Aho-Corasick algoritam za pretragu u tekstu

Ivan Pop-Jovanov

Septembar 2024.

Sažetak

Aho-Corasick algoritam pronalazi sva pojavljivanja nekog od konačnog broja unapred zadatih niski unutar teksta. Algoritam konstruiše konačan deterministički automat od zadatih niski u linearnom vremenu, i zatim ga koristi da paralelno traži sve niske u ciljnom tekstu u jednom prolazu kroz tekst. Složenost primene automata na tekst ne zavisi od broja traženih niski i linearna je po dužini teksta koji se pretražuje.

U ovom radu će biti prikazana implementacija algoritma, njegova upotreba, i analiza njegove prostorne i vremenske složenosti. Prateći kod se može naći na github repozitorijumu:

https://github.com/IvanPopJovanov/KIAA2-Aho-Corasick-2024

Sadržaj

1 Opis problema

Algoritam je predložen 1975. godine u radu koji su napisali Alfred V. Aho i Margaret J. Corasick.[1]

Neka je $K = \{y_1, y_2, ... , y_k\}$ konačan niz (koji ćemo zvati *rečnik*) niski (koje ćemo zvati *ključne reči*), i neka je x proizvoljna niska. Problem koji *Aho-Corasick* algoritam rešava je nalaženje svih pojavljivanja ključnih reči unutar teksta x. Izlaz iz algoritma predstavlja indekse pozicija pojavljivanja ključnih reči, koje mogu i da se preklapaju.

Algoritam radi u dva koraka. U prvom koraku se konstruiše konačan automat (koji ćemo zvati *mašina za pronalaženje obrazaca*) za rečnik K, a u drugom koraku se dobijeni konačni automat koristi za procesiranje niske x. Bitno je napomenuti da jednom ovako napravljen automat može da se primenjuje nad više različitih tekstova u kojima želimo da nađemo pojavljivanja ključnih reči.

1.1 Mašina za pronalaženje obrazaca

Mašina za pronalaženje obrazaca se sastoji od skupa *stanja* koja su obeležena brojevima, od pravila prelaska između tih stanja, i od skupa izlaznih vrednosti za svako stanje. Struktura sadrži tri funkcije:

1. *goto* funkciju `g(s,c)` koja za trenutno stanje `s`, i naredni karakter `c` vraća naredno stanje ili poruku *fail*.

2. *failure* funkciju `f(s)` koja za trenutno stanje `s` vraća u koje stanje treba preći u slučaju da *goto* funkcija vrati poruku *fail*.

3. *output* funkciju `output(s)` koja vraća skup ključnih reči koje su pronađene kada se dođe do stanja `s`. Ovaj skup može biti (i uglavnom će biti) prazan.

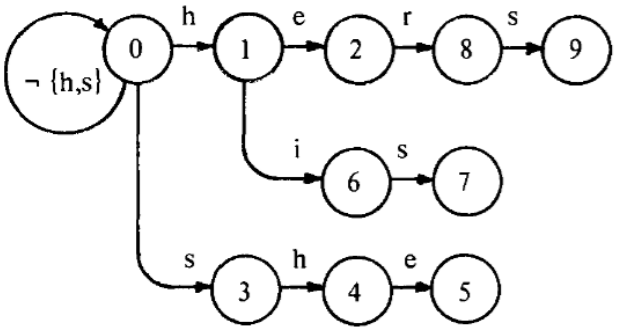
Ilustracija primera mašine za pronalaženje obrazaca i vrednosti opisanih funkcija za skup ključnih reči {he, she, his, hers} je data na slici 1.

1.2 Primena mašine za pronalaženje obrazaca

Nalaženje pojavljivanja ključnih reči unutar teksta x se sprovodi nizom iteracija nad pojedinačnim karakterima teksta. Jedna iteracija (ili *operacioni ciklus*) uzima u obzir trenutno stanje s, naredni karakter c, i odlučuje o narednom stanju i potencijalnom ispisu pronađenih ključnih reči.

Neka je $g(s, c) = s'$. Ako je $s' = fail$, trenutna iteracija se ponavlja nad stanjem $f(s)$. U suprotnom, prelazi se u stanje s' i ispisuje se sadržaj skupa $output(s')$, kao i trenutna pozicija u tekstu. Karakter c dobija vrednost narednog karaktera u tekstu.

Iako možda izgleda kao da se gubi linearna vremenska složenost zbog potencijalnog ponavljanja iteracija kada se naiđe na *fail* stanje, videćemo kasnije da je broj ovakvih situacija odozgo ograničen dvostrukom dužinom teksta koji se pretražuje, pa se zato održava željeno asimptotsko ponašanje.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Slika 1: Mašina za pronalaženje obrazaca, ilustracija preuzeta iz originalnog rada.[1] Prikazuje korišćene funkcije za skup ključnih reči {he, she, his, hers}.

Aho-Corasick algoritam se može posmatrati kao uopštenje KMP algoritma, zbog činjenice da kada rečnik K sadrži tačno jednu ključnu reč, ova dva algoritma postaju identični.

1.3 Konstrukcija mašine za pronalaženje obrazaca

Konstrukcija mašine za pronalaženje obrazaca se svodi na konstruisanje *goto*, *failure*, i *output* funkcija. U prvoj fazi konstruišemo skup stanja i funkciju *goto*, a u drugoj fazi funkciju *failure*. Funkcija *output* se delimično računa u prvoj, a delimično u drugoj fazi.

Prva faza se sastoji od građenja grafa stanja. Počinjemo sa jednim čvorom koji predstavlja *nulto* stanje. U graf dalje dodajemo jednu po jednu ključnu reč. Prolazimo karakter po karakter kroz ključnu reč i počevši od nultog čvora dodajemo nove čvorove tako da postoji usmereni put kroz graf koji počinje od korena i pređenim granama prati karaktere ključne reči. U *output* funkciju sta-

nja koje se nalazi na kraju takvog puta dodajemo samu ključnu reč. U koliko je u procesu dodavanja ključnih reči potrebno dodati granu koja već postoji, jednostavno pratimo postojeću granu i nastavljamo konstrukciju ostatka trenutne reči. Kada prođemo kroz sve ključne reči, na kraju je potrebno dodati grane koje koren grafa preslikavaju u samog sebe za sve karaktere koji ne započinju ni jednu ključnu reč (ovom akcijom graf prestaje da predstavlja stablo). Primer ovako dobijenog grafa može se videti na slici 1 u delu pod (a). Ovaj graf predstavlja *goto* funkciju.

U drugoj fazi je potrebno da konstruišemo *failure* funkciju i da završimo konstrukciju *output* funkcije koju smo započeli u prvoj fazi. Definišimo dubinu stanja kao najmanji broj grana u grafu koji je potreban da bi se došlo do njega počevši od nultog stanja. Funkciju *failure* konstruišemo rekurzivno, tako što njenu vrednost nekog stanja određene dubine računamo pomoću njene vrednosti za sva stanja manjih dubina. Bazu rekurzivne funkcije predstavljaju stanja dubine jedan, za koje će njena vrednost biti stanje nula. U stanju nula je funkcija nedefinisana, zbog činjenice da *goto* funkcija nikada neće vratiti vrednost *fail*. Pretpostavimo da znamo vrednosti *failure* funkcije za sva stanja dubine manje ili jednake od d. Želimo da izračunamo vrednosti na dubini $d + 1$. Prolazimo redom kroz svako stanje r dubine d. Neka je s neko stanje takvo da $g(r, a) = s$ za neko a. Proveravamo da li $g(f(r), a)$ nije jednako *fail*. Ako nije, postavljamo $f(s) = g(f(r), a)$. Ako jeste, proveravamo isti uslov za $r = f(r)$, i tako radimo u petlji dok ne naiđemo na stanje koje zadovoljava uslov. Ovakvo stanje sigurno postoji zato što u najgorem slučaju se vraćamo do nultog stanja za koje *goto* funkcija nikad ne vraća *fail*. Kad god nađemo da je $f(s) = s'$, *output* vrednosti za stanje s' dodajemo i u *output* za stanje s.

Pseudokodovi algoritama za konstrukciju ove tri funkcije mogu se naći u originalnom radu[1].

2 Implementacija

Prateći kod ovog rada se može naći na github repozitorijumu:
https://github.com/IvanPopJovanov/KIAA2-Aho-Corasick-2024

Unutar repozitorijuma je priloženo nekoliko različitih implementacija Aho-Corasick algoritma, u zavisnosti od toga koje strukture su korišćene za predstavljanje mašine za pronalaženje obrazaca.

Prva implementacija se nalazi u klasi `PatternMatchingMachine`. Ova klasa ima za cilj da prati idejnu strukturu algoritma, žrtvujući u procesu efikasnost. Graf konačnog automata je predstavljen povezanim čvorovima stanja (klasa `State`). Unutar jednog čvora su smeštene *goto*, *failure*, i *output* funkcije koje vraćaju očekivane vrednosti za to konkretno stanje. Klasa `PatternMatchingMachine` čuva niz stanja, popunjava njihove vrednosti, i implementira Aho-Corasick algoritam. Klasi se konstruktorom prosleđuju skup K i niska x, i poziva se pretraga metodom `match()`. Isti interfejs korišćenja će biti prisutan i u preostalim klasama.

Druga implementacija se nalazi u klasi `PmmMatrix`. Ovaj put stanja nisu

implementirana kao objekti, nego se koristi njihov broj za identifikaciju. *Goto* funkcija je implementirana preko matrice

```
int g[MAX_STATES][ALPHABET_SIZE];
```

koja čuva stanja u koja se prelazi za svaki mogući par stanja i karaktera. Vrednost *fail* je predstavljena vrednošću -1. Funkcija *failure* je predstavljena nizom

```
int f[MAX_STATES];
```

koji čuva njenu vrednost za svako stanje. Funkcija *output* je predstavljena preko dva niza

```
bool output[MAX_STATES];
std::list<std::string> outputs[MAX_STATES];
```

gde `output` čuva da li stanje predstavlja kraj neke od ključnih reči, a `outputs` čuva listu ključnih reči koje treba da se ispišu za konkretno stanje.

Treća implementacija se nalazi u klasi `PmmTree`, u kojoj je *goto* funkcija predstavljena binarnim pretraživačkim stablom unutar strukture `std::map` na sledeći način:

```
std::map<char, int> g[MAX_STATES];
```

3 Analiza složenosti

Ukupna složenost algoritma je $O(m+n+r)$, gde je m ukupna dužina ključnih reči, n dužina teksta koji se pretražuje, a r broj pronađenih pojavljivanja. Algoritam spada u klasu algoritama izlazno-zavisne složenosti. Razmotrimo kako se dolazi do ove složenosti u sekcijama 3.1 i 3.2.

3.1 Složenost pretrage teksta

U svakom operacionom ciklusu (iteraciji) pravimo nula ili više promena stanja *failure* funkcijom, i tačno jednu promenu stanja *goto* funkcijom. Dubina trenutnog stanja se povećava za tačno jedan pri svakom pozivu *goto* funkcije, a smanjuje za jedan ili više pri svakom pozivu *failure* funkcije. Ako uzmemo u obzir da dubina ne može da bude manja od nule, dolazimo do zaključka da za svaki poziv *failure* funkcije, morala je prethodno bar jednom biti pozvana *goto* funkcija, odnosno da je broj poziva *failure* funkcije strogo manji od broja poziva *goto* funkcije. Zbog toga je ukupan broj iteracija odozgo ograničen sa $2n$, gde je n broj iteracija, odnosno broj karaktera u tekstu koji pretražujemo. Moguće je dodatno ubrzati algoritam korišćenjem determinističkog konačnog automata, u kom slučaju se potpuno eliminiše *failure* funkcija, i tada je broj iteracija jednak tačno n.

Složenost pojedinačne iteracije zavisi od izbora strukture koja je korišćena za implementaciju *goto*, *failure*, i *output* funkcija. Najveća (vremenska) efikasnost se postiže kada su prve dve funkcije implementirane pomoću nizova, a treća

pomoću liste, mada se žrtvuje prostorna efikasnost. U tom slučaju je složenost jedne iteracije konstantna, $O(c) = O(1)$, pa je ukupna vremenska složenost pretrage $O(n)$.

Bolja prostorna složenost se može postići, na primer, implementacijom *goto* funkcije preko binarnog pretraživačkog stabla. U tom slučaju ukupna vremenska složenost pretrage postaje $O(nlogn)$.

Bitno je napomenuti da ovde govorimo o složenosti same pretrage, a ne i ispisa rešenja. U opštem slučaju, broj pronađenih niski može da bude u $O(n^2)$. Na primer:

$$K = \{a, aa, aaa, aaaa\}$$

$$x = aaaaa$$

Očigledno, ni jedan algoritam u ovakvoj situaciji ne može da izbegne kvadratnu složenost.

3.2 Složenost konstrukcije mašine za pronalaženje obrazaca

Konstrukcija *goto* funkcije redom prolazi karakter po karakter kroz svaku ključnu reč. Dakle, složenost je linearna po ukupnoj dužini ključnih reči. Odavde takođe sledi i da je broj stanja u automatu najviše za jedan veći (zbog nultog stanja) od ukupne dužine ključnih reči.

Konstrukcija *failure* funkcije složenost dobija iz rekurzivnog spuštanja niz dubinu pri koraku $r = f(r)$. Ukupan broj pozivanja ovog koraka je takođe ograničen ukupnom dužinom ključnih reči. Ovaj dokaz je malo komplikovaniji, ali intuitivno je sličan dokazu o složenosti pretrage u sekciji 3.1. Kada se računa *failure* funkcija za stanje s, gleda se stanje manje dubine r koje pokazuje na stanje s, odnosno $g(r, a) = s$. Računanjem vrednosti *failure* funkcije za stanje r je pri spuštanju niz dubinu korakom $r = f(r)$ ovo prethodno stanje preskočilo neke nivoe, i kako sada $f(r)$ pokazuje na neko stanje potencijalno nekoliko nivoa niže, pri računanju $f(s)$ će ti nivoi moći direktno da se preskoče.

Konstrukcija *output* funkcije može da se implementira preko povezanih listi, tako da operacija unije ima linearnu složenost.

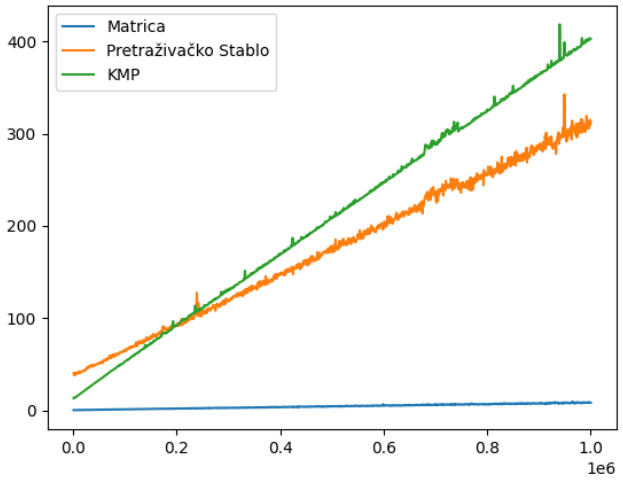
4 Evaluacija

U ovoj sekciji će biti prikazana dva poređenja nad tri algoritma. Algoritmi koji se porede su:

1. Aho-Corasick sa matričnom implementacijom *goto* funkcije
2. Aho-Corasick sa implementacijom *goto* funkcije korišćenjem samobalansirajućih binarnih pretraživačkih stabala
3. KMP algoritam

Poređenje algoritama je vršeno nad nasumično generisanim tekstom unapred zadate dužine, i nasumično generisanim ključnim rečima, gde su zadati broj ključnih reči, minimalna dužina ključne reči i maksimalna dužina ključne reči. Ključne reči ne moraju da se nalaze u tekstu, a mogu i da se pojave više puta ili da se preklapaju.

U prvom poređenju su fiksirani broj ključnih reči na 100, minimalna dužina na 5, i maksimalna dužina na 15. Dužina teksta koji se pretražuje ide od 10^3 do 10^6 i u svakoj iteraciji se zapisuje dužina izvršavanja. Grafik sa rezultatima se može videti na slici 2.

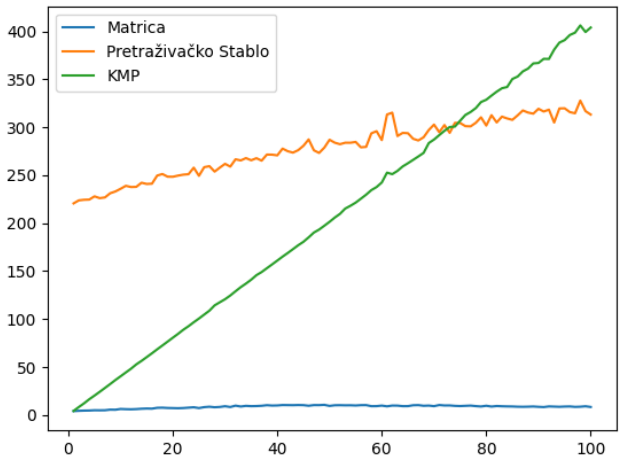


Slika 2: Variranje dužine teksta koji se pretražuje. X osa predstavlja broj karaktera u niski koja se pretražuje, a Y osa predstavlja dužinu izvršavanja u milisekundama.

Možemo primetiti da se algoritmi ponašaju tačno onako kako smo i očekivali na osnovu analize složenosti. Vreme izvršavanja za sva tri algoritma raste linearno u odnosu na dužinu teksta. Matrična implementacija je ubedljivo najefikasnija zbog direktnog pristupa narednom stanju. Za male tekstove pretraživačko stablo je sporije od KMP algoritma zbog konstantnog faktora koji dodaje pretraga kroz stablo, ali za veće tekstove ipak postaje brže, što je i očekivano.

U drugom poređenju je fiksirana dužina teksta koji se pretražuje na 10^6, a minimalna dužina ključne reči na 5, i maksimalna dužina na 15. Broj ključnih reči ide od 1 do 100 i u svakoj iteraciji se zapisuje dužina izvršavanja. Grafik sa rezultatima se može videti na slici 3.

Ovaj primer je verovatno čak i zanimljiviji od prvog, jer se suština Aho-Corasick algoritma baš nalazi u činjenici da složenost ne zavisi mnogo od broja ključnih reči. Matrična implementacija radi gotovo u konstantnom vremenu zato što vreme utrošeno na pretragu dominira nad vremenom utrošenim na konstruk-



Slika 3: Variranje broja ključnih reči koje tražimo. X osa predstavlja broj ključnih reči, a Y osa predstavlja dužinu izvršavanja u milisekundama.

ciju automata. Implementacija preko pretraživačkog stabla raste logaritamski u odnosu na broj ključnih reči zbog operacija pretrage i umetanja. KMP za svaku novu ključnu reč mora da pretražuje iznova ceo tekst i zato raste linearno sa brojem ključnih reči.

5 Primene

U originalnom radu[1], predstavljen je primer primene Aho-Corasick algoritma u kontekstu Bibliografske pretrage. Istraživačima je bilo potrebno da efikasno pretražuju bazu podataka od oko 150 hiljada naučnih radova, ukupne dužine otprilike 10^7 karaktera. Bilo je potrebno pronaći sve radove koji sadrže neke od ključnih reči koje korisnik unese u program za pretragu. Korišćenje Aho-Corasick algoritma je ubrzalo pretragu 5 do 10 puta u odnosu na prethodnu implementaciju (što je više ključnih reči bilo traženo od jednom, to je ubrzanje bilo primetnije). Takođe je moguće izmeniti algoritam da umesto fiksnih ključnih reči radi sa podskupom *regex* izraza.

Naredni primeri su preuzeti sa veb sajta *Algorithms for Competitive Programming*[2].

- **Nalaženje leksikografski najmanje niske date dužine koja ne sadrži ni jednu od ključnih reči**

  Konstruišemo konačni automat od datih ključnih reči, i zatim obrišemo sve grane koje direktno vode ka stanjima koja imaju neprazan *output* skup.

U novonastalom grafu jednostavno pronađemo leksikografski najmanji put tražene dužine, tako što pratimo granu leksikografski najmanjeg karaktera u svakoj iteraciji. Složenost je $O(L)$, gde je L tražena dužina.

- **Nalaženje najkraće moguće niske koja sadrži sve ključne reči**

  Algoritmom BFS možemo pretraživati graf dok ne naiđemo na neki put koji sadrži u *output* skupovima sve ključne reči. Prvi nađen ovakav put će biti i najkraći, zbog činjenice da će BFS algoritmom svi kraći putevi biti već pretraženi.

- **Nalaženje leksikografski najmanje niske određene dužine koja sadrži tačno k ključnih reči**

  Ovaj put radimo DFS pretragu do tražene dužine, dok ne naiđemo na prvi put koji sadrži tačno k ključnih reči u uniji *output* skupova. Ako obilazimo grane abecednim redom, dobijena niska će biti najmanja takva leksikografski.

6 Zaključak

Aho-Corasick algoritam je primetno brži na konkretnom problemu za koji je kreiran. Jedno od mogućih daljih unapređenja koje nisam implementirao za ovaj projekat je prevođenje nedeterminističkog konačnog automata u deterministički konačni automat. U teoriji, takva promena bi eliminisala *failure* prelaske između stanja, i time efektivno prepolovila vreme pretrage, ali u praksi automat većinu vremena provodi u nultom stanju gde i ovako ne pravi dodatne iteracije, pa takva optimizacija i ne donosi mnogo poboljšanja.

Takođe mi je žao što nisam uspeo da nađem elegantno rešenje za automatsko računanje prostorne zahtevnosti različitih struktura u C++ jeziku, jer bi time dobijeni grafici prikazali prednost korišćenja stabala pretrage umesto matrica, koja samo gledano u odnosu na vremensku složenost izgledaju kao dosta gora opcija.

Zanimljivo je napomenuti da postoji i unapređena verzija ovog algoritma, predložena od strane Bertranda Mejera 1985. godine, koja dozvoljava naknadnu izmenu skupa ključnih reči bez ponovne konstrukcije automata[3].

Aho-Corasick algoritam je predstavljao osnovu za Unix komandu `fgrep`.

Literatura

[1] Aho, Alfred V. i Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search." Communications of the ACM 18, br. 6 (1975): 333-340.

[2] Aho-Corasick algorithm - Algorithms for Competitive Programming. cp-algorithms.com/string/aho_corasick.html.

[3] Meyer, Bertrand. "Incremental string matching."Information Processing Letters 21, br. 5 (1985): 219-227.