tekst

gustav

January 8, 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 $\text{LCS}_{\text{k++}}$ [Pavetić, Žužić, Šikić], koja relaksira uvjet LCS_k tako što dozvoljava preklapanja. Drugim riječima, u $\text{LCS}_{\text{k++}}$ razmatraju se zajednički podnizovi sastavljeni od podstringova duljine **barem** k.

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

2 opis algoritma, primjer

2.1 $\mathbf{calc_{matches}}$

Originalni $\mathrm{LCS_{k++}}$ 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 $\mathrm{LCS_k}$, ili $\mathrm{LCS_{k++}}$ 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. 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.

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).