6. Vyhledávání v textu *(zapsal K. Kaščák, M. Klaučo, M. Vachna)*

Úkol: V textu s délkou S najít všechny výskyty hledaného slova s délkou J.

Hloupý algoritmus

Algoritmus prochází sekvenčně textem a hledaným vzorovým slovem. Při neshodě se ve vzorovem slově vrací na začátek a v textu pokračuje znakem, ve kterém nastala neshoda. Časová složitost je $\mathcal{O}(S)$. Tento algoritmus funguje pouze pro vzorová slova, ve kterých se neopakuje první znak.

Příklad: Hledání vzorového slova `jehla` v textu `vkupcejejehla`. Ve chvíli kdy máme prefix `je` a na vstupu dostaneme `j`, dochází k neshodě a pokračujeme v hledání od tohoto znaku. Pro tenhle případ algoritmus najde vzorové slovo. To ale již neplatí pro vzorové slovo `kokos` v textu `clanekokokosu`. Ve chvíli kdy máme prefix `koko` a na vstupu dostaneme `k`, dochází k neshodě a pokračujeme v hledání od tohoto znaku, tím ale zahodíme potřebnou část a algoritmus selže.

Neefektivní algoritmus

Algoritmus prochází text od začátku až do konce a pro každou pozici v textu zkontroluje, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provede až S porovnání znaků, čili celkem až SJ porovnání. Proto je časová složitost $\mathcal{O}(SJ)$.

Chytrý algoritmus

Algoritmus je vylepšením Neefektivního algoritmu, konkrétně způsobu, jakým sa vrací v textu při neshodě mezi znakem textu a znakem vzorového slova.

Příklad: Pro vzorové slovo `ajaajak` jsme našli v textu prefix `ajaaja`. Očekávame `k`.

- Když ale dostaneme `a` a budeme mít prefix `ajaajaa`, vracíme se v textu za první `aja`, tedy prefix zkrátíme na `ajaa` a pokračujeme v hledání.
- Když je nasledující znak `j` a budeme mít prefix `ajaajaj`, vracíme se v textu za `ajaaj`, tedy prefix zkrátíme na `aj` a pokračujeme v hledání.
- V případě, že dostaneme jiný znak, se v textu nevracíme a pokračujeme dalším znakem textu.

Definice a značení pro řetězce (slova):

Definice:

- *Abeceda* Σ je konečná množina znaků, ze kterých tvoříme text, řetězce, slova jako konečné posloupnosti znaků z Σ. Příkladem extrémních abeced je binární abeceda složená z nul a jedniček. Příklad z druhého konce je abeceda, která má jako znaky slova českého jazyka. V algoritmech nebudeme uvažovat velikost abecedy (počet znaků), budeme předpokládat, že je to konstanta.
- Σ^* je množina všech slov nad abecedou Σ.

Značení:

- *Slova* budeme značit malými písmeny řecké abecedy α,β… a *znaky* malými písmeny latinky a,b… .
- *Prázdné slovo* značíme písmenem ε.
- *Délka slova* $|\alpha|$ pro $\alpha \in \Sigma^*$ je počet jeho znaků.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $\alpha\varepsilon = \varepsilon\alpha = \alpha$, $|\alpha\beta| = |\alpha| + |\beta|$.
- $\alpha[i]$ je i-té písmeno slova α, indexuje se od 0.
- $\alpha[i : j]$ je podslovo tvořené písmeny $\alpha[i]$,…,$\alpha[j-1]$. Příklady: $\alpha[i : i+1] = \alpha[i]$, $\alpha[i : i] = \varepsilon$. Vynechaním první meze získame prefix ($\alpha[: j]$), druhé meze suffix ($\alpha[i :]$), obou mezí dostaneme celé slovo ($\alpha[:]=\alpha$).
- $\alpha[: j]$ je *prefix* obsahující prvních j znaků slova α.
- $\alpha[i :]$ je *suffix* obsahující znaky slova α počínaje i-tým znakem.
- Každé slovo je prefixem i suffixem sebe sama, takovému prefixu/suffixu říkáme *nevlastní*. Všem ostatním *vlastní*.
- Prázdné slovo je podslovem, prefixem i suffixem každého slova včetně prázdného slova.

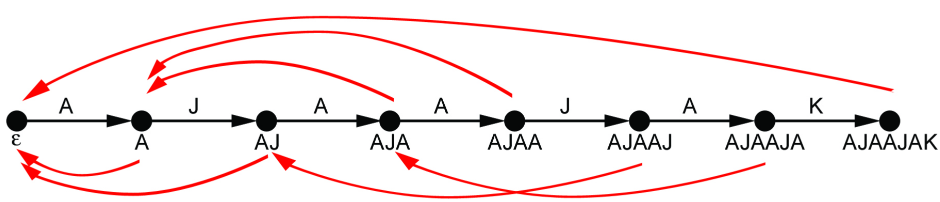
Problém:

Vstupem je ι hledané slovo (jehla) délky $J = |\iota|$ a σ text (seno) délky $S = |\sigma|$.

Výstupem jsou všechny vyskyty hledaného slova ι v textu σ: $\{i|\sigma[i : i + J] = \iota\}$

Vyhledávací automat (Knuth, Morris, Pratt)

Vyhledávací automat bude graf, jehož vrcholům říkame stavy automatu. Jména stavů budou všechny prefixy slova ι. Počáteční stav je prázdné slovo ε a koncový je celá ι. Dopředné hrany grafu budou popisovat přechod mezi stavy ve smyslu zvětšení délky jména stavu (dopředná funkce $d(\alpha, X)$), tedy každá taková hrana bude označena písmenem X a bude popisovat dané zvětšení délky jména stavu, tedy $\alpha \to \alpha X$. Zpětné hrany grafu budú popisovat přechod (zpětná funkce $z(\alpha)$) mezi stavem α a nejdelším vlastním suffixem α, který je prefixem ι, když nastane neshoda.



Vyhledávací automat

Vyhledávání:

1. $\alpha \leftarrow \varepsilon$.

2. Pro $c \in \Sigma$ postupně:
3. Dokud $\neg \exists d(\alpha, c) \wedge \alpha \neq \varepsilon : \alpha \leftarrow z(\alpha)$.
4. Když $\exists d(\alpha, c) \Rightarrow \alpha \leftarrow d(\alpha, c)$.
5. Když $\alpha = \iota \Rightarrow$ hledané slovo je v textu.

Alternatíva: Automat může být reprezentovaný i polem. Při této reprezentaci odpadá starost o dopřední hrany (stačí zvětšit hodnotu, kterou v poli indexujeme). Hodnota na dané pozici v poli určuje kam směruje zpětná hrana (index v poli).

Alternatívní vyhledávání:

1. $k \leftarrow 0$.
2. pro $c \in \Sigma$ postupně:
3. Dokud $c \neq \iota[k] \wedge k > 0 : k \leftarrow z[k]$
4. Je-li $c = \iota[k] \Rightarrow k \leftarrow k + 1$
5. Když $k = J \Rightarrow$ hledané slovo je v textu

Invariant: Nejdelší suffix β, který je prefixem $\iota = \alpha(\beta)$. Kde β je přečtení vstup. Z invariantu vyplýva korektnost vyhledávací části algoritmu KMP.

Důkaz: Indukcí podle $|\beta|$. Na začátku pro prázdný načtený vstup platí invariant, tedy prázdny suffix β je prefixem ι. V kroku n máme načtený vstup β a k němu načteme znak c. Jestli si odmyslíme c, tedy když si od jména stavu odmyslíme poslední písmenko, dostaneme znovu jméno stavu. Tak stav, který pasuje na konec vstupu bez toho c je stav, který pasuje na konec původního vstupu, toho o jeden znak kratšího. Tím pádem to musí být něco, co je maximálně tak dlouhé jako původní stav, u kterého jsme byli, protože to byl nejdelší, který pasoval. Stačí procházet postupně všechny stavy, které pasují na konec toho vstupu od nejdelšího k nejkratšímu a vzít první, který se dá rozšířit o c. To je přesně to, co algoritmus dělá. Protože zpětná funkce řekne nejbližší kratší jméno stavu. Takže algoritmus iteruje přes stavy, které tam pasují, až najde jeden, který se dá rozšířit o c a jelikož iteroval od toho nejdelšího, tak to je logicky ten nejdelší, který tam pasuje. ♡

Lemma: Vyhledávání doběhne v čase $\mathcal{O}(S)$.

Důkaz: Pro každý znak vstupního textu mohou nastat dva případy. Znak rozširuje aktuální prefix, nebo musíme použít zpětnou funkci (zpětnou hranu). Rozširování trvá konstantně mnoho času, zatímco zpětná funkce muže být pro jeden znak volána až J-krát. Při každém volání klesne délka aktuálního stavu minimálně o jedna a zároveň platí, že kdykoliv stav prodlužujeme, roste právě o jeden znak. Proto všech zkrácení dohromady muže být nejvýše tolik, kolik bylo všech prodloužení, t.j. kolik jsme přečetli znaku textu. Celkem je tedy počet kroků lineární vzhledem k délce textu. ♡

Konstrukce zpětné funkce:

1. Sestrojíme dopředné hrany
2. $z(\varepsilon) \leftarrow 0$, $z(\iota[0]) \leftarrow \varepsilon$
3. $\alpha \leftarrow \varepsilon$
4. pro $i = 1$ do J

5. $\alpha \leftarrow krok(\alpha, \iota[i])$
6. $z(\iota[0:i+1]) \leftarrow \alpha$

Vysvětlení: Všimněte si, že $z(i)$ je přesně stav, do nejž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\iota[2:i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco α označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku. Takže $z()$ získáme tak, že spustíme vyhledávání na část samotného slova ι. Jenže k vyhledávání zase potřebujeme zpětnou funkci z. Proto budeme zpětnou funkci vytvářet postupne od nejkratších prefixů. Zřejmě $z(1)=\varepsilon$. Pokud již máme $z(i)$, pak výpočet $z(i+1)$ odpovídá spuštení automatu na slovo délky i a pritom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku, protože $(i+1)$-ní prefix je prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celý řetězec ι a sledovat, jakými stavy bude procházet. To budou přesně hodnoty zpětné funkce. Vytvoření zpětné funkce se tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$.

Algoritmus Rabin & Karp

Tento algoritmus funguje tak, že porovnává hash hledaného řetězce s hashem aktuálního podřetězce („posuvné okénko“ stejné délky jako hledaný řetězec) v textu a aktuální podřetězec porovná se vzorkem pouze v případě, když mají shodný hash. Když si zvolíme tu správnou hashovací funkci, budeme moci vypočítat hash následujíciho podřetězce na základe hashe toho aktuálního. Jako hashovací funkci $h:\Sigma^J \rightarrow \mathbb{Z}$ použijeme následující: $h(x_0,...,x_{J-1}) = (\sum_{i=0}^{J-1} x_i.p^{J-1-i}) \bmod N$, kde N je velikost prostoru, do kterého hashujeme. Jak zjistíme hash následujícího podřetězce?

- $h = x_0.p^J + x_1.p^{J-1} + ... + x_{J-1}.p^1$
- $h^{'} = x_1.p^J + x_2.p^{J-1} + ... + x_J.p^1$
- $h^{'} = (h - x_0.p^J).p + x_J.p^1$

Tady můžeme vidět, že hash následujícího řetězce lze přepočítat na základě toho předchozího v konstantním čase. Časová složitost je v nejlepším případě lineární vzhledem k délce textu, zatímco nejhorší případ múže trvat až $\Theta(JS)$.