HIT CS&E

第九章 字符串处理算法

骆吉洲
计算机科学与技术学院

HIT CS&E

Outlines

HIT CS&E

9.1 精确字符串匹配

字符串匹配问题

- 输入:
  - 文本 T = “at the thought of”
    - n = length(T) = 17
  - 模式 P = “the”
    - m = length(P) = 3
- 输出:
  - 所有s
    - 整数 $(0 \le s \le n - m)$ 满足 $T[s .. s+m-1] = P[0 .. m-1]$
  - –1, 如果不存在这样的 s

```
0123      …       n-1
at the thought of
s=3 012
    the
```

HIT CS&E

9.1.1蛮力算法

基本思想

在T的每个位置i，检查P是否匹配T的一个子串

T: ABABABCCA　　T: ABABABCCA　　T: ABABABCCA
P: ABABC　　P: ABABC　　P: ABABC

Naive-String-Match(T,P)

Input: 文本T，模式P

Output: P在T中出现的所有位置

1. n←length(T)
2. m←length(P)
3. For k←1 to n-m-1 do
4. IF $P[1,…,m] = T[k,k+1,…,k+m-1]$ THEN print k

时间复杂性$O((n\text{-}m\text{-}1)m)$

HIT CS&E

9.1.2 Rabin-Karp算法

- 基本思想
  - $a \equiv b \mod n$ 则a可能等于b
  - 否则，$a \ne b$
  - 将字符串的比较转化成数的比较
- 其性能优于前面的算法
- 易于推广处理其他类似问题
  - 如二维模式匹配问题

HIT CS&E

```
0 1 2
t h e
20 8 5
```

$20*27^2+8*27+5 \mod 19$ → p=0

a	t		t	h	e		t	h	o	u	g	h	t		o	f
1	20	0	20	8	5	0	20	8	21	15	7	8	20	0	15	6

0 1 2 3 … n-1

$t_3 = 20*27^2+8*27+5 \mod 19 = 0$

$t_2 = 0*27^2+20*27+8 \mod 19 = 16$

$t_1 = 20*27^2+0*27+20 \mod 19 = 8$

$t_0 = 1*27^2+20*27+0 \mod 19 = 15$

如何提高计算效率？

0	1	2
t	h	e
20	8	5

$20*27^2+8*27+5 \bmod 19$ → p=0

a	t		t	h	e		t	h	o	u	g	h	t		o	f
1	20	0	20	8	5	0	20	8	21	15	7	8	20	0	15	6

0 1 2 3 … n-1

$t_3= 20*27^2+8*27+5 \bmod 19 = 0 = 27*t_2-27^3*0+5 \bmod 19$

$t_2= 0*27^2+20*27+8 \bmod 19 = 16 = 27*t_1-27^3*20+8 \bmod 19$

$t_1= 20*27^2+0*27+20 \bmod 19 =8 = 27*t_0-27^3*1+20 \bmod 19$

$t_0= 1*27^2+20*27+0 \bmod 19 =15$

- $d=|\Sigma|$
  - Σ^*内任意字符串x可以看成一个d进制数
  - 如$P[0,2,\ldots,m-1]$可以看成

    $p = P[m-1]+ d (P[m-1]+ d (P[m-2]+\ldots+dP[0])\ldots)$

    计算p需要$O(m)$的时间
  - 类似地，$T[k,k+1,\ldots,k+m-1]$可以看成

    $t_k = T[k+m-1]+ d (T[k+m-2]+ d (T[k+m-3]+\ldots+dT[k])\ldots)$

    计算t_k需要$O(m)$的时间
- 如果$p=t_k$,则P在T中位置k处可能有一个匹配
- 如果在每个位置k均重新计算t_k，则$O((n-m+1)m)$
- 能否提高计算t_k的效率

$t_k = T[k+m-1]+ d (T[k+m-2]+ d (T[k+m-3]+\ldots+dT[k])\ldots)$

$t_{k+1} = T[k+m]+ d (T[k+m-1]+ d (T[k+m-2]+\ldots+dT[k+1])\ldots)$

$t_{k+1}-dt_k=T[k+m]-d^mT[k]$

$t_{k+1}=dt_k+T[k+m]-d^mT[k]$
- d^m是与k无关的常数，可以事先计算出来
- 根据t_k来计算t_{k+1}仅需常数开销

Rabin-Karp-Matcher(T,P,d,q)

Input: 文本T，模式P，基数d，素数q

Output: P在T中出现的所有位置

1. n←length(T)
2. m←length(P)
3. h← d^m mod q
4. p← 0
5. t_0← 0
6. For i←0 to m-1 do   //计算p和t_0 $O(m)$
7. p← $dp+P[i]$ mod q;
8. t_0← $dt_0+T[i]$ mod q;
9. for k ←0 to n-m do n-m+1遍 次数c
10.   If $p=t_k$ Then
11.   If $P[0,\ldots,m-1] = T[k,k+1,\ldots,k+m-1]$ Then print k $O(m)$
12. t_{k+1} ←dt_k-$T[k]h + T[k+m]$ mod q $O(1)$

时间复杂度分析 $O(n-m+1 + cm)$

如果$c=O(1)$,则算法的时间复杂度为$O(n+m)$

9.1.3 FA与字符串匹配

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
$P[i]$			a	b	a	b	a	c	a		

扫描T的过程

准备扫描

第一个字符　P[1]已经被匹配

第二个字符　P[1,2]已经被匹配

第三个字符　P[1,2,3]已经被匹配

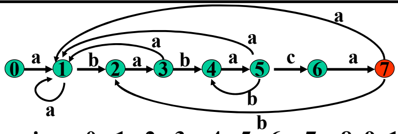
第四个字符　P[1,2,3,4]已经被匹配

第五个字符　P[1,2,3,4,5]已经被匹配

第六个字符　P[1,2,3,4]已经被匹配

……

	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	



i		0	1	2	3	4	5	6	7	8	9	10
$T[i]$		a	b	a	b	a	b	a	c	a	b	
$\phi(T[i])$	0	1	2	3	4	5	4	5	6	7	2	

Finite-Autoation-Matcher(T,δ,m)

Input: 文本T,FA的状态转移函数,模式长度m

Output: P在T中出现的所有位置

1. n←length(T)
2. q←0
3. For k←0 to n-1 do
4. q← $\delta(q,T[k])$
5.   IF $q=m$ THEN print k-m+1

如果FA存在,则找出所有匹配仅需对$T[1,2,\ldots,n]$线性扫描

关键是如何构造这个FA?

HIT CS&E

模式P的FA的构造

$$\begin{array}{cccccccccc}
t_1 & \cdots & t_i & \cdots & t_{j-r+1} & \cdots & t_{j-1} & t_j & \cdots \\
 & & p_1 & \cdots & p_{k-r+1} & \cdots & p_{k-1} & & \\
 & & & & p_1 & \cdots & p_{r-1} & p_r & \cdots
\end{array}$$

扫描到$T[j-1]$, 有穷状态机进入状态k-1
扫描到$T[j]$时

如果$p_k = t_j$,则 有穷状态机直接进入状态k

否则，假设有穷状态机进入状态r，意味着…
$t_1, t_2,\dots,t_{j-1},t_j$的所有后缀中能够与P的某个前缀匹配的最大长度等于$r$

$\forall x \in \Sigma^*$ 令$\sigma(x)=\max\{k: P_k ▸ x\}$ 且$\sigma(\varepsilon)=0$

例如 P=ab 则$\sigma(\varepsilon)=0$ $\sigma(ccaca)=1$ $\sigma(ccab)=2$

则 $\delta(k\text{-}1, a) = \sigma(T_{j-1}a) = \sigma(P_{k-1}a)$

Compute-Transition-Function(P,Σ)

Input: 模式P，字符表Σ

Output: P对应有穷自动机的状态转移函数δ

1. $m \leftarrow$length(P)
2. For $q \leftarrow 0$ to m do
3. For $\forall a \in \Sigma$ do
4. $k \leftarrow \min\{m+1, q+2\}$
5. repeat $k \leftarrow k$-1 until $P_k ▸ P_q a$
6. $\delta(q,a) \leftarrow k$
7. return δ;

时间复杂度$O(m^3|\Sigma|)$

可以用更快的算法将时间复杂度改进到O($m|\Sigma|$)

HIT CS&E

9.1.4 Knuth-Morris-Pratt算法

考察最简单最直接的扫描过程

	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b	T
—— s ——→					a	b	a	b	a	c	a	P				

–P_5已经被匹配上但P_6不匹配
–由于a≠b,在s+1开始的扫描不可能得到正确匹配,应逃过
–下一个可能正确的匹配应起始于何处?

	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b	T
—— s'=s+2 ——→						a	b	a	b	a	c	a	P			

一般而言，下面的问题可以加速扫描过程

设$P[1,\dots,q]$匹配了$T[s+1,\dots,s+q]$,最小的s'是什么，使得(1)s'>s; (2) $P[1,\dots,k]=T[s'+1,\dots,s'+k]$; (3) $s+q = s'+k$

	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b	T
—— s ——→					a	b	a	b	a	P_q						
—— s' ——→							a	b	a	P_k						

仅需将P_q与其自身进行比较
等价于找最大的k使得$P_k ▸ P_q$
无需存储s'，只需存储s'-s

如果上述问题知道答案
–如果$T[s+q+1]=P[q+1]$扫描继续进行
–否则
•扫描跳过若干无效扫描位置
•直接转入起始于s'+1开始新的扫描
•在新的扫描位置，有k个字符已匹配，无需再扫描
答案要借助哪些信息才能计算，如何计算?

前缀函数—可用于高效计算9.1.3中的状态转移函数

给定模式P=P[1,2,3,…,m],模式P的前缀函数是一个函数$\pi:\{1,2,\dots,m\} \rightarrow \{0,1,2,\dots,m\text{-}1\}$使得$\pi(q)=\max\{k:k<q$且$P_k ▸ P_q\}$

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4				

a	b	a	b	a	P_5
	a	b	a	b	
		a	b	a	

$O(m^2)=0$

	1	2	.	…	q-1	q	
	a	b	.	…	a	b	P_q
$k=\pi(q\text{-}1)$			.	…	a		
$k=\pi(k)=\pi(\pi(q\text{-}1))$				…	a	b	

a	b	a	b	a	b	P_6
k=3		a	b	a	b	

$\pi(1)=0$
$\pi(2)=0$
$\pi(3)=1$
$\pi(4)=2$
$\pi(5)=3$
$\pi(q)=k+1$
$\pi(6)=4$

上述过程可以重复，最坏情况下直到k=0

从而避免两重循环的扫描

经过平摊分析知上述过程的时间开销为O(m)

Compute-Prefix-Function(P)

Input: 模式P

Output: P的前缀函数π

1. $m \leftarrow$length(P)
2. $\pi[1] \leftarrow 0$; $k \leftarrow 0$;
3. For $q \leftarrow 2$ to m do
4. While k>0 & $P[k+1] \neq P[q]$ do
5. $k \leftarrow \pi[k]$;
6. If $P[k+1]=P[q]$ then $k \leftarrow k+1$;
7. $\pi[q] \leftarrow k$;
8. return π;

第5步/每次,势能增量≤-1
第6步/每次,势能增量1

For循环体的每次执行
- 第5步执行遍数x
- While循环代价为x
- 第6步执行1遍
- 第6步的代价为1

For循环体的每次执行
- α=c+势能增量
  =(x+1)+势能增量
  ≤1 (x≠0)
  ≤2 (x=0)

令状态k的势能为k

经过平摊分析知上述过程的时间开销为O(m)

KMP-Matcher(*P*,*T*)

Input: 模式*P*,*T*

Output: *P*在T匹配的所有起始位置

1. m←length(P); n←length(T)
2. π←Compute_Prefex_Function(P);
3. q←0; //用于跟踪已匹配的字符数
4. For i←2 to n do //从左到右扫描T
4. While k>0 & $P[q+1]\neq T[i]$ do
5. q← $\pi[q]$;
6. If $P[q+1]=T[i]$ then q←q+1;
7. If q=m then
8. print i-m+1;
8. q ← $\pi[q]$;

令状态q的势能为q,经过平摊分析知上述过程的时间开销为O(m+n)

9.1.5 Boyer-Moore算法

- 还存在比KMP算法效率更高的匹配算法吗？
- 还能从其他角度处理字符串匹配问题吗？
- 常用的字符串匹配算法是如何实现的呢？

- 德克萨斯大学的Robert S. Boyer和J Strother Moore 1977年发明的算法

基本思想

将输入串T与模式P的开始位置对齐，从后往前比较

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6
```

模式P的最后一个字符没匹配上，意味着…

- "坏"字符 –刻画了P在整体上与T在该位置的不相似
  - T,P位置对齐
  - 从后向前扫描
  - 首个不匹配字符
- P需要向后移动若干个字符才可能发生匹配

T[6]≠P[6]且 T[6]∉P，意味着…

- P至少向后移动|P|个位置才可能发生匹配

基本思想（续）

将输入串T与模式P的开始位置对齐，从后往前比较

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
          0 1 2 3 4 5 6
```

T[13]≠P[6]但T[13]=P[4]，意味着…

- P至少向后移动6-4=2个位置可能发生匹配

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

            EXAMPLE
            0 1 2 3 4 5 6
```


基本思想（续）

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

          EXAMPLE
          0 1 2 3 4 5 6
```

好后缀 ： P[6]
出现位置：6

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

          EXAMPLE
          0 1 2 3 4 5 6
```

好后缀 ： P[5,6]
出现位置：5

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

          EXAMPLE
          0 1 2 3 4 5 6
```

好后缀 ： P[4-6]
出现位置：4

"好"后缀：从后向前扫描"坏字符"前遇到每个P-后缀

基本思想（续）

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

          EXAMPLE
          0 1 2 3 4 5 6
```

好后缀 ： P[3-6]
出现位置：3

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

          EXAMPLE
          0 1 2 3 4 5 6
```

"坏"字符又出现了，应该把模式向后移动几个位置？

"坏"字符规则

- 如果T[i]≠P[j]且，则后移次数k=j-a
- a= T[i]最近扫描时在P中的出现位置
- a= 如果最近扫描没有遇到过T[i]，则-1

除了"坏"字符规则外，还有其他机制使移动更高效吗？

HIT CS&E

基本思想（续）

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

好后缀 ： P[3-6]
出现位置：3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

“坏”字符又出现了，应该把模式向后移动几个位置？

“好”后缀规则

- 如果T[i]≠P[j]且，则后移次数$k=b-a$
- b= 最近扫描时“好”后缀在P中的出现位置
- a= 该好后缀在模式中的上一次出现位置

HIT CS&E

基本思想（续）

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

好后缀 ： P[3-6]
出现位置：6

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

“坏”字符又出现了，应该把模式向后移动几个位置？

“坏”字符规则，后移次数=2-(-1)=3

“好”后缀规则，后移次数=6-(0) =6

HIT CS&E

基本思想（续）

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

最终规则

- 后移次数k=max\{好后缀规则，坏字符规则\}
- “坏”字符规则 = 6-4 = 2
- “好”后缀规则=0-(-1)=1

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

哪些信息可以预计算以提高效率？
算法如何形式化描述？
算法的复杂性如何？

HIT CS&E

9.1.6 Boyer-Moore-Horspool算法

- 还存在比BM算法效率更高的匹配算法吗？
  - BM算法仍存在冗余扫描
  - 跨度时机不好选择
- 德克萨斯大学的Robert S. Boyer和J Strother Moore
  - 1977年发明的算法
  - 不再关注失配字符
  - 而是关注失配发生时，与P对齐的最后一个字符x

x

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

后移量 = |P|-1 – x在P中的最后出现位置
后移量 = |P| x在P中未出现

HIT CS&E

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE
EXAMPLE
0 1 2 3 4 5 6

后移量 = 7-1-4 = 2

HIT CS&E

基本思想（续）

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
HERE IS A SIMPLE EXAMPLE

EXAMPLE
0 1 2 3 4 5 6

HIT CS&E

基本思想（续）

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 H E R E   I S   A   S I M P L E   E X A M P L E
                   E X A M P L E
                   0 1 2 3 4 5 6
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 H E R E   I S   A   S I M P L E   E X A M P L E
                   E X A M P L E
                   0 1 2 3 4 5 6
```

后移量 = 7　　　I未在P中出现

HIT CS&E

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 H E R E   I S   A   S I M P L E   E X A M P L E
                       E X A M P L E
                       0 1 2 3 4 5 6
```

后移量 = 7-1-5=1　　　L在P[5]出现

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 H E R E   I S   A   S I M P L E   E X A M P L E
                                   E X A M P L E
                                   0 1 2 3 4 5 6
```

理解BMH算法之后，写出代码并做分析

HIT CS&E

9.2 后缀树及其构造算法

- 在模式P给定且被预处理的情况下，任给文本*T*, 模式匹配问题可以在 *O*(|*T*|)时间内求解
  - ***信息过滤系统
  - 在病人DNA序列中寻找给定疾病的DNA模式
- 在文本T给定且可以被预处理的情况下，任给模式*P*, 模式匹配问题需要多长时间才能求解？怎样求解？
  - 静态网页中的关键字搜索
  - 数字图书馆中的信息查询

这些实际问题中的科学问题到底是什么?

HIT CS&E

- 预处理 *P*
  - Gusfield
  - Boyer-Moore
  - Knuth-Morris-Pratt
  - Boyer-Moore-Horspool
- 预处理*T*
  - 后缀树

HIT CS&E

9.2.1 相关概念

- 问题定义
  - Input: 模式*P* 和文本 *S*
    （*S*可被预处理，*T*有其他用途）
  - Output: *P*在*S*中的一次出现.
- 例
  - S = b b a b b a a b
  - P = b a a

HIT CS&E

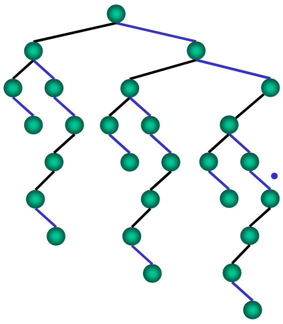
后缀

```
T       =  b b a b b a a b
T[1…8]= b b a b b a a b      1-后缀
T[2…8]=   b a b b a a b      2-后缀
T[3…8]=     a b b a a b      3-后缀
T[4…8]=       b b a a b      4-后缀
T[5…8]=         b a a b      5-后缀
T[6…8]=           a a b      6-后缀
T[7…8]=             a b      7-后缀
T[8…8]=               b      8-后缀
```

注意：*P*在*S*中出现当且仅当*P*是某个*i*-后缀的前缀

所谓预处理就是恰当地组织左右后缀，使得…

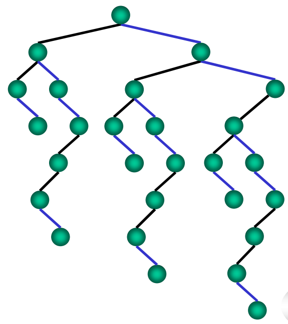
T = S的suffix trie



b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

- 下列论断等价
  - ***P*** 在***S***中出现
  - ***P*** 是***S***的某个后缀的前缀.
  - ***P*** 在***S***的后缀树***T***中对应一条起始于根的路径.

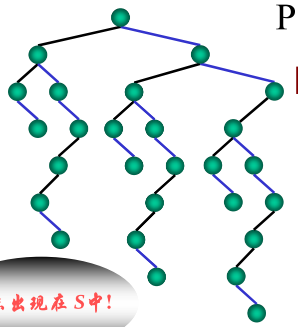
P = b a b b a



b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

P在S中出现!

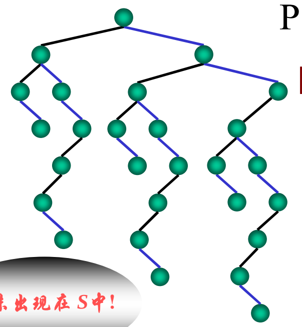
P = b b a a b a



b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

P未出现在S中!

P = a b b b a a

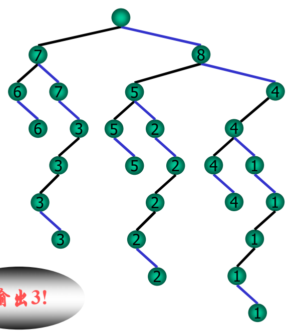


b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

P未出现在S中!

P在S中何处出现?

P = a b b a a



b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

输出3!

问题

存储suffix trie的空间开销为

$\Theta(|S|)$ $\Theta(|S| \log |S|)$ $\Theta(|S|^2)$ $\Theta(|S|^3)$

为S构建其suffix trie *T*的时间开销为多少? $\Theta(|S|^2)$

是否存在suffix trie的更紧凑的表示使得其构建时间和存储空间均是线性的?

HIT
CS&E

后缀树

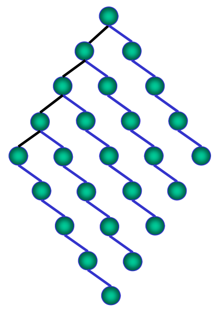
suffix trie的一种更紧凑的表示

HIT
CS&E

后缀树

观察*S*的suffix Trie *T*

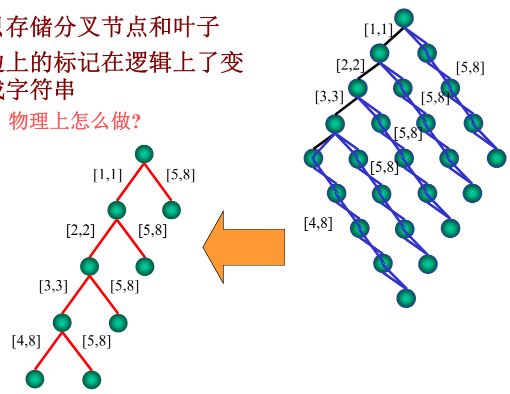
- ***T*** 至多有 |*S*|个叶子.
  - Why?
- ***T*** 至多有 |*S*|个分叉内节点.
  - Why?



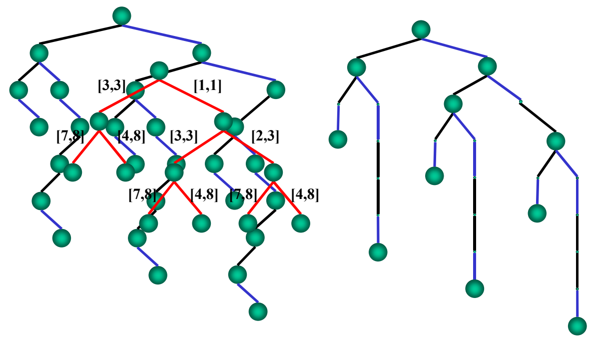
HIT
CS&E

S = a a a a b b b b

- 只存储分叉节点和叶子
- 边上的标记在逻辑上了变成字符串
  - 物理上怎么做?



S = b b a b b a a b



问题

- 后缀树的空间开销
  - *O*(|*S*|)
  - *O*(|*S*| log |*S*|)
  - *O*(|*S*|²)
  - *O*(|*S*|³)
- 为什么?
  - 后缀树中的节点数 =
  - 后缀树中的边数 =
  - 每条边需要的存储空间 =

挑战

如何在线性时间内构造后缀树?

HIT
CS&E

后缀树的历史点滴

- [Weiner, *IEEE FOCS* 1973]
  - 线性时间 空间开销巨大.
  - D. E. Knuth: "the algorithm of 1973".
- [McCreight, *J. ACM* 1976]
  - O(*n*)时间 O(*n*²)空间
- [Ukkonen, *Algorithmica* 1995]
  - O(*n*)时间 O(*n*)空间
  - 更稳定可靠.

HIT CS&E

Academy Professor , Department of Computer Science , University of Helsinki, Finland

http://www.cs.helsinki.fi/u/ukkonen/

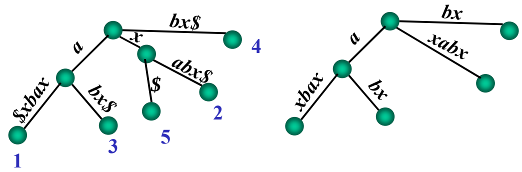
Esko Ukkonen: On-line construction of suffix-trees. Algorithmica 14 (1995), 249-260

HIT CS&E

隐式后缀树

S$的后缀树*T*中

- **step 1.** 删除$符号
- **step 2.** 删除无标签的边
- **step 3.** 删除只有一个孩子的顶点



HIT CS&E

算法基本思想

1.T_1为*S*[1]的隐式后缀树
2.For *i*=2 to |*S*|
3. 更新T_{i-1}为*S*[1,….,*i*]的隐式后缀树T_i
4. end for
5.更新$T_{|S|}$为*S*$的隐式后缀树,即*S*的后缀树

关键：如何加速第3步

HIT CS&E

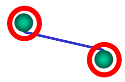
Ukkonen's构建后缀树的方法

T[0]

S= b b a b b a a b

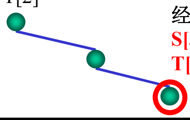
S[1] = b
T[0]经过什么变化才能变为T[1]?

T[1]



经过一次叶柄变长
S[2] = b
T[1]经过什么变化才能变为T[2]?

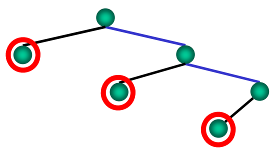
T[2]



经过一次叶柄变长
经过一次若无其事
S[3] = a
T[2]经过什么变化才能变为T[3]?

HIT CS&E

T[3]



S= b b a b b a a b

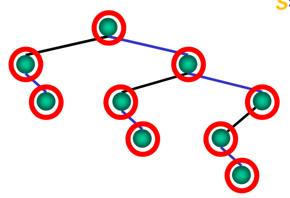
b b a
b a
a

经过一次叶柄变长
经过两次长新叶子
S[4] = b
T[3]经过什么变化才能变为T[4]?

Case 1: 叶柄变长
Case 2: 长新叶子
Case 3: 若无其事

HIT CS&E

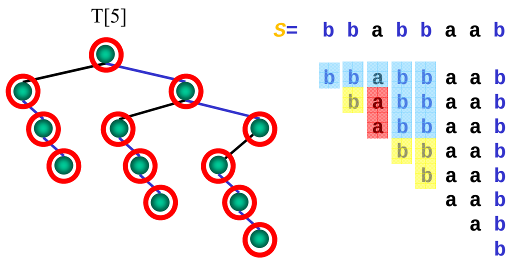
T[4]



S= b b a b b a a b

b b a b b a a b
b a b b a a b
a b b a a b
b b a a b
b a a b
a a b
a b
b

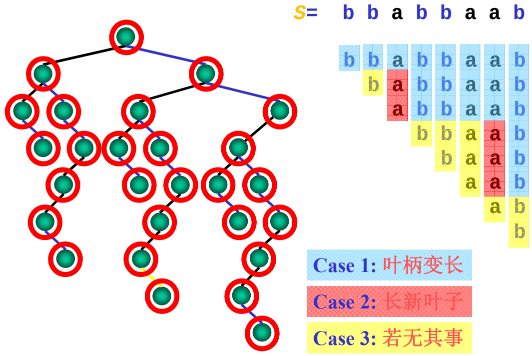
Case 1: 叶柄变长
Case 2: 长新叶子
Case 3: 若无其事



Case 1: 叶柄变长

Case 2: 长新叶子

Case 3: 若无其事



Case 1: 叶柄变长

Case 2: 长新叶子

Case 3: 若无其事

构建后缀树时涉及到三类动作

- Case 1: 在一个叶子节点上延申边，叶柄变长
- Case 2: 在一个内节点处长新叶子，长新叶子
- Case 3: 树结构不发生任何变化 ，若无其事

三阶段定理

处理$S[k]$的k个步骤由如下一系列步骤构成:

(或，处理后缀的第k列的k个步骤如下构成:)

- •首先是一系列 (至少一个) 叶柄变长
- •然后是一系列 (可能0个) 长新叶子
- •最后是一系列 (可能0个) 若无其事

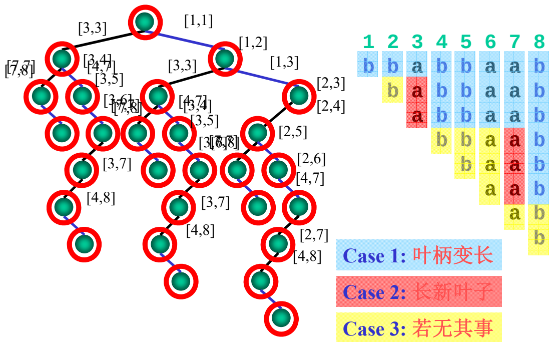
引理1.在构建$T_1,\ldots,T_m$的过程中，一旦一个叶子产生出来就不会消失，后续过程中该叶子可能的变化只能是叶柄变长

引理2.将构建T_i的过程中最后一次长新叶子的操作时刻记为j_i,则$j_i \le j_{i+1}$

引理3.在构建T_i的过程中第一次若无其事发生后，以后的操作均是若无其事

结合下面的实例，自己给出证明

Thinking in Suffix Tree



Case 1: 叶柄变长

Case 2: 长新叶子

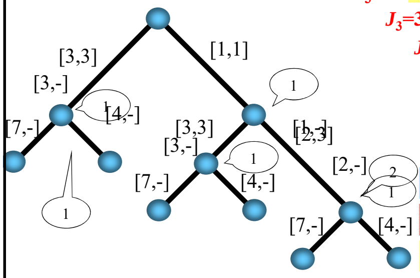
Case 3: 若无其事

利用三阶段定理加速构建

- 利用形如[i,-]标记叶边，-是记录字符串结束位置的标记，叶柄变长无需显式操作，常数时间即可完成所有已知的叶柄变长
- J_i记录T_i构建过程中最后一次长新叶子的操作位置,帮助我们定位已知的叶子

HIT
CS&E

S = b b a b b a a b



	1	2	3	4	5	6	7	8
J_1=1	b	b	a	b	b	a	a	b
J_3=3		b	a	b	b	a	a	b
J_3=3			a	b	b	a	a	b
J_4=3				b	b	a	a	b
J_5=3					b	a	a	b
J_7=6						a	a	b
J_8=6							a	b
								b

Case 1: 叶柄变长

Case 2: 长新叶子

Case 3: 若无其事

HIT
CS&E

进一步加速—思想梗概

T_i->T_{i+1}

考虑将$S[j,\ldots i]S(i+1)$添加到T_i的过程

1.$S[j,\ldots i]$必然被某条路径标记

2.结束位置不一定是结点

3.如何有效发现距结束位置最近的结点

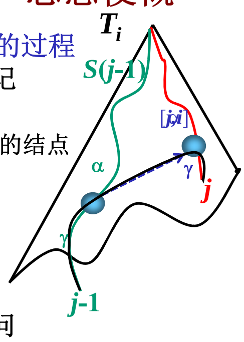
将$S[j-1,j,\ldots i]S(i+1)$添加到T_i时

1.经过路径$S[j-1,j\ldots i]S(i+1)$

2.中间某个结点有指针指向α的结束结点

3.省略很多不必要的路径访问

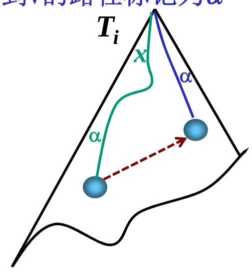
4.这种指针被称作**suffix link**



HIT
CS&E

Suffix link

T_i中从根结点出发到结点u的路径标记为$x\alpha$,则结点u的**suffix link**指向结点v, 其中x是一个字符且从根结点到v的路径标记为α



HIT
CS&E

Suffix Links性质

性质1

- suffix link的出发点总是后缀树的内部节点
  - 不可能是叶节点
  - 也不可能是一条边的中间位置
- 为什么?

性质2

- 后缀树的每个内部节点均是某个suffix link的出发点
- 为什么?

HIT
CS&E

性质2的证明

设

(1)T_i已经构造出来

(2)每个内结点均有一个suffix link

(3)$J_i=k\geq1$

T_i->T_{i+1}

(1)S[1,2,…,i]S(i+1)->T_i

(2)S[2,…,i]S(i+1)->T_i

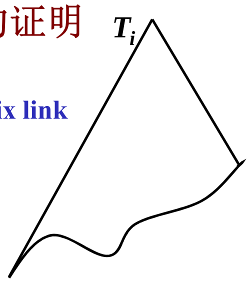
…

(3)S[k,….,i]S(i+1)-> T_i

(4)上述操作均是叶柄变长—隐式地完成

不增加新的内结点，故各内结点均有suffix link

(5)只需考虑S[j,….,i]S(i+1)-> T_i (j≥k≥2)



HIT
CS&E

性质2的证明(续)T_{i+1}

T_i->T_{i+1}:S[j,….,i]S(i+1)-> T_i (j≥k≥2)

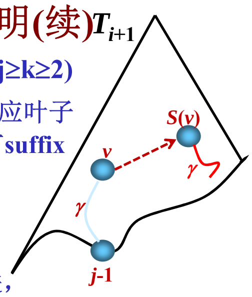
(1)S[j-1,j,…,i,i+1]在当前T_{i+1}对应叶子

(2)从该叶子上行找到第一个有suffix link的顶点v

(3)经过路径的标签为γ

(4)由suffix link从v到达$s(v)$

(5)从s(v)沿标记为γ的路径前进，或长新叶子，或若无其事



问题：如果第(5)步添加内结点w从而长新叶子，$s(w)$从何而来？

Suffix link的维护

T_i->T_{i+1}:S[j,….,i]S(i+1)-> T_i (j≥k≥2)

添加了内结点w

S[j+1,….,i]S(i+1)-> T_i添加过程相同

从s(v)沿标记为δ的路径前进，在δ结束位置有两种情况

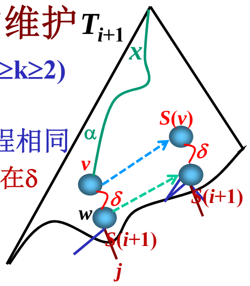
Case 1: δ结束于某个结点u

s(w)=u

Case 2: δ结束于某条边内部

S(i+1)必然导致新的内结点u产生

s(w)=u



运用后缀链加速后缀树的构建

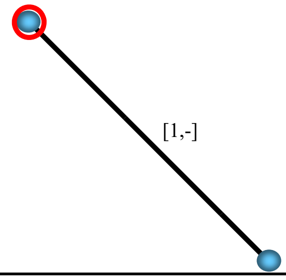
1 2 3 4 5 6 7 8

S= b b a b b a a b

S_1= b

将S_1的所有后缀插入T

J_1=1



运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

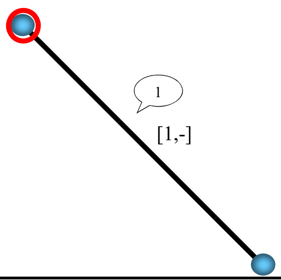
S= b b a b b a a b

J_1=1

S_2= b b

将S_2的所有后缀插入T

J_2=1



运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

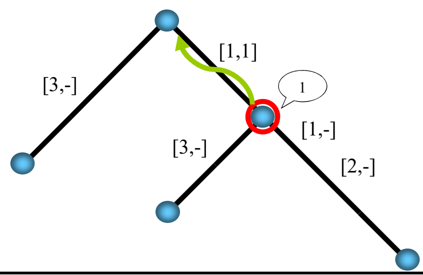
S= b b a b b a a b

J_2=1

S_3= b b a

将S_3的所有后缀插入T

J_3=3



运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

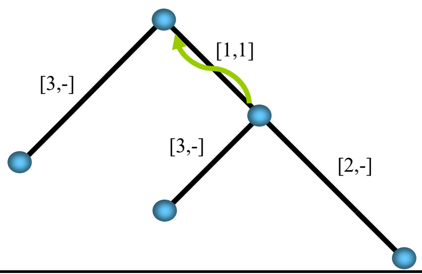
S= b b a b b a a b

J_3=3

S_4= b b a b

将S_4的所有后缀插入T

J_4=3



运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

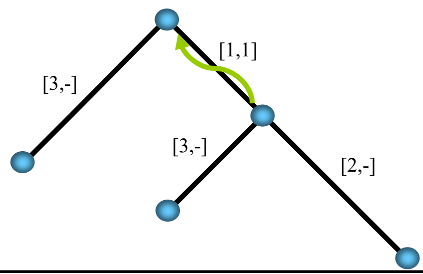
S= b b a b b a a b

J_4=3

S_5= b b a b b

将S_5的所有后缀插入T

J_5=3



HIT CS&E

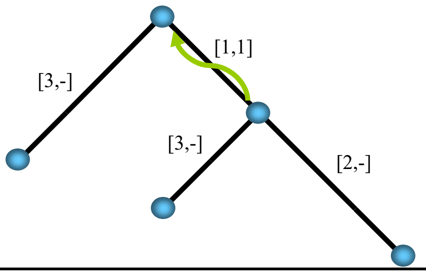
运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

S= b b a b b a a b J_5=3

S_6= b b a b b a 将S_6的所有后缀插入T

J_6=3



HIT CS&E

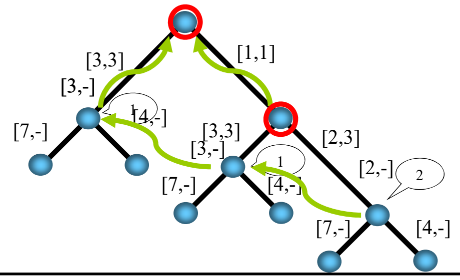
运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

S= b b a b b a a b J_6=3

S_7= b b a b b a a 将S_7的所有后缀插入T

J_7=4



HIT CS&E

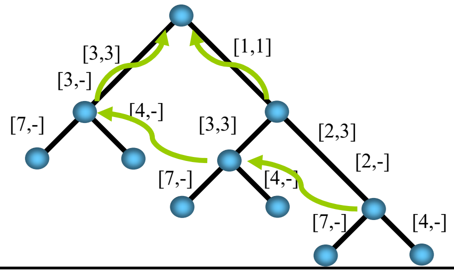
运用后缀链加速后缀树的构建

1 2 3 4 5 6 7 8

S= b b a b b a a b J_7=4

S_8= b b a b b a a b 将S_7的所有后缀插入T

J_7=4



HIT CS&E

Suffix tree的构造时间

- O(n) —略

HIT CS&E

9.3 近似字符串匹配

- 问题定义
- 相似性度量函数
- 一些典型求解思路及研究前沿

HIT CS&E

DBLP 作者搜索

HIT
CS&E

尝试一下这些名字 (good luck!)



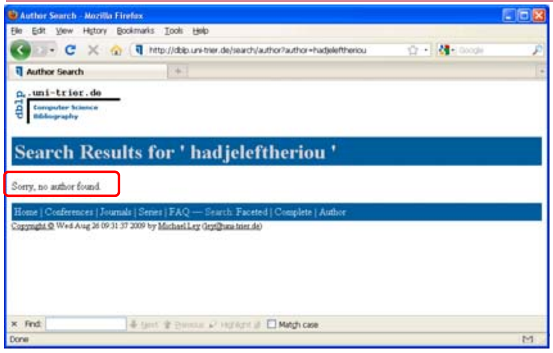
UCSD	Case Western	AT&T--Research
Yannis Papakonstantinou	Meral Ozsoyoglu	Marios Hadjieleftheriou

http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html

79

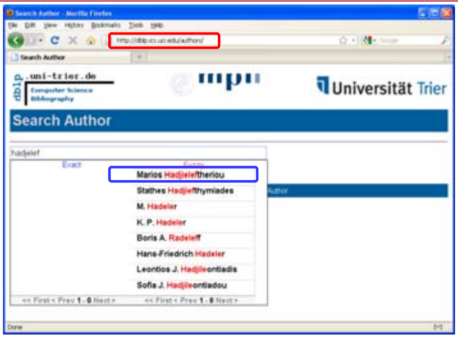
HIT
CS&E

☹



HIT
CS&E

更好的系统?



http://dblp.ics.uci.edu/authors/

81

HIT
CS&E

UC Irvine的人名搜索

http://psearch.ics.uci.edu/

82

HIT
CS&E

Web Search

- 查询的错误
- 数据的错误
- 使得查询和有意义结果接近

```
488941 britney spears
 40134 brittany spears
 36315 brittney spears
 24342 britany spears
  7331 britny spears
  6633 briteny spears
  2696 britteny spears
  1807 briney spears
  1635 brittny spears
  1479 brintey spears
  1479 britanny spears
  1338 britiny spears
  1211 britnet spears
  1096 britiney spears
```

Google 搜集的实际查询

http://www.google.com/jobs/britney.html

83

HIT
CS&E

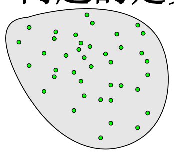
数据清洗

R	S
informix	infromix
microsoft	…
…	mcrosoft
…	…

84

HIT
CS&E

问题的定义



输入：字符串*q*,字符串集合*D*,相似性函数dist(.,.)
输出：*D*中与*q*相似的所有字符串：dist(Q,D) ≤ δ
例: 找到和 “**hadjeleftheriou**”相似的字符串

你有什么办法呢？

性能很重要!
-10 ms: 100 查询/秒 queries per second (QPS)
- 5 ms: 200 QPS

85

HIT
CS&E

基础知识

86

HIT
CS&E

相似性函数

- 相似性函数:
  - 领域相关函数
  - 返回字符串间的相似性值
- 例如:
  - 编辑距离
  - Hamming距离
  - Jaccard相似性
  - Soundex
  - TF/IDF, BM25, DICE
  - ……

87

HIT
CS&E

编辑距离

- 一种广泛使用的字符串相似性测度
- Ed(s1,s2) =将s1变化到s2需要的最小操作数
  - 增加、删除、修改
- 例:

  s1: Tom Hanks

  s2: Ton Hank

  ed(s1,s2) = 2

88

HIT
CS&E

技术: Oracle 10g

- Oracle Text
- CREATE TABLE engdict(word VARCHAR(20), len INT);
- 创建文本索引:

```
begin ctx_ddl.create_preference('STEM_FUZZY_PREF', 'BASIC_WORDLIST');
ctx_ddl.set_attribute('STEM_FUZZY_PREF','FUZZY_MATCH','ENGLISH');
ctx_ddl.set_attribute('STEM_FUZZY_PREF','FUZZY_SCORE','0');
ctx_ddl.set_attribute('STEM_FUZZY_PREF','FUZZY_NUMRESULTS','5000');
ctx_ddl.set_attribute('STEM_FUZZY_PREF','SUBSTRING_INDEX','TRUE');
ctx_ddl.set_attribute('STEM_FUZZY_PREF','STEMMER','ENGLISH'); end; /
```

- CREATE INDEX fuzzy_stem_subst_idx ON engdict (word) INDEXTYPE IS ctxsys.context PARAMETERS ('Wordlist STEM_FUZZY_PREF');
- 使用方法:

```
SELECT * FROM engdict
WHERE CONTAINS(word, 'fuzzy(universisty, 70, 6, weight)', 1) > 0;
```

- 限制: 不能处理首字母错误:

  Katherine VS Catherine

89

HIT
CS&E

Microsoft SQL Server

- SQL Server 2005提供数据清洗工具
- 信息集成工具的一部分
- 支持模糊查询
- 相似性函数: 基于TF/IDF的打分

90

HIT CS&E

Lucene

- 使用Levenshtein Distance (编辑距离).
- 先基于前缀过滤，然后执行搜索 (效率?)

91

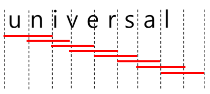
HIT CS&E

基于Gram的算法

92

HIT CS&E

"q-grams"

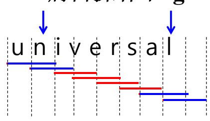


2-grams

93

HIT CS&E

编辑操作和gram的关系



固定长度: q

k 个操作会影响$k * q$ 个gram

如果ed(s1,s2) <= k, 那么他们公共gram的数量 >=
$(|s_1|- q + 1) – k * q$

94

HIT CS&E

q-gram 的倒排表

id	strings
0	rich
1	stick
2	stich
3	stuck
4	static

2-grams →

at	4
ch	0 → 2
ck	1 → 3
ic	0 → 1 → 2 → 4
ri	0
st	1 → 2 → 3 → 4
ta	4
ti	1 → 2 → 4
tu	3
uc	3

95

HIT CS&E

使用倒排表的搜索

- 查询: "shtick", ED(shtick, ?)≤1

sh ht ti ic ck

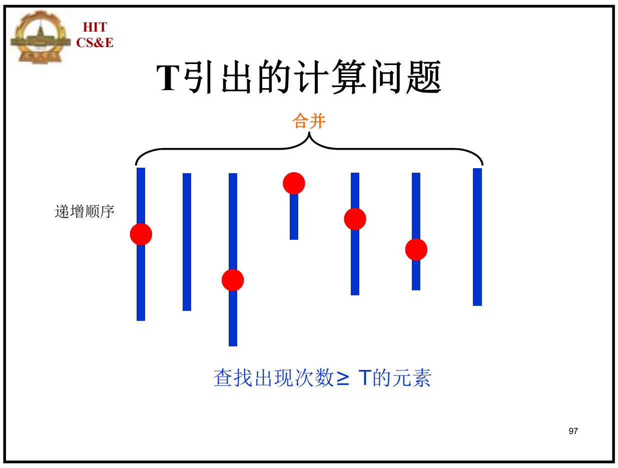
公共 gram的数量 >= 3

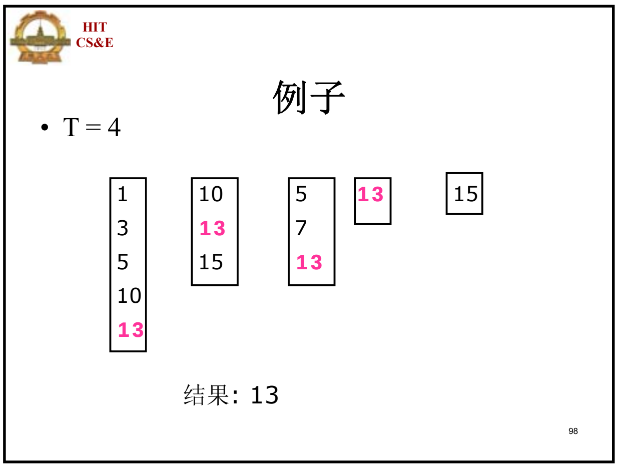
id	strings
0	rich
1	stick
2	stich
3	stuck
4	static

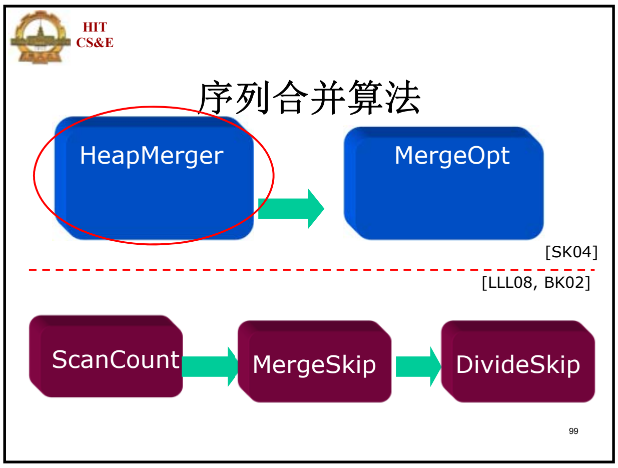
2-grams →

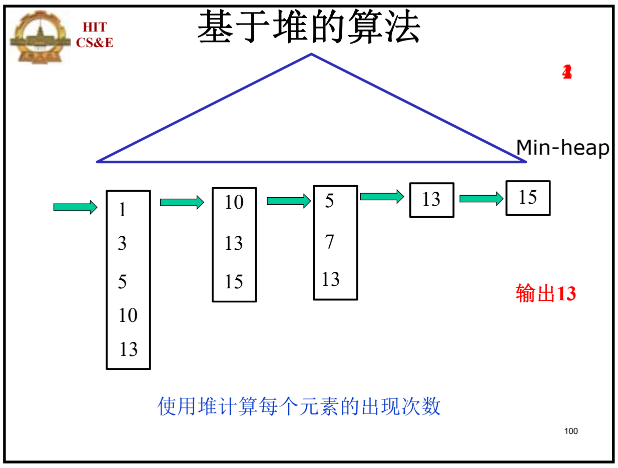
at	4
ch	0 → 2
ck	1 → 3
ic	0 → 1 → 2 → 4
ri	0
st	1 → 2 → 3 → 4
ta	4
ti	1 → 2 → 4
tu	3
uc	3

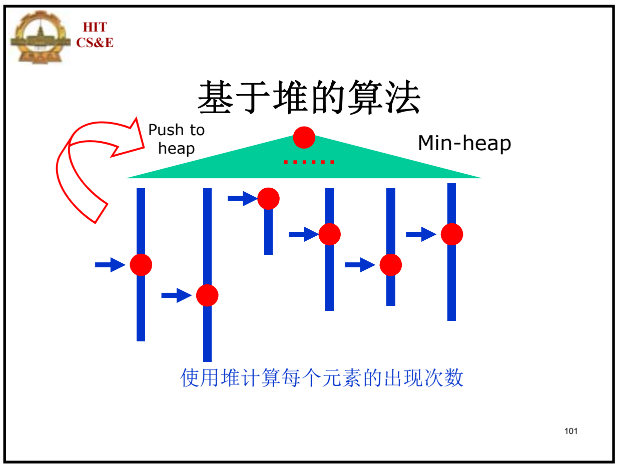
96

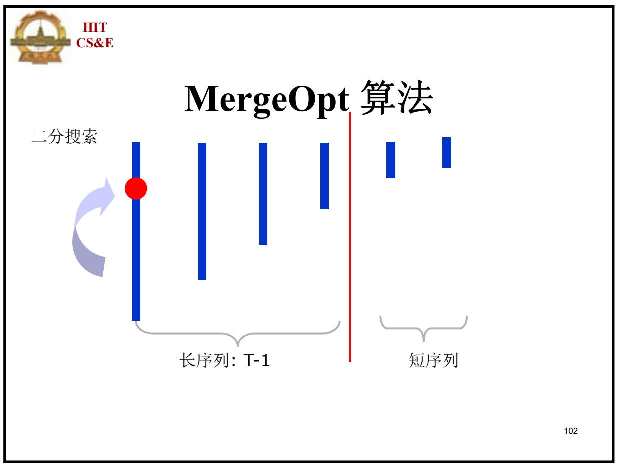

HIT
CS&E
T引出的计算问题
合并
递增顺序
查找出现次数≥ T的元素
97

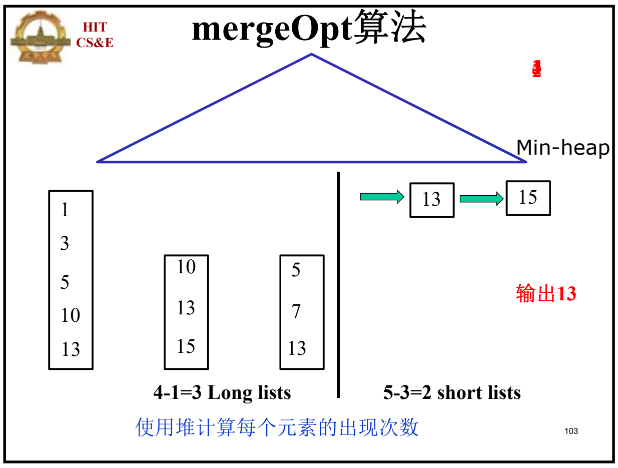

HIT
CS&E
例子
• T = 4
1
3
5
10
13
10
13
15
5
7
13
13
15
结果：13
98

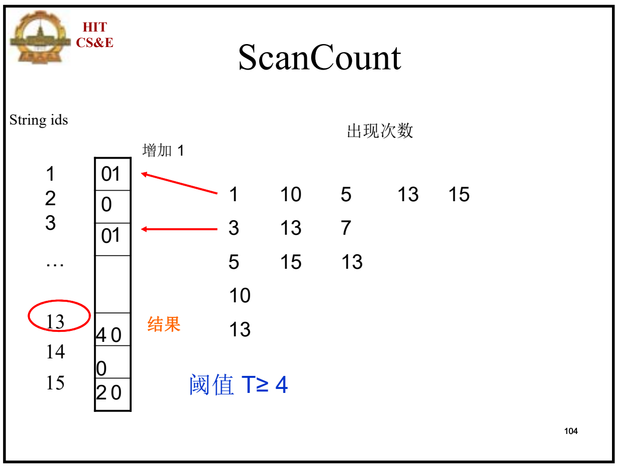

HIT
CS&E
序列合并算法
HeapMerger
MergeOpt
[SK04]
[LLL08, BK02]
ScanCount
MergeSkip
DivideSkip
99

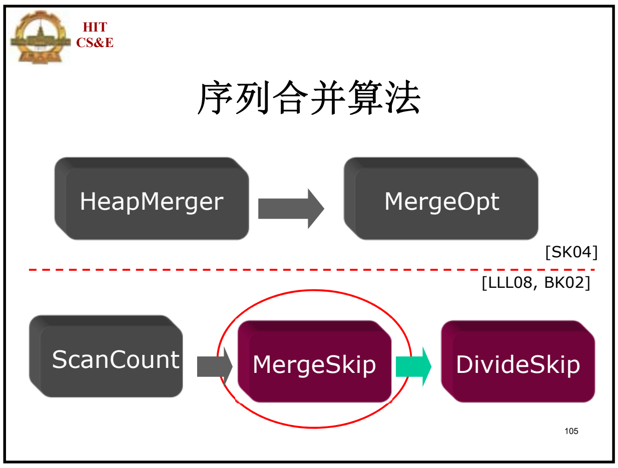

HIT
CS&E
基于堆的算法
Min-heap
1
3
5
10
13
10
13
15
5
7
13
13
15
输出13
使用堆计算每个元素的出现次数
100

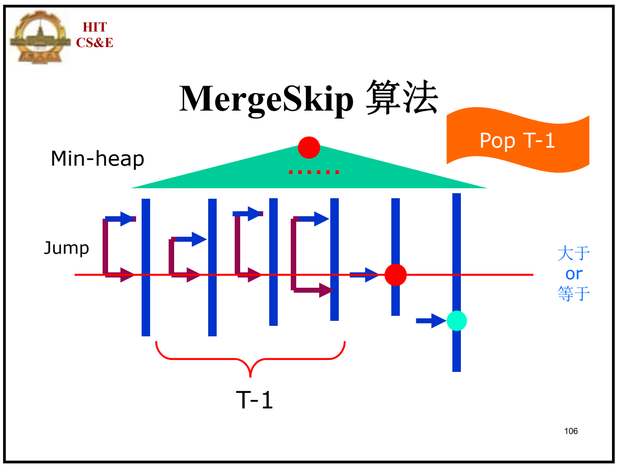

HIT
CS&E
基于堆的算法
Push to
heap
Min-heap
使用堆计算每个元素的出现次数
101

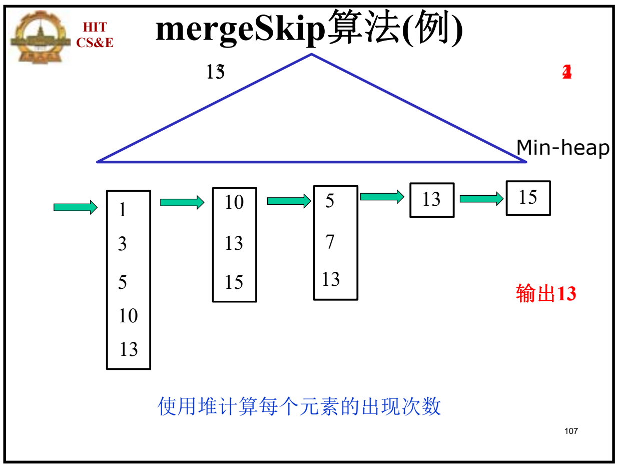

HIT
CS&E
MergeOpt 算法
二分搜索
长序列：T-1
短序列
102

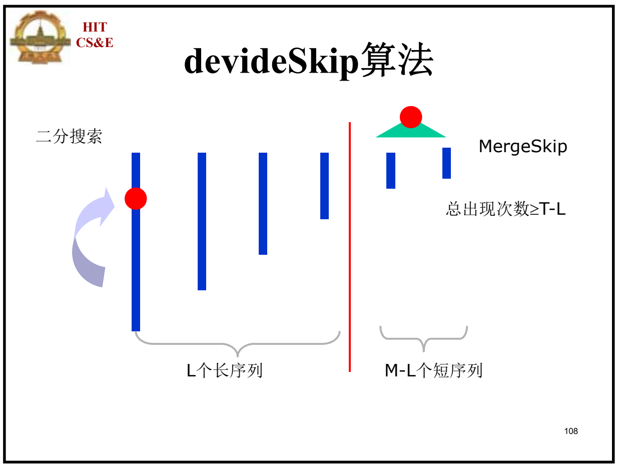

HIT CS&E
mergeOpt算法
Min-heap
1
3
5
10
13
10
13
15
5
7
13
13
15
输出13
4-1=3 Long lists
5-3=2 short lists
使用堆计算每个元素的出现次数
103


HIT CS&E
ScanCount
String ids
出现次数
增加 1
1
2
3
…
13
14
15
01
0
01
4 0
0
2 0
1 10 5 13 15
3 13 7
5 15 13
10
13
结果
阈值 T≥ 4
104


HIT CS&E
序列合并算法
HeapMerger
MergeOpt
[SK04]
[LLL08, BK02]
ScanCount
MergeSkip
DivideSkip
105

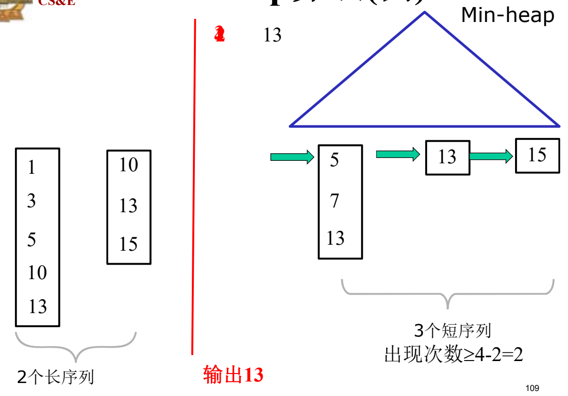

HIT CS&E
MergeSkip 算法
Pop T-1
Min-heap
Jump
大于
or
等于
T-1
106


HIT CS&E
mergeSkip算法(例)
Min-heap
1
3
5
10
13
10
13
15
5
7
13
13
15
输出13
使用堆计算每个元素的出现次数
107


HIT CS&E
devideSkip算法
二分搜索
MergeSkip
总出现次数≥T-L
L个长序列
M-L个短序列
108

HIT CS&E

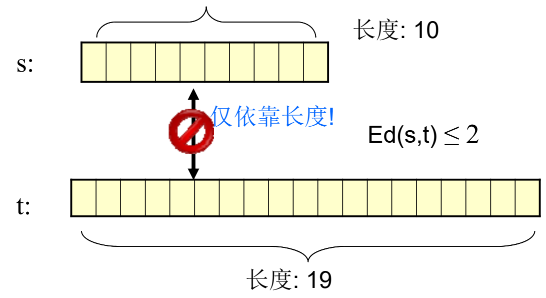
devideSkip算法(例)



109

HIT CS&E

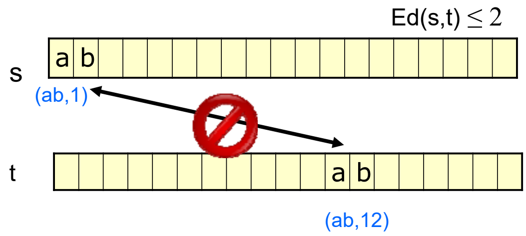
长度过滤



110

HIT CS&E

位置过滤



111

HIT CS&E

保持相似度的集合摘要

- **Shingle** 集合非常大。一个**4 shingle** 集合也是原始文件的**4**倍
- 想办法计算文件的签名（一个较小的文件），通过计算签名的相似性来推断文件之间的相似性。

HIT CS&E

集合的矩阵表示

- 矩阵的列表示各个集合，行表示所有可能的元素
- **S1={a,d},s2={c},s3={b,d,e},s4={a,c,d}**

元素	s1	s2	s3	s4	
a	1	0	0	1	
b	0	0	1	0	
c	0	1	0	1	
d	1	0	1	1	
e	0	0	1	0	

HIT CS&E

最小哈希

- 首先选择行的一个排列变换。
- 任意一列的最小哈希值是在该行排列顺序下第一个列值为1的行的行号。
- **H(S1)=a,H(s2)=c,H(s3)=b,H(s4)=a**

元素	S1	S2	s3	s4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

HIT
CS&E

最小哈希及Jaccard相似度

- 两个集合经随机排列转换之后得到的两个最小哈希值相等的概率等于这两个集合的jaccard相似度
- 假设只考虑s1和s2两个集合对应的列

HIT
CS&E

四种类型

◆给定列C_1和C_2, 行可以如下划分:

	C_1	C_2
a	1	1
b	1	0
c	0	1
d	0	0

◆两列同时为0的行对于这两列的相似性没有贡献。

◆.设x是两列都为1的行的数目

◆ y是其中一行为1的数目

◆注意*Sim* (C_1, C_2) = x /(x +y).

HIT
CS&E

最小哈希

- 对行进行随机转换后，从上往下扫描。遇到两行均为1的概率是x/(x+y)
  - 总共有x+y行，两行均为1的个数有x个。
  - 两行均为1，也就是H(s1)=H(s2)

HIT
CS&E

最小哈希签名

- 对于每一个行排列方式，每一个集合都有一个最小哈希值
- 如果有n个行排列方式（n一般是1百到数百），每个集合就能产生n个最小哈希值，把这些哈希值写成一个列向量。也称为哈希签名。
- 每个集合的列向量组合在一起，写成一个矩阵，称为签名矩阵。
- 文档的相似性就等于最小哈希值相等的概率

HIT
CS&E

最小哈希签名的计算

- 操作矩阵较为困难，操作较小的签名矩阵是可以的
- 用签名矩阵记录行变换的结果，每一个位置只记录当前变换的最小的位置

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.
2. For each column c do the following:
   (a) If c has 0 in row r, do nothing.
   (b) However, if c has 1 in row r, then for each $i = 1, 2, \ldots, n$ set $\text{SIG}(i, c)$ to the smaller of the current value of $\text{SIG}(i, c)$ and $h_i(r)$.

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 3.4: Hash functions computed for the matrix of Fig. 3.2

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 3.4: Hash functions computed for the matrix of Fig. 3.2

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

HIT CS&E

文档的局部敏感哈希算法

- 即使可以使用最小哈希将大文档压缩成小的签名并同时保存任意文档对之间的相似度，寻找相似文档对仍然是不可能的
  – 文档的数目太大
- 实际上，只需要寻找那些相似性大于某个阈值的文档对，而不是全部文档对。这就是局部敏感哈希

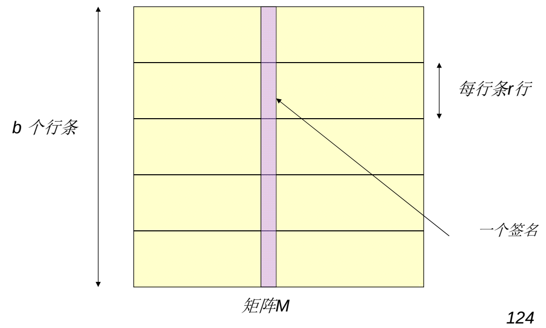
HIT CS&E

文档的局部敏感哈希算法

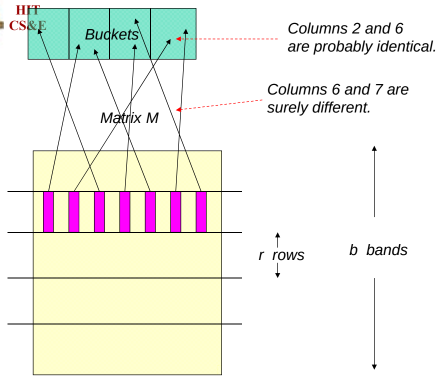
- 假设
  – 文档先表示为shingle集合，通过哈希处理，变为短签名集合
- 基本想法：
  – 把签名矩阵分成子矩阵，使用多次哈希函数。
  – 具有相同部分的列将被哈希到同一个桶中
  – 只考察那些哈希到同一个桶里面的列的相似性

HIT CS&E

行条划分

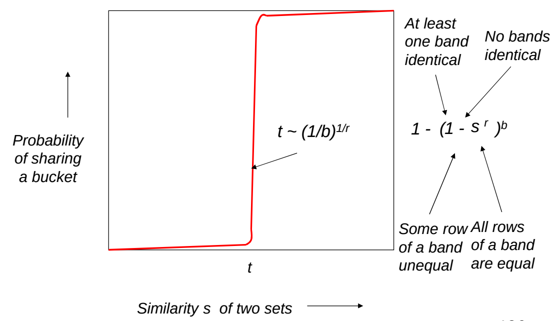


HIT CS&E



HIT CS&E

What b Bands of r Rows Gives You



HIT CS&E

Example: $b = 20$; $r = 5$

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

127

HIT CS&E

LSH小结

- 找出具有相似签名的集合对，删除大部分不相似的集合对
- 在内存里面检查这些候选的集合对

128

HIT CS&E

算法设计永远在路上……

算法课没有结束……

HIT CS&E

当前的研究问题

- 近似算法
- 随机算法
- 图论算法
- 计算几何
- 经济/博弈算法
- 算法复杂性
- 分布式算法

- NP难问题和P问题的近似算法
- 实际应用中的问题
- 经典问题的算法

HIT CS&E

一些最新算法研究结果…..

- A Simpler and Faster Strongly Polynomial Algorithm for Generalized Flow Maximization
- Simple Mechanisms for Subadditive Buyers via Duality
- Faster Space-Efficient Algorithms for Subset Sum and k-Sum
- Homomorphisms Are a Good Basis for Counting Small Subgraphs
- Local Max-Cut in Smoothed Polynomial Time
- Almost-Linear-Time Algorithms for Markov Chains and New Spectral Primitives for Directed Graphs
- Uniform Sampling through the Lovasz Local Lemma
- Sampling Random Spanning Trees Faster than Matrix Multiplication
- Online and Dynamic Algorithms for Set Cover
- Exponential Separations in the Energy Complexity of Leader Election
- Decremental Single-Source Reachability in Planar Digraphs
- A Strongly Polynomial Algorithm for Bimodular Integer Linear Programming