Sufixové polia

©kuko 24.11.2020

1 Úvod

- sufixové pole (SA) je jednoducho pole sufixov utriedené lexikograficky
- motivácia: sufixové stromy (ST) zaberajú príšerne veľa pamäte – aj keď si dáme pozor, 10–20B/znak
- sufixové polia potrebujú 1 int/znak; ak máme text do 4miliárd znakov, môžeme použiť 32bitový int, čo je 4B/znak + samotný text
- vezmime si napríklad ľudský genóm, čo je reťazec asi 3miliardy znakov nad abecedou `A`, `C`, `G`, `T`
- samotný string teda zaberie asi 3GB (ak použijeme 1B/znak), alebo 750MB, ak použijeme zhustenú reprezentáciu a 2bity/znak
- sufixový strom bude zaberať 30–60GB a sufixové pole asi 12GB (+samotný reťazec 0.75GB)
- a to je len pamäť výslednej štruktúry, kde nepočítame pamäť použitú dočasne počas konštrukcie
- pri spracovaní väčších vstupov nás teda bude limitovať veľkosť RAM

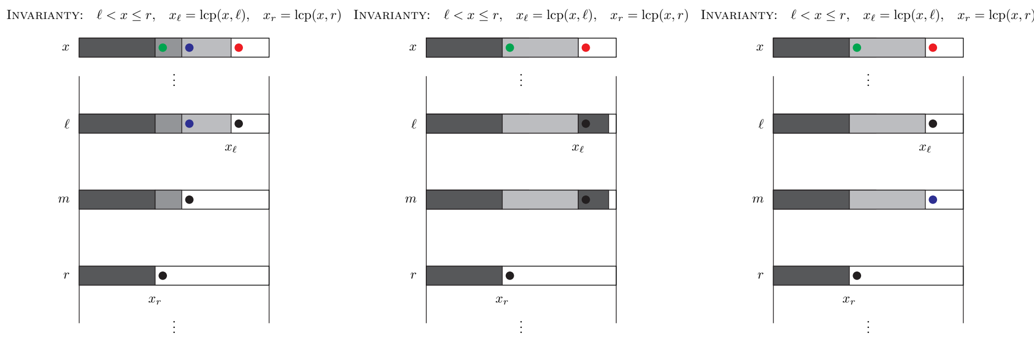
2 Vyhľadávanie

- binárne vyhľadávanie: $O(m \log n)$ (horšie ako ST – $O(m)$)
- sufixy začínajúce na P tvoria jeden súvislý úsek v SA
- dá sa zlepšiť na $O(m + \log n)$ za cenu väčšej pamäte:
- nech $\mathrm{lcp}(i, j)$ je najdlhší spoločný prefix i-teho a j-teho sufixu v poradí
- nápad 1: ak horný a dolný odhad majú lcp > 0, tieto znaky môžeme preskočiť (nestačí)
- označme x hľadaný text, sufixy ℓ, r – dolná a horná hranica
- invariant: $\ell < x \le r$, $x_\ell = \mathrm{lcp}(x, \ell)$, $x_r = \mathrm{lcp}(x, r)$
- tzn. x_ℓ a x_r je počet písmen zo začiatku, kde sa x a ℓ, resp. x a r zhodujú (pozri šedé úseky na obr.); $\ell[x_\ell] < x[x_\ell]$ (červený znak) a $x[x_r] < r[x_r]$ (zelený znak)
- BUNV nech $x_\ell > x_r$, pozrime sa na prostredný sufix m; aké je $p = \mathrm{lcp}(\ell, m)$?
  - ak $p < x_\ell$ (obr. vľavo), tak to znamená, že spoločný prefix $\mathrm{lcp}(\ell, m)$ je kratší ako spoločný prefix $\mathrm{lcp}(x, \ell)$; zároveň $\ell < m$, tzn. $\ell[p] < \ell[m]$; ale $\ell[p] = x[p]$, pretože ich lcp je dlhšie (modrý znak na obr. vľavo je rovnaký v x a ℓ a menší ako čierny znak v m); z toho vyplýva $x < m$ a v konštantom čase sme zistili, že treba pokračovať v prvej polovici
  - naopak, ak $p > x_\ell$ (obr. v strede), tak ℓ a m majú viac spoločných znakov ako x_ℓ a konkrétne teda aj x_ℓ-tý znak, v ktorom sa ℓ a x líšia; z toho vyplýva $x > m$ a treba hľadať v druhej polovici (opäť čas $O(1)$)
  - iba ak $p = x_\ell$ (obr. vpravo), nevieme rozhodnúť hneď – v tomto prípade začneme porovnávať znaky (od pozície p) a rozhodneme sa podľa toho; v každom prípade nám stúpne $\max(x_\ell, x_r)$, takže takýchto porovnaní môže byť najviac m
- ak $x_\ell \le x_r$, postupujeme symetricky (porovnáme $p = \mathrm{lcp}(m, r)$)

3 LCP

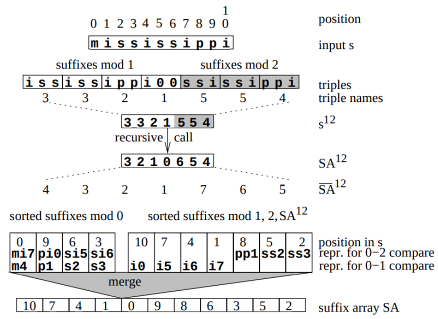
4 Konštrukcia

- qsort – $O(n^2 \log n)$ (mehh)
- radix sort – $O(n^2)$ (mehh)
- Manber–Myers: sufix sufixu je sufix!



  - ak máme pole utriedené podľa prvých K písmen, vieme ho ľahko zotriediť podľa prvých $2K$ písmen
  - budeme mať $\log n$ fáz; v k-tej fáze triedime podľa prvých 2^k písmen
  - nech $\mathrm{rank}[i] = j$, ak je sufix s_i v abecednom poradí j-ty (podľa prvých 2^k písmen; ak majú dva sufixy rovnakých prvých 2^k písmen, ranky budú rovnaké)
  - fáza: stačí zotriediť trojice $(\mathrm{rank}[i], \mathrm{rank}[i+2^k], i)$
- ešte lepšie? áno, existuje lineárna konštrukcia:

- rozdelíme na sufixy na pozíciach nedeliteľných vs. deliteľných 3
- rekurzívne utriedime pozície $\equiv 1, 2 \pmod 3$ (a spočítame si pre každý sufix pozíciu v utriedenom poli)
- keď to máme, pozície deliteľné 3 utriedime tak, že ödrolujeme"jedno písmenko a pozrieme sa na poradie sufixov nedeliteľných 3 – radix sortom utriedime dvojice (prvé písmeno, pozícia v o 1 kratšieho sufixu v už utriedenom poli)
- teraz máme dve utriedené polia (sufixy na pozíciach deliteľných a nedeliteľných 3) – tie zmergeujeme klasickým algoritmom s tým, že porovnanie bude v $O(1)$:
  - ak porovnávame sufixy na pozíciach 1 vs. $0 \pmod 3$, ödrolujeme"jedno písmenko a dostaneme pozície 2 vs. 1 (oba sufixy sú v utriedenom poli, takže v $O(1)$ vieme zistiť, ktorý je skôr)
  - ak porovnávame sufixy na pozíciach 2 vs. $0 \pmod 3$, ödrolujeme"dve písmená a dostaneme pozície 1 vs. $2 \pmod 3$ (opäť oba zvyšky sú v utriedenom poli)
- výsledná zložitosť: $T(n) = T(\frac{2}{3}n) + O(n)$ kde $T(\frac{2}{3}n)$ je čas rekurzívneho volania a $O(n)$ je triedenie pozícií deliteľných 3 a merge-ovanie; táto rekurencia má riešenie $O(n)$
- pozor, rekurzívne volanie treba upraviť – potrebujeme sa zavolať na tú istú úlohu ("spočítať SA pre vybrané pozície"nie je tá istá úloha)
- finta: zoberieme pôvodný string od 1. písmena,
- ak budeme každú trojicu znakov považovať za 1 písmeno, tak sufixy tohto stringu zodpovedajú sufixom na poz. $\equiv 0, 1 \pmod 3$ v pôvodnom stringu
- nový problém: veľká abeceda; riešenie: reťazec dĺžky n môže obsahovať najviac n rôznych písmen, t.j. stačí znaky zotriediť a prečíslovať (v $O(n)$)



5 Praktické algoritmy a implementácia

- qSufSort - $8n$ bytes (Larsson and K. Sadakane. Faster suffix sorting)
  - doubling technika Manbera a Myersa
  - Bentley–McIlroyov ternárny quicksort (delíme na $<$, $=$, $>$)
- Seward copy [30]
  - utriedime podľa prvých 2 písmen (buckety $B_{\alpha\beta}$)
  - následne triedime od najkratších bucketov po najdlhšie, keď je bucket hotový, prejdeme ho celý, pričom pre každý sufix i pozrieme na predošlé písmeno a sufix T_{i-1} hodíme na začiatok bucketu $B_{T[i-1]T[i]}$
  - pre každé B_α si necháme $B_{\alpha\alpha}$ na koniec a poradie sufixov odvodíme už od zotriedených
- Ferragina–Manzini (deep-shallow DS2)
  - multi-key quicksort rozdelíme podľa prvého písmena na menšie, rovné a väčšie; rekurzívne triedime utriedime všetky časti, pričom v strednej už neporovnávame prvé písmeno (Bentley–Sedgewick, viď `https://en.wikipedia.org/wiki/Multi-key_quicksort`)
  - rekurzívne delíme, kým nemáme interval, kde majú všetky reťazce spoločný prefix dĺžky aspoň L
  - ak je interval malý (menej ako n/Q, kde $Q \geq 1000$), použijeme "blind sort"(naházdžeme do písmenkového stromu a prejdeme zľava doprava)
  - veľké intervaly sa triedia pomocou TSQS
  - ešte predtým generalized induced copying – ak dve písmená $\leq L$ nájdeme v už utriedenom buckete, skopírujeme z neho správne poradie:
  - 1) utriedime interval podľa pozícií, aby sme vedeli vyhľadávať, 2) nájdeme prvý sufix intervalu v buckete (s pomocnou pamäťou sa dá rýchlo) 3) odtiaľ postupujeme doľava a doprava a značíme si, ktoré sufixy bucketu patria do nášho intervalu 4) skopírujeme správne poradie
- SA-IS (induced sorting; Nong et. al) – $O(n)$
  - označíme sufixy $\nearrow$ ak $T_{i+1} > T_i$ a $\searrow$ ak $T_{i+1} < T_i$; $\nearrow$-sufix pred ktorým je $\searrow$-sufix budeme značiť $\searrow\!\!\nearrow$ – "dolinka"
  - text rozdelíme na podreťazce ohraničené dolinkami (tie začínajú $\searrow\!\!\nearrow$, potom niekoľko $\nearrow$, potom niekoľko $\searrow$ a končia $\searrow\!\!\nearrow$) – "kopčeky"
  - kopčeky zotriedime (pričom ak je jeden prefix druhého, možno na konci rozhodne $\searrow$ vs. $\nearrow$)
  - T' vytvoríme z poradia kopčekovitých podreťazcov, ak poradie nie je jednoznačné, rekurzívne zostrojíme SA T'
  - keď máme $SA(T')$, vieme poradie dolinkových sufixov, ktoré nám pomôžu utriediť zvyšok
  - sufixy môžeme rozdeliť podľa 1. písmena + šípky ($\searrow$ bude pred $\nearrow$)
  - 0) init: $SA[i] = -1$, na koniec každého bucketu prídu $\searrow\!\!\nearrow$-sufixy v poradí podľa $SA(T')$
  - 1) zotriedime $\searrow$-sufixy: ideme cez SA, ak $SA[i] \neq -1$ a $T_{SA[i]-1}$ je $\searrow$, doplníme $SA[i] - 1$ na začiatok príslušného bucketu
  - 2) $\nearrow$-sufixy: ideme sprava doľava, pre každé $SA[i] \neq -1$ a predchádzajúce písmeno je $\nearrow$, doplníme $SA[i] - 1$ na koniec príslušného bucketu

- idea: ak T_i a T_j sú dva $\searrow$-sufixy začínajúce rovnakým písmenom, potom $T_i < T_j \iff T_{i+1} < T_{j+1}$ takto by sme mohli porovnávať znaky, kým neprídeme na pozíciu $T[i+k] \neq T[j+k]$, alebo aspoň jedno z $T[i+k], T[j+k]$ je $\nearrow$ – ak je práve jeden $\nearrow$-sufix, tak je zjavne neskôr; ak sú oba $\nearrow$, sú to dolinky a tie máme zotriedené v $SA(T')$
- extra $2n$ bytov pamäť (v rekurzií môže byť veľká abeceda → až $n/2$ bucket pointrov; v skutočnosti $SA(T')$ sa môže vypĺňať priamo do $SA(T)$)

- divsufsort – `https://github.com/y-256/libdivsufsort`
  - čiastočne paralelizovaná verzia
- pozri `https://github.com/sacabench/sacabench`

- optimalizačné stratégie:
- 1) menšia pamäť (32? 40? 64-bitov? bitvektor, prípadne si ukradneme bit z iného poľa)
- 2) cache-efektivita
- 3) spracujeme viac znakov naraz (64-bitový int je 8 znakov; až 512-bitové AVX registre = 64 bytov)

- algoritmy pre externú pamäť?
- paralelné algoritmy?
- GPU prefix doubler