Zápočtová práce z Algoritmů a Datových Struktur II (NTIN061)

Hledání v textu algoritmem Boyer–Moore

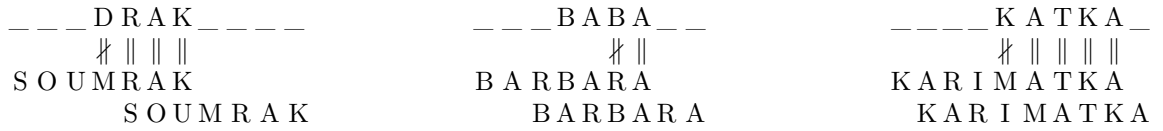
David Pěgřímek

http://davpe.net

Algoritmus Boyer–Moore[1] slouží k vyhledání vzoru V v zadaném textu T. Stejně jako v naivním algoritmu je vzor zarovnán se začátkem textu, avšak porovnává se zprava. V případě neúspěchu navrhnou dvě heuristiky počet pozic, o které by se měl vzor posunout doprava. Algoritmus si vybere větší z navržených posunů.
Nadále budeme značit délku vzoru V jako $\mathcal{V}$ a délku textu T jako $\mathcal{T}$. Všechna pole jsou indexována od jedničky, pokud není řečeno jinak.

Posun při špatném znaku (Bad character shift)

Představme si, že právě zprava porovnáváme vzor textem a na pozici t textu T se znak $c = T[t]$ liší od právě porovnávaného znaku vzoru na místě v. Pak mohou nastat tyto možnosti.
1. Znak c ve vzoru vůbec nevyskytuje, nemá smysl se pozicí t zabývat. Posuneme tedy vzor tak, aby jeho první znak byl těsně za pozicí t v textu.
2. Znak c ve vzoru vyskytuje jen a pouze nalevo od $V[v]$. Vezměme znak $c = V[u]$, který je nejblíže k $V[v]$. Pak nás znaky mezi $V[u]$ a $V[v]$, nebudou zajímat, neboť každý z nich selže. Proto posuneme vzor doprava tak, aby znak $V[u]$ byl právě pod znakem $T[t]$.
3. Znak c ve vzoru vyskytuje nalevo i napravo od $V[v]$. Dávalo by smysl posunout vzor stejně jako v předchozím případě. Bohužel, museli bychom si pamatovat všechny pozice znaku c, což by nám algoritmus zpomalilo. Posuneme vzor jen o jednu pozici doprava a spolehneme se na druhou heuristiku.



1. Znak c =D se ve vzoru nevyskytuje.

2. Znak c =B je ve vzoru jen vlevo, posouváme na znak B.

3. Znak c =K je ve vzoru i vpravo, posouváme o jednu pozici.

Obrázek 1: Příklady posunů při špatném znaku.

Abychom poznali, o kolik znaků vzor posunout budeme si pro něj pamatovat pole `bcs` indexované abecedou Σ, se kterou pracujeme (v našich příkladech to budou písmena A–Ž a mezera) definovanou takto

$$\texttt{bcs}[c] = \begin{cases} \mathcal{V} - i & \text{pokud } V[i] = c \ \wedge \ i \neq \mathcal{V} \ \wedge \ i \text{ je maximální možné} \\ \mathcal{V} & \text{pokud } V[\mathcal{V}] = c \vee c \notin V \end{cases}$$

Jinými slovy hodnota `bcs`[c] je počet znaků mezi posledním výskytem písmene c ve vzoru a místem za koncem vzoru. Pokud písmeno c je na posledním místě vzoru, nebo v něm vůbec není bude hodnota rovna délce vzoru.

Proč právě tyto hodnoty? Představme si, že selhalo hned první písmeno c, které jsme porovnávali (tedy poslední písmeno současného prefixu textu). Pak hodnota, o kterou musíme vzor posunout, abychom nám už znak c neselhal je právě `bcs`[c]. Pokud už jsme úspěšně porovnali s znaků (délka úspěšně porovnaného sufixu vzoru je s), posuneme vzor doprava o hodnotu `bcs`[c] - s (pokud vyjde posun záporně, je to třetí případ a posuneme vzor jen o jednu pozici).

A	K	M	O	R	S	U	*
1	7	3	5	2	6	4	7

Posun vzoru doprava o `bcs`[D] $- s = 7 - 3 = 4$

A	B	R	*
2	3	1	7

Posun vzoru doprava o `bcs`[B] $- s = 3 - 1 = 2$

A	I	K	M	R	T	*
3	5	1	4	6	2	9

Posun `bcs`[K] $- s = 1 - 4 = -3$ je záporný, posouváme o 1

Obrázek 2: Tabulky `bcs` pro příklady výše (znaky nevyskytující se ve vzoru jsou označeny hvězdičkou).

Algoritmus pro vyplnění pole `bcs`

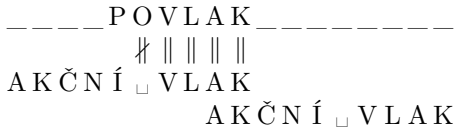
```
bcs[Σ] ← V
for i = 1 to V − 1 do
  bcs[V[i]] ← V − i
end for
```

Na začátku nastavíme všech znakům hodnotu $\mathcal{V}$. Poté procházíme vzor zleva a nastavíme hodnotu `bcs` pro znak $V[i]$ na $\mathcal{V} - i$.
Správnost algoritmu je zřejmá z definice `bcs`. Časová složitost je $O(|\Sigma| + \mathcal{V})$, paměťová složitost je $O(|\Sigma|)$.

Posun při správném sufixu (Weak good-suffix shift)

Představme si, že při porovnávání vzoru zprava s textem jsme již úspěšné porovnali sufix $V[s], \ldots, V[\mathcal{V}]$, ale znak $c = V[s-1]$ už selhal. Heuristika good-suffix se ve vzoru snaží najít podřetězec, který by byl stejný (nebo kratší) jako úspěšně porovnaný sufix. [1]
1. Najdeme ve vzoru stejný podřetězec, jako úspěšně porovnaný sufix (tedy $V[s], \ldots, V[\mathcal{V}]$). Poté posuneme vzor doprava tak, aby pod původním úspěšně porovnaným sufixem byl nově nalezený sufix. Budeme tak mít zaručeno, že už máme úspěšně porovnáno s znaků.
2. Pokud takový řetězec neexistuje, najdeme nejdelší prefix vzoru, který je sufixem úspěšně porovnaného sufixu.

1. Najdeme ve vzoru nejbližší řetězec „KUS“.



2. Nejdelší prefix vzoru „AK“, který je sufixem vzoru.

Oproti předchozí heuristice je tato o něco složitější a její algoritmus má dvě části. V první části si vyrobíme pole `pref` a `ferp`. Na pozici `pref`[i] bude velikost nejdelšího prefixu vzoru, který je vlastním sufixem $V[1..i]$. Na pozici `ferp`[i] bude to samé co na pozici `pref`[i] pro vzor napsaný pozpátku. Budeme jej značit V_{rev}. Ve druhé části tato pole použijeme ke zkonstruování finálního pole `gss`. Na jeho i-tém místě bude nejmenší počet pozic, o které musíme vzor posunout doprava, abychom na místo $V[i+1..\mathcal{V}]$ (což je úspěšně porovnaný sufix) nebo jeho sufixu dostali stejný řetězec.

	0	1	2	3	4	5	6	7	8	9	10	11
V		K	U	S	K	U	S	U	␣	K	U	S
`pref`		0	0	0	1	2	3	0	0	1	2	3
`ferp`		0	0	0	0	0	1	2	3	1	2	3
`gss`	8	8	8	8	8	8	8	8	5	5	5	8

1. Posun vzoru o `gss`[7]=5.

	0	1	2	3	4	5	6	7	8	9	10
V		A	K	Č	N	Í	␣	V	L	A	K
`pref`		0	0	0	0	0	0	0	0	1	2
`ferp`		0	0	0	0	0	0	0	0	1	2
`gcs`	8	8	8	8	8	8	8	8	8	8	8

1. Posun vzoru o `gss`[5]=8.

Počítání polí `pref` a `ferp`

Nadefinujeme si formálně obě pole.

$$\begin{aligned} \texttt{pref}[i] &= \max(\{k : V[1..k] \text{ je vlastní sufix } V[1..i]\}) \\ \texttt{ferp}[i] &= \max(\{k : V_{rev}[1..k] \text{ je vlastní sufix } V_{rev}[1..i]\}) \end{aligned}$$

Funkci, která počítá pole `pref` nazveme π. A dokonce už ji známe; je to stejná funkce, která v KMP automatu konstruuje zpětné hrany a máme o ní i dokázáno, že běží v čase $O(\mathcal{V})$. Pole `ferp` získáme jednoduše, stačí funkcí π předat vzor, který je pozpátku. Budeme jej značit V_{rev}.

[1] Existuje silnější varianta zvaná Strong good suffix shift. Není součástí původního algoritmu, avšak umožňuje dokázat jeho linearitu. Přidává navíc podmínku, že se před nově nalezeným podřetězcem nesmí vyskytovat znak c.

Počítání `gss`

Nejprve si nadefinujeme pole `gss`. Poté budeme pomocí různých pozorování definici upravovat, až vytvoříme základ pro algoritmus počítající jeho hodnoty.

$$\texttt{gss}[i] = \mathcal{V} - \max(\{k : 0 \leq k < \mathcal{V} \wedge V[i+1..\mathcal{V}] \text{ je sufix } V[1..k] \text{ nebo naopak}\})$$

Pozorování: Platí $\forall i : \texttt{gss}[i] \leq \mathcal{V} - \texttt{pref}[\mathcal{V}]$.
Důkaz: Upravme nerovnici na tvar $\mathcal{V} - k \leq \mathcal{V} - \texttt{pref}[\mathcal{V}]$, kde k pochází z definice `gss` a je maximální. Pak $k \geq \texttt{pref}[\mathcal{V}_1]$ což platí, neboť prefix vzoru V délky k, jehož sufix je i sufixem V je zjevně delší, než prefix vzoru V, který je sufixem vzoru V.

Dále si všimneme, že pokud v poslední podmínce definice `gss` platí varianta $V[0..k]$ je sufixem V, pak $k \leq \texttt{pref}[\mathcal{V}]$. Dle předchozího pozorování nám tento případ hodnoty `gss` nezmění a můžeme jej zanedbat. Upravme si definici `gss`.

$$\texttt{gss}[i] = \mathcal{V} - \max(\{\texttt{pref}[\mathcal{V}]\},\ \{k : \texttt{pref}[\mathcal{V}] < k < \mathcal{V} \wedge V[i+1..\mathcal{V}] \text{ je sufix } V[1..k]\})$$

Tvrzení: Nechť $V[i+1..\mathcal{V}]$ je sufix $V[1..k]$ a k je největší takové. Pak $\texttt{ferp}[l] = \mathcal{V} - i$, kde $l = (\mathcal{V} - k) + (\mathcal{V} - i)$.
Důkaz: Neboť $V[i+1..\mathcal{V}]$ je sufix $V[1..k]$ pak zřejmě $V_{rev}[1..\mathcal{V}-i]$ je sufix $V_{rev}[1..l]$. Pak $\texttt{ferp}[l] \geq \mathcal{V} - i$. Nechť pro spor $p > \mathcal{V} - i$ kde $p = \texttt{ferp}[l]$. Z definice `ferp` plyne $V_{rev}[1..p]$ je sufix $V_{rev}[1..l]$, což je ekvivalentní tomu, že $V_{rev}[1..p] = V_{rev}[l-p+1..l]$. Přepíšeme-li tento vztah z *rev* tvaru do běžného, dostaneme $V[\mathcal{V}-p+1..\mathcal{V}] = V[\mathcal{V}-l+1..\mathcal{V}-l+p]$. Nyní provedeme substituci pro $l = 2\mathcal{V} - k - i$ a dostaneme $V[\mathcal{V}-p+1..\mathcal{V}] = V[k-\mathcal{V}+i+1..k-\mathcal{V}+i+p]$. To znamená, že $V[\mathcal{V}-p+1..\mathcal{V}]$ je sufix $V[1..k-\mathcal{V}+i+p]$. Neboť $p > \mathcal{V} - i$ pak $i+1 > \mathcal{V}-p+1$ a $V[i+1..\mathcal{V}]$ je sufix $V[\mathcal{V}-p+1..\mathcal{V}]$. Z tranzitivity sufixů plyne $V[i+1..\mathcal{V}]$ je sufix $V[1..k-\mathcal{V}+i+p]$. Neboť $p > \mathcal{V} - i$, pak jsme našli $k' > k$, kde $k' = k - \mathcal{V} + i + p$ což je spor s maximalitou k.

Využijeme předchozí tvrzení a se znalostí, že $i = \mathcal{V} - \texttt{ferp}[l]$ a $k = \mathcal{V} - l - \texttt{ferp}[l]$ naposledy upravíme definici `gss` a předvedeme algoritmus na jeho spočtení.

$$\begin{aligned}
\texttt{gss}[i] &= \mathcal{V} - \max(\{\texttt{pref}[\mathcal{V}]\},\ \{\mathcal{V} - l - \texttt{ferp}[l] : 1 \leq l \leq \mathcal{V} \wedge i = \mathcal{V} - \texttt{ferp}[l]\}) \\
&= \min(\{\mathcal{V} - \texttt{pref}[\mathcal{V}]\},\ \{l - \texttt{ferp}[l] : 1 \leq l \leq \mathcal{V} \wedge i = \mathcal{V} - \texttt{ferp}[l]\})
\end{aligned}$$

Algoritmus pro vyplnění pole `gss`

```
pref ← π(V)
ferp ← π(V_rev)
for i = 0 to V do
  gss[i] ← V − pref[V]
end for
for l = 1 to V do
  i ← V − ferp[l]
  if gss[i] > l − ferp[l] then
    gss[i] ← l − ferp[l]
  end if
end for
```

Správnost algoritmu plyne z tvrzení a pozorování výše, časová i paměťová složitost je zřejmě $O(\mathcal{V})$.

Algoritmus Boyer–Moore

Pseudokód samotného algoritmu je uveden níže. Předpokládá, že pole `bcs` a `gss` pro obě heuristiky již máme předpočítané.

$t \leftarrow 0$
while $t \leq \mathcal{T} - \mathcal{V}$ **do**
  $v \leftarrow \mathcal{V}$
  while $v > 0 \wedge V[v] = T[t + v]$ **do**
    $v \leftarrow v - 1$
  end while
  if $v = 0$ **then**
    print „Vzor se v textu vyskytuje na pozici “ t
    $t \leftarrow t + \texttt{gss}[0]$
  else
    $t \leftarrow t + \mathbf{max}(\texttt{gss}[v], \texttt{bcs}[T[t + v]] - (\mathcal{V} - v), 1)$
  end if
end while

Správnost algoritmu plyne ze správnosti obou heuristik. Časová složitost je podobná jako u naivního algoritmu $O(\mathcal{TV} + |\Sigma|)$, v praxi je však algoritmus mnohem rychlejší. Uveďme ještě dolní, daleko zajímavější odhad časové složitosti $\Omega(\mathcal{T}/\mathcal{V})$. Příkladem budiž text, tvořený samými písmeny „A“ a vzor tvořený písmeny „B“. Paměťová složitosti je $O(\mathcal{T} + \mathcal{V})$.

Reference

[1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of ACM*, 20(10):762-772, 1977