

UNIVERZITET U NIŠU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA RAČUNARSKE NAUKE



Algoritmi za pretragu stringova

MASTER RAD

Student:

Ivan Stošić

Mentor:

Marko Petković

Niš
Septembar, 2019.

Sadržaj

1	Uvod	2
1.1	Uvodne definicije	2
1.2	Složenost algoritama	3
1.3	Implementacije algoritama	4
2	Osnovni algoritmi pretrage	5
2.1	Naivna pretraga	5
2.2	Knuth-Morris-Pratt algoritam	6
2.3	Z-algoritam	9
3	Dinamičko programiranje nad stringovima	11
3.1	LCP matrica	11
3.2	Najduži zajednički podniz	11
3.2.1	Hirschberg-ov algoritam	11
3.2.2	Najduži palindromski podniz	11
3.3	Levenshtein udaljenost	11
4	Prefiksne strukture podataka	12
4.1	Trie	12
4.2	Aho-Corasick algoritam	12
5	Sufiksne strukture podataka	13
5.1	Sufiks niz	13
5.2	Sufiks stablo	13
5.3	Sufiks automat	13
6	Heširanje	14
6.1	Rabin-Karp algoritam	14
6.2	Primene	14
6.3	Konstrukcija anti-heš primera	14
7	Palindromi	15
7.1	Manacher-ov algoritam	15
7.2	Palindromsko stablo	15

1 Uvod

1.1 Uvodne definicije

Neformalno, string je niz simbola iz nekog alfabeta. U opštem slučaju, alfabet može biti bilo koji konačan skup. Međutim, za potrebe pojedinih algoritama, potrebno je da alfabet bude totalno uređen skup. Kod implementacije algoritama, dodatno možemo pretpostaviti da je alfabet jednak nekom konačnom skupu uzastopnih celih brojeva, najčešće $\{0, 1, \dots, k-1\}$ ili $1, 2, \dots, k$ za neko $k \in \mathbb{N}$. Tradicionalno, ovaj alfabet se označava grčkim slovom Σ .

Definicija 1.1 *Ako je Σ alfabet a n prirodan broj, Σ^n označava skup svih uređenih n -torki $(s_0, s_1, \dots, s_{n-1})$, gde je $s_i \in \Sigma$ za svako $i \in \{0, \dots, n-1\}$.*

Ovakvu n -torku možemo kraće zapisati sa s , a njenu dužinu (broj elemenata) sa $|s|$. Koristi se i kraći zapis n -torke: $s_0 s_1 \dots s_{n-1}$.

Definicija 1.2 *Ako je Σ alfabet, onda je*

$$\Sigma^+ = \bigcup_{n=1}^{\infty} \Sigma^n$$

skup svih nepraznih reči nad alfabetom Σ .

Ovom skupu možemo dodati i praznu reč, koju označavamo sa e ili ϵ .

Definicija 1.3 *Skup svih reči nad alfabetom Σ je skup $\Sigma^* = \Sigma^+ \cup \{e\}$.*

Za string možemo definisati i njegove podstringove na sledeći način.

Definicija 1.4 *Podstring počev od pozicije l , do pozicije r ne uključujući r , nekog stringa s , gde važi $0 \leq l \leq r \leq |s|$ je string $s_l s_{l+1} \dots s_{r-1}$. Ovaj podstring kraće zapisujemo i $s_{[l,r)}$.*

Podstringove stringa s kod kojih je $l = 0$ nazivamo prefiksima tog stringa, dok podstringove kod kojih je $r = |s|$ nazivamo sufiksima tog stringa. Ukoliko je $l = r$ radi se o praznom podstringu. Ukoliko je podstring različit od celog stringa, onda se radi o pravom podstringu, sufiksu odnosno prefiksu. Izbor notacije sa indeksiranjem od nule i poluotvorenim intervalom olakšava implementaciju većine algoritama sa stringovima.

Definicija 1.5 *Ciklični podstring stringa s dužine n počev od pozicije l do pozicije r je string $s_{[l,r]} = s_l \bmod n s_{(l+1) \bmod n} \cdots s_{(r-1) \bmod n}$.*

Ciklični podstring uopštava pojam podstringa. Zaista, ako je $0 \leq l \leq r \leq n$, ciklični podstring jednak je običnom podstringu.

Definicija 1.6 *Ciklični pomerač stringa s dužine n počev od pozicije i je string $s_{[i,i+n)}$.*

Stringovi se mogu i nadovezivati odnosno konkatenerirati. Skup Σ^+ , odnosno Σ^* zajedno sa operacijom konkatencije čini algebarsku strukturu polugrupe, odnosno monoida.

Definicija 1.7 *Ako su s, p stringovi, tada je njihova konkatencija string $sp = s_0 s_1 \cdots s_{|s|-1} p_0 p_1 \cdots p_{|p|-1}$.*

Ukoliko je skup Σ totalno uređen, definišemo leksikografsko poređenje stringova kao uređenje skupa Σ^* , na sledeći način.

Definicija 1.8 *Za string s kažemo da je leksikografski manji od stringa p ukoliko postoji ceo broj $k \geq 0$ takav da je $k < \min\{|s|, |p|\}$, $s_{[0,k)} = p_{[0,k)}$ i $s_k < p_k$ ili ako je s pravi prefiks stringa p .*

Teorema 1.1 *Leksikografsko poređenje je totalno uređenje skupa Σ^* . \square*

Definicija 1.9 *Za svako $x \in \Sigma$, $\text{Ord}(x)$ je broj elemenata skupa Σ koji su strogo manji od x .*

1.2 Složenost algoritama

Vreme, odnosno broj koraka i količina utrošene memorije tokom izvršenja nekog algoritma zavisi od ulaznih parametara. *Veliko O* notacija nam olakšava opisivanje i izračunavanje ovih funkcionalnih zavisnosti. Neka je u narednim definicijama domen funkcija f, g skup \mathbb{N}_0 a kodomen $\mathbb{R}^+ \cup \{0\}$.

Definicija 1.10 *Skup $O(g)$ definišemo kao skup svih funkcija f za koje važi da postoje konstante c i n_0 takve da je $f(n) \leq cg(n)$ za svako $n \geq n_0$.*

Ovu notaciju koristimo kada želimo da opišemo gornju granicu neke funkcije, do na proizvod sa konstantom. Problem ove notacije je upravo u tome što samo daje gornju granicu ponašanja neke funkcije. Zato se uvodi

Θ -notacija.

Definicija 1.11 *Skup $\Theta(g)$ definišemo kao skup svih funkcija f za koje važi da postoje pozitivne konstante c_1, c_2 i n_0 takve da je $c_1g(n) \leq f(n) \leq c_2g(n)$ za svako $n \geq n_0$.*

Za algoritam čiji je ulazni parametar n , što može biti broj elemenata niza, broj vrsta matrice, broj čvorova grafa, itd. kažemo da ima vremensku složenost $\Theta(g(n))$ odnosno $O(g(n))$ ako je f , gde je $f(n)$ broj elementarnih koraka tokom izvršenja algoritma, u skupu $\Theta(g)$ odnosno $O(g)$. Slično definišemo memorijsku složenost preko broja iskorišćenih elementarnih memorijskih lokacija.

1.3 Implementacije algoritama

Za sve implementacije biće korišćen programski jezik C++, kompajler GCC, verzija 9.1.0. Radi jednostavnosti, kodovi će biti dati bez `main` funkcije, `#include` direktiva i naredbe `using namespace std;`. Pretpostavlja se da su uključene sve standardne biblioteke, što se kod GCC-a može postići sa `#include <bits/stdc++.h>`.

2 Osnovni algoritmi pretrage

Jedan od osnovnih problema pretrage stringova je problem nalaženja svih pojavljivanja jednog stringa u drugom. Formalno, za data dva stringa s, p dužina n i m , redom, naći sve indekse i takve da je $0 \leq i \leq n-m$ i $s_{[i, i+m)} = p$. String s unutar kojeg se traži se često naziva tekstom, dok se p naziva rečju (iako ne mora biti jedna reč) ili *pattern*-om odnosno šablonom. U literaturi na engleskom jeziku se često sreću i pojmovi *haystack* (plast sena) i *needle* (igla).

2.1 Naivna pretraga

Naivna pretraga rešava prethodno opisani problem tako što za svako celo i iz segmenta $[0, n - m]$ upoređuje karakter po karakter stringove $s_{[i, i+m)}$ i p , pri čemu se odmah zaustavlja ukoliko naiđe na dva različita karaktera. Ovo poređenje ima vremensku složenost $O(m)$ pa ceo algoritam ima vremensku složenost $O((n - m)m)$, ova složenost se dostiže npr. za stringove koji se sastoje samo od slova a.

Implementacija naivne pretrage

```
1 vector<int> naive_search(const string& s, const string& p) {
2     int n = s.size(), m = p.size();
3     vector<int> result;
4     for (int i=0; i<=n-m; i++) {
5         bool ok = true;
6         for (int j=0; j<m; j++) {
7             if (s[i+j] != p[j]) {
8                 ok = false;
9                 break;
10            }
11        }
12        if (ok)
13            result.push_back(i);
14    }
15    return result;
16 }
```

2.2 Knuth-Morris-Pratt algoritam

KMP je prvi otkriveni algoritam koji rešava problem pretrage stringa u linearnom vremenu, odnosno u složenosti $O(n + m)$. Da bismo razumeli rad algoritma, definišimo sledeće pojmove.

Definicija 2.1 *Sufiks-prefiks nepraznog stringa s je svaki string p različit od s , uključujući i prazan string, koji je istovremeno i sufiks i prefiks stringa s , tj. $s_{[0,|p|)} = s_{[|s|-|p|,|s|)} = p$.*

Označimo sa $g(s)$ dužinu najdužeg sufiks-prefiksa stringa s . Za prazan string s definišemo $g(s) = -1$. Ova funkcija zadovoljava sledeću, važnu osobinu:

Teorema 2.1 *Neka je s neprazan string. Neka je $s' = s_{[0,|s|-1)}$, tada je $g(s) \leq g(s') + 1$.*

Dokaz. Pretpostavimo suprotno, da je $g(s) > g(s') + 1$. Neka je p najduži sufiks prefiks stringa s . Neka je $p' = p_{[0,|p|-1)}$. Tada je p' sufiks-prefiks stringa s' , pa je $g(s') \geq |p'| = |p| - 1 = g(s) - 1$, kontradikcija.

Ukoliko je p najduži sufiks-prefiks stringa s , tada je i svaki drugi sufiks-prefiks stringa s ujedno i sufiks-prefiks stringa p . Ova osobina nam omogućava da opišemo sve sufiks-prefikse nekog stringa s kao lanac, gde je svaki naredni string najduži sufiks-prefiks prethodnog, sve dok se ne dođe do praznog stringa.

Definicija 2.2 *Niz neuspeha stringa s je niz $f_0, f_1, \dots, f_{|s|}$, gde je $f_i = g(s_{[0,i)})$.*

Prvi korak KMP algoritma je nalaženje niza neuspeha za string p . Prvo upisujemo $f_0 = -1$. Zatim, za svako $i > 0$, tražimo najduži sufiks-prefiks stringa $p_{[0,i-1)}$ koji se može proširiti slovom p_{i-1} , odnosno, nalazimo najveći broj r takav da je r dužina nekog sufiks-prefiksa stringa $p_{[0,i-1)}$ i važi $p_r = p_{i-1}$. Ukoliko ne nađemo takav broj r , onda je $f_i = 0$. U suprotnom, pošto je $p_{[0,r)} = p_{[i-1-r,i-1)}$ i $p_r = p_{i-1}$ imamo da je $p_{[0,r+1)} = p_{[i-r-1,i)}$, odnosno, $r + 1$ je dužina jednog sufiks-prefiksa stringa $p_{[0,i)}$. Iz prethodnog razmatranja imamo da je ovo upravo najduži sufiks-prefiks stringa $p_{[0,i)}$, pa upisujemo $f_i = r + 1$. Niz svih sufiks-prefiksa možemo naći primenom opisane osobine lanca i činjenice da smo u i -tom koraku već izračunali dužine najdužih sufiks-prefiksa za sve prefikse stringa p dužine manje od i .

Primer. Posmatrajmo string `atamatata`. Njegov niz neuspeha dat je u sledećoj tabeli. Indeksi u donjem redu su pomereni za jedno mesto da bi se bolje videlo o kom prefiksu je reč.

	a	t	a	m	a	t	a	t	a
-1	0	0	1	0	1	2	3	2	3

Posmatrajmo šta se dešava pri računanju f_i za pretposlednji prefiks, dužine 8. U prethodnom koraku smo imali string `atamata`, kome je najduži sufiks-prefiks string `ata`. Ako pokušamo da dodamo na njega slovo `t`, nećemo dobiti poklapanje jer je na poziciji 3 slovo `m`, zato pokušavamo sa narednim sufiks-prefiksom odnosno $r = f_3 = 1$. Sada uspešno dodajemo slovo `t` i upisujemo $r + 1 = 2$ u f_8 .

Implementacija nalaženja niza neuspeha kod KMP algoritma

```

1 vector<int> kmp_ff(const string& p) {
2     int m = p.size();
3     vector<int> f(m+1);
4     f[0] = -1;
5     for (int i=1; i<=m; i++) {
6         int r = f[i-1];
7         while (r != -1 && p[r] != p[i-1])
8             r = f[r];
9         f[i] = r+1;
10    }
11    return f;
12 }
```

Ocenimo složenost ovog algoritma. Jasno je da složenost srazmerna zbiru broju iteracija ove dve petlje. Posmatrajmo vrednost $2i - r$. Nakon svake iteracije *while* petlje imamo da se r smanjuje, jer je $f_r < r$ pa se $2i - r$ povećava. U jednoj iteraciji *for* petlje imamo da se i i r povećavaju za tačno 1, pa se $2i - r$ ponovo povećava. Kako je početna vrednost $2i - 3$ jednaka 3, a važi $2i - r \leq 2m + 1$, ukupan broj iteracija, a samim tim i složenost algoritma je $O(m)$.

Opišimo sada glavni algoritam za traženje stringa. Algoritam za svaki prefiks i stringa s nalazi najduži sufiks koji je ujedno i prefiks stringa p . Ukoliko je dužina tog prefiksa jednaka $|p|$, tada dolazi do poklapanja na poziciji $i - |p|$. Neka je ta dužina jednaka r_i . Za $i = 0$ imamo $r_0 = 0$. Za svako $i > 0$, krećemo od prefiksa stringa p dužine r_{i-1} i proveravamo da li je naredno slovo jednako slovu s_{i-1} . Ako jeste, zaustavljamo se i upisujemo

dužinu nađenog prefiksa u r_i , u suprotnom, sledeći prefiks koji pokušavamo da produžimo je prefiks dužine $f_{r_{i-1}}$. Ovaj postupak ponavljamo sve dok ne dođemo do poklapanja ili do fiktivnog prefiksa -1 , u tom slučaju dužina poklapanja je 0 . Primetimo da je ovaj deo algoritma veoma sličan nalaženju niza neuspeha.

Implementacija glavnog dela KMP algoritma

```

1  vector<int> kmp_main(const string& s, const string& p) {
2      vector<int> f = kmp_ff(p), result;
3      int n = s.size(), m = p.size(), r = 0;
4      for (int i=1; i<=n; i++) {
5          while (r != -1 && p[r] != s[i-1])
6              r = f[r];
7          r++;
8          if (r == m) {
9              result.push_back(i-m);
10             r = f[r];
11         }
12     }
13     return result;
14 }
```

Na potpuno isti način se ocenjuje složenost glavnog dela algoritma, posmatranjem vrednosti izraza $2i - r$. Složenost je $O(n + m)$, zajedno sa prvom fazom, ukupna složenost je takođe $O(n + m)$.

Cela implementacija algoritma se može značajno pojednostaviti na sledeći način: Posmatrajmo string $p\$s$, gde je $\$$ karakter koji se ne javlja u stringovima p, s . Na osnovu njegovog niza neuspeha možemo da zaključimo gde se sve pojavljuje p u s , tačnije, i je pojavljivanje p u s akko je $f_{i+2m+1} = m$.

Pojednostavljena implementacija celog algoritma

```
1 vector<int> kmp_simple(const string& s, const string& p) {
2     string q = p + '\\0' + s;
3     int n = s.size(), m = p.size();
4     vector<int> f(n+m+2), result;
5     f[0] = -1;
6     for (int i=1; i<=n+m+1; i++) {
7         int r = f[i-1];
8         while (r != -1 && q[r] != q[i-1])
9             r = f[r];
10        f[i] = ++r;
11        if (r == m && i >= 2*m+1)
12            result.push_back(i-2*m-1);
13    }
14    return result;
15 }
```

2.3 Z-algoritam

Osnovna ideja ovog algoritma je da se izračuna Z-niz, koji se definiše na sledeći način.

Definicija 2.3 Za string s dužine n , Z-niz je niz z_1, \dots, z_{n-1} gde je z_i dužina najdužeg zajedničkog prefiksa za stringove s i $s_{[i,n]}$.

Z-niz se može iskoristiti za pretragu stringova. Ukoliko nađemo Z-niz za string ps , i je pojavljivanje p u s akko je $z_{i+m} \geq m$.

Z-algoritam je efikasan algoritam za nalaženje Z-niza. Označimo sa q string za koji nalazimo Z-niz. Algoritam za svako i direktno izračuna z_i poklapanjem slova na pozicijama $z_i, i + z_i$. Ključna ideja je da se prethodno izračunate vrednosti Z-niza mogu iskoristiti da se postavi bolja početna vrednost za z_i , umesto da se svaki put kreće od nule. Naime, neka je r najveća nađena vrednost izraza $j + z_j$ za $j < i$, a neka je pritom $l = j$ za koje se dostiže taj maksimum. Drugim rečima, $[l, r)$ je prozor koji odgovara nekom do sada pronađenom podstringu koji je jednak nekom prefiksu stringa, i to onom kod kojeg je r najveće. Ukoliko važi $l \leq i < r$, tada znamo da je $q_{[i,r)} = q_{[i-l,r-l)}$. Takođe, važi $q_{[i-l,i-l+z_{i-l})} = q_{[0,z_{i-l})}$ pa, ako označimo sa $t_i = \min(z_{i-l}, r - i)$, važi $q_{[i,i+t_i)} = q_{[0,t_i)}$ odnosno $z_i \geq t_i$.

Implementacija Z-algoritma za pretragu stringa

```
1 vector<int> z_algorithm(const string& s, const string& p) {
2     int n = s.size(), m = p.size();
3     string q = p + s;
4     vector<int> z(n+m, 0), result;
5     for (int i=1, l=0, r=0; i<n+m; i++) {
6         if (i < r)
7             z[i] = min(z[i-l], r-i);
8         while (i+z[i] < n+m && q[i+z[i]] == q[z[i]])
9             z[i]++;
10        if (i+z[i] > r)
11            l = i, r = i+z[i];
12        if (z[i] >= m)
13            result.push_back(i-m);
14    }
15    return result;
16 }
```

Dokažimo da ovaj algoritam ima složenost $O(n+m)$. Očigledno, kritična je unutrašnja *while* petlja. Dokazaćemo da svaka iteracija *while* petlje odgovara povećanju vrednosti promenljive r za bar 1. Posmatrajmo sledeće slučajeve:

- Pre ulaska u *while* petlju važi $i \geq r$. U ovom slučaju z_i ima početnu vrednost nula, na kraju *while* petlje će važiti $i + z_i \geq r$ pa će r dobiti vrednost $i + z_i$, odnosno r će se povećati za barem z_i , što je veće ili jednako od broja iteracija *while* petlje.
- Pre ulaska u *while* petlju važi $i < r$ i $z_{i-l} < r - i$. Sada z_i dobija početnu vrednost z_{i-l} . Dokažimo da će *while* petlja izvršiti tačno nula iteracija. Pretpostavimo suprotno, da je $q_{i+z_i} = q_{z_i}$, odnosno $q_{i+z_{i-l}} = q_{z_{i-l}}$. Iz definicije prozora $[l, r)$ imamo da je $q_{[0, r-l)} = q_{[l, r)}$, pošto je $l \leq i + z_{i-l} < r$, odnosno, ova pozicija je unutar prozora, važi $q_{i+z_{i-l}} = q_{i-l+z_{i-l}}$, odnosno, po pretpostavci, $q_{z_{i-l}} = q_{i-l+z_{i-l}}$, što znači da z_{i-l} ima pogrešno izračunatu vrednost jer se poklapaju karakteri na kraju odgovarajućih podstringova, što dovodi do kontradikcije.
- Pre ulaska u *while* petlju važi $i < r$ i $z_{i-l} \geq r - i$. Sada z_i dobija početnu vrednost $r - i$. Ako petlja izvrši k iteracija, na kraju će važiti $z_i = k + r - i$ odnosno $i + z_i = k + r$, što znači da će nova vrednost r biti za bar k veća.

Kako je $r \leq n + m$ zaključujemo da je ukupna složenost $O(n + m)$.

3 Dinamičko programiranje nad stringovima

3.1 LCP matrica

3.2 Najduži zajednički podniz

3.2.1 Hirschberg-ov algoritam

3.2.2 Najduži palindromski podniz

3.3 Levenshtein udaljenost

4 Prefiksne strukture podataka

4.1 Trie

4.2 Aho-Corasick algoritam

5 Sufiksne strukture podataka

5.1 Sufiks niz

5.2 Sufiks stablo

5.3 Sufiks automat

6 Heširanje

6.1 Rabin-Karp algoritam

6.2 Primene

6.3 Konstrukcija anti-heš primera

7 Palindromi

7.1 Manacher-ov algoritam

7.2 Palindromsko stablo