Najkratšie cesty s celočíselnými dĺžkami

©kuko 25.10.2022

1 Dijkstrov algoritmus

- všetky vrcholy vložíme do haldy so začiatočnou vzdialenosťou ∞, až na počiatočný vrchol – ten má 0
- postupne z haldy vyberáme vždy najbližší vrchol (x) a jeho susedom upravíme vzdialenosť: ak $d(y) \leftarrow \min\{\, d(y), d(x) + c(x, y) \,\}$, pričom používame funkciu `decrease_key`
- výsledná zložitosť je $O(n \times (T_{\texttt{insert}} + T_{\texttt{get\_min}}) + m \times T_{\texttt{decrease\_key}})$
- implementácia s poľom je $O(n \times (1 + n) + m \times 1) = O(n^2)$, s binárnou haldou $O(n \times (\log n + \log n) + m \times \log n) = O(m \log n)$, s Fibonacciho haldou $O(n \times (1 + \log n) + m \times 1) = O(m + n \log n)$, ale toto riešenie je nepraktické

2 Celočíselné dĺžky

- predpokladajme teraz, že všetky dĺžky sú celočíselné z rozsahu $[0, 1, \ldots, C]$ – vieme implementovať čosi lepšie?
- extrémny prípad: ak by boli všetky dĺžky 1, môžeme použiť BFS, čo je $O(m)$
- ak by sme mali iba veľmi malé dĺžky, môžeme hranu dĺžky c nahradiť cestou dĺžky c (c jednotkových hrán) a použiť BFS – graf sa nám nafúkne na C-násobok a zložitosť bude $O(C \times m)$

- iný nápad: všetky vzdialenosti budú z rozsahu $[0, 1, \ldots, n \times C]$, takže by sme mohli použiť jednoduché pole, pričom na políčku d budeme mať všetky vrcholy vo vzdialenosti d
- postupne prechádzame týmto poľom zľava doprava – niektoré políčka budú prázdne, ale keď natrafíme na neprázdne, tieto vrcholy sú v minimálnej vzdialenosti (zpomedzi nespracovaných vrcholov)
- okrem toho, že musíme prejsť celé pole, všetky operácie sú konštantné, takže $O(m + n \times C)$

- dá sa lepšie?

3 Radix halda

- všimnime si dve dôležité vlastnosti, ktoré súvisia s tým, ako Dijkstrov algoritmus využíva svoju haldu
  - `get_min` operácie vracajú rastúcu (resp. neklesajúcu) postupnosť vzdialeností – ak už spracujem nejaký vrchol, všetky ďalšie, čo budem spracovávať majú ešte väčšiu vzdialenosť!
  - ak vzdialenosť nie je ∞, potom je najviac o C väčšia ako posledný spracovaný vrchol! (pretože v momente, keď sme sa k tomuto vrcholu dostali nejakou cestou, tak vzdialenosť od spracovávaného vrcholu bola najviac C – lebo iba také dlhé sú hrany; odvtedy tá vzdialenosť mohla len klesnúť; vzdialenosti spracovaných vrcholov len rastú)

- z toho vyplývajú dve veci:
  - stačí nám tzv. monotónna halda, ktorá predpokladá, že výsledky `get_min` budú iba rásť a do haldy nikdy nevložím prvok menší ako súčasné minimum
  - mala by nám stačiť $O(C)$ pamäť – ak vzdialenosť vrcholu rátame relatívne, o koľko sú väčšie ako posledná `get_min` hodnoty, potom máme iba hodnoty $[0, \ldots, C]$ a ∞

3.1 Myšlienka

- budeme mať buckety s rozsahmi hodnôt $1, 1, 2, 4, 8, 16, 32, \ldots$ (stačí $O(\log n)$ bucketov), plus si pamätáme hodnotu minima
- prvky budeme ukladať relatívne od tohto minima, napríklad ak je to 7, tak v nultom buckete budú všetky 7mičky, v prvom 8mičky, v druhom hodnoty $[9 \ldots 10]$, v treťom hodnoty $[11 \ldots 14]$, štvrtom $[15 \ldots 22]$, atď.
- vo všeobecnosti, ak bola posledná vybraná hodnota x, tak buckety budú:

$$\underbrace{[x]}_{1}, \underbrace{[x+1]}_{1}, \underbrace{[x+2 \ldots x+3]}_{2}, \underbrace{[x+4 \ldots x+7]}_{4}, \underbrace{[x+8 \ldots x+15]}_{8}, \ldots$$

- (toto nie je úplne pravda, lebo prvky budeme trochu lenivo upratovať, ale äk by boli všetky prvky upratané, boli by takto")

- ak chceme vložiť nejaký nový prvok, proste ho vložíme do správneho bucketu
- ak chceme zmeniť hodnotu nejakého prvku (pričom poznáme jeho polohu v halde), proste ho zoberieme a vložíme, kam treba
- a ako funguje `get_min`?
  - nuž samotné minimum máme v nultom buckete – stačí ho vybrať; ak v buckete ostali ešte ďalšie prvky s rovnakou minimálnou hodnotou, môžeme skončiť
  - ak sa nám však minimálny bucket vyprázdnil, potrebujeme nájsť nové minimum:
  - lineárne prechádzame zľava doprava, kým nenájdeme prvý neprázdny bucket (max $O(\log C)$ krokov)
  - následne celý tento bucket prejdeme a nájdeme nové minimum (pozor, v bucketoch čísla nemusia byť utriedené, iba sú tam čísla z nejakého rozsahu)
  - toto minimum premiestnime do nultého bucketu a všetky ostatné prvky z tohto bucketu premiestnime do "svojich správnych"bucketov, kam patria, relatívne v závislosti od nového minima!
  - napríklad ak je nové minimum $x+8$ (pozri príklad vyššie), potom hodnoty celého štvrtého bucketu $[x+8 \ldots x+15]$ rozdelíme do bucketov 0 až 3; ich nové rozsahy budú

    $$\underbrace{[x+8]}_{1}, \underbrace{[x+9]}_{1}, \underbrace{[x+10 \ldots x+11]}_{2}, \underbrace{[x+12 \ldots x+15]}_{4}, \ldots$$

    respektíve v závislosti od nového minima $x' = x+8$:

    $$\underbrace{[x']}_{1}, \underbrace{[x'+1]}_{1}, \underbrace{[x'+2 \ldots x'+3]}_{2}, \underbrace{[x'+4 \ldots x'+7]}_{4}, \ldots$$

  - prvky v ostatných bucketoch necháme tak (nemôžeme si dovoliť upratať úplne celú haldu) – síce nebude platiť, že sú "v tých správnych bucketoch", ale to je v poriadku, pretože sú dosť ďaleko – buď ich počas `decrease_key` preradíme na správne miesto, alebo najskôr musíme vybrať všetky buckety pred nimi a následne ich upraceme počas `get_min`, keď sa k nim dostaneme
- koľko času zaberajú tieto operácie?
  - `insert` a `decrease_key` je v $O(1)$
  - `get_min` je v $O(\log C + B)$, kde B je veľkosť bucketu, ktorý upraceme – v najhoršom prípade to je až $O(N)$, avšak amortizovane je to stále $O(\log C)$ – skúsme sa pozrieť na životný cyklus jedného prvku: najprv ho vložíme do nejakého bucketu a odvtedy sa hýbe už iba smerom doľava! `decrease_key` ho pohne len doľava a pri upratovaní počas `get_min` sa pohne zase len doľava no a keďže máme len $O(\log C)$ bucketov, každý prvok sa môže premiestniť iba $O(\log C)$-krát (môžeme si to predstaviť tak, že pri `insert`e vložíme prvku na účet $\lg C\$$ a všetky presuny potom platíme z toho)

3.2 Implementačné detaily

- ak si prečítate pôvodný teoretický článok, `https://people.ksp.sk/~kuko/ds/mat/sssp.pdf`, tam buckety implementujú pomocou obojsmerného spájaného zoznamu – to v praxi samozrejme nechceme, kvôli cache-efektivite; namiesto toho použijeme obyčajný vektor
- v tomto článku si tiež explicitne pamätajú rozsahy bucketov (pre každý bucket majú hodnotu $u(i)$ a v i-tom buckete sú hodnoty $[u(i-1)+1\dots u(i)]$); takto vieme presne popísať jednotlivé rozsahy, ako sa menia, prípadne o nich niečo dokazovať; v skutočnosti, ako sme si popísali algoritmy vyššie, si tieto rozsahy netreba pamätať explicitne
- ak chcete z vektora vybrať nejaký prvok "uprostred", tak určite *nechcete* použiť funkciu, ktorá prvok zmaže a všetky prvky za ním posunie – toto má lineárnu zložitosť! (ako to spraviť v $O(1)$?)
- ako nájdeme číslo správneho bucketu? ak min je x, kam patrí prvok y?
  - spočítame $d = y - x$ a potom chceme zobraziť $0 \to 0$, $1 \to 1$, $2, 3 \to 2$, $4, 5, 6, 7 \to 3$, atď. v zásade to je niečo ako dvojkový logaritmus zaokruhlený nadol
  - prosím vás, nie že mi to budete počítať pomocou matematickej funkcie logaritmus a s reálnymi číslami – to je strašne pomalé
  - stačí nájsť najvyšší jednotkový bit; súčasné počítače toto vedia spočítať jednou inštrukciou, viď BSR (bit scan reverse) `https://c9x.me/x86/html/file_module_x86_id_20.html` v `gcc` môžeme napr. použiť `31-__builtin_clz(d)` (ak d je 32-bitové číslo); CLZ znamená "count leading zeros", pozri `https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Other-Builtins.html` pre zoznam prístupných built-in funkcií; väčšina sa skompiluje na jednoduché inštrukcie

- ešte o chlp efektívnejší variant je popísaný napr. tu: `http://ssp.impulsetrain.com/radix-heap.html`
- namiesto *rozdielu* $y - x$ budeme priradzovať prvky do bucketu podľa XORu $x \oplus y$; ak $x \oplus y = 0$, tak $x = y$ a prvok ide do nultého bucketu; ak najvyšší bit, v ktorom sa x a y líšia je i-ty, tak šup s ním do bucketu $i + 1$