |
| guten\_16g | 26.0 | 53.0 | 0.79 | 23.4 | 195.2 | 3.20 |
| uniprot | 25.9 | 53.0 | 0.74 | 22.7 | 162.0 | 2.50 |
| proteins | 25.9 | 53.0 | 0.58 | 24.1 | 172.3 | 2.33 |

## 6.6 本章总结

本章提出两个后缀数组事中验证算法，算法需与采用IS方法设计的任意后缀数组构造算法联用。我们在此基础上提出了两个同步构造和验证后缀数组的算法，其中算法6-2的验证复杂度与其构造复杂度相比可忽略不计。与算法5-2相同，这两个算法均假设构造过程的最初递归层的实现是正确的。目前，采用IS方法设计的内存和外存后缀数组构造算法的性能在同功能算法中处于领先地位。本章提出的两个算法拓展了IS方法的应用范畴，提供了同步构造和验证后缀数组的候选解决方案。

# 结 论

本文提出了若干算法用于解决前、后缀索引的查找、构造和验证问题，并在理论和实验两个方面分析了算法的性能。

第一章提出的流水路由查找系统采用随机分布策略将路由表的前缀树索引均匀地映射到系统的多个存储块中。实验结果表明：较之目下流行的方案，我们所提出的方案能够长期保持存储负载和工作负载的平衡。

第二章提出的后缀数组外存构造算法采用归纳排序方法计算输入字符串的后缀数组。该算法继承了其内存原型的线性时、空复杂度。实验结果表明：算法程序的性能优于同时期的其他外存算法。该算法的设计和实现仍有较大的改进空间，我们近期的工作发现：通过使用文件块链技术和新的子串排序、命名方法，可以将算法程序的磁盘空间降低到7.25n，接近理论最优值。

第三章提出的有限阶最长公共前缀数组构造算法的通用性佳、可部署在内存、外存和分布式模型。特别地，算法在分布模型上的通信和工作负载被均匀地分摊给各计算节点。目前，算法的性能尚有待优化，我们将在后续工作中给出改进方案。

第四章提出的两个后缀数组和最长公共前缀数组事后验证算法可与任意构造算法联用。特别地、第一个算法可以检测稀疏或完整后缀数组和最长公共前缀数组。这一工作填补了该领域的研究空白。

第五章提出的两个后缀数组事中验证算法需与采用IS方法设计的后缀数组构造算法联用。实验结果表明：当同步构造和验证后缀数组时，第二个算法的程序的开销可忽略不计。目前，IS方法设计的后缀数组内存和外存算法在同功能算法中处于领先地位。本章提出的算法具有构建同步构造和验证后缀数组的高效解决方案的潜力。

# 参考文献

1. V. Srinivasan and G. Varghese. Fast IP lookups using controlled prefix expansion, ACM Trans. Comput. Commun, 1999, pp. 1-40.
2. M. Berger. IP lookup with low memory requirement and fast update, Proc. IEEE HPSR, 2003, pp. 287-291.
3. S. Hsieh, Y. Huang and Y. Yang. A novel dynamic router-tables design for IP lookup and update, IEEE Int. Con. On Future Information Technology, 2010, pp. 1-6.
4. S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE J. Select. Areas Commun. (1999) 1083–1092.
5. S. Sahni, K. Kim, Efficient construction of multibit tries for IP lookup, IEEE/ACM Trans. Netw. (2003) 650–662.
6. M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: Proc. ACM SIGCOMM, 1997, pp. 3–14.
7. V. Ravikumar, R. Mahapatra, J. Liu, Modified LC-trie based efficient routing lookup, in: Proc. IEEE Int. Symp. on MASCOTS, 2003, pp. 177–182.
8. S. Sahni, K. Kim, Efficient construction of fixed-stride multibit tries for IP lookup, in: IEEE Workshop on Future Trends of Distributed Computing Systems, 2001, pp. 178–184.
9. S. Sahni, K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, in: Proc. Symp. on Applications and the Internet, 2002, pp. 220–227.
10. S. Sahni, H. Lu, Dynamic tree bitmap for IP lookup and update, in: IEEE Int. con. on Networking, 2007, pp. 79–84.
11. Y. Chang, Y. Lin, C. Su, Dynamic multiway segment tree for IP lookups and the fast pipelined search engine, IEEE Trans. Comput. (2010) 492–506.
12. H. Lim, C. Yim, E. Swartzlander, Priority tries for IP address lookup, IEEE Trans. Comput. (2010) 784–794.
13. W. Lu, S. Sahni, Recursively partitioned static IP router-tables, IEEE Trans. Comput. (2010) 1683–1690.
14. W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, ACM SIGCOMM Comput. Commun. Rev. (2004) 97–122.
15. M. Abouelhodaa, S. Kurzb and E. Ohlebuscha. Replacing suffix trees with enhanced suffix arrays, J. Discr. Algorithms, vol. 2, no. 1, pp. 53-86, Nov. 2004.
16. U. Manber and G. Myers. A new method for online string searches, SIAM J. Comput., vol. 22, no. 5, pp. 935-948.
17. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction, Proc. 30th Int. Colloq. Autom. Languages Program., 2003, pp. 943-955.
18. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays, Proc. 14th Annu. Symp. Combinatorial Pattern Matching, 2003, pp. 200-210.
19. D. K. Kim, J. S. Sim, H. Park and K. Park. Linear time construction of suffix arrays, Proc. 14th Annu. Symp. Combinatorial Pattern Matching, 2003, pp. 186-199.
20. G. Nong, S. Zhang and W. H. Chan. Two efficient algorithms for linear time suffix array construction, IEEE Trans. Comput., vol. 60, no. 10, pp. 1471-1484, Oct. 2011.
21. R. Dementiev, J. Kärkkäinen, J. Mehnert and P. Sanders. Better external memory suffix array construction, ACM J. Exp. Algorithmics, vol. 12, no. 3, pp. 4:1-4:24, Aug. 2008.
22. P. Ferragina, T. Gagie and G. Manzini. Lightweight data indexing and compression in external memory, Algorithmica, vol. 63, no. 3, pp. 707-730, 2012.
23. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm, Algorithmica, vol. 40, pp. 30-50, Sep. 2004.
24. J. Kärkkäinen and D. Kempa. Engineering a lightweight external memory suffix array construction algorithm, Proc. 2nd Int. Conf. Algorithms Big Data, 2014, pp. 53-60.
25. G. Nong, W. H. Chan, S. Zhang and X. F. Guan. Suffix array construction in external memory using D-critical substrings, ACM Trans. Inform. Syst., vol. 32, no. 1, pp. 1:1-1:15, Jan. 2014.
26. G. Nong, W. H. Chan, S. Q. Hu and Y. Wu. Induced sorting suffixes in external memory, ACM Trans. Inform. Syst., vol. 33, no. 3, pp. 12:1-12:15, Mar. 2015.
27. J. Fischer. Inducing the LCP-array, Algorithms in Data Struct., vol 6844, pp. 374-385. 2011.
28. P. Flick and S. Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays, Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal., 2015, pp. 1-10.
29. T. K. G. Lee, H. Arimura, S. Arikawa and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications, Proc. 20th Annu. Symp. Combinatorial Pattern Matching, 2009, pp. 124-135.
30. J. Kärkkäinen, D. kempa and S. J.Puglisi. Parallel external memory suffix sorting, Proc. 26th Annu. Symp. Combinatorial Pattern Matching, 2015, pp. 329-342.
31. T. Bingmann, J. Fischer and V. Osipov. Inducing suffix and LCP arrays in external memory, In proceedings of the 15th Workshop on Algorithm Engineering and Experiments, 2012, pp. 88-102.
32. J. Kärkkäinen, D. Kempa, S. J. Puglisi and et al. Engineering external memory induced suffix sorting. In proceedings of the 19th workshop on Algorithm Engineering and Experiments, 2017, pp. 98-108.
33. J. Kärkkäinen, G. Manzini and S. J. Puglisi. Permuted longest-common-prefix array. In proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching, 2009, pp. 181-192.
34. S. J. Puglisi and T. Andrew. Space-time tradeoffs for longest-common-prefix array computation. In proceedings of the 19th International Symposium on Algorithms Computing, 2008, pp. 124-135.
35. M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In proceedings of the 18th ACM SIGPLAN Symposium on Principles Practice Parallel Program, 2013, pp. 197-206.
36. V. Osipov. Parallel suffix array and least common-prefix for the GPU. In proceedings of the International Symposium on String Processing Information Retrieval , 2012, pp. 379-384.
37. L. Wang, S. Baxter and J. Owens. Fast parallel suffix array on the GPU. In Proceedings of the 21st International Conference on Parallel Distributed Computing, 2015, pp. 573-587.
38. J. Kärkkäinen, D.Kempa and S. J. Puglisi. Parallel external memory suffix sorting. In Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching, 2015, pp. 329-342.
39. J. Kärkkäinen and D. Kempa. Faster external memory LCP array construction. In Proceedings of the 24th European Symposium on Algorithms, August 2016, pp. 61:1-61:16.
40. G. Manzini. Two space saving tricks for linear time LCP array computation. In proceedings of the 9th Workshop on Algorithm Theory, pages 372-383, Humlebaek, Denmark, July 2004.
41. J. Kärkkäinen and D. Kempa. LCP Array Construction in External Memory. In proceedings of the 13th International Symposium on Experimental Algorithms, pages 412-423, Copenhagen, Denmark, June 2014.
42. M. Bauer, A. C. G. Rosone and M. Sciortino. Lightweight LCP Construction for Next-Generation Sequencing Datasets. In proceedings of the 12th International Workshop on Algorithms in Bioinformatics, pages 326-337, Ljubljana, Slovenia, 2012.
43. P. Bille, I. L. GØrtz, T. Kopelowitz, B. Sach and H. W. VildhØj. Sparse Suffix Tree Construction in Small Space. In proceedings of the 40th International Colloquium on Automata, Languages, and Programming, 148-159, Riga, Latvia, July 2013.
44. S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In proceedings of the 14th Symposium on Combinatorial Pattern Matching, 2003, pp. 55-69.
45. A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. IEEE/ACM Transactions on Networking, 2005, 690-703.
46. K. Kim and S. Sahni. Efficient construction of pipelined multibit-trie router-tables, IEEE Transactions on Computers, 2007, 32-43.
47. F. aboescu, D. Tullsen, G. Rosu, S. Singh. A tree based router search engine architecture with single port memories. In Proceedings of the International Symposium on Computer Architecture, 20005, pp. 4-8.
48. S. Kumar, M. Becchi, P. Corwley and J. Turner. CAMP: fast and efficient IP lookup architecture. ACM/IEEE Symposium on Architecture for Networking and Communications Systems, 2006, pp. 51-60.
49. Y. Li, D. Zhang, K. Huang, D. He and W. Long. A memory-efficient parallel routing lookup model with fast updates. Elsevier Journal of Computer Communications, 2014, 60-71.
50. Y. Li, D. Zhang, K. Huang, D. He and W. Long. Scalable tree-based architecture for IPv4/IPv6 lookup using prefix partitioning. IEEE Transactions on Computers, 2012, pp. 3039-3052.
51. R. Jangid, C. P. Gupta, I. Sharma. Prefix length-based disjoint set tries for ipv6 lookup. In proceedings of the ICT for Sustainable Development, Singapore, 2016.
52. T. Anderson, S. Owicki, J. Saxe and C. Thacker. High-speed switch scheduling for local area networks. ACM Transactions on Computing Systems, 1993, 319-352.
53. N. McKeown. The islip scheduling algorithm for input-queued switches. IEEE/ACM Transactions on Networking, 1999, 188-201.
54. L. Arge and M. Thorup. RAM-efficient external memory sorting. Algorithms Computing, vol. 9293, no. 3, pp. 491-503, 2013.

# 

# 攻读博士（硕士）学位期间发表学术论文情况

1. **吴裔**,农革, Mournir Hamdi. Scalable pipelined IP lookup with prefix tries. Elsevier Journal of Computer Networks, 2017年， 120:1-11. SCI检索期刊，本文SCI检索号：000401877300001。（本博士学位论文第二章）
2. 农革，陈伟康，胡圣青，**吴裔**. Induced Sorting Suffixes in External Memory, ACM Transactions on Information Systems, 2015. （本博士学位论文第三章）
3. **吴裔**，农革，陈伟康，劳斌. Building and checking suffix array by induced sorting method. to be submitted。（本博士学位论文第三章和第五章）
4. **吴裔**，韩凌波，陈伟康，农革. Scalable K-order LCP array construction for massive data. Proceedings of the 8th IEEE Parallel Architectures, Algorithms and Programming, 2017年, EI检索会议，本文EI检索号：尚未提供。（本博士学位论文第四章）
5. **吴裔**,农革,陈伟康,韩凌波. Checking big suffix and LCP arrays by probabilistic methods. IEEE Transactions on Computers, 2017年, 66(10): 1667-1675. SCI检索期刊，本文SCI检索号：尚未提供。（本博士学位论文第五章）

# 致 谢

学位论文中不得书写与论文工作无关的人和事，对导师的致谢要实事求是。

一同工作的同志对本研究所做的贡献应在论文中做明确的说明并表示谢意。

这部分内容不可省略。

书写格式说明：

标题“致谢”选用模板中的样式所定义的“标题1”，再居中；或者手动设置成字体：黑体，居中，字号：小三，1.5倍行距，段后11磅，段前为0。

致谢正文选用模板中的样式所定义的“正文”，每段落首行缩进2字；或者手动设置成每段落首行缩进2字，字体：宋体，字号：小四，行距：多倍行距 1.3，间距：前段、后段均为0行。

# 中山大学学位论文版权使用授权书

注：页眉，居中，宋体，五号，填写内容为“中山大学博士（硕士）研究生学位论文”。

阅后删除此文本框。

注：此页为封底。

阅后删除此文本框。

本学位论文作者及指导教师完全了解“中山大学硕士、博士（硕士）学位论文版权使用规定”，同意中山大学保留并向国家有关部门或机构送交学位论文的复印件和电子版，允许论文被查阅和借阅。本人授权中山大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，也可采用影印、缩印或扫描等复制手段保存和汇编学位论文。

作者签名：

导师签名：

注：此页内容不需要任何改修，手写签名和日期即可。

阅后删除此文本框。

年 月 日