Porovnávanie reťazcov

Úvod

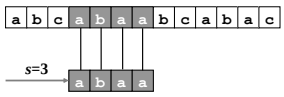
- Porovnávanie reťazcov, presnejšie hľadanie výskytov reťazca v reťazci (string-matching) je pomerne dôležitou súčasťou širokej domény zaoberajúcej sa spracovaním textu. Algoritmy na porovnávanie textov sa využívajú pri implementácií softvérových systémov, ktoré sú reálne nasadené v praxi. Takisto však hrajú dôležitú rolu v teoretickej informatike, kde môžu byť výzvou pre navrhovanie efektívnejších algoritmov.

základné pojmy

- S="AGCTTGA"
- $|S|$=7, dĺžka reťazca S
- podreťazec: $S_{i,j}=S_iS_{i+1}\ldots S_j$
  - príklad: $S_{2,4}$="GCT"
- podpostupnosť reťazca S: vymazaním niekoľkých (vrátane žiadneho) znakov z S
  - "ACT" and "GCTT" sú podpostupnosti.
- predpona S: $S_{1,k}$
  - "AGCT" je predpona S.
- prípona S: $S_{h,|S|}$
  - "CTTGA" je prípona S.

základné pojmy

- Uvažujme 2 reťazce:
  - Vzor P[1...*m*], ktorý má dĺžku *m*
  - Text T[1...*n*], ktorý má dĺžku *n*
- Vzor P sa nachádza v texte T s posunutím *s* ak platí:
  - T[*s+1...s+m*] = P[*1...m*]
- Príklad: T = abcabaabcabac, P = abaa
  - *m*=4, *n*=13, *s*=3

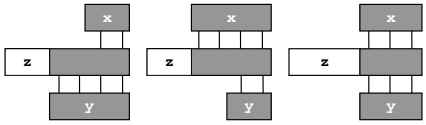


základné pojmy

- Predpona (prefix): reťazec *w* je predponou reťazca *x*, ak (existuje reťazec y taký, že) *x* = *wy*, kde *y* je akýkoľvek reťazec z použitej abecedy *Σ,* t.j. prvok z množiny Σ^*
  - Napr: pre(*ab,abcca)*
- Prípona (suffix): reťazec *w* je príponou reťazca *x*, ak (existuje reťazec y taký, že) *x* = *yw*, kde *y* je akýkoľvek reťazec z použitej abecedy *Σ,* t.j. prvok z množiny Σ^*
  - Napr: suf(*cca,abcca)*

Lema

- Predpokladajme, že "*x*", "*y*" a "*z*" sú reťazce, pre ktoré platí suf(*x*, *z*) a suf(*y*, *z*), potom:
  - ak $|x| \leq |y|$, *tak suf(x,y)*
  - *ak* $|x| \geq |y|$, *tak suf(y,x)*
  - *ak* $|x| = |y|$, *tak x = y*



najznámejšie algoritmy

Algoritmus	fáza predspracovania	Vyhľadávacia fáza
Naivný	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Konečný automat	$O(m\|\Sigma\|)$	$\Theta(m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(m)$

- Ďalšie algoritmy, ich opisy, vizualizácie a zdrojové kódy môžete nájsť napr. na : http://www-igm.univ-mlv.fr/~lecroq/string/

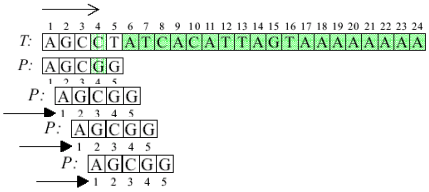
7

Naivné hľadanie výskytu reťazca v reťazci

- Nemá fázu predspracovania
- Vždy sa posúva len o 1 pozíciu doprava
- Porovnávanie môže prebiehať v akomkoľvek poradí
- Veľká časová zložitosť
- Vykoná sa 2n porovnávaní textu

8

naivné = hrubou silou



čas: O(*mn*) kde *m*=|*P*| a *n*=|*T*|.

9

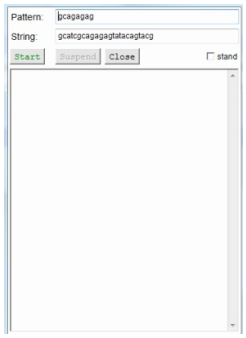
Naivné hľadanie výskytu reťazca v reťazci

- Samotný algoritmus pozostáva z porovnávania znakov na všetkých miestach medzi 0 a *n-m*. Pri každom kroku sa posúva iba o 1 miesto doprava.

Naivne porovnavanie (T, P)
n ←*length[T]*
m ←*length[P]*
for *s* ←*0 to n −m*
  do if *P[1 . . . m] = T[s+1. . . s+m]*
    then
    print "vyskytuje sa s posunutim" *s*

10

Naivné hľadanie výskytu reťazca v reťazci



11

Naivné hľadanie výskytu reťazca v reťazci

```
C++:
void BF(char *x, int m, char *y, int n)
{
int i, j;
/* Searching */
for (j = 0; j <= n - m; ++j)
{
  for (i = 0; i < m && x[i] == y[i + j]; ++i); if (i >= m)
            OUTPUT(j);
} }
```

12

2 druhy algoritmov hľadania výskytu reťazca

- predspracovanie vzoru P (P sa nemení, T sa mení)
  - napr: dopyt P do databázy T
  - algoritmy:
    - Knuth – Morris - Pratt
    - Boyer – Moore
- predspracovanie textu T (T sa nemení, P sa mení)
  - napr: hľadá sa vzor P v slovníku T
  - algoritmus:
    - príponový strom

13

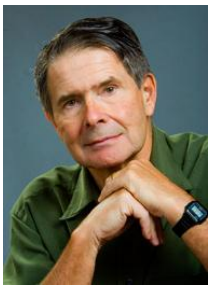
Predspracovanie vzoru

- dvojfázové algoritmy
  - fáza 1：vygeneruj pole, ktoré bude indikovať smer pohybu.
  - fáza 2：použi to pole na pohyb a hľadanie výskytu
- príklady
- algoritmus KMP:
  - navrhli Knuth, Morris a Pratt v 1977.
- algoritmus BM:
  - navrhli Boyer a Moore v 1977.

14

James H. Morris

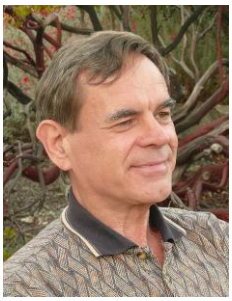
- 1941, Pittsburgh –
- Bc CMU
- Master manažment, MIT
- PhD informatika, MIT
- profesor, CMU
- práce
  - Donald Knuth, James H. Morris, Jr., and Vaughan Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2):323–350. 1977.



15

Vaughan Pratt

- 1944 -
- emeritný profesor
- Stanfordova univerzita
- práce
  - Donald Knuth, James H. Morris, Jr., and Vaughan Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2):323–350. 1977.



16

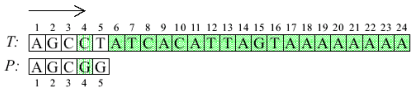
Knuthov-Morrisov-Prattov algoritmus

- KMP algoritmus vychádza z analýzy algoritmu naivného vyhľadávania. V určitých situáciách vie využiť informáciu získanú čiastočným porovnávaním vybraného podreťazca a vzoru a posunúť podreťazec o viac než jeden znak.

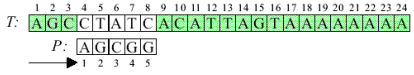
17

prvý prípad v KMP algoritme

- prvý symbol vzoru P sa viac vo vzore P nenachádza.
- možno sa posunúť až ku T_4, pretože $T_4 \neq P_4$ (a $T_i = P_i$ pre prvé tri pozície i=1,2,3) v (a).



(a)

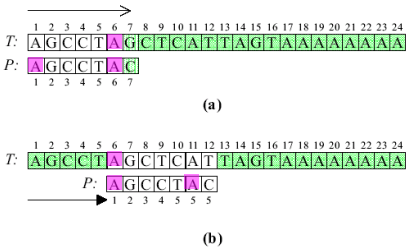


(b)

18

druhý prípad v KMP algoritme

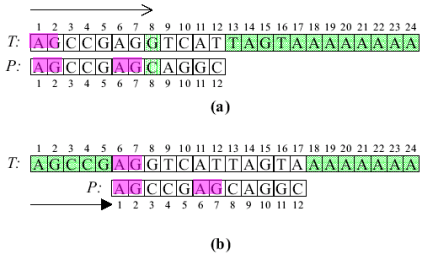
- prvý symbol vo vzore *P* sa v ňom ešte nachádza na niektorom ďalšom mieste.
- $T_7 \neq P_7$ v (a). treba sa posunúť ku T_6, pretože $P_6=P_1=T_6$.



19

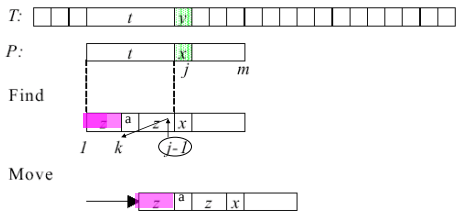
tretí prípad v KMP algoritme

- predpona vzoru *P* sa vyskytuje v *P* ešte raz.
- $T_8 \neq P_8$ v (a). Treba sa posunúť k T_6, pretože $P_{6,7}=P_{1,2}=T_{6,7}$.



20

základná myšlienka KMP algoritmu



21

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D

Y Y Y N

m = 0

22

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D

Y Y Y Y Y Y N

m = 4

23

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D

N

m = 10

24

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	..
A	B	C	..

Y Y Y

m = 11

25

Knuthov-Morrisov-Prattov algoritmus

algorithm *kmp_search*:
Input pole znakov T (text, v ktorom sa hľadá výskyt vzoru)
pole znakov P (text, vzor, ktorého výskyt sa hľadá)
Output celé číslo (miesto v poli T, počínajúc ktorým sa našiel výskyt vzoru P)

define variables:
integer m ← 0 (začiatočné miesto súčasného výskytu v T)
integer i ← 0 (miesto súčasného znaku v P)
pole celých čisiel π (tabuľka, ktorá sa vypočíta inde)
while m + i < dĺžka π do:
if P[i] = T[m + i] **then**
let i ← i + 1
if i = dĺžka P **then return** m
else
let m ← m + i - π[i],
if i > 0 **then let** i ← π[i]

26

KMP algoritmus – pomocná tabuľka

algorithm *kmp_table*:
input: pole znakov P (vzor, ktorého výskyt sa hľadá)
out**put**: pole celých čisiel π (tabuľka, ktorú treba vyplniť)

define variables:
integer i ← 2 (súčasné miesto v π, pre ktoré sa počíta hodnota π)
integer j ← 0 (pozícia znaku v P, počítajúc od nuly, ktorý je nasledujúcim znakom v súčasnom podreťazci, ktorý je kandidátom na zhodu)

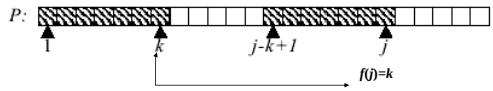
let π[0] ← -1, π[1] ← 0
while i < dĺžka P, do:
(prvý prípad: podreťazec pokračuje)
if P[i - 1] = P[j]
then let π[i] ← j + 1, i ← i + 1, j ← j + 1
(druhý prípad: nepokračuje, ale možno sa vrátiť)
else if j > 0 **then let** j ← π[j]
(tretí prípad: nie sú ďalší kandidáti. Všimnime si, že j = 0)
else let π[i] ← 0, i ← i + 1

27

definícia funkcie počítajúcej predponu

$f(j)$= najväčšie $k < j$ také, že $P_{1,k}=P_{j-k+1,j}$

$f(j)$=0 ak také k neexistuje

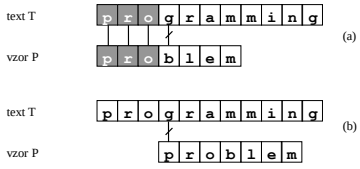


j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

28

Knuthov-Morrisov-Prattov algoritmus

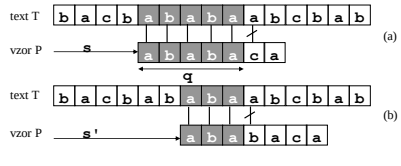
Porovnávanie vzoru s reťazcom začína na prvom znaku zľava (vzor je zarovnaný s reťazcom). Algoritmus postupuje, kým nenarazí na nezhodu na štvrtej pozícii medzi znakmi **b** a **g** (obr. a). Z predchádzajúcich znakov okamžite vieme, že posun vzoru o jeden alebo dva znaky nemá význam. Preto nastane posun o tri znaky. Tým sa vzor zarovná s textom nad znakom, kde nastala nezhoda. Od tohto miesta môže ďalej pokračovať porovnávane.



29

Knuthov-Morrisov-Prattov algoritmus

- V tomto príklade vidíme, že vzor je posunutý o **s** a *nastáva nové porovnávanie. Pri ňom sa zistilo, že nezhoda nastala na 6. pozícii reťazca, čo indikuje posun o 5 znakov (q). V tomto prípade však takýto posun nie je možný. Posunúť sa môžeme iba o 2 znaky, pretože na 3. znaku sa nachádza zhoda medzi týmto znakom a prvým znakom vzoru (na tomto mieste môže začínať vzor)*



30

Knuthov-Morrisov-Prattov algoritmus

- Posun pri prehľadávaní je nezávislý od prehľadávaného reťazca. Jeho veľkosť určuje tzv. predponová funkcia.
- predponová funkcia (Prefix function)

π pre P, |P| = m:

$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$

$\pi(q) = \max\{k : k < q, P_{1,k} \supset P_{q-k+1,q}\}$.

- Jednotlivé posuny sa vypočítavajú vo fáze predspracovania.

31

Knuthov-Morrisov-Prattov algoritmus

```
KMP POROVNANIE(T, P)
n ← dĺžka(T)
m ← dĺžka(P)
π ← PREDPONOVA FUNKCIA(P)
q ← 0
for i ← 1 to n        //prehľadávaj dalši zľava doprava
        do while q > 0 and P[q + 1] ≠ T[i]
                do q ← π[q]        //nezhoduje sa, bude „ústup“
           if P[q + 1] = T[i]
                then q ← q + 1    //zhoduje sa
           if q = m
                then print "Vzor sa v retazci vyskytuje s posunom " i-m
                     q ← π[q]
```

32

Knuthov-Morrisov-Prattov algoritmus

```
PREDPONOVA FUNKCIA (P)
// funkcia sa reprezentuje tabuľkou π, jej
hodnoty sa tu vypočítavajú a zapisujú do π
m ← dĺžka(P)
π[1] ← 0
k ← 0
for q ← 2 to m
        do while k > 0 and P[k + 1] ≠ P[q]
                do k ← π[k]
           if P[k + 1] = P[q]
                then k ← k + 1
           π[q] ← k
return π
```

33

príklad: vypočítať π pre vzor 'P':

p

a	b	a	b	a	c	a

na začiatku: m = dĺžka[P] = 7
$\pi[1] = 0$
k = 0

krok 1: q = 2, k=0
$\pi[2] = 0$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

krok 2: q = 3, k = 0,
$\pi[3] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

krok 3: q = 4, k = 1
$\pi[4] = 2$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

krok 4: q = 5, k =2
$\pi[5] = 3$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

krok 5: q = 6, k = 3
$\pi[6] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	1	

krok 6: q = 7, k = 1
$\pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	1	1

po 6 opakovaniach je výpočet funkcie π úplný: →

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	1	1

príklad: nech sú dané reťazec 'S' a vzor 'P':

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P

a	b	a	b	a	c	a

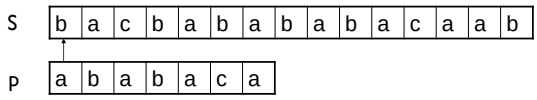
Na nájdenie výskytu vzoru p v texte S použijeme algoritmus KMP.

najprv sa pre vzor 'P' vypočíta predponová funkcia π daná tabuľkou (pozri prechádzajúci príklad)

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

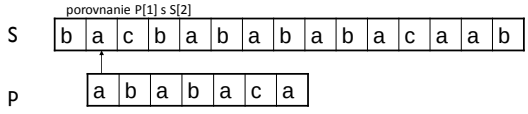
Na začiatku: n = dĺžka(S) = 15;
m = dĺžka(p) = 7

krok 1: i = 1, q = 0
porovnanie P[1] s S[1]



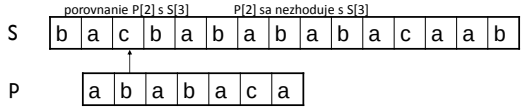
P[1] sa nezhoduje s S[1]. 'P' sa posunie o 1 miesto doprava.

krok 2: i = 2, q = 0
porovnanie P[1] s S[2]



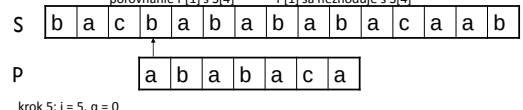
P[1] sa zhoduje s S[2]. Keďže došlo ku zhode, P sa neposunie.

krok 3: i = 3, q = 1
porovnanie P[2] s S[3] P[2] sa nezhoduje s S[3]

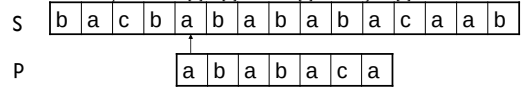


ústup v P, porovnanie P[1] s S[3]

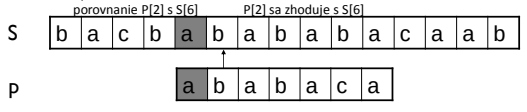
krok 4: i = 4, q = 0
porovnanie P[1] s S[4] P[1] sa nezhoduje s S[4]



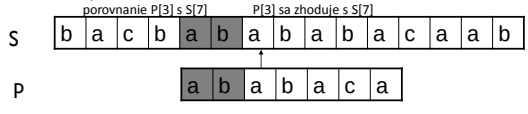
krok 5: i = 5, q = 0
porovnanie P[1] s S[5] P[1] sa zhoduje s S[5]



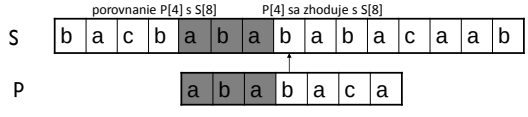
krok 6: i = 6, q = 1
porovnanie P[2] s S[6] P[2] sa zhoduje s S[6]



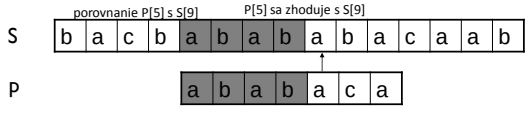
krok 7: i = 7, q = 2
porovnanie P[3] s S[7] P[3] sa zhoduje s S[7]



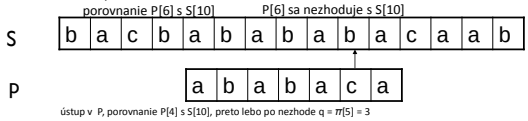
krok 8: i = 8, q = 3
porovnanie P[4] s S[8] P[4] sa zhoduje s S[8]



krok 9: i = 9, q = 4
porovnanie P[5] s S[9] P[5] sa zhoduje s S[9]

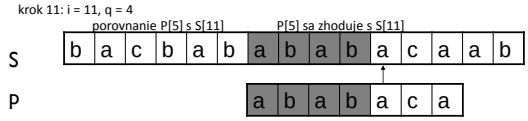


krok 10: i = 10, q = 5
porovnanie P[6] s S[10] P[6] sa nezhoduje s S[10]

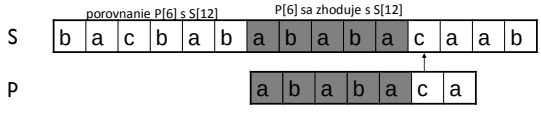


ústup v P, porovnanie P[4] s S[10], preto lebo po nezhode q = π[5] = 3

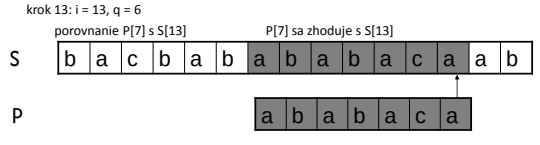
krok 11: i = 11, q = 4
porovnanie P[5] s S[11] P[5] sa zhoduje s S[11]



krok 12: i = 12, q = 5
porovnanie P[6] s S[12] P[6] sa zhoduje s S[12]



krok 13: i = 13, q = 6
porovnanie P[7] s S[13] P[7] sa zhoduje s S[13]



Vzor 'P' sme našli (jeho úplný výskyt) v texte 'S'. Celkový počet posunov, ktoré sa vykonali, aby sa našiel výskyt je: i – m = 13 – 7 = 6.

odhad času výpočtu

- *PREDPONOVA FUNKCIA (P)*

```
1  m ← dĺžka[p]          //"p' vzor
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
```

Cyklus for v riadkoch 4 až 10 sa vykoná 'm' ráz. Kroky 1 až 3 sa vykonajú v konštantnom čase. Preto odhad času výpočtu predponovej funkcie je Θ(m).

- KMP POROVNANIE(*T, P*)

```
1  n ← dĺžka[T]
2  m ← dĺžka[p]
3  Π ← predponová funkcia(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 "Vzor sa v retazci vyskytuje s
   posunom"  i – m
12             q ← Π[ q]
```

Cyklus začínajúci v riadku 5 sa vykoná 'n' ráz, t.j. toľko ráz, ako je dlhý text T. Keďže kroky 1 až 4 sa vykonajú v konštantnom čase, o celkovom čase výpočtu rozhoduje cyklus. Preto odhad času výpočtu KMP porovnania je Θ(n).

KMP algoritmus – časová zložitosť

- časová zložitosť: $O(m+n)$
  - $O(m)$ výpočet funkcie f
  - $O(n)$ hľadanie vzoru P

43

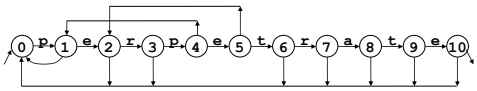
KMP algoritmus a konečný automat

- KMP algoritmus do určitej miery súvisí s konečnými automatmi. Predpokladajme, že máme vzor P dĺžky m. Definujeme si konečný automat, ktorý bude mať m+1 stavov. Prechody medzi jednotlivými stavmi budú postupne určené jednotlivými písmenami vzoru. Teda napr. prechod medzi nultým a prvým stavom bude podľa písmena p_1, prechod medzi prvým a druhým stavom podľa p_2 atď. Ostatné prechody (teda akési chybové) bude určovať práve predponová funkcia. Vstupným stavom bude stav 0 a výstupným stav m. Samotné vyhľadávanie bude realizované ako práca takéhoto automatu so vstupom, ktorý odpovedá zadanému reťazcu. Rozdiel je iba v tom, že pokiaľ sa pomocou predponovej funkcie vrátime do niektorého predchádzajúceho stavu, okamžite skúsime cez ten istý znak (ktorý spôsobil nezhodu) prejsť do nasledujúceho stavu.

44

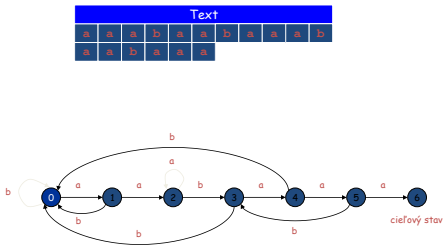
KMP algoritmus a konečný automat

- Príklad konečného automatu pre vzor perpetrate



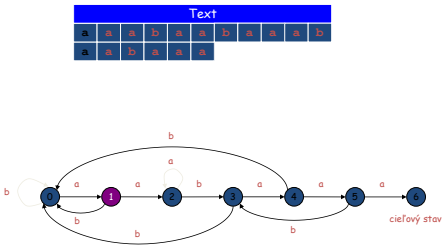
45

Knuth-Morris-Pratt

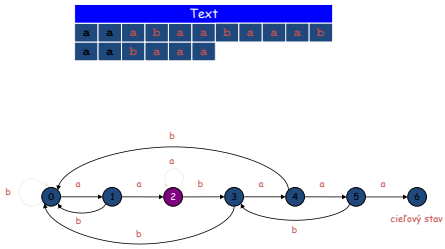


Knuth-Morris-Pratt

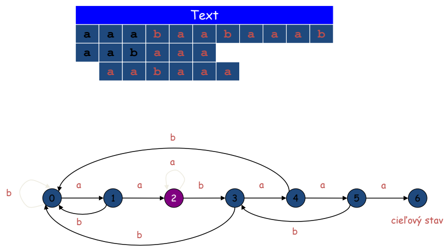
- .



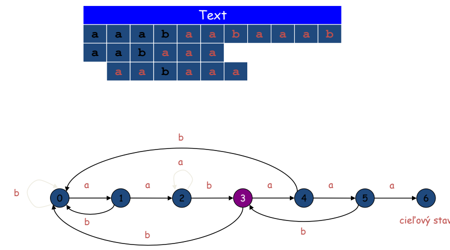
Knuth-Morris-Pratt



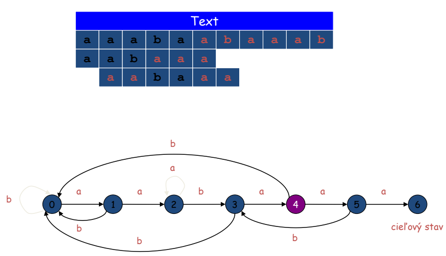
Knuth-Morris-Pratt



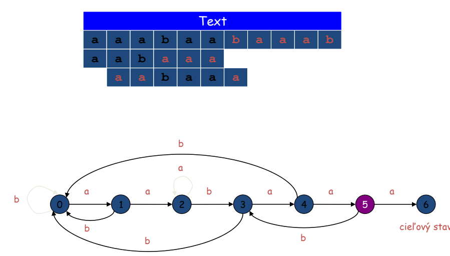
Knuth-Morris-Pratt



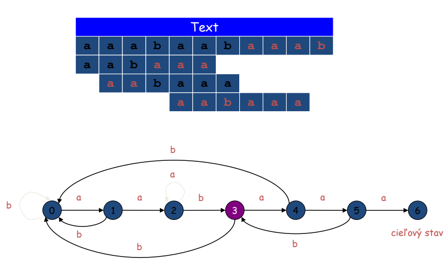
Knuth-Morris-Pratt



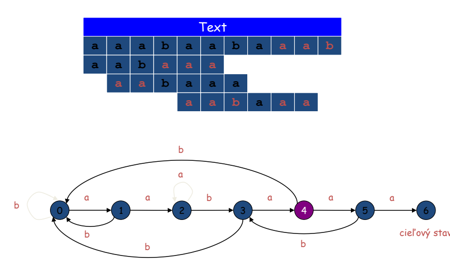
Knuth-Morris-Pratt



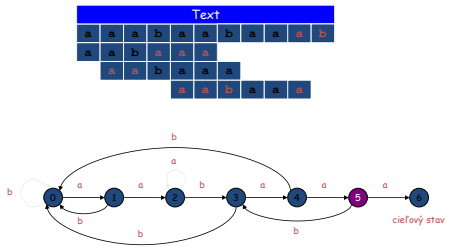
Knuth-Morris-Pratt



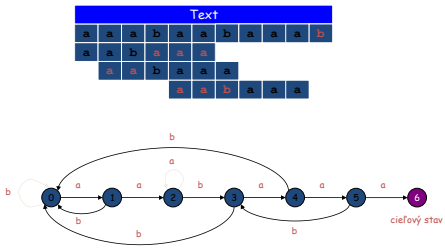
Knuth-Morris-Pratt



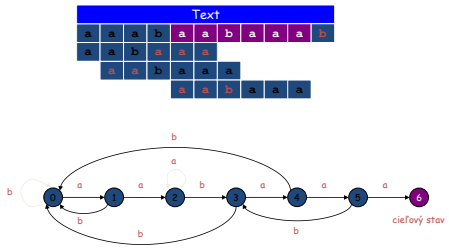
Knuth-Morris-Pratt



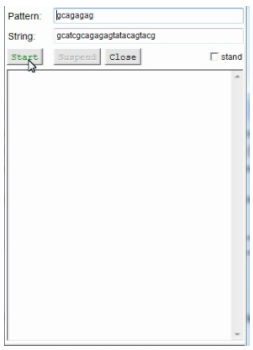
Knuth-Morris-Pratt



Knuth-Morris-Pratt



Knuthov-Morrisov-Prattov algoritmus



58

Knuthov-Morrisov-Prattov algoritmus - príklad implementácie

```
void preKmp(char *x, int m, int kmpNext[])
{
    int i, j; i = 0; j = kmpNext[0] = -1;
    while (i < m)
    {
            while (j > -1 && x[i] != x[j])
                    j = kmpNext[j];
            i++;
            j++;
            if (x[i] == x[j])
                    kmpNext[i] = kmpNext[j];
            else
                    kmpNext[i] = j;
    }
}
```

59

Knuthov-Morrisov-Prattov algoritmus - príklad implementácie

```
void KMP(char *x, int m, char *y, int n)
{
    int i, j, kmpNext[XSIZE];
    /* Preprocessing */
    preKmp(x, m, kmpNext);
    /* Searching */
    i = j = 0;
    while (j < n)
    {
            while (i > -1 && x[i] != y[j])
                    i = kmpNext[i];
            i++;
            j++;
            if (i >= m)
            {
                    OUTPUT(j - i);
                    i = kmpNext[i];
            }
    }
}
```

60

Robert S. Boyer

- emeritný profesor
- Computer Science Department
- The University of Texas at Austin
- práce:
  - BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

61

J Strother Moore

- profesor
- Computer Science Department
- The University of Texas at Austin
- práce:
  - BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.
  - dokazovanie teorém

62

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

N

Vo vzore nie je žiadny znak E: z toho plynie, že vzor nemožno nájsť nikde pred znakom t[3]. Preto sa možno a teda aj treba posunúť o 4 miesta doprava.

63

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

N

Nevyskytuje sa. Ale vo vzore sa B vyskytuje. Treba sa posunúť o dve miesta doprava.

64

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

Y Y Y Y

65

Boyerov-Moorov algoritmus

- porovnáva sprava doľava
- 2 predvypočítané funkcie
  - posun o dobrú príponu
  - posun o zlý znak

66

Boyerov-Moorov algoritmus

myšlienka č. 1: porovnávať sprava doľava

```
          12345678901234567
T:        xpbctbxabpqxctbpq
P:          tpabxab
```


Boyerov-Moorov algoritmus

```
          12345678901234567
T:        spbctbsabpqsctbpq
P:          tpabsab
P:            tpabxab
```

myšlienka č. 2: pravidlo zlého znaku

R(x): najpravejší výskyt znaku x v P. R(x)=0 ak sa x nevyskytuje. R(t)=1, R(s)=5.

i: miesto, kde došlo k nezhode v P. i=3

k: zodpovedajúce miesto v T. k=5. T[k]=t

pravidlo zlého znaku: Vzor P sa posunie doprava o max{1, i-R(T[k])}. čiže ak je najpravejší výskyt znaku T[k] v P na mieste j (j<i), tak P[j] sa ocitne pod T[k] po posunutí.

Boyerov-Moorov algoritmus

- myšlienkou pravidla zlého znaku je posunúť P o viac než o jeden znak, ak je to možné.
- pravidlo neúčinné, ak j>i
- bohužiaľ, prípad j>i je častý

```
          12345678901234567
T:        spbctbsatpqsctbpq
P:          tpabsat
P:      tpabsat
```


Boyerov-Moorov algoritmus

označme x=T[k] znak, ktorý sa nezhoduje so vzorom v T.

myšlienka č. 3: rozšírené pravidlo zlého znaku: P sa posunie doprava tak, že najbližšie x vľavo od miesta i v P bude pod T[K].

```
          12345678901234567
T:        spbctbsatpqsctbpq
P:          tpabsat
P:            tpabsat
```


Boyerov-Moorov algoritmus

aby sme mohli používať rozšírené pravidlo zlého znaku,
potrebujeme poznať:
pre každé miesto i v P, pre každý znak x v abecede, miesto najbližšieho výskytu x vľavo od miesta i.

možný prístup: dvojrozmerným poľom: n*| Σ |

pamäťovo a časovo príliš náročné

Boyerov-Moorov algoritmus

iný prístup: prezrieť P sprava doľava a pre každý znak x si držať zoznam miest, kde sa x vyskytuje (v klesajúcom poradí).

```
P:  tpabsat
    t→7,1
    a→6,3  …
```

ak dôjde k nezhode medzi P[i] a T[k], (označme x=T[k]), prezri zoznam výskytov znaku x, nájdi prvé číslo (označme ho j), ktoré je menšie než i a posuň P doprava tak, aby P[j] sa dostalo pod T[k].

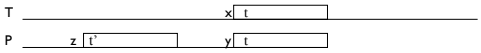
```
          12345678901234567
T:        spbctbsatpqsctbpq
P:          tpabsat
P:            tpabsat
```

ak také číslo j neexistuje, tak posuň P za T[k]

čas a pamäť: lineárna

Boyerov-Moorov algoritmus

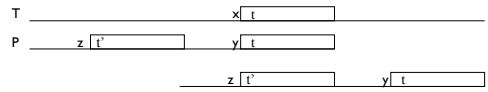
myšlienka č. 4: pravidlo dobrej prípony



t je prípona vzoru P taká, že sa zhoduje s podreťazcom t textu T a x≠y
t' je najpravejší výskyt t v P taký, že t' nie je príponou P a z≠y

Boyerov-Moorov algoritmus

pravidlo dobrej prípony:
(1) ak existuje t' tak posuň P doprava tak, že t' v P je pod t v T




```
  123456789012345678
T: prstabstubabvqxrst
P:   qcabdabdab
P:         qcabdabdab
P:       qcabdabdab
```


Boyerov-Moorov algoritmus

- rozšírené pravidlo zlého znaku sa sústreďuje na znaky.
- pravidlo dobrej prípony sa sústreďuje na podreťazce.

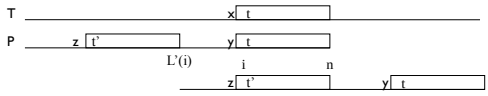
- ako teraz získať informáciu, ktorú treba pre pravidlo dobrej prípony? ako pre nejaké t nájsť t`?

Boyerov-Moorov algoritmus

L'(i): pre každé i, L'(i) je najväčšie miesto menšie než n také, že podreťazec P[i,…,n] sa zhoduje s príponou reťazca P[1,…, L'(i)] a navyše znak predchádzajúci tejto prípone je rôzny od P[i-1].

ak také miesto neexistuje tak L'(i) =0.

nech t= P[i,…,n], potom L'(i) je miesto pravého konca t'.




```
T:  prstabstubabvqxrst
P:    qcabdabdab
      1234567890
L'(9)=4, L'(10)=0, L'(8)=?, L'(7)=? L'(6)=?
```

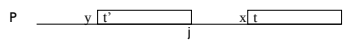

Boyerov-Moorov algoritmus

nech t= P[i,…,n], potom L'(i) je miesto pravého konca t'.

aby sa dalo používať pravidlo dobrej prípony, treba poznať L'(i) pre všetky i=1,…,n.

pre vzor P:

N_j je dĺžka najdlhšieho podreťazca, ktorý končí na mieste j a ktorý je príponou P.

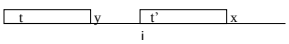


t=t';
j=|t'|=|t|;
x≠y

Boyerov-Moorov algoritmus

N_j: dĺžka najdlhšieho podreťazca, ktorý končí na mieste j a ktorý je príponou P.

Z_i: dĺžka najdlhšieho podreťazca, ktorý začína na mieste i a ktorý je predponou P.



Boyerov-Moorov algoritmus

N je obrátené Z!
P^r je obrátený reťazec P

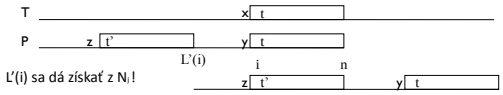
všimnime si, že $N_j(P)=Z_{n-j+1}(P^r)$

```
    1 2 3 4 5 6 7 8 9 0        1 2 3 4 5 6 7 8 9 0
P:  q c a b d a b d a b    Pʳ:  b a d b a d b a c q
Nj: 0 0 0 2 0 0 5 0 0 0    Zi   0 0 0 5 0 0 2 0 0 0
```

Boyerov-Moorov algoritmus

pre daný vzor P,
N_i (pre j=1,…,n) sa dá vypočítať v lineárnom čase O(n) algoritmom Z.

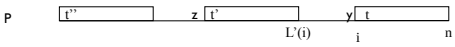
prečo treba určiť N_j ?
aby sa dalo použiť pravidlo dobrej prípony, treba poznať L'(i) pre všetky i=1,…,n.



L'(i) sa dá získať z N_i !

Boyerov-Moorov algoritmus

Pre miesto i, nech t=P[i,…n].
L'(i) je najväčšie miesto j menšie než n také, že N_i=|t|



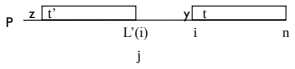

```
    1 2 3 4 5 6 7 8 9 0        1 2 3 4 5 6 7 8 9 0
P:  q c a b d a b d a b    Pʳ:  b a d b a d b a c q
Nj: 0 0 0 2 0 0 5 0 0 0    Zi   0 0 0 5 0 0 2 0 0 0
L'(i): 0 0 0 0 0 7 0 0 4 0
```


Boyerov-Moorov algoritmus

Ako získať L'(i) z N_i v lineárnom čase?

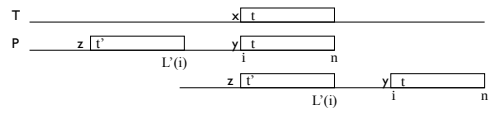
vstup: vzor P
výstup: L'(i) pre i=1,…,n
Algoritmus

```
vypočítaj Nj pre j=1,…,n na základe algoritmu Z
for i=1; i<=n; i++
    L'(i)=0;
for j=1; j<n; j++
    i=n-Nj+1
    L'(i)=j;
```



Boyerov-Moorov algoritmus

Pravidlo dobrej prípony:
(1) ak existuje t', tak posuň P doprava takým spôsobom, že t' v P je pod t v T



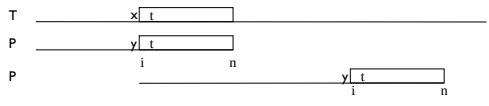

```
   123456789012345678
T: prstabstubabvqxrst
P:   qcabdabdab        i=9; L'(9)=4
P:         qcabdabdab
```


Boyerov-Moorov algoritmus

Pravidlo dobrej prípony:

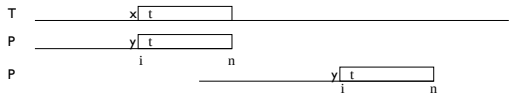
(1) Ak nastáva nezhoda na mieste i-1 v P a L'(i)>0 (t.j. existuje t'), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.

(2) Čo ak nastane nezhoda na mieste i-1 v P a L'(i)=0 (t.j. t' neexistuje)? Možno P posunúť takto



Boyerov-Moorov algoritmus

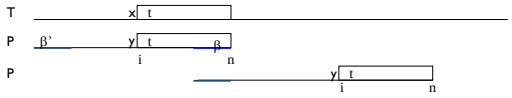
Dá sa však spraviť viac!



85

Boyerov-Moorov algoritmus

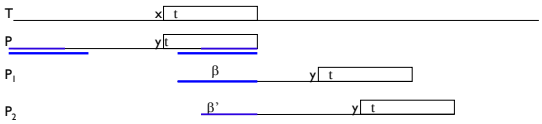
Pozorovanie 1: ak β je predponou P a je tiež príponou P, tak...



86

Boyerov-Moorov algoritmus

pozorovanie 2: ak je viac kandidátov β, tak posuň P o čo najmenšiu dĺžku



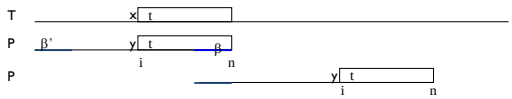
87

Boyerov-Moorov algoritmus

Pravidlo dobrej prípony: keď nastane nezhoda na mieste i-1 vzoru P

(1) ak L'(i)>0 (t.j. t' existuje), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.

(2) inak ak L'(i)=0 (t.j. t' neexistuje)? Možno posunúť P poza ľavý koniec t o najmenší počet miest taký, že predpona posunutého vzoru sa zhoduje s príponou t.

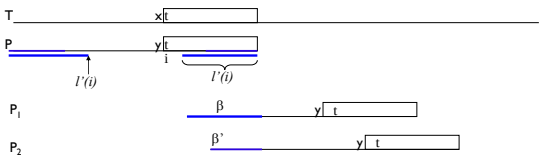


88

Boyerov-Moorov algoritmus

l'(i) : dĺžka najväčšej prípony P[i,...,n] takej, že je aj predponou vzoru P. Ak taká neexistuje, tak *l'(i)=0.*

l'(i) je dĺžka prekrytia medzi neposunutým a posunutým vzorom.

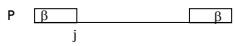


89

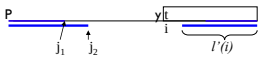
Boyerov-Moorov algoritmus

l'(i) sa rovná najväčšiemu $j \leq |P[i,\ldots n]|$ takému, že $N_i=j$

1. $N_i=j$ tak β je predponou P a tiež príponou P



2. a chceme najväčšie j



90

Boyerov-Moorov algoritmus

l'(i) sa rovná najväčšiemu *j*≤|*P[i,…n]*| takému, že N_i=j

	1	2	3	4	5	6	7	8	9	0
P:	a	b	d	a	b	a	b	d	a	b
N_j:	0	2	0	0	5	0	2	0	0	0
l'(i):	5	5	5	5	5	5	2	2	2	0

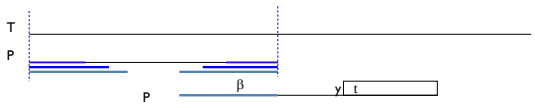
Boyerov-Moorov algoritmus

Ako vypočítať *l'(i)* z N_i v lineárnom čase?

Boyerov-Moorov algoritmus

Čo ak sa nájde zhoda?

posuň P o 1 miesto…ale…

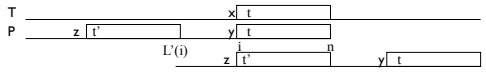


Posuň P o najmenší počet miest taký, že predpona posúvaného vzoru sa zhoduje s t, t.j. posuň P doprava o n-*l'*(2)

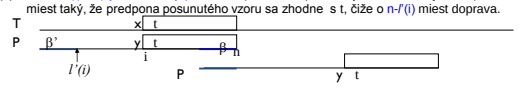
Boyerov-Moorov algoritmus

Pravidlo dobrej prípony: keď nastane nezhoda na mieste i-1 vzoru P

(1) ak L'(i)>0 (t.j. t' existuje), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.



(2) Inak ak L'(i)=0 (t.j. t' neexistuje)? Možno posunúť P poza ľavý koniec t o najmenší počet miest taký, že predpona posunutého vzoru sa zhodne s t, čiže o n-*l'*(i) miest doprava.



(3)Ak sa nájde nezhoda, tak posuň P doprava o n-*l'*(2)

Boyerov-Moorov algoritmus

rozšírené pravidlo zlého znaku vs. pravidlo dobrej prípony

```
    123456789012345678
T:  prstabstubabvqxrst
P:    qcabd   dab
        P:    qcabdabdab
  P:    qcabdabdab
```

```
    123456789012345678
T:  prstabstuqabvqxrst
P:    qcabd   dab
        P:    qcabdabdab
          P:    qcabdabdab
```

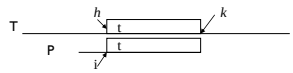

Boyerov-Moorov algoritmus

Posuň P o najväčší počet miest určený niektorým z oboch pravidiel. To je podstata Boyerovho-Moorovho algoritmu!

vstup: text T, vzor P; výstup: nájdi výskyty vzoru P v T

Algoritmus **Boyer-Moore**

```
vypočítaj L'(i), L`(i) a R(x)
k=n;
while (k≤m) do
    i=n
    h=k
    while i>0 and P[i]=T[h] do
        i--;
        h--;
    if i=0
        oznám výskyt vzoru P v T končiaci na mieste k;
        k=k+n-l`(2)
    else posuň P (zvýš k) o väčší počet miest z počtu určeného rozšíreným
pravidlom zlého znaku a počtu určeného pravidlom dobrej prípony.
```



Boyerov-Moorov algoritmus

- výkonnosť závisí od dĺžky vzoru
- O(n/m)
- Dlhšie vzory = lepšia výkonnosť
- Najmenší vzor = m = 1
  O(n) – lineárne hľadanie

97

porovnávanie pomocou automatu

- Na základe vzoru sa vytvorí minimálny deterministický konečný automat, pomocou ktorého sa rozpoznáva vzor v zadanom reťazci.

98

porovnávanie pomocou automatu

Def.: Konečný automat (finite automaton) je usporiadaná 5-tica
$(Q, q_0, A, \Sigma, \delta)$, kde
- Q je konečná množina stavov
- q_0 počiatočný stav
- A množina koncových stavov (akceptujúce)
- Σ je vstupná abeceda
- δ je tzv. prechodová funkcia z $Q \times \Sigma$ do Q.

Rozšírenie δ funkcie - $\delta^* : Q \times \Sigma^* \rightarrow Q$ je definované induktívne:
$\delta^*(q, \varrho) = q$
$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$

99

porovnávanie pomocou automatu

funkcia koncového stavu - vracia stav automatu po spracovaní nejakého slova

príponová funkcia pre P, |P| = m je
$\sigma : \Sigma^* \rightarrow \{0, 1, \ldots, m\}$
definovaná ako
$\sigma(x) = \max\{k : P_k \sqsupset x\}$,
kde $u \sqsupset v$ znamená, že u je príponou v a $P_k = P[1..k]$.

100

porovnávanie pomocou automatu

Definícia automatu: pre P, |P| = m
$Q = \{0, 1, \ldots, m\}$, $q_0 = 0$, A = {m}, $\delta(q, a) = \sigma(P_q a)$.
Vždy, keď sa počas simulácie vstupného slova T na automate dostane automat do stavu m, našiel sa podvýraz P a jeho posun je daný aktuálnym miestom v reťazci zmenšeným o m.
Platí:
- Pre každý reťazec x a znak a platí: $\sigma(xa) \leq \sigma(x) + 1$.
- Pre každý reťazec x a znak a, ak $q = \sigma(x)$, tak $\sigma(xa) = \sigma(P_q a)$

101

porovnávanie pomocou automatu

```
POROVNANIE KONEČNÝM AUTOMATOM(T, δ, m)
n ← dĺžka(T)
q ← 0
for i ← 1 to n
      do q ← δ(q, T[i])
            if q = m
            then
              print "Vzor sa v retazci vyskytuje s posunom " i-m
```

102

porovnávanie pomocou automatu

- VÝPOČET PRECHODOVEJ FUNKCIE (P, Σ)
  m := |P|
  for q := 0 to m do
  for každý symbol a ∈ Σ do
  k := min(m+1, q+2)
  repeat k := k − 1
  until $P_k \supset P_q a$
  δ(q, a) := k
  return δ
- Zložitosť algoritmu:
  - Preprocesná fáza: ***O***(*m* |*Σ*|)
  - Fáza vyhľadávania: ***O***(*n)*

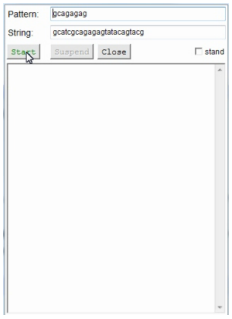
103

porovnávanie pomocou automatu

- Zložitosť algoritmu:
  - fáza predspracovania: ***O***(*m.*|*Σ*|)
  - fáza vyhľadávania: ***O***(*n)*

104

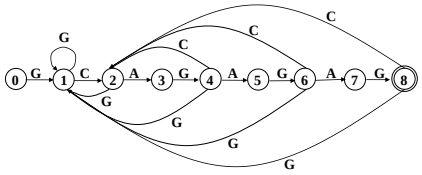
porovnávanie pomocou automatu



105

porovnávanie pomocou automatu

- DFA z príkladu:



106

Michael O. Rabin

- 1931 Breslau (Nemecko) dnes Wroclaw (Poľsko)
- 1953, M.Sc. Hebrew University of Jerusalem
- 1956, Ph.D. Princeton University
- práce:
  - výpočtová zložitosť,
  - matematická logika
  - Karp, Richard M.; Rabin, Michael O. (March 1987). Efficient randomized pattern-matching algorithms

107

Richard M. Karp

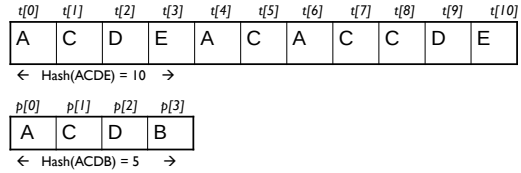
- 3.1.1935, Boston –
- Bc 1955, Harvard
- Master 1956, Harvard
- PhD 1959 aplikovaná matematika, Harvard
- International Computer Science Institute in Berkeley
- práce:
  - Karp, Richard M.; Rabin, Michael O. (March 1987). Efficient randomized pattern-matching algorithms

108

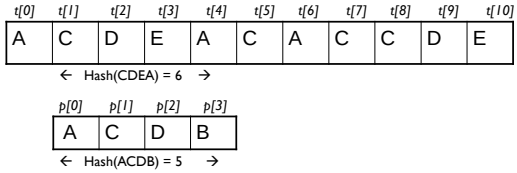
Rabinov-Karpov algoritmus

- Namiesto priameho porovnávania reťazca so vzorom sa porovnavajú výstupy hešovacích funkcií. Porovnáva sa heš vzoru s hešom vybraného podreťazca (vyberá sa podreťazec taký dlhý ako je dĺžka vzoru). Pokiaľ sa heše zhodujú, uskutočňuje sa porovnávanie jednotlivých znakov.

Rabinov-Karpov algoritmus – príklad 1/1



Rabinov-Karpov algoritmus – príklad 1/2

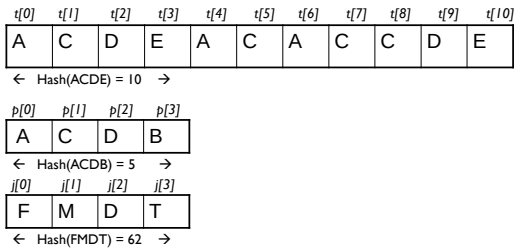


Rabinov-Karpov algoritmus

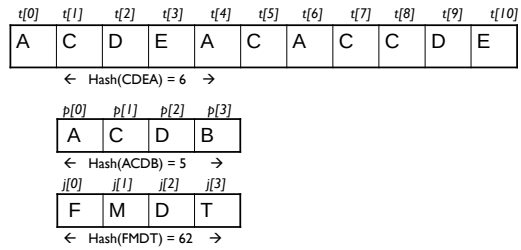
```
algorithm RabinKarp:
Input   pole znakov T, dĺžka n
        pole znakov P, dĺžka m

  hP := hash(P[1..m])
  hT := hash(T[1..m])
  for i from 1 to n-m+1
    if hT = hP
      if T[i..i+m-1] = P
        return i
    hT := hash(T[i+1..i+m])
  return nenašiel sa výskyt
```


Rabinov-Karpov algoritmus – príklad 2/1



Rabinov-Karpov algoritmus – príklad 2/2



Rabinov-Karpov algoritmus

- Hešovacia funkcia by mala mať tieto vlastnosti:
  - jednoducho vypočítateľná
  - s malou pravdepodobnosťou kolízií
  - Heš posunutého podreťazca by mal byť jednoducho odvoditeľný z predchádzajúceho hešu (táto vlastnosť výrazne uľahčí výpočet a algoritmus sa tým stáva omnoho efektívnejší ako naivný)

Rabinov-Karpov algoritmus

- Hešovacia funkcia:

Predpokladáme, že nahradíme reťazec M znakov určitým celým číslom. Ak použijeme konštantu b - maximálny počet možných znakov v abecede, tak definujeme:

- $x = t[i]b^M + t[i+1]b^{M-1}+...+t[i+M]$

Pokročme v texte o jeden znak dopredu a hodnota x' bude:

- $x' = t[i+1]b^M + t[i+2]b^{M-1}+...+t[i+M+1]$

ak preskúmame x a x' , tak zistíme, že:

- $x' = (x -t[i]b^M)b + t[i+M+1]$

Rabinov-Karpov algoritmus

Hešovacia funkcia:

- Tretej požiadavke vyhovuje napríklad hešovacia funkcia definovaná ako polynóm (m-1). stupňa, kde hodnoty znakov vystupujú ako koeficienty. Aby sme sa vyhli problémom s príliš veľkými číslami pri výpočtoch, používa sa modulo aritmetika:
- $pathash = (f^{m-1} \text{ord}(pat_0) + f^{m-2} \text{ord}(pat_1) + ... + f \text{ ord}(pat_{m-2}) + \text{ord}(pat_{m-1})) \bmod p$
- $texthash_i = (f^{m-1} \text{ord}(text_i) + f^{m-2} \text{ord}(text_{i+1}) + ... + f \text{ ord}(text_{i+m-2}) + \text{ord}(text_{i+m-1})) \bmod p$
- $texthash_{i+1} = (f^{m-1} \text{ord}(text_{i+1}) + f^{m-2} \text{ord}(text_{i+2}) + ... + f \text{ ord}(text_{i+m-1}) + \text{ord}(text_{i+m})) \bmod p$

$= \mathbf{(f (texthash_i - f^{m-1} ord(text_i)) + ord(text_{i+m})) \bmod p}$

- Hešovaciu funkciu ovplyvňujú parametre f a p.

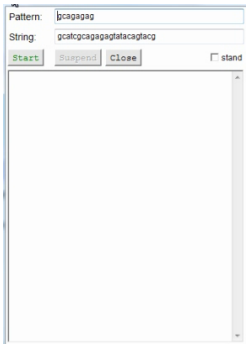
Rabinov-Karpov algoritmus

```
RABIN-KARP POROVNANIE(T, P, d, q)
n ← dĺžka(T)
m ← dĺžka(P)
h ← d^(m-1) mod q
p ← 0
t0 ← 0
for i ← 1 to m          //spracovanie
        do p ← (dp + P[i]) mod q
           t0 ← (dt0 + T[i]) mod q
for s ← 0 to n - m      //párovanie
        do if p = ts
                then if P[1..m]=T[s+1..s+m]
                        then print "Vzor sa v retazci vyskytuje s posunom" s
           if s < n-m
                then ts+1 ← (d(ts-T[s+1]h)+T[s+m+1]) mod q
```


Rabinov-Karpov algoritmus

- Zložitosť algoritmu:
  - fáza predspracovania
    má časovú zložitosť $\boldsymbol{O}(m)$
  - vyhľadavacia fáza
    má časovú zložitosť $\boldsymbol{O}(mn)$
  - Očakávaná doba behu algoritmu
    je $\boldsymbol{O}(n+m)$

Rabinov-Karpov algoritmus



Rabinov-Karpov algoritmus

- C++:

```
#define REHASH(a, b, h) (((h) - (a)*d) << 1) + (b))
void KR(char *x, int m, char *y, int n)
{
        int d, hx, hy, i, j;
        /* Preprocessing */
        /* computes d = 2^(m-1) with the left-shift operator */
        for (d = i = 1; i < m; ++i)
                d = (d<<1);
        for (hy = hx = i = 0; i < m; ++i)
        {
                hx = ((hx<<1) + x[i]);
                hy = ((hy<<1) + y[i]);
        }
        /* Searching */
        j = 0;
        while (j <= n-m)
        {
                if (hx == hy && memcmp(x, y + j, m) == 0)
                                OUTPUT(j);
                hy = REHASH(y[j], y[j + m], hy);
                ++j;
        }
}
```


časová efektívnosť

- Jeden vzor
  - BM O(n/m)
  - KMP O(n)
  - RK O(mn)
- Viac vzorov
  - BM, KMP O(n.k)
  - RK O(n + k)

aplikácie

- BM - textové editory – search/replace
- Karp-Rabin – vyhľadávanie plagiátov

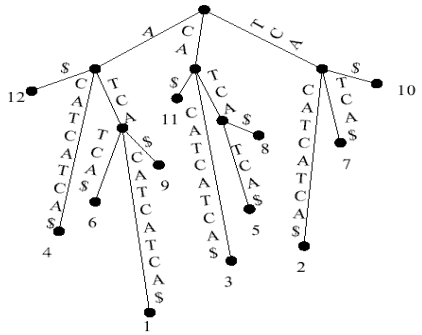
prípony

- Prípony reťazca *T*="ATCACATCATCA"

ATCACATCATCA	$T_{(1)}$
TCACATCATCA	$T_{(2)}$
CACATCATCA	$T_{(3)}$
ACATCATCA	$T_{(4)}$
CATCATCA	$T_{(5)}$
ATCATCA	$T_{(6)}$
TCATCA	$T_{(7)}$
CATCA	$T_{(8)}$
ATCA	$T_{(9)}$
TCA	$T_{(10)}$
CA	$T_{(11)}$
A	$T_{(12)}$

príponový strom

- príponový strom reťazca T="ATCACATCATCA"



vlastnosti príponového stromu

- každá hrana je ohodnotená nejakým podreťazcom reťazca *T*.
- každý vnútorný vrchol na najmenej dvoch potomkov.
- každá prípona $T_{(i)}$ má svoju ohodnotenú cestu z koreňa do listu pre $1 \leq i \leq n$.
- príponový strom má n listov.
- hrany vychádzajúce z toho istého vrchola majú ohodnotenia, ktoré sa určite nezačínajú tým istým znakom.

algoritmus vytvorenia príponového stromu

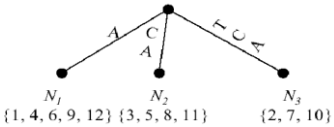
krok 1: rozdeľ všetky prípony do skupín podľa prvého znaku a vytvor vrchol.

krok 2: pre každú skupinu: ak obsahuje len jednu príponu, tak vytvor listový vrchol a hranu ohodnotenú touto príponou, inak nájdi najdlhšiu spoločnú predponu medzi príponami v tejto skupine a vytvor hranu vychádzajúcu z vrchola ohodnotenú najdlhšou spoločnou predponou. Vymaž túto predponu zo všetkých prípon v tejto skupine.

krok 3: opakuj predchádzajúci postup pre každý vrchol, ktorý nie je ukončený.

127

príklad vytvorenia príponového stromu



- T="ATCACATCATCA".
- začiatočné znaky: "A", "C", "T"
- v N_3,

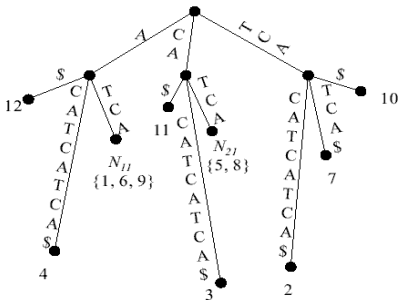
  $T(2)$ ="TCACATCATCA"

  $T(7)$ ="TCATCA"

  $T(10)$ ="TCA"
- najdlhšia spoločná predpona N_3 je "TCA"

128

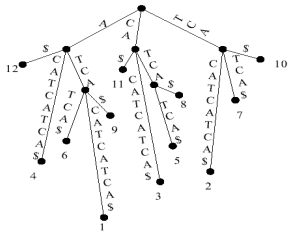
- T="ATCACATCATCA".
- druhá rekurzia:



129

nájdenie podreťazca pomocou príponového stromu

- T = "ATCACATCATCA"
- P ="TCAT"
  - P je na mieste 7 v T.
- P ="TCA"
  - P je na mieste 2, 7 a 10 v T.
- P ="TCATT"
  - P sa nenachádza v T.



130

príponový strom – časová zložitosť

- Príponový strom pre textový reťazec T dĺžky n sa dá zostrojiť v čase O(n) time (pomocou zložitého algoritmu).
- Vyhľadanie vzoru P dĺžky m v príponovom strome vyžaduje O(m) porovnaní.
- Hľadanie presného výskytu reťazca: O(n+m) time

131

Príponové pole

- V príponovom poli sú všetky prípony reťazca T v neklesajúcom lexikálnom poradí.
- napr. T="ATCACATCATCA"

i	1	2	3	4	5	6	7	8	9	10	11	12
A	12	4	9	1	6	11	3	8	5	10	2	7

4	ATCACATCATCA	$T_{(1)}$
11	TCACATCATCA	$T_{(2)}$
7	CACATCATCA	$T_{(3)}$
2	ACATCATCA	$T_{(4)}$
9	CATCATCA	$T_{(5)}$
5	ATCATCA	$T_{(6)}$
12	TCATCA	$T_{(7)}$
8	CATCA	$T_{(8)}$
3	ATCA	$T_{(9)}$
10	TCA	$T_{(10)}$
6	CA	$T_{(11)}$
1	A	$T_{(12)}$

1	A	$T_{(12)}$
2	ACATCATCA	$T_{(4)}$
3	ATCA	$T_{(9)}$
4	ATCACATCATCA	$T_{(1)}$
5	ATCATCA	$T_{(6)}$
6	CA	$T_{(11)}$
7	CACATCATCA	$T_{(3)}$
8	CATCA	$T_{(8)}$
9	CATCATCA	$T_{(5)}$
10	TCA	$T_{(10)}$
11	TCACATCATCA	$T_{(2)}$
12	TCATCA	$T_{(7)}$

132

Hľadanie v príponovom poli

- ak *T* sa reprezentuje príponovým poľom, vzor *P* sa dá nájsť v *T* v čase O(*m*.log*n*) binárnym vyhľadávaním.
- príponové pole sa dá určiť v čase O(*n*) lexikálnym hľadaním do hĺbky v príponovom poli.
- celkový čas: O(*n*+*m*.log*n*)

133

hľadanie približného výskytu reťazca

- Textový reťazec *T*, |*T*|=*n*
  vzor (reťazec) *P*, |*P*|=*m*
  k chýb, kde chybami môžu byť náhrada, vymazanie alebo vloženie znaku.
- napr:
  T =“pttapa”, *P* =“patt”, *k* =2,
  $T_{1,2}$,$T_{1,3}$,$T_{1,4}$ a $T_{5,6}$ sú všetko reťazce vzdialené nie viac než 2 chyby od vzoru *P*.

134

vzdialenosť editovania prípony

- Nech sú dané 2 reťazce S_1 a S_2, vzdialenosť editovania prípony je najmenší počet náhrad, vložení a výmazov, ktoré prepíšu nejakú príponu S_1 do S_2.
- napr:
  - S_1=“ptt” a S_2=“pt”. Vzdialenosť editovania prípony medzi S_1 a S_2 je 1.
  - S_1=“ptt” a S_2=“p”. Vzdialenosť editovania prípony medzi S_1 a S_2 je 2.
  - S_1=“pt” a S_2=“patt”. Vzdialenosť editovania prípony medzi S_1 a S_2 je 2.

135

vzdialenosť editovania prípony

- Nech *T* a *P* sú reťazce, ak aspoň jedna zo vzdialeností editovania prípony medzi $T_{1,1}$, $T_{1,2}$, …, $T_{1,n}$ a *P* nie je väčšia než *k*, tak P sa približne vyskytuje v T s chybou nie väčšou než *k*.
- napr: *T* =“pttapa”, *P* =“patt”, *k*=2
  - pre $T_{1,1}$=“p” a *P* =“patt”, vzdialenosť editovania prípony je 3.
  - pre $T_{1,2}$ =“pt” a *P* =“patt”, vzdialenosť editovania prípony je 2.
  - pre $T_{1,5}$ =“pttap” a *P* =“patt”, vzdialenosť editovania prípony je 3.
  - pre $T_{1,6}$ =“pttapa” a *P* =“patt”, vzdialenosť editovania prípony je 4.

136

hľadanie približného výskytu reťazca

- riešenie metódami dynamického programovania
  - nech $E(i,j)$ označuje najmenšiu vzdialenosť editovania prípony medzi $T_{1,j}$ a $P_{1,i}$,
  - nech m = |P| označuje dĺžku vzoru,
  - potom $E(m,j)$ nám pomôže vyriešiť:
- úloha nájsť „najpribližnejší“ (t.j. s čo najmenším počtom chýb k) výskyt vzoru P v texte T:
  - vezmi podreťazec textu T, pre ktorý je $E(m,j)$ najmenšie.
- úloha nájsť približný výskyt vzoru P v texte T s najviac *k* chybami:
  - vezmi ľubovoľný podreťazec textu T, pre ktorý je $E(m,j)$ <= k.

137

najmenšia vzdialenosť editovania prípony

- nech $E(i,j)$ označuje najmenšiu vzdialenosť editovania prípony medzi $T_{1,j}$ a $P_{1,i}$.
- pre všetky miesta *i* vo vzore P a všetky miesta *j* v texte T spočítame najmenšiu vzdialenosť editovania medzi prvými *i* znakmi vzoru P a ľubovoľným podreťazcom $T_{h,j}$ reťazca T, ktorý sa končí na mieste *j* (t.j. nejakou príponou $T_{1,j}$)
- to znamená prejsť všetky podreťazce T končiace v mieste j a určiť, ktorý z nich má najmenšiu vzdialenosť editovania prípony ku vzoru P. Jeho vzdialenosť je hodnota $E(i,j)$
- efektívne sa to dá počítať pomocou formuly (porovnaj s počítaním Levensteinovej vzdialenosti medzi reťazcami):

$$E(i,j) = E(i-1, j-1) \quad \text{if } P_i = T_j$$

$$E(i,j) = \min\{E(i,j-1), E(i-1,j), E(i-1,j-1)\}+1 \quad \text{if } P_i \neq T_j$$

138

najmenšia vzdialenosť editovania prípony

$E(i, j) = E(i-1, j-1)$ if $P_i = T_j$

$E(i, j) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

- výsledok bude matica n x m, n = |T|, m = |P|. Ako ju zrátať? Postupne, podľa formuly. Pomôže pridať 1 riadok s nulami.
- keďže vieme, že výsledok chceme použiť nielen na zistenie najmenšej vzdialenosti, ale aj na určenie približného podreťazca vyskytujúceho sa v texte T (ktorý je toľko vzdialený od vzoru P), je užitočné si zapamätať, ktorú z troch alternatív $E(i, j-1)$ alebo $E(i-1, j)$ alebo $E(i-1, j-1)$ sme v tom-ktorom kroku použili.

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0					
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1,1) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1				
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1, 2) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1				
P	2	a	2						
	3	t	3						
	4	t	4						

- alebo
- $E(1, 2) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1			
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1, 3) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1		
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1, 4) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1,5) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2						
	3	t	3						
	4	t	4						

- $E(1, 6) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2	1					
	3	t	3						
	4	t	4						

- $E(2, 1) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2	1	1				
	3	t	3						
	4	t	4						

- $E(2, 2) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2	1	1	2			
	3	t	3						
	4	t	4						

- $E(2, 3) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

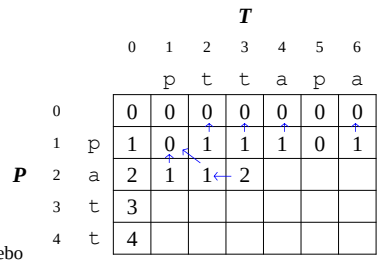
- napr: T ="pttapa", P ="patt"

			T						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2	1	1	2			
	3	t	3						
	4	t	4						

- alebo
- $E(2, 3) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

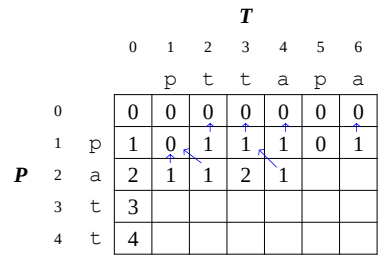
- napr: T ="pttapa", P ="patt"



- alebo
- $E(2, 3) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

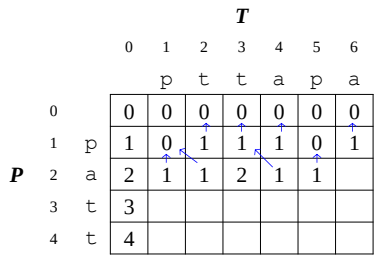
- napr: T ="pttapa", P ="patt"



- $E(2,4) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

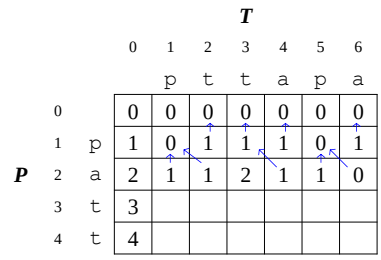
- napr: T ="pttapa", P ="patt"



- $E(2, 5) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

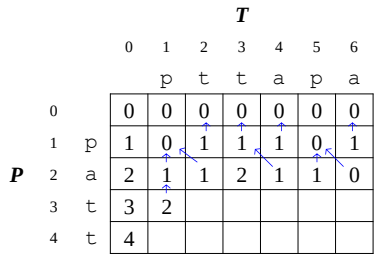
- napr: T ="pttapa", P ="patt"



- $E(2,6) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

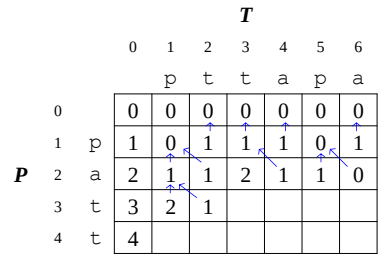
- napr: T ="pttapa", P ="patt"



- $E(3, 1) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"



- $E(3,2) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1			
4	t		4						

- $E(3,3) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1	2		
4	t		4						

- $E(3, 4) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1	2	2	
4	t		4						

- $E(3, 5) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1	2	2	1
4	t		4						

- $E(3, 6) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1	2	2	1
4	t		4	3					

- $E(4, 1) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
0			0	0	0	0	0	0	0
1	p		1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
3	t		3	2	1	1	2	2	1
4	t		4	3	2				

- $E(4,2) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	2	1			

P

- $E(4,3) = E(i-1, j-1)$ if $P_i = T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	2	1	2		

P

- $E(4, 4) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	2	1	2	3	

P

- $E(4, 5) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

počítanie matice E

- napr: T ="pttapa", P ="patt"

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	2	1	2	3	2

P

- $E(4, 6) = \min\{E(i, j-1), E(i-1, j), E(i-1, j-1)\}+1$ if $P_i \neq T_j$

hľadanie najpribližnejšieho výskytu reťazca

- napr: T ="pttapa", P ="patt":
- pozrieme sa na posledný riadok. najmenšia vzdialenosť je v 3. stĺpci: E(4,3)=1. Z miesta (4,3) ideme naspäť podľa šípiek až do prvého riadku. Je to miesto (1,1). Stĺpec určuje začiatok podreťazca v T, ktorý je najpribližnejším výskytom vzoru P ~ $T_{1,3}$ = ptt. Jediná chyba sa „napraví" jedným vložením znaku a.

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	②	①	②	3	②

P

hľadanie približného výskytu reťazca s chybou najviac 2

- napr: T ="pttapa", P ="patt", k=2:
- pozrieme sa na posledný riadok. Hodnoty E(4,j) < = k sú v stĺpcoch 2, 3, 4 a 6. Zodpovedajú približným výskytom $T_{1,2}$ = pt, $T_{1,3}$ = ptt, $T_{1,4}$ = ptta, $T_{5,6}$ = pa.

T

		0	1	2	3	4	5	6
			p	t	t	a	p	a
0		0	0	0	0	0	0	0
1	p	1	0	1	1	1	0	1
2	a	2	1	1	2	1	1	0
3	t	3	2	1	1	2	2	1
4	t	4	3	②	①	②	3	②

P

Hľadanie najdlhšej spoločnej podpostupnosti

- Longest common subsequence (LCS) problém
  - Dané sú dve postupnosti x[1..m] a y[1..n]; máme nájsť najdlhšiu podpostupnosť, ktorá sa vyskytuje v oboch postupnostiach
  - Podpostupnosť: prvky v pôvodnej postupnosti nemusia byť nevyhnutne vedľa seba, ale ich poradie ostáva nezmenené
- x = {A B C B D A B}, y = {B D C A B A}
  - {B B A} je podpostupnosť oboch postupností **x** a **y**
- Algoritmus hrubej sily
  - Pre každú podpostupnosť v **x**, zisti či nie je podpostupnosťou **y**. Vráť najdlhšiu.
  - Koľko podpostupností je v **x**?
    - 2^m
  - Aká by bola časová náročnosť?
    - 2^m podpostupností **x** porovnať s ***n*** prvkami postupnosti **y**
    - $O(n\,2^m)$

Hľadanie najdlhšej spoločnej podpostupnosti

- Úloha: Porovnanie dvoch DNA reťazcov
- X = {A B C B D A B}, Y = {B D C A B A}
- Algoritmom hrubej sily porovnáme každú podpostupnosť X so znakmi v Y
  - X = A B C B D A B
  - Y = B D C A B A
- LCS problém má optimálnu subštruktúru: riešenie čiastkových problémov je časť konečného riešenia
- Čiastkový problém
  - Nájsť najdlhšiu spoločnú podpostupnosť párov prefixov X a Y
- Na vyriešenie tohto problému môžeme použiť dynamické programovanie!

Definícia dĺžky najdlhšej spoločnej podpostupnosti

- V prvom rade nájdeme dĺžku LCS. Neskôr zmodifikujeme algoritmus pre nájdenie LCS samotnej.
- Definujeme X_i a Y_j ako predpony X a Y dĺžky *i* respektíve *j*
- *Definujeme c[i,j] ako dĺžku LCS X_i a Y_j*
- *Potom dĺžka LCS X a Y bude c[m,n]*
- *Rekurzívna definícia c[i,j]*

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & ak\ x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & inak \end{cases}$$

počítanie dĺžky najdlhšej spoločnej podpostupnosti

- Začneme s *i=j=0* (prázdna podpostupnosť X a Y)
  - Pretože X_0 a Y_0 sú prázdne reťazce, ich LCS je vždy prázdna (c[0,0]=0)
- LCS prázdneho reťazca a hocijakého reťazca je pre každé i a j: c[0,j]=c[i,0]=0
- Pre výpočet c[i,j] sa rozhodujeme medzi dvoma prípadmi:
  - ***x[i] = y[j]***
    - Pri zhode symbolu v postupnostiach X a Y je dĺžka LCS X_i a Y_j rovnaká ako dĺžka LCS menšej postupnosti X_{i-1} a Y_{j-1}, plus 1
  - ***x[i] != y[j]***
    - Ak sa symboly nezhodujú dĺžka ostáva nezmenená *(max(c[i-1,j], c[i, j-1]))*

Algoritmus na nájdenie dĺžky najdlhšej spoločnej podpostupnosti

```
LCS DĹŽKA(X,Y)
m = length(X)
n = length(Y)
for i = 1 to m c[i,0] = 0        //X0
for i = 1 to m c[i,0] = 0        //Y0
for i = 1 to m        //pre všetky Xi
        for j = 1 to n        //pre všetky Yj
                if (Xi == Yj)
                        c[i,j] = c[i-1,j-1]+1
                else
                        c[i,j] = max(c[i-1,j],c[i,j-1])
return c
```

- Príklad: X = ABCB; Y = BDCAB
  - LCS(X, Y) = BCB
  - X = A B C B
  - Y = B D C A B

Najdlhšia spoločná podpostupnosť – príklad (inicializácia)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

ABCB X = ABCB; m = |X| = 4
BDCAB Y = BDCAB; n = |X| = 5
alokácia 2rozmerného poľa c[0..4, 0..5]

Najdlhšia spoločná podpostupnosť – príklad (1)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0					
2	B		0					
3	C		0					
4	B		0					

ABCB
BDCAB

for *i* = 1 **to** m *c[i,0]* = 0
for *j* = 1 **to** n *c[0,j]* = 0

175

Najdlhšia spoločná podpostupnosť – príklad (2)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0				
2	B		0					
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
  c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

176

Najdlhšia spoločná podpostupnosť – príklad (3)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0		
2	B		0					
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
  c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

177

Najdlhšia spoločná podpostupnosť – príklad (4)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	
2	B		0					
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
  c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

178

Najdlhšia spoločná podpostupnosť – príklad (5)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0					
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
  c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

179

Najdlhšia spoločná podpostupnosť – príklad (6)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
  c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

180

Najdlhšia spoločná podpostupnosť – príklad (7)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

181

Najdlhšia spoločná podpostupnosť – príklad (8)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

182

Najdlhšia spoločná podpostupnosť – príklad (9)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

183

Najdlhšia spoločná podpostupnosť – príklad (10)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

184

Najdlhšia spoločná podpostupnosť – príklad (11)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

185

Najdlhšia spoločná podpostupnosť – príklad (12)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

186

Najdlhšia spoločná podpostupnosť – príklad (13)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (13)

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

c[4,5] obsahuje dĺžku najdlhšej spoločnej podpostupnosti

Analýza algoritmu pre nájdenie najdlhšej spoločnej podpostupnosti

- LCS algoritmus vypočíta hodnoty každého vstupu poľa *c[m,n]. Aký je teda výpočtový čas?*
- *O(m*n)*
  - Každá hodnota *c[i,j]* je spočítaná v konštantnom čase, a v poli máme *m*n* prvkov
- Zatiaľ sme našli len dĺžku najdlhšej spoločnej podpostupnosti.
- Ďalej je potrebné nájsť najdlhšiu spoločnú podpostupnosť.
- Musíme modifikovať algoritmus aby nám dával výstup najdlhšej spoločnej podpostupnosti postupností X a Y
  - V poli *c[i,j]* máme všetko zaznamenané
  - Každá hodnota *c[i,j]* závisí na *c[i-1,j]* alebo *c[i,j-1]*
  - Pre každú hodnotu *c[i,j]* vieme určiť ako sme ju dosiahli

Hľadanie najdlhšej spoločnej podpostupnosti

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & ak\ x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & inak \end{cases}$$

2	2
2	3

V tomto prípade
c[i,j] = c[i-1,j-1] + 1 = 2 + 1 = 3

1	1
2	2

V tomto prípade
c[i,j] = max(c[i,j-1], c[i-1,j]) = max(2, 1) = 2

- Môžeme začať z c[m,n] a ísť späť
- Ak *c[i,j] = c[i-1,j-1]* + 1, zapamätáme si *x[i]*
  - *x[i]* je časť z najdlhšej spoločnej podpostupnosti
- Ak *i* = 0 alebo *j* = 0 (dosiahneme začiatok), výstupom sú písmená odpamätané v X, usporiadané v opačnom poradí

Hľadanie najdlhšej spoločnej podpostupnosti

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

Hľadanie najdlhšej spoločnej podpostupnosti

i		j	0	1	2	3	4	5
			Y_j	B	D	C	A	B
0	X_i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

Najdlhšia spoločná postupnosť (odzadu): **B C B**
Najdlhšia spoločná postupnosť: **B C B**