Horspoolov algoritem

Tadej Žerak

→ Gotovo ste že kdaj na spletni strani ali v kakšnem besedilu iskali določeno besedo. Kako ste se tega lotili? Ena možnost je, da ste brali celotno besedilo in upali, da besedo sami čim hitreje najdete. Verjetneje pa je, da ste uporabili posebno funkcijo, vpisali iskano besedo in kaj hitro vam je računalnik pokazal, kje se iskana beseda nahaja. Ali ste se kdaj vprašali, kako to iskanje sploh deluje?

Načinov je več. Najenostavnejši je iskanje z naivno metodo, ki primerja vsak znak iz besedila z vsakim iz iskanega niza, vendar je ta metoda časovno potratna. V tem prispevku bomo predstavili Boyer-Moore-Horspool algoritem oz. krajše Horspoolov algoritem, ki ga je leta 1980 objavil ameriški profesor računalništva R. Nigel Horspool. Algoritem učinkoviteje in enostavno reši problem iskanja vzorca v besedilu.

Delovanje

Horspoolov algoritem razdeli iskanje na dva dela, in sicer na predprocesiranje in na samo iskanje. Recimo, da imamo besedilo, dolžine n ter dolžino iskanega vzorca m.

Predprocesiranje

Horspoolov algoritem poteka tako, da najprej izračuna, kolikšen zamik je potreben besedilu za vsak različen znak v besedilu, ta pa znaša toliko, da dosežemo ujemanje zadnjega znaka v vzorcu z besedilom. Naredimo tabelo, ki jo imenujemo tabela zamikov (ang. *Bad Match Table*, krajše *BMT*); za znak x jo izračunamo tako:

- $BMT[x] =$

$$= \begin{cases} \min(m-i-1); \\ \quad i = \text{indeks mesta, kjer se nahaja znak } x, \\ m; \quad \text{znaka ni v vzorcu, ali pa je zadnji.} \end{cases}$$

Poudarimo naj, da v primeru, da se znak x v vzorcu pojavi večkrat, hkrati pa tudi na zadnjem mestu, za ta znak uporabimo število, izračunano z enačbo $\min(m-i-1)$.

Algoritem bomo predstavili na primeru.
Besedilo, po katerem iščemo:
BONUMCOMMUNECOMMUNITATIS.
Vzorec: *ECOMMU.*

Za posamezni znak v vzorcu bomo izračunali zamik.

- $\text{BMT}[znak] = m - i - 1, \qquad m = 6$

- $\text{BMT}["E"] = 6 - 0 - 1 = 5$
  $\text{BMT}["C"] = 6 - 1 - 1 = 4$
  $\text{BMT}["O"] = 6 - 2 - 1 = 3$
  $\text{BMT}["M"] = 6 - 3 - 1 = 2$
  $\text{BMT}["M"] = 6 - 4 - 1 = 1$
  $\text{BMT}["U"] = 6$ (ker je zadnji znak v nizu)

Imamo torej takšen rezultat, kot je prikazano v tabeli 1.

E	C	O	M	U	Vsi ostali
5	4	3	1	6	6

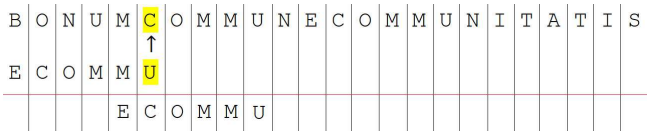
TABELA 1.

Iskanje

Po končani izdelavi tabele zamikov lahko izvedemo iskanje vzorca v besedilu, ki ga bomo razložili na primeru.

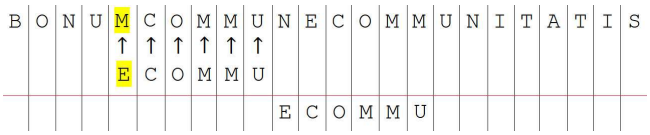
Najprej poravnamo vzorec in besedilo na prvem znaku. Algoritem deluje tako, da primerjamo vzorec z besedilom od desne proti levi. Pri primerjavi zadnjega znaka vzorca z istoležnim znakom v besedilu sta dve možnosti: ali se zadnji znak ujema ali se pa ne.

V primeru, da se zadnji znak vzorca ne ujema z znakom v besedilu, sam vzorec prestavimo naprej za toliko mest, kolikor je v tabeli zamikov določeno, in sicer za tisti znak iz besedila, katerega smo primerjali. Če se znaki ujemajo, pa primerjamo znake posamično od desne proti levi tako dolgo, dokler ne najdemo neujemanja ali pridemo do začetka vzorca. V zadnjem primeru z iskanjem zaključimo, saj smo našli popolno ujemanje. V primeru, da najdemo neujemanja, pa se cel niz ponovno zamakne za toliko mest, kot je določeno v tabeli zamikov, in sicer za prvi primerjani znak v besedilu.



SLIKA 1.
Primerjani znak U se ne ujema s C.

Kot vidimo na sliki 1, že pri prvem primerjanju pride do neujemanja, zato se vzorec zamakne za toliko mest, kolikor ima C vrednost v tabeli zamikov. Vzorec se torej zamakne za štiri mesta. Ko je zamaknjen, postopek ponovimo.

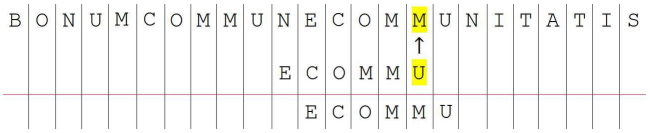


SLIKA 2.
Primerjani znaki se ujemajo, razen E z M.

Nadaljujemo z iskanjem; slika 2 kaže, da je prišlo do ujemanja zadnjega znaka. V tem primeru primerjamo znake posamično od desne proti levi.

Imamo ponovno ujemanje, tokrat znaka M, zato ponovimo postopek. Sledijo ujemanja znakov M, O ter C. Za tem sledi neujemanje znakov, in sicer znakov E in M. Zaradi neujemanja znaka se cel vzorec zamakne za toliko mest, kot je določeno v tabeli neujemanj, in sicer za prvi primerjani znak v besedilu, torej za BMT[U], ki je enako 6.

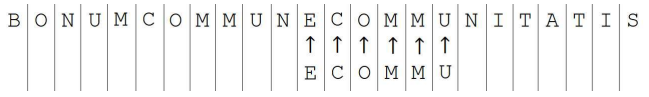
V naslednjem primerjanju po zamiku vzorca vidimo na sliki 3, da imamo neujemanje znakov U ter M, zato zamaknemo vzorec za BMT[M]= 1 mesto. V



SLIKA 3.
Primerjani znak U se ne ujema z M.

primeru, da bi bil na mestu znaka M drug znak, ki ne bi imel posebnega mesta v tabeli zamikov, npr. znak A, bi se vzorec zamaknil za šest mest, kot je v tabeli zamikov zapisano pod *Vsi ostali*, torej za celotno dolžino vzorca. Razmislite, zakaj to naredimo?

V naslednjem koraku dobimo:



SLIKA 4.
Primerjani znaki se ujemajo.

Po prvi primerjavi na sliki 4 opazimo, da imamo ujemanje znaka U, zato se premaknemo za en znak v levo in naredimo novo primerjavo. Sledi ujemanje znakov M, M, O, C ter E. Opazimo, da smo prišli do začetka vzorca, zato smo zaključili z iskanjem le-tega v danem besedilu.

Celotno iskanje torej razdelimo na dva dela: na predprocesiranje in iskanje. Predprocesiranje deluje po naslednjem algoritmu:

Algoritem 1 Predprocesiranje Horspoolovega algoritma

Input: vzorec
Output: tabela zamikov - *BMT*

```
1: for each znak v vzorcu do
2:     i ← mesto znaka v vzorcu
3:     if znak ni zadnji v vzorcu then
4:         BMT[znak] ← min(BMT[znak], m−i−1)
5:     else
6:         BMT[znak] ← min(BMT[znak], m)
7:     end if
8: end for
```

Po končanem predprocesiranju nadaljujemo iskanje po naslednjem algoritmu:

Algoritem 2 Iskanje s Horspoolovim algoritmom

Input: vzorec, besedilo, tabela zamikov
Output: mesto prvega znaka ujemanja vzorca v besedilu

```
1:  t ← m − 1                          ▷ t... prvi znak primerjanja
2:  k ← m − 1   ▷ k... števec, ki primerja po besedilu
3:  while k ≤ n do
4:      i ← m − 1 ▷ i... števec, ki primerja po vzorcu
5:      while vzorec[i] = besedilo[k] do
6:          i ← i − 1
7:          k ← k − 1
8:      end while
9:      if i = −1 then
10:         return k + 1
11:     else
12:         t ← t + BMT[besedilo[t]]
13:     end if
14:     k ← t
15: end while
16: return Ni ujemanja
```

Preizkus

V teoriji bi naj Horspoolov algoritem deloval hitreje kot Boyer-Moorov algoritem in naivna metoda. Ali to drži tudi v praksi? Kot vemo, je v algoritmu potrebno predprocesiranje, prav tako pa je potrebnih več operacij za ugotovitev, da se npr. zadnji znak v vzorcu ne ujema z istoležečim znakom v besedilu. Preskok vzorca je lahko večji.

Implementirali smo Horspoolov algoritem in naivno metodo ter prišli do naslednjih rezultatov: za najdbo vseh ponovitev besede *and* v približno 25.000 znakov dolgem besedilu je naivna metoda potrebovala povprečno 83 ms, Horspoolov algoritem pa le 64 ms. Za iskanje malo daljše besede *captain* je naivna metoda povprečno potrebovala 82 ms, medtem ko pa je Horspoolov algoritem potreboval 56 ms. Kot vidimo, Horspoolov algoritem pridobi na hitrosti z večjo dolžino vzorca.

Opazili smo tudi, če imamo majhno abecedo, torej majhen nabor različnih znakov v iskanem besedilu, Horspoolov algoritem pogosto ni hitrejši od enostavnejše naivne metode, kar je posledica predprocesiranja ter več potrebnih operacij za prvo primerjavo.

Oba algoritma smo preizkusili tudi na daljši datoteki, dolgi 692.945 znakov; vzorec je bil dolg 222 znakov. Naivni algoritem je ves dokument pregledal in našel vzorec s povprečnim časom 2302 ms, Horspoolov algoritem pa s povprečnim časom 1239 ms. Vsa iskanja so bila izvedena petkrat.

Kot vidimo, je implementacija Horspoolovega algoritma dobra odločitev, saj lahko tudi razpolovi potreben čas iskanja. To trditev potrjuje dejstvo, da ga programerji vpeljujejo v večje projekte, kot je iskanje po spletnih straneh v spletnih brskalnikih Google Chrome in Mozilla Firefox ali pa v Microsoft Office paketu, katerega del je tudi Microsoft Office Word.

Literatura

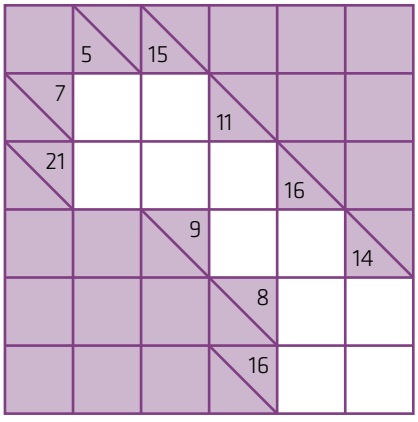
[1] G. Navarro, M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*, Cambridge press, Cambridge, 2002.

× × ×

Križne vsote

↓ ↓ ↓

→ Naloga reševalca je, da izpolni bele kvadratke s številkami od 1 do 9 tako, da bo vsota števk v zaporednih belih kvadratkih po vrsticah in po stolpcih enaka številu, ki je zapisano v obarvanem kvadratku na začetku vrstice (stolpca) nad (pod) diagonalo. Pri tem morajo biti vse števke v posamezni vrstici (stolpcu) različne.



× × ×