Vyhledávání řetězců v textu

Rabinův-Karpův algoritmus

Konečné automaty

Knuttův-Morrisův-Prattův algoritmus

Shrnutí

Tomáš Oberhuber

Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague

Video na Youtube

Vyhledávání řetězců v textu

Budeme řešit úlohu vyhledání zadaného řetězce v textu, tj. stejnou operaci jako je "hledat/find" v textových editorech.

- máme text délky n - $T[1 \dots n]$
- hledáme v něm řetězec délky m - $P[1 \dots m]$

Definition 1

Říkáme, že P se vyskytuje v T na pozici $s+1$ pro $0 \leq s \leq n-m$ pokud

$$T[s+1 \dots s+m] = P[1 \dots m],$$

tj. $T[s+j] = P[j]$ pro $j = 1, \dots, m$.

Vyhledávání řetězců v textu

Definition 2

Pokud se P vyskytuje v T na pozici $s + 1$, říkáme, že s je přípustný posuv (shift). Jinak jde o nepřípustný posuv.

Definition 3

Úloha vyhledávání v textu je úloha nalezení všech přípustných posuvů.

Vyhledávání řetězců v textu

Definition 4

Buď Σ abeceda, tj. množina všech přípustných znaků. Pak:

- Σ^* značí množinu všech konečných řetězců v abecedě Σ.
- ϵ značí prázdný řetězec a platí, $\epsilon \in \Sigma^*$.
- Délku řetězce $x \in \Sigma^*$ značíme $|x|$.
- Navázání dvou řetězců $x, y \in \Sigma^*$ značíme xy a platí $|xy| = |x| + |y|$.
- Je-li řetězec $w \in \Sigma^*$ prefixem (předponou) řetězce $x \in \Sigma^*$, tj. $\exists y \in \Sigma^*$ takové, že $x = wy$, pak budeme psát $w \sqsubset x$.
- Je-li řetězec $w \in \Sigma^*$ sufixem (příponou) řetězce $x \in \Sigma^*$, tj. $\exists y \in \Sigma^*$ takové, že $x = yw$, pak budeme psát $w \sqsupset x$.

Vyhledávání řetězců v textu

Remark 5

Budeme používat značení $P_k = P[1 \dots k]$, *tj.:*

- $|P_k| = k$,
- $P_k \sqsubset P$,
- $P_0 = \epsilon$,
- $P_m = P$.

Naši úlohu pak lze také přeformulovat tak, že hledáme všechna s taková, že $P \sqsupset T_{s+m}$.

Naivní algoritmus

```
1: procedure NAIVEFIND( T, n, P, m )
2:     for s=0,..., n-m do
3:         if P[1,...,m] = T[s+1,...,s+m] then
4:             print "Pattern occurs at position " s+1
5:         end if
6:     end for
7: end procedure
```

- složitost je $O((n-m+1)m)$
- nejhorší případ nastává v situaci

$$T = a^n \text{ a } P = a^m$$

- pro $m = \lfloor n/2 \rfloor$ je pak složitost $O(n^2)$

Rabinův-Karpův algoritmus

- algoritmus je založen na celočíselném přepisu textu a operaci modulo
- pro jednoduchost budeme předpokládat

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 9\}$$

- i pro obecné Σ pak platí, že každý řetězec $x \in \Sigma^*$ lze zapsat jako číslo v soustavě o základu $d = |\Sigma|$
- použijeme značení
  - $P[1 \ldots m] \rightarrow p$ je číslo reprezentující vzor P
  - $T[s+1 \ldots s+m] \rightarrow t_s$ je číslo reprezentující sufix T_{s+m} délky m

Rabinův-Karpův algoritmus

- snadno vidíme, že

$$T[s+1 \ldots s+m] = P[1 \ldots m] \Leftrightarrow t_s = p$$

- pokud dokážeme napočítat p se složitostí $O(m)$ a všechna t_s se složitostí $O(n-m+1)$, pak celková složitost bude

$$O(m) + O(n-m+1) = O(n)$$

- pro začátek neřešme, zda se p a t_s vejde do počítačové aritmetiky potažmo registru CPU

Rabinův-Karpův algoritmus

- p lze spočítat pomocí Hornerova schématu se složitostí $O(m)$

$$p = P[m]+10(P[m-1]+10(P[m-2]+\ldots 10(P[2]+10P[1]))\ldots))$$

- pokud známe t_s, pak platí

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Example 6

- $T = 314152$, $n = 6$, $m = 5$

$$t_0 = 31415,$$
$$t_1 = 10(31415 - 10^4 \cdot 3) + 2 = 10(31415 - 30000) + 2 = 14152$$

- konstantu 10^{m-1} lze předpočítat
- pak lze t_{s+1} získat ze znalosti t_s se složitostí $O(1)$
- složitost výpočtu všech t_s je pak skutečně $O(n-m+1)$

Rabinův-Karpův algoritmus

- co ale dělat, když se p nebo t_s nevejde do počítačové aritmetiky?
- využijeme faktu

$$t_s = p \Rightarrow (t_s \bmod q) = (p \bmod q)$$

- neboli

$$(t_s \bmod q) \neq (p \bmod q) \Rightarrow t_s \neq p$$

Rabinův-Karpův algoritmus

- pokud tedy platí

$$(t_s \bmod q) \neq (p \bmod q)$$

  pak víme, že $t_s \neq p$, jinak musíme provést porovnání $T[s+1 \ldots s+m]$ a P
- operace $\bmod q$ tedy dělí všechny řetězce do q různých tříd
- porovnáváme pouze řetězce, které padnou do stejné třídy
- q proto volíme co největší možné
- větší počet menších tříd snižuje pravděpodobnost, že dva řetězce padnou do stejné třídy

Rabinův-Karpův algoritmus

- pro obecnou abecedu Σ pro $d = |\Sigma|$ pak máme
- použijeme značení $t_s^q = t_s \bmod q$

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

$$t_{s+1}^q = \left[d \left(\underbrace{t_s \bmod q}_{=t_s^q} - \left(\underbrace{\left(d^{m-1} \bmod q\right)}_{=h} T[s+1] \right) \bmod q \right) \bmod q - T[s+m+1] \right] \bmod q,$$

$$t_{s+1}^q = \left[\underbrace{d \left(\underbrace{t_s^q}_{\leq q} - \underbrace{hT[s+1]}_{\leq dq} \bmod q \right)}_{\leq dq} \bmod q + T[s+m+1] \right] \bmod q$$

- celkem tedy požadujeme, aby se dq vešlo do počítačové aritmetiky
- podle toho pak volíme q co největší možné

Rabinův-Karpův algoritmus

Algoritmus pak vypadá takto:

```
1:  procedure RabinKarpFind( T, n, P, m )
2:      h = d^{m-1} mod q
3:      p = 0
4:      t = 0
5:      for i = 1, ..., m do
6:          p = (dp + P[i]) mod q
7:          t = (dt + T[i]) mod q
8:      end for
9:      for s = 0, ..., n - m do
10:         if p = t then
11:             if P[1...m] = T[s+1...s+m] then
12:                 print "Pattern occurs at position " s+1
13:             end if
14:         end if
15:         if s < n - m then
16:             t = (t - (hT[s+1] mod q)) mod q
17:             t = ((dt mod q) + T[s+m+1]) mod q
18:         end if
19:     end for
20: end procedure
```

Rabinův-Karpův algoritmus

- nejhorší možný případ nastane např. opět v situaci $T = a^n$ a $P = a^m$
- algoritmus pak musí provádět porovnání pro každé s a k tomu dělá ještě vlastně zbytečné výpočty navíc oproti naivní implementaci
- jinak je složitost

$$\underbrace{O(n)}_{p \neq t} + \underbrace{O\left(m\left(\nu + \frac{n}{q}\right)\right)}_{p = t},$$

kde

- ν je počet výskytů P v T
- $\frac{n}{q}$ po odhad falešných shod kdy $p = t$ a $P[1 \ldots m] \neq T[s+1 \ldots s+m]$
- $\frac{1}{q}$ je pravděpodobnost, že t padne do stejné třídy jako p

Konečné automaty

- ukážeme si, jak k danému vzoru P sestrojit konečný automat M pro efektivní vyhledání vzoru P v textu T

Definition 7

Konečný automat M je pětice:

1. Q - konečná množina stavů
2. $q_0 \in Q$ - počáteční stav
3. $A \subset Q$ - množina akceptovaných stavů
4. Σ - vstupní abeceda
5. $\delta : Q \times \Sigma \rightarrow Q$ - přechodová funkce

Konečné automaty

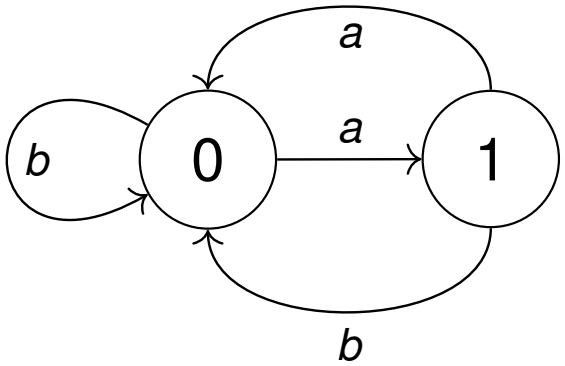
- automat začíná ve stavu q_0
- načte první písmeno $T[1]$
- přechodová funkce δ podle $T[1]$ určí, do kterého stavu se automat přepne
- dále se načte $T[2]$ a δ opět podle současného stavu určí nový stav
- dostane-li se automat do nějakého stavu $q \in A$, znamená to, že akceptuje daný vstup

Konečné automaty

Example 8

- $Q = \{0, 1\}$, $q_0 = 0$, $A = \{1\}$
- $\delta \equiv$

	a	b
0	1	0
1	0	0



- takový automat akceptuje řetězce končící lichým počtem 'a'
-

T	-	a	b	a	a	a
q	0	1	0	1	0	1

-

T	-	a	b	a	a
q	0	1	0	1	0

Konečné automaty

Jak sestrojit automat, který bude akceptovat právě zadaný vzor P?

- automat má $m + 1$ stavů
- číslo stavu říká, v kolika znacích se shoduje P s posledně načtenými znaky T
- jinými slovy platí, že prefix P délky q se shoduje se sufixem T_{s+q} délky q
- automat ale chceme konstruovat bez znalosti T
- jsme-li ale ve stavu q, pak známe posledních q znaků textu T, neboť ty se shodují s prvními q znaky P

T	C	G	A	T	G	A	T	G	A
P	-	-	A	T	G	A	T	C	G
q			1	2	3	4	5		
s							↑		

Konečné automaty

- definujme funkci $\sigma : \Sigma^* \to N$

$$\sigma(x) = \max_{k=0,1,\ldots,m} \{k | P_k \sqsupset x\}$$

- je to tedy délka nejdelšího prefixu P, který je zároveň sufixem x
- automat M definujeme takto

$$\begin{aligned} Q &= \{0, 1, \ldots, m\}, \\ q_0 &= 0, \\ A &= \{m\}, \\ \delta(q, a) &= \sigma(P_q a). \end{aligned}$$

Konečné automaty

- automat je ve stavu q a načítá znak $a = T[s+1]$
- chceme se dostat do stavu q, který odpovídá délce nejdelšího prefixu P, který je zároveň sufixem $T_s a$, tj. odpovídá posledním q znakům T_{s+1}
- takové q je dáno vztahem

$$q = \sigma(T_s a) = \sigma(P_q a)$$

- je-li $a = P[q+1]$, pak automat přejde do stavu $q+1$
- pokud $a \neq P[q+1]$, pak najdeme jiný nejlepší prefix P_q
- právě tím nezahodíme vše, co jsme dosud porovnali, ale "zrecyklujeme" maximální počet znaků

Konečné automaty

Example 9

- předpokládejem tento stav

T	C	G	A	T	G	A	T	G	A	T	C	G
P			A	T	G	A	T	C	G			
q			1	2	3	4	5					
s							↑					

- jelikož G≠C, shoda tu nenastane, ale můžeme využít prefix ATG

T	C	G	A	T	G	A	T	G	A	T	C	G
P						A	T	G	A	T	C	G
q						1	2	3	4			
s								↑				

Konečné automaty

Example 10

- P =ababaca
- $q_0 = 0, A = \{7\}$,
-

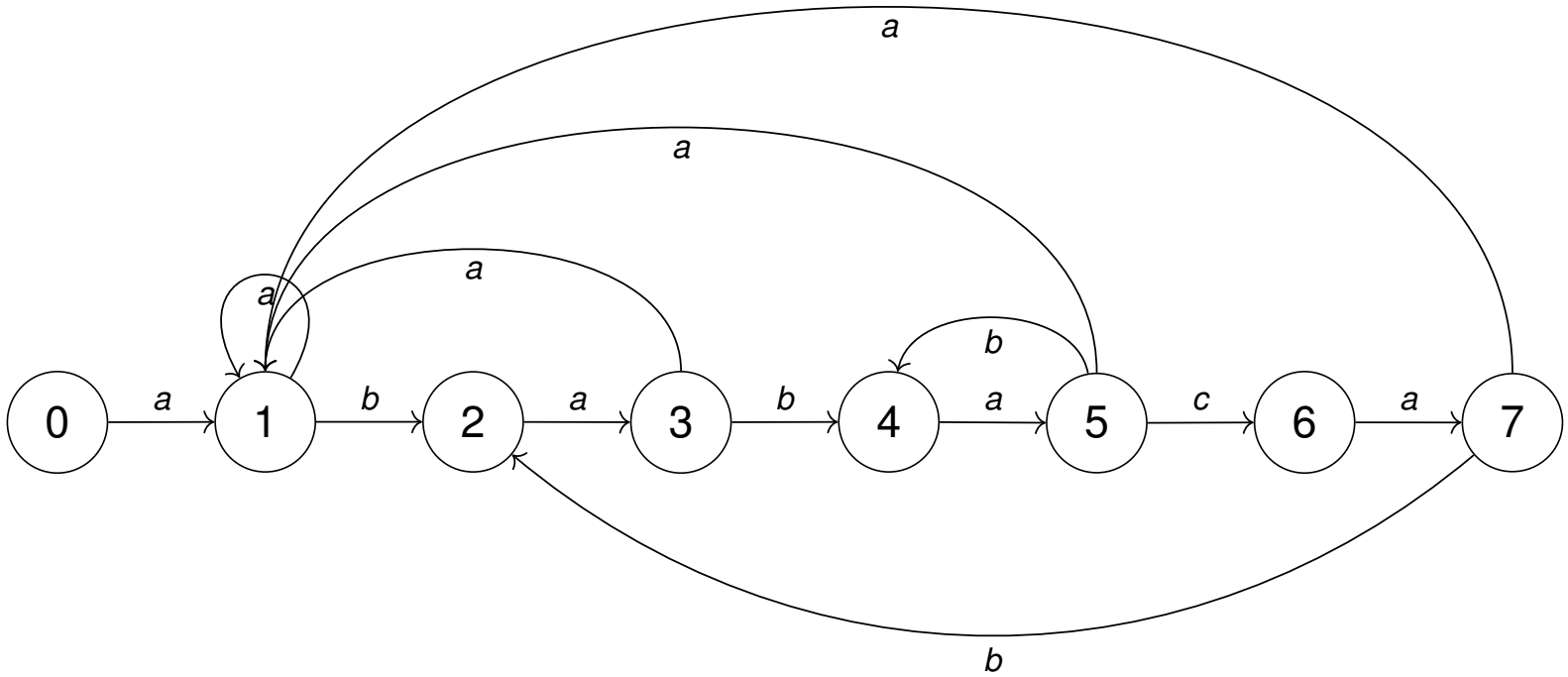
i	P_i	δ		
		a	b	c
0		1	0	0
1	a	1	2	0
2	ab	3	0	0
3	aba	1	4	0
4	abab	5	0	0
5	ababa	1	4	6
6	ababac	7	0	0
7	ababaca	1	2	0

Konečné automaty

- $\delta =$

	0	1	2	3	4	5	6	7
a	1	1	3	1	5	1	7	1
b	0	2	0	4	0	4	0	2
c	0	0	0	0	0	6	0	0

-



-

s	0	1	2	3	4	5	6	7	8	9	10	11
$T[s]$		a	b	a	b	a	b	a	c	a	b	a
q	0	1	2	3	4	5	4	5	6	**7**	2	3

Konečné automaty

Algoritmus pak vypadá takto:

```
1: procedure FiniteAutomatonFind( T, n, δ, m )
2:     q = 0
3:     for s = 1, ..., n do
4:         q = δ(q, T[s])
5:         if q = m then
6:             print "Pattern occurs at position " s
7:         end if
8:     end for
9: end procedure
```

Složitost je $O(n)$.

Konečné automaty

Výpočet δ funkce:

1: **procedure** FINITEAUTOMATONDELTA(P, m)
2: **for** $q = 0, \ldots, m$ **do**
3: **for** $a \in \Sigma$ **do**
4: $k = \min\{m, q+1\}$
5: **while** not $P_k \sqsupset P_q a$ **do**
6: $k = k - 1$
7: **end while**
8: $\delta(q, a) = k$
9: **end for**
10: **end for**
11: return δ
12: **end procedure**

Složitost je $O(m^3|\Sigma|)$.

KMP algoritmus

- autory jsou Knutt, Morris a Pratt, 1977
- mějme tuto situaci

T	b	a	c	b	a	b	a	b	a	b	c	b	a	b
					|	|	|	|	|	$\neq$			
P	$\underbrace{\quad\quad}_{s}$				a	b	a	b	a	c	a			
					$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad}_{q}$									

- chceme najít nejmenší posun d, aby $P_{q-d} \sqsupset T_{s+q} \equiv P_q$

T	b	a	c	b	a	b	a	b	a	b	c	b	a	b
							|	|	|	=	$\neq$			
P		$\underbrace{\quad\quad\quad\quad\quad\quad}_{s+2}$					a	b	a	b	a	a	c	a
							$\underbrace{\quad\quad\quad\quad\quad\quad}_{q-2}$							

KMP algoritmus

- tedy hledáme

$$\min \left\{ d | d = 1, \ldots, q-1 \wedge P_{q-d} \sqsupset P_q \right\},$$

nebo také

$$\max \left\{ k | k = 1, \ldots, q-1 \wedge P_k \sqsupset P_q \right\},$$

- definujeme funkci π

$$\pi(q) = \max \left\{ k | k = 1, \ldots, q-1 \wedge P_k \sqsupset P_q \right\}$$

- snadno vidíme, že $\pi(1) = 0$

KMP algoritmus

```
1:  procedure KMPFind( P, m )
2:      q = 0
3:      for i = 1, ..., n do
4:          while q > 0 and P[q + 1] ≠ T[i] do
5:              q = π(q)
6:          end while
7:          if P[q + 1] = T[i] then
8:              q = q + 1
9:          end if
10:         if  q = m then
11:             print "Pattern occurs at position " i
12:         end if
13:     end for
14: end procedure
```

- složitost je $O(n)$
  - q se v každé iteraci for cyklu (ř.3) zvyšuje maximálně o 1 (ř.8)
  - while cyklus (ř.4) je tedy úměrný hodnotě q nakumulované za poslední iterace

KMP algoritmus

Výpočet funkce $\pi = \max\{k | k = 1, \ldots, q-1 \wedge P_k \sqsupset P_q\}$:

Example 11

- P =ababaca

q	P_q	k	P_k	$\pi(q)$	q	P_q	k	P_k	$\pi(q)$
1	a	0	ϵ ✓	**0**	6	ababac	0	ϵ ✓	**0**
2	ab	0	ϵ ✓	**0**			1	a	
		1	a				2	ab	
3	aba	0	ϵ ✓	0			3	aba	
		1	a ✓	**1**			4	abab	
		2	ab				5	ababa	
4	abab	0	ϵ ✓		7	ababaca	0	ϵ ✓	0
		1	a				1	a ✓	**1**
		2	ab ✓	**2**			2	ab	
		3	aba				3	aba	
5	ababa	0	ϵ ✓				4	abab	
		1	a ✓	1			5	ababa	
		2	ab				6	ababac	
		3	aba ✓	**3**					
		4	abab						

KMP algoritmus

Algoritmus:

- máme nejdelší sufix P_k prefixu P_q
- hledáme co nejdelší sufix prefixu P_{q+1}
- mohu se pokusit rozšířit P_k o $P[q+1]$
- pokud to nejde, tj. $(P_k P[q+1])$ není sufix P_{q+1}, můžu zkusit stejně rozšířit kratší sufix prefixu P_q
- jeho délku mi udává $\pi(k)$

q	P_q												
5	a	b	a	b	a	c	a						$\pi(5) = 3$
3			a	b	a	b	a	c	a				$\pi(3) = 1$
1					a	b	a	b	a	c	a		

KMP algoritmus

```
1:  procedure KMPPrefixFunction( P, m )
2:      π(1) = 0
3:      k = 0
4:      for q = 2, ..., m do
5:          while k > 0 and P[k + 1] ≠ P[q] do
6:              k = π(k)
7:          end while
8:          if P[k + 1] = P[q] then
9:              k = k + 1
10:         end if
11:         π(q) = k
12:     end for
13:     return π
14: end procedure
```

- vidíme, že algoritmus je tedy téměř stejný jako vyhledávání samotné
- složitost je tedy stejná jako u vyhledávání, tj. $O(m)$
- celková složitost KMP algoritmu je $O(m + n)$

Shrnutí

- Rabinův-Karpův algoritmus ukazuje myšlenku kombinace vyhledávání a hešování, resp. rozklad vzorů do tříd
- vyhledávání pomocí automatů ukazuje konkrétní aplikaci konečných automatů
- vyhledávání pomocí automatů a KMP algoritmus jsou si podobné
- oba mají smysl zejména tehdy, když ve vyhledávaném vzoru existují podobnosti v formě sufixů prefixů
- taková šance roste zejména pro malé abecedy jako např. binární kód nebo DNA sekvence