**中山大学博士学位论文**

**中文题目：前、后缀索引构造和正确性验证算法的研究与实现**

**英文题目：Research and Implementation on Building and Checking Algorithms for 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：前、后缀索引构造和正确性验证算法的研究与实现

注：奇数页页眉，居中，宋体，五号，填写内容为“中山大学博士（硕士）学位论文”。阅后删此文本框。

专业：计算机科学与技术

博士生：吴裔

指导教师：农革

# 摘 要

对原始数据构造索引可大幅提高在其上执行字符串匹配操作的效率。前、后缀索引是常用的索引数据结构，广泛运用于各类字符串处理程序中。本文研究前、后缀索引的构造和正确性验证问题，所设计的算法适用于内存、外存和主从分布式内存模型。本文的主要贡献如下：

第一、在内存模型上设计一种流水字符串匹配算法。该算法采用任意指针型前缀树组织输入字符串集，树中各节点以随机分布策略存储于由复数个流水阶组成的流水系统。本文采用仿真技术，在路由查找这一具体应用上下文中实现基于binary trie、prefix trie、multi-bit trie和fixed-stride trie的流水路由查找系统。仿真结果表明，该系统的单位时间吞吐量与业界主流技术相近，但其存储开销最小。

第二、在外存模型上设计一种后缀数组构造算法。该算法采用归纳排思想推倒输入字符串中所有后缀的字符序，时空复杂度与外存整数排序相当。实验结果表明：该算法的时间效率与同时期最优算法相当，空间效率是后者的两倍。在此基础上，本文提出一种空间优化方法，优化后的算法计算40-bit编码的后缀数组的工作空间为1.5n，n为输入字符串的长度，趋近理论最有值。

第三、在外存模型上设计后缀数组和最长公共前缀数组的两个事后验证算法。算法一从数组定义触发，可验证全阶、有限阶的后缀数组及最长公共前缀数组的正确性；算法二首先复用算法一来验证输入数组中的一个子集的正确性，然后采用归纳排序方法验证整个数组的正确性。两个算法均为概率性验证算法，其采用指纹技术计算字符串的哈希值并通过比较哈希值来确定对应字符串之间是否相等，理论时空复杂度与外存整数排序相当。实验结果表明：给定全阶后缀数组及最长公共前缀数组，算法一的磁盘峰值劣于算法二，但其I/O量和速度优于后者。截止本文撰写时期，尚未有工作报告最长公共前缀的事后验证方法，上述工作填补了这一空白。

第四、在外存模型上设计后缀数组的两个事中验证算法。两个算法的设计均采用归纳排序方法。算法一为概率性验证算法，采用指纹函数确定字符串间大小关系，可在外存排序复杂度内同步构造和验证后缀数组，也可在相同时空复杂度内同步构造和验证最长公共前缀数组。算法二为确定性验证算法，可在线性时间和常数空间复杂度内同步构造和验证后缀数组。实验结果表明：较之构造过程的时空开销，算法一的开销可忽略不计，而现存后缀数组验证算法的空间开销较大。

第五、在内存、外存和分布式模型上设计有限阶最长公共前缀数组的概率性构造算法。该算法可在对数时间和线性空间内计算得到输入字符串的有限阶最长公共前缀数组。较之现有确定性和非确定性构造算法，算法可运行于主从分布式系统，各子节点的时空复杂度均衡。实验结果表明：在子节点个数为d的主从分布式系统中，各节点的时空复杂度为总时空复杂度的d分之一，且算法的性能与子节点规模呈近线性增长。

**关键词：前缀和后缀索引；构造和正确性验证；内存、外存和分布式计算模型**

Title: Research and Implementation on Building and Checking Algorithms for Prefix and Suffix Indices

注：偶数页页眉，居中，宋体，五号，论文中文题目。

阅后删除此文本框。

Major: Computer Science and Technology

Name: Yi Wu

Supervisor: Ge Nong

# Abstract

Index technique can dramatically speeds up string matching operations on raw data. Prefix and Suffix indices are index data structure commonly widely applied in various string processing applications, such as text clustering, data mining and genome aligning. This paper is dedicated to finding out novel solutions for prefix and suffix indices construction and validation on internal, external and classic distributed computational models. Our main contribution is summarized as below:

We propose a pipeline string matching system specific for IPv4/IPv6 route lookup on internal memory model. This system uses any trie-based data structure to organize a set of strings, where nodes of a trie are connected together via pointers and scattered among a number of memory blocks consisting of a pipeline system. Searching in the system is cast as a bipartite matching problem and multiple searching requests can be executed in parallel to achieve a high throughput. Our simulation results indicate that this work achieves comparable throughput against the prior arts and performs better in storage requirement.

We propose a suffix array construction algorithm on external memory model. This algorithm sorts all the suffixes for any string drawn from constant or integer alphabet within a sorting complexity according to the induced sorting principle. From our experiments, this algorithm is comparable to the state-of-the-art with respect to time and I/O volume, while its peak disk use is only half as that of the latter. We will also show that, this algorithm can achieve nearly optimal space performance in theory with better design and engineering. For example, the program for the optimized suffix array builder takes no more than 1.5n work space to construct a suffix array encoded by 40-bit integer array, where n is the number of characters in the input string.

We propose two algorithms for checking constructed suffix and longest common prefix arrays. The first algorithm is designed based on the definition of the two arrays. It can be employed to check any finite or infinite-order (sparse) suffix and longest common prefix arrays. The second algorithm mainly consist of two steps, it first reuses the first algorithm to check a subset of the two arrays and then adopts the induced sorting method to check the whole arrays. Both two algorithms make use of a fingerprinting technique to correctly compare strings by their fingerprints with a negligible error probability. When implemented on external memory model, the time, space and I/O complexities of the two algorithms are consistent with a disk-based sorting complexity. We will demonstrate that the second algorithm uses less disk space to run and the first algorithm has an advantage in both speed and I/O volume. At the time of writing this paper, there exist only one work for checking suffix array and no works for checking longest common prefix array. Our work provides the first candidate for checking the two arrays at the same time.

We propose two algorithms for checking a suffix array when it is being built by any induced sorting suffix array builder. The first algorithm is probabilistic, it employs a fingerprinting technique to determine equality of strings and can be implemented on disk within a sorting complexity. Particularly, this algorithm can be extended for checking both suffix and longest common prefix arrays simultaneously. On the contrary, the second algorithm is deterministic and specific for checking suffix array only. Both theoretical analysis and experimental results show that, in comparison with that for suffix array construction, the time and space for validation by this algorithm is negligible and that for checking by the existing checker is considerably large.

We propose a probabilistic algorithm for building K-order longest common prefix array within logarithmic time and linear space, where K is considerably smaller than the length of the whole string. This algorithm is universal for internal, external and distributed computational models, its parallel version can achieve load balance in a distributed system consisting of d computing nodes, leading to a high system scalability.

**Key Words：Prefix and Suffix Indices; Construction and Validation; internal, external and distributed Computational Model**

目 录

注：在该页面中点击鼠标右键，选择“更新域…”，在弹出窗口中选择“更新整个目录”，确定即可自动生成目录。标题“目录”，字体：黑体，字号：小三。章、节标题和页码，字体：宋体，字号：小四。阅后删除此文本框。

[论文题目：前、后缀索引构造和正确性验证算法的研究与实现 I](#_Toc493661009)

[摘 要 I](#_Toc493661010)

[Abstract III](#_Toc493661011)

[第一章 绪论 1](#_Toc493661012)

[1.1研究背景 1](#_Toc493661013)

[1.2研究内容 1](#_Toc493661014)

[第二章 流水路由查找 5](#_Toc493661015)

[2.1 基础知识 5](#_Toc493661016)

[2.2 索引结构 6](#_Toc493661017)

[2.3体系架构 7](#_Toc493661018)

[2.3排队模型和调度算法 8](#_Toc493661019)

[2.3.1排队模型 8](#_Toc493661020)

[2.3.2调度算法 9](#_Toc493661021)

[2.3.3分析 10](#_Toc493661022)

[2.6仿真实验 10](#_Toc493661023)

[2.6.1实验配置 10](#_Toc493661024)

[2.6.2存储需求评估 11](#_Toc493661025)

[2.6.3吞吐量评估 15](#_Toc493661026)

[2.7硬件设计 16](#_Toc493661027)

[2.8本章小结 17](#_Toc493661028)

[第三章 后缀数组构造算法 18](#_Toc493661029)

[3.1基础知识 18](#_Toc493661030)

[3.2算法原理 19](#_Toc493661031)

[3.2.1数据结构 19](#_Toc493661032)

[3.2.1 收缩阶段 20](#_Toc493661033)

[3.2.2 解推导阶段 21](#_Toc493661034)

[3.3算法优化 22](#_Toc493661035)

[3.3.1 设计优化 22](#_Toc493661036)

[3.3.2 工程优化 23](#_Toc493661037)

[3.4实验分析 23](#_Toc493661038)

[3.5本章总结 23](#_Toc493661039)

[第四章 有限阶最长公共前缀数组构造算法 24](#_Toc493661040)

[4.1基础知识 24](#_Toc493661041)

[4.2内存算法设计 24](#_Toc493661042)

[4.3 外存算法设计 25](#_Toc493661043)

[4.4 分布式算法设计 26](#_Toc493661044)

[4.5算法优化 27](#_Toc493661045)

[4.5.1设计优化 27](#_Toc493661046)

[4.5.2工程优化 28](#_Toc493661047)

[4.6实验分析 28](#_Toc493661048)

[4.6本章小结 28](#_Toc493661049)

[第五章 后缀数组和最长公共前缀数组事后验证算法 29](#_Toc493661050)

[5.1 基础知识 29](#_Toc493661051)

[5.2 算法1 29](#_Toc493661052)

[5.2.1 算法原理 29](#_Toc493661053)

[5.2.2算法设计 30](#_Toc493661054)

[5.3 算法2 31](#_Toc493661055)

[5.3.1 算法原理 31](#_Toc493661056)

[5.3.2 算法设计 32](#_Toc493661057)

[5.4 算法优化 33](#_Toc493661058)

[5.4.1 设计优化 33](#_Toc493661059)

[5.4.2 工程优化 33](#_Toc493661060)

[5.5 实验分析 33](#_Toc493661061)

[5.6 本章总结 33](#_Toc493661062)

[第六章 后缀数组事中验证算法参考文献 35](#_Toc493661063)

[参考文献 36](#_Toc493661064)

[结 论 40](#_Toc493661065)

[攻读博士（硕士）学位期间发表学术论文情况 42](#_Toc493661066)

[致 谢 43](#_Toc493661067)

[中山大学学位论文版权使用授权书 44](#_Toc493661068)

# 第一章 绪论

## 1.1研究背景

对文本（集）中的文本段执行聚合操作可快速定位目标字符串在文本（集）中的出现。在各类字符串应用中，对文本（集）构建前、后缀索引是常用的预处理技术。本文主要研究前缀树和后缀数组的构造、查找及验证方法，所设计的算法适用于内存、外存和经典分布式计算模型。

## 1.2研究内容

给定字符串集X和字符串Y，常见的前缀匹配问题包括以下两类：（1）在X中查找以Y为前缀的所有字符串，返回匹配项；（2）在X中查找与Y的任意前缀匹配的所有字符串，返回（最长）匹配项。第一类问题的经典应用场景是词频统计和建议查询，第二类问题的经典应用场景是路由表查找，可通过查找X的前缀树PT求解两问题，时间复杂度为O(d)，d为PT的深度。现有前缀树构造算法包括binary tree、prefix tree[1]、fixed-stride tree[2]和multi-prefix tree[3]等。各算法的主要区别在于结点跨度，binary tree和prefix tree为单结点单跨度前缀树，而fixed-stride tree和multi-prefix tree为单结点多跨度前缀树。为便于说明，表1-1给出了定义在字符集{0,1}上的一个字符串集X，其对应的binary tree和2跨度fixed-stride tree分别如图1-2和1-3所示。图中每个结点对应一个字符串，每条边对应一个分支条件，需要注意的是，fixed-stride tree采用了前缀扩展技术将每个长度不被2整除的前缀扩展为两个子前缀（例如，字符串011被扩展为0110和0111，此情况下的匹配过程还需比较字符串长度）。在图1-1的binary tree上查找以Y为前缀的所有字符串的过程包括两步：第一步，从根结点出发，在深度为i的结点上以Y中第i个字符的值为分支条件下行至深度为i+1的子结点；第二步，递归执行第一步直到子结点不存在或Y的第i个字符不存，返回以当前结点为根的子树所表示的解空间。现有工作表明，将前缀树映射到多阶流水系统上可实现多任务并行执行，但前缀树上的动态更新操作会导致各流水阶之间的工作负载和存储需求不均衡。为解决该问题，本文提出一种新型流水系统，该系统运用于路由表查找这一具体上下文中，可在处理大量前缀树更新请求后依然保持流水系统的负载均衡。

|  |  |  |  |
| --- | --- | --- | --- |
| 表1-1 字符串集X示例，字符集{0, 1} | | | |
| # | string | # | string |
| 1 | 0 | 6 | 011 |
| 2 | 00 | 7 | 110 |
| 3 | 01 | 8 | 1101 |
| 4 | 000 | 9 | 01001 |
| 5 | 010 | 10 | 01010 |

|  |  |
| --- | --- |
|  |  |
| 图1-1 binary tree示例 | 图1-2 2-fixed-stride tree示例 |

后缀树是另一种重要的索引数据结构。较之前缀树，后缀树一般用于组织单个字符串中的所有后缀。给定字符串X和Y，可在X的后缀树ST上查找Y的所有出现，时间复杂度为O(l)，l为Y的长度。后缀树包含结点、父子指针和后缀链，空间开销大。后缀数组是后缀树的一个精简替代，存储空间小，可与最长公共前缀联合模拟后缀树上的查找过程。现有多个工作针对不同计算模型设计了高校的后缀数组构造算法。内存模型上的最快线性算法是SA-IS，可用于构造常数和整数字符集上的后缀数组。该算法的核心思想是归纳排序，通过比较两个后缀的首字符和后继后缀来确定大小关系。目前已有多个工作利用归纳排序思想设计外存后缀数组构造算法eSAIS、DSA-IS和SAIS-PQ，其理论时、空复杂度优于DC3、bwt-disk、SAscan和pSAscan等其他算法。然而，这些三个外存算法的实际磁盘峰值和I/O仍然处于较高水平。分析实验数据可知，pSAscan构造一个40-bit整数编码的后缀数组需要7.5n的磁盘空间，而eSAIS、DSA-IS和SAIS-PQ在同等条件下分别需要24n、18n和15n。近期的一份工程研究报告指出，采用文件块链技术可大幅改进后缀数组构造过程中的磁盘表现。本文从改进算法设计和程序实现等两方面出发，在DSA-IS算法的基础上提出新的后缀数组构造算法。新算法的运行速度和I/O量优于原算法，且其磁盘峰值达到7.5n，趋近理论最优值。

结合最长公共前缀数组，后缀数组可模拟后缀树的自顶向下或自底向上的遍历过程。近年来，前缀树构造算法的研究工作也取得了一定的进展。已知多个最长公共前缀构造算法，可按算法的输入将其分为两大类：第一类算法同时计算后缀数组和最长公共前缀数组，输入为原始字符串；第二类算法除要求原始字符串外还需输入Burrows-Wheeler变换或后缀数组等其他辅助数据。在内存模型上，Kasai等人提出了一个线性时间构造最长公共前缀数组的方法，该方法的输入包括原始字符串、后缀数组和逆后缀数组，Mazini等人优化了该方法的空间性能。Kärkkäinen等人提出一个二步计算最长公共前缀数组的方法，该方法首先计算转置最长公共前缀数组，然后将此中间量在线性时间内转换为最终结果。Kärkkäinen等人还提出了该方法的外存版本LCPscan，时空复杂度与内存大小成反比。作者提供的实验数据表明，当原始字符串与可用内存之比较小时，LCPscan的运行速度好于eSAIS；当原始字符串与可用内存之比变大时，LCPscan的性能衰减更快，最终将劣于eSAIS。目前，时空复杂度最优的最长公共前缀数组构造算法采用归纳排序思想，可在线性时间内同时计算得到后缀数组和最长公共前缀数组，适用于内存和外存计算模型。除了上述串行算法，已有多个工作致力于设计在共享内存模型上的后缀数组和最长公共前缀数组构造算法。这些工作利用多核CPU/GPU的计算能力，取得了不错的效果。在实际应用中，原始字符串通常由复数个子串前后相连而成，可设计算法构造字符串集合的广义后缀数组和最长公共前缀。Baul等人设计的eGSA算法可同时计算输入字符串的广义后缀数组和最长公共前缀数组，该算法采用分治法和多路合并思想将子问题求解得到的子数组合并为最终结果。Bille等人设计的exLCP算法可在外存模型上计算广义最长公共前缀数组，算法首先计算原始字符串集的广义Burrows-Wheeler变换，然后将其转换为最终结果。exLCP和eGSA等算法的缺点在于逻辑复杂，难以扩展至并行和分布式计算模型上。为解决该问题，本文提出一种有限阶广义最长公共前缀数组构造算法，该算法的设计简单、扩展性好，可实施于内存、外存和传统分布式计算模型上。

互联网技术和采样技术的发展导致待处理数据规模不断膨胀，间接推动了索引构造技术的进步。以（广义）后缀数组和最长公共前缀数组的构造算法为例，单机内存算法可处理GB级数据，单机外存算法可处理TB级数据，分布式内存/外存算法可处理PB级数据。然而，随着算法可处理数据规模的增长，其设计和实现的复杂性也不断提高。例如：内存SA-IS算法的代码不到200行，但外存DSA-IS算法的代码超过3000行。目前，所有开源的后缀数组和最长公共前缀数组的构造算法均不能保证程序无错。此外，即便算法的实现没有问题，其运行平台上的软、硬件故障也可能导致错误的计算结果。因此，需要提供验证技术来检测结果的正确性。Burkhardt 等人设计的算法是目前唯一可检测后缀数组正确性的方法[33]，其内存和外存实现分别包含在DC3和eSAIS的软件包内。此外，并无任何文献记载最长公共前缀数组的验证方法。本文提出事中和事后两类验证算法。第一类算法在数组的构造过程中同步验证其正确性，第二类算法在数组构造完毕后验证其正确性。第一类算法包括两个概率性验证算法，可在排序复杂度内同时检测输入的后缀数组和最长公共前缀数组的正确性，验证出错的概率可忽略不计，适用于单机和外存模型。第二类算法包括一个概率性验证算法和一个确定性验证算法。其中，概率性验证算法可在排序复杂度内验证构造中的后缀数组和最长公共前缀的正确性，验证出错的概率可忽略不计；确定性验证算法可在线性时间和常量空间内验证构造中的后缀数组的正确性。第二类算法需与采用归纳排序思想的后缀数组（和最长公共前缀数组）构造算法联合使用。

# 第二章 流水路由查找

## 2.1 基础知识

IP路由对数据包执行两步操作：路由和交换。换言之，数据包在到达路由的某个输入端口后将先后经历路由阶段和交换阶段。在路由阶段，路由提取数据包的目的IP地址并查找路由表中与该地址拥有最长公共前缀的匹配项来获取通向下一跳的转发端口。在交换阶段，路由将数据包从输入端口调度至指定的输出端口。由于骨干路由的吞吐量需求在不断上升，如何提高包处理速度依然面临着严峻的挑战。

过去二十年，前缀树是解决路由查找问题的关键数据结构，可将路由表中的IP地址前缀组成一个字符串集合并使用任意前缀树构造算法构造该集合的前缀索引树，树中各结点以指针相连。当数据包到达路由的某个输入端口时，路由系统创建一个查找请求，在前缀索引树上查找与包内目的地址匹配的最长前缀。该查找过程可描述如下：从根结点出发，按目的地址对应位的字符值为分支判定条件从当前结点移动到某个孩子结点，迭代执行该过程直到到达某个叶子结点或达到特定条件。目前已有多个工作研究路由表的前缀索引树构造算法[1][2][3][4][5][6][7][8][9][10][11][12][13][14]。表2-1是由表1-1扩展得到的一张路由表，各表项分别指定了下一跳信息。如图2-1所示，各结点代表0个或1个路由表中的前缀字符串，各边标记分支判定条件。假设数据包的目的地址D为0100111，则在binary tree中查找与D匹配的最长前缀的过程为：因D的首字符为0，从根结点的左分支出发访问左孩子得到一个长度为1的匹配前缀字符串0\*；因D的次字符为1，从当前结点的右分支出发访问右孩子得到一个长度为2的匹配前缀字符串01\*；重复上述过程，依次发现010\*和01001\*，从中选择最长的01001\*并返回对应的下一跳信息E9。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表2-1 路由表示例 | | | | | |
| 编号 | 前缀字符串 | 下一跳 | 编号 | 前缀字符串 | 下一跳 |
| 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 |

在前缀树索引上并行查找可提高查找效率。由于路由表可能随时发生变化，其对应前缀树也许及时更新或重构。因此，软件实现的并行查找一般通过不同粒度的对象锁保证数据一致性。可将前缀树结点映射到一个硬件实现的流水系统中，该系统包括多个配备有专用存储块的流水阶，支持并行访问位于不同流水阶中的前缀结点。已知存在多个采用前缀树索引的流水路由体系结构。文献[34][35]提出了采用fixed-stride trie的线形流水路由体系结构，核心思想是将前缀树的根结点存储至第一流水阶，其下各层结点依次存储至后继流水阶。文献[36][37]提出了环形流水路由体系结构，核心思想是将前缀树划分为多棵子树，各子树的根结点按一定规律分散储存到某一流水阶，其下各层结点依次存储至后继流水阶。近年出现的一些研究工作（例如：[38][39][40]）首先将路由表中的前缀划分为多个不相关子集，然后将为各子集分别构建前缀树并将树结点映射到线形或环形流水系统中，由于不同子集上的路由查找可并行执行，从而进一步提高了总的吞吐量。

线状流水系统存在以下问题：（1）各流水阶的存储需求不均衡。原因包括：一是前缀树低层结点数远少于高层结点数；二是路由表中前缀字符串的长度分布不均衡，以IPv4路由表为例，16和24比特前缀字符串占表70%以上；三是路由表的长期增量更新使得对应前缀树上发生大量结点增删（2）各流水阶的访问负载不均衡。这主要是因为前缀索引树中的大部分叶子结点对应16和24比特前缀字符串，这导致24比特前缀所在流水阶的后继流水阶的访问频次远小于其他流水阶。环状流水系统同样存在上述问题。本章提出一种新型流水路由体系结构，该体系结构同样采用前缀索引树来组织路由表的前缀项，但其将树中结点随机分布到各流水阶中以平衡存储需求和访问负载。

## 2.2 索引结构

索引由一个快表和一个前缀索引森立构成。将路由表中前缀分为长、短两类：若长度超过L比特则为长前缀，否则为短前缀。对所有长前缀，按前L比特将其划分为2^L个不相关子集，采用一种前缀索引树构造算法将每个子集转换为一棵前缀索引树；对所有短前缀，将其插入到一个包含2^L个表项的快表中，表中各项由以下数据域组成：

* 扩展前缀：每个短前缀对应于一个L比特的扩展前缀，该扩展前缀为短前缀右侧补零所得。例如：给定L=3，则01\*和1\*的扩展前缀分别为010\*和100\*。
* 掩码数组：最多有L个短前缀共享一个扩展前缀。使用一个长度为L比特的掩码数组来标记哪些短前缀出现在路由表中。例如：给定扩展前缀000\*，若对应掩码数组的前两比特均为1，则表示0\*和00\*出现在路由表中。
* 下一跳数组：与掩码数组一起使用，第i个数据项纪录掩码数组第i比特对应的短前缀的下一跳信息。
* 根结点指针：同一子集中的长前缀的前L比特相同，可将前L比特唯一标识长前缀子集。该数据域存储对应长前缀子集的前缀索引树的根结点。

将一个长度为*l*的前缀p插入索引中的过程如下：若p为短前缀，则将其转化为对应的扩展前缀，以扩展前缀为键查询快表得到对应表项，更新表项中的掩码数组和下一跳数组。若p为长前缀，则以其前L比特为键查询快表得到对应表项，获取表项中的根结点指针，将p插入到该根结点所在的前缀索引树中。假设L=3，图2-1中路由表的索引的快表和前缀索引森林分别如表2-2和图2-3所示。其中，前缀0\*，00\*和000\*的扩展前缀均为000\*，因此它们被插入到快表的同一表项，该表项的掩码数组中的各比特位均为1且下一跳数组中记录了E1，E2和E4等三个前缀的下一跳信息。对于路由表中的长前缀1101\*，其所在前缀索引树的根结点记录在扩展前缀为110\*的快表项中。

|  |  |  |  |
| --- | --- | --- | --- |
| 表2-2 快表示例 | | | |
| 扩展前缀 | 掩码数组 | 根结点指针 | 下一跳数组 |
| 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\* | 001 | to trie B | (-, -, E7) |
| 111\* | 000 | null | (-, -, -) |

|  |
| --- |
| 图2-3 前缀索引森林示例 |
|  |

## 2.3体系架构

图2-4给出了所提出的路由体系结构，其包括：（1）多个存储单元（一般为SRAM），用于存储索引；（2）一个缓冲单元，用于排队等待路由的数据包；（3）一个调度单元，用于控制缓冲数据包的查找请求的执行过程。该体系结构将快表存储到一个单独的存储单元中并将前缀森林中的每个结点映射到{m1, m2,…,mM}中的存储单元，其中{m1, m2,…,mM}构成一个流水线。为使{m1, m2,…,mM}得存储需求平衡，按如下方式将前缀索引中的每个结点映射到流水线的某个存储单元中：使用一个随机数生成装置按等概率从{1,2,…,M}中选取一个数，将前缀结点存储到以该数为下表的存储块中。长期来看，每个存储快被选取的次数趋近1/M，从而实现存储需求的平衡。

已知索引的存储方式，每个新到的数据包首先在快表中查找最长匹配前缀，如果快表中对应表项的根结点指针域为空则查找结束；否则该数据包被缓存在缓冲单元内等待下一步处理。系统为缓冲单元内的每个数据包创建一个查找请求，该查找请求需要由调度单元授权访问位于{m1, m2,…,mM}的前缀树结点来获得最长匹配前缀。由于前缀树中父子关系的结点可能位于不相邻的流水阶的存储块内，一个查找请求可能需要跳跃地访问流水线来执行整个查找过程，当其当前访问的前缀树结点存储着最长匹配前缀时，查找请求返回数据包转发信息且数据包离开缓冲单元。必须注意的是：若多个查找请求同时访问同一个存储单元，则将造成访问冲突；反之，若多个查找请求若同时访问不同的存储单元，则可并行执行。假设系统时间被等比切分为离散的时间片且每个存储块在单个时间片内仅被允许访问依次，若想最大化流水路由体系结构的吞吐量，就必须在每个时间片内尽可能调度更多的可无冲突访问存储块的查找请求。

## 2.3排队模型和调度算法

### 2.3.1排队模型

为了建立路由体系结构的排序模型，我们给出如下一般假设：

* 系统时间被切分为固定大小的时隙。
* 路由缓冲区容量无线。
* 数据包仅在某个时隙的开始达到。
* 在每个时隙的结尾，调度单元将从缓冲单元中的数据包中选取一个无冲突访问的集合并授权集合中的数据包的查找请求执行一次访问操作。若一个查找请求在执行完当前访问操作后找到最长匹配前缀，则对应的数据包离开缓冲单元。

我们引入参数N来表示在缓冲单元中的数据包的数量。图2-x给出了路由体系结构的排队模型。在该模型中，每个数据的查找请求需访问一次或多次存储单元。

|  |
| --- |
| 图2-x |
|  |

### 2.3.2调度算法

可将调度问题转化为一个二分图匹配问题。定义缓冲单元中的每个数据包和流水线中的每个存储块为图中的一个顶点，从数据包顶点到存储块顶点之间的一条边表示数据包需要访问存储在存储块中的某个前缀树结点。调度的目标是寻找一个最大的边集合，使得每个结点（无论是数据包结点还是存储块结点）最多只有一条边与其相连。调度单元必须使用某种冲突仲裁策略（例如：先来先服务）解决复数条边与单个结点相连的情况。注意到单个查找请求可能需要访问多次存储块且每次访问需消耗一个系统时间片，我们将一个t阶路由查找请求表示为R(s1, s2,…,si,…,st)的形式。其中，si取自{m1, m2 . . . mM}，表示该查找请求在第i步需访问的存储块。例如，R(m1, m2, m5, m4, m3)表示一个需在第1,2,3,4,5阶段依次访问m1, m2, m5, m4, m3的查找请求。在二部图模型的基础上，我们给出一个并行调度的算法框架：

1. 每个查找请求发送一个访问请求至目标前缀树结点所在的存储块。
2. 每个存储块按一定策略从接收到的访问请求中选取一个并通知调度单元给予授权。
3. 每个被授权的查找请求执行访问操作。

按此调度方式，多个数据包的无冲突访问操作可在一个时隙内并行完成，从而实现流水作业。表2-x和表2-y给出了一个具体的运行实例，该实例在上述算法框架的步骤2采用FCFS策略来解决可能出现的访问冲突。假设流水线中的存储块的数量为4且缓冲单元初始为空，数据包P1 − P8分别在时隙0–7到达路由且系统分别为其创建查找请求R1-R8。当生成一个查找请求后，该请求将在每个时间片反复争抢目标存储块的使用授权直到其完成查找任务。例如，R1在第0时隙请求访问存储块m1，因为当前没有其他请求，R1得到访问授权。如表2-y所示，第0时隙的最大匹配集（MMS）为{(R1,m1}。在第1时隙，R2参与到调度中并请求访问m1，而R1请求访问m2。因为R1和R2的目标存储单元不相同，从而它们都得到访问授权且该时隙的最大匹配集为{(R1, m2), (R2, m1)}。在第2时隙，R1 和 R2同时请求访问m3，从而发生访问冲突。根据先来先服务原则，R1在争抢中获胜并得到访问授权，从而该时隙的最大匹配集为{(R1, m3), (R3, m1)}。依据上述过程继续完成执行时隙3-12的调度作业，我们就可以得到表2-xx中的最终结果。

### 2.3.3分析

在每个时隙，调度单元构造和执行最大匹配集，从而流水线的效率和系统吞吐量与最大匹配集的平均大小成正比。为了便于分析，令P(M,N)表示N个查找请求访问某个存储块的概率。因为前缀索引森林中的所有结点被随机地均匀分布到各流水阶中，所以任意查找请求访问任意存储块的概率ρ=1/M，从而我们得到P(M,N)的计算公式为：。图2-x给出了应变量P(M,N)与自变量N之间的变化趋势图，其中M分别取定值2,4,8,16。由图可知，虽然P(M,N)随着M的增大而变小，但系统的总体性能仍然在不断提升。这表明我们可以通过增大M和/或N来扩展系统的总吞吐量。

在多存储块路由体系结构中，如何调度数据包的查找请求是影响系统吞吐量的关键要素。在本章仿真实验中，我们使用如下调度算法：假设缓冲单元中的数据包按达到顺序排列且所有数据包均被允许参与调度。在每个时隙的开始，若缓冲单元非空，调度单元将执行指定的调度算法来寻找数据包和存储块之间的最大匹配集。然后，数据包的查找请求访问存储快来获取目标前缀树结点。为了寻找最大匹配集，我们实例化2.3节中的算法框架，采用先来先服务的策略处理步骤2中可能发生的访问冲突。在每个时隙的开始，所有数据包和存储块都处于未匹配状态。为了解决多个数据包访问同一存储块所引起的访问冲突，调度单元根据先来先服务策略授权“年龄最大”的数据包访问存储块。具体调度过程如下所示：

* 请求：每个未匹配的数据包发送一个请求至目标前缀结点所在存储单元。
* 授权：若一个未匹配的存储块接收到一个或多个请求，则其选取 “年龄最大”的数据包并通知调度单元授权访问。
* 接受：如果一个未匹配的数据包获得一个授权，则其接受该授权。

## 2.6仿真实验

### 2.6.1实验配置

本章提出的流水路由体系结构采用前缀树索引和随机存储技术。我们进行一系列的仿真实验来评估该体系结构的存储需求和吞吐量，仿真结果与已知的线状和环状体系结构进行比对。这三类路由体系结构均采用前缀树索引来组织路由表并采用不同的存储策略将索引映射到多个存储块构成的流水中：

* 线状流水：简记为LP（linear pipeline）。每棵前缀树的根结点位于首个流水阶的存储块中，子孙结点按层存储到后继流水阶的存储块中，每个存储块存储一层结点。数据包的查找请求依次访问流水阶，从前缀树的根结点下行至存储着最长匹配前缀的结点。
* 环状流水：简记为CP（circular pipeline）。该体系结构改造自[48]，每棵前缀树的根结点按当前各存储块的存储使用情况分配到某个流水阶的存储块中，其子孙结点按层存储到后继流水阶的存储块中，每个存储块存储一层结点。一个查找请求从目标前缀树的根结点所在流水阶出发依次通过各流水阶，从前缀树的根结点下行至存储着最长匹配前缀的结点。
* 随机流水：所提出的流水路由体系结构，简记为RP（random pipeline）。每个结点等概率随机分布到某个流水阶的存储块中。多个查找过程的并行执行被抽象为二部图匹配。

为表明RP的可扩展性，我们将4个前缀树路由算法BT、PT[2]、FST[1]和MPT[3]改造为快表+前缀树森林的形式，并将改造后的索引分别标记为BT-based, PT-based, FST-based和MPT-based索引。

### 2.6.2存储需求评估

我们分两个方面评估流水系统的存储开销：首先，我们测量在初始阶段存储索引的开销；其次，我们测量用于存储增量更新（包括删除和增加路由表前缀项）的开销。

不失一般性，假设流水系统中的每个流水阶的最大存储空间相等，则总的存储开销取决于单个流水阶的最大存储开销。由于前缀索引树中各层结点数量差异很大，LP的存储空间利用率非常低下。为解决这一问题，CP采用启发式算法来决定将前缀树根结点存储至哪一块存储块，但其仍然采用按层存储的方式将根结点的子孙结点依次存储到后继流水阶的存储块中。不同于CP，通过将每个结点等概率地分配到各存储块上来平衡存储需求。

我们采用BT-based组织IPv4和IPv6 的BGP表并将索引存储到LP、CP和RP中，BGP表取自<http://www.ris.ripe.net上的数据集rrc00-11>。测试的性能指标为单个存储块的空间开销，我们将测量值除以流水系统的总的存储开销以辅助分析。假设长、短前缀的分类阀值L=10，我们从表2-x观察到LP中的单个存储块最大存储开销是最小存储开销的上千倍，此处流水阶/存储块的数量M=C-L+1，其中C为IP地址的长度。我们还从表中观察到CP和RP在存储IPv4 BGP表的索引时达到了存储平衡。然而，CP在存储IPv6 BGP表的索引时出现了存储差异。例如：当存储数据集rrc00的IPv6 BGP表时，单个存储块的最大和最小存储开销分别为2.36E-02和1.19E-03。

前述的观察所得可解释如下：IPv6 BGP表的前缀分布不均衡，路由表索引中的各前缀树大小相差悬殊。因为LP和CP都采用“层到块”的结点映射策略，导致各流水阶的存储需求不平衡。另一方面，RP采用随机分布策略将前缀树的结点均匀散步到各流水阶中，这使得单个流水阶的存储需求与前缀的分布情况没有关联，从而在IPv4和IPv6的路由表上均取得理想的性能表现。

需要指出的是：LP的规模与索引中前缀树的最大高度成正比。这表明一个固定规模的线状流水系统只适用于特定的前缀树路由算法，这是因为前缀树的高度一般取决于所使用的前缀树路由算法。另一方面，CP和RP均不受此约束的限制，其能够将多种前缀树路由算法构造得到的索引映射到固定数量的存储块中。为了深入研究，我们评估16阶CP和RP在L=10的情况下的存储空间利用率，实验采用的索引构造算法为BT-based，PT-based和FST-based。如图2-xx所示，实验结果与表2-xx所得结论相一致。详细地说， CP和RP在存储IPv4 BGP表时均达到了存储需求的平衡，但前者在存储IPv6 BGP表时出现了性能的波动。

注意到在BT、PT和FST上的每个路由查找均始于根结点并止于一个叶子结点，在该查找路径上的每个结点均被访问一次。然而，MPT上的查找过程与此截然不同，一个查找请求可能在访问某个内部结点的子孙结点后再次访问该内部结点。结果是，只有RP支持MPT的查找过程。如图2-xx所示，当使用MPT-based构造路由表的索引时，RP中的各流水阶同样达到了均衡的存储需求。这表明RP的通用性必LP和CP更高。

为了评估路由表增量更新对存储块的空间利用率的影响，我们采用BT-based构造IPv4和IPv6 BGP表的索引并在其上执行一个月的路由表更新操作。从表xx可以看到，当执行完IPv4路由更新后，LP、CP和RP的存储开销并未发生显著变化。这是因为对索引的大部分修改均是由于短期内的链接失效所导致。当链接恢复后，上次更新后从索引的前缀树上删除的结点被重新添加回索引中。此外，在处理IPv6增量更新时，RP的存储开销逐渐趋于平衡。这表明当路由表逐渐增大时，每个存储块中的前缀结点个数也越发接近总数的1/M。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表xxx | | | | |
| 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xxx | | | | |
| 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 |

### 2.6.3吞吐量评估

每个查找请求必须首先访问根结点所在的存储块。假设单个存储块在一个时隙内仅允许依次访问操作，则LP最多允许一个查找请求进入流水系统。与之相反，CP和RP允许多个查找请求在同一时隙进入流水系统，只要这些查找请求所访问的前缀树的根结点位于不同的存储块内。在本节，我们将仿真评估16阶CP和RP处理IPv4和IPv6数据流时最大查找吞吐量。为此，我们引入多个参数以便描述：

* S：单个数据包在执行查找过程中的平均步数。每步访问一次存储块，每次访问读取一个前缀结点信息。一个查找请求包含一个或多个查找步。
* B：单个时隙内达到路由的数据包个数。数据包在每个时隙的开始到达路由，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，计算式为。
* Q：单个时隙缓冲单元中数据包的最大个数。注意到CP和RP采用某种缓冲策略来解决存储块访问冲突问题。其中，CP为每个存储块提供一个缓冲单元，而RP为所有存储块提供一个统一的缓冲单元。我们通过监视所有缓冲单元中的数据包总数来计算CP的Q值。

仿真实验的性能指标是Q与W之间的变换关系。下文中展示的实验数据均为重复两次相同实验得到的结果的均值，每次实验处理100万个数据包，每个数据包的目的地址从数据源rrc00的IPv4/IPv6表项中随机抽选。我们在图2-x和2-y中分别给出CP和RP处理IPv4和IPv6数据流时的吞吐量的表现。其中，L=10且M=16，所采用的前缀树索引构造算法包括BT-based、PT-based、MPT-based和FST-based。从图中，我们可以观察得到以下结论：

* 在初始阶段，各曲线随W的增长而平滑上扬；当W接近临界点时，系统出现过载，各曲线上扬趋势陡增。例如：从图5(a)可见，当B=2时，CP和RP处理IPv4数据流的临界阀值W’为0.85。
* 在给定W的情况下，当CP和RP处理IPv4和IPv6数据流时，Q随B的增大而不断增长。此外，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的表现相近；但当处理IPv6数据流时，后者的最大工作负载交之前者高出约10个百分点。例如：图6表明CP和RP的负载临界值W’分别为0.85和0.75左右，所使用的索引为BT-based。类似现象也出现在采用PT-based和FST-based的CP和RP之间。

在本节剩余部分，我们讨论增量更新对查找吞吐量的影响及其在所提出的路由体系结构中的执行方式。当在索引中插入或删除一个短前缀时，可以直接修改块表中的相应表项，时间复杂度为常量。例如当从图x的路由表中删除000\*时，我们只需修改快表中的首项：设置掩码数组的第3个比特为0并清空下一跳数组的第3各数据域。然而，当在索引中插入或删除一个长前缀时，路由系统需要首先找到该前缀在前缀索引森林中的位置，然而调用增添或删除操作来执行索引更新程序。文献[45]提出了一个片下预处理器，该预处理器在执行更新操作前预先计算该操作对索引结构的影响，然后将多个更新操作合并为一个可流水执行的“写泡”。在LP和CP上，这些“写泡”与每个查找请求一样从某个流水阶进入流水线并依次经过后续的流水阶来完成任务，这使得查找与更新操作可以一同并行完成。为了在RP上支持在线更新，将待处理的写操作划分为多个批处理作业，同一个批处理作业中的写操作的目标存储块互不相同，从而单个批处理作业可在一个系统时隙内完成。系统为每个批处理作业创建一个更新请求并将其与当前系统内的所有查找请求一同处理。为了保持前缀树在更新前后的状态一致性，每个更新请求较之查找请求由更高的调度优先级。因为更新请求的执行频率一般远小于查找请求的执行频率，我们可以合理地预期这些更新请求不会对系统的查找吞吐量产生显著的负面影响。

## 2.7硬件设计

我们在2.3节提出了采用先来先服务的调度算法，该算法曾被用于解决数据包在交换阶段的端口冲突问题。因此，我们可以参考虚拟输出缓存交换机中的仲裁单元的硬件实现来设计所提出的路由体系结构中的调度单元。根据文献[52][53]所载，我们在图xx中给出一个多存储块路由体系结构的参考硬件设计，该设计采用一个先来先服务的调度单元来控制对各存储块的访问操作。调度单元包括M个授权仲裁单元、H个接受仲裁单元以及一个用于缓存和存储仲裁结果的决策寄存器。其中，每个授权仲裁单元对应一个存储块而每个接受仲裁单元对应一个位于缓冲单元中的数据包。如图所示，缓冲单元E的状态可以表示为H个向量{e1,e2,…,eH}，每个向量包含M个比特且各比特的值表示该向量所对应的数据包是否要求访问指定的存储块。例如：给定M=8且ei=[01000000]，则缓冲单元中的第i个数据包请求访问存储快m2。令{G1,G2,…,GM}和{A1,A2,…,AH}分别表示授权和接受仲裁单元，则系统执行以下步骤来调度数据包对存储块的访问请求：

* 请求：若ei≠0，则数据包i尚未匹配。任意ei[j] ≠0被发送至授权仲裁单元Gj。
* 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，其根据先来先服务策略从中选取一个。若来自ei的请求被选中，则将访问授权发送给接受仲裁单元Ai。
* 接受：每个尚未匹配的接受仲裁单元可能接受到最多一个授权比特。其接受授权并将结果存入决策寄存器中。

该调度单元的应届复杂度为O(H×M)。从本章的仿真实验结果可知，给定M=8，系统在H=20时达到最大查找吞吐量。此时，H和M的取值适中。

## 2.8本章小结

本章提出了一种基于前缀树索引的流水路由体系结构。该体系结构具有通用性，可部署于其他具体的字符串处理应用上下文中。在本章末尾，我们进一步总结所述体系结构RP和时下流行的LP、CP之间的优劣：

* + RP是一个通用、可扩展的IP查找解决方案，该方案适可流水并行化任意采用指针型前缀树作为索引数据结构的路由查找算法。不同于LP和CP，在RP中的调度问题被建模为二部图匹配问题，该问题与交换机体系结构中的调度问题异曲同工，可相互借鉴。
  + RP采用一个缓冲单元缓存待处理数据包，每个数据包的路由延迟并非常量。给定M个存储块，RP表现为一个配备有M个服务器的排队系统。通过采用包交换过程中的流量工程技术，缓冲单元的数据包丢失概率可以被控制在一个较低的水平。
  + RP采用随机分布策略存储前缀树结点，这导致不同查找请求通过流水线的模式大概率不同。与之相对，LP和CP采用按层顺存策略存储前缀树结点，这使得每个查找请求通过流水线的模式相同。这导致RP的性能较LP和CP更易产生波动。
  + 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，每个存储快的电量开销预期最优。

目前，我们在本章的工作基础上着手开发一个实用的路由系统。

# 第三章 后缀数组构造算法

## 3.1基础知识

查找一个模式在给定字符串中的所有出现是字符串处理程序的常见操作。已知给定字符串的后缀树，可在O(m)时间内确定任意模式在字符串中是否存在出现。后缀数组是后缀树的一个精简替代，一个字符串的后缀数组记录该字符串所有后缀的字符序排序。结合最长公共前缀数组，可通过后缀数组模拟后缀树的遍历过程。现有多个工作提出可在线性时空内运行的内存后缀数组构造算法。其中，采用归纳排序思想的SA-IS算法常数因子最小、实际性能最优。随着待处理数据规模的不断增长，可修改内存算法使其在外存上运行以便处理大容量输入。现有采用归纳排序思想的外存算法eSAIS，该算法较之其他算法具有更好的时间复杂度，但其磁盘性能处于劣势。例如：eSAIS在构造40-bit编码的后缀数组时的平均磁盘峰值为24n，而pSAscan的磁盘峰值为8n。有研究工作表明，可通过工程方法来优化该算法的实现，使其磁盘峰值降为8n，接近理论最优值。在本节的剩余部分，我们使用如下术语描述采用归纳排序思想的后缀数组构造算法的执行框架：

* 字符、后缀和子串。给定长度为n的输入字符串x，记suf(i)为以x[i]为首的后缀，记sub(i,j)为由区间x[i,j]中的字符构成的子串。
* 字符、后缀和子串的分类。给定长度为n的输入字符串x，其中任意字符被划分为L和S两类。具体而言，当满足以下任一条件时，x[i]为L类型字符，否则x[i]为S类型字符：
  + i=n-1；
  + x[i]>x[i+1]；
  + x[i]=x[i+1]且x[i+1]为L类型字符。

特别地，若x[i]和x[i+1]分别为S和L类型字符，则x[i]也为S\*类型字符。进一步地，x中的后缀或子串与其首字符的类型相同。

* 后缀桶。记x的后缀数组为sa，则sa中的所有后缀被自然划分为多个后缀桶，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。每个桶可被进一步切分为左、右两个子桶，左子桶仅包含L类型后缀而右子桶仅包含S类型后缀。我们分用sa\_bkt(c)、sa\_bktL(c)和sa\_bktS(c)表示首字符为c的所有后缀构成的后缀桶及其左、右子桶。
* S\*类型后缀的后缀数组。令x中的S\*类型后缀个数为n1，则sa\*[0, n1]按字符序存储了所有S\*类型后缀。
* 类型数组。数组t记录了x中字符的类型信息。其中，t[i]=0或1分别表示x[i]为L或S类型字符。
* 收缩字符串。可将x视为多个S\*类型子串的拼接，每个S\*类型子串仅首尾为S\*类型字符且相邻两个S\*类型子串重叠一个S\*类型字符。排序x中的所有S\*类型子串并对其命名，记命名后的字符串为x1并称x1为x的收缩字符串。
* 前继和后继字符。记x[i]的前继和后继字符为x[i-1]和x[i+1]。类似地，记

|  |
| --- |
|  |
| 图3 |

如图3-x所示，采用归纳排序思想设计的后缀数组构造算法主要包括两个阶段：收缩和解推导阶段。在收缩阶段，根据归纳排序思想排序和命名所有S\*类型子串来计算x的收缩字符串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，从sa1计算S\*后缀的后缀数组并根据归纳排序思想排序所有S\*类型后缀。在收缩和解推导阶段，关键操作是从已排序的子串/后缀推导待排序子串/后缀之间的字符序，原理为：给定后缀suf(i)和suf(j)，可通过比较它们的首字符和后继后缀的大小来确定两者的字符序。

（此处举个例子）

## 3.2算法原理

### 3.2.1数据结构

eSAIS算法是SA-IS算法的一个外存工程实现，其继承了SA-IS的线性时间复杂度，但磁盘峰值较大。我们将SA-IS移植到外存，提出一种新型外存后缀数组构造算法DSA-IS，该算法的磁盘峰值仅为eSAIS的三分之二。在移植SA-IS的过程中，我们遇到如下困难：SA-IS的运行过程中需要频繁地随机访问输入字符串x和后缀数组sa。在内存模型上，随机访问内存的开销小、速度快。然而，在外存模型上，随机访问外存的开销大、速度慢。为此，需要设计新的数据结构和存取模式来批量处理外存读写请求。我们回顾归纳排序思想可知，待排序子串/后缀的字典序是由其首字符和后继字符的字典序决定的。换言之，当推导某个已排序子串/后缀在所有子串/后缀中的排序位置时，我们需要知道其自身在所有子串/后缀中的排序位置、其前继子串/后缀的首字符和该子串包含的字符。如果能够预先将这些信息按需存储到外存中，那么我们就能在需要时通过顺序I/O操作从外存中将信息批量地读入内存。为此，我们引入一个新的数据结构DSAItem，其包括以下数据域：

* p：suf(p)或sub(p,p2)的位置索引，p2为x[p]右侧最近S\*字符的位置。
* c：suf(p)或sub(p,p2)的首字符，即x[p]。
* r：suf(p)或sub(p,p2)在所有后缀或子串中的字典序。
* t：x[p]的前继字符的类型，即t[p-1]。

DSA-IS算法的输出为由n个DSAItem构成的数组DSA，该算法主要由收缩和解推导两个阶段组成，后续两节给出各阶段算法设计的详细描述。

### 3.2.1 收缩阶段

收缩阶段排序字符串中的所有S\*子串，每个S\*子串的首尾为S\*字符且子串内部无其他S\*字符，任意两个位置连续的S\*子串之间重叠一个S\*字符。

收缩阶段的第一步是将x划分为多个分块，每个分块包含一个或多个相连的S\*子串。若内存大小为M，则分块最大容纳m个字符，其中m=O(M)。按此分块方法，分块总数k不超过。具体而言，算法从右向左读入x中的字符，依次获取各S\*子串并将一个或连续的多个S\*子串组织为如下两类分块中的一类：

* 单子串块：块内仅包含一个S\*子串。该子串与左侧或右侧S\*子串的总长度超过m，根据约束条件只能单独成块。需要注意的是，单个S\*子串的长度可能超过m。
* 多子串块：块内包含二个以上位置连续的S\*子串。相邻子串重叠一个S\*字符，且块内字符总数不查过m。

对于多子串块，算法重用SA-IS的收缩阶段来排序块内S\*子串并记录排序过程所需的子串首字符，这些字符按其访问顺序依次存储到外存中。同时，算法还在外存中记录多子串块中排序后的各子串包含的字符，记录顺序与子串的排序顺序一致。对于单子串块，其中的S\*子串已有序。

收缩阶段的第二步是将各分块的排序结果合并。该步骤可细分以下3个子步骤：

1. 创建一个外存小顶堆，堆中数据类型为DSAItem，排序关键字为<c,t,r,p>。从右向左读入x中的字符，依次获取各S\*字符并将其DSAItem插入到小顶堆中。此时，所有S\*字符在小顶堆中有序。
2. 依次取出小顶堆中的堆顶元素，推导其前继子串，前继子串的首字符可通过顺序I/O从外存读入内存。若前继子串为L类型，则将其DSAItem插入到小顶堆中。在此过程中，将从小顶堆中取出的所有L类型子串的DSAItem插入到一个外存大顶堆。当小顶堆为空时，所有L类型子串均在大顶堆中有序，排序关键字为<c,t,r,p>。
3. 依次取出大顶堆中的堆顶元素，推导其前继子串，前继子串的首字符可通过顺序I/O从外存读入内存。若前继子串为S类型，则将其DSAItem插入到大顶堆中。在此过程中，将从大顶堆中取出的所有S\*类型子串的DSAItem依次存储到外存。当小顶堆为空时，所有S\*类型子串均有序。

收缩阶段的第三步是对排序后的S\*类型子串命名。算法依次比较相邻两个S\*类型子串是否相等，若相等则后者与前者同名，否则，后者的名字比前者名字大1。需要注意的是，算法需要获得子串包含的字符来执行比较操作。由于各分块中的子串已在第一步排序完毕且排序后的子串所包含的字符也按序存储到外存，算法可根据子串首字符位置判定其所在块的块号并通过顺序I/O从外存中获取这些子串所包含的字符。最终，算法得到了x的收缩子串x1。若x1包含同名字符，则递归执行收缩结点，输入为x1；否则，从x1直接计算其后缀数组sa1并执行解推导阶段。

### 3.2.2 解推导阶段

解推导阶段排序字符串x中的所有S\*类型后缀。

解推导阶段的第一步将x划分为多个分块，分块的划分方法与收缩阶段的第一步相同。对于多子串块，其中各S\*类型子串的排序可从sa1获悉，所以以这些S\*类型子串为前缀的S\*类型后缀的排序也可推知。具体而言，算法首先从sa1中取出各多子串块中S\*类型子串所对应的名字，然后执行SA-IS的解推导阶段来推导各多子串块中后缀的字典序。对于单桶块，其中所有后缀已经有序。

类似于收缩阶段，解推导阶段的第二步也是将各分块的排序结果合并。该步骤可细分为以下3个步骤：

1. 创建一个外存小顶堆，堆中数据类型为DSAItem，排序关键字为<c,t,r,p>。从sa1获得各S\*类型后缀的字典序并将各S\*类型后缀的DSAItem插入到最小堆中。此时，所有S\*类型后缀在小顶堆中有序。
2. 依次取出小顶堆中的堆顶元素，推到其前继后缀，前继后缀的首字符通过顺序I/O从外存读入内存。若前继后缀为L类型，则将其DSAItem插入到小顶堆中。在此过程中，将从小顶堆中取出的所有L类型后缀的DSAItem插入到一个外存大顶堆中。当小顶堆为空时，所有L类型后缀均在大顶堆中有序，排序关键字为<c,t,r,p>。
3. 依次取出大顶堆中的堆顶元素，推导其前继后缀，前继后缀的首字符可通过顺序I/O从外存读入内存。若前继后缀为S类型，则将其DSAItem插入到大顶堆中。在此过程中，将从大顶堆中取出的所有L和S类型后缀的DSAItem依次存储到外存。当小顶堆为空时，所有后缀均有序，从而得到当前x的sa。

## 3.3算法优化

本章实验数据表明，DSA-IS的速度和I/O与eSAIS相当，而前者的磁盘峰值仅为后者的三分之二。尽管如此，DSA-IS在处理中小规模数据集时的表现较之SAscan和pSAscan等非线性时间算法相距较远。本节从算法设计和代码工程等两方面优化DSA-IS，改进算法的最终实现可以在7.5n的磁盘空间内计算得到x的后缀数组。

### 3.3.1 设计优化

本节的第一个优化是重新设计DSA-IS算法的收缩阶段，使得子串排序和命名过程能够同步执行。在收缩阶段，DSA-IS使用外存堆排序S\*类型子串的DSAItem，排序关键字为<c,t,r,p>。其中，r表示DSAItem对应子串的后继在所有子串中的字典序。换言之，可将r视为后继子串的名字。同步排序和命名S\*类型子串的依据在于：两个子串相等，当且仅当子串的首字符和其后继子串的名字均相同。因此，在收缩阶段的步骤二，算法可比较连续从大顶堆中取出的DSAItem，若它们的c和r数据域的值均相等，则对应的两个S\*类型子串也相等并被赋予相同的名字。值得一提的是，该方法无需通过比较字面值来确定两个子串是否相等，因此也就无需在步骤一预先存储各块中排序后的子串的字符。

在真实数据集中，大部分S\*类型子串的长度很短（只有几个字符），而DSAItem结构臃肿，在外存中对其排序是算法的瓶颈之一。为进一步降低磁盘空间和I/O量，本节的第二个优化方法将所有S\*类型子串划分为长、短两类，并按如下方式分类处理：

* 长子串处理方式：对长度大于D的S\*类型子串，重用DSA-IS的方法对其进行排序和命名。可使用第一个优化方法合并排序和命名过程。
* 短子串处理方式：对长度D不超过D的S\*类型子串，在DSA-IS的步骤一中以分块为单位在内存中排序，将排序后的子串存储到外存。

此时，x中的所有长子串已全局有序，而x中的短子串按分块部分有序。算法使用一个内存最小堆合并长、短子串。对堆中任意两个子串，最小堆按如下比较规则确定两者的字典序：

* 若均为两个短子串，则按字面值比较两个子串。
* 若有一个为长子串，则按字面值比较两个子串的前D个字符。若前D个字符相等，则短子串为长子串的前缀且短子串比长子串小。

需要注意的是：在任意时刻，最小堆中最多包含k+1个子串，其中k为分块数。因此，其最多占用的内存空间。

DSA-IS使用外存堆排序子串的DSAItem，从而模拟SA-IS排序子串的过程。本节第三个优化方法简化DSAItem的结构，从而进一步降低磁盘峰值。根据归纳排序原理，两个子串的字典序由它们的首字符和后继子串的字典序确定。在排序子串的过程中，名字相同（字面值相等）的子串聚集在一起，算法扫描已排序子串并推导它们前继的字典序。可将推导出的同名子串写到外存xxxx。

### 3.3.2 工程优化

改进算法实现可大幅提高算法性能，本节提出多个可行的工程优化方法，能够提升运行速度或降低磁盘空间需求。可结合实际运行条件，优化算法的程序实现。

在真实数据集上，DSA-IS的磁盘空间较大，这主要是因为算法程序在运行过程中产生大量临时数据，而这些数据在使用完毕后未能及时释放。可通过文件块链实现对临时数据的细粒度分配和释放。具体而言，将临时数据按使用顺序划分为多个子集，每个子集存储在单独的一个文件中。当某个文件中的临时数据被废弃时，可删除其所在文件来释放磁盘空间。通过控制单个文件的大小，可以将磁盘空间需求维持在一个较低的水平。

在收缩和解推导阶段，DSA-IS首先按分块排序x中的子串和后缀。各分块的处理过程数据无关，可采用空间换时间的策略，并行执行多个分块中的排序任务。

DSA-IS使用外存堆排序固定大小的元组，排序键为整数。可使用GPU基数排序算法提高外存排序的吞吐量，加快运行速度。

## 3.4实验分析

## 3.5本章总结

# 第四章 有限阶最长公共前缀数组构造算法

## 4.1基础知识

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。根据归纳排序思想，可在线性时、空内同步构造后缀数组和最长公共前缀数组。算法适用于单机内存和外存模型，但难以在分布式计算模型上实施。在实际应用场景中，待匹配模式一般仅包含几十或几百个字符，无需提供全阶后缀数组和最长公共前缀数组作为查找索引。为此，本章提出一种有限阶最长公共前缀数组的构造算法，通用性好，可在经典分布式计算模型上实施。后文中使用以下名称术语和参数变量进行描述：

* 最长公共前缀数组。记lcp为输入字符串x和后缀sa的最长公共前缀数组，lcp[i]记录和之间的最长公共前缀的长度。
* 有限阶最长公共前缀数组。若两个前缀的最长公共前缀*l*大于等于K，则其K阶最长公共前缀等于K；否则，其K阶最长公共前缀等于*l*。记lcpK为输入字符串x和后缀sa的K阶最长公共前缀数组，lcpk[i]记录和之间的K阶最长公共前缀的长度。
* : 公式的缩写。

## 4.2内存算法设计

已知字符串x和其后缀数组sa，可通过字面值比较来计算两个字符串的K阶最长公共前缀，时间复杂度为。可采用指纹函数加速计算过程，计算结果可能存在谬误。此概率性算法在内存计算模型上的框架如图xx所示。

|  |
| --- |
|  |
| 图xxx |

该算法的核心思想是计算一个后缀对（包含两个后缀）的最左K/2个字符的指纹值并比较指纹值来确定对应后缀的前K/2个字符是否相等。若相等，则继续比较后续K/4个字符的指纹值；否则，比较最左K/4各字符的指纹值。递归执行该过程次，即可得到该后缀对的K阶最长公共前缀。可使用迭代指纹函数在线性时间内计算比较复数个后缀对所需的所有指纹值，具体描述如下：

已知x和sa，定义集合，集合中每个元组是一个位置索引对，标识一对后缀的首字符在x中的位置。对P中各元组，寻找,使得且，则即为起始位置为的后缀对的K阶前缀字符串长度。如图xx所示，算法初始将赋值为并继而执行轮循环。在第k轮循环，算法判定中每个元组是否满足来生成：若,将插入；否则，将插入。这里的关键操作是比较两个子串的字面值。为加快比较速度，可首先计算子串的指纹，然后通过比较指纹来判定子串是否相等。算法采用Karp-Rabin指纹技术，该技术首先使用公式xxx迭代计算x中所有前缀的指纹，然后使用公式xxx和xxx计算x中任意子串的指纹。需要注意的是：两个字面值相同的子串的指纹值必然相同，反之未必。可以令指纹函数的参数L和取大整数来降低谬误的出现概率。采用指纹技术处理并生成的过程包括以下三步：

* 向右遍历x中字符，根据公式xxx迭代计算的指纹值，。若满足，则将指纹值插入一个内存哈希表。
* 对每个，根据公式xxx-xxx计算的指纹值，其中和从哈希表中查询得到。
* *对于每个元组，比较和。*若两者的指纹值相同，将插入; 否则，将插入。

循环执行完毕后，，算法可在时间内计算得到P中每对后缀的K阶最长公共前缀。该算法的时间和空间复杂度均为。

## 4.3 外存算法设计

可将4.2节的内存算法改造为外存算法，两者的差别在于循环体的执行。内存算法采用一个哈希表来随机存取x中前缀的指纹，空间复杂度为。然而，当x的规模变大时，无法在内存中维持x和哈希表。本节提出的外存算法使用外存整数堆实现顺序I/O读写操作，避免因随机I/O产生性能瓶颈。算法在第k次递归的过程中引入两个数组CPk和PPk，数组元素为一个用三元组<idx,pos,fp>表示的LCPItem，各数据域分别存储数组下标、位置值和指纹值。

在第k次递归的开始阶段，外存算法按如下方式处理中的每对元组，从而将映射为CPk和PPk：

* 创建两个fp值为空的LCPItem，pos值分别设为和。将前者添加至CPk末尾，将后者添加至PPk末尾。
* 创建两个fp值为空的LCPItem，pos值分别设为和。将前者添加至CPk末尾，将后者添加至PPk末尾。

其后，使用外存堆排序CPk和PPk的元素，排序关键字为<pos>，排序结果为数组ICPk和IPPk。从左向右遍历x中的字符并使用Karp-Rabin指纹函数迭代计算x中前缀的指纹。在此过程中，若当前访问的x中元素的位置与CPk或PPk中元素的pos相等，则将对应指纹更新元素的fp值。当遍历结束后，使用外存堆排序ICPk和IPPk中的元素，排序关键字为<idx>，排序结果为更新了fp值得CPk和PPk。从左向右遍历CPk和PPk中的元素并按如下三步计算：

1. 从CPk中取出最左未访问的两个LCPItem，各LCPItem对应x中以x[pos]结尾的前缀，前缀的指纹存储在fp中。两个前缀的末尾位置限定了一个子串的开始和结束位置，使用Karp-Rabin指纹函数计算该子串的指纹。
2. 从PPk中取出最左未访问的两个LCPItem，重用步骤1中的方法计算两个LCPItem所限定的子串的指纹。
3. 比较步骤1和2中计算得到的子串的指纹。若指纹相等，认为两个子串的前个字符相等；否则，认为两个子串的前个字符不相等。据此可由计算得到。

外存算法中，每轮递归的时空复杂度与外存整数排序复杂度相当，劣于采用归纳排序思想设计的最长公共前缀构造算法。然而，该算法的优势在于扩展性好，下一节将描述如何将外存算法改造为可在对称分布式模型上运行的版本。

## 4.4 分布式算法设计

|  |
| --- |
|  |
| 图4-xx 由d个计算节点构成的对称分布系统 |

如图xx所示分布式系统由计算节点互联而成，各节点配备有容量为E的存储设备。将输入字符串x和后缀数组sa划分为，其中。令, and ，我们引入以下符号和名词术语：

* 字符串分块：且位于节点。
* 后缀数组分块：且位于节点。
* Pi,k：。
* CP和PP数组分块和： 且。
* ICP和IPP数组分块和：且 。
* 发送缓冲区 and ：在的内存中为节点维持两个发送缓冲区 and 。前者缓存将要发送至的中的元素，后者缓存将要发送至的中的元素。
* 接收缓冲区and ：在的内存中为节点维持两个接收缓冲区and 。前者缓存接收自的中的元素，后者缓存接收自的中的元素。

分布式算法使用缓冲区模拟外存算法的循环。在第k次递归，节点执行顺次执行以下步骤从Pk计算得到Pk+1：

1. 首先遍历xi，创建和。然后，根据数组中各LCPItem的pos值将其发送到其他结点。
2. 接收来自其他结点的LCPItem，根据pos值排序，得到和。然后，根据数组中各LCPItem的idx值将其发还到其他结点。
3. 接收来自其他结点的LCPItem，根据idx值排序，得到和。然后，使用Karp-Rabin指纹函数从Pi,k计算得到Pi,k+1。

在分布式算法的执行过程中，每个结点的内存被划分为两部分，一部分用于创建通信缓冲区而另一部分用于创建I/O缓冲区。在算法的递归循环执行完毕后，在本地计算和的最长公共前缀数组并将计算结果就地存储。

## 4.5算法优化

### 4.5.1设计优化

本章算法的时间复杂度受循环的递归次数影响。在循环的第k次递归阶段，算法从Pk计算Pk+1。可修改算法流程，跳过Pk+1，从Pk直接计算Pk+2。需重新定义CPk和PPk如下，其中i和j分别为sa中相邻两个后缀的起始位置：

* 创建四个fp值为空的LCPItem，pos值分别为。将前二者添加至CPk末尾，将后二者添加至PPk末尾。
* 创建四个fp值为空的LCPItem，pos值分别为。将前二者添加至CPk末尾，将后二者添加至PPk末尾。

其后，遵循原算法更新CPk和PPk中各元素的fp值并按如下步骤从Pk计算Pk+1：

1. 从CPk中取出最左未处理的四个LCPItem，使用Karp-Rabin指纹函数计算、、的指纹。
2. 从PPk中取出最左未处理的四个LCPItem，使用Karp-Rabin指纹函数计算、、的指纹。
3. 比较与的指纹。若不相等则执行步骤4；否则，执行步骤5。
4. 比较和的指纹。若不相等，则；否则，。结束。
5. 比较和的指纹。若不相等，则；否则，。结束。

此方法的思路是空间换时间，可减少循环的递归次数。

### 4.5.2工程优化

可采用后缀数组构造算法中引入的文件块链和GPU基数排序等技术，改善算法执行过程中的磁盘峰值和运行速度。

## 4.6实验分析

## 4.6本章小结

本章提出了一个可扩展有限阶最长公共前缀数组构造算法，适用于内存、外存和分布式计算模型上。该算法易于实现，其外存版本的C++实现不到600行。给定阶值K，在由d个计算节点互联而成的对称型分布式系统中，算法在单节点上的时、空复杂度分别为和。目前，尚未有文献记载可在分布式环境下运行的最长公共前缀数组构造算法，我们的工作填补了这一研究空白。目前，该算法在各计算模型上的实现尚有改进的空间，后续工作将致力于提高算法程序的运行速度和I/O性能。

# 第五章 后缀数组和最长公共前缀数组事后验证算法

## 5.1 基础知识

随着待处理输入规模的不断增长，后缀索引构造算法的设计复杂度也不断提高。现有多个开源工具包（例如：SA-IS、eSAIS和bwt-disk等）提供后缀索引构造算法程序，但其并不保证程序实现的正确性。因此，需要提供快速、有效的验证算法，检测程序的运行结果。本章提出两个后缀数组事后验证算法。算法一从定义出发，同步验证后缀数组和最长公共前缀数组。算法二首先调用算法一验证所有S\*后缀的后缀数组，然后采用归纳排序思想验证全局后缀数组。

## 5.2 算法1

### 5.2.1 算法原理

根据后缀数组和最长公共前缀数组的定义，引理1给出一组验证后缀数组和最长公共前缀数组正确性的充要条件：

引理1：已知长度为n的字符串x，和分别为x的后缀数组和最长公共前缀数组，当且仅当以下条件成立，其中：

1. sa是的一个全排列。
2. 。
3. 。

证明1：引理1的充要性可从后缀数组和最长公共前缀数组的定义直接得到。具体而言，条件1）保证了x中的每个后缀都出现在sa中，条件2）和3）保证了sa中任意两个相邻后缀和其最长公共前缀均正确。

若按字面值比较两个子串来确定他们的最长公共前缀，最坏时间复杂度为。可用一个完美哈希函数将待比较子串转化为整数。当且仅当两个子串的哈希值相同，它们的字面值才相等。从而，可将字面值比较转化为耗时更少的整数比较。考虑到寻找满足条件完美哈希函数较为困难，我们选择使用Karp-Rabin指纹技术，将子串转化为整型的指纹值。根据第四章引入的指纹函数，可在线性时间内计算x中某个子串的指纹：从右向左遍历x中的字符，迭代计算各后缀的指纹并记录以目标子串的首尾位置作为起始位置的两个后缀的指纹，根据记录的指纹计算得到目标子串的指纹。需要再次重申，两个字面值不等的子串可能具有相同的指纹，可将指纹函数中的参数设为大整数来大幅降低出错概率。

推论1：已知长度为n的字符串x，当以下条件成立时，和大概率分别为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. 。
3. 。

图5-x给出了一个验证实例，根据推论1检测输入的后缀和最长公共前缀数组。如图所示，令且，第4-8行根据公式xxx迭代计算x中各前缀的指纹，第10-20行根据前缀的指纹值计算目标子串的指纹值。具体而言，考虑sa中最左相邻后缀和，两者的最长公共前缀指定的子串为和。根据计算公式，为和之差，而为和之差。从而，我们在第10-14行得到目标子串的指纹并比较得知两者相等。

### 5.2.2算法设计

在随机访问模型上设计验证算法，算法初始化两个数组和，前者存储x中所有前缀的指纹，后者检测sa是否为的一个全排列。算法按如下步骤检测sa和lcp是否满足推论1中的条件：

1. 依次访问，迭代计算以当前访问的字符结尾的前缀的指纹并将其存储于。
2. 依次访问，按如下子步骤检测和的正确性：
   1. 令，根据和计算得到，赋值。
   2. 根据和计算。
   3. 判定且是否成立。
   4. 赋值。
3. 判定对所有是否成立。

上述过程在内存模型上的时空复杂度为，若将其移植到外存模型上，难点在于步骤2需频繁地随机访问fp和mk数组。可预先将数组中的元素按访问顺序排序并存于外存，然后在需要时通过顺序I/O读入内存。如图5-xxx所示，为达此目的，算法1在第2-5行遍历sa和lcp来创建三个数组、和并按最左域值排序各数组中的元素。当排序完毕，第6-21行迭代计算x中所有前缀的指纹并按如下方法将计算结果更新至、和中的元素：当计算得到时，按排序后的顺序遍历、和中的元素，取出最左域值为i的元素并在其中缓存指纹，将更新后的元素分别存储到、和中。因为中的所有元素的最左域值构成一个sa的拷贝，该算法可在按排序后的顺序遍历的元素时验证推论1的条件1）是否得到满足。最终，第23-31行将、和中的元素按其在、和的位置排回原序。

|  |
| --- |
|  |
| 图5-xx 基于推论1的外存验证算法 |

我们指出，第25和27行在计算指纹时需要快速获取。可在内存中维持一个哈希表来存储，空间开销较大。由于任意两个后缀的最长公共前缀的长度不超过n，可将n分解为以2为底的多项式的积。换言之，对任意，我们有成立，其中。该方法的空间复杂度为，用于存储预先计算的。

算法1执行多次数组遍历和整数排序操作。已知内存容量为M、外存容量为D且外存块大小为B（M、D、B均以字节为单位），单次数组遍历的时间和I/O复杂度分别为和，单次数组排序的时间和I/O复杂度分别为和。算法1的磁盘峰值出现在第5和22行的排序过程。假设输入的字符串和后缀数组/最长公共前缀数组的单个元素被表示为和个字节的整型，而单个指纹值被表示为个字节的整型，则排序所需的磁盘空间为，而排序和的空磁盘空间为。为节省空间，可分别排序、和中的元素。

## 5.3 算法2

### 5.3.1 算法原理

本节提出一种采用归纳排序思想设计的事后验证算法，该算法所需磁盘空间远小于算法1。由第xx章可知，采用归纳排序思想设计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等两个阶段，前者计算所有S\*后缀的后缀数组sa\*和最长公共前缀数组lcp\*，后者从sa\*和lcp\*计算全局的后缀数组sa和最长公共前缀数组lcp。换言之，若sa\*和lcp\*均已知，则可从它们归纳排序sa和lcp。这启发我们根据引理2验证sa和lcp的正确性。

引理2. 给定字符串x、后缀数组sa和最长公共前缀数组lcp，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，其中和是根据归纳排序原理从sa\*和lcp\*计算得到。

可使用算法1来概率性验证sa\*和lcp\*的正确性，可以得到推论2：

推论2. 给定字符串x、后缀数组sa和最长公共前缀数组lcp，当满足以下条件时，sa和lcp大概率是正确的，其中，，且：

1. 是S\*类型且。
2. fp(。
3. 。
4. 且。其中，和是根据归纳排序原理从sa\*和lcp\*计算得到。

### 5.3.2 算法设计

如图xx所示，我们设计算法2来检测推论2中的验证条件。该算法的第一步是计算sa\*和lcp\*并调用算法1来概率性验证两者的正确性。首先，第2-3行将sa中的后缀按其在x中的位置排序。其后，第4-13行遍历x来从排序后的后缀中选出S\*类型的后缀。再后，第14-27行将S\*类型的后缀按其在sa中的位置，排序结果构成sa\*。假定和为sa\*中的一对相邻后缀，则其最长公共前缀为{}中的最小值。据此，第14-27行同时基三sa\*中各对相邻后缀的lcp值，计算结果构成lcp\*。需要注意的是：第7-12行按后缀在x中的位置排序遍历所有后缀，此过程同时判定推论2中的条件1）是否成立。此时，第28-30行调用算法1判定条件2）和3）是否成立，若验证为真则第31-36行根据归纳排序思想从sa\*和lcp\*计算得到和并判定条件4）是否成立。

|  |
| --- |
|  |
| 图5-xx 基于推论2的外存验证算法 |

## 5.4 算法优化

### 5.4.1 设计优化

算法2首先计算和，然后将其与输入的sa和lcp比较来验证后两者的正确性。假设x的字符取自常数字符集，可在计算和的过程中同步比较其与sa和lcp中的元素，从而无需存储和。思路是：当根据归纳排序思想推导出和中的元素，不将元素插入和中而是将元素与sa和lcp中对应位置的元素直接比较。由于归纳排序思想的核心是从已排序/已计算的后缀/最长公共前缀推导出未排序/未计算的前继后缀/最长公共前缀，若/的某个元素与其在sa/lcp中的对应元素相同，则在推导其前继后缀/最长公共前缀时，不从/取出该元素，而是从sa/lcp中取出该元素的对应元素来推导其前继。

### 5.4.2 工程优化

算法1的性能瓶颈在于外存排序、和，三者的排序过程没有数据依赖，可并行执行。文件块链和GPU基数排序等技术通用于外存程序，可优化算法1和算法2的实现。

## 5.5 实验分析

## 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验证算法，可大概率检测常量或整型字符串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均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1通用性好，可验证有限阶/无限阶的完整/稀疏后缀数组和最长公共前缀数组。另一方面，算法2仅可验证无限阶完整后缀数组和最长公共前缀数组。算法1使用灵活，适用于各类应用上下文。一个用例为：一个后缀数组或最长公共前缀数组可能由于软、硬件问题出现谬误。若未能提供备份，需要重构整个数组，开销较大。可使用算法1定位数组中的坏区，并调用稀疏后缀数组构造算法来恢复坏区所在的数组部分。另一个用例为：可使用算法1检测稀疏后缀数组，因为稀疏后缀数组的通常远小于完整后缀数组，算法1的时、空开销可大幅缩减。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2的运行速度慢于算法1，但前者所需的磁盘空间远小于后者。算法1和算法2的设计与实现仍有较大的改进空间。例如：算法1的多个外存排序过程可并行执行，而算法2在处理常量字符串的后缀数组和最长公共前缀数组时可使用更小的磁盘空间。

# 第六章 后缀数组事中验证方法

## 6.1 基础知识

目前，仅有一篇文献记载后缀数组验证方法，为事后验证方法，该方法检测引理6.1中的条件是否成立：

引理6.1 已知字符串x和数组sa，当且仅当以下条件成立时，sa为x的后缀数组：

1. sa是的一个全排列。
2. 对任意且，。其中，和表示和在x的所有后缀中的字典序。

证明：若条件1）为真，则sa包含x中的所有后缀。若条件2）为真，则sa中任意两个相邻后缀的字典序由首字符和其后继的字典序确定。根据sa的定义和归纳排序思想，x中的所有后缀在sa中的排序符合字典序。

基于该方法设计的外存后缀数组验证算法执行两轮外存排序，排序关键字为整型，排序规模为。

近期研究工作表明：采用归纳排序思想设计的后缀数组构造算法可在8n的磁盘空间上构造长度不超过的字符串的后缀数组，这体现了归纳排序思想的威力。如第xx章所述，采用归纳排序思想设计的后缀数组构造算法首先递归执行收缩阶段计算各递归层的字符串的收缩子串直到收缩子串无重复字符，然后递归执行解推导阶段计算各递归层的字符串的后缀数组，输出为输入字符串的后缀数组。方法1的思路是在最高最高递归层执行解推导阶段的同时验证所推导的后缀数组的正确性。为便于理解，我们首先给出解推导阶段的执行过程，其包括以下步骤：

1. 从中获取x的所有S\*类型后缀的字典序，按此排序S\*类型后缀的首字符位置得到sa\*。
2. 清空所有和桶。从右向左遍历sa\*中的后缀。将当前访问的后位于，则将插入到桶的最右空位。
3. 从左向右遍历sa中的后缀。若当前访问的后缀位于且，则将插入到桶的最左空位。
4. 清空所有桶。从右向左遍历sa中的后缀。若当前访问的后缀位于且，则将插入到桶的最右空位。

图6-xx给出了上述步骤的一个运行实例。已知x和，第6行将每个S\*类型后缀插入到对应的后缀桶中。例如：依次被放置于桶的最右空位，插入顺序与这些后缀在中的位置顺序一致。其后，第8-9行首先寻找各后缀桶的最左位置（用符号∧标记），然后查看（用符号@标记）并试图推导其前继。因为是L类型，所以我们将插入到的最左空位。以此类推，可在从左向右遍历sa的过程中归纳排序所有L类型后缀，结果如第17行所示。最后，第19-20行首先寻找各后缀桶的最右位置（用符号∧标记），然后查看（用符号@标记）并试图推导其前继。因为是S类型，所以我们将插入到的最右空位。以此类推，可在从右向左遍历sa的过程中归纳排序所有S类型后缀，结果如第28行所示。

可检测引理6.2中的条件是否成立来验证后缀数组：

引理6.2 对任意采用归纳排序思想设计的后缀数组构造算法程序，当以下条件在程序执行的最高递归层成立时，输出的是输入字符串的后缀数组：

1. 最高递归层的是正确的。
2. 最高递归层的解推导阶段的实现是正确的。

证明：令和为和在最高递归层推出前继。因为最高递归层的解推导阶段的实现是正确的，所以有。这表明引理6.1的条件1）成立。假设且，因为最高递归层的解推导阶段的实现是正确的，所以且。将替换为并重复前述推理过程直到为S\*类型，则包含重复后缀。但因为是正确的，所以必不包含重复后缀，导致矛盾。这表明引理6.1的条件2）成立。

引理6.2的逻辑是：采用归纳排序思想设计的外存后缀数组构造算法虽然实现复杂，但其最高递归层的解推导阶段的代码量很小（C++代码不到百行），可特化该段代码以保证其实现是正确的。在此前提条件下，可将sa的验证问题转化的验证问题，后者可由一个稀疏后缀数组验证方法解决。

本章提出两个后缀数组事中验证方法，方法需与采用归纳排序思想设计的后缀数组构造算法联用，在构造的最高递归层的解推导阶段检测引理6.2的条件是否成立。其中，方法1是概率性验证方法，可验证后缀数组和最长公共前缀数组；方法2是确定性验证方法，仅可验证后缀数组。

## 6.2 方法1

我们提出的

### 6.2.1 方法原理

Lemma 4.2 indicates that we can check sa∗ instead to perform SA verification at the top recursion level if the second condition of Lemma 4.2 is always true. In fact, the code snippet for the induction phase at the top recursion level only consists of tens of C++ code lines. This is considerably smaller than the whole program for a disk-based IS suffix sorting algorithm. Following the idea, we assume S1”-S4” are correctly implemented and propose in this section two methods for checking SA, where the methods adopt different approaches to ensure the correctness of sa∗ . In particular, our first method can be used to check both suffix and LCP arrays.

### 6.2.2 算法设计

T

## 6.3 算法2

### 6.3.1 算法原理

### 6.3.2 算法设计

## 6.4 算法优化

### 6.4.1 设计优化

### 6.4.2 工程优化

## 6.5 实验分析

## 6.6 本章总结参考文献

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.

# 结 论

结论是理论分析和实验结果的逻辑发展，是整篇论文的归宿。结论是在理论分析、试验结果的基础上，经过分析、推理、判断、归纳的过程而形成的总观点。结论必须完整、准确、鲜明、并突出与前人不同的新见解。

书写格式说明：

标题“结论”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：小二

结论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 1.25，间距：前段、后段均为0行。附录A 附录内容名称

以下内容可放在附录之内：

(1) 正文内过于冗长的公式推导；

(2) 方便他人阅读所需的辅助性数学工具或表格；

(3) 重复性数据和图表；

(4) 论文使用的主要符号的意义和单位；

(5) 程序说明和程序全文。

这部分内容可省略。如果省略，删掉此页。

书写格式说明：

标题“附录A 附录内容名称”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。

附录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。

# 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. 在审。（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