tekst

gustav

January 9, 2017

Contents

1 opis, primjer

Mjere udaljenosti stringova od velike su važnosti u bioinformatici. Najpoznatiji primjeri uključuju, Levenshteinovu (odnosno edit) udaljenost i LCS (longest common subsequence), odnosno najdulji zajednički podniz.

Obje udaljenosti za dva stringa a i b u općenitom se slučaju računaju dinamičkim programiranjem u složenosti $O(|a||b|)$, što ih, usprkos sofisticiranim optimizacijama kako memorije, tako i vremena izvođenja, nepraktičnim za velike primjere.

Zbog toga su razvijene modifikacije problema koje daju rjeđe matrice dinamičkog programiranja i time omogućuju implementaciju koja je dovoljno efikasna na primjerima iz stvarnoga svijeta.

Primjer takve modifikacije je LCS_k [Benson], koja zahtijeva da se zajednički podniz sastoji od ne-preklapajućih podstringova zadane duljine k.

Jasno je da povećanjem k dobivamo manji broj parova jednakih podstringova dvaju stringova. S druge strane za prevelik k udaljenost je nula i mjera postaje beskorisna.

Mjera kojom smo se bavili u okviru ovog projekta je $\mathrm{LCS}_{\mathrm{k++}}$ [Pavetić, Žužić, Šikić], koja relaksira uvjet $\mathrm{LCS}_{\mathrm{k}}$ tako što dozvoljava preklapanja. Drugim riječima, u $\mathrm{LCS}_{\mathrm{k++}}$ razmatraju se zajednički podnizovi sastavljeni od podstringova duljine **barem** k.

Uzmimo za primjer stringove a = "ABBABDCDAD" i b = "BCBABB-DCDBAD". $LCS_{2++}(a,b) = 8$, podstring je "ABBDCDAD" ({boldano na primjeru}). $LCS_{3++}(a,b) = 6$, podstring je "ABBDCD" ({boldano na primjeru}).

2 opis algoritma, primjer

2.1 calc$_{\mathbf{matches}}$

Originalni $\mathrm{LCS}_{\mathrm{k++}}$ algoritam [Pavetić, Žužić, Šikić], kao i neki drugi LCS algoritmi, kao početni korak traže sve parove indeksa (i, j) na kojima se ulazni par stringova (a, b) "poklapa". U kontekstu običnog LCS-a, radi se o točkama za koje $a[i] = b[j]$. U kontekstu $\mathrm{LCS}_{\mathrm{k}}$, ili $\mathrm{LCS}_{\mathrm{k++}}$ promatramo točke za koje $a[i..i+k-1] = b[j..j+k-1]$, odnosno za koje su podstringovi duljine k koji počinju na pripadajućim pozicijama jednaki. $\mathrm{LCS}_{\mathrm{k}}$ i $\mathrm{LCS}_{\mathrm{k++}}$ se naravno svode na LCS u slučaju $k = 1$. U nastavku ćemo se fokusirati na ovu drugu definiciju, te ćemo parove koji je zadovoljavaju zvati jednostavno *točkama*, a njihove elemente *koordinatama*.

Općenito rješenje ovog koraka moguće je napraviti u složenosti $O(|a| + |b| + |r|)$ gdje je |r| ukupan broj točaka [poljski rad]. Ideja je konstruirati sufiksno polje nad stringom ab, te ga podijeliti na segmente s $LCP >= k$. Unutar takvog segmenta svaki par sufiksa gdje jedan dolazi iz a, a jedan iz b, definira jednu točku. Uz odgovarajuća preslagivanja sufiksa unutar segmenata moguće je točke generirati u redoslijedu rastuće prve, pa druge koordinate.

Ovaj pristup, iako teoretski zadovoljavajuć (složenost je optimalna), u praksi se ne ponaša toliko dobro. Slijedeći [Pavetić, Žužić, Šikić], fokusirali smo se na manje vrijednosti k, za koje je moguće napraviti savršeno sažimanje (*perfect hashing*) u 64-bitne riječi. U cilju poboljšanja efikasnosti uveli smo neke *low-level* optimizacije. Tako primjerice za k do 20 i abecedu do 4 elementa, podstring od k znakova možemo zapisati u 40 bitova. U preostalih 24 bita možemo pohraniti indeks i oznaku stringa iz kojeg podstring dolazi. Sortiranje niza cijelih brojeva moguće je izvesti puno efikasnije od najbržih algoritama za sortiranje sufiksa. Za to smo koristili vlastitu eksperimentalno optimiranu varijantu *radix sorta*.

{opis tog sorta? detalji? jel treba uopće?}

Nakon sortiranja algoritam je sličan prethodnom, niz (u ovom slučaju podstringova, a ne sufiksa), dijeli se na segmente prema jednakosti, te se iz odgovarajućih parova unutar segmenta generiraju točke.

{primjer sortiranja, sortirati parove (substring, string, indeks)}

Valja napomenuti da u slučaju malog broja točaka ovaj dio algoritma vremenski potpuno dominira nad ostatkom, te je dobar dio napora uložen kako bi se toliko ubrzao (više o samom ubrzanju u kasnijem poglavlju).

2.2 traženje LCSk++ iz točaka

Preostaje iz već poznatih točaka pronaći sam $\mathrm{LCS}_{\mathrm{k++}}$. Ako s P označimo skup točaka konstruiran u prethodnom koraku, rekurzivna relacija dinamičkog programiranja na tako prorijeđenoj matrici ima sljedeći oblik:

{dp relacija za sparse} { za (i, j) iz P: dp(i, j) = max { (1) k + $\max_{\text{i' <= i - k, j' <= j - k}}$ {dp(i', j')}, (2) 1 + dp(i-1, j-1) ako (i-1, j-1) $\in$ P } }

Ako imamo niz V točaka koji sadrži točke iz P sortirane po prvoj, pa po drugoj koordinati, lako je za svaku točku (i, j) pronaći indeks točke $(i-1, j-1)$, u slučaju da je prisutna u nizu. To možemo napraviti u amortizirano linearnom vremenu jednim prolaskom kroz niz V, i time je dio (2) riješen. U [Pavetić] isto je izvedeno binarnim pretraživanjem, što je neznatno sporije. Ako točke (i, j) i $(i-1, j-1)$ postoje, kažemo da je druga *nastavak* prve.

Dio (2) je nešto složeniji. U [Pavetić] koristi se prolaz po retcima pa po stupcima (u smislu koordinata točaka), pri čemu se održava struktura podataka (Fenwickovo stablo) koja omogućuje računanje gornjeg maksimuma u logaritamskoj složenosti.

Naš pristup vođen je idejom algoritma za LCS iz [Hunt], koju je uz neke manje trivijalne opservacije moguće prilagoditi za $\mathrm{LCS}_{\mathrm{k++}}$. Za početak ćemo objasniti ideju za LCS, a onda i proširenje na $\mathrm{LCS}_{\mathrm{k++}}$.

2.2.1 Hunt

U [Hunt] se također radi prolaz po retcima pa po stupcima. Glavna ideja iz [Hunt] je (po opisu iz [Survey]) održavati niz $MinYPos[l]$, koji uz pretpostavku da smo trenutno u retku i označava minimalni j takav da je $LCS(a[1..i], b[1..j]) = l$. Primijetimo da je $MinYPos$ nužno rastući niz.

Pretpostavimo da smo obradili točke (i', j') s $i' < i$, te sada promatramo točke (i, j), za neki fiksni i. Pretpostavimo da $MinYPos[l] < j$. Tada postoji LCS duljine l koji završava točkom (i', j') gdje $i' < i$ i $j' < j$. Taj je LCS moguće proširiti točkom (i, j), pa znamo da nakon obrade trenutnog retka mora vrijediti $MinYPos[l + 1] <= j$.

Vidimo da je dovoljno pronaći l takav da $MinYPos[l] < j <= MinYPos[l+1]$, što možemo napraviti binarnim pretraživanjem, te postaviti $MinYPos[l+1]$ na j (jer niz duljine l koji završava na $MinYPos[l]$ proširujemo u niz duljine $l + 1$ koji završava na j).

Ovdje treba napomenuti da je redoslijed obilaska točaka za fiksni i bitan. Točke treba obići padajuće po stupcima, kako bi se promjene niza $MinYPos$ dogodile efektivno paralelno. U protivnom se može dogoditi da izgradimo ilegalan LCS koji sadrži točke u istom retku.

2.2.2 Kuo & Cross

Algoritam iz [Hunt] pojednostavljen je u [Kuo] tako da se umijesto binarnog pretraživanja radi amortizirano linearan prolaz po trenutnom retku i nizu $MinYPrefix$. Dakle dok prolazimo kroz točke u trenutng retka, ujedno održavamo odgovarajući indeks l u $MinYPrefix$, koji za trenutnu točku (i, j) povećavamo dok $MinYPrefix[l] < j$. Na prvi pogled to pogoršava složenost algoritma. Taj instinkt je točan u općenitom slučaju, no svejedno analizirajmo detaljnije složenosti tih dvaju pristupa.

Recimo da u trenutnom retku i imamo t točaka. Huntov algoritam primijenjen na jednom retku ima složenost $O(t \log r)$, gdje je r najdulji LCS koji smo do sad pronašli. S druge strane algoritam Kuoov algoritam ima složenost $O(t + r)$. Dakle jasno je da je za veći r bolji Huntov algoritam, a za manji Kuoov.

2.2.3 Proširenje na $\mathbf{LCS_{k++}}$

Prvo ćemo malo modificirati značenje niza $MinYPrefix$. Za algoritam koji slijedi $MinYPrefix[l]$ označava najmanji j takav da $LCS_{k++}(a[1..i + k - 1], b[1..j + k - 1]) >= l$. Razlika je u tome što smo znak jednakosti zamijenili u znak nejednakosti (promijenili smo i intervale da uzmemo u obzir duljinu k). Ta promjena je ključna za očuvanje važnog svojstva $MinYPrefix$: niz mora biti ne-padajući kako bismo ga mogli binarno pretraživati.

Kako bismo se uvjerili u nužnost uvjeta, zamislimo instancu u kojoj nema nastavaka (dakle ako postoji točka (i, j), tada ne postoji $(i - 1, j - 1)$). U tom slučaju $MinYPrefix[l]$ može biti veći od nule samo za l djeljiv s k. Za

$k > 1$ takav niz će rijetko biti ne-padajuć.

Tijekom obrade točaka iz retka i, potreban nam je $MinYPrefix$ u stanju u kojem je bio nakon obrade retka $i - k$, pa zasad jednostavno pretpostavimo da nam je dostupan. Označimo ga s $MinYPrefix_{i-k}$. Promatramo točku (i, j). Ona se može nastaviti na $\mathrm{LCS}_{\mathrm{k++}}$ koji završava u točki (i', j') s $i' <= i - k$ i $j' <= j - k$. Po gornjoj pretpostavci točke uračunate u $MinYPrefix$ zadovoljavaju prvu nejednakost. Za drugu, slično kao u Huntovom, odnosno Kuoovom algoritmu, pronađemo l takav da $MinYPrefix[l] < j-k+1 <= MinYPrefix[l+1]$. Tada znamo da postoji $\mathrm{LCS}_{\mathrm{k++}}$ duljine l koji se točkom (i, j) može proširiti u $\mathrm{LCS}_{\mathrm{k++}}$ duljine $l + k$. Štoviše, u koliko (i, j) nema nastavak, znamo da je $dp(i, j) = l + k$. Ako ipak postoji, \$dp(i, j) = max{l+k, 1+dp(i-1, j-1)}. Time smo izračunali vrijednosti tablice dinamičkog programiranja u svim točkama i-tog retka. Konačan algoritam bira između Huntovog i Kuoovog pristupa jednostavnom procjene vremena izvršavanja temeljene na njihovim teorijskim složenostima i empirijski utvrđenoj konstanti.

Zatim moramo modificirati $MinYPrefix$ uzevši u obzir rezultate i-tog retka. U slučaju da $dp(i, j) = dp(i-1, j-1) + 1$, jednostavno postavimo $MinYPrefix_i[dp(i, j)] = min\{MinYPrefix_{i-1}[dp(i, j)], j\}$. Inače $dp(i, j) = l+k$, te postavljamo $MinYPrefix_i[l+s] = min\{MinYPrefix_{i-1}[l+s], j\}$ za sve s iz $[1..k]$. Prvi instinkt je uzeti samo $s = k$, ali po gornjoj redefiniciji $MinYPrefix$, to nije dovoljno. Znamo da $MinYPrefix_{i-1}[l+s] <= j$, za s iz [-l..0], budući da $MinYPrefix_{i-k}[l + s] <= j$ za s iz istog intervala (slijedi iz definicije l i činjenice da vrijednosti $MinYPrefix$ za fiksni indeks ne mogu rasti). Ali moguće je na primjer da $MinYPrefix_{i-1}[l + k - 1] > j$ (naravno uz pretpostavku $k > 1$). Ali budući da točkom u stupcu j možemo dobiti $\mathrm{LCS}_{\mathrm{k++}}$ duljine $l + k$, i $l + k >= l + k - 1$, mora vrijediti $MinYPrefix_i[l + k - 1] <= j$. Analogan argument vrijedi za ostale s iz $[1..k - 1]$.

Za kraj preostaje objasniti kako dobiti $MinYPrefix_{i-k}$. Najjednostavnije je podijeliti obradu retka na računanje vrijednosti tablice dinamičkog programiranja i osvježavanje niza $MinYPrefix$. Tako nakon što osvježimo $MinYPrefix$ za redak $i - k$, možemo odmah izračunati dp za redak i. A kad dođemo do retka i, prvo osvježimo $MinYPrefix$, a zatim računamo dp za redak $i + k$ (ako takav postoji), i tako dalje.

3 mjerenja, rezultati

4 zaključak