Prirodno Matematički Fakultet u Nišu

Traženje uzorka u tekstu

Student:
Nikola Milosavljević

Profesor:
Predrag Stanimirović

Niš
Februar, 2009.

Sadržaj

1 Uvod

Čest je slučaj da je za neki element potrebno ispitati da li pirpada datom skupu. Algoritam pomoću kojeg će se to odrediti, kao i njegova efikasnost, zavisi od osobina samog elementa kao i od strukture datog skupa. Karakterističan je slučaj kada se radi o *stringovima*, pri čemu problem postaje pronalaženje neke specificirane reči ili uzorka u datom tekstu.

Mnogi editori danas raspolažu nizom funkcija za manipulaciju teksta. Jedna od karakterističnih funkcija je *search* koja služi da se pronađe specificirani uzorak (najčešće jedna reč) u nekom dokumentu. Ovi objekti mogu biti prilično veliki (u većim dokumentima, broj slova je često reda veličine 10^6) pa efikasni algoritmi imaju veliku ulogu u njihovoj manipulaciji. Za velike dokumente, uglavnom je neprihvatljiva funkcija pretraživanja čija je složenost veća od linearne.

Problem nalaženja uzorka u tekstu ima primenu i u drugim oblastima, na primer u molekularnoj biologiji kada je potrebno izdvojiti neke uzorke u okviru velikih molekula DNK ili RNK, ili kada je, na primer, potrebno pronaći određeni podniz u nekom velikom nizu brojeva (npr. među ciframa iracionalnog broja). Takođe, problem se može proširiti i u $2D$ kada je potrebno obrađivati slike.

1970. godine Kuk (S. A. Cook) je teoretski dokazao da postoji apstraktna mašina koja pronalazi uzorak dužine M u tekstu dužine N u složenosti $O(N + M)$. Knut (D. E. Knuth) i Prat (V. R. Pratt) su na osnovu ideja Kukovog dokaza uspeli da pronađu praktičan algoritam sa datom složenošću. Istovremeno, ali nezavisno od njih, Moris (J. H. Morris) je otkrio isti algoritam na drugačiji način. Knut, Moris i Prat nisu objavili svoj algoritam do 1976. a u međuvremenu su Bojer (R. S. Boyer) i Mur (J. S. Moore) otkrili algoritam iste složenosti ali koji se pokazivao nešto brži u praksi. 1980. Karp (R. M. Karp) i Rabin (M. O. Rabin) su izmislili algoritam čije je prosečna složenost linearna, ali koristi znatno dugačije ideje.

2 Osnovni pojmovi i definicije

2.1 String

Neka je $\mathbf{\Sigma}$ neprazan skup, koji nazivamo **alfabetom**, a njegove elemente **simbolima (slovima)**.

Definicija 2.1 Reč (string) *nad alfabetom* $\mathbf{\Sigma}$ *je konačan niz*

$$x_1 x_2 \cdots x_n$$

gde je $n \in \mathbb{N}$ *i* $x_i \in \mathbf{\Sigma}$ *za svako* $i \in \{1, 2, \ldots, n\}$.

Definicija 2.2 *Dužina stringa* x*, u oznaci* $|x|$*, je broj simbola u njegovom zapisu.*

Definicija 2.3 *Dva stringa* $u = x_1 x_2 \cdots x_n$ *i* $v = y_1 y_2 \cdots y_m$ *su jednaka akko je* $n = m$ *i* $x_i = y_i$*, za svako* $i \in \{1, 2, \ldots, n\}$.

Skup svih stringova nad alfabetom $\mathbf{\Sigma}$ označavamo sa $\mathbf{\Sigma}^+$.

Definicija 2.4 *Konkatenacija (spajanje) je funkcija* $f : \mathbf{\Sigma}^+ \times \mathbf{\Sigma}^+ \to \mathbf{\Sigma}^+$ *tako da za svaka dva stringa* $u = x_1 x_2 \cdots x_n$ *i* $v = y_1 y_2 \cdots y_m$ *iz* $\mathbf{\Sigma}^+$ *važi*

$$f(u, v) = x_1 x_2 \cdots x_n y_1 y_2 \cdots y_m$$

Umesto $f(x, y)$ *češće se piše* xy.

Definicija 2.5 *Neka je* ε *element takav da* $\varepsilon \notin \mathbf{\Sigma}^+$*, koji nazivamo* **prazan string**. *Tada pišemo* $\mathbf{\Sigma}^* = \mathbf{\Sigma}^+ \cup \{\varepsilon\}$ *i dodefinišemo operaciju spajanja sa* $u\varepsilon = u$ *i* $\varepsilon u = u$ *za svako* $u \in \mathbf{\Sigma}^*$. *Uzima se da je* $|\varepsilon| = 0$.

2.2 Složenost algoritma

Za domen svake od funkcija iz definicija se podrazumeva skup $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$.

Definicija 2.6 *Za datu funkciju* $g(n)$*,* $O(g(n))$ *(veliko "O" od* $g(n)$*) predstavlja skup funkcija* $f(n)$ *za koje postoje pozitivne konstante* c *i* n_0 *tako da važi* $0 \leq f(n) \leq cg(n)$*, za sve* $n \geq n_0$.

O-notaciju koristimo kada želimo da odredimo **gornju granicu** neke funkcije f do na konstantu.

Definicija 2.7 *Za datu funkciju* $g(n)$*,* $\Omega(g(n))$ *(veliko Omega od* $g(n)$*) predstavlja skup funkcija* $f(n)$ *za koje postoje pozitivne konstante* c *i* n_0 *tako da važi* $0 \leq cg(n) \leq f(n)$*, za sve* $n \geq n_0$.

Ω-notaciju koristimo kada želimo da odredimo **donju granicu** neke funkcije f do na konstantu.

Definicija 2.8 *Za datu funkciju $g(n)$, $\Theta(g(n))$ (Teta od $g(n)$) predstavlja skup funkcija $f(n)$ za koje postoje pozitivne konstante c_1, c_2 i n_0 tako da važi $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, za sve $n \geq n_0$.*

Θ-notacija najbolje određuje funkciju, jer važi $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$.

Umesto izraza $f(n) \in \Theta(g(n))$ (analogno za O i Ω) koristi se izraz $f(n) = \Theta(g(n))$ koji ima svoje prednosti.

3 Tekst i uzorak

3.1 Formalizacija

Da bismo efektivno opisali algoritme za nalaženje uzorka u tekstu, moramo formalizovati problem.

Pretpostavimo da je tekst niz $T[1..n]$ dužine n i da je uzorak niz $P[1..m]$ dužine $m \leq n$. Dalje pretpostavimo da su elementi nizova P i T simboli iz konačnog alfabeta Σ. Na primer, možemo imati $\Sigma = \{0, 1\}$ ili $\Sigma = \{a, b, \ldots, z\}$. Primetimo da ovako definisani, P i T predstavljaju stringove.

Kažemo da se uzorak P **pojavljuje sa pomerajem (eng. shift)** s u tekstu T (ekvivalentno, uzorak P **se pojavljuje počevši od pozicije** $s+1$ u tekstu T) ako je $0 \leq s \leq n-m$ i $T[s+1..s+m] = P[1..m]$ (tj. $T[s+i] = P[i]$ za $1 \leq i \leq m$). Ako se P pojavljuje sa pomerajem s u T tada za s kažemo da je **korektan pomeraj**; inače, kažemo da je s **nekorektan pomeraj**. Problem upoređivanja stringova je problem nalaženja svih korektnih pomeraja s sa kojima se dati uzorak P pojavljuje u datom tekstu T.

Slika 1: Pomeraj $s = 3$ je korektan za $T =$ abcabaabcabac i $P =$ abaa.

Osim naivnog Brute-Force algoritma, mnogi algoritmi za ovu klasu problema prvo obavljaju neko izračunavanje na osnovu uzorka, a zatim traže sve korektne pomeraje. Prvu fazu ćemo zvati "preprocesiranje" a drugu "upoređivanje". Tabela 1 pokazuje preprocesirajuće i upoređivajuće vreme poznatijih algoritma za pronalaženje uzorka u tekstu.

Algoritam	*Preprocesiranje*	*Upoređivanje*
Naivni	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Konačni automat	$O(m\|\Sigma\|)$	$\Theta(n)$
KMP	$\Theta(m)$	$\Theta(n)$
Bojer-Mur	$\Theta(m)$	$\Theta(n)$

Tabela 1: Vreme preprocesiranja i upoređivanja poznatijih algoritama

Složenost svakog algoritma jednaka je zbiru složenosti preprocesiranja i upoređivanja. Iz tabele vidimo da je složenost naivnog algoritma u najgorem slučaju $\Theta((n-m+1)m)$ baš kao i kod Rabin-Karp algoritma. Međutim, ovaj drugi mnogo bolje radi u proseku i u praksi. Takođe, postoji interesantan algoritam u obliku konačnog automata čije vreme preprocesiranja zavisi od alfabeta ali zato za upoređivanje troši samo $\Theta(n)$ vremena. Ipak, najbolji pristup ima KMP algoritam koji, poput konačnog automata, za upoređivanje takođe troši $\Theta(n)$ vremena ali redukuje vreme preprocesiranja na svega $\Theta(m)$ što ga čini najefikasnijim algoritmom iz ove grupe i jednim od najboljih za ovu klasu problema. Bojer-Mur algoritam radi na sličnom principu kao KMP, s tim što posmatra uzorak sa desna na levo.

3.2 Notacija i terminologija

Koristićemo oznaku $\mathbf{\Sigma}^*$ ("sigma zvezda") za označavanje skupa svih stringova konačne dužine sastavljene od karaktera alfabeta $\mathbf{\Sigma}$. Dakle, razmatramo samo stringove konačne dužine. **Prazan string**, dužine 0, u oznaci ε, takođe pripada $\mathbf{\Sigma}^*$. Dužinu stringa x označavamo sa $|x|$. Za $a \in \mathbf{\Sigma}^*$ uvodimo oznaku $a^n = \underbrace{aa\ldots a}_{n}$.

Definicija 3.1 *String y je* **prefiks** *stringa x, u oznaci $y \sqsubset x$, ako postoji string $w \in \mathbf{\Sigma}^*$ tako da je $x = yw$. Analogno, string y je* **sufiks** *stringa x, u oznaci $y \sqsupset x$, ako postoji string $w \in \mathbf{\Sigma}^*$ tako da je $x = wy$.*

Na primer **ab** $\sqsubset$ **abcca** i **cca** $\sqsupset$ **abcca**. Primetimo da ako je $y \sqsubset x$ ili $y \sqsupset x$ tada je $|y| \leq |x|$. Prazan string ε je i prefiks i sufiks svakom stringu. Korisno je primetiti da je za proizvoljne stringove x i y i proizvoljni simbol a važi $x \sqsupset y$ akko $xa \sqsupset ya$. Nije teško dokazati da su $\sqsubset$ i $\sqsupset$ tranzitivne relacije.

Sledeća lema biće korisna kasnije.

Lema 3.2 (Lema o preklapajućim sufiksima) *Neka su x, y i z stringovi takvi da je $x \sqsupset z$ i $y \sqsupset z$. Ako je $|x| \leq |y|$ tada je $x \sqsupset y$. Ako je $|x| \geq |y|$ tada je $y \sqsupset x$. Ako je $|x| = |y|$ tada je $x = y$.*

Dokaz: Jednostavno grafičko predstavljanje stringova x, y i z je dovoljno da bi se uverili u korektnost leme. □

Zbog sažetosti notacije, u daljem tekstu ćemo k-slovni prefiks $P[1..k]$ uzorka $P[1..m]$ označavati sa P_k. Prema tome $P_0 = \varepsilon$ i $P_m = P = P[1..m]$. Analogno definišemo k-slovni prefiks teksta T kao T_k. Koristeći ovu notaciju mozemo problem upoređivanja stringova preformulisati kao nalaženje svih pomeraja s, $0 \leq s \leq n - m$, takve da je $P \sqsupset T_{s+m}$.

U pseudokodovima upoređivanje stringova jednakih dužina uzeto je za primitivnu operaciju. Ako se stringovi upoređuju sa leva na desno (znak po znak) i upoređivanje prestaje kada dođe do neslaganja, pretpostavljamo da je utrošeno vreme za to upoređivanje linearna funkcija od broja poklapajućih znakova. Preciznije, pretpostavlja se da se za test "$x = y$" utroši $\Theta(t + 1)$ vremena, gde je t dužina najdužeg stringa z takvog da je $z \sqsubset x$ i $z \sqsubset y$. (Pišemo $\Theta(t+1)$ umesto $\Theta(t)$ da bi smo uključili granični slučaj $t = 0$. Prvi znakovi se ne poklapaju ali je potrebna pozitivna količina vremena za ovo upoređivanje.)

4 Naivni algoritam

4.1 Opis

Naivni algoritam nalazi sve korektne pomeraje koristeći petlju koja proverava uslov $P[1..m] = T[s + 1..s + m]$ za svaku od $n - m + 1$ mogućih vrednosti pomeraja s. Ukoliko dođe do poklapanja, pomeraj se štampa.

Naive-String-Matcher(T, P)
```
1   n ← length[T]
2   m ← length[P]
```

```
3  for s ← 0 to n − m
4     do if P[1..m] = T[s + 1..s + m]
5        then print "uzorak se pojavljuje sa pomerajem" s
```

Ova procedura se može interpretirati grafički kao da uzorak "klizi" ispod teksta, detektujući one pomeraje za koje je odgovarajući deo teksta iznad uzorka jednak samom uzorku. **For** petlja u liniji 3 eksplicitno ispituje sve moguće pomeraje. Test u liniji 4 određuje da li je trenutni pomeraj validan ili ne; on obuhvata petlju za proveru poklapanja znaka na odgovarajućoj poziciji dok se sve pozicije ne poklope ili dok ne dođe do prvog neslaganja. Linija 5 štampa svaki korektan pomeraj.

Složenost procedure NAIVE-STRING-MATCHER je očigledno $O((n - m + 1)m)$ i ova granica je dostižna u najgorem slučaju. Na primer, uočimo tekst a^n i uzorak a^m. Za svaku od $n - m + 1$ mogućih vrednosti pomeraja s, implicitna petlja za upoređivanje iz linije 4 mora se izvršiti m puta da bi potvrdila korektnost pomeraja. Prema tome, u najgorem slučaju je složenost $\Theta((n - m + 1)m)$, što je za $m \approx \frac{n}{2}$ upravo $\Theta(n^2)$. Ukupna složenost naivnog algoritma jednaka je složenosti upoređivanja jer preprocesiranja nema.

Kao što ćemo videti, NAIVE-STRING-MATCHER nije optimalna procedura za ovu vrstu problema. Glavni nedostatak ovog algoritma je u tome što se dobijene informacije o tekstu za jedan pomeraj s u potpunosti ignorišu prilikom razmatranje sledećeg pomeraja $s + 1$. Na primer, ako je $P = aaab$ i ustanovili smo da je $s = 0$ korektan pomeraj, tada nijedan od pomeraja 1, 2 ili 3 nisu korektni jer je $P[4] = b \neq a$, ali će ih naivni algoritam svejedno ispitati. Ili ako je, na primer, $P = 00000001$ i $T = 00\ldots001$ (50 nula) tada će brojač kod funkcije za upoređivanje biti menjan (51 - 8 + 1)7 puta, tj. nigde se neće koristiti činjenica da ako je pomeraj s korektan, onda se prvih 7 znakova (nule) poklapaju i u pomeraju $s + 1$. U narednim poglavljivam razmotrićemo nekoliko načina za efektivnu upotrebu takvih informacija.

Istina, degenerisani stringovi iz primera su retki u običnim tekstovima ali je algoritam znatno sporiji ako se koristi na binarnim tekstovima, npr. prilikom obrađivanja slika ili sistemskih aplikacija. Sa druge strane, može se pokazati da je za **slučajno** izabrane P i T iz $\mathbf{\Sigma}^*$, pri čemu važi $|\Sigma| = d \geq 2$, očekivani (ukupni) broj upoređivanja implicitne petlje iz linije 4 naivnog algoritma jednak

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1).$$

Prema tome, za slučajno izabrane stringove, naivni algoritam je prilično efikasan.

5 Rabin-Karp

5.1 Brojevi i stringovi

Rabin i Karp su predložili algoritam za traženje uzorka u tekstu koji se pokazuje kao veoma efikasan u praksi i generalizuje se za srodne probleme. Složenost preprocesiranja ovog algoritma je $\Theta(m)$, dok je ukupna složenost u najgorem slučaju $\Theta((n-m+1)m)$.

Bez umanjenja opštosti možemo pretpostaviti da je $|\Sigma| = d$ i da je $\Sigma = \{0, 1, \ldots, d-1\}$ (uvek možemo izvršiti takvo preslikavanje nad Σ). Dakle, pretpostavljamo da svaki znak iz Σ predstavlja jednu cifru u sistemu sa osnovom d. Prema tome, string od k uzastopnih znakova iz Σ predstavlja k-cifreni broj iz sistema sa osnovom d (npr. za $d = 10$, string "31415" posmatramo kao broj 31 415).

Za dati uzorak $P[1..m]$, neka p predstavlja njegovu vrednost u sistemu sa osnovom d (sam broj p ćemo zapisivati kao decimalan broj, tj. u sistemu sa osnovom 10). Slično, za dati tekst $T[1..n]$ neka t_s prestavlja vrednost stringa $T[s+1..s+m]$ za $0 \leq s \leq n-m$. Očigledno, $t_s = p$ ako i samo ako je $T[s+1..s+m] = P[1..m]$; sledi da je s korektan pomeraj ako i samo ako je $t_s = p$. Ako bismo mogli da izračunamo p u složenosti $\Theta(m)$ i sve vrednosti t_s u $\Theta(n-m+1)$, tada bismo mogli da odredimo sve korektne pomeraje s u složenosti $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ tako što ćemo upoređivati p sa svakim od t_s. (Za sada, zanemarimo činjenicu da p i t_s mogu biti veoma veliki brojevi.)

Možemo izračunati p u složenosti $\Theta(m)$ na sledeći način:

$$p = P[m] + d(P[m-1] + d(P[m-2] + \ldots + d(P[2] + dP[1]))\ldots) \qquad (1)$$

Potpuno analogno možemo izračunati t_0, vrednost stringa $T[1..m]$, u složenosti $\Theta(m)$. Da bi se izračunale ostale vrednosti $t_1, t_2, \ldots, t_{n-m}$ u složenosti $\Theta(n-m)$ dovoljno je primetiti da se t_{s+1} može izračunati uz pomoć t_s u konstantnom vremenu jer je

$$t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1] \qquad (2)$$

Na primer, neka je $m = 5$, $d = 10$ (običan dekadni sistem), $t_s = 31415$ i neka je sledeći broj $T[s + 5 + 1] = 2$. Dakle, potrebno je ukloniti nastariju cifru 3, i dodati na kraju cifru 2.

$$t_{s+1} = 10(t_s - 10^4 T[s+1]) + T[s+5+1] = 10(31415 - 10000 \cdot 3) + 2 = 14152.$$

Oduzimanjem $d^{m-1}T[s+1]$ uklanjamo cifru najveće težine iz t_s, množenjem sa d "pomeramo" broj ulevo za jednu poziciju i dodavanjem $T[s + m + 1]$ se odgovarajuća cifra najmanje težine dovodi na svoje mesto. Ako se konstanta d^{m-1} izračuna na početku (može u $O(\lg m)$ ali je dovoljno u $\Theta(m)$) tada svako izvršavanje jednačine (2) koristi konstantan broj aritmetičkih operacija. Prema tome, možemo izračunati p i t_0 u vremenu $\Theta(m)$ i $t_1, t_2, \ldots, t_{n-m}$ u vremenu $\Theta(n - m + 1)$, pa možemo naći sva pojavljivanja uzorka P u tekstu T koristeći $\Theta(m)$ vremena za preprocesiranje i $\Theta(n - m + 1)$ vremena za upoređivanje.

5.2 Algoritam

Očigledno, glavni problem opisane procedure je u tome što brojevi p i t_s mogu biti previše veliki što onemogućava lak rad sa njima. Ako P sadrži m znakova, tada je pretpostavka da svaka aritmetička operacija nad p (koji ima m "cifara" od kojih svaka može biti višecifren broj u decimalnom sistemu) troši "konstantno vreme" potpuno nerazumna.

Srećom, postoji lek za ovaj problem: izračunati p i t_s-ove po modulu nekog odgovarajućeg modula q. Kako se i izračunavanja p, t_0 i rekurentna jednakost (2) mogu raditi preko modula q, zaključujemo da možemo izračunati p po modulu q u složenosti $\Theta(m)$ i sve t_s-ove po modulu q u složenosti $\Theta(n-m+1)$. Za moduo q se obično bira prost broj tako da je veličina dq približno jednaka veličini kompjuterske reči (opsega) koju koristimo za pamćenje brojeva (ali ne veća). To nam omogućava da sve operacije izvršavamo sa sigurnošću da neće doći do prekoračenja i maksimalno koristimo odgovarajući opseg. Sada nam rekurentna jednačina (2) postaje:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \ \ mod \ \ q \tag{3}$$

gde je $h = d^{m-1} \ mod \ q$.

Slika 2: (a) Primer izračunavanja t_6 za $d = 10$ i $q = 13$. Pretpostavljamo da je $P = 31415$ (osenčen deo). (b) Korektno i slučajno poklapanje. (c) Izračunavanje t_{s+1} preko t_s.

Rešenje sa modulom q nije savršeno jer činjenica $t_s \equiv p \pmod{q}$ ne implicira $t_s = p$. Sa druge strane, ako je $t_s \not\equiv p \pmod{q}$, definitivno znamo da je $t_s \neq p$, što znači da je pomeraj s nekorektan. Prema tome, možemo koristiti test $t_s \equiv p \pmod{q}$ za izbacivanje nekorektnih pomeraja s. Svaki pomeraj s za koji važi $t_s \equiv p \pmod{q}$ mora se dodatno proveriti da bi utvrdilo da li je zaista korektan ili je samo **slučajno poklapanje**. Provera se obavlja tako što se eksplicitno proverava da li je $P[1..m] = T[s+1..s+m]$.

Za dovoljno veliko q, možemo se nadati da je pojavljivanje slučajnih poklapanja dovoljno retko, što bi značajno smanjilo vreme koje bi utrošili na dodatne provere.

Sledeća procedura (pseudokod) prikazuje direktnu primenu prethodnih ideja. Ulazni podaci procedure su tekst T, uzorak P, veličina alfabeta d (podrazumeva se da je $\Sigma = \{0, 1, \ldots, d-1\}$) i prost broj q koji koristimo kao moduo. Izostavljena je funkcija koja proizvoljni alfabet Σ prevodi u $\{0, 1, \ldots, d-1\}$.

Rabin-Karp-Matcher(T, P, d, q)

```
 1 n ← length[T]
 2 m ← length[P]
 3 h ← d^(m−1) mod q
 4 p ← 0
 5 t_0 ← 0
 6 for i ← 1 to m                    // preprocesiranje
 7    do p ← (dp + P[i]) mod q
 8       t_0 ← (dt_0 + T[i]) mod q
 9 for s ← 0 to n − m                // upoređivanje
10    do if p = t_s
11          then if P[1..m] = T[s + 1..s + m]
12                  then print "pomeraj s je korektan"
13       if s < n − m
14          then t_{s+1} ← (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Sledi opis rada procedure RABIN-KARP-MATCHER. Svi simboli su interpretirani kao cifre u sistemu sa osnovom d. Indeksi za niz t su stavljeni samo zbog jasnoće; očigledno, procedura radi korektno i bez njih. Linija 3 pseudokoda inicijalizuje promenljivu h vrednošću najstarije cifre u m-cifrenom broju (u bazi d). Linije 4 - 8 računaju p i t_0, što predstavlja vrednosti $P[1..m]$ mod q i $T[1..m]$ mod q, respektivno.

For petlja iz linija 9 - 14 prolazi kroz sve moguće pomeraje s, pri čemu je bitno što je u svakom trenutku $t_s = T[s{+}1..s{+}m] \bmod q$. Ako je $p = t_s$ u liniji 10 (poklapanje) onda u liniji 11 proveravamo da li je $P[1..m] = T[s{+}1..s{+}m]$ da bi smo izbacili mogućnost slučajnog poklapanja. Svi korektni pomeraji se štampaju u liniji 12. Ako je $s < n - m$ znači da će se **for** petlja izvršiti bar još jednom i zbog toga je potrebno računati vrednost za t_{s+1}. U liniji 14 se obavlja izračunavanje $t_{s+1} \bmod q$ pomoću $t_s \bmod q$ u konstantnom vremenu, direktno koristeći jednačinu (3).

RABIN-KARP-MATCHER vrši preprocesiranje u složenosti $\Theta(m)$ (što se lako vidi iz linija 1 - 8 pseudokoda), dok je složenost upoređivanja $O((n-m+1)m)$ (linije 9 -14). Složenost upoređivanja je u najgorem slučaju $\Theta((n-m+1)m)$ jer (poput naivnog algoritma iz poglavlja 4) Rabin-Karp algoritam eksplicitno potvrđuje svaki korektan pomeraj. Ako je $P = a^m$ i $T = a^n$, potvrđivanje troši tačno $\Theta((n-m+1)m)$ vremena jer je svaki od $n-m+1$ pomeraja korektan.

U mnogim aplikacijama očekujemo svega nekoliko korektnih pomeraja. Takođe, cesto je opravdano očekivanje da je broj slučajnih poklapanja reda $O(\frac{n}{q})$, jer je verovatnoća da će proizvoljni t_s biti jednak sa p po modulu q jednaka $\frac{1}{q}$. Označimo broj korektnih pomeraja sa k. Pošto se glavna petlja za upoređivanje (linija 9) izvršava $O(n)$ puta i kako za svako pronađeno poklapanje trošimo $O(m)$ vremena, složenost upoređivanja u Rabin-Karp algoritmu postaje

$$O(n) + O(m(k + \frac{n}{q})). \tag{4}$$

Prema tome, ako je očekivani broj korektnih pomeraja mali ($O(1)$) i ako je izabrani prost broj q veći od dužine uzorka ($q \geq m$) onda je složenost upoređivanja Rabin-Karp algoritma svega $O(n+m) = O(n)$, jer je $m \leq n$. Primetimo da pretpostavke koje su dovele do značajne redukcije složenosti nisu nerealne.

6 KMP algoritam

6.1 Korišćenje informacija

U ovom poglavlju je predstavljen linearni algoritam za pronalaženje uzorka u tesktu (pronalaženja podstringa u stringu). Algoritam su otkrili Knut, Moris i Prat i zbog toga se zove Knuth-Morris-Pratt algoritam ili jednostavno KMP.

Ovaj algoritam ima zajedničkih tačaka sa algoritmom na principu konačnog automata, ali mnogo pametnije koristi informacije iz uzorka što dovodi do značajne redukcije složenosti prilikom preprocesiranja u odnosu na pomenuti algoritam. Njegova složenost upoređivanja je $\Theta(n)$ koristeći samo pomoćnu funkiciju $\pi[1..m]$ izračunatu iz uzorka u složenosti $\Theta(m)$.

Kao što je pomenuto u poglavlju o naivnom algoritmu, ključna stvar prilikom upoređivanja dva stringa je maksimalno korišćenje informacija koje možemo dobiti iz uzorka P. Prefiksna funkcija π uzorka P koristi informacije kako se uzorak poklapa sa sobom, tj. da li se neki prefiksi stringa P pojavljuju sa nekim pomerajima u stringu P i koliki su ti pomeraji. Ova informacija se koristi kako bi se izbegla nepotrebna testiranja poput onih u naivnom algoritmu.

Razmotrimo upoređivanje uzorka $P = \mathbf{ababaca}$ sa tekstom T prikazano na slici 3. U ovom primeru prikazana je provera jednog pomeraja s i za taj pomeraj $q = 5$ simbola uzorka T su se poklopila sa tekstom, dok je kod 6. simbola došlo do neslaganja. Informacija da je došlo do poklapanja q simbola automatski određuje odgovarajuće simbole teksta T. Znajući tih q simbola teksta T, odmah možemo utvrditi da su neki pomeraji nekorektni (bez ponovnog upoređivanja). U primeru sa slike, pomeraj $s + 1$ je sigurno nekorektan jer bi se inače prvi simbol **a** uzorka P "poravnao" sa simbolom teksta za koji znamo da se poklapa sa drugim simbolom **b** uzorka P što je nemoguće ($\mathbf{a} \neq \mathbf{b}$). Sa druge strane, pomeraj $s' = s+2$ "poravnava" prva tri simbola uzorka sa tri simbola teksta za koje znamo da se sigurno poklapaju (tj. ta tri simbola ne moramo da proveravamo) na osnovu informacije o tom delu teksta.

Slika 3: (a) Uzorak je poravnat sa tekstom tako da se $q = 5$ simbola poklapaju. (b) Koristeći informacije koje posedujemo zaključujemo da je $s + 1$ nekorektan pomeraj, dok je pomeraj $s' = s + 2$ konzistentan sa svim što znamo o tekstu. (c) Informacije za takve zaključke smo dobili upoređujući uzorak sa samim sobom.

Uopšte, vrlo je korisno znati odgovor na sledeće pitanje: Ako se simboli $P[1..q]$ uzorka poklapa sa simbolima $T[s + 1..s + q]$ teksta za neki pomeraj s, koji je najmanji pomeraj $s' > s$ za koji važi

$$P[1..k] = T[s' + 1..s' + k] \tag{5}$$

gde je $s' + k = s + q$?

Takav pomeraj s' je prvi pomeraj veći od s za koji ne možemo sa sigrunošću da tvrdima da je nekorektan na osnovu poznavanja dela teksta $T[s+1..s+q]$, Tada su pomeraji $s+1, \ldots, s'-1$ sigurno nekorektni, jer bi u protivnom neki od njih zadovoljavao uslov (5) što je nemoguće jer je po definiciji s' najmanji pomeraj veći od s koji ga zadovoljava. U najboljem slučaju je $s' = s + q$ (tj. $k = 0$, svi simboli teksta $T[s+1..s+q]$ su različiti od $P[1]$) što znači da su svi pomeraji $s+1, \ldots, s+q-1$ nekorektni (znamo bez ikakve provere). U svakom slučaju, za novi pomeraj s' ne moramo da upoređujemo prvih k simbola uzorka P sa odgovarajućim simbolima teksta T, jer se oni sigurno poklapaju na osnovu jednakosti (5).

6.2 Prefiksna funkcija

Neophodne informacije za izračunavanje pomeraja s' možemo dobiti prilikom upoređivanja uzorka sa samim sobom, kao na Slici 3 (c). Šta je nama potrebno da znamo? Pošto je $T[s'+1..s'+k]$ deo teksta $T[s+1..s+q]$ (tačnije njegov sufiks, jer je $s'+k = s+q$) koji se poklapao sa $P[1..q]$, sledi da je $T[s'+1..s'+k]$ sufiks stringa P_q. Na osnovu ovoga i uslova (5), da bi našli traženi pomeraj s' potrebno (i dovoljno) je naći najveće $k < q$ tako da je $P_k \sqsupset P_q$. Tada je $s' = s + (q-k)$.

Za nalaženje takvog broja k koristimo funkciju π:

Definicija 6.1 *Za dati uzorak* $P[1..m]$, **prefiksna funkcija** *uzorka* P *je funkcija* $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ *za koju važi*

$$\pi[q] = max\ \{k \mid k < q \wedge P_k \sqsupset P_q\}.$$

Drugim rečima, $\pi[q]$ je dužina najdužeg prefiksa uzorka P (različitog od P_q) koji je ujedno i sufiks stringa P_q. U Tabeli 2 date su vrednosti prefiksne funkcije π za uzorak $P =$ **abababababca**.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

Tabela 2: Prefiksna funkcija π za $P =$ abababababca.

Dokazaćemo dve teoreme koje pomažu da se niz π efektivno izračuna. Uvedimo prvo neke oznake. Neka je

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \ldots, \pi^{(t)}[q]\},$$

gde se $\pi^{(i)}[q]$ definiše rekurzivno: $\pi^{(0)}[q] = q$ i $\pi^{(i+1)}[q] = \pi[\pi^{(i)}[q]]$, za svako $i \geq 1$. Podrazumeva se da se niz $\pi^*[q]$ završava kada je $\pi^{(t)}[q] = 0$.

Teorema 6.2 *Neka je P uzorak dužine m i neka je π njegova prefiksna funkcija. Tada za svako $q = 1, 2, \ldots, m$ važi: $\pi^*[q] = \{k \mid k < q \wedge P_k \sqsupset P_q\}$.*

Dokaz: Primetimo da iz $i \in \pi^*[q]$ sledi $P_i \sqsupset P_q$. Zaista, ako $i \in \pi^*[q]$ sledi da postoji $n > 0$ tako da je $i = \pi^{(n)}[q]$. Koristeći tranzitivnost relacije $\sqsupset$ (biti sufiks), indukcijom po n lako dokazujemo tvrđenje.

Pretpostavimo sada da postoji ceo broj k tako da je $P_k \sqsupset P_q$ i da $k \notin \pi^*[q]$. Neka je j najveći od svih takvih brojeva i neka je j' najmanji broj iz $\pi^*[q]$ koji je veći od j (takav broj postoji jer $\pi[q] \in \pi^*[q]$ i $\pi[q] = max\ \{k \mid k < q \wedge P_k \sqsupset P_q\}$ pa je sigurno $\pi[q] > j$). Na osnovu dokazanog prvog dela teorema, važi $P_{j'} \sqsupset P_q$. Kako je i $P_j \sqsupset P_q$, na osnovu Leme 3.2 sledi $P_j \sqsupset P_{j'}$, jer je $j' > j$. Prema pretpostavci, j je najveća vrednost manja od j' sa ovim svojstvom, pa sledi da je $\pi[j'] = j$. Ali kako $j' \in \pi^*[q]$ i $\pi[j'] = j$ sledi da i $j \in \pi^*[q]$, što je kontradikcija.

Ovim je dokaz kompletan. □

Primetimo da Teorema 6.2 govori da se svi sufiksi stringa P_q, dužine manje od q koji su prefiksi stringa P (i samo oni) nalaze u skupu $\pi^*[q]$.

Teorema 6.3 *Neka je P uzorak dužine m i neka je π njegova prefiksna funkcija. Tada, za svako $q = 1, 2, \ldots, m$ važi: ako je $\pi[q] > 0$, tada $\pi[q] - 1 \in \pi^*[q-1]$.*

Dokaz: Ako je $r = \pi[q] > 0$ tada je $r < q$ i $P_r \sqsupset P_q$. Sledi da je $r - 1 < q - 1$ i $P_{r-1} \sqsupset P_{q-1}$ (odbacivanjem poslednjih simbola stringova P_r i P_q). Prema tome, iz Teoreme 6.2, sledi da je $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. □

Ovo znatno pomaže prilikom izračunavanja funkcije (niza) π. Pretpostavimo da smo izračunali $\pi[i]$ za sve $i < q$. Na osnovu Teoreme 6.3 važi $\pi[q] - 1 \in \pi^*[q-1]$, odakle sledi da je $\pi[q]$ maksimum skupa $\{k+1 \mid k \in \pi^*[q-1] \wedge P_{k+1} \sqsupset P_q\}$ koji je ekvivalentan sa skupom $\{k+1 \mid k \in \pi^*[q-1] \wedge P_k \sqsupset P_{q-1} \wedge P[k+1] = P[q]\}$ koji je ekvivalentan sa skupom $\{k+1 \mid k \in \pi^*[q-1] \wedge P[k+1] = P[q]\}$ jer je prema Teoremi 6.2 $\{k \mid k < q-1 \wedge P_k \sqsupset P_{q-1}\} = \pi^*[q-1]$. Prema tome, da bi izračunali $\pi[q]$ dovoljno je posmatrati samo brojeve iz skupa $\pi^*[q-1]$ i proveriti jednostavan uslov $(P[k+1] = P[q])$. Pokazaćemo da je složenost ovoga linearna po m.

6.3 Algoritam

Sledeći pseudokod prikazuje način upoređivanja KMP algoritma korišćenjem prefiksne funkcije, kao i efektivan način izračunavanja iste.

KMP-Matcher(T, P)

```
 1 n ← length[T]
 2 m ← length[P]
 3 π ← Compute-Prefix-Function(P)
 4 q ← 0
 5 for i ← 1 to n
 6    do while q > 0 and P[q + 1] ≠ T[i]
 7          do q ← π[q]
 8       if P[q + 1] = T[i]
 9          then q ← q + 1
10       if q = m
11          then print "Uzorak se pojavljuje sa pomerajem" i − m
12               q ← π[q]
```

Compute-Prefix-Function(P)

```
 1 m ← length[P]
 2 π[1] ← 0
 3 k ← 0
 4 for q ← 2 to m
 5    do while k > 0 and P[k + 1] ≠ P[q]
 6          do k ← π[k]
 7       if P[k + 1] = P[q]
 8          then k ← k + 1
 9       π[q] ← k
10 return π
```

KMP-MATCHER radi tako što procesira tekst s leva na desno pri čemu u svakom trenutku u promenljivoj q pamti broj poklopljenih simbola. Preciznije, kada se glavna petlja iz linije 5 izvrši $i-1$ puta, znamo da je $P_q = T[i-q..i-1]$ i da ne postoji veći broj q sa ovim svojstvom. Ukoliko je $P[q+1] \neq T[i]$ linije 6 - 7 traže najmanji pomeraj s' za koji znamo da nije nekorektan, uz pomoć izračunate prefiksne funkcije π. Trenutni pomeraj je $s = i - q$ a sledeći pomeraj s' će biti $i - q + (q - \pi[q]) = i - \pi[q]$ pa zatim $i - \pi[\pi[q]]$ itd. sve dok ne dođe do poklapanja $P[q+1]$ sa $T[i]$ ili dok q ne postane 0. Ako postane $P[q+1] = T[i]$, povećavamo dužinu prefiksa uzorka P koji se poklapa sa nekim delom teksta (linije 8 - 9). Ukoliko dođe do poklapanja, štampa se odgovarajući pomeraj i postavlja se vrednost promenljive q na $\pi[q]$ jer je to sledeći validan pomeraj i nema smisla proveravati jednakost za $P[m+1]$ jer taj simbol ne postoji.

COMPUTE-PREFIX-FUNCTION izračunava π direktno koristeći Teoreme 6.2 i 6.3. Očigledno je $\pi[1] = 0$ jer je, pre svega, $\pi[q] < q$. Prilikom svakog ulaska u **for** petlju iz linije 4, u promenljivoj k se nalazi vrednost $\pi[q-1]$. Kako je pokazano da je $\pi[q] - 1 \in \pi^*[q-1]$, dovoljno je ispitivati samo te vrednosti. Linije 5 - 6 omogućavaju prolaz kroz skup $\pi^*[q-1]$ i to od njegovih najvećih elemenata ka najmanjim ($k > \pi[k] > \pi[\pi[k]] > \ldots$) pa će u trenutku poklapanja (linija 7) $\pi[q]$ dobiti najveću odgovarajuću vrednost iz skupa $\pi^*[q-1]$.

Iako se iz koda ne može lako primetiti, i složenost preprocesiranja i složenost upoređivanja je linearna.

Posmatrajmo kod za preprocesiranje. **While** petlja iz linija 5 - 6 može da se izvrši $O(k)$ puta, za fiksirano k i u tim linijama se k smanjuje jer je $k > \pi[k]$ ali je uvek $k \geq 0$. Sa druge strane, k se povećava jedino u liniji 8 i to samo za 1, pa prema tome $k \leq m$ za sve vreme izvršavanja funkcije jer imamo m prolaza kroz petlju. Ako se **while** petlja iz linija 5 - 6 izvršila X puta, to znači da će na kraju programa biti $k \leq m - X$ jer je na početku $k = 0$, u toku programa linija 8 ne može se izvršiti više od m puta i svaki put kada se **while** petlja izvrši k se smanji za bar 1. Kako je uvek $k \geq 0$, sledi $m - X \geq 0$ odakle zaključujemo da je $X \leq m$, odakle je $X = O(m)$. Osim **while** petlje, ostatak programa radi u složenosti $\Theta(m)$ pa je ukupna složenost preprocesiranja $O(m) + \Theta(m) = \Theta(m)$.

Zbog velike sličnosti procedura za upoređivanje i preprocesiranje, dokaz da KMP-MATCHER radi u složenosti $\Theta(n)$ je analogan, samo što se umesto k posmatra q.

Prema tome, složenost preprocesiranja KMP algoritma je $\Theta(m)$ dok je složenost upoređivanja $\Theta(n)$. Primetimo da za razliku od naivnog i Rabin-Karp algoritma, ovde ne postoji najgori slučaj, čak i za stringove oblika a^n KMP algoritam radi stabilno. Takođe primetimo da azbuka Σ uopšte ne utiče na rad algoritma, kao ni činjenica da li je to konačna ili beskonačna azbuka, što KMP algoritam čini univerzalnim.

7 Zaključak

Kao što je rečeno, efikasnos algoritama igra veliku ulogu u procesiranju teksta. Korišćenje drugačijih algoritama mož ubrzati pretragu više puta, što je vrlo značajno za velike dokumente kada se trajanje pretrage meri desetinama sekundi ili minutima.

U radu su opisana 3 algoritma sa različitim pristupom traženja uzorka u tekstu. Koji od njih primeniti - zavisi od same prirode problema. Generalno, KMP je najbolje rešenje za ovu klasu problema, ali nije trivijalan za razumevanje. Videli smo, međutim, da za slucajno izabrane uzorke, i složenost Rabin-Karp i naivnog algoritma postaje linearna. Takođe, za mali alfabet (npr. binarni nizovi) veća je verovatnoća da se pojavi najgori sluąj kod ova dva algoritma. Još jedan faktor koji treba uzeti u razmatranje je dužina uzorka. Često je potrebno naći samo jednu reč u tekstu, pa u tim sluăjevima možemo pretpostaviti da je dužina uzorak konstanta i tada će i naivni algoritam raditi u linearnom vremenu.

Literatura

[1] Kisačanin B, *Mala matematika*, Novi Sad, Univerzitet u Novom Sadu & Stylos, 1995

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, Second Edition, The MIT Press, 2001.

[3] Sedgewick R, *Algorithms*, Addison-Wesley, 1984.

[4] Živković M, *Algoritmi*, Matematički fakultet, Beograd, 2000