Rabin-Karp algoritem

Pisni izdelek pri predmetu *Računalništvo 2*

Tomaž Tratnik

11. junij, 2022

1 Uvod

Naš algoritem mora poiskati točna ujemanja vzorca v danem besedilu. Za vhodne podatke prejme besedilo dolžine n in vzorec dolžine $m \leq n$. Za izhodni podatek mora vrniti tabelo indeksov v besedilu, kjer smo našli točna ujemanja vzorca.

Zgled 1.1. Recimo, da iščemo vzorec *čaka* v besedilu *Kdor čaka, dočaka.* Algoritem bi moral v tem primeru vrniti tabelo indeksov $[5, 13]$.

2 Naiven pristop

Najpreprostejši algoritem primerja vsak podniz besedila dolžine m z vzorcem, vsak znak posebej. Podniz premikamo od indeksa 0 do indeksa $n-m$. Na vsaki poziciji primerjamo prvi znak podniza s prvim znakom vzorca, drugi znak podniza z drugim znakom vzorca in tako naprej do zandjega znaka. Če se vsi znaki ujemajo, lahko zapišemo pozicijo podniza v tabelo indeksov, kjer smo našli ujemanja. Na vsaki poziciji torej opravimo m primerjav. Ena očitna izboljšava je, da nehamo primerjati podniz in vzorec takoj, ko se kak znak ne ujema. Ker moramo preveriti $n-m$ posameznih pozicij in ker moramo na vsaki poziciji primerjati m posameznih znakov, je časovna zahtevnost tega algoritma $O(n \cdot m)$.

Zgled 2.1. Vzemimo spet za vzorec *čaka* in besedilo *Kdor čaka, dočaka.* V prvem koraku vzamemo za podniz *Kdor*. Sedaj primerjamo znak na prvem mestu v podnizu, to je *K*, s prvim znakom v vzorcu, to je *č*. Ker se ne ujemata, bi izboljšan algortiem končal primerjavo. Sedaj premaknemo podniz naprej, torej bo v drugem koraku podniz *dor* . Ker se prva dva znaka spet ne ujemata, lahko nadaljujemo. V tretjem koraku bi bil podniz *or č*, ki se spret ne ujema. V petem koraku bi pa bil podniz *čaka*. Ker se vsaka črka podniza ujema z vsako črko v vzorcu, bi v tabelo dodali indeks 5. Enako se podniz in vzorec ujemata na indeksu 13. Postopek bi ponavljali do indeksa 14, kjer bi bil podniz *aka.* Ker smo pišli do konca besedila, je postopek končan. V tem primeru je dolžina beseila $n = 18$, dolžina vzorca pa $m = 4$. Neizboljšan algoritem bi torej naredil $(n - m) * m = 56$ posameznih primerjav znakov. Izboljšan algoritem bi pa naredil 20 primerjav.

3 Rabin-Karp algortiem

Tako kot naiven algoritem, Rabin-Karp algoritem premika podniz od začetka besedila do konca. Razlika je v tem, da ne primerjamo samih nizov po znakih, vendar primerjamo zgoščene vrednosti (hash value) vzorca in podniza. Rabin-Karp algoritem bo torej izračunal zgoščeno vrednost vzorca in vseh možnih podnizov dolžine

m. Če je zgoščena vrednost kakega podniza enaka zgoščeni vrednosti vzorca, bo pa algoritem preveril ujemanje s primerjavo znakov, enako kot naiven algoritem.

3.1 Zgoščevalna funkcija

Zgoščevalne funkcije so funkcije, ki pretvorijo podatke poljubne velikosti v vrednost fiksne velikosti. Natačneje je to preslikava, ki slika iz izbrane domene v množico števil $0, 1, ..., N-1$. Število N izberemo odvisno od željenih lastnosti funkcije.

Zgoščevalna funkcija, ki jo želimo uporabiti v Rabin-Karp algoritmu, mora imeti določene lastnosti:

- Zgoščene vrednosti morajo biti čim bolj enakomerno porazdeljene čez izhodni interval. Drugače povedano, za dva različna vhodna podatka mora biti verjetnost, da sta izhodni vrednosti enaki (pride do trčenja), čim manjša. S tem se izognemo nepotrebnemu primerjanu nizov po znakih.
- Mora biti deterministična, za enake vhodne podatke mora vedno vrniti enake zgoščene vrednosti.
- V našem primeru se podatki, ki jih moramo zgostiti, prekrivajo. Podniz dolžine m na mestu i, se bo razlikoval s podnizom na mestu $i+1$ samo v enem znaku. Iz zgoščene vrednosti podniza na mestu i moramo izračunati novo vrednost za podniz na mestu $i+1$ v konstantem času. Takim zgoščevalnim funkcijam rečemo vozeča zgoščevalna funkcija ali rolling hash v angleščini.

3.1.1 ASCII vrednosti

Najbolj preprosta zgoščevalna funkcija, ki jo lahko uporabimo, je kar vsota ASCII vrednosti vseh znakov v podnizu. Od sedaj naprej bomo s $s[i]$ označevali ASCII vrednost znaka na i-tem mestu v besedilu, s $H(i,m)$ pa zgoščeno vrednost podniza z začetkom na indeksu i in dolžino m. Hash vrednost bi torej izračunali z:

$$H(i,m) = s[i] + s[i+1] + ... + s[i+m-2] + s[i+m-1] = \sum_{j=i}^{i+m-1} s[j]$$

Da dobimo podniz na mesti $i+1$ iz podniza na mestu i, moramo odstraniti prvi znak prejšnjega podniza in dodati en znak na konec. Torej zgoščeno vrednost podniza na mestu $i+1$ iz prejšnje vrednosti izračunamo z:

$$H(i+1,m) = s[i+1] + s[i+2] + ... + s[i+m-1] + s[i+m] = \sum_{j=i+1}^{i+m} s[j]$$

$$H(i+1,m) = H(i,m) - s[i] + s[i+m]$$

Na ta način lahko novo zgoščeno vrednost izračunamo iz prejšnje v konstantnem času. Ta zgoščevalna funkcija je preprota, vendar v praksi ni uporabna, saj ne upošteva vrstnega reda znakov. Vsaka permutacija nekega podniza bi dala enako zgoščeno vrednost. Na primer vrednost podniza *aab* bi bila enaka vrednosti *aba*.

3.1.2 Polinomska zgoščevalna funkcija

Da se izognemo slabosti prejšnje funkcije, lahko podniz predstavimo kot število v neki bazi B. Za bazo B vzamemo neko praštevilo, ki je večje od vseh ASCII vrednosti, na primer 257. Računanje s praštevili nam zmanjša verjetnost trčenj. Če množimo določeno število s praštevilom, je možnost, da dobimo isti produkt s kakim drugim zaporedjem množenj manjša, kot če ne bi množili s praštevilom. V tem primeru je verjetnost, da imata dva različna niza enako zgoščeno vrednost, manjša. Če bi pa za bazo vzeli število, ki je enako kateri od ASCII vrednosti, bi se nam verjetnost trčenj povečala.

Na tak način bi zgoščeno vrednost izračunali z:

$$\begin{aligned} H(i,m) &= s[i] \cdot B^{m-1} + s[i+1] \cdot B^{m-2} + \ldots + s[i+m-2] \cdot B + s[i+m-1] \\ &= \sum_{j=i}^{i+m-1} s[j] \cdot B^{m+i-j-1} \end{aligned}$$

Zgoščeno vrednost podniza na mestu $i+1$ izračunamo z:

$$\begin{aligned} H(i+1,m) &= s[i+1] \cdot B^{m-1} + s[i+2] \cdot B^{m-2} + \ldots + s[i+m-1] \cdot B + s[i+m] \\ &= \sum_{j=i+1}^{i+m} s[j] \cdot B^{m+i-j}, \end{aligned}$$

iz prejšnje vrednosti pa z:

$$H(i+1,m) = (H(i,m) - s[i] \cdot B^{m-1}) \cdot B + s[i+m]$$

V izrazu za izračun naslednje zgoščene vrednosti se pojavi člen B^{m-1}, ki za izračun poterbuje $m-1$ število operacij. Ker pa je ta člen vedno enak za vsak korak, ga lahko izračunamo pred izvajanjem algoritma. Tako lahko spet izračunamo novo zgoščeno vrednost iz prejšnje v konstantem času. Ta zgoščevalna funkcija se izogne problemu različnih permutacij podniza. Ima pa to slabost, da zgoščene vrednosti lahko postanejo zelo velika števila, še posebej pri daljših iskanih vzorcih. Da se izognemo temu problemu, bomo računali zgoščene vrednosti po modulu.

3.1.3 Polinomska zgoščevalna funkcija z modulom

Pri tej zgoščevalni funkciji računamo po modulu Q, da se izognemo velikim številom. Za Q najpogosteje vzamemo neko praštevilo, ki je reda 10^9, na primer

1000000007. S tem zagotovimo, da bomo vedno računali s 64-bitnimi števili. Produkt dveh števil reda 10^9 lahko predstavimo s 64-bitnimi števili. Zgoščeno vrednost ne moremo več računati na enak način, saj v izrazu

$$H(i,m) = (s[i] \cdot B^{m-1} + s[i+1] \cdot B^{m-2} + \ldots + s[i+m-2] \cdot B + s[i+m-1]) mod Q$$

lahko še vedno pride do računanja z velikimi števili. Zato bomo vrednosti računali po Hornerjevi metodi.

Po Hornerjevi metodi lahko polinome izračunamo z

$$a + b \cdot x^1 + c \cdot x^2 + d \cdot x^3 = a + x \cdot (b + x \cdot (c + d \cdot x)).$$

S tem postopkom lahko izračunamo zgoščeno vrednost po korakih, kjer vsakič računamo po modulu Q. Zgoščeno vrednost bomo torej izračunali z naslednjim postopkom:

```
def hash_modulo(besedilo, i, m, B, Q):
  '''Izracuna hash vrednost podniza na i-tem mestu dolzine m.
                                        '''
  h = 0
  for j in range(m):
      h = (h * B + ord(besedilo[i+j])) % Q
      #ord() poda ASCII vrednost znaka
  return h
```

Na ta način zagotovimo, da števila ne presežejo meje 64-bitnega zapisa.

Vrednost podniza na mestu $i+1$ iz prejšnje vrednosti s pomočjo modula izračunamo z

$$H(i+1,m) = ((H(i,m) + Q - (M \cdot s[i-1]) mod Q) \cdot B + s[i+m-1]) mod Q.$$

V zgornjem izrazu črka M pedstavlja člen $B^{m-1} mod Q$, ki ga moramo izračunati na poseben način, da se izogemo velikim številom. Izračunamo ga z naslednjim postopkom:

```
M = 1
for j in range(m-1):
    M *= B
    M = M % Q
```

Zgornja metoda spet izračuna novo vrednot v konstantnem času. Polinomska zgoščevalna funkcija z modulom Q sedaj slika iz nizov v množico $0, 1, ..., Q-1$. Z njo smo se izognili računanju z velikimi števili, smo pa povečali možnost trčenj. Možnost trčenj je približno $\frac{1}{Q}$ torej približno 10^{-9}.

3.2 Algoritem

Kot smo že povedali, Rabin-Karp algoritem primerja zgoščene vrednosti vseh možnih podnizov dolžine m z zgoščeno vrednostjo vzorca. Pred izvajanjem glavne zanke algoritma moramo izračunati zgoščeno vrednost vzorca in prvega podniza, to je podniz na mestu 0 dolžine m. Prav tako moramo izračunati člen $M = b^{m-1} mod Q$, ki se pojavi v vozeči zgoščevalni funkciji.

Sedaj lahko začnemo premikati podniz po besedilu od indeksa 0 do indeksa $n - m$. Na vsakem koraku izračunamo novo zgoščeno vrednost s pomočjo vozeče zgoščevalne funkcije. To vrednost primerjamo z zgoščeno vrednostjo vzorca. Če sta enaki, primerjamo podniz z vzorcem po posameznih znakih, tako kot pri naivnem pristopu. Če s tem načinom ugotovimo, da sta niza enaka, lahko dodamo trenuten indeks v tabelo ujemanja. Ko podniz doseže konec besedila, je postopek končan.

Zgled 3.1. Poiščimo spet vzorec *čaka* v besedilu *Kdor čaka, dočaka.* Za bazo v zgoščevalni funkciji bomo vzeli $B = 257$, za modul pa $Q = 999999999989$. V tem primeru bo zgoščena vrednost vzorca enaka 4572599866, zgoščena vrednost prvega podniza *Kdor* pa 1279728016. Sedaj lahko premikamo podniz po besedilu. V naslednjem koraku bo zgoščena vrednost podniza *dor* enaka 1704820069, ki se ne ujema z vrednostjo vzorca, torej nadaljujemo. Naslednji podniz *or č* ima vrednost 1891717902, ki se spet ne ujema z vzorcem, torej nadaljujemo. Enako se zgodi za vse podnize, razen na petem in trinajstem mestu. Na teh dveh mestih se vrednost podniza ujema z vrednostjo vzorca, torej primerjamo podniz z vzorcem vsako črko posebaj. Na koncu dobimo tabelo indeksov ujemanj [5, 13].

Spodaj je še celotna koda algoritma v python jeziku:

```
  def hash_modulo(besedilo, i, m, B, Q):
    '''polinomska hash funkcija z modulom'''
    h = 0
    for j in range(m):
        h = (h * B + ord(besedilo[i+j])) % Q
    return h

def rolling_hash_modulo(besedilo, i, m, B, prejsnji_hash, Q, M):
    '''polinomska rolling hash funkcija s polinomom'''
    return ((prejsnji_hash + Q - (M * ord(besedilo[i - 1]) % Q ))
                                      * B + ord(besedilo[i + m -
                                    1])) % Q

def RabinKarp_modulo(besedilo, vzorec, B, Q):
    n = len(besedilo)
    m = len(vzorec)
    ujemanja = []
```

```
    M = 1
    for j in range(m-1): #izrauna B^(m-1)modQ
        M *= B
        M = M % Q

    hash_vzorec = hash_modulo(vzorec, 0, m, B, Q) #hash vzorca

    hash_podniz = None
    st_primerjav = 0
    for i in range(n-m + 1): #premika podniz cez besedilo
        if hash_podniz is None: #zacetni hash podniza
            hash_podniz = hash_modulo(besedilo, i, m, B, Q)
        else: #naslednji hash podniza
            hash_podniz = rolling_hash_modulo(besedilo, i, m, B,
                                              hash_podniz, Q, M)

        if hash_podniz == hash_vzorec: #vrednosti ujemata
            ujema = True
            for j in range(m): #preveri po crkah
                st_primerjav += 1
                if vzorec[j] != besedilo[i+j]:
                    ujema = False
                    break
            if ujema:
                ujemanja.append(i)

    return ujemanja, st_primerjav
```

Začetni izračuni zgoščene vrednosti vzorca in prvega podniza ter izračun člena $M = b^{m-1} mod Q$ imajo časovno zahtevnost $O(m)$. V zanki moramo (n-m)-krat izračunati novo vrednost v konstantnem času. Opraviti moramo še k primerjav z zahtevnostjo $O(m)$, če imamo k ujemanj zgoščenih vrednosti. V najslabšem primeru ima algoritem časovno zahtevnost $O(n \cdot m)$, v najboljšem pa $O(n + m)$.

3.3 Nedeterministična verzija

Pri tej verziji bomo uporabili dve polinomski zgoščevalni funkciji z različnima bazama B in moduloma Q. Na začetku postopka z obema funkcijama izračunamo dve različni zgoščeni vrednosti za vzorec in prvi podniz. Podobno kot pri navadni verziji, sedaj podniz premikamo čez besedilo. Razlika je v tem, da ne primerjamo samo vrednosti ene zgoščevalne funkcije in pri enakih vrednostih primerjamo znake. Pri tej verziji primerjamo vrednost podniza prve funkcije z vrednostjo vzorca prve funkcije in vrednost podniza druge funkcije z vrednostjo vzorca druge funkcije. Če je prva vrednost vzorca enaka prvi vrednosti podniza in druga vrednost vzorca enaka drugi vrednosti podniza, predpostavimo, da sta niza enaka. V tem primeru dodamo indeks v tabelo, brez primerjanja posameznih znakov.

Zgled 3.2**.** Recimo, da imamo dve različni zgoščevalni funkciji, določen vzorec in določeno besedilo. Prva funkcija vrne za zgoščeno vrednost vzorca vrednost 10, druga pa 20. Sedaj premikamo podniz po besedilu in z vsako funkcijo izračunamo zgoščeno vrednost podniza. Indeks, kjer se niza ujemata, dodamo v tabelo samo v primeru, ko je prva vrednost podniza enaka 10 in druga vrednost podniza enaka 20.

To verzijo imenujemo nedeterministična, saj ne vemo zagotovo, da sta niza, ki data obe vrednosti enaki, res enaka. Kot smo že povedali, je verjetnost, da vrneta dva različna niza enako zgoščeno vrednost, približno enaka $\frac{1}{Q}$. Verjetnost, da pride do trčenja pri dveh različnih funkcijah hkrati, je torej $\frac{1}{Q_1} \cdot \frac{1}{Q_2}$, torej približno 10^{-18}.

4 Viri

- Rabin–Karp algorithm, Wikipedia (Dostopno 23.6.2022). Pridobljeno s `https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm`
- Rolling hash, Wikipedia (Dostopno 23.6.2022). Pridobljeno s `https://en.wikipedia.org/wiki/Rolling_hash`
- Overview of Rabin-Karp Algorithm, Baeldung (Dostopno 23.6.2022). Pridobljeno s `https://www.baeldung.com/cs/rabin-karp-algorithm`
- Rabin-Karp Algorithm Using Polynomial Hashing and Modular Arithmetic, Medium (Dostopno 23.6.2022). Pridobljeno s `https://medium.com/swlh/rabin-karp-algorithm-using-polynomial-hashing-and-modular-arithmetic-437627b37db6`
- Rabin-Karp Algorithm for Pattern Searching, GeeksforGeeks (Dostopno 23.6.2022). Pridobljeno s `https://www.geeksforgeeks.org/rabin-karp-algorithm-for-patterr`