软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(12):3799−3819 [doi: 10.13328/j.cnki.jos.005326]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种带稀疏间隙约束的并行模式匹配算法*

周开来[1,2,3], 陈 红[1,2], 熊子绎[1,2], 李翠平[1,2], 孙 辉[1,2]

[1](数据工程与知识工程国家教育部重点实验室(中国人民大学),北京 100872)

[2](中国人民大学 信息学院,北京 100872)

[3](郑州轻工业大学 软件学院,河南 郑州 450001)

通讯作者: 陈红, E-mail: chong@ruc.edu.cn

摘 要: 带通配符的模式匹配是一个经典的研究问题,带有可变间隙约束的模式匹配是近年来比较热门的研究方向.为适应某些查询精度要求较高的应用领域,提出一种在稀疏间隙约束条件下求解模式匹配完备解的算法SGPM-SAI(pattern matching with sparse gaps constraint based on suffix automaton index).SGPM-SAI 通过对文本串预处理,建立一种称为 W-SAM 的图索引结构,然后对模式串分段查找 EndPos 集合,最后以集合归并求交的方法得到模式匹配的完备解.实验结果表明:在不考虑预处理时间的情况下,相比几种最典型的模式匹配算法(KMP,BM,AC,suffix array),SGPM-SAI 算法性能优势显著,至少高出 3~5 倍.通过与 SAIL 算法的最新优化版本(SAIL-Gen)进行比较,在稀疏间隙约束条件下,SGPM-SAI 的性能要显著优于 SAIL-Gen 算法.此外,为有效利用现代处理器的大规模并行处理单元,提出了并行优化后的算法 Parallel SGPM-SAI.实验结果表明:Parallel SGPM-SAI 算法的加速效果显著,且具有良好的并行可扩展性,能够充分利用现代众核处理器的高并行计算优势.

关键词: 模式匹配;稀疏间隙约束;后缀自动机;并行算法;通配符

中图法分类号: TP311

中文引用格式: 周开来,陈红,熊子绎,李翠平,孙辉.一种带稀疏间隙约束的并行模式匹配算法.软件学报,2018,29(12):3799−3819. http://www.jos.org.cn/1000-9825/5326.htm

英文引用格式: Zhou KL, Chen H, Xiong ZY, Li CP, Sun H. Parallel pattern matching algorithm with sparse gap constraint. Ruan Jian Xue Bao/Journal of Software, 2018,29(12):3799−3819 (in Chinese). http://www.jos.org.cn/1000-9825/5326.htm

Parallel Pattern Matching Algorithm with Sparse Gap Constraint

ZHOU Kai-Lai[1,2,3], CHEN Hong[1,2], XIONG Zi-Yi[1,2], LI Cui-Ping[1,2], SUN Hui[1,2]

[1](Key Laboratory of Data Engineering and Knowledge Engineering of the Ministry of Education (Renmin University of China), Beijing 100872, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](School of Software, Zhengzhou University of Light Industry, Zhengzhou 450001, China)

Abstract: Pattern matching with wildcards is a classic problem, and matching with variable gap constraints is a popular direction in this field in recent years. In order to meet the requirement of high accuracy in some query applications, this paper proposes an algorithm

* 基金项目: 国家重点研发计划(2016YFB1000702); 国家重点基础研究发展计划(973)(2014CB340402); 国家高技术研究发展计划(863)(2014AA015204); 国家自然科学基金(61272137, 61202114, 61532021); 国家社会科学基金(12&ZD220); 中国人民大学科学研究基金(中央高校基本科研业务费专项资金资助)项目成果(15XNLQ06)

Foundation item: National Key Research & Develop Plan (2016YFB1000702); National Basic Research Program of China (973) (2014CB340402); National High Technology Research and Development Program of China (863) (2014AA015204); National Natural Science Foundation of China (61272137, 61202114, 61532021); NSSFC (12&ZD220); Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (15XNLQ06)

收稿时间: 2017-02-13; 修改时间: 2017-05-25; 采用时间: 2017-07-03

(referred to as SGPM-SAI) to obtain a complete solution of pattern matching under the condition of sparse gaps constraint. SGPM-SAI firstly creates an index structure called W-SAM (wildcard suffix automation) for the preprocessed text, and then get EndPos collection for each pattern segmentation by searching string from W-SAM, and finally get the complete solution of pattern matching by means of EndPos sets intersection. Experimental results show that, regardless of pretreatment time, the performance of SGPM-SAI algorithm is at least 3~5 times higher than other competitive algorithms, such as KMP, BM, AC, suffix array. Compared with the latest version (SAIL-Gen) of SAIL algorithm, the performance of SGPM-SAI is significantly better under the condition of sparse gaps constraint. In addition, this paper introduces parallel process methods for SGPM-SAI algorithm so as to effectively utilize the massive parallel processing units of modern processors. Experimental results show that the acceleration of Parallel SGPM-SAI algorithm has significant effect, as well as good parallel scalability. This indicates that the presented method can take full advantage of the high parallel computation capability of modern many-core processors.

Key words: pattern matching; sparse gaps constraint; suffix automaton; parallel algorithm; wildcards

字符串模式匹配是计算机科学领域内的一个基础性问题,有着极其广泛的应用领域,比如文献检索、基因序列比对、频繁模式挖掘等.近年来,随着传感器网络技术的发展、生物信息学领域的进步以及互联网的普及,人们获取数据的能力显著提高,数据量呈现爆发式的增长.为此,寻求更加高效的模式匹配算法,成为学术界重新关注的焦点.

在现实生活中,有很多应用的查询具有某些不确定的成分,例如在数据库查询时,通常用缺失部分号码的车牌信息查询相关车主,如使用 SQL 语句 SELECT user_name FROM vehicles WHERE plate_number LIKE ‘京 AC?B?8’.此时精确字符串匹配方法无法满足用户的需要,这就需要引入带有通配符的串匹配.

带通配符的字符串匹配是一个传统的研究问题.Fischer 等人[1]最早提出了带通配符的模式匹配概念.通常会引入两种通配符“?”和“*”,其中,“?”表示任意单一字符,而“*”表示任意多个字符.以此为基础,此后的研究者提出了各种各样的带通配符的模式匹配方案.目前来看,这些研究成果主要分为 3 类[2]:(1) 具有固定长度的模式匹配[3,4],例如,模式“京 AC?B?8”长度固定为 7;(2) 具有任意长度的模式匹配,即字符之间可插入表示任意长度的通配符[5−7],例如,模式“北京*幸福小区”表示“北京”与“幸福小区”之间可插入任意多个字符;(3) 带有可变长度的间隙约束[8−13],即模式串中包含若干间隙,每个间隙$[a,b]$($a\leqslant b$)中,a表示所跨越的字符长度下界,b表示上界.例如,模式“中国[2,4]大学”表示“中国”和“大学”之间可跨越的字符最少是 2 个,最多是 4 个,“中国人民大学”、“中国科学技术大学”均满足匹配要求.可以看出,前两类问题可视为第(3)类的特例:当满足 $a=b$ 时,即第(1)类问题;当 $a=0$,$b=+\infty$时即第(2)类问题.从通用性考虑,本文研究第(3)类问题,即具有可变间隙约束的模式匹配问题.

具有可变间隙约束的模式匹配问题已被证明为是 NP-Hard 问题[11,34],因此,大多数研究者放弃寻求该问题的完备解,转而去寻找该问题的近似解.为有效降低问题的复杂度,目前典型的做法是引入各种约束条件限制解的空间,如文献[13,14]附加了 Non-overlap 条件,文献[8,9]则采用了 One-off 条件.但是这些约束降低了问题的求解精度,匹配结果损失了部分信息,并不适用于某些精度要求较高的应用领域.比如刑侦领域,警察需要根据模式匹配查找嫌疑人,就必须查全所有信息,否则犯罪嫌疑人就有可能漏网.

通过观察我们发现:在信息检索时,用户所提交的模式串通常不确定成分仅占很小部分的比例.这是因为人们已经意识到如果提交的模式串过于模糊,检索系统很难给出自己所满意的查询结果,因此总是尽可能提交较为精确的模式串.我们把这种情形称为带稀疏间隙约束的模式匹配问题,即间隙数目仅占模式串中很小比例,且每个间隙的变动范围维持在一个较小区间.另一方面,随着半导体工艺的进步,大容量、高带宽的内存体系已经成为现代计算机的标配.同时,具有高度并行计算能力的多核/众核处理器也成为现代服务器的主流配置,如 GPU(graphics processing unit),MIC(many integerated core),FPGA(field-programmable gate array)等.硬件技术的进步,客观上也为传统意义上比较困难的问题创造了解决的条件.鉴于上述考虑,本文对稀疏间隙约束条件下模式匹配完备解的求解问题进行了研究,其主要贡献如下.

(1) 提出了一种基于后缀自动机的带稀疏间隙约束的模式匹配算法 SGPM-SAI.该算法通过对文本串预处理,建立一种图索引结构 W-SAM(wildcard suffix automation);然后,在 W-SAM 上对模式串进行分段查找,以获取每个分段的 EndPos 集合;最后,对 EndPos 集合进行归并求交获得模式匹配的完备解;

(2) 提出了 SGPM-SAI 算法的并行化方案.为充分利用现代多核处理器的高并行计算能力,提出了 SGPM-SAI 算法的并行化方案,包括优化线程数目、任务均衡分配及 EndPos 集合并行化归并等;

(3) 验证了算法的有效性.与一些经典的模式匹配算法(包括 KMP、BM、suffix array、AC 自动机及 SAIL 算法)进行了大量对比实验,实验结果表明:在不考虑文本预处理时间的情况下,SGPM-SAI 算法较这些经典算法有着显著的性能提升.测试了 Parallel SGPM-SAI 算法的并行化效果,实验表明:在间隙可变度较大时,Parallel SGPM-SAI 算法的加速效果显著,且具有良好的并行可扩展性.

本文第 1 节对带有可变长度通配符的模式匹配相关工作进行总结.第 2 节对本文所研究的问题进行描述和定义.第 3 节介绍后缀自动机模型,包括其定义、性质、构建算法及改进方案.第 4 节介绍基于 W-SAM 的带稀疏间隙约束的模式匹配算法 SGPM-SAI,并分析算法的空间复杂度和时间复杂度.第 5 节给出 SGPM-SAI 算法的并行化方案,提出 Endpos 集合的并行获取算法及并行归并算法.第 6 节通过大量对比性实验验证 SGPM-SAI 算法的性能及并行化效果.第 7 节给出本文的结论及讨论.

1 相关工作

带有通配符的模式匹配问题源于 20 世纪 70 年代,Fischer 等人[1]首先在字符串匹配问题中引入了通配符概念,但他们的问题中,通配符的数目是固定的,为了适应更加灵活的应用需求.后来的研究中放宽了对模式串长度固定的要求,Manber 等人[16]最早研究了带可变长度通配符的模式匹配,在其研究中,将通配符描述为$[0,g]$间隙形式,其中,g 表示可变长度.但他们研究的方法只能处理单个可变间隙约束.伴随研究的不断深入,模式匹配技术由之前的单个可变间隙约束扩展为支持多个可变间隙约束,如,Navarro 等人[17]运用具有多个可变间隙约束的模式匹配技术进行蛋白质查找.在多可变间隙约束情况下,只要一个子模式的出现位置发生变化,就是该模式的一次出现,这直接导致解的规模组合呈指数级别增长.设序列长度为 n,模式串中间隙数量为 m,间隙间距为ω,则解空间为 $O(n\omega^m)$.显然,在这种增长模式下,当 m 较大时,要获得所有出现的匹配位置是不现实的.为了降低解空间的大小,当前的研究中引入了各种约束条件,如:Chen 等人[9]发现,序列中的每个字符被使用一次具有很重要的实际意义,于是引入了 One-off 条件并提出了 SAIL 算法.引入了 One-off 条件后,解的空间大小降低到 $O(n/m)$.另一种较为常见的约束条件是 Ding 等人提出的 Non-overlap 条件[14],该条件定义为:假设 $s_1=(c_0,c_1,\ldots,c_{m-1})$和 $s_2=(c'_0,c'_1,\ldots,c'_{m-1})$是长度为 m 的模式 P 的两个出现,若任取 $0\leqslant i\leqslant m-1$,都有 $c_i\neq c'_i$,则称 s_1 和 s_2 满足 Non-overlap 条件.

在具有可变长度通配符的模式匹配中引入上述各种约束条件限制,固然可以大幅降低解空间的规模,但代价是损失了大量匹配信息,并不适用于一些精度要求较高的应用场景,例如数据库的 SQL 查询.考虑到某些领域模式串中通配符的稀疏性以及现代处理器的高并行计算能力,仍有部分学者在研究如何获取模式匹配的完备解.一种朴素的匹配思想是:以间隙为分割点,将模式串分割为一系列不含通配符的字符串片段,然后分别查找每个片段在文本序列中的出现位置,最后根据这些片段的位置进行合并.上述思想的核心是如何快速获取每个子串的出现位置,这样,问题转换为不带通配符的模式匹配问题.解决这类问题有大量经典的方法,典型的算法如下.

(1) 单模式匹配算法:KMP(Knuth-Morris-Pratt)算法[18]、BM(Boyer-Moore)算法[19]、KR(Karp-Rabin)随机算法[20]等;

(2) 多模式匹配算法:Aho-Corasick 算法[21]、Wu-Manber 算法[22]等.

进入 21 世纪以来,随着硬件技术的进步,上述经典算法面向新硬件的优化版本也大量出现,比如面向 GPU 加速的 KMP 算法[23]、PFAC(parallel failureless-AC)算法[24]、GWM(GPU-based Wu-Manber)[25]等.

解决模式匹配问题的另一类思路是对文本进行预处理,建立某种形式的索引,然后用模式串在索引上进行查询.目前来看,这些索引结构主要有后缀树[26]、后缀数组[27,28]、哈希表[31]以及 DAWG(directed acyclic word graph)[32]图.Cole 等人[26]最早将后缀树作为基本索引结构引入到带有通配符的近似模式匹配问题中.但是树存储模式结构比较复杂,占用空间较大,因此后来出现了后缀树的一种变体作为索引结构,即后缀数组,从而压缩

了文本预处理的时间和空间代价[27,28].

DAWG最初是由Blumer等人提出的一种面向词的有向无环图[30],DAWG利用最小确定自动机模型将文本串的所有后缀索引起来形成图结构.设 $w\in\Sigma^*$,定义 $DAWG(w)$是能识别字符串 w 所有后缀的最小确定性有限状态自动机.为便于理解,后来学者广泛将 $DAWG(w)$称作 w 的后缀自动机.DAWG有许多优秀特性[31],比如:对于给定长度为$|p|$的模式串,我们可以利用 $DAWG(w)$在 $O(|p|)$时间复杂度内判别 p 是否属于 w 的子串.此外,不同于off-line 模式构造的后缀树和后缀数组,DAWG 能够以 on-line 模式构造,且构造的时间复杂度及空间复杂度均为 $O(n)$.Blumer等人提出的DAWG只支持不带通配符的字符串匹配,Inenaga等人对DAWG作了进一步扩展,提出了支持通配符的WDAWG(wildcard directed acyclic word graph)[32,33].WDAWG索引是通过在DAWG中加入通配符实现的,某一字符串 w 的自动机能够接受所有和自己匹配的带通配符的模式串.基于WDAWG索引的匹配算法将字符串匹配问题转化为了图匹配问题,在文本长度较短时,具有很高的匹配效率.但是,对于 WDAWG 这样的图数据结构存在着一个关键问题,即索引占用的空间过大.当字母表$|\Sigma|\geqslant 2$ 时,对于长度为 n 字符串 w, $WDAWG(w)$的状态节点数为 $O(n^2)$[33].显然,对于长文本,这种平方级的空间增长趋势使得 WDAWG 仅具有理论意义,难以应用到实践中.

为了既保持 DAWG 的线性增长空间需求,又能有效支持带通配符的模式匹配,本文对 DAWG 的数据结构和构建算法进行了改进,提出了支持带通配符的模式匹配自动机模型 W-SAM(wildcard suffix automation).同时,为了充分利用现代处理器的高并行计算能力,设计了适应稀疏间隙约束的并行模式匹配算法.

2 问题描述与定义

本文所研究的问题简单描述为:给定一个带有间隙约束的模式串 P,在一个大的文本 T 中进行查找,报告 P 在 T 中的所有出现位置.下面给出本文研究问题的形式化定义.

定义 1. 设Σ是字母表集合,Σ^*表示字符串的集合,带间隙约束的模式 P 定义为

$$\beta_0\Psi_0[a_0,b_0]\beta_1\ldots\beta_j\Psi_j[a_j,b_j]\beta_{j+1}\ldots\Psi_{m-2}[a_{m-2},b_{m-2}]\beta_{m-1},$$

其中,$\beta_j\in\Sigma^*$, $\Psi_j[a_j,b_j]$称为间隙(gap),$[a_j,b_j]$称为间距(gap range).设 x 表示单字符通配符的数目,则 x 可为$[a_j,b_j]$区间内任一整数,且 $0\leqslant a_j\leqslant x\leqslant b_j$.

例 1:假设$\Sigma=\{a,b,c\}$,$P=baa[2,3]c[0,2]ac$,则该模式中,$\beta_0=baa$,$\beta_1=c$,$\beta_2=ac$, $\Psi_0[a_0,b_0]=[2,3]$, $\Psi_1[a_1,b_1]=[0,2]$.

定义 2. 设模式 $P=\beta_0\Psi_0[a_0,b_0]\beta_1\ldots\beta_j\Psi_j[a_j,b_j]\beta_{j+1}\ldots\Psi_{m-2}[a_{m-2},b_{m-2}]\beta_{m-1}$,定义模式 $P_i=\beta_0[w_0]\beta_1\ldots\beta_j[w_j]\beta_{j+1}\ldots[w_{m-2}]\beta_{m-1}$ 为 P 的一个定长子模式,其中,$[w_j]$表示该处含有 w_j 个单字符通配符“?”,且 $a_j\leqslant w_j\leqslant b_j$.

显然,定长子模式具有以下性质:

$$|P_i|=\sum_{j=0}^{m-1}|\beta_j|+\sum_{j=0}^{m-2}w_j,\text{且}|P_i|_{\max}=\sum_{j=0}^{m-1}|\beta_j|+\sum_{j=0}^{m-2}b_j,|P_i|_{\min}=\sum_{j=0}^{m-1}|\beta_j|+\sum_{j=0}^{m-2}a_j.$$

例 2:设 $P=baa[2,3]c[0,2]ac$,则以下模式串集合:{*baa*??*cac*,*baa*??*c*?*ac*,*baa*??*c*??*ac*,*baa*???*cac*,*baa*???*c*?*ac*,*baa*???*c*??*ac*}均属于 P 的定长子模式,其最大长度为 11,最小为 8.

定理 1. 带间隙约束的模式,其定长子模式集合的大小等于所有间隙间距的乘积.

证明:设 $P=\beta_0\Psi_0[a_0,b_0]\beta_1\ldots\beta_j\Psi_j[a_j,b_j]\beta_{j+1}\ldots\Psi_{m-2}[a_{m-2},b_{m-2}]\beta_{m-1}$ 是定义在字母表Σ上的一个带间隙约束的模式,根据定义,其定长子模式由$\beta_0\square\beta_1\square\ldots\beta_j\square\beta_{j+1}\ldots\square\beta_{m-1}$ 和空位上的通配符组合而成,其中,$\beta_j\square\beta_{j+1}$ 之间所含通配符的数目取自区间$[a_j,b_j]$中的任一整数,因此根据乘法原理,这样的组合数共有:

$$(b_0-a_0+1)\times\ldots\times(b_j-a_j+1)\times\ldots\times(b_{m-2}-a_{m-2}+1).$$ □

例 3:根据定理 1,例 2 中,P 定长子模式集合的大小为$(3-2+1)\times(2-0+1)=6$.

定义 3. 给定带间隙约束的模式 P,定义 P 的间隙可变度为该模式定长子模式集合的大小.

由定理 1 可知:P 的间隙可变度既与间隙数目有关,也与间隙间距有关.

定义 4. 设有文本 $T\in\Sigma^*$,给定带间隙约束的模式 P,若在 T 上存在位置$\langle s,t\rangle$,满足以下条件.

(1) $T[s+k]=F[k]$或 $F[k]$=“?”,且 $0\leqslant k\leqslant t-s$;

(2) F 属于 P 的一个定长子模式,且$|F|=t-s+1$,

则称$\langle s,t\rangle$为模式 P 在文本串 T 上的一次出现.

本文的研究任务就是:当 P 的定长子模式集合规模较小时,研究如何查找 P 在 T 上的所有出现.

3 图索引结构

由定理 1 可知:带间隙约束的模式匹配实质上是一系列带通配符的定长子模式在文本串上的匹配,为了达到快速匹配的目的,对文本进行预处理,建立某种结构的索引,是比较理想的一种方法.如第 1 节所述,这种索引结构最常采用的形式是后缀树,后缀数组或者单向无环图.

后缀自动机是一种单向无环图结构,其最早由 Blumer 等人[30]于 1983 年首次提出,虽然该数据结构被证明是一种非常高效的全文索引结构,但由于图索引本身需要占用较大的内存空间,受限于当时的硬件环境,内存空间极其有限,因此并没有得到广泛的应用.近年来,由于硬件技术的进步,配置大容量 DRAM 的计算机系统已经非常普及,面向后缀自动机的各种应用也越来越受到关注.

3.1 经典后缀自动机模型

定义 5(后缀自动机(suffix automaton,简称 SAM)). 给定字符串 T,T 的 SAM 是一个能够识别 T 的所有后缀的确定性有限状态自动机.SAM 是一张有向无环图,其中,顶点是状态,而边代表了状态之间的转移.SAM 有一个唯一的初态和若干个终态,边上标记为转移的字符.一般用 $Trans(root,str)$表示从初态 $root$ 出发,沿字符序列 str 所能到达的状态.

例 4:字符串 T="$bcabcaabc$"的后缀自动机如图 1 所示.在图 1 中,自动机有唯一的初态 0,状态{2,11,12}均为终态.从初态 0 出发,沿着转移边到达任一终态,所经路径上的字符序列构成 T 的一个后缀.例如:沿路径 0→2→3→7→9→11 所构成的串"$caabc$"即是串 T 的一个后缀.

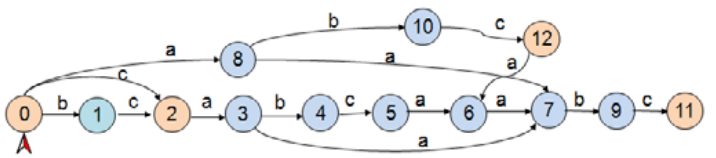


Fig.1 Suffix automaton of string "bcabcaabc"

图 1 "bcabcaabc"的后缀自动机

3.1.1 后缀自动机的性质

后缀自动机具有以下优良性质[33].

(1) 状态的数量:由长度为 n 的字符串 T 建立的后缀自动机的状态个数不超过 $2n-1$(对于 $n\geqslant 3$);

(2) 转移边的数量:由长度为 n 的字符串 T 建立的后缀自动机中,转移边的数量不超过 $3n-4$(对于 $n\geqslant 3$);

(3) 任取一字符串 str,若 str 不是原串 T 的子串,则 $Trans(root,str)=\varnothing$;若 str 是 T 的子串,则 $Trans(root,str)\neq\varnothing$;特别的,若 str 是 T 的后缀,则 $Trans(root,str)$属于终态集;

(4) 任取一字符串 str,判别 str 是否属于 T 的子串,查询时间复杂度是 $O(|str|)$.

3.1.2 后缀自动机的构建

后缀自动机虽然具有很多优良特性,但是结构复杂,要在 $O(n)$时间复杂度内构建它,还需要补充一些定义,并附加一些额外的数据结构.

定义 6. **Endpos** 集合:考虑字符串 T 的任意非空子串 t,若 t 在 T 中出现了 $k(k>0)$次,有 $t=T[l_1,r_1]=T[l_2,r_2]=\ldots=T[l_k,r_k]$,则定义 $Endpos(T,t)=\{r_1,r_2,\ldots,r_k\}$.

例 5:T="$bcabcaabc$",t="bc",$Endpos(T,t)=\{2,5,9\}$.

定义 7(状态(state)). 任取 T 的两个子串 s_1 和 s_2,定义 $State(T,s_1)=State(T,s_2)$当且仅当 $Endpos(T,s_1)=Endpos(T,$

s_2).即 T 中具有相同 *Endpos* 集合的子串构成自动机的一个状态.

定义 8(*Len*(*v*)). 一个状态 v 所能识别的最长子串长度.

定义 9(后缀链接 *Suffix_link*(*v*)). 给定 SAM 上的一个状态 v,定义状态 w=*Suffix_link*(v),若 w 满足 *Endpos*(w)是包含 *Endpos*(v)的最小集合.

定义 10(*Suffix_Tree*(*S*)树). 由所有 *Suffix_link* 组成的一棵以状态 S 为根的树,这棵树事实上是所有 *Endpos* 集合的树状包含关系.

当前,SAM 流行的构建算法是采用增量法构造,即每次添加一个字符,更新当前的 SAM,使得它成为包含这个新字符的新 SAM.在构建过程中需要两个指针:*last* 指向上一次插入的结点位置,*cur* 指向当前结点.具体的构造过程如下.

1. 首先建立一个空结点,表示空串的状态 v,其 *Len*(v)=0,*Suffix_link*(v)=*NULL*;
2. 假设已经构建了前 i–1 个字符的后缀自动机,现在考虑加入第 i 个字符 x.首先新建一个节点 *NewState*,令 *cur* 指向该节点,从 *last* 向 *cur* 连接一条标识为 x 的出边,然后令 *cur.len*=*last.len*+1;
3. 接下来需要在 Parent 树上沿着 *last* 的 *Suffix_link* 指针向根节点跳转,回溯途中:若当前节点不存在标识为 x 的出边,则从当前节点向 *cur* 连接一条标识为 x 的出边;否则,终止回溯.终止跳转时会遇到以下几种情况:

   Case 1:如果后溯到了 *NULL*,说明当前 SAM 中没有 x 这个字符,那么直接把 *cur* 的 *Suffix_link* 指针指向 *root*;

   Case 2:如果沿着 *Suffix_link* 链后溯到了一个节点 p,存在 *Trans*(p,x)=q,则又分以下两种情形处理:

   (1) 若 *q.len*=*p.len*+1,则 *Suffix_link*(*cur*)=q;

   (2) 若 *q.len*>*p.len*+1,则执行以下操作:

   a. 创建一个新的节点 *CloneState*,令 *nq* 指向该结点;

   b. 拷贝 q 的所有出边到 *nq*;

   c. 修改后缀链接:*Suffix_link*(*cur*)=*Suffix_link*(q)=*nq*;

   d. 把 p 以及所有 p 的祖先中到 q 的转移全部替换成 *nq*.

例 6:构造字符串“*bcabcaabc*”的后缀自动机.

(1) 假设字符串“*bc*”的后缀自动机已经构建(如图 2 所示),图中实线箭头表示转移边,而虚线箭头则表示后缀链接.下面要构建“*bca*”的自动机,因新添加的字符“*a*”没有在 SAM 中,满足 Case 1 的条件,则构造的新自动机如图 3 所示.

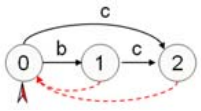


Fig.2 Suffix automaton of string “*bc*”

图 2 字符串“*bc*”的后缀自动机

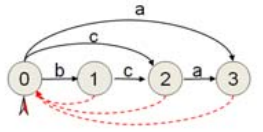


Fig.3 Suffix automaton of string “*bca*”

图 3 字符串“*bca*”的后缀自动机

(2) 继续构造字符串“*bcab*”的后缀自动机.构造新节点 S_4,沿着 *last* 节点 S_3 的 *Suffix_link* 链回溯到节点 S_0,存在 *Trans*(S_0,b)=S_1,而且满足 S_1.*len*=S_0.*len*+1,此时按 Case 2 的方法(1)操作,令 *Suffix_link*(S_4)=S_1.新构造的自动机如图 4 所示.

(3) 继续构造字符串“*bcabca*”的后缀自动机如图 5 所示,现在要构造“*bcabcaa*”的后缀自动机.建立新节点 S_7,沿着 *last* 节点 S_6 的 *Suffix_link* 链回溯到节点 S_0,存在 *Trans*(S_0,a)=S_3,但是 S_3.*len*>S_0.*len*+1,此时按 Case 2 的方法(2)操作,创建一个新的节点 S_8,将节点 S_3 的状态信息克隆到 S_8.新生成的自动机如图 6 所示.

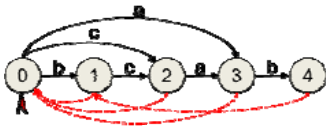


Fig.4 Suffix automaton of string "*bcab*"

图 4 字符串"*bcab*"的后缀自动机

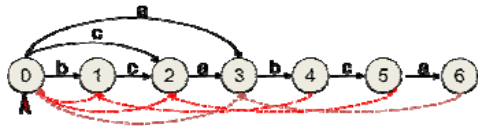


Fig.5 Suffix automaton of string "*bcabca*"

图 5 字符串"*bcabca*"的后缀自动机

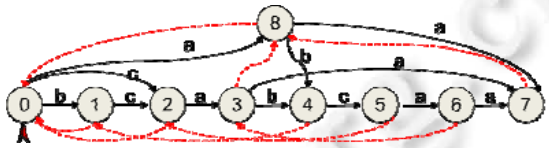


Fig.6 Suffix automaton of string "*bcabcaa*"

图 6 字符串"*bcabcaa*"的后缀自动机

完整创建的 SAM 如图 7 所示.从创建过程可以发现:在 SAM 上每次增加一个字符,新 SAM 上最多增加两个节点,因此对应长度为 n 的字符串,其 SAM 的状态数最多为 $2n$ 个,仅需要 $O(n)$的空间.并且根据文献[34],它能在 $O(n)$时间内被构造(如果我们将字母表的大小视作常数,否则是 $O(n\times\log|\Sigma|)$).

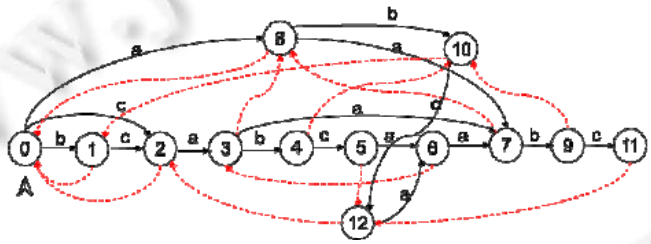


Fig.7 Suffix automaton with suffix links for string "*bcabcaabc*"

图 7 "*bcabcaabc*"带有后缀链接的后缀自动机

3.2 W-SAM自动机模型

以 *SAM*(*T*)作为文本串 *T* 的索引,给定查询串 *t*,我们只需要从 *root* 出发,沿 *t* 个字符的转移边进行遍历,遍历完成若能成功到达 SAM 的某个状态 *s*,即可确认 *t* 是 *T* 的子串;而若需要获取 *t* 在 *T* 上的所有出现位置,则需要返回状态 *s* 的 *Endpos* 集合.现在问题的关键是如何获取某个状态 *s* 的 *Endpos* 集合.一种朴素的做法是在每个节点上设置一个用来存储 *Endpos* 的容器(例如数组),在创建的过程中,每次更新 *Suffix_link* 时,同步更新相应节点的 *Endpos*,但该方法会使每个节点的存储空间膨胀,整个 SAM 的空间复杂度也将增大到 $O(n^2)$.为了既能满足 *Endpos* 的查询需求,又能有效压缩 SAM 的空间大小,本文对传统 SAM 数据结构及构造算法进行了改进,提出 W-SAM 模型.

W-SAM 的构造算法与 SAM 基本一致,仍然采取增量法构造,其不同点如下.

令当前字符串为 *T*,当前自动机为 *W-SAM*(*T*),现在新字符为 *x*,需要构造新字符串 *Tx* 的自动机,即 *W-SAM*(*T*)→*W-SAM*(*Tx*),对前述 SAM 的构造方法作如下更改.

1. 对每个节点增加一个 *Position* 标记,标识该状态最长子串在文本串中首次出现的位置,在增量更新自动机时,每增加一个状态:

   (1) 若该状态是新增的节点 *NewState*,令 *cur* 指向该节点,其 *Position* 为

   $$cur.Postion=last.Postion+1;$$

   (2) 若该状态是由某个状态 *q* 分裂而来的复制节点 *CloneState*,令 *nq* 指向该结点,其 *Position* 为

   $$nq.Postion=-|q.Postion|;$$

2. 对每个结点增加两个指针 *ptrOldestSon*,*ptrBrother*,分别指向该结点在 *Suffix_Tree* 逆序树上的最大儿子及其一个兄弟结点.初始默认设置为 *NULL*,每增加一个状态 *s* 时,若 *Suffix_link*(s)=*NULL*,则令 *Suffix_link*(*s*).*ptrOldestSon*=*s*;否则,令上次添加的兄长结点(假设为 *lastBrother*)指向自己,即令:

lastBrother.*ptrBrother*=*s*.

文本串"*bcabcaabc*"的 W-SAM 结构如图 8 所示,图中圆点虚线箭头表示 *ptrOldestSon*,而短划线虚线箭头则表示 *ptrBrother*,圆圈中的数字表示该结点的 *Postion* 值.

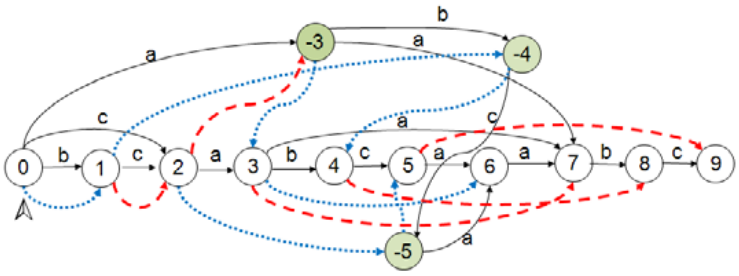


Fig.8 Wildcard suffix automaton of string "*bcabcaabc*"

图 8 "*bcabcaabc*"的 W-SAM 模型

相比 SAM 模型,W-SAM 模型每个结点仅增加了 3 个成员,分别为 1 个整型变量和两个指针,因此所增加的空间代价并不显著.从时间代价上考虑,仅增加了 3 条赋值语句,也未显著增加创建 W-SAM 的时间复杂度.构建 W-SAM 的时间和空间复杂度依然为 $O(n)$.

定理 2. 任取状态 $S\in$*W-SAM*,则 *Endpos*(*S*)={*st*.*Position*|*st*∈*Suffix_Tree*(*S*)∧*st*.*Position*>0}.即:任取自动机上一状态 *S*,则该状态的 *Endpos* 集合等于以该状态为根结点的 *Suffix_Tree* 树上所有 *Position*>0 的结点的 *Position* 的并集.

证明:根据 W-SAM 的构造算法,每次增量更新 W-SAM 时,最多会增加两个结点 *NewState* 和 *CloneState*,其对应的 *Position* 分别大于 0 和小于 0.而只有 *NewState* 才对应新扩充的最长字符串,可以产生新的 *Endpos* 元素;*CloneState* 状态仅用于分裂却并不能产生新的 *Endpos* 元素.根据定义,*Position* 正好是结点 *NewState* 最长子串在文本上首次出现位置,于是便有 *Endpos*(*NewState*)={*NewState*.*Position*}.同样根据定义 9,*Suffix_Tree*(*S*)包含了以 *S* 为根结点的树上所有结点的 *Endpos* 集合,因此,合并所有这些结点的 *Endpos* 即为根结点 *S* 的 *Endpos* 集合.

证毕. □

4 带间隙约束的模式匹配算法

4.1 SGPM-SAI算法

设有文本 *T* 及带间隙约束的模式串 *P*.SGPM-SAI 算法的主要思想是:首先建立文本串 *T* 的 *W-SAM*(*T*),然后将模式串 *P* 以间隙为界划分为若干小的查询片段 $\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}$,依次查询每个片段 β_j 在 *W-SAM*(*T*)上的 *Endpos* 集合.接下来计算所有间隙的笛卡尔积 *D*,对每项 D_i,按照各个片段的先后顺序对 *Endpos* 进行集合交运算,这样归并的最终集合即是所有匹配的位置.详细匹配过程的伪代码见算法 1.

算法 1. 带间隙约束的模式匹配算法.

输入:文本串 *T*,模式串 $P=\beta_0\Psi_0[a_0,b_0]\beta_1\ldots\beta_j\Psi_j[a_j,b_j]\beta_{j+1}\ldots\Psi_{m-2}[a_{m-2},b_{m-2}]\beta_{m-1}$;

输出:*P* 在 *T* 上的所有出现.

1. building *W-SAM*(*T*)

2. $\{\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}\}\leftarrow$*SplitPattern*(*P*)

3. **for each** $\beta_j\in\{\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}\}$

4. State S_j←*Trans*(*root*,β_j)
5. **if** (S_j==*NULL*) **then return** *NULL*
6. *Endpos*(β_j)←*GetEndpos*(S_j)
7. **end for**
8. $cnt \leftarrow \prod_{j=0}^{m-2}(b_j - a_j + 1)$
9. **for** *i*=0 to *cnt* step 1
10. D_i←*GetDecare*({$\Psi_0[a_0,b_0], \Psi_1[a_1,b_1],\ldots, \Psi_j[a_j,b_j],\ldots, \Psi_{m-2}[a_{m-2},b_{m-2}]$},*i*)
11. *M*←*Endpos*(β_0)
12. **for** *j*=1 to *m*−1 step 1
13. **for** *t*=0 to *M.size* step 1
14. *M*[*t*]←*M*[*t*]+D_i[*j*−1]+|β_j|
15. **end for**
16. *tmp*←*GetIntersect*(*Endpos*(β_j),*M*)
17. *free*(*M*)
18. **if** (*tmp*==∅) **then**
19. *free*(*tmp*)
20. *M*←∅
21. **break**
22. **end if**
23. *M*←*tmp*
24. **end for**
25. **if** (*M*≠∅) **then**
26. $patternLength \leftarrow \sum_{j=0}^{m-2} D_i[j] + \sum_{j=0}^{m-1} |\beta_j|$
27. **for each** *pos* in *M*
28. *output*(⟨*pos-patternLength*+1,*pos*⟩)
29. **end for**
30. **end if**
31. **end for**

- 上述代码第 1 行是建立文本串 *T* 的后缀自动机索引 *W-SAM*(*T*);
- 第 2 行将模式串 *P* 以间隙为界划分为若干小的查询片段$\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}$;
- 第 3 行~第 7 行查询每个片段β_j在 *W-SAM*(*T*)上的 *Endpos* 集合,此间若出现某个片段的 *Endpos* 集合为空,则显然,整个模式串 *P* 不可能在文本 *T* 中匹配成功,返回为空;
- 第 8 行计算模式串 *P* 的定长子模式集合的大小;
- 从第 9 行开始,对所有定长子模式进行匹配;
- 第 10 行调用 *GetDecare* 函数计算模式串中所有间隙区间的笛卡尔积,例如,例 1 中的模式串 *baa*[2,3]*c*[0,2]*ac*,调用 *GetDecare*({[2,3],[0,2]})函数可得 *D*={{2,0},{2,1},{2,2},{3,0},{3,1},{3,2}}.其中,D_4={3,1},D_4[0]=3,D_4[1]=1;
- 对每项笛卡尔积 D_i,第 11 行~第 24 行将各个片段的 *Endpos* 按其在模式串中的出现顺序进行集合的交操作,集合交操作按以下递推式进行迭代并存储在集合 *M* 中:

$$\begin{cases} M_0 = Endpos(\beta_0) \\ M_j = Endpos(\beta_j) \cap \{pos \mid r \in M_{j-1} \wedge pos = r + D_i[j-1] + |\beta_j|\}, 1 \leqslant j \leqslant m-1 \end{cases};$$

- 第 24 行~第 29 行判断归并结果 M 是否为空:若不为空,则计算卡氏积 D_i 所对应的定长子模式串的长度,并输出该子模式在文本串 T 上所有出现的起止位置.

4.2 算法分析

首先分析算法的空间复杂度,算法 1 中主要涉及的数据结构有:1) 文本串 T 的后缀自动机 *W-SAM*(T);2) 数组 D,用来暂存所有间隙区间的笛卡尔积;3) 数组 M,用来暂存各个片段 *Endpos* 的交集.

- 对于数据结构 1),设文本 T 的长度为 n,由第 3.2 节 W-SAM 的构建算法可知,其空间复杂度为 $O(n)$;
- 对于数据结构 2),由定理 1 可知,若模式串为 $P=\beta_0\Psi_0[a_0,b_0]\beta_1\ldots\beta_j\Psi_j[a_j,b_j]\beta_{j+1}\ldots\Psi_{m-2}[a_{m-2},b_{m-2}]\beta_{m-1}$,则笛卡尔积的大小 $\prod_{j=0}^{m-2}(b_j-a_j+1)$,但对卡氏积的每项元素我们只需使用一次,因此可重复使用相同的数组进行存储,其空间复杂度为 $O(m)$;
- 对于数据结构 3),数组 M 的大小取决于 *Endpos*(β_0)的大小,显然 *Endpos*(β_0)≤n,因此 M 的空间复杂度为 $O(n)$.

综上所述,因为通常 $n>>m$,故我们算法的空间复杂度为 $O(n)$.

接下来分析算法的时间复杂度.算法 1 的主要时间开销发生在:1) 第 1 行 building W-SAM;2) 循环第 3 行~第 7 行;3) 循环第 9 行~第 31 行.

- 对于情形 1),设文本 T 的长度为 n,由第 3.2 节 W-SAM 的构建算法可知,其时间复杂度为 $O(n)$;
- 对于情形 2),每个分段 β_j 的查询时间为 $O(|\beta_j|)$,故总的查询时间为 $O(length(P))$,*length*(P)表示模式串的长度(不含间隙);
- 对于 3),循环次数取决于 $\prod_{j=0}^{m-2}(b_j-a_j+1)$ 的大小,考虑到我们还要输出所有的模式出现(循环第 27 行~第 29 行),故算法的最坏时间复杂度为 $O(n\times\max\{b_j-a_j+1\}^m)$.

综上分析,为了在可接受的时间代价内使问题求解,我们需要对 P 的定长子模式集合大小设置一个上限阈值,即 $\max\{b_j-a_j+1\}$和 m 均要小于某一个常数,这就是我们在前文所述间隙的稀疏性.

5 SGPM-SAI 算法的并行化

随着硬件技术的进步,多核甚至众核已经成为现代处理器的主流配置.为更好适应现代处理器的高度并行处理能力,以提高算法性能,我们在算法 1 的基础上提出了并行化方案.

考察算法 1 中代码第 3 行~第 7 行的循环,该循环依次获取各个模式串分段的 *Endpos* 集合,由于各个分段可独立地在自动机上查询,因此可以并行获取 *Endpos* 集合.假定输入线程数目为 ns,算法 2 描述了并行获取各个分段 *Endpos* 集合的详细过程.

算法 2. *Endpos* 集合的并行获取算法.

输入:文本串 T,模式串 P,线程数 ns;

输出:各个分段的 *Endpos* 集合.

1. $\{\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}\}\leftarrow SplitPattern(P)$
2. **if** $ns>m$ **then** $ns\leftarrow m$
3. $\{G_0:\{\beta_0,\ldots,\beta_k\},G_1:\{\beta_{k+1},\ldots,\beta_{2k}\},\ldots,G_{ns}:\{\beta_{ns\cdot k+1},\ldots,\beta_{m-1}\}\}\leftarrow GroupByThreads(P)$
4. **for each** thread t **where** $1\leqslant t\leqslant ns$ parallel **do**
5. **for each** $\beta_j\in G_t$
6. State $S_j\leftarrow Trans(root,\beta_j)$
7. **if** (S_j==*NULL*) **then return** *NULL*
8. $Endpos(\beta_j)\leftarrow GetEndpos(S_j)$
9. **end for**
10. **end** parallel **do**

- 上述伪代码的第 1 行将模式串 P 以间隙为界划分为若干小的查询片段$\beta_0,\beta_1,\ldots,\beta_j,\ldots,\beta_{m-1}$;
- 第 2 行、第 3 行则按照给定线程数目对各个分段进行分组,每个分组大小 $k=m/ns$(最后一个分组可能例外),即将总的分段数 m 按照线程数 ns 进行平摊;
- 对于每一个分组,安排一个线程获取该分组内各个分段的 *Endpos* 集合(第 4 行~第 10 行).

算法 1 中,代码第 8 行~第 31 行用于对各个片段的 *Endpos* 集合按其在模式串中的顺序进行归并,求这些集合的交集.考察这一过程,其循环次数为 $\prod_{j=0}^{m-2}(b_j-a_j+1)$,显然,这是一个非常耗时的过程,需要充分并行化来提高算法性能,算法 3 的伪代码描述了实现这一过程的具体流程.

算法 3. *Endpos* 集合归并求交的并行化算法.

输入:各个分段的 *Endpos* 集合,线程数 ns;

输出:P 在 T 上的所有出现.

1. $k_{\max}$←Get max integeral k, where k satisfy $\prod_{j=0}^{k}(b_j-a_j+1)\leqslant ns$
2. $g\leftarrow\prod_{j=0}^{k_{\max}}(b_j-a_j+1)$
3. ns←g
4. **for each** thread t **where** $1\leqslant t\leqslant ns$ parallel **do**
5. $D_i\leftarrow GetDecare(\{\Psi_0[a_0,b_0],\Psi_1[a_1,b_1],...,\Psi_{k_{\max}}[a_{k_{\max}},b_{k_{\max}}]\},t)$
6. M←$Endpos(\beta_0)$
7. **for** j=1 to $k_{\max}$ step 1
8. **for** t=0 to $M.size$ step 1
9. $M[t]$←$M[t]+D_t[j-1]+|\beta_j|$
10. **end for**
11. tmp←$GetIntersect(Endpos(\beta_j),M)$
12. $free(M)$
13. **if** (tmp==∅) **then**
14. $free(tmp)$
15. M←∅
16. **break**
17. **end if**
18. M←tmp
19. **end for**
20. **if** (M==∅) **then return** *NULL*
21. $Len\_pre\leftarrow\sum_{j=0}^{k_{\max}}D_i[j]$
22. $Left\_cnt\leftarrow\prod_{j=k_{\max}+1}^{m-2}(b_j-a_j+1)$
23. **for** i=0 to $Left\_cnt$ step 1
24. $D_i\leftarrow GetDecare(\{\Psi_{k_{\max}+1}[a_{k_{\max}+1},b_{k_{\max}+1}],...,\Psi_{m-2}[a_{m-2},b_{m-2}]\},i)$
25. M_L←M
26. **for** j=$k_{\max}$+1 to m−1 step 1
27. **for** t=0 to $M\_L.size$ step 1
28. $M\_L[t]$←$M\_L[t]+D_i[j-1]+|\beta_j|$
29. **end for**
30. tmp←$GetIntersect(Endpos(\beta_j),M\_L)$

31. *free*(*M_L*)
32. **if** (*tmp*==∅) **then**
33. *free*(*tmp*)
34. *M*←∅
35. **break**
36. **end if**
37. *M_L*←*tmp*
38. **end for**
39. if (*M_L*≠∅) **then**
40. $patternLength \leftarrow Len\_pre + \sum_{j=0}^{m-2} D_i[j] + \sum_{j=0}^{m-1} |\beta_j|$
41. **for each** *pos* in *M_L*
42. *output*(⟨*pos*-*patternLength*+1,*pos*⟩)
43. **end for**
44. **end if**
45. **end for**
46. **end** parallel **do**

上述代码依据给定的线程数目将模式 P 拆分为若干子模式,每个子模式由一个线程独立完成匹配任务.为了均衡每个线程的任务,使每个子模式的间隙可变度相等,我们采用以下方式对模式 P 进行拆分.

1) 求满足 $\prod_{j=0}^{k}(b_j - a_j + 1) \leqslant ns$ 的最大整数,设为 $k_{\max}$;
2) 设拆分后的子模式数目为 g,令 $g = \prod_{j=0}^{k_{\max}}(b_j - a_j + 1)$;
3) 以 $k_{\max}$ 为分界点,将模式串 P 的间隙分为两部分,计算 $k_{\max}$ 之前(含 $k_{\max}$)间隙的笛卡尔积;
4) 将第 3)步笛卡尔积中的每一项与 $k_{\max}$ 之后的间隙合并,重新构造一个模式.

例 7:有模式 *P*=*baa*[2,3]*c*[0,2]*ac*[1,3]*bc*,输入的线程数目 *ns*=7,求 *P* 拆分后的子模式集合.

- 根据步骤 1),$k_{\max}$=1.这是因为(3−2+1)×(2−0+1)=6<7,而(3−2+1)×(2−0+1)×(3−1+1)=18>7,满足上述不等式只能取前 2 项,即第 0 项(3−2+1)和第 1 项(2−0+1),分界点为 1;
- 根据步骤 2),子模式数 *g*=(3−2+1)×(2−0+1)=6;
- 根据步骤 3),计算 $k_{\max}$ 之前间隙的笛卡尔积,有:

  GetDecare({[2,3],[0,2]})={⟨2,0⟩,⟨2,1⟩,⟨2,2⟩,⟨3,0⟩,⟨3,1⟩,⟨3,2⟩};
- 根据步骤 4),取出步骤 3)中每一项笛卡尔积,与 $k_{\max}$ 之后的间隙合并,可重新构造为以下 6 个子模式:

  {*baa*[2,2]*c*[0,0]*ac*[1,3]*bc*,*baa*[2,2]*c*[1,1]*ac*[1,3]*bc*,*baa*[2,2]*c*[2,2]*ac*[1,3]*bc*,
  baa[3,3]*c*[0,0]*ac*[1,3]*bc*,*baa*[3,3]*c*[1,1]*ac*[1,3]*bc*,*baa*[3,3]*c*[2,2]*ac*[1,3]*bc*}.

由于按照上述方式划分后的子模式,其匹配操作是完全独立的,因此可由多个线程并行完成,从代码第 4 行开始的循环详细描述了每个线程的匹配流程,其处理过程与算法 1 类似.

6 实验结果及分析

6.1 实验设置与环境

i. 实验平台

我们实验所采用的服务器为 PowerEdge R730,配置有 2 颗 Intel Xeon E5-2650 v3 CPU 以及 512G DDR3 内存.每颗CPU 拥有 10 个处理核,支持 20 个物理线程.实验采用的操作系统为 CentOS Linux(kernel version 2.6.32),编译器采用 Intel icc 15(O3 优化),以 Pthreads 方式设计并行程序,核间线程分配方式使用 scatter.

ii. 工作负载

我们所采用的数据集主要有两类数据集:一类是随机产生的数据集,另一类是真实 DNA 序列数据.随机数据集的产生方法是:

1) 给定间隙数目 *gap_cnt*,最大间距 *MaxGapVal*,模式串长度 *Len_Pat*,文本串长度 *Len_T*;
2) 随机产生长度为 *Len_Pat* 的字符串 *P*,在该串中,随机选取 *gap_cnt* 个不重复的位置,在每个位置插入[*a*,*b*],*a*,*b* 为随机产生的整数,且保证 $0 \leqslant a \leqslant b \leqslant MaxGapVal$;
3) 以 *P* 为模板,构造 *k* 个定长子串 $P_1,P_2,\ldots,P_i,\ldots,P_k$.每个子串 P_i 的构造方法为:对 *P* 中的每个间隙[*a*,*b*]用 *x* 个随机字符替换,其中,$a \leqslant x \leqslant b$;
4) 随机产生长度为 *Len_T* 的字符串 *T*,在 *T* 中依次选取 *k* 个不重叠的位置 $pos_1,pos_2,\ldots,pos_k$,然后从每个位置 Pos_i 开始,用 P_i 替换$|P_i|$个连续的字符,则 *k* 即是模式在 *T* 中出现的次数,而$\langle pos_i,pos_i+|P_i|-1\rangle$则是模式出现的位置.

为了验证在真实数据集上的表现,我们也选取了美国国家生物中心网站(http://www.ncbi.nlm.nih.gov/-genomes)提供的真实 DNA 序列数据作为文本串 *T*,并从文本串中随机抽取一个片段转变为模式串 *P*.

iii. 测试基准

为验证本文所提出的算法性能,我们采用 3 类典型的模式匹配算法作为基准比较方法进行了对比,分别是:

1) 单模式匹配算法 KMP[18],BM[19];
2) 多模式匹配算法 AC 自动机[21];
3) 基于文本索引的 Suffix Array[27]匹配算法.

同时,我们也用真实数据集与经典匹配算法 SAIL 的优化版本 SAIL-Gen[8]进行了实验对比(SAIL-Gen 源代码来源于 http://wuc.scse.hebut.edu.cn/nettree/dcnp-2).

6.2 W-SAM与Suffix Array的比较

传统的单文本串索引中,后缀树与后缀数组是最常使用的两种数据结构.相比后缀树,后缀数组有更小的空间需求.因此,为了考察 W-SAM 的空间占用情况,我们以 Suffix Array 作为参照的标准,对 W-SAM 所占空间大小随文本串长度的变化进行了测试,结果如图 9 所示.图中横轴表示文本串的长度,纵轴表示索引所占的内存空间大小.

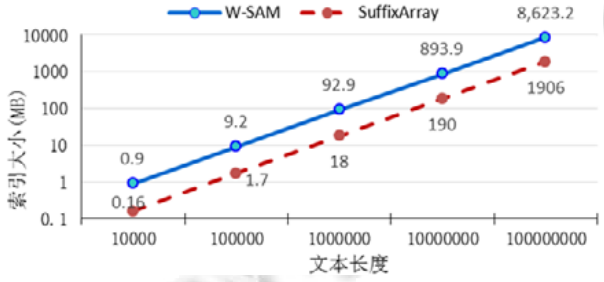


Fig.9 Comparison of index size between suffix array and W-SAM

图 9 Suffix Array 与 W-SAM 索引空间大小比较

从图 9 可见:随着文本串长度增大,后缀数组与 W-SAM 都呈线性增长的趋势.相比后缀数组,W-SAM 的索引大小要多出 4 倍~5 倍.这是因为后缀数组以最简压缩形式存储了文本串的所有后缀,而 W-SAM 是以图的方式进行存储,除了基本状态外,还有状态间大量的链接指针需要存储.

图 10 是创建两种索引的时间对比图,图中横轴表示文本串的大小,纵轴表示创建索引的时间(单位:秒).从图 10 可见:随着文本串长度增大,创建后缀数组与 W-SAM 所需时间都呈线性增长.相比后缀数组,创建 W-SAM 的时间要多出 1 倍~2 倍.这是由于创建 W-SAM,除了要构造 DAG 图外,还需要同步维护后缀链接树以及 ptrOldestSon 和 ptrBrother 指针,因此相比后缀数组,需要耗费更多的时间.

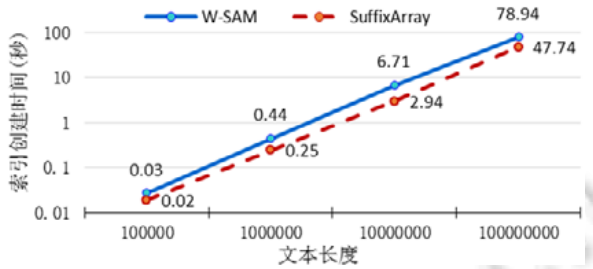


Fig.10 Comparison of building time between suffix array and W-SAM

图 10 Suffix Array 与 W-SAM 创建时间比较

综合图 9、图 10 可见,构建 W-SAM 的时间和空间代价都比 Suffix Array 要大.但这种差距仅维持在数倍,在现代大容量内存平台下,仍属可以接受的范围.且从下文的实验结果将可以发现:在匹配阶段,基于 W-SAM 的匹配算法至少要比 Suffix Array 快 5 倍以上,某些情况下甚至可以达到数量级的优势,因此面向稳定文本的某些特殊应用领域,如 OLAP 中历史数据查询,这种代价是值得的.此外,从构建算法可知:相比后缀数组的离线方式构造,W-SAM 可以在线方式增量构造,这对于流数据处理极具优势.

6.3 固定间隙的模式匹配算法性能比较

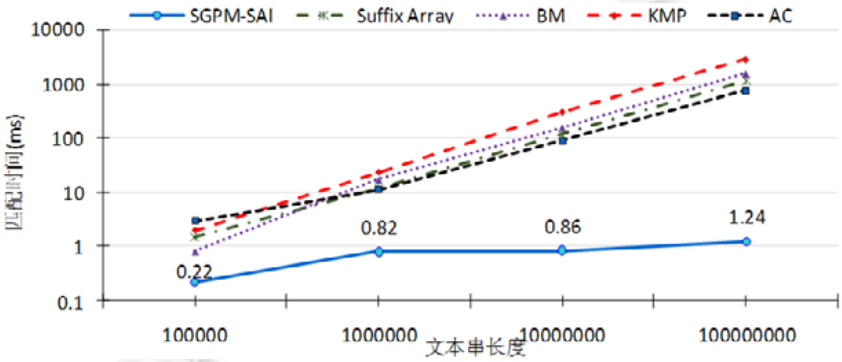
带固定间隙(间隙的上界与下界相等)的模式匹配是研究可变间隙模式匹配的基础,因此,我们首先测试了定长模式匹配的性能.我们采取的做法是:以间隙为界,将模式串分隔为若干片段,然后按片段的位置次序逐段匹配,则最后一个片段的匹配位置即为查询结果.求解此类问题的算法有多种,为检验 SGPM-SAI 算法的性能,我们选取以下典型的算法进行了比较,分别是 KMP、BM、AC 自动机和 Suffix Array.影响定长模式匹配时间的主要因素有:

(1) 文本串的长度;

(2) 模式串长度;

(3) 间隙数目.

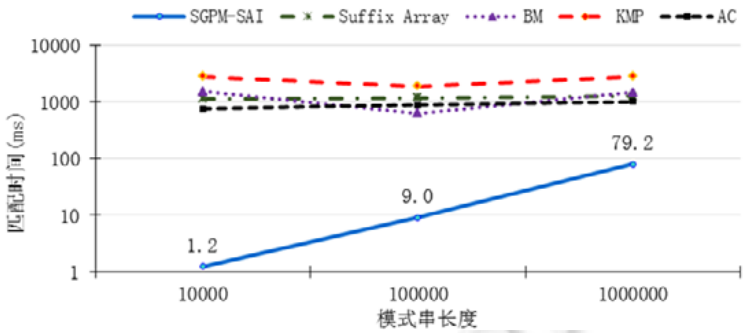
为此,我们设计了 3 组对比实验,分别如图 11~图 13 所示.需要说明的是:对于 Suffix Array 和 SGPM-SAI,以下实验中,我们未将创建文本串索引的预处理时间计算在内.



(模式串长度:10 000,间隙数目:10)

Fig.11 Impact of text length on pattern matching time with fixed gaps condition

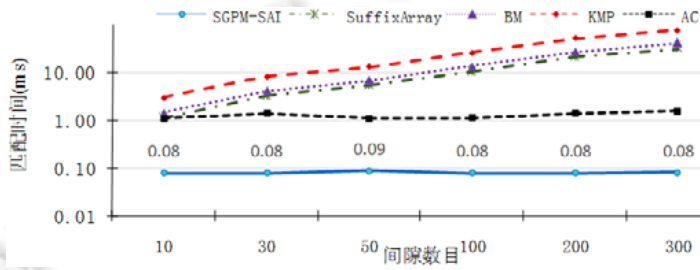
图 11 定长模式匹配时间受文本长度变化的影响



(文本串长度:100 000 000,间隙数目:10)

Fig.12 Impact of pattern length on pattern matching time with fixed text length

图 12 定长文本的模式匹配时间受模式串长度变化的影响



(文本串长度:100 000 000,模式串长度:1 000 000)

Fig.13 Impact of the number of gaps on matching time with fixed text length

图 13 定长模式的模式匹配时间受间隙数目变化的影响

由图 11 可知,当文本串长度>1000000 时:

- 从绝对时间上来看:SGPM-SAI 算法的性能最好,AC 算法其次,KMP 性能最差,SGPM-SAI 的性能优于其他算法 1~3 个数量级;
- 从增长趋势来看:其他算法的匹配时间都随文本串长度的增大而显著增加,唯有 SGPM-SAI 却几乎维持不变.

从时间复杂度上分析,我们不难发现原因.假设文本串长度为 n,模式串长度为 m,分段数目为 k,平均每个分段长度为 m/k,表 1 显示了定长模式下的各种算法的平均时间复杂度.

Table 1 Time complexities of pattern matching algorithms with fixed gaps condition

表 1 定长模式下的模式匹配算法时间复杂度

算法名称	时间复杂度
KMP	$O(kn+m)$
BM	$O(kn/(m+k))$
AC	$O(m+n)$
Suffix Array	$O(m\log n)$
SGPM-SAI	$O(k\times O(m/k))$

从表 1 各算法时间复杂度不难发现:在所有匹配算法中,除了 SGPM-SAI 与 n 无关外,其余算法皆与 n 呈正相关关系.因此,随着文本串长度 n 的增大,SGPM-SAI 基本保持不变,而其他算法的匹配时间皆显著增长.

由图 12 可知:

- 从绝对时间上来看:SGPM-SAI 算法的性能最好,其性能优于其他算法 1~3 个数量级;

- 从增长趋势来看:随着模式串长度增加,SGPM-SAI 的匹配时间显著增加,而其他算法的匹配时间变化较小.

从表 1 的分析可知:SGPM-SAI 只与模式串长度有关,因此模式串长度的变化会显著影响匹配时间;而其他算法与模式串长度 m 与文本串长度 n 都相关,但一般而言,$m<<n$,因此决定匹配时间的主要因素是 n 而不是 m,故 m 的变化对整体匹配时间的影响较小.

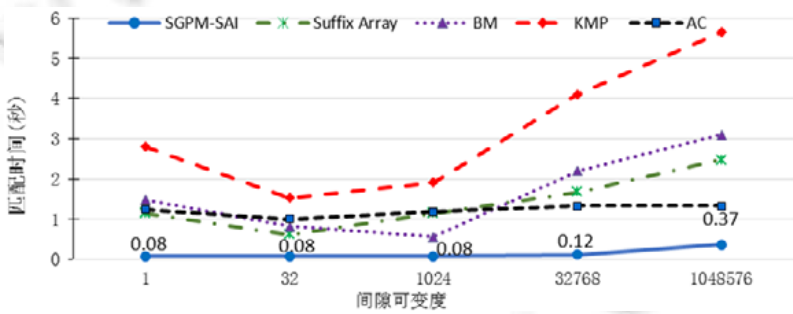
由图 13 可知:

- 从绝对时间上来看:SGPM-SAI 的性能最好,AC 算法其次,KMP 性能最差,SGPM-SAI 的性能平均优于 AC 一个数量级左右;
- 从增长趋势来看:随着间隙数目增加,模式串的分段数量 k 也增多,在此情形下,SGPM-SAI 与 AC 算法的匹配时间变化较小,而其他算法的匹配时间则增长显著.这是因为对于单模式算法 KMP,BM,分段数目越多,遍历文本串的次数就越多;而 AC 自动机是一个多模式算法,每个分段作为一个模式,不会出现遍历文本串的次数随着分段数目增多而增加的情况.因此,KMP 和 BM 的模式匹配时间会因间隙数目增加而增长,而 AC 自动机却基本保持不变.

从表 1 可知,基于后缀自动机的算法,其时间复杂度为 $O(k\times O(m/k))\approx O(m)$.分段数目 k 对 SGPM-SAI 算法基本无影响.

6.4 可变间隙约束的模式匹配算法性能比较

为研究可变间隙约束下各种算法模式匹配的性能,我们对模式串构造 k 个间隙,并对每个间隙的间距进行配置,使得这些间隙可变度分别等于 1,32,1 024,32 768 和 1 048 576,相应的实验结果如图 14 所示.



(文本串长度:100 000 000,模式串长度:1 000 000)

Fig.14 Impact of the variation range of gaps on pattern matching time

图 14 间隙可变度对模式匹配时间的影响

从图 14 可知:

- 从绝对时间上来看:SGPM-SAI 的性能最好,KMP 的性能最差.相比其他次优算法,SGPM-SAI 的性能至少提高 3 倍~5 倍;
- 从增长趋势来看:当间隙可变度较小时,除 SGPM-SAI 外,其他算法均出现小幅波动;当间隙可变度较大时(例如大于 1 024),随着可变度增大,各种算法的匹配时间均出现不同程度增长,其中,KMP,BM,SGPM-SAI 及 Suffix Array 增长显著,而 AC 算法增长较为和缓.这是因为在间隙可变度较小时,各分段的归并代价很小,其性能的主导因素在于各个分段的查找时间;但随着间隙可变度增加,算法 2 中第 9 行~第 31 行处的循环次数将大幅增长,使得片段序列的归并代价成为主导因素,匹配时间也将显著增长.相比较而言,因为多模式的 AC 自动机算法采用了不同的归并模式,因此匹配时间的增长相对平缓.

为进一步了解 SGPM-SAI 算法在稠密间隙约束及真实数据集下的表现,我们从美国国家生物中心网站下载了一个流感病毒真实 DNA 序列数据作为文本串 T(其长度为 1 000 000),并从文本串中随机抽取一个子串转

变为模式串 *P*.模式串 *P* 的构造方法如下.

1) 从文本串 *T* 随机抽取一个子串 *S*,例如 *aggagggagcaatagttggagaaatttcaccattacctt*;
2) 从 *S* 中随机选取 *k* 个位置 $p_1,p_2,\ldots,p_k$,从每个位置 p_i 开始删除 r_i 个字符,并插入[0,r_i–1].这相当于在 *S* 中随机插入 *k* 个间隙,每个间隙的间距为 r_i.例如向 *S* 中插入 3 个间隙,间距分别为 3,2,4,则一个可能的模式 *P* 为:*agga*[0,2]*agcaat*[0,1]*ttggagaaattt*[0,3]*attacctt*.

按照上述方法,由于 *k* 取不同值,我们产生一组稀疏度不同的模式串,其配置见表 2.

SAIL 是文献[9]提出的一种在线算法,具有良好的时间性能来解决满足 one-off 条件下的模式匹配问题.本文选取 SAIL 算法的一种优化算法 SAIL-Gen 作为基准比较方法进行了对比.比较结果见表 2.

Table 2 Impact of the strength of gaps constraint on SGPM-SAI VS. SAIL-Gen (s)

表 2 间隙约束稀疏度对 SGPM-SAI VS. SAIL-Gen 两种算法的影响 (s)

Gaps count	Gaps range	SGPM-SAI	SAIL-Gen
10	4	0.249	2.279
10	5	2.550	2.627
10	6	13.291	3.314
10	7	76.001	7.383
10	2	0.016	2.080
20	2	2.426	1.935
30	2	300.630	39.353

从表 2 可知:当间隙约束比较稀疏时,即间隙数目较少且间距较小时,SGPM-SAI 算法的性能要显著优于 SAIL-Gen 算法;但当间隙约束比较稠密时,即间隙数目较多或者间距较大时,SGPM-SAI 算法的匹配时间快速增长,其耗时显著高于 SAIL-Gen 算法.这是由于 SAIL-Gen 算法舍弃了大量的匹配信息,只能返回部分最优解,而 SGPM-SAI 算法却可以返回满足约束条件的完备解.

6.5 SGPM-SAI的并行加速效果

从前面的实验结果可以看出,基于间隙约束的模式匹配是一个非常耗时的过程,尤其是当间隙较为稠密时.为了验证算法的并行优化效果,我们采用不同数目的线程测试了算法 2(get endpos)和算法 3(merge endpos)两个阶段在不同间隙可变度下的运行时间.实验所采用的文本串 *T* 长度为 100 000 000,模式串 *P* 长度为 1 000 000,*gaps*=64,实验结果见表 3.

Table 3 Impact of the variation range of gaps on parallel pattern matching time (ms)

表 3 间隙可变度对并行模式匹配时间的影响 (ms)

Variation range	Phase	Number of threads					
		1	4	16	64	256	1 024
65 536	Get endpos	78.48	62.08	30.67	8.57	8.70	8.58
	Merge endpos	19.21	7.56	3.50	2.81	2.21	3.44
16 777 216	Get endpos	78.50	61.75	30.75	8.52	8.66	8.74
	Merge endpos	4 483.24	1 790.73	474.00	113.63	37.85	36.36
4 294 967 296	Get endpos	78.51	61.67	30.76	8.54	8.61	8.63
	Merge endpos	772 316.34	468 202.26	113 748.29	27 748.98	6 806.52	1 688.27

由表 3 可知:

- 从绝对时间来看:在间隙可变度较小时,获取 *Endpos* 集合的时间要高于归并这些集合的时间;而在间隙可变度较大时,*Endpos* 集合的归并时间远高于获取这些集合的时间;
- 从并行效果来看:两个阶段都获得了良好的加速,但 *Endpos* 集合归并阶段的加速比要高于获取阶段,尤其是当间隙可变度较大时.

图 15 描绘了 getEndpos 阶段的并行加速比曲线,从图中可以看出:从 1~64 线程均具有良好的加速效果,加速比最高可达 9.2 倍;但当 64 线程以后,增加线程数目并不能带来额外的性能收益,加速比基本维持不变.分析算法 2 不难发现,产生上述现象的原因在于:getEndpos 阶段的并行度取决于分段数目,由于我们限定了间隙数目为

64,而间隙数目与分段数目是一一对应的.

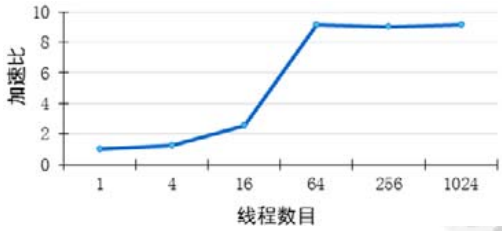


Fig.15 Variation of speedup with the number of threads for getEndpos phase

图 15 getEndpos 阶段的加速比随线程数目的变化

图 16 描绘了 mergeEndpos 阶段的并行加速比曲线,从图中可以看出:间隙可变度不同的 3 条曲线并不一致,可变度较小的曲线加速效果差,而可变度越大的曲线加速效果越好.例如:图中可变度为 65 536 的曲线,仅在 256 线程处取得 8.6 倍的加速,往后随着线程数目增加反而加速比下降;而可变度为 4 294 967 296 的曲线,其加速比始终随着线程数目增加而增加,在 1 024 线程处可达 455 倍,呈现出良好的并行加速效果.分析算法 3 不难发现,产生上述现象的原因在于:归并阶段的并行度取决于间隙笛卡尔积的大小,而笛卡尔积与间隙可变度是一一对应的.

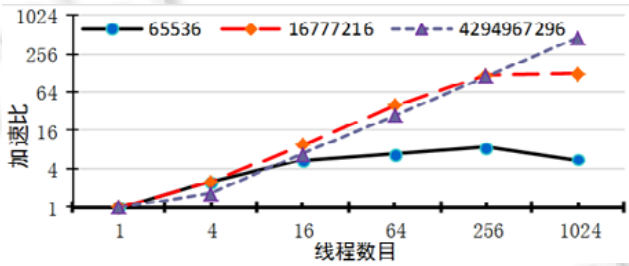


Fig.16 Variation of speedup with the number of threads for mergeEndpos phase

图 16 mergeEndpos 阶段的加速比随线程数目的变化

由表 4 可知:随着间隙可变度增大,带间隙约束的模式匹配时间快速增长.从前面的分析可知,这主要是因为可变度增大,相应的归并代价也增大,因此匹配时间快速增长.随着线程数目的增加,并行度扩大,模式匹配时间快速减少,这表明我们并行算法具有良好的并行可扩展性.

Table 4 Impact of the variation range of gaps and the number of threads on parallel pattern matching time (s)

表 4 间隙可变度及线程数目对并行模式匹配时间的影响 (s)

可变度	1	4	16	64	256	1 024
65 536	0.098	0.070	0.034	0.011	0.011	0.012
16 777 216	4.562	1.852	0.505	0.122	0.047	0.045
4 294 967 296	772.395	468.264	113.779	27.758	6.815	1.697

图 17 进一步描绘了 Parallel SGPM-SAI 算法的加速比随线程数目的变化趋势,由图可知:

- 在间隙可变度较小的情况下,Parallel SGPM-SAI 算法加速效果有限,如在可变度为 65 536 时,仅在 64 线程处取得 8.6 倍的加速;随后,随着线程数目增加,加速比反而呈下降趋势;
- 在间隙可变度较大时,Parallel SGPM-SAI 算法的加速效果显著,如在可变度为 4 294 967 296 时,算法的加速比始终随线程数目增长而增长.

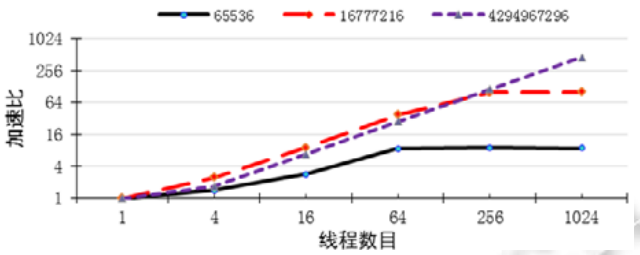


Fig.17 Variation of speedup with the number of threads for parallel SGPM-SAI

图 17 Parallel SGPM-SAI 算法的加速比随线程数目的变化

7 结论与讨论

带通配符的模式匹配是一个经典的研究问题,带有可变间隙约束的模式匹配则是近年来比较热门的研究方向.为高效求解带稀疏间隙约束条件下模式匹配的完备解,本文提出了一种基于图索引的匹配算法,即 SGPM-SAI 算法.该算法通过对文本串预处理,建立一种称为 W-SAM 的索引结构,然后采用我们所提出的模式匹配算法进行匹配,并返回匹配结果的完备解.通过对比实验,在不考虑预处理时间的情况下,对于固定间隙约束的模式匹配,相比几种现有的典型匹配算法,SGPM-SAI 算法性能高出 1~3 个数量级;而对于可变间隙约束的模式匹配,SGPM-SAI 算法性能则可高出 3~5 倍.SAIL 具有良好的时间性能来解决满足 one-off 条件下的模式匹配问题,本文选取 SAIL 算法的一种优化算法 SAIL-Gen 作为基准比较方法进行了对比,结果表明:当间隙约束比较稀疏时,SGPM-SAI 算法的性能要显著优于 SAIL-Gen 算法.为有效利用现代处理器的大规模并行处理能力,本文提出了 SGPM-SAI 算法的并行优化方案.实验结果表明:在间隙可变度较大时,Parallel SGPM-SAI 算法的加速效果显著,且具有良好的并行可扩展性,非常适宜利用现代处理器的大规模并行处理能力.

虽然我们的模式匹配算法相比经典算法具有更加优越的性能,但代价是算法需要较长的时间来对文本进行预处理,以及较大的内存空间来存储图结构的数据索引.因此,SGPM-SAI 算法比较适用于待匹配的文本较为稳定、而用来查找的模式串变动较为频繁的应用场景,如数据仓库 OLAP 中对历史数据的模糊查询.在下一步的研究工作中,如何进一步压缩创建文本索引的时间和空间代价,将是我们研究的重点.

致谢 本文部分工作是在作者访问中国人民大学的萨师煊大数据管理和分析中心时完成的,该中心获国家高等学校学科创新引智计划(111 计划)的资助.

References:

[1] Fischer MJ, Paterson MS. String-Matching and other products. In: Proc. of the 7th SIAM AMS Complexity of Computation. Cambridge, 1974. 113−125.

[2] Qiang JP, Xie F, Gao J, *et al*. Pattern matching with arbitrary-length wildcards. Acta Automatica Sinica, 2014,40(11):2499−2511 (in Chinese with English abstract). [doi: 10.3724/SP.J.1004.2014.02499]

[3] Clifford P, Clifford R. Simple deterministic wildcard matching. Information Processing Letters, 2007,101(2):53−54. [doi: 10.1016/j.ipl.2006.08.002]

[4] Cole R, Hariharan R. Verifying candidate matches in sparse and wildcard matching. In: Proc. of the Annual ACM Symp. on Theory of Computing. 2002. 592−601. [doi: 10.1145/509907.509992]

[5] Kalai A. Efficient pattern-matching with don't cares. In: Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms. Philadelphia, 2002. 655−656.

[6] Wu XD, Zhu XQ, He Y, Arslan AN. PMBC: Pattern mining from biological sequences with wildcard constraints. Computers in Biology and Medicine, 2013,43(5):481−492. [doi: 10.1016/j.compbiomed.2013.02.006]

[7] Sitaridi EA, Ross KA. GPU-Accelerated string matching for database applications. VLDB Journal, 2015. 1−22. [doi: 10.1007/s00778-015-0409-y]

[8] Xin C, Jia XF, Wu YX, *et al*. Strict pattern matching with general gaps and one-off condition. Ruan Jian Xue Bao/Journal of Software, 2015,26(5):1096−1112 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4707.htm [doi: 10.13328/j.cnki.jos.004707]

[9] Chen G, Wu XD, Zhu XQ, Arslan AN, He Y. Efficient string matching with wildcards and length constraints. Knowledge and Information Systems, 2006,10(4):399−419. [doi: 10.1007/s10115-006-0016-8]

[10] Wu YX, Liu YW, Guo L, Wu XD. Subnettrees for strict pattern matching with general gaps and length constraints. Ruan Jian Xue Bao/Journal of Software, 2013,24(5):915−932 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4381.htm [doi: 10.3724/SP.J.1001.2013.04381]

[11] Wu XD, Xie F, Qiang JP. Pattern Matching with Wildcards and Length Constraints. Beijing: Science Press, 2016.

[12] Bille P, G¨ortz IL, Vildh´oj HW, Wind DK. String matching with variable length gaps. Theoretical Computer Science, 2012,443(1): 25−34. [doi: 10.1016/j.tcs.2012.03.029]

[13] Wu YX, Shen C, Jiang H. Strict pattern matching under non-overlapping condition. Science China Information Sciences, 2017, 60(1): 1−16. [doi: 10.1007/s11432-015-0935-3]

[14] Ding BL, Lo D, Han JW, Khoo SC. Efficient mining of closed repetitive gapped subsequences from a sequence database. In: Proc. of the 25th Int'l Conf. on Data Engineering. Shanghai: IEEE, 2009. 1024−1035. [doi: 10.1109/ICDE.2009.104]

[15] Wang H, Wang HP, Wu XD. Models for pattern matching with wildcards and length constraints. Computer Science, 2016,43(4): 279−283 (in Chinese with English abstract).

[16] Manber U, Baeza-Yates R. An algorithm for string matching with a sequence of dont cares. Information Processing Letters, 1991, 37(3):133−136. [doi: 10.1016/0020-0190(91)90032-D]

[17] Navarro G, Raffinot M. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. Journal of Computational Biology, 2003,10(6):903−923. [doi: 10.1089/106652703322756140]

[18] Knuth DE, Jr Morris JH, Pratt VR. Fast pattern matching in strings. SIAM Journal on Computing, 1977,6(1):323−350. [doi: 10.1137/0206024]

[19] Boyer RS, Moore JS. A fast string searching algorithm. Communications of the ACM, 1977,20(10):762−772. [doi: 10.1145/359842.359859]

[20] Karp RM, Rabin MO. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 1987,31(2): 249−260. [doi: 10.1147/rd.312.0249]

[21] Aho AV, Corasick MJ. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 1975,18(6): 333−340. [doi: 10.1145/360825.360855]

[22] Wu S. A fast algorithm for multi-pattern searching. Technical Report, Report TR-94-17, Tucson: Department of Computer Science, University of Arizona, 1994.

[23] Bellekens X, Atkinson R, Andonovic I, *et al*. Investigation of GPU-based pattern matching. In: Proc. of the Post Graduate Symp. on the Convergence of Telecommunications, Networking and Broadcasting. 2013. [doi: 10.6084/m9.figshare.3821922.v1]

[24] Lin CH, Tsai SY, Liu CH, *et al*. Accelerating string matching using multi-threaded algorithm on GPU. In: Proc. of the IEEE Global Telecommunications Conf. IEEE, 2010. 1−5. [doi: 10.1109/GLOCOM.2010.5683320]

[25] Xu D, Zhang H, Fan Y. The GPU based high-performance pattern-matching algorithm for intrusion detection. Journal of Computational Information Systems, 2013. 3791−3800. [doi: 10.12733/jcis5781]

[26] Cole R, Gottlieb LA, Lewenstein M. Dictionary matching and indexing with errors and dont cares. In: Proc. of the 36th Annual ACM Symp. on Theory of Computing. New York: ACM Press, 2004. 91−100. [doi: 10.1145/1007352.1007374]

[27] Karkkainen J, Sanders P. Simple linear work suffix array construction. In: Proc. of the Int'l Colloquium on Automata Languages and Programming. 2003. 943−955. [doi: 10.1007/3-540-45061-0_73]

[28] Ko P, Aluru S. Space efficient linear time construction of suffix arrays. In: Proc. of the Combinatorial Pattern Matching. 2003. 200−210. [doi: 10.1007/3-540-44888-8_15]

[29] Khancome C, Boonjing V. New Hashing based multiple string pattern matching algorithms. In: Proc. of the Int'l Conf. on Information Technology: New Generations. 2012. 195−200. [doi: 10.1109/ITNG.2012.34]

[30] Blumer A, Blumer J, Haussler D, *et al*. The smallest automaton recognizing the subwords of a text. Theoretical Computer Science, 1985,40(1):31−55. [doi:10.1016/0304-3975(85)90157-4]

[31] Crochemore M. Transducers and repetitions. Theoretical Computer Science, 1986,45(1):63−86. [doi: 10.1016/0304-3975(86)90041-1]

[32] Inenaga S, Takeda M, Shinohara A, *et al*. The minimum DAWG for all suffixes of a string and its applications. LNCS, 2002. 153−167. [doi: 10.1007/3-540-45452-7_14]

[33] Inenaga S, Bannai H, Shinohara A, *et al*. Discovering best variable-length-don't-care patterns. In: Proc. of the Discovery Science. 2002. 86−97. [doi: 10.1007/3-540-36182-0_10]

[34] Lothaire M. Algebraic Combinatorics on Words. Cambridge University Press, 2002.

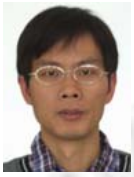
附中文参考文献:

[2] 强继朋,谢飞,高隽,等.带任意长度通配符的模式匹配.自动化学报,2014,40(11):2499−2511. [doi: 10.3724/SP.J.1004.2014.02499]

[8] 柴欣,贾晓菲,武优西,等.一般间隙及一次性条件的严格模式匹配.软件学报,2015,26(5):1096−1112. http://www.jos.org.cn/1000-9825/4707.htm [doi: 10.13328/ j.cnki.jos.004707]

[10] 武优西,刘亚伟,郭磊,吴信东.子网树求解一般间隙和长度约束严格模式匹配.软件学报,2013,24(5):915−932. http://www.jos.org.cn/1000-9825/4381.htm [doi: 10.3724/SP.J.1001.2013.04381]

[15] 汪浩,王海平,吴信东.带有通配符和长度约束的模式匹配问题求解模型.计算机科学,2016,43(4):279−283.

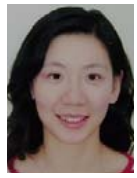


周开来(1978－),男,湖北南漳人,博士,副教授,主要研究领域为数据库,数据挖掘,并行计算.

陈 红(1965－),女,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为数据仓库与数据挖掘,物联网中的数据管理.

熊子绎(1995－),男,硕士生,主要研究领域为数据库,并行计算.

李翠平(1972－),女,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为社会网络分析,社会推荐,大数据分析和挖掘.



孙 辉(1977－),女,博士,讲师,CCF 专业会员,主要研究领域为数据库与数据挖掘,并行计算.