Kompaktna priponska drevesa

Jani Suban
Študent Računalništva in informatike,
Univerza na Primorskem Fakulteta za matematiko,
naravoslovje in informacijske tehnologije,
Koper, Slovenija
89222015@student.upr.si

21. junij 2024

Povzetek: Pogosti problem v bioinformatiki in procesiranju naravnih jezikov je iskanje vzorcev v besedilu. Če želimo ugotoviti ali zgolj en vzorec obstaja v besedilo, lahko le to storimo z uporabo Knuth–Morris–Pratt algoritma. Če pa želi najti vse ponovitve vzorca ali poiskati prisotno več vzorcev potrebujem indeks nad besedilom. V tem članku bo predstavljena podatkovna struktura priponsko drevo, ki je pogosto uporabljeno za indeksiranje besedila. Poleg priponskega drevesa bo predstavljena tudi kompaktna predstavitev priponskega drevesa.

Ključne besede: drevesne podatkovne strukture, obdelava besedil

1 Uvod

Pogost problem v bioinformatiki je prisotnost specifičnih genov ali drugih DNK sekvenc v genomu. Podoben problem v procesiranju naravnih jezikov je prisotnost specifične fraze ali besede v besedilu. Oba problema lahko rešimo z istimi metodami, saj DNK sekvenca in beseda sta vzorca, ki ju iščemo, ter genom ter vhodno besedilo sta besedili, v katerih iščemo ta vzorca. Če iščemo zgolj en vzorec v besedilu lahko le tega najdemo v času $O(n + m)$, kjer je n dolžina besedila in m dolžina vzorca, z uporabo Knuth–Morris–Pratt algoritma [1] ali z uporabo končnega avtomata [2].

Pri tem se pojavi novi problem, ko želimo poiskati vse pojave določenega vzorca ali pa želimo poiskati, kateri od iskanih vzorcev se pojavi v besedilu. Če je število vzorcev $O(n)$ potrebujeta algoritma $O(n(n+m))$ časa za najt vse vzorce. Temu problemu se lahko izognemo z izgradnjo indeksa nad vhodnim besedilom. Če se uporablja indeks, se lahko vsak vzorec najde v indeksu v čas $O(m)$. Indeks se lahko zgradi v času $O(n)$. Indeks je lahko priponsko polje ali priponsko drevo. Članek se bo osredotočil na priponska drevesa ter na njihovo kompaktno predstavitev.

Članek je razdeljen na tri dele. V prvem delu so predstavljene osnovne definicije potrebne v nadaljevanju članka. Zatem je predstavljeno priponsko drevo ter njegova izgradnja. Na koncu pa je predstavljena kompaktna predstavitev priponskih dreves.

2 Definicije

Preden začnemo razlagati, kaj je priponsko drevo in kako deluje, potrebno predstaviti in definirati nekaj izrazov, ki bodo uporabljeni v članku. Besedilo T dolžine n je polje črk `T[1..n]`. Vsaka črka v besedilu T je del abecede Σ, torej $T[i] \in \Sigma; 1 \leq i \leq n$. Pri tem se vsako besedilo konča s simbolom, ki ni prisoten v abecedi Σ. Ta simbol predstavimo z \$. Podniz besedila T, ki se začne na i-tem in konča na j-tem znaku ($1 \leq i \leq j \leq n$), zapišemo kot $T[i, j]$. Torej vzorec P je besedilo, ki ima dolžino m. Vzorec je del besedila, če obstajata i in j, za katera velja $P[1, m] = T[i, j]$. Podobno kot za besedilo velja tudi za vzorec P, da vsaka črka vzorca je del abecede Σ, torej $P[i] \in \Sigma; 1 \leq i \leq m$, saj drugače ni mogoče najti vzorca v besedilu.

Ker v članku govorimo o priponskih drevesih, je potrebno definirati, kaj je pripona. Pripona $x = T[i, n]; 1 \leq i \leq n$ je niz v besedilu, ki se začne na i-tem mestu in se konča na koncu besedila. Pri tem velja, da je lahko pripona tudi prazni niz.

Naslednja stvar, ki mora biti definirana, je priponsko drevo. Priponsko drevo je potrebo definirati, kot abstraktno strukturo, saj v članku bodo predstavljene različne implementacije priponskega drevesa. Za to lahko uporabimo definicijo priponskega drevesa iz [3]. Priponsko drevo je definirano na sledeči način:

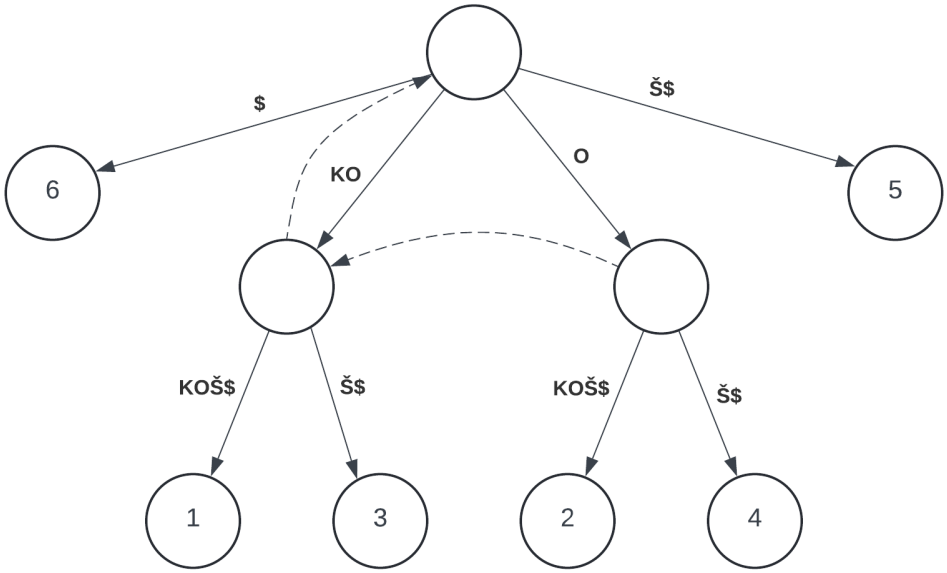
Definicija 1. Priponsko drevo nad besedilo T podpira naslednje operacije:

1. $koren()$: vrne koren priponskega drevesa,
2. $jeList(v)$: vrne Da, če je vozlišče list, sicer vrne Ne,
3. $otrok(v, z)$: vrne otroka w, katerega povezava se začne z znakom z. Če otrok ne obstaja vrne 0,
4. $prviOtrok(v)$: vrne vozlišče w, ki je prvi otrok vozlišča v,
5. $brat(v)$: vrne vozlišče w, ki je naslednji brat od vozlišča v,
6. $starš(v)$: vrne vozlišče w, ki je starš od vozlišča v,
7. $povezava(v, i)$: vrne i-to črko na povezavi do vozlišča v,
8. $globinaNiza(v)$ vrne število znakov na poti iz korena do vozlišča v,
9. $lca(v, w)$: vrne najnižjega skupnega prednika od v in w,
10. $sl(v)$: vrne vozlišče w na katerega kaže priponska povezava iz vozlišča v.

Iz definicije priponskega drevesa je predstavljena priponska povezava (ang. *suffix link*). Priponske povezave se en zgolj uporabljajo pri konstrukciji priponskega drevesa, ampak se jih uporabljajo tudi pri iskanju vzorcev v drevesu. Priponska povezava je definiramo, kot:

Definicija 2. Priponska povezava $sl(v)$ je povezava iz notranjega vozlišča v z nizom $y = \alpha x$ na poti od korena do vozlišča v nad notranje vozlišče w z nizom x na poti od korena do vozlišča w. Pri tem je lahko niz x prazen.

Ker v članku bodo uporabljene kompaktne podatkovne strukture je potrebno le te definirati. Kompaktne podatkovne strukture omogočajo enake operacije kot ekvivalentne ne kompaktne različice. Pri tem pa more celotna struktura biti shranjena v $O(n)$ bitov. Kompaktnost podatkovne strukture nam omogoča, da jo v celoti shranimo v pomnilniku, kar vpliva na hitrost iskanja vzorcev v besedilu. [4]



Slika 1: Slika prikazuje priponsko drevo za besedo *KOKOŠ$*. Polne črte predstavljajo povezave med vozlišči ter napisi predstavljajo podnize. Črtkane črte pa predstavljajo priponske povezave znotraj drevesa.

3 Priponska drevesa

Sedaj, ko smo definiral abstraktno podatkovno strukturo priponsko drevo, lahko predstavimo implementacijo prvo implementacijo le te. Prva implementacija je priponsko drevo, ki je bila predstavljena v [5]. Čeprav je implementacija starejša kot abstraktna definicija, ta podpira vse definirane operacije.

Priponsko drevo $D(T)$ prikazuje vse pripone besedila T. Vsaka pripona je predstavljena kot pot od korena do lista drevesa, pri čemer vsaka povezava predstavlja podniz besedila T. Primer priponskega drevesa za besedo *KOKOŠ$* lahko vidim na Sliki 1.

Kot je iz Slike 1 vidno je priponsko drevo Patricijino drevo, saj iz številskega drevesa so bila združena vsa enojna (vozlišča z enim otrokom) vozlišča. Torej je lahko priponsko drevo definirano na sledeči način: [6]

Definicija 3. Priponsko drevo nad besedilo T dolžine n ima sledeče lastnosti:

1. drevo ima natanko n listov ter vsak list je označen s številom med 1 in n,
2. vsako notranje vozlišče, razen korena, ima vsaj dva otroka,
3. vsaka povezava predstavlja ne prazni podniz besedila T,
4. ne obstajata dve povezavi, ki začneta iz istega vozlišča in se začneta s isto črko,
5. niz pridobljen s konkatenacijo podnizov na poti iz korena do lista i predstavlja pripono $T[i, n]$ za vsak i, za katerega velja $i \leq i \leq n$.

Definicija 3 predstavi podatkovno strukturo, ki potrebuje $O(n)$ prostora za pravilno delovanje. Pri tem velja prostor ni štet v bitih, ampak v referencah, katerih velikost je odvisna od arhitekture računalnika, ki je uporabljen.

3.1 Izgradnja priponskega drevesa

V članku je bila predhodno predstavljena podatkovna struktura priponsko drevo, ki potrebuje $O(n)$ referenc za pravilno delovanje. Ampak pri tem se pojavi nov vprašanje, kako enostavno zgraditi priponsko drevo $D(T)$ besedila T v optimalnem času. Na ta problem obstajata dve rešitvi. Prva rešitev je McCreight algoritem [7]. Algoritem zgradi drevo iz besedila T v času $O(n)$. Algoritem zgradi drevo $D(T)$ iz desne proti levi.

Pri tem se pojavi nov problem, besedilo moramo predhodno poznati in ga imeti shranjenega, da lahko zgradimo drevo. Pri mi bilo boljše, če bi se dalo izgraditi drevo $D(T)$ iz leve proti desni, tako da se prebere črko po črko, dokler se ne prebere znaka za konec besedila, ki je prikazan kot \$. To pa je mogoče storit z uporabo Ukkonenovega algoritma [8].

Ukkonenov algoritem v i-tem koraku zgradi priponsko implicitno priponsko drevo $D(T[1,i])$ za podniz $T[1,i]$. V naslednjem koraku, ko se doda nova črka α, se le ta lahko doda na tri načine:

1. če se α doda na koncu poti, ki konča v listu, se niz na zadnji povezavi podaljša za α. Torej enkrat, ko je vozlišče zgrajeno ne more postati notranje vozlišče,

2. če ne obstaja taka poti proti listom, ki se nadaljuje po nizu s in se nadaljuje s črko α. Pri tem obstaja vsaj ena pot, ki vodi proti listom. Takrat naredimo nov list, na katerega kaže povezava z oznako α. Če s se konča v sredini niza med dvema vozliščema, se ustvari novo vozlišče. Povezava, ki kaže na vozlišče, ima niz, ki se je ujemal s pripono s-ja. Iz vozlišča pa kažeta dve povezavi prva kaže na list in predstavlja niz α, druga pa kaže na vozlišče, na katerega je kazala predhodna povezava ter predstavlja preostanek predhodnega niza,

3. če obstaja taka pot proti listom, ki se nadaljuje po nizu s in se nadaljuje s črko α. V tem primeru se ne stori ničesar, saj drevo je že v implicitni obliki.

Pri tem algoritem pomni dve točki: aktivno točko (ang. *Active point*) in končno točko (ang. *End point*). Aktivna točka predstavlja, najglobljo točko v drevesu, kjer se lahko zgodi 2. način dodajanja črke α v drevesu $D(T[1:i])$. Končna točka pa predstavlja zadnjo točko na poti po priponskih povezavah, kjer se lahko zgodi 2. način dodajanja črke α v drevo $D(T[1:i])$. Poleg tega algoritem hrani tudi zadnje novo ustvarjeno vozlišče, v i-tem koraku. Na ta način algoritem ustvari priponske povezave med novo ustvarjenim ter predhodno ustvarjenim vozliščem. [6, 8]

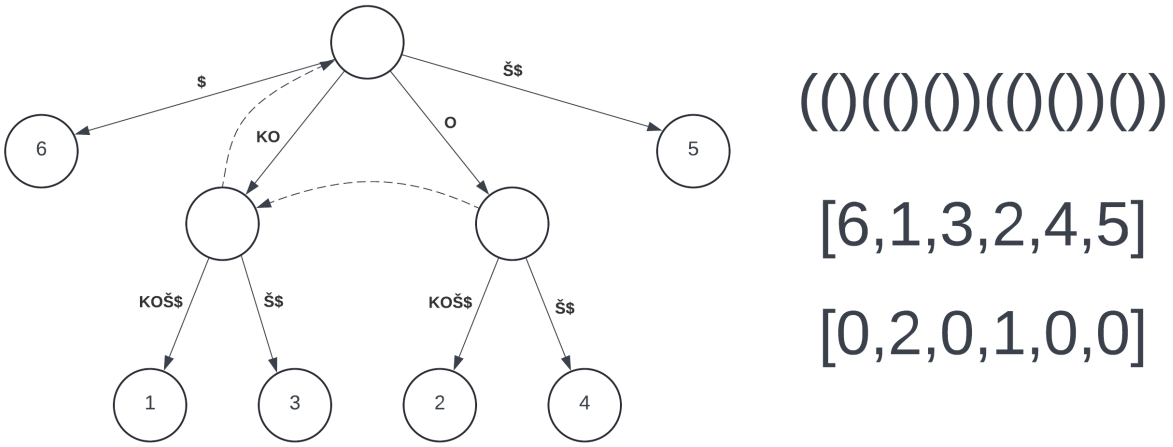
Izrek 1. *Ukkonenov algoritem zgradi priponsko drevo D besede T dolžine n v času $O(n)$. Pri tem izgradi tudi vse priponske povezave drevesa D.*

Dokaz izreka je predstavljen v [8] ter v [6]. V obeh dokazih je natančno predstavljeno, kako algoritem doseže časovno zahtevnost $O(n)$. Glava ideja dokaza je preštevanje, koliko novih vozlišč je bilo ustvarjeni v posamičnem koraku.

4 Kompaktna priponska drevesa

Do sedaj je bilo predstavljena osnovna implementacija priponskega drevesa. Implementacija potrebuje $O(n \log n)$[1] bitov za shraniti celotno priponsko drevo. Pri tem pa velja da lahko celotno besedilo T shranimo v $n \log |\Sigma|$ bitov. Välimäk idr. [9] predstavijo, da je DNK sekvenca dolžine $n = 10^7$ potrebujemo 2,4MB za zapis besedila z 2 biti za znak ter 347,5 MB če shranimo besedilo s priponskim drevesom.

[1] v nadaljevanju članka $\log n$ predstavlja $\log_2 n$



Slika 2: Slika prikazuje priponsko drevo za besedo *KOKOŠ$* na levi ter njeno kompaktno predstavitev na desni. Polne črte predstavljajo povezave med vozlišči ter napisi predstavljajo podnize. Črtkane črte pa predstavljajo priponske povezave znotraj drevesa. Prva vrstica na desni prikazuje prikaz drevesa z uravnoteženimi oklepaji, druga vrstica prikazuje priponsko polje ter zadnja vrstica prikazuje polje najnižjih skupnih predhodnikov pripon.

Ker priponska drevesa potrebujejo preveč prostora za biti shranjena, bi bilo mogoče jih stisniti na velikost besedila. Strukturo samega drevesa je mogoče zakodirati z uporabo kompaktnih prikazov drevesa, kot so na primer prikaz z urejenimi oklepaji [4]. Pri tem se pojavi nov problem, izgubi se ves pomen povezav, saj podnizi besedila so bili shranjeni na povezavah, ter se izgubijo tudi priponske povezave.

Zato Sadakane [3] predstavi podatkovno strukturo kompaktno priponsko drevo (ang. *Compacpressed Suffix Tree* ali CST). Podatkovna struktura je definiran na sledeči način:

Definicija 4. Kompaktno priponsko drevo sestavljajo sledeče podatkovne strukture:

1. prikaz drevesa z uravnoteženimi oklepaji,
2. kompaktno priponsko polje,
3. polje najnižjih skupnih predhodnikov.

Na Sliki 2 je prikazano priponsko drevo besede *KOKOŠ$* na levi ter njegova kompaktna oblika na desni.

Kot je bilo predhodno omenjeno kompaktno priponsko drevo uporablja za shraniti strukturo drevesa uporablja prikaz z uravnoteženimi oklepaji. Vsako vozlišče je prikazano kot par oklepajev $(\dots)$. Ko se vozlišče prvič obišče zapišemo uklepaj ter se doda zaklepaj šele ko pregledamo celotno poddrevo vozlišča. Torej za vsako vozlišče sta potrebna 2 bita. Torej za zakodirati priponsko drevo, ki ima največ $2n-1$ vozlišč, je potrebno $4n + o(n)$ bitov.[4, 10]

Naslednja struktura, ki je del kompaktnega priponskega drevesa, je kompaktno priponsko polje (ang. *Compacpressed Suffix Array* ali CSA). Kompaktno priponsko polje omogoča indeksiranje pripon v abecednem vrstnem redu. Ampak za razliko od priponskega polja, ki potrebuje $O(n \log n)$ bitov, kompaktno priponsko polje potrebuje $O(n \log |\Sigma|)$ bitov. Pri čemer se časovna zahtevnost dostopa do pripon poveča iz konstantne na $O(\log^{\epsilon} n)$ za poljubno konstanto $0 < \epsilon \leq 1$. [11]

Zadnja podatkovna struktura, ki je potrebna za implementacijo kompaktnih priponskih dreves, je polje najnižjega skupnega predhodnika (ang. *Height array*, *lcp-array*). S pomočjo polja najnižjega skupnega predhodnika nad zaporednimi priponami, je možen sprehod po drevesu iz listov proti korenu. Polje označeno kot *lcpArray* je definirano na sledeči način:

Definicija 5.

$$lcpArray[i] = \begin{cases} lcp(T[SA[i], n], T[SA[i+1], n]); & 1 \leq i \leq n-1 \\ 0; & i = n, \end{cases}$$

pri tem funkcija $lcp(v, w)$ vrne dolžin najdaljše skupne predpone med dvema priponama v in w. Prostorska zahtevnost te strukture je $2n + o(n)$ bitov ter $lcpArray[i]$ je izračunan v konstantenem času s pomočjo i-tega elementa v priponskem polju. [3, 9]

Poleg implementacije, ki jo je predstavil Sadakane [3], obstaja še izboljšava prostorske zahtevnost podatkovne strukture. Russo idr. [12] predstavil implementacijo priponskega drevesa, ki zniža prostorsko zahtevnost iz $|CSA|+6n+o(n)$ bitov na $|CSA|+n/((\log_\sigma \log n) \log n) \log n^2$ bitov (zmanjša za približno $6n$ bitov) pri tem pa poveča časovno zahtevnost operacij iz konstantnega časa na $O((\log_\sigma \log n) \log n)$. To isto podatkovno strukturo je Russo idr. [13] naredil dinamično. To pomeni da podatkovna struktura omogoča brisanje in dodajanje novega teksta T' v in iz priponskega drevesa. Podatkovna struktura potrebuje enak prostor kot statična verzija ($|CSA| + n/((\log_\sigma \log n) \log n) \log n)$, ampak časovna zahtevnost operacij se dvige iz $O((\log_\sigma \log n) \log n)$ na $O((\log_\sigma \log n) \log^2 n)$.

5 Zaključek

V članku so bila predstavljena priponska drevesa. Pri tem je bila predstavljena tudi izgradnja priponskih dreves. Ker je velikost priponskega drevesa sorazmerna velikosti besedila, lahko njegova velikost pri velikih besedilih preraste velikost pomnilnika, kar odločilno vpliva na hitrost iskanja. Zato je v članku bila predstavljena kompaktna predstavitev priponskih dreves, ki reši ta problem.

V nadaljevanju želimo še dodatno empirično primerjati priponska drevesa in kompaktna priponska drevesa. Primerjalo se bo izgradnja in uporaba obeh struktur. Pri izgradnji obeh struktur želimo primerjati čas izgradnje obeh struktur ter prostor, ki ga zasedeta obe strukturi. Pri uporabi obeh struktur pa želimo primerjati čas iskanja vzorcev v besedilih.

Literatura

[1] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, fourth edition.* MIT Press, 2022.

[3] K. Sadakane, "Compressed suffix trees with full functionality," *Theory of Computing Systems*, vol. 41, p. 589–607, Feb. 2007.

[4] G. Navarro, *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

[5] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pp. 1–11, 1973.

[2] σ predstavlja velikost uporabljene abecede Σ.

[6] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[7] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, p. 262–272, 4 1976.

[8] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, p. 249–260, Sept. 1995.

[9] N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen, "Engineering a compressed suffix tree implementation," in *Experimental Algorithms* (C. Demetrescu, ed.), (Berlin, Heidelberg), pp. 217–228, Springer Berlin Heidelberg, 2007.

[10] J. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pp. 118–126, 1997.

[11] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.

[12] L. M. S. Russo, G. Navarro, and A. L. Oliveira, "Fully-compressed suffix trees," in *LATIN 2008: Theoretical Informatics* (E. S. Laber, C. Bornstein, L. T. Nogueira, and L. Faria, eds.), (Berlin, Heidelberg), pp. 362–373, Springer Berlin Heidelberg, 2008.

[13] L. M. S. Russo, G. Navarro, and A. L. Oliveira, "Dynamic fully-compressed suffix trees," in *Combinatorial Pattern Matching* (P. Ferragina and G. M. Landau, eds.), (Berlin, Heidelberg), pp. 191–203, Springer Berlin Heidelberg, 2008.