Elegantní algoritmus pro konstrukci sufixových polí

22.10.2014

Obsah

Zadání

Najděte v anglicky psané odborné literatuře článek v délce alespoň 5 stránek o nějakém algoritmu řešícím libovolný problém. Algoritmus popište do českého referátu tak, aby ho podle vašeho názoru pochopil běžný student 2. ročníku informatiky. Váš text musí svědčit o tom, že algoritmu rozumíte, a musí ho z něj pochopit i nezasvěcený čtenář.

Definice

Sufixové pole je datová struktura, která se používá v problematice zpracování řetězců jak pro lingvistické texty, tak pro biologická data. Sufixová pole byla vymyšlena jako paměťově méně náročná verze sufixového stromu.

Sufixové pole se sestává ze seřazených sufixů (přípon) daného řetězce. Přesněji z indexů začátků sufixů. Tedy například pro řetězec „banana$“ (kde „$“ je znak pro konec řetězce, který je jedinečný a lexikograficky menší než ostatní znaky) by se sufixové pole dalo konstruovat následovně (převzato z (2)):

- Nejdříve vytvoříme sufixy:

sufixy	index
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

- Tyto sufixy poté lexikograficky seřadíme:

sufixy	index
$	1
a$	2
ana$	3
anana$	4
banana$	5
na$	6
nana$	7

- Sufixové pole by poté vypadalo takto:

index	1	2	3	4	5	6	7
sufixy	7	6	4	2	1	5	3
	$	a $	a n a $	a n a n a $	b a n a n a $	n a $	n a n a $

Analýza problému

Originální algoritmus dosahuje složitosti $O(n \log n)$. Je založen na technice zvané „prefix doubling“. Tato metoda je založena na postupném řazení podle délky prefixů. Přesněji řečeno, nejdříve roztřídíme sufixy do jednotlivých skupin podle jejich prvního znaku. Poté každou skupinu seřadíme podle prefixu dvojnásobné délky, dokud nebude velikost každé skupiny rovna jedné. Těchto kroků nebude více než $\log n$ a zároveň žádné řazení netrvá déle než $O(n)$, proto je složitost řešení $O(n \log n)$.

V roce 2003 tři nezávislé skupiny vynalezli první lineární řešení, které nevyžaduje předchozí vytvoření sufixového stromu. Jeden z nich například nazývaný „skew“ nejdříve seřadí sufixy i, kde $i \; mod \; 3 \neq 0$ pomocí rekurze. Pořadí těchto sufixů je poté použito pro odvození sufixů, kde $i \; mod \; 3 = 0$. Jakmile jsou vytvořeny tyto dvě skupiny, lze porovnat sufixy z první skupiny se sufixy z druhé v konstantním čase. Poslední krok je spojení těchto dvou skupin v lineárním čase.

Nicméně algoritmus s nejlepšími výsledky v praxi, algoritmus „BPR“, dosahuje asymptotické složitosti $O(n^2)$. Tento algoritmus nejdříve seřadí sufixy do určité hloubky, poté se zaměří na jednu skupinu a tu opakovaně seřadí do podskupin.

V tomto referátu se však budeme zabývat elegantním algoritmem pro konstrukci sufixových polí, který s velkou pravděpodobností pracuje v lineárním čase, kde ona pravděpodobnost je v poli všech možných vstupů.

Jednotlivé algoritmy

Algoritmus SA1

Princip prvního algoritmu lze popsat pseudokódem následovně (převzato z (1)):

Algorithm SA1

Sort $P_1, P_2, \ldots, P_n$ using radix sort;
The above sorting partitions the P_i's into buckets where equal ℓ-mers are in the same bucket;
Let these buckets be $B_1, B_2, \ldots, B_m$ where $m \leq n$;
for $i := 1$ **to** m **do**
    if $|B_i| > 1$ **then** sort the suffixes corresponding to the ℓ-mers in B_i using any relevant algorithm;

Pseudokód algoritmu SA1

Princip je založen na tom, že algoritmus řadí sufixy pouze podle jejich l-meru, což jsou podřetězce o velikosti l. Celý tento důvtip lze použít, jelikož jeli $l \geq \log_6 n$, kde n je počet písmen slova a б je počet písmen v abecedě, je s velkou pravděpodobností dosaženo stavu, kde jsou všechny skupiny o velikosti jedna a v tu chvíli již je připravené sufixové pole. Přesný důkaz je v (1).

Další důležitou částí je radix sort. Tento řadící algoritmus pracuje v $O(n)$ nebo přesněji v $O(m * C(n))$, m je počet znaků řazených řetězců, n je velikost dat $C(n)$ je složitost vnitřního stabilního řadícího algoritmu. Pseudokód tohoto řadícího algoritmu vypadá následovně (převzato z (3)):

```
1. function radixSort(String s)
2.     for i in (s.length - 1)  -> 0 do
3.         stableSort(s[i])
```

Pseudokód radix sortu

Princip radix sortu (popsaný v (3)) vychází přímo z definice stabilního řazení – řadicí algoritmus je stabilní, pokud zachovává pořadí klíčů, které mají stejnou hodnotu (tj. pokud struktury seřadíme napřed dle klíče A, poté podle klíče B, tak jsou seřazeny podle B, a kde jsou si hodnoty B rovny, tam jsou struktury v pořadí daném klíčem A).

Neboli radix sort se nesnaží seřadit data najednou, nýbrž pouze rozřazuje do skupin postupně jejich části (nejčastěji od nejméně významné pozice). Každý takový průchod je $O(n)$.

Algoritmus SA2

Je možné mít také algoritmus, který použije jiný algoritmus za předpokladu, že velikost některé ze skupin je větší než jedna. Příklad pseudokódu (převzato z (1)):

Algorithm SA2

1 Sort $P_1, P_2, \ldots, P_n$ using radix sort;
The above sorting partitions the P_i's into buckets where equal ℓ-mers are in the same bucket;
Let these buckets be $B_1, B_2, \ldots, B_m$ where $m \leq n$;
2 **if** $|B_i| = 1$ for each i, $1 \leq i \leq m$
3 **then** output the suffix array and quit;
4 **else** use another algorithm (let it be **Algorithm SA**) to find and output the suffix array;

Pseudokód algoritmu SA2

Jak algoritmus SA1, tak algoritmus SA2 pracují v $O(n)$ alespoň v $(1 - n^{-\alpha})$ případech všech možných vstupů, kde α je parametr pravděpodobnosti (typicky konstanta ≥ 1). Také pokud algoritmus SA je $t(n)$, pak očekávaná složitost algoritmu SA2 je $(1 - n^{-\alpha})O(n) + n^{-\alpha}(O(n) + t(n))$. Například pokud bude algoritmus SA algoritmus „skew“, složitost algoritmu SA2 je stále $O(n)$.

Oba algoritmy (SA1 a SA2) pracují s jakoukoliv velikosti abecedy. Přístupy se liší pouze ve volbě velikosti l-meru.

Algoritmus RadixSA

```
Algorithm RadixSA
1.  radixSort all suffixes by d characters
2.  let b[i] = bucket of suffix i
3.  for i := n down to 1 do
4.      if (b[i].size > 1) then
5.          if detectPeriods(b[i]) then
6.              handlePeriods(b[i]);
7.          else radixSort all suffixes j ∈ b[i] with respect to b[j + d]
```

Pseudokód algoritmu RadixSA

Ve výše znázorněném pseudokódu (převzato z (1)) nás mohou zarazit řádky 5 a 6, kde se řeší periody. V tomto kódu jde o to, že pokud jsou sufixy začínající na indexu i, i+p i+2p,… ve stejné skupině b a skupina b je momentálně seřazená podle $d \geq p$ znaků je jasné, že byly nalezeny řetězce, kde je perioda P o délce p. A tedy dané sufixy mají formu $PS_{i+p+1\ldots n}$, $PPS_{i+p+1\ldots n}$, $PPPS_{i+p+1\ldots n}$, atd. Tento problém lze řešit tak, že se úseky P považují za jedno písmeno.

Nicméně algoritmus RadixSA pracuje v $O(n\sqrt{n})$. Tato složitost lze zlepšit na $O(n \log n)$. A to tak, že pokud je během smyčky `for` nějaká skupina navštívena více než C-krát, pak se tato skupina přeskočí. To znamená, že smyčka `for` zabere lineární čas. Na konci pokud existuje nějaká přeskočená skupina, vrátíme se k ní. Další průchod každé skupiny s velikostí více jak 1 používá alespoň C+1-krát větší hloubku prohledávání než předchozí průchod. Což ve výsledku znamená, že složitost algoritmu je $O(n \log n)$.

Detaily k implementaci si je možné prohlídnout v (1) v části „Practical Implementation“.

Experimentální výsledky

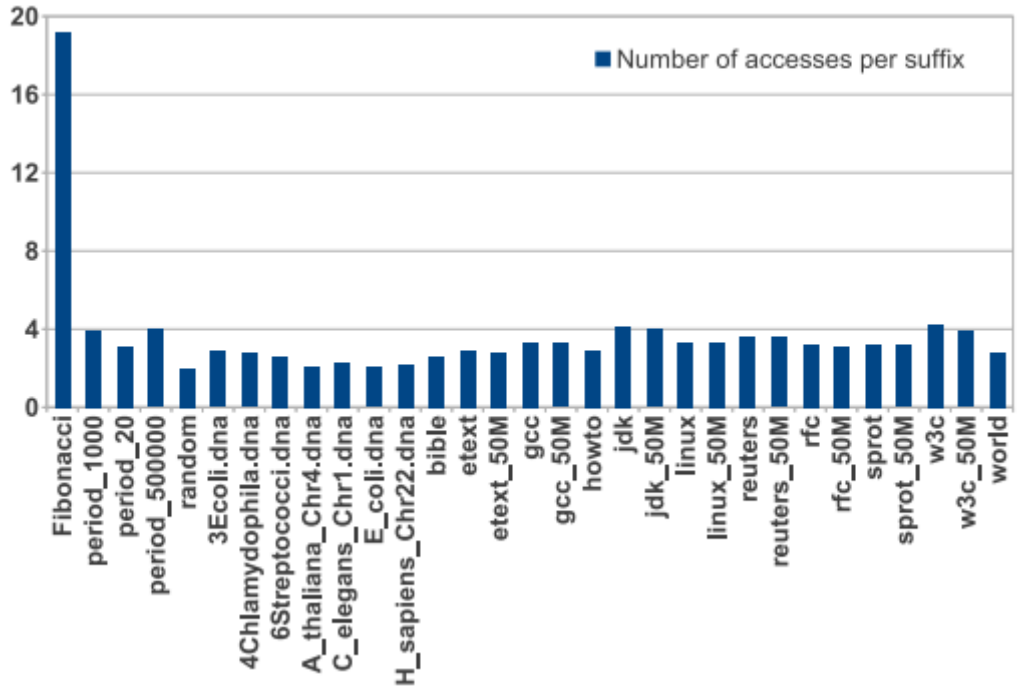
Autoři porovnali algoritmus RadixSA s jinými algoritmy pro konstrukci sufixových polí s velmi dobrými výsledky. Viz table2 převzatá z (1):

Table 2
Comparison of run times on datasets from [25] on a 64-bit Intel CORE i3 machine with 4 GB of RAM, Ubuntu 11.10 Operating System, Sun Java 1.6.0_26 and gcc 4.6.1. Run times are in seconds, averaged over 10 runs. Bold font indicates the best time. ML means out of memory, TL means more than 1 hour.

Dataset			Run time							
Name	Length	$\|\Sigma\|$	RadixSA	BPR2	BPR.9	DivSuf Sort	QSuf Sort	SAIS	skew	Deep Shallow
Fibonacci	20 000 000	2	7.88	12.48	14.05	6.81	26.44	**5.50**	14.53	369.48
period_1000	20 000 000	26	**2.12**	3.52	5.71	3.15	20.42	6.59	23.27	TL
period_20	20 000 000	17	**1.44**	1.95	43.39	1.83	11.05	2.83	7.15	TL
period_500000	20 000 000	26	**2.78**	4.60	6.31	4.74	23.32	8.56	25.68	2844.37
random	20 000 000	26	**2.25**	3.34	4.87	6.35	5.02	11.75	22.05	5.69
3Ecoli.dna	14 776 363	5	**2.23**	2.67	3.43	4.00	13.85	6.14	19.62	433.54
4Chlamydophila.dna	4 856 123	6	**0.61**	0.67	0.90	1.71	3.24	1.93	5.24	4.80
6Streptococci.dna	11 635 882	5	**1.63**	1.79	2.38	2.88	7.08	4.98	14.88	4.26
A_thaliana_Chr4.dna	12 061 490	7	**1.27**	1.74	2.40	3.02	5.13	5.37	15.71	3.52
C_elegans_Chr1.dna	14 188 020	5	**1.61**	1.95	2.65	3.21	6.91	5.69	17.18	6.92
E_coli.dna	4 638 690	4	**0.41**	0.51	0.58	1.36	1.72	1.96	5.04	1.37
H_sapiens_Chr22.dna	34 553 758	5	**4.40**	5.66	8.21	7.76	15.31	15.59	49.70	10.98
bible	4 047 391	63	0.51	**0.48**	0.80	1.24	1.38	1.56	4.64	1.08
etext	105 277 339	146	**19.40**	23.09	43.46	26.56	62.63	54.70	ML	119.96
etext_50M	50 000 000	120	**8.13**	9.74	17.26	11.94	26.40	24.46	88.57	79.07
gcc	86 630 400	150	**13.84**	15.58	24.50	15.84	46.20	33.62	135.12	80.78
gcc_50M	50 000 000	121	**7.21**	9.56	13.26	8.31	28.43	17.73	68.65	264.90
howto	39 422 104	197	**5.96**	6.35	10.26	8.41	17.64	16.67	64.73	16.33
jdk	69 728 898	113	**12.07**	12.54	26.86	12.74	39.92	24.66	102.76	58.22
jdk_50M	50 000 000	110	**8.32**	**8.30**	17.05	8.91	26.30	17.58	71.31	36.98
linux	116 254 720	256	**19.27**	**19.34**	29.67	21.17	61.99	44.47	ML	58.71
linux_50M	50 000 000	256	**7.62**	**7.60**	10.50	8.84	27.54	18.18	76.10	31.92
reuters	114 711 150	93	**19.76**	25.08	60.72	25.07	74.78	49.17	ML	87.57
reuters_50M	50 000 000	91	**7.84**	9.53	20.41	10.24	26.94	20.29	77.25	33.68
rfc	116 421 900	120	**21.18**	22.08	42.75	22.55	66.28	47.99	ML	42.14
rfc_50M	50 000 000	110	**8.23**	8.39	14.85	9.24	24.80	19.61	76.64	16.63
sprot	109 617 186	66	**18.48**	22.79	47.07	25.52	69.58	50.40	ML	48.69
sprot_50M	50 000 000	66	**7.57**	9.10	16.81	10.88	28.07	21.69	78.47	20.03
w3c	104 201 578	256	**18.82**	**18.78**	35.94	20.01	74.09	38.29	ML	1964.80
w3c_50M	50 000 000	255	**7.93**	8.33	17.67	8.73	25.95	17.26	71.42	36.59
world	2 473 399	94	0.30	**0.27**	0.42	0.91	0.86	0.91	2.35	0.78

Nejlepší výsledky jsou zvýrazněny tučně.

Pro každé data byl každý algoritmus spuštěn 10-krát. Navíc autoři počítali u algoritmu RadixSA průměrný počet přístupů do každého sufixu. Převzato z (1):



Jak je vidět algoritmus obstál na výbornou s výjimkou Fibonacciho řetězce.

Závěr

Výše zmíněné algoritmy pro konstrukci sufixových polí jsou založené hlavně na předpokladu, že stačí seřadit sufixy pouze podle několika znaků, což funguje s velkou pravděpodobností a také na radix sortu, který pracuje v $O(n)$ a rozřazuje jednotlivé sufixy do skupin, z kterých lze poté sestavit sufixové pole.

Na konec autoři ukázali, že další nadstavba algoritmu, nazývaná RadixSA, dosahuje lepších výsledků než jiné známé algoritmy s výjimkou Fibonacciho řetězce.

Bibliografie

1. S. Rajasekaran, M. Nicolae / Journal of Discrete Algorithms 27 (2014) 21–28 [Online] http://www.sciencedirect.com/science/article/pii/S1570866714000173
2. Suffix array, Wikipedia, The Free Encyclopedia [Online] http://en.wikipedia.org/wiki/Suffix_array
3. Pavel Mička, Radix sort, Algoritmy.net [Online] http://www.algoritmy.net/article/109/Radix-sort