Úsporné dátové štruktúry

©kuko 3.5.2018

1 Úsporné dátové štruktúry

- budeme sa venovať DŠ, ktoré zaberajú málo miesta v pamäti
- presnejšie, nech OPT je minimálny počet bitov potrebný na reprezentáciu štruktúry
- hovoríme, že DŠ je
  - *kompaktná*, ak zaberá $O(\mathrm{OPT})$ bitov
  - *úsporná*, ak zaberá $\mathrm{OPT} + o(\mathrm{OPT})$ bitov
  - *implicitná*, ak zaberá $\mathrm{OPT} + O(1)$ bitov
- napríklad
  - na n-bitový reťazec treba $\mathrm{OPT} = n$ bitov;
  - na k-prvkovú podmnožinu z n treba $\mathrm{OPT} = \lg \binom{n}{k}$ bitov;
  - n-vrcholových binárnych stromov je $C_n = \frac{1}{n+1}\binom{2n}{n} \sim 4^n/n^{3/2}\sqrt{\pi}$, takže na binárny strom treba $\lg C_n = 2n - \Theta(\lg n)$ bitov

2 Rank&Select

- problém: daný reťazec S dĺžky n, chceme podporovať operácie:
  - $\mathrm{rank}_c(S, i)$ = počet znakov c od začiatku po i-tu pozíciu
  - $\mathrm{select}_c(S, i)$ = pozícia i-teho znaku c v S
- zjavne $\mathrm{rank}_c(S, \mathrm{select}_c(S, i)) = i$ a $\mathrm{select}_c(S, \mathrm{rank}_c(S, i)) \leq i$ je posledná pozícia znaku c na pozícii $\leq i$ (rovnosť nastáva, ak $S_i = c$)
- uvažujme najskôr binárnu abecedu; nech m je počet jednotiek
- $\mathrm{rank}_0(S, i) = i - \mathrm{rank}_1(S, i)$, takže stačí podporovať jednu operáciu
- jednoduché riešenia:
  - utriedené pole – $m \lg n$ bitov, $O(\lg m)$ čas
  - predpočítaný rank – $n \lg n$ bitov, $O(1)$ čas
  - predpočítaný rank pre blok dĺžky t – $n + (n/t) \lg n$ bitov, čas $O(t)$; pre $t = \Omega(\lg n)$ dostávame kompaktnú, pre $t = \omega(\lg n)$ úspornú DŠ, ale čas je $\Omega(\lg n)$

3 Úsporný Rank

- úsporné riešenie v čase $O(1)$ (pre binárnu abecedu) dostaneme na 3 kroky:
- rozdelíme bitvektor na superbloky dĺžky $t_1 = \lg^2 n$ a predpočítame rank pre každý superblok – $O(n/\lg n) = o(n)$ bitov
- každý superblok rozdelíme na bloky dĺžky $t_2 = \frac{1}{2}\lg n$ a predpočítame rank každého bloku v *rámci superbloku*
- blokov je $O(n/\lg n)$, ale keďže každý rank je od začiatku superbloku, t.j. má hodnotu najviac $\lg n$, stačí na reprezentáciu $O(\lg \lg n)$ bitov – spolu $O(n \lg \lg n/\lg n) = o(n)$
- nakoniec $\forall$ bitvektor dĺžky $\frac{1}{2}\lg n$ (tých je len $2^{\frac{1}{2}\lg n} = \sqrt{n}$) a $\forall$ pozíciu predpočítame jednu „globálnu“ tabuľku s výsledkami – tá má veľkosť $O(\underbrace{\sqrt{n}}_{\#\text{bitvektorov}} \cdot \underbrace{\lg n}_{\#\text{pozícií}} \cdot \underbrace{\lg \lg n}_{\text{veľkosťvýsledku}}) = o(n)$
- $\mathrm{rank}_1(S, i)$ dostaneme tak, že spočítame rank pre superblok + rank pre blok + rank z tabuľky ($O(1)$)
- pamäť je n bitov na bitvektor + $o(n)$ navyše

- v praxi toto riešenie nie je najrýchlejšie, pretože v najhoršom prípade máme 3 cache missy

- namiesto tabuľky spočítame rank v rámci bloku pomocou inštrukcie popcnt, ktorú majú všetky moderné procesory
- dobrý kompromis medzi časom a pamäťou je 1-úrovňové riešenie (bez superblokov):
  - rozdelíme bitvektor na bloky po $7 \cdot 64 = 448$bitov
  - medzi tieto bloky pridáme predpočítané ranky – jeden 64-bitový int
  - celkovo teda rank+1blok zaberie 64B, čo je 1 cache line (max 1 cache miss)
  - toto riešenie zaberá 14% pamäte navyše oproti samotnému bitvektoru

4 Úsporný Select

- podobná metóda ako pri ranku, hoci riešenie je trochu zložitejšie
- predpočítame $\text{select}(S, i)$ pre všetky násobky $t_1 = \lg n \lg\lg n$ (pamäť $O(n/\lg\lg n)$)
- to nám rozdelí bitvektor na superbloky rôznych veľkostí
- nech r je veľkosť jedného superbloku; rozlišujeme 2 možnosti:
  - pre veľké a riedke superbloky ($r \geq t_1^2$) zapíšeme priamo pozície jednotiek
  - môžeme si to dovoliť, pretože takýchto veľkých a riedkych superblokov je málo a majú málo jednotiek: pamäť bude $O(\underbrace{n/t_1^2}_{\#\text{superblokov}} \cdot \underbrace{t_1}_{\#\text{jednotiek}} \cdot \underbrace{\lg n}_{\text{zápis1pozície}}) = O(n/\lg\lg n)$ bitov
  - ostáva nám vyriešiť malé superbloky ($r < t_1^2$)
- použijeme ten istý prístup s $t_2 = (\lg\lg n)^2$:
- predpočítame select pre násobky t_2 v rámci malého superbloku, t.j. každá pozícia zaberá len $\lg\lg n$ bitov, spolu $O(n/t_2 \cdot \lg\lg n) = O(n/\lg\lg n)$
- pre veľké a riedke bloky ($r' \geq t_2^2$) uložíme priamo pozície všetkých jednotiek; jedna pozícia zaberá $\lg t_1^2 = O(\lg\lg n)$ bitov – spolu $O(n/t_2^2 \cdot t_2 \cdot \lg\lg n) = O(n/\lg\lg n)$
- blokov veľkosti $< t_2^2 = (\lg\lg n)^4$ je veľmi málo, takže stačí globálna tabuľka s predpočítanými hodnotami select $\forall$ bitvektor $\forall$ pozíciu

- nie je známy vzťah medzi select_0 a select_1 ako pre rank
- v praxi je najjednoduchšie (a často postačujúce) použiť binárne vyhľadávanie cez rank

5 Komprimovaný bitvektor a RRR

- entropia $H(S) = \sum_a \frac{n_a}{n} \lg \frac{n}{n_a}$
- riešenie je rovnaké ako pri úspornom ranku, bitvektor rozdelíme na bloky a superbloky
- jediný rozdiel je, že každý blok skomprimujeme:
- namiesto celého bitvektoru dĺžky t_2 uložíme dvojicu (c, o), kde
  - c je #jednotkových bitov a
  - o je offset – poradie bitvektoru, ak by sme všetkých $\binom{t_2}{c}$ bitvektorov s c jednotkami zotriedili lexikograficky
- dvojica (c, o) je index do predpočítanej globálnej tabuľky rankov
- c zaberá $\lg t_2$ bitov, o $\lg \binom{t_2}{c}$ bitov
- nech c_i je počet jednotiek v i-tom bloku, $n_1 = \sum c_i$
- potom offsety zaberajú $\sum_i \lceil \lg \binom{t_2}{c_i} \rceil \leq o(n) + \sum_i \lg \binom{t_2}{c_i} = o(n) + \lg \prod_i \binom{t_2}{c_i} \leq o(n) + \lg \binom{n}{n_1}$
- $\lg \binom{n}{n_1} = \lg n! - \lg n_1! - \lg n_0! = n \lg n - n - n_1 \lg n_1 + n_1 - n_0 \lg n_0 + n_0 + O(\lg n)$
  $= n_1 \lg \frac{n}{n_1} + n_0 \lg \frac{n}{n_0} + O(\log n) = nH(S) + O(\log n)$
- keďže komprimované bloky nemajú rovnakú dĺžku, potrebujeme ešte pole pointrov – kde začína i-ty superblok a kde začína j-ty blok (v rámci superbloku) – toto zaberá $o(n)$ pamäte
- pre každé $c \in \{0, \ldots, t_2\}$ tiež predpočítame hodnotu $\lceil \lg \binom{t_2}{c} \rceil$, čo je dĺžka zápisu offsetu

- v praxi je opäť výhodné namiesto 2-och úrovní (bloky a superbloky) zvoliť len jednu úroveň (kvôli cache missom) a namiesto tabuľky, kde sú predpočítané ranky pre každú pozíciu mať tabuľku s pôvodnými bitvektormi a výsledok zrátať pomocou popcnt

6 Malá abeceda

- ako rozšíriť riešenie pre binárnu abecedou na malú abecedu, napr. DNA=$\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$?
- jedna možnosť je vyrobiť jeden bitvektor pre každý symbol zvlášť – takéto riešenie nie je úsporné, keďže potrebujeme 4 bity namiesto 2bitov/symbol
- lepšie riešenie je rozšíriť naše praktické riešenie: text rozdelíme na bloky, na začiatok každého bloku zapíšeme rank bloku pre každé písmeno, zvyšné pozície dopočítame pomocou bitových operácií (xor, shift, and a popcnt)

7 Väčšia abeceda – Wavelet stromy

- rozdelíme abecedu na dve časti $\Sigma = \Sigma_0 \cup \Sigma_1$
- vytvoríme bitvektor B: $B[i] = j \iff S[i] \in \Sigma_j$
- predspracujeme B pre rank
- vytvoríme reťazce S_0 a S_1, kde S_j obsahuje písmená z S, ktoré sú v Σ_j ($|S_0| + |S_1| = |S|$)
- rekurzívne pokračujeme v delení abecedy a reťazcov, až kým nedostaneme abecedy veľkosti 1, kde je úloha triviálna
- takouto konštrukciou dostaneme binárny wavelet strom (WT), kde každý list reprezentuje jedno písmeno, resp. 1-písmenovú abecedu a každý vrchol reprezentuje abecedu Σ_v písmen pod ním, $\Sigma_v = \Sigma_{\text{left}(v)} \cup \Sigma_{\text{right}(v)}$; každý podstrom v kóduje reťazec $S_v = S|\Sigma_v$
- iný pohľad na WT je, že každý znak $c \in \Sigma$ zakódujeme binárne; tento kód potom špecifikuje cestu z koreňa do c (0=vľavo, 1=vpravo)
- $\text{rank}_c(i)$: začneme z koreňa; nech $c \in \Sigma_j$; spočítame $i' = \text{rank}_j(B, i)$ a rekurzívne zavoláme $\text{rank}_c(i')$ na ľavom/pravom synovi
- ak chceme podporovať aj select, predspracujeme bitvektory pre select; výpočet select_c je inverzný ku rank_c:
- $\text{select}_c(i)$: začneme v liste c; nech pre otca vrcholu platí $c \in \Sigma_j$; spočítame $i' = \text{select}_j(B, i)$, rekurzívne zavoláme $\text{select}_c(i')$ na otcovi
- dokonca vieme zrekonštruovať písmeno S na i-tej pozícií, takže ak zostrojíme WT, samotný reťazec si už nemusíme pamätať
- $S[i]$: začneme v koreni; nech $j = B[i]$, $i' = \text{rank}_j(B, i)$; pokračujeme s hľadaním $S_j[i']$ v synovi

- ostáva rozhodnúť detaily:
  - ako budeme deliť abecedu v každom vrchole?
  - ako budeme reprezentovať celý strom?
  - ako budeme reprezentovať jednotlivé bitvektory?
- prirodzená možnosť je deliť Σ vždy na polovicu, čím dostávame perfektne vyvážený binárny strom
- jeho výška je $\lg \sigma$ a na každej úrovni máme bitvektory s celkovou dĺžkou n – prirodzená reprezentácia je jednoducho ich zreťaziť do 1 poľa
- pamäť $n \log \sigma + o(n \log \sigma)$, čas rank/select/access je $O(\log \sigma)$
- druhá možnosť je reprezentovať všetky bitvektory pomocou RRR – takáto štruktúra zaberie $nH(S) + o(n \log \sigma)$:
  - nech n_0, n_1 je počet 0/1 v koreni a $n_{00}, n_{01}, n_{10}, n_{11}$ počet 0/1 v ľavom a pravom synovi
  - RRR v koreni zaberie $n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$ bitov
  - synovia zaberú $n_{00} \lg(n_0/n_{00}) + n_{01} \lg(n_0/n_{01}) + n_{10} \lg(n_1/n_{10}) + n_{11} \lg(n_1/n_{11})$
  - keď to sčítame, dostaneme $n_{00} \lg(n/n_{00}) + n_{01} \lg(n/n_{01}) + n_{10} \lg(n/n_{10}) + n_{11} \lg(n/n_{11})$
  - keď takto budeme pokračovať, na konci dostaneme $\sum_c n_c \lg(n/n_c) = nH(S)$
- keďže bitvektory sú komprimované, nemajú rovnakú dĺžku a na reprezentáciu stromu potrebujeme pomocné dáta
  - jedna možnosť je mať explicitne vrcholy so smerníkmi na otca a synov
  - druhá možnosť je RRR štruktúry zreťaziť a zapísať si offset, kde začína každý vrchol

- ďalšia možnosť je, že abecedu nebudeme deliť vždy na polovicu, t.j. nebudeme mať pre Σ kód pevnej dĺžky, ale vezmeme napríklad Huffmanov kód
- vďaka tomu bude pamäť $nH(S)$, ak keď jednotlivé bitvektory reprezentujeme priamo
- čas jednej operácie bude $O(\sigma)$ v najhoršom prípade, ale $O(H(S))$ ($H(S) \leq \log \sigma$)) v priemernom prípade (ak k jednotlivým znakom pristupujeme s približne rovnakou frekvenciou ako sú zastúpené v stringu S)

- v praxi sa ukazuje, že je celkom výhodné použiť riešenie rank pre malú abecedu (veľkosti 4 alebo 8) a namiesto binárneho WT zostrojiť 4- alebo 8-árny WT
- tiež sa osvedčila veľmi jednoduchá kompresia namiesto priamej reprezentácie bitvektoru: rozdelíme na superbloky a bloky, (1blok 64bitov, 1superblok 32blokov) a vynecháme bloky, kde sú samé 0 / samé 1