Longest common subsequence korišćenjem Genetskog i Beam search algoritma

Projekat iz Računarske inteligencije
Matematički fakultet
Univerzitet u Beogradu

Pavle Cvejović
mi18024@alas.matf.bg.ac.rs
Viktor Novaković
mi18092@alas.matf.bg.ac.rs

September 2022

Apstrakt

Algoritmi na sekvencama[1] simbola su izučavani duže vreme i sada formiraju fundamentalni deo računarskih nauka. Jedan od veoma bitnih problema u analiziranju sekvenci je problem pronalaženja najduže zajedničke podsekvence. U generalnom slučaju, kada imamo proizvoljan broj ulaznih sekvenci, ovaj problem je NP-težak [5]. Ovde opisujemo dva pristupa za rešavanje ovog problema - jedan baziran na genetskom algoritmu a drugi na Beam search algoritmu.

[1] Koristimo termin *sekvenca* a ne *niska* jer se ovaj problem odnosi na nizove proizvoljnih tipova i ne koristimo termin *niz* jer će se on upotrebljavati u drugom kontekstu

Sadržaj

1 Uvod

Problem pronalaženja najduže zajedničke podsekvence skupa od k sekvenci (k-Longest Common Subsequence, k-LCS) je jedan od najizučavanijih problema u računarskim naukama u posednjih 30-ak godina jer igra bitnu ulogu u poredjenju sekvenci podataka. Ima potencijalne primene u mnogim područijima. Ovaj problem je koristan u prepoznavanju uzoraka, obradi i kompresiji teksta i podataka [8] i molekularnoj biologiji [1]. Može se posmatrati kao *mera bliskosti* k sekvenci jer se sastoji iz pronalaženja najvećeg broja identičnih elemenata svih k sekvenci ali takvih da je bitno uredjenje tih elemenata. Na primer, poredjenje k DNK sekvenci da se vidi njihovo podudaranje ili traženje reči u rečniku blizu pogrešno napisane reči u aplikacijama koje proveravaju pravopis.

2 Opis problema

Ako imamo dve sekvence S i T na nekoj fiksnoj azbuci Σ, sekvenca T je *podsekvenca* od S ukoliko se T može dobiti iz S brisanjem nekih elemenata iz S. Primetimo da uredjenje preostalih elemenata u S mora biti očuvano. Dužina sekvence S je broj elemenata u njoj i zapisuje se sa $|S|$. Radi jednostavnosti, sa $S[i]$ ćemo obeležavati i-ti element u sekvenci S, a sa $S[i,j]$ podsekvencu od S koja se sastoji od i-tog do j-tog elementa iz S. Praznu sekvencu obeležavamo sa ϵ.

Problem: Ako su nam date sekvence $S_i, 1 \leq i \leq k$, na nekoj fiksnoj azbuci Σ, pronadji sekvencu T koja je podsekvenca S_i za svako $i \in \{1, 2, ..., k\}$

3 Brute force algoritam

3.1 Implementacija

Za implementaciju *brute force* algoritma koristili smo tehniku **dinamičkog programiranja** pronalaženja LCS od k sekvenci. Prvo, podsetimo se kako izgleda funkcija koja prima kao argumente dužine (n_1, n_2) 2 sekvence (a_1, a_2) i računa dužinu njihovih LCS:

$$f(n_1, n_2) = \begin{cases} 0 & \text{ako } n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1 & \text{ako } a_1[n_1 - 1] = a_2[n_2 - 1] \\ \max\left(f(n_1 - 1, n_2), f(n_1, n_2 - 1)\right) & \text{inače} \end{cases}$$

Ukoliko bi želeli da generalizujemo ovu funkciju za k sekvenci $a_1, a_2, ..., a_k$ sa dužinama $n_1, n_2, ...n_k$ dobili bi smo sledeću rekurzivnu formulu:

$$f(n_1, n_2, ..., n_k) = \begin{cases} 0, \quad \text{ako je } n_1 = 0 \vee n_2 = 0 \vee \cdots \vee n_k = 0 \\ f(n_1 - 1, n_2 - 1, ..., n_k - 1) + 1, \quad \text{ako je } a_1[n_1 - 1] = a_2[n_2 - 1] = \cdots = a_k[n_k - 1] \\ \max\left(f(n_1 - 1, n_2, ..., n_k), f(n_1, n_2 - 1, ..., n_k), ..., f(n_1, n_2, ..., n_k - 1), \quad \text{inače}\right. \end{cases}$$

Dalje, kreirajmo k-dimenzioni niz gde je svaki od nizova iste duzine kao odgovarajuća sekvenca (i inicijalizujmo ih tako da su im svi elementi 0) koji će nam koristiti kao DP tablica za enumeraciju svih sekvenci za računanje puta od LCS. Korišćenjem prethodne formule, možemo jednostavno da izvučemo formulu i za formiranje DP tablice koja će nam pomoći u rekonstrukciji LCS:

Algoritam 1 Formiranje DP tablice

Ulaz: niz sekvenci $a_1, a_2, ..., a_k$ **i njihovih dužina** $n_1, n_2, ...n_k$
Izlaz: formirana k**-dimenziona DP tablica**

$\texttt{DP} \leftarrow \mathbf{0}_{n_1 \times n_2 \times \cdots \times n_k}$
for $(i_1, i_2, ..., i_k), (el_1, el_2, ..., el_k)$ `in` enumerate$(a_1 \times a_2 \times \cdots \times a_k)$ **do**
  if $el_1 = el_2 = \cdots = el_k$ **then**
    $DP[i_1, i_2, ..., i_k] \leftarrow DP[i_1 - 1, i_2 - 1, ..., i_k - 1] + 1$
  else
    $DP[i_1, i_2, ..., i_k] \leftarrow \max(DP[i_1 - 1, i_2, ..., i_k], DP[i_1, i_2 - 1, ..., i_k], ..., DP[i_1, i_2, ..., i_k - 1])$
  end if
end for
return `DP`

Sada možemo ići "unazad" kroz DP tablicu da rekonstruišemo LCS tako što kada god nadjemo pogodak u tablici nadovežemo element iz bilo koje sekvence (koristeći bilo koji indeks) na rezultujuću podsekvencu:

Algoritam 2 Formiranje LCS

Ulaz: k**-dimenziona tablica DP, niz sekvenci** $a_1, a_2, ..., a_k$ **i njihovih dužina** $n_1, n_2, ...n_k$
Izlaz: LCS ulaznih sekvenci

`lcs` $\leftarrow$ `[]`
while $n_i > 0 \ \ (\forall i \in \{1, .., k\})$ **do**
  `step` $\leftarrow DP[n_1, n_2, ..., n_k]$
  if $(\exists i \in \{1, ..., k\})$ `step` $= DP[n_1, n_2, ..., n_i - 1, ..., n_k]$ **then**
    $n_i \leftarrow n_i - 1$
  else
    `lcs.append(`$a_1[n_1 - 1]$`)`
    $n_i \leftarrow n_i - 1 \ \ (\forall i \in \{1, .., k\})$
  end if
end while
return `lcs`

4 Genetski algoritam

4.1 Uopšteno

Pri prvom pristupu problemu, odlučili smo da koristimo genetski algoritam. Kod generacijskog genetskog algoritma, incijalizujemo populaciju sa nasumičnim jedinkama, biramo jedinke roditelje, od njih kreiramo jedinke decu, i sa odredjenom šansom mutiramo dobijene jedinke. Cilj je da pri svakom odabiru roditelja, jedinke sa boljom ocenom imaju veće šanse da budu odabrane. Na ovaj način pokušavamo da imitiramo evoluciju u prirodi.

4.2 Predstavljanje jedinki

Pretpostavimo bez umanjenja opštosti da je sekvenca najmanje dužine na prvoj poziciji, *i=1*. Tada svaku jedinku u genetskom algoritmu možemo predstaviti kao niz *bulovskih* vrednosti dužine $|S_1|$. Sekvenca koju jedinka predstavlja je sastavljena od karaktera sekvence S_1 uz koje u odgovarajućem elementu niza stoji vrednost *Tačno*. Očigledno, ovakva jedinka može da predstavi sve moguće validne sekvence za naš problem.

4.3 Fitnes funkcija

Ako uzmemo prethodnu pretpostavku, za sekvence $S_1, S_2, ..., S_k \in \Sigma^*$ važi $|S_1| \leq |S_i|$ za svako $i \in 2, ..., k$. Neka je $n = |S_1|$ i neka za svaku jedinku $s \in \{0,1\}^n$ *c(s)* bude kandidat za rešenje koji je predstavljen jedinkom s, i neka *k(s)* bude broj sekvenci $S_1, S_2, ..., S_k$ kojima je *c(s)* podsekvenca. Odlučili smo se da izaberemo fitnes funkciju iz rada [4] koju definišemo kao:

$$f(s) = \begin{cases} 3000(|c(s)| + 30k(s) + 50) & \text{ako } |c(s)| = n \wedge k(s) = k \\ 3000(|c(s)| + 30k(s)) & \text{ako } |c(s)| < n \wedge k(s) = k \\ -1000(|c(s)| + 30k(s) + 50)(k - k(s)) & \text{ako } |c(s)| = n \wedge k(s) < k \\ -1000(|c(s)| + 30k(s))(k - k(s)) & \text{ako } |c(s)| < n \wedge k(s) < k \end{cases}$$

4.4 Selekcija

Za *selekciju* smo koristili turnirsku i ruletsku selekciju raznih veličina. Najbolje se pokazala turnirska selekcija veličine 10. Nasumično se bira 10 jedinki i uzima se najbolja, tj. jedinka medju njima koja ima najveću vrednost fitnes funkcije.

4.5 Ukrštanje, mutacija i elitizam

Pokretali smo algoritam za veliki broj kombinacija parametera i tipova za ukrštanje, mutaciju i elitizam, i oni koji su se ispostavili da su najbolji iz empirijskih rezultata su *uniformno* ukrštanje (svaki od gena deteta ima jednaku šansu da bude nasledjen od prvog roditelja kao od drugog), *višestruka* mutacija sa verovatnoćom 0.05 (nasumično se bira broj mutiranih gena do 75% dužine cele jedinke i negira im se vrednost), i elitizam koji cuva 20% jedinki.

4.6 Uslov zaustavljanja

Program se zaustavlja ako se dostigne maksimalni broj iteracija (koje zavise od veličine sekvenci i broja sekvenci na ulazu), ili ako jedinka sa najboljim fitnesom ostane nepromenjena u prethodnih nekoliko generacija (ovaj broj takodje zavisi od veličine sekvenci i broja sekvenci na ulazu), ili ako je pronadjeno optimalno rešenje (ovo je samo u slučaju da je pre GA pokrenut i brute force algoritam sa kojim se može uporediti).

5 Beam search algoritam

5.1 Uopšteno

Beam search algoritam se svodi na nepotpunu pretragu stabla u širinu. Skup čvorova, zvani *beam*, se zadržava. Na početku, *beam* sadrži samo koreni čvor. U svakoj iteraciji, čvorovi *beam*-a se šire da bi se dobili njihovi potomci na narednom nivou. Od dobijenih potomaka, čuvaju se $\beta > 0$ sa najboljom ocenom heuristike.

5.2 Graf stanja

Neka n bude maksimalna dužina sekvenci $S_i, i \in 1, ..., k$.

Neka je $p^L \in \mathbb{N}^k$ vektor prirodnih brojeva takav da važi $1 \leq p_i^L \leq |S_i|$, za $i = 1, ..., k$. Skup $S[p^L]$ definišemo kao $\{S_i[p_i^L, |S_i|] | i = 1, ..., k\}$. Skup $S[p^L]$ se sastoji od uzastopne podsekvence svakog od početnih sekvenca od pozicije iz p^L do kraja. Vektor p^L nazivamo levo-pozicioni vektor, i on predstavlja LCS potproblem na sekvencama $S[p^L]$.

Graf stanja LCS problema je usmereni aciklički graf $G = (V, E)$, gde je čvor $v \in V$ predstavljen odgovarajućim levo-pozicionim vektorom $p^{L,v}$ i dužinom l_v parcijalnog rešenja. Grana $a = (v_1, v_2) \in E$ sa labelom $l(a) \in \Sigma$ postoji ako je l_{v_2} za jedan veće od l_{v_1}, i ako se od parcijalnog rešenja dobijenog dodavanjem karaktera $l(a)$ na parcijalno rešenje čvora v_1 dobija potproblem $S[p^{L,v_2}]$. Koreni čvor r grafa stanja je originalni problem, koji se može zapisati kao prazno parcijalno rešenje, tj. $r = ((1, ..., 1), 0)$, koje predstavlja praznu sekvencu ϵ. Da bi se dobilo dete čvora $v \in V$, mora se prvo odrediti kojim je karakterima moguće produžiti v, tj. karakteri $a \in \Sigma$ koji se pojavljuju makar jednom u svakoj od sekvenci iz $S[p^{L,v}]$. Za svaki od karaktera a, neka je pozicija prvog pojavljivanja u $S_i[p_i^{L,v}, |S_i|]$ zapisana kao $p_{i,a}^{L,v}$. Novi čvor dobijen dodavanjem karaktera a na parcijalno rešenje predstavljeno čvorom v je onda $v' = (p^{L,v'}, l_v + 1)$, gde je $p_i^{L,v'} = p_{i,a}^{L,v} + 1$, za svako $i = 1, ..., m$. Listove ovako definisanog stabla nazivamo *kompletnim* čvorovima.

Takodje uvodimo termin *dominantnog* levo-pozicionog vektora. Levo-pozicioni vektor p^{L,v_1} je *dominantan* nad levo-pozicionim vektorom p^{L,v_2} **akko** važi $p^{L,v_1}[i] < p^{L,v_2}[i], i = 1, ..., k$.

5.3 Implementacija

Pre prikaza samog algoritma, potrebno je definisati funkcije koje ćemo definisati unutar njega.

ExtendAndEvaluate(B, h) generiše sve potomke svakog od čvorova $v \in B$, odredjuje svakom od potomaka vrednost heuristike h i sortira ih neopadajuće po odgovarajućim *h-vrednostima*, dobijeni skup čvorova nazivamo V_{ext}. Filter(V_{ext}, k_{best}) je jedan od opcionih koraka koji smo odlučili da koristimo. Filter briše sve čvorove iz V_{ext} koji su dominirani od strane drugih čvorova iz V_{ext}, ali radi efikasnosti, ne proverava se za svaki od čvorova da li je dominantan nad datim čvorom, nego

se za proveru uzimaju najboljih k_{best} iz V_{ext}. Reduce(V_{ext}, β) vraća novi *beam* koji se sastoji od najbolje ocenjenih β čvorova iz V_{ext}. Bitno je napomenuti i opcioni korak Prune(V_{ext}, ub_{prune}) koji iz V_{ext} briše sve čvorove $v \in V_{ext}$ za koje važi $l_v + ub_{prune} \leq |s_{lcs}|$, gde je ub_{prune} funkcija koja računa gornju granicu broja karaktera koji mogu da se dodaju na trenutni čvor, a $|s_{lcs}|$ dužina trenutno najboljeg rešenja, ali, radi efikasnosti, odlučili smo da ne koristimo ovaj korak.

Algoritam 3 Beam Search

Ulaz: skup sekvenci S i odgovarajuća azbuka Σ, funkcija heuristike h za ocenu čvorova, parametar k_{best} za filtriranje, β - veličina *beam*-a

Izlaz: Ostvareno LCS rešenje

$B \leftarrow \{r\}$
$s_{lcs} \leftarrow \epsilon$
while $B \neq \emptyset$ **do**
  $V_{ext} \leftarrow$ ExtendAndEvaluate(B, h)
  ažuriranje s_{lcs} ako je dostignut *kompletan* čvor v sa novim najvećim l_v
  $V_{ext} \leftarrow$ Filter(V_{ext}, k_{best})
  $B \leftarrow$ Reduce(V_{ext}, β)
end while
return s_{lcs}

5.3.1 Funkcije za ocenjivanje čvorova

Fraser [3] je kao gornju granicu za broj karaktera koji mogu da se dodaju na parcijalno rešenje predstavljeno čvorom v, tj. dužinu LCS za potproblem $S[p^{L,v}]$ koristio $\mathrm{UB}_{min}(v) = \mathrm{UB}_{min}(S[p^{L,v}]) = min_{i=1,\ldots,k}(|S_i| - p_i^{L,v} + 1)$

Za funkcije heuristike Mousavi i Tabataba [7, 6] predlažu dva pristupa. Prvi pristup se zasniva na pretpostavci da su sekvence problema uniformno nasumično generisane i medjusobno nezavisne. Autori su izveli rekurziju koja odredjuje verovatnoću $\mathcal{P}(p, q)$ da će uniformno nasumična sekvenca dužine p biti podsekvenca uniformno nasumične sekvence dužine q. Potrebne verovatnoće mogu da se izračunaju pri pretprocesiranju i čuvaju se u matrici. Koristeći fiksirano t, pod pretpostavkom da su sekvence problema nezavisne, svaki čvor se ocenjuje sa $\mathrm{H}(v) = \mathrm{H}(S[p^{L,v}]) = \prod_{i=1}^{k} \mathcal{P}(t, |S_i| - p_i^{L,v} + 1)$. Ovo odgovara verovatnoći da parcijalno rešenje predstavljeno preko v može da se produži za t karaktera. Vrednost t se heuristički odredjuje kao $t := max(1, \lfloor \frac{1}{|\Sigma|} \cdot min_{v \in V_{ext}, i=1,\ldots,k}(|S_i| - p_i^{L,v} + 1) \rfloor)$. Druga heuristička procena je takozvana *power* heuristika:

$$\mathrm{Pow}(v) = \mathrm{Pow}(S[p^{L,v}]) = \left(\prod_{i=1}^{k} (|S_i| - p_i^{L,v} + 1) \right)^{q} \cdot \mathrm{UB}_{min}(v), q \in [0, 1).$$

Gleda se kao generalizovani UB_{min}. Autori tvrde da bi trebalo da se uzme manja vrednost q u slučaju većeg k i postavljaju $q = a \times exp(-b \cdot k) + c$, gde su $a, b, c \geq 0$ zavisni od ulaznih parametara.

6 Testiranje i rezultati

Oba algoritma su pokrenuta na računaru sa sledećim specifikacijama: CPU — Amd Ryzen 5 1400x @ $4 \times 3.2GHz$, RAM — 7836MiB, OS — Windows 10, python — Python 3.10.6[2].

6.1 Genetski

Pokrenuli smo genetski algoritam da radi sa 108 različitih kombinacija parametara (generation_size, chromosome_size, tournament_size, mutation_rate, itd...) a prikazujemo rezultate GA dobijene kombinacijom parametara gorespomenutih u tekstu i uporedjujemo sa rezultatima izvršavanja brute force algoritma:

Params			DP		GA		BS-H	
$\|\Sigma\|$	n	k	$t[s]$	s_{best}	$t[s]$	s_{best}	$t[s]$	s_{best}
2	10	2	0.01	7	0.01	7	0.001	7
2	20	2	0.01	15	0.08	15	0.001	15
2	10	3	0.01	6	0.01	6	0.0	6
2	20	3	0.02	12	0.1	12	0.0	12
2	10	5	0.7	5	0.03	5	0.0	5
2	20	5	21.9	11	0.13	11	0.001	11
4	10	2	0.01	5	0.03	5	0.0	5
4	20	2	0.01	11	0.15	10	0.001	11
4	10	3	0.01	4	0.04	4	0.0	4
4	20	3	0.03	8	0.29	7	0.001	8
4	10	5	0.72	3	0.02	3	0.0	3
4	20	5	23.6	7	0.2	7	0.001	7
26	10	2	0.01	2	0.02	2	0.0	2
26	20	2	0.01	5	0.09	4	0.001	5
26	10	3	0.01	1	0.01	1	0.0	1
26	20	3	0.03	2	0.05	2	0.0	2
26	10	5	0.73	0	0.01	0	0.0	0
26	20	5	22.8	1	0.02	1	0.001	1

[2]Ime kolone u tabelama sa -Lit. u sufiksu se odnosi na rezultate odgovarajućeg algoritma iz literature, dok se $|\Sigma|$ odnosi na veličinu azbuke, n na veličinu najduže sekvence, m broj sekvenci, $t[s]$ na dužinu izvršavanja i s_{best} na dužinu pronadjene LCS

6.2 Beam search

Pri testiranju pokazalo se da je heuristika Pow brža, ali je zbog toga heuristika H davala bolje rezultate. Za heuristiku H vrednost za k_{best} koja je uzeta je 50, a za Pow je 100. Rezultate za fiksno $n = 600$ smo poredili sa [2]:

Params			GA		BS-H		BS-Pow		BS-H-Lit.		BS-Pow-Lit.	
$\|\Sigma\|$	β	k	$t[s]$	s_{best}	$t[s]$	s_{best}	$t[s]$	s_{best}	$t[s]$	s_{best}	$t[s]$	s_{best}
4	50	20	2.75	144	0.18	181	0.17	173	0.04	189	0.10	191
4	50	100	8.82	135	0.21	149	0.19	141	0.05	158	0.09	156
4	50	150	10.79	131	0.25	147	0.21	142	0.06	151	0.10	150
4	50	200	15.78	129	0.25	144	0.24	140	0.07	150	0.11	148
4	200	20	2.75	144	0.51	186	0.42	180	0.19	191	0.29	191
4	200	100	8.82	135	0.60	155	0.51	148	0.36	158	0.40	158
4	200	150	10.79	131	0.59	151	0.49	143	0.26	151	0.34	151
4	200	200	15.78	129	0.69	148	0.51	139	0.38	150	0.38	150
4	600	20	2.75	144	1.31	192	1.28	185	0.71	192	1.20	191
4	600	100	8.82	135	1.32	157	1.28	150	0.68	158	1.28	158
4	600	150	10.79	131	1.45	151	1.37	143	1.05	152	1.43	152
4	600	200	15.78	129	1.40	150	1.29	148	1.15	151	1.00	150
20	50	20	2.02	29	0.11	42	0.09	36	0.08	46	0.14	46
20	50	100	6.19	22	0.19	29	0.13	28	0.08	31	0.13	31
20	50	150	5.81	22	0.20	28	0.12	24	0.11	29	0.13	29
20	50	200	7.79	20	0.23	27	0.22	24	0.11	28	0.13	27
20	200	20	2.02	29	0.65	44	0.56	39	0.45	47	0.50	47
20	200	100	6.19	22	0.67	32	0.54	28	0.31	31	0.49	31
20	200	150	5.81	22	0.88	30	0.46	26	0.46	29	0.41	29
20	200	200	7.79	20	0.88	28	0.50	25	0.43	28	0.47	27
20	600	20	2.02	29	1.83	45	1.66	40	1.48	48	1.71	47
20	600	100	6.19	22	1.88	31	1.68	25	1.20	31	1.09	32
20	600	150	5.81	22	1.94	29	1.68	25	1.29	29	1.66	29
20	600	200	7.79	20	2.01	27	1.70	23	1.43	28	1.62	28

7 Zaključak

Na osnovu prethodnih analiza možemo zaključiti da Genetski algoritam uglavnom upadne u lokalni optimum iz koga se ne može izboriti čak ni kroz nekoliko hiljada generacija bez obzira na izbor parametara i tipova operatora dok Beam search algoritam daje bolja rešenja i to u kraćem vremenskom periodu. Takodje možemo primetiti kod Beam search algoritma da su rešenja "visokog kvaliteta" (sa većim parametrom β) samo 10% bolja a vreme izvršavanja je preko 100% veće.

Literatura

[1] R. Beal et al. “A new algorithm for “the LCS problem” with application in compressing genome resequencing data”. In: (2016), pp. 370–380.

[2] M. Djukanovic and G. R. Raidland C. Blum. “A Beam Search for the Longest Common Subsequence Problem Guided by a Novel Approximate Expected Length Calculation”. In: (2019), p. 11.

[3] Fraser. “Subsequences and Supersequences of Strings”. In: (1995), p. 940.

[4] T. Jansen and D. Weyland. “Analysis of Evolutionary Algorithms for the Longest Common Subsequence Problem”. In: (2007), p. 940.

[5] David Maier. “The Complexity of Some Problems on Subsequences and Supersequences”. In: (1978), pp. 322–336.

[6] S. R. Mousavi and F. Tabataba. “A hyper-heuristic for the Longest Common Subsequence problem”. In: 36 (2012), pp. 42–54.

[7] S. R. Mousavi and F. Tabataba. “An improved algorithm for the longest common subsequence problem”. In: 39 (2012), pp. 512–520.

[8] J. Storer. “Data Compression: Methods and Theory”. In: (1988).