后缀树的并行构造算法^{*})

葛 健 王国仁 于 戈

(东北大学信息学院 沈阳110004)

摘 要 后缀树是一种非常重要的数据结构,它在与字符串处理相关的各种领域里有着非常广泛的应用。构造后缀树是应用后缀树解决问题的前提和关键。虽然很多现有的后缀树构造算法都是线性时间和空间的,但是,当被索引的字符串的长度很长时,构造其后缀树所消耗的时间和空间仍将非常巨大,这极大地限制了后缀树的实际应用。而并行技术是解决这一问题的很好途径,因此人们提出了后缀树的并行构造算法。本文对后缀树的三种并行构造算法进行了综述,通过系统的比较和分析,总结出当前存在的问题,并指明了下一步的研究方向。

关键词 后缀树,并行构造算法

Parallel Construction of Suffix Trees

GE Jian WANG Guo-Ren YU Ge

(Faculty of Information Science & Engineering, Northeastern University, Shenyang 110004)

Abstract The suffix tree is a very important data structure, which finds a wide variety of applications in many areas related to string processing. While using suffix trees, how to construct the suffix trees efficiently is the key problem. The serial suffix tree construction algorithms available now can run in linear time and space, however, when the string is too long, the time and space consumption is still unendurable, which greatly restricts the employability of suffix trees. While, parallelism seems to be a good way to solve the problem, so some researchers present parallel construction algorithms of suffix trees. In this paper, we survey three kinds of parallel construction algorithms of suffix trees and give them a detailed comparison. Also, we point out the remaining problems and research directions in this area.

Keywords Suffix tree, Parallel construction

1 引言

后缀树是存储了给定字符串的所有后缀的压缩 trie 树(compact trie)^[1]。它将给定字符串的所有后缀以树的形式索引起来,从而使原字符串的任何一个子串都出现在树中由根结点开始的某一路径上。基于这种数据结构,我们可以在线性时间内解决很多字符串处理方面的复杂的问题。例如字符串匹配问题,即字符串 S 中是否含有字符串 t 的问题。传统算法的时间复杂度为 $O(|S| * |t|)$,优化的查找算法(例如 Boyer-Moore 算法)的时间复杂度可以达到 $O(|S| + |t|)$,而利用后缀树的查找算法可以在 $O(|t|)$ 时内完成,即线性于查询字符串的长度而与查找的字符串长度无关。由于这种特点,后缀树被广泛应用在与字符串处理相关的各种领域里,如生物信息学、分子生物学以及数据处理、信息检索、数据压缩和信号处理等领域^[1]。

由于后缀树的建立时间与被索引的字符串长度成正比,而在后缀树上进行查找的时间仅与要查找的模式字符串的长度成正比,因此,在应用后缀树解决问题时,如何高效地构造后缀树是问题的关键。目前,对于串行的后缀树构造算法的研究已经非常成熟,人们提出了很多线性时间和空间的算法。其中 Weiner 的算法^[2]、McCreight 的 McC 算法^[3]和 UkKonen 的 UkK 算法^[4]是三种经典算法,它们都能在 $O(n)$ 时间和空间内完成(n 是字符串的长度)。Weiner 的算法是最早提出的

线性时间算法,但该算法使用了较多的存储空间;McC 算法使用后缀链接(Suffix Link)等技术来使该算法时间复杂度减少到线性,与其它两种算法相比 McC 算法更高效,使用的存储空间更少;UkK 算法也使用了后缀链接技术,其最大特点是它是一种在线(On-line)算法,也就是说算法的任何阶段都生成当前子串的后缀树。

虽然这些算法的性能已经很令人满意,但是在某些领域里这些算法并不能满足实际的需求。例如在生物信息领域,一条 DNA 序列的长度可能达到数 G,对于这种超长、超大规模的生物序列,这些传统的后缀树算法就不再适用了。其原因在于:一方面,上述这些算法都是基于主存的算法,而且它们对主存的要求是倍数于字符串的长度的。例如:McC 算法^[3]要求 28 倍于字符串长度的存储空间,而 Kurtz^[5]提出的算法要求 20 倍,Manber 和 Myers 提出的算法^[6]要求 18.8~22.4 倍, Karkkainen 提出的算法^[7]要求 15~18 倍, Crochemore 提出的算法^[8]要求 32.7 倍, Strmat 软件包^[9]要求 24~28 倍。很显然,对于这种超长字符串,这些基于主存的算法要受到主存大小的限制,因此它们并不适用于这一领域。基于这种情况, Ela Hunt 等人提出了一些基于磁盘的后缀树构造算法^[10~12],很好地解决了主存限制的问题。另一方面,由于这些生物序列的长度非常长,因此构造其后缀树所消耗的时间也很长。如果为了避免主存限制而使用基于磁盘的算法,其时间开销更是巨大。例如,使用文[12]中的基于磁盘的算法,构造长度为

^{*})本文受国家自然科学基金项目资助(60273079)。

20.5Mbp 的 DNA 序列的后缀树所需要的时间是34小时。由此可见,当字符串的长度很长时,构造其后缀树的时间和空间开销是令人无法忍受的。因此,人们提出了后缀树的并行构造算法来解决这些问题。

第一个并行后缀树构造算法^[13]是 Landau 和 Vishkin 在 1986 年提出的。该算法的时间复杂度为 $O(\log n)$, 工作量是 $O(n^2)$ 。Apostolico 等人提出的算法^[14]使用了命名技术^[15]将工作量减少到 $O(n \log n)$, 其空间复杂度却是 $O(n^2)$ 。Sahinalp 和 Vishkin 提出的算法^[16]以及 Hariharan 提出的算法^[17]进一步提高了性能。这些并行算法都是工作在 PRAM (Parallel Random Access Machine) 模型^[18]上的。PRAM 模型是一种理论模型,是研究并行技术的仿真并行环境,我们将在后面的部分进行介绍。

2 PRAM 模型

PRAM (Parallel Random Access Machine) 模型,即并行随机访问机模型,是模拟并行计算机的最简单的方法之一。PRAM 模型包括一系列有序的处理器,处理器的数量是不受限制的。该模型假设所有处理器都可以高效、同步地访问一个全局的共享内存,其访问效率就如同访问其本地的内存一样。PRAM 模型的主要优点是它易于实现并行而且不考虑并行计算的通信和同步代价。

因此,PRAM 模型所提供给我们的是这样一种并行环境:一系列有序的完全同步的处理器和一个全局的共享内存。这使得 PRAM 模型成为并行算法设计时最常用的模型。

根据对某一个内存单元并发访问策略的不同,有四种不同的 PRAM 模型。ER (Exclusive Read) 或 EW (Exclusive Write) PRAM 模型,即唯一读或唯一写 PRAM 模型,是不允许共享内存的并发访问的。而 CR (Concurrent Read) 或 CW (Concurrent Write) PRAM 模型,即并发读或并发写 PRAM 模型,则允许共享内存的并发读写。我们将这几种访问策略组合起来可以得到四种 PRAM 模型:EREW、ERCW、CREW、CRCW PRAM 模型。

而且,对于 CW PRAM 模型,有如下四种规则来决定并发写操作如何处理。

- (1) Arbitrary PRAM 模型:如果多个处理器同时写某一个内存单元,那么只有一个处理器会得到写权限;
- (2) Common PRAM 模型:多个处理器只能向某一内存单元中写入相同的值;
- (3) Priority PRAM 模型:优先级最高的处理器得到写权限;
- (4) Combining PRAM 模型:将所有欲写入某一内存单元的值作某种操作,然后将结果写入该单元。

3 Apostolico 等的算法

该算法是 1988 年由 Apostolico 等人提出的,它工作在 Arbitrary CRCW-PRAM 模型上。该算法并行构造字符串 $x[1:n]$ 的后缀树,它可以分为两个阶段:命名阶段和精炼阶段。命名阶段使用 Karp-Miller-Rosenberg 的命名技术^[15]为字符串 x 的所有长度为 2^r ($0 \leq r \leq \log n$) 的子串命名,并保证相同的子串得到相同的名字;精炼阶段构造 $\log n + 1$ 棵精炼树 $D^{(\log n)}, D^{(\log n - 1)}, \dots, D^{(0)}$, 每一棵精炼树都更接近于所求的后缀树,最后得到的 $D^{(0)}$ 即为字符串 x 的后缀树。

3.1 Karp-Miller-Rosenberg 命名技术

该算法的目的是为字符串 $x[1:n]$ 的所有长度为 2^r ($0 \leq r \leq \log n$) 的子串作标识。标识也称为名字,是 $[1:n+1]$ 内的整数。相同的子串必须得到相同的名字。

算法的主要操作是构造 n 个数组 $NAME_i[g]$, 其中 $1 \leq i \leq n, 0 \leq g \leq \log n$ 。 $NAME_i[g]$ 的值为子串 $x[i:i+2^g-1]$ 的名字,即从 i 开始,长度为 2^g 的子串的名字。而且,当且仅当 $x[i:i+2^g-1] = x[j:j+2^g-1]$ 时, $NAME_i[g] = NAME_j[g]$ 。有了这样一个数组,我们就可以在常数时间内判断两个字符串是否相等,前提是这两个字符串的长度都是 2 的幂。

我们知道字符串 $x[i:i+2^g-1]$ 可以被看作两个字符串 $x[i:i+2^{g-1}-1]$ 和 $x[i+2^{g-1}:i+2^g-1]$ 的连接,这样 $NAME_i[g]$ 的值就依赖于值对 $(NAME_i[g-1], NAME_{i+2^{g-1}}[g-1])$ 。因此,算法用递归的方法计算 $NAME_i[g]$ 的值。为了保证为相等的子串分配相同的名字,算法使用一个 $n \times (n+1)$ 矩阵——BB。BB 的特点是:当有多个处理器向 BB 的同一位置写数据时,只有一个处理器能够得到写权限,并将值写入 BB。

该算法使用 n 个处理器 $P_1, \dots, P_n$, 字符串 x 的第 i 个字符 $x[i]$ 被分配给处理器 P_i ($1 \leq i \leq n$)。算法首先为长度为 1 的子串命名,在这一阶段中每个处理器 P_i ($1 \leq i \leq n$) 并行地将 $NAME_i[0]$ 赋值为 $x[i]$;之后,算法为所有长度为 2^r ($1 \leq r \leq \log n$) 的子串命名。在计算 $NAME_i[g]$ 时,假设所有长度为 2^{g-1} 的子串已经得到了名字,即 $NAME_i[g-1]$ ($1 \leq i \leq n$) 已经计算完毕。令 $\eta_1 = NAME_i[g-1]$, $\eta_2 = NAME_{i+2^{g-1}}[g-1]$, 则每个处理器 P_i ($1 \leq i \leq n$) 并行地将其编号 i 写入 $BB[\eta_1, \eta_2]$ 的位置。最后令 $NAME_i[g] = BB[\eta_1, \eta_2]$ 。因为只有一个处理器能将其编号写入 $BB[\eta_1, \eta_2]$, 所以保证了相同的子串得到相同的名字。

3.2 精炼算法

经过命名阶段后,所有长度为 2^r ($0 \leq r \leq \log n$) 的子串都得到了名字,而且相同的子串拥有相同的名字。精炼阶段包括一个初始步骤和 $\log n$ 个精炼步骤。

在初始步骤中初始的精炼树 $D^{(\log n)}$ 被构造出来。与后缀树相比,精炼树的结构有两点不同:其一,后缀树的边是从父亲结点指向孩子结点,而精炼树却相反,是从孩子结点指向父亲结点;其二,后缀树边上的标识在精炼树中被移到了结点上。初始的精炼树 $D^{(\log n)}$ 包括一个根结点和 n 个叶子结点。每个叶子结点上都标识着 x 的一个后缀。算法将叶子结点由 1 到 n 编号,则叶子结点 i 上的标识为后缀 $x[i:n]$ 。

其后的 $\log n$ 个精炼步骤将对 $D^{(\log n)}$ 进行提炼,生成 $D^{(\log n - 1)}, D^{(\log n - 2)}, \dots$, 直到生成 $D^{(0)}$ 。 $D^{(0)}$ 即为 x 的后缀树(除了结构稍有不同)。为了保证最后得到的 $D^{(0)}$ 是真正的后缀树,算法要求精炼树 $D^{(r)}$ 满足条件 $condition(r)$ ($0 \leq r \leq \log n$):

- (1) $D^{(r)}$ 有 n 个叶子结点,但不能有“一元”结点(即只有一个孩子的结点);
- (2) 每个内部结点上标识着 x 的一个子串;
- (3) 对于任意两个兄弟结点,它们的标识不允许存在长度为 2^r 的公共前缀;
- (4) 每个叶子结点 i 上都标识着 x 的唯一后缀 $x[i:n]$ ($0 \leq i \leq n$)。

精炼过程的主要操作是将 $D^{(r)}$ 精炼为 $D^{(r-1)}$ 。对于 $D^{(r)}$ 中的任意一个结点,在其孩子结点中,若某些结点的标识具有长度为 2^{r-1} 的公共前缀,则算法将这些结点归为一个等价类。这

一过程是通过比较各个结点标识的长度为 2^{i-1} 的前缀的名字实现的。对于任意一个等价类 C (C 中的结点记为 $c_i, i=1, 2, \dots$), 算法在精炼树 $D^{(i-1)}$ 中生成一个新结点 w , w 的标识为 c_i 的标识的公共前缀(长度为 2^{i-1})。在精炼树 $D^{(i-1)}$ 中 w 成为 c_i 的父亲结点, 而 c_i 原来的父亲结点在 $D^{(i-1)}$ 中将成为 w 的父亲结点。 c_i 的标识也要作相应的调整(去掉其公共前缀)。最后, 算法删除精炼树 $D^{(i-1)}$ 中可能生成的“一元”结点, 并调整相应结点的标识。

这样, 我们就得到了精炼树 $D^{(r)}$ 。精炼树 $D^{(r)}$ 满足 condition(r), 我们可以证明, 经过上述精炼算法得到的精炼树 $D^{(r)}$ 满足 condition($r-1$)。证明过程请参见文[14]。

4 Sahinalp 和 Vishkin 的算法

这个算法是由 Sahinalp 和 Vishkin 在1994年提出的, 它同样工作在 Arbitrary CRCW-PRAM 模型上。Sahinalp 和 Vishkin 对 Apostolico 等人的算法进行了分析, 认为其命名技术是算法性能的瓶颈。因为在这一过程中, 需要为字符串 x 的所有长度为 2^i 的 $n-2^i+1$ 个子串命名($1 \leq i \leq \log n$), 这些子串的数量是与 $n \log n$ 成比例的。而且, 应用 Karp-Miller-Rosenberg 的命名技术只为字符串 x 的一部分子串命名也是一个很困难的问题。这些都导致 Apostolico 的算法的计算工作量不能达到线性^[6]。

因此 Sahinalp 和 Vishkin 使用 Deterministic Coin Tossing (DCT) 技术^[9]为子串分配标识, 在这一过程中他们打破了原有的对称性, 从而将要作标识的子串的数量减少到线性, 提高了性能。这里给出基本算法, 优化的算法请参见文[16]。该算法构造字符串 x 的后缀树, 有三个主要阶段。

4.1 第一阶段

在第一阶段中, 算法构造 $\log n$ 个字符串 $S(0), S(1), \dots, S(\log n)$ 。其中 $S(0)=x$, 而 $S(i)$ ($1 \leq i \leq \log n$)是通过“压缩” $S(i-1)$ 得到的。为了压缩 $S(i-1)$ 得到 $S(i)$, 算法首先将 $S(i-1)$ 分片成至多 $n/2^i$ 个片段。然后, 为每一个片段作标识(标识为1到 n 之间的整数), 这些标识被称为 i -标识。每个 i -标识都是字符串 $S(i)$ 中的一个字符, 称之为 i -字符。所有的 i -字符按照顺序连接起来构成了字符串 $S(i)$ 。上述的分片和标识过程必须遵循下面两个重要的规则:

(1) 分片一致性: 若 x_i 和 x_j 表示字符串 x 的两个相等的子串, 那么在对 x 进行分片时, x_i 和 x_j 的分片方式必须相同(但 x_i 和 x_j 的左右边缘部分的分片方式可能不同)。

(2) 标识一致性: 所有包含相同子串的片段必须得到相同的标识。

事实上, 每个 i -标识都代表了原字符串 x 的一个子串, 称之为 i -子串。由此可见, 构造这 $\log n$ 个字符串的过程实际上就是为字符串 x 的子串命名的过程。

4.2 第二阶段

第一阶段中的分片一致性规则是一个很弱的规则, 因为它不能保证任意两个相等的子串被以相同的方式分片并得到相同的标识。这是由于两个子串的边缘部分可能以不同的方式分片造成的。这样, 在构造后缀树时, 算法就不能正确地识别两个相等的子串。为了解决这一问题, 算法的第二阶段在 $S(0), S(1), \dots, S(\log n)$ 的基础上又构造另外 $\log n$ 个字符串 $C(0), C(1), \dots, C(\log n)$ 。其中 $C(0)=S(0)$, $C(i)$ 是由 $S(i)$ 构造出来的, 而且 $|C(i)|=|S(i)|$ ($0 \leq i \leq \log n$)。字符串 $C(i)$ 中的字符称为 i -核心。事实上 i -核心是通过在 $S(i)$ 中的 i -字符的

左右两端适当地加上一个扩展得到的。更准确地说, $C(i)$ 的第 k 个 i -核心是由 $S(i)$ 的第 k 个 i -字符扩展得到的。扩展规则请参见文[16]。这样得到的 $C(i)$ 实际上是 $C(i-1)$ 的“松弛”分片, 即对 $C(i-1)$ 分片所得到的各个片段之间可能有重叠的部分。这样, 当我们构造后缀树时, 相等的子串(包括其边缘部分)将会被覆盖在一个 i -核心里, 从而保证了算法能够正确地识别相等的子串。

4.3 第三阶段

算法逐步构造字符串 $C(\log n), \dots, C(0)$ 的后缀树。令 $T(i)$ 为字符串 $C(i)$ 的后缀树, $T(i)$ 是在 $T(i+1)$ 的基础上构造的。最后得到的 $T(0)$ 就是我们所求的 x 的后缀树。为了由 $T(i+1)$ 生成 $T(i)$, 算法首先将 $T(i+1)$ 的边上的标识, 也就是 $(i+1)$ -核心, 替换成它们所包含的 i -核心。由于 i -核心之间可能存在重叠, 因此在替换过程中要避免重叠的产生。经过这一过程, 算法得到了一个中间结果 $T(i)_0$ 。对 $T(i)_0$ 进行一些预处理之后, 应用 CONTRACT 过程, 得到了另一个中间结果 $T(i)_1$ 。再对 $T(i)_1$ 应用 MERGE 过程, 从而得到 $T(i)$ 。具体细节请参见文[16]。

5 Hariharan 的算法

这个算法是由 Hariharan 在1994年提出的, 它工作在 CREW-PRAM 模型上。该算法是针对于二进制字符串而提出的, 而且是第一个优化工作量的并行算法。与前面两个算法不同, 该算法不需要为子串命名, 而且它利用了字符串和后缀树自身的一些性质。我们给出算法的主要思想, 有兴趣深入了解的读者请参见文[17]。

我们首先定义字符串 x 的 r -后缀树的概念: 对于某个整数 $r(r \geq 1)$, 令 r -后缀树为一棵压缩的 trie 树, 它存储了字符串 x 的所有长度为 r 的子串以及长度小于 r 的后缀(不包括 '\$'), 记为 r -ST。显然, 当 $r > n$ 时 r -ST就是 x 的后缀树。该算法构造字符串 x 的后缀树, 其主要的操作就是构造一系列的后缀树: r -ST, $2r$ -ST, 2^2r -ST, $\dots, 2^s r$ -ST, 其中 $r=2^{\lceil \log^3 n \rceil}$, s 为使 $2^s r \geq n+1$ 的最小整数。显然, 当算法生成 $2^s r$ -ST时就得到了所求的后缀树。我们用 T_i 代表 $2^i r$ -ST($0 \leq i \leq s$), 那么算法包括 $s+1$ 个阶段, 第 i 阶段构造后缀树 T_i 。

在初始阶段, 算法应用 McCreight 算法的并发版本来并行地构造 T_0 。首先, 算法将 x 分片成有重叠的长度为 $2r$ 的片段, 每对相邻的片段之间重叠 r 个字符。每个片段被分配给一个处理器。各个处理器都运行 McCreight 算法, 将长度为 r 的子串插入到同一个压缩的 trie 树中。这样我们就得到了 T_0 。

在后面的 s 个阶段中, 算法在 T_0 的基础上, 逐步构造 $T_1, T_2, \dots, T_s$ 。最后得到的 T_s 就是字符串 x 的后缀树。在第 i 阶段中, 算法构造 T_i , 可以分为两个步骤。因为在某个阶段中, 所有特定长度的子串都将被处理, 所以在第一步中, 算法首先从上述的即将被处理的子串集合中选取一个子集并构造这一子集的压缩 trie 树。这一过程中使用了前面提到的字符串的组合性和周期性, 以及 Cole 的并行合并排序算法^[20]。在第二步中, 算法应用压缩 trie 树的重复性, 由第一步中得到的压缩 trie 树构造出所有即将被处理的子串的压缩 trie 树, 从而得到了 T_i 。

6 对三种算法的比较

表1给出了三种并行算法的比较, 包括工作环境、时间复杂度、空间复杂度和工作量。Apostolico 等的算法工作在

Arbitrary CRCW-PRAM 模型上。该算法使用命名技术提高了算法的性能,其时间复杂度为 $O(\log n)$,工作量为 $O(n \log n)$,空间复杂度为 $O(n^2)$,这里 n 是字符串的长度。该算法有一个工作在 CREW-PRAM 模型上的空间优化算法,其空间复杂度被优化为线性,而时间复杂度为 $O(\log^2 n)$,工作量为 $O(n \log^2 n)$ 。

Sahinalp 和 Vishkin 提出的算法也工作在 Arbitrary CRCW-PRAM 模型上。该算法使用了打破对称性和命名等技术,其时间复杂度为 $O(\log^2 n)$,工作量为 $O(n \log \log n)$,空间复杂度为 $O(n^{1+\epsilon})$, $\epsilon > 0$ 。对于固定字母表,该算法的工作量减少为 $O(n)$ 。

如果字符串 x 中的字符来源于一个通用的无限的字母表,那么任何并行算法的工作量都不会优于 Apostolico 等的算法。但是,即使对于二进制字母表,Apostolico 等的算法仍然在上述的性能下运行^[17]。Hariharan 的算法是针对二进制字母表提出的,它工作在 CREW-PRAM 模型上而且是第一个工作量优化的算法,其时间复杂度为 $O(\log^4 n)$,工作量和空间复杂度均为 $O(n)$ 。与前面两种算法不同,该算法使用了字符串和后缀树自身的性质,例如字符串的组合型和周期性,后缀树的重复性等。该算法同样可以处理来源于固定字母表的字符串,并维持其原有的性能。对于来源于通用字母表的字符串,算法的时间复杂度为 $O(\log^4 n)$,工作量和空间复杂度均为 $O(n \log |\Sigma|)$, $|\Sigma|$ 为字符串中不同字符的个数。

表1 三种并行算法的比较

	工作环境	时间复杂度	空间复杂度	工作量
Apostolico 等的算法	Arbitrary CRCW-PRAM	$O(\log n)$	$O(n^2)$	$O(n \log n)$
Sahinalp 和 Vishkin 的算法	Arbitrary CRCW-PRAM	$O(\log^2 n)$	$O(n^{1+\epsilon})$	$O(n \log \log n)$
Hariharan 的算法	CREW-PRAM	$O(\log^4 n)$	$O(n)$	$O(n)$

由表1我们可以看到,虽然这些并行算法大大降低了后缀树的构造时间,但是它们对空间的需求却并没有降低,甚至有的算法的空间复杂度要高于现有的串行构造算法。也就是说,这些并行算法并没有摆脱主存大小的限制。因此,我们认为,对于后缀树的并行构造算法的研究应该集中在如何降低算法的空间复杂度的问题上。同时,为了适应生物信息等领域的需求,基于磁盘的后缀树将是其必然的选择。因此,如何并行地构造基于磁盘的后缀树,以降低其构造时间同样是一个值得研究的方向。

结论 随着生物信息等领域的快速发展,在全球范围内积累了越来越多的超长、超大规模的字符串数据。面对这种数据,传统的后缀树构造算法已经不再适用。而并行的构造算法逐渐成为这一领域的研究焦点。本文简述了后缀树的三种并行构造算法,并对它们进行了比较和分析,发现目前的并行构造算法没有解决主存限制的问题。因此,如何降低并行算法的空间需求以及如何并行地构造基于磁盘的后缀树将是这一领域的研究重点。

参考文献

- Grossi R, Italiano G F. Suffix Trees and their Applications in String Algorithms. In: Proc. 1st South American Workshop on String Processing (WSP 1993), Sep. 1993. 57~76
- Weiner P. Linear Pattern Matching Algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, 1973. 1~11
- McCraith E M. A Space-economical Suffix Tree Construction Algorithm. J. ACM, 1976, 23: 262~272
- Ukkonen E. On-line Construction of Suffix Trees. Algorithmica, 1995, 14: 249~260
- Kurtz S. Reducing the Space Requirement of Suffix Trees. Software Practice and Experience, 1999, 29: 1149~1171
- Manber U, Myers E W. Suffix Arrays: A New Method for On-line String Searches. SIAM Journal on Computing, 1993, 22(5): 935~948
- Karkkainen J. Suffix cactus: a cross between suffix tree and suffix array. In: Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM'95), 1995. 191~204
- Crochmore M, Verin R. Direct construction of compact acyclic word graphs. In: Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM'97), 1997. 116~129
- Knight J, Gusfield D, Stoye J. The strmat software-package, 1999. <http://www.cs.ucdavis/~gusfield/strmat.tar.gz>
- Hunt E, Jama P. Stores and Suffix Tree Indexing for Bioinformatics Applications. In: Proc. of ECOOP'00, 2000
- Hunt E, Irving R W, Atkinson M. Persistent Suffix Trees and Suffix Binary Search Trees as DNA Sequence Indexes: [Technical Report TR-2000-63]. Department of Computing Science, University of Glasgow, UK, 2000
- Hunt E, Atkinson M P, Irving R W. A Database Index to Large Biological Sequences. In: Proc. of 27th Intl. Conf. on Very Large Data Bases, 2001. 139~148
- Landau G, Vishkin U. Introducing Efficient Parallelism into Approximate String Matching. In: Proc. 18th ACM Symposium on Theory of Computing, 1986. 220~230
- Apostolico A, Iliopoulos C, Landau G, Shieber B, Vishkin U. Parallel Construction of a Suffix Tree with Application. Algorithmica, 1988, 3: 347~365
- Karp R, Miller R E, Rosenberg A L. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In: Proc. 4th ACM Symposium on Theory of Computing, 1972. 125~136
- Sahinalp S J, Vishkin U. Symmetry breaking for suffix tree construction. In: Proc. 26th ACM Symposium on Theory of Computing, 1994
- Hariharan R. Optimal Parallel Suffix Tree Construction. In: Proc. 26th ACM Symposium on Theory of Computing, 1994
- J. Ja'Ja'. An Introduction to Parallel Algorithms. Addison-Wesley, Reading, MA, 1992
- Cole R, Vishkin U. Deterministic Coin Tossing with Application to Parallel List Ranking. Information and Control, 1986, 70: 32~53
- Cole R. Parallel Merge Sort. SIAM J. Comput, 1988, 17: 770~785