6. Vyhledávání v textu

(zapsal: Petr Jankovský)

Nyní se budeme věnovat následujícímu problému: v textu délky S (seně) budeme chtít najít všechny výskyty hledaného slova délky J (jehly). Nejprve se podívejme na jeden primitivní algoritmus, který nefunguje. Je ale zajímavé rozmyslet si, proč.

Hloupý algoritmus

Začneme prvním písmenkem hledaného slova a budeme postupně procházet text, až najdeme první výskyt počátečního písmenka. Poté budeme testovat, zda souhlasí i písmenka další. Pokud nastane neshoda, v hledaném slově se vrátíme na začátek a v textu pokračujeme znakem, ve kterém neshoda nastala. Podívejme se na příklad.

Příklad: Budeme hledat slovo `jehla` v textu `jevkupcejejehla`. Vezmeme si tedy první písmenko `j` v hledaném slově a zjistíme, že v textu se nachází hned na začátku. Vezmeme tedy další písmenko `e`, které se vyskytuje jako druhé i v textu. Při třetím písmenku ale narazíme na neshodu. V tuto chvíli tedy zresetujeme a opět hledáme výskyt písmenka `j`, tentokrát však až od třetího písmene v textu. Takto postupujeme postupně dál, až narazíme na další `je`, které ovšem není následováno písmenem `h`, tudíž opět zresetujeme a nakonec najdeme shodu s celým hledaným řetězcem. V tomto případě tedy algoritmus našel hledané slovo.

Tento algoritmus však zjevně může hanebně selhat. Může se stát, že začneme porovnávat, až v jednu chvíli narazíme na neshodu. Celý tento kus tedy přeskočíme. Při tom se ale v tomto kusu textu mohl vyskytovat nějaký překrývající se výskyt hledané „jehly“. Hledejme například řetězec `kokos` v textu `clanekokokosu`. Algoritmus tedy začne porovnávat. Ve chvíli kdy najde prefix `koko` a na vstupu dostane `k`, dochází k neshodě. Proto algoritmus zresetuje a pokračuje v hledání od tohoto znaku. Najde sice ještě výskyt `ko`, ovšem s dalším písmenkem `s` již dochází k neshodě a algotimus selže. Nesprávně se totiž „upnul“ na první nalezené `koko` a s dalším `k` pak „zahodil“ i správný začátek.

Máme tedy algoritmus, který i když je špatně, tak funguje určitě kdykoli se první písmenko hledaného slova v tomto slově už nikde jinde nevyskytuje – což `jehla` splňovala, ale `kokos` už ne.

Hloupý algoritmus se na každé písmenko textu podívá jednou, tudíž časová složitost bude lineární s délkou textu ve kterém hledáme – tedy $\mathcal{O}(S)$.

Pomalý algoritmus

Zkusíme algoritmus vylepšit tak, aby fungoval správně: pokud nastane nějaká neshoda, vrátíme se zpátky těsně za začátek toho, kdy se nám to začalo shodovat. To je ovšem vlastně skoro totéž, jako brát postupně všechny možné začátky v „seně“ a pro každý z něj ověřit, jestli se tam „jehla“ nachází či nikoliv.

Tento algoritmus evidentně funguje. Běží však v čase: S možných začátků, krát čas potřebný na jedno porovnání (zda se na dané pozici nenachází „jehla“), což nám může trvat až J. Proto je časová složitost $\mathcal{O}(SJ)$. V praxi bude algoritmus často

rychlejší, protože typicky velmi brzo zjistíme, že se řetězce neshodují, ale je možné vymyslet vstup, kde bude potřeba porovnání opravdu tolik.

Nyní se pokusme najít takový algoritmus, který by byl tak rychlý, jako *Hloupý algoritmus*, ale chytrý, jako ten *Pomalý*.

Chytrý algoritmus

Než vlastní algoritmus vybudujeme, zkusíme se cestou naučit přemýšlet o řetězcích občas trochu překrouceným způsobem. Podívejme se na ještě jeden příklad.

Příklad: Vezměme si například staré italské přízvisko `barbarossa`, které znamená Rudovous. Představme si, že takovéto slovo hledáme v nějakém textu, který začíná `barbar`. Víme, že až sem se nám hledaný řetězec shodoval. Řekněme, že další písmenko textu se shodovat přestane – místo `o` načteme například opět `b`. *Hloupý algoritmus* by velil vrátit se k `a` a od něj hledat dál. Uvědomme si ale, že když se vracíme z `barbar` do `arbar` (tedy řetězce, který již známe), můžeme si předpočítat, jak dopadne hledání, když ho pustíme na něj. V předpočítaném bychom tedy chtěli ukládat, že když máme řetězec `arbar`, tak `ar` a `r` nám do hledaného nepasuje a až `bar` se bude shodovat. Tedy místo toho, abychom spustili nové hledání od `a`, můžeme ho spustit až od `b`. Co víc, my dokonce víme, jak dopadne to – pokud totiž nastane neshoda po přečtení `barbar`, je to stejné, jako kdybychom přečetli pouze `bar`, na které se (původne neshodující se) `b` už navázat dá. Kdyby se nedalo navázat ani tam, tak bychom opět zkracovali… Nejen, že tedy víme, kam se máme vrátit, ale víme dokonce i to, co tam najdeme.

Myšlenka, ke které míříme, je předpočítat si nějakou tabulku, která nám bude říkat, jak se máme při hledání vracet a jak to dopadne, a pak už jenom prohlédávat s použitím této tabulky.

Aby se nám o těchto algoritmech lépe mluvilo a především psalo, pojďme si povědět několik definic.

Definice:

- *Abeceda* Σ je konečná množina znaků[1], ze kterých tvoříme text, řetězce, slova.
- Σ^* je množina všech slov nad abecedou Σ. Čili množina všech neprázdných konečných posloupností znaků ze Σ.

[1] Můžeme při tom jít až do extrémů. Příkladem extrémních abeced je binární abeceda složená pouze z nul a jedniček. Příklad z druhého konce (který rádi dělají lingvisti) je abeceda, která má jako abecedu všechna česká slova. Všechny české věty, pak nejsou nic jiného, než slova nad touto abecedou. Použitá abeceda tedy může být i relativně obrovská. Dalším takovým příkladem může být Unicode. Pro naše potřeby ale zatím budeme předpokládat, že abeceda je nejen konstantně velká, ale i rozumně malá. Budeme si moci tedy dovolit například indexovat pole znakem abecedy (kdybychom nemohli, tak bychom místo pole použili například hashovací tabulku, či něco podobného…).

Značení: Aby se nám nepletlo značení, budeme rozlišovat proměnné pro slova, proměnné pro písmenka a proměnné pro čísla.

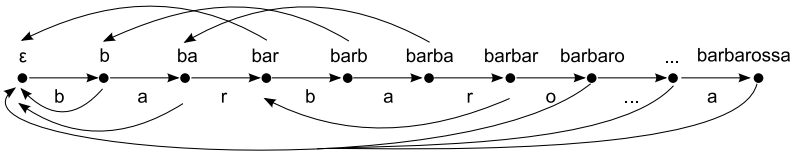
- *Slova* budeme značit malými písmenky řecké abecedy α, β, ...
- ι bude označovat „jehlu“
- σ bude označovat „seno“
- *Znaky* označíme malými písmeny latinky a, b, ...
- *Čísla* budeme značit velkými písmeny A, B, ...
- *Délka slova* $|\alpha|$ pro $\alpha \in \Sigma^*$ je počet jeho znaků.
- *Prázdné slovo* značíme písmenem ε, $|\varepsilon| = 0$.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $|\alpha\beta| = |\alpha| + |\beta|$, $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- $\alpha[k]$ je k-tý znak slova α, indexujeme od 0.
- $\alpha[k : l]$ je podslovo začínající k-tým znakem a l-tý znak je první, který v něm není. Jedná se tedy o podslovo skládající se z $\alpha[k]$,$\alpha[k+1]$,...,$\alpha[l-1]$. Platí tedy: $\alpha[k : k] = \varepsilon$, $\alpha[k : k+1] = \alpha[k]$. Jednu (či obě) meze můžeme i vynechat, tento zápis pak bude znamenat buď „od začátku slova až někam“, nebo „odněkud až do konce“:
- $\alpha[: k]$ je *prefix* obsahující prvních k znaků slova α ($\alpha[0]$,...,$\alpha[k-1]$).
- $\alpha[k :]$ je *suffix* obsahující znaky slova α počínaje k-tým znakem až do konce.
- $\alpha[:] = \alpha$

Všimněme si, že prázdné slovo je prefixem, suffixem i podslovem jakéhokoliv slova včetně sebe sama. Každé slovo je také prefixem, suffixem i podslovem sebe sama. To se ne vždy hodí. Někdy budeme chtít říct, že nějaké slovo je *vlastním* prefixem nebo suffixem. To bude znamenat, že to nebude celé slovo.

α je *vlastní prefix* slova $\beta \equiv \alpha$ je prefix β & $\alpha \neq \beta$.

Vyhledávací automat (Knuth, Morris, Pratt)

Vyhledávací automat bude graf, jehož vrcholům budeme říkat *stavy*. Jejich jména budou prefixy hledaného slova a hrany budou odpovídat tomu, jak jeden prefix můžeme získat z předchozího prefixu přidáním jednoho písmene. 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 $h(\alpha)$, určující znak na dopředné hraně z α). Zpětné hrany grafu budou popisovat přechod (zpětná funkce $z(\alpha)$) mezi stavem α a nejdelším vlastním suffixem α, který je prefixem ι, když nastane neshoda.



Vyhledávací automat.

Hledej(σ):

1. $\alpha \leftarrow \varepsilon$.
2. Pro $x \in \sigma$ postupně:
3. Dokud $h(\alpha) \neq x$ & $\alpha \neq \varepsilon : \alpha \leftarrow z(\alpha)$.
4. Pokud $h(\alpha) = x : \alpha \leftarrow \alpha x$.
5. Pokud $\alpha = \iota$, ohlásíme výskyt.

Vstupem je ι, hledané slovo (jehla) délky $J = |\iota|$ a σ, text (seno) délky $S = |\sigma|$. Výstupem jsou všechny výskyty hledaného slova ι v textu σ, tedy množina $\{k \mid \sigma[k : k + J] = \iota\}$

Pojďme nyní dokázat, že tento algoritmus správně ohlásí všechny výskyty.

Definice: $\alpha(\tau)$:= stav automatu po přečtení τ

Invariant: Pokud algoritmus přečte nějaký vstup, nachází se ve stavu, který je nejdelším suffixem přečteného vstupu, který je nějakým stavem. $\alpha(\tau)$ = nejdelší stav (nejdelší prefix jehly), který je suffixem τ (přečteného vstupu).

Pojďme si rozmyslet, že z tohoto invariantu ihned plyne, že algoritmus najde to, co má. Kdykoli totiž ohlásí nějaký výskyt, tak tam tento výskyt opravdu je. Kdykoli pak má nějaký výskyt ohlásit, tak se v této situaci jako suffix toho právě přečteného textu vyskytuje hledané slovo, přičemž hledané slovo je určitě stav a zároveň nejdelší ze všech existujících stavů. Takže invariant opravdu říká, že jsme právě v koncovém stavu a algoritmus nám tedy ohlásí výskyt.

Důkaz: (invariantu) Indukcí podle kroku algoritmu. Na začátku pro prázdný načtený vstup invariant triviálně platí, tedy prázdný suffix τ je prefixem ι. V kroku n máme načtený vstup τ a k němu připojíme znak x. Invariant nám říká, že nejdelší stav, který je suffixem, je nejdelší suffix, který je stavem. Nyní se ptáme, jaký je nejdelší stav, který se dá „napasovat“ na konec řetězce τx. Kdykoli však takovýto suffix máme, tak z něj můžeme x na konci odebrat, čímž dostaneme suffix slova τ.

Tedy: pokud β je neprázdným suffixem slova τx, pak $\beta = \gamma x$, kde γ je suffix τ.

Suffix, který máme sestrojit, tedy vznikne z nějakého suffixu slova τ připsáním x. Chceme najít nejdelší suffix slova τx, který je stavem, takže chceme najít i nejdelší suffix původního slova τ, za který se dá přidat x tak, aby vyšlo jméno stavu. Stačí tedy už jen „probírat“ suffixy slova τ od nejdelšího po nejkratší, zkoušet k nim přidávat x a až to půjde, tak jsme našli nejdelší suffix τx. Přesně toto ovšem algoritmus dělá, neboť zpětná funkce mu vždy řekne nejbližší kratší suffix, který je stavem. Pokud pak nemůžeme x přidat ani do ε, pak je řešením prázdný suffix. Algoritmus tedy funguje. ♡

Nyní pojďmě zkoumat to, jak je ve skutečnosti náš algoritmus rychlý. K tomu bychom si ale nejdřív měli říct, jak přesně budeme automat reprezentovat. V algoritmu vystupují nějaká porovnávání stavů, přičemž není úplně jasné, jak zařídit, aby vše trvalo konstantně dlouho. Vyjde nám to ale docela snadno. K reprezentaci automatu nám totiž budou stačit pouze dvě pole.

Reprezentace automatu: Očíslujeme si stavy délkami příslušných prefixů, tedy čísly $0 \ldots J$. Poté ještě potřebujeme nějakým způsobem zakódovat dopředné a zpětné hrany. Vzhledem k tomu, že z každého vrcholu vede vždy nejvýše jedna dopředná a nejvýše jedna zpětná, tak nám evidentně stačí pamatovat si pro každý typ hran pouze jedno číslo na vrchol. Budeme mít tedy nějaké pole dopředných hran, které nám pro každý stav řekne, jakým písmenkem je nadepsaná dopředná hrana ze stavu I do $I+1$. To jsou ale přesně písmenka jehly, takže si stačí pamatovat jehlu samotnou. Čili z I do $I+1$ vede hrana nadepsaná $\iota[I]$. Pro zpětné hrany pak budeme potřebovat pole Z, které nám pro stav I řekne číslo stavu, do kterého vede zpětná hrana. Tedy $Z[I]$ je cíl zpětné hrany ze stavu I. S touto reprezentací již dokážeme naši hledací proceduru přímočaře přepsat tak, aby sahala pouze do těchto dvou polí:

1. $I \leftarrow 0$.
2. Pro znaky x z textu:
3. Dokud $\iota[I] \neq x$ & $I \neq 0 : I \leftarrow Z[I]$.
4. Pokud $\iota[I] = x$, pak $I \leftarrow I + 1$.
5. Pokud $I = J$, ohlásíme výskyt.

Zatím se v algoritmu ještě skrývá drobná chyba – totiž algoritmus se občas zeptá na dopřednou hranu z posledního stavu. Pokud jsme právě ohlásili výskyt (jsme tedy v posledním stavu) a přijde nějaký další znak, algoritmus se ptá, zda je roven tomu, co je na dopředné hraně z posledního stavu. Ta ale ovšem neexistuje. Jednoduše to ale napravíme tak, že si přidáme fiktivní hranu, na které se vyskytuje nějaké „nepísmenko“ – něco, co se nerovná žádnému jinému písmenku. Zajistíme tak, že se po této hraně nikdy nevydáme. Dodefinujeme tedy $\iota[J]$ odlišně od všech znaků.[2]

Lemma: Funkce *Hledej* běží v čase $\mathcal{O}(S)$.

Důkaz: Funkce *Hledej* chodí po dopředných a zpětných hranách. Dopředných hran projdeme určitě maximálně tolik, kolik je délka sena. Pro každý znak přečtený ze sena totiž jdeme nejvýše jednou po dopředné hraně. Se zpětnými hranami se to má tak, že na jeden přečtený znak z textu se můžeme po zpětné hraně vracet maximálně J-krát. Z tohoto by nám však vyšla složitost $\mathcal{O}(JS)$, čímž bychom si nepomohli. Zachrání nás ale přímočarý potenciál. Uvědomme si, že chůze po dopředné hraně zvýší I o jedna a chůze po zpětné hraně I sníží alespoň o jedna. Vzhledem k tomu, že I není nikdy záporné a na začátku je nulové, zjistíme, že kroků zpět může být maximálně tolik, kolik kroků dopředu. Časová složitost hledání je tedy lineární vzhledem k délce sena. ♡

Nyní nám zbývá na první pohled maličkost – totiž zkonstruovat automat. Zkonstruovat dopředné hrany zvládneme zjevně snadno, jsou totiž explicitně popsané

[2] V jazyce C se toto dodefinování provede vlastně zadarmo, neboť každý řetězec je v něm ukončen znakem s kódem nula, který se ve vstupu nevyskytne... Algoritmus bude tedy fungovat i bez tohoto dodefinování. V jiných jazycích je ale třeba na něj nezapomenout!

hledaným slovem. Těžší už to bude pro hrany zpětné. Využijeme k tomu následující pozorování:

Pozorování: Představme si, že automat už máme hotový a tím, že budeme sledovat jeho chování, chceme zjistit, jak v něm vedou zpětné hrany. Vezměme si nějaký stav β. To, kam z něj vede zpětná hrana zjistíme tak, že spustíme automat na řetězec β bez prvního písmenka a stav, ve kterém se automat zastaví, je přesně ten, kam má vést i zpětná hrana z β. Jinými slovy víme, že $z(\beta) = \alpha(\beta[1:])$. Proč takováto věc funguje? Všimněme si, že definice z a to, co nám o α říká invariant, je téměř totožné – $z(\beta)$ je nejdelší vlastní suffix β, který je stavem, $\alpha(\beta)$ je nejdelší suffix β, který je stavem. Jediná odlišnost je v tom, že definice z narozdíl od definice α zakazuje nevlastní suffixy. Jak nyní vyloučit suffix β, který by byl roven β samotné? Zkrátíme β o první znak. Tím pádem všechny suffixy β bez prvního znaku jsou stejné jako všechny vlastní suffixy β.

K čemu je toto pozorování dobré? Rozmysleme si, že pomocí něj už dokážeme zkonstruovat zpětné hrany. Není to ale trochu divné, když při simulování automatu na řetězec bez prvního znaku už zpětné hrany potřebujeme? Není. Za chvíli zjistíme, že takto můžeme zjišťovat zpětné hrany postupně – a to tak, že používáme vždy jenom ty, které jsme už sestrojili.

Takovémuhle přístupu, kdy při konstruování chtěného už používáme to, co chceme sestrojit, ale pouze ten kousek, který již máme hotový, se v angličtině říká *bootstrapping*⟨3⟩. Všimněme si, že při výpočtu se vstupem β projde automat jenom prvních $|\beta|$ stavů. Automat se evidentně nemůže dostat dál, protože na každý krok dopředu (doprava) spotřebuje písmenko β. Takže kroků doprava je maximálně tolik, kolik je $|\beta|$. Jinými slovy kdybychom již měli zkonstruované zpětné hrany pro prvních $|\beta|$ stavů (tedy $0 \ldots |\beta|-1$), tak při tomto výpočtu, který potřebujeme na zkonstruování zpětné hrany z β, ještě tuto zpětnou hranu nemůžeme potřebovat. Vystačíme si s těmi, které již máme zkonstruované.

Nabízí se tedy začít zpětnou hranou z prvního znaku (která vede evidentně do ε), pak postupně brát další stavy a pro každý z nich si spočítat, kdy spustíme automat na jméno stavu bez prvního znaku a tím získáme další zpětnou hranu. Toto funguje, ale je to kvadratické .... Máme totiž J stavů a pro každý z nich nám automat běží v čase až lineárním s J. Jak z toho ven?

Z prvního stavu povede zpětná funkce do ε. Pro další stavy chceme spočítat zpětnou funkci. Z druhého stavu $\iota[0:2]$ tedy automat spustíme na $\iota[1:2]$, dále pak na $\iota[1:3]$, $\iota[1:4]$, atd. Ty řetězce, pro které potřebujeme spoštět automat, abychom

⟨3⟩ Z tohoto slova vzniklo i *bootování* počítačů, kdy operační systém v podstatě zavádí sám sebe. Bootstrap znamená česky štruple – tedy očko na konci boty, které slouží k usnadnění nazouvání. A jak souvisí štruple s algoritmem? To se zase musíme vrátit k příběhům o baronu Prášilovi, mezi nimiž je i ten, ve kterém baron Prášil vypráví o tom, jak sám sebe vytáhl z bažiny za štruple. Stejně tak i my budeme algoritmus konstruovat tím, že se budeme sami vytahovat za štruple, tedy bootstrappovat.

dostali zpětné hrany, jsou tedy ve skutečnosti takové, že každý další dostaneme rozšířením předchozího o jeden znak. To jsou ale přesně ty stavy, kterými projde automat při zpracovávání řetězce ι od prvního znaku dál. Jedním průchodem automatu nad jehlou bez prvního písmenka se tím pádem rovnou dozvíme všechny údaje, které potřebujeme. Z předchozího pozorování plyne, že nikdy nebudeme potřebovat zpětnou hranu, kterou jsme ještě nezkonstruovali a jelikož víme, že jedno prohledání trvá lineárně s délkou toho, v čem hledáme, tak toto celé poběží v lineárním čase. Dostaneme tedy následující algoritmus:

Konstrukce zpětné funkce:

1. $Z[0] \leftarrow ?$, $Z[1] \leftarrow 0$.
2. $I \leftarrow 0$.
3. Pro $k = 2 \ldots J$:
4. $\quad I \leftarrow \mathit{Krok}(I, \iota[k])$.
5. $\quad Z[k] \leftarrow I$.

Začínáme tím, že nastavíme zpětnou hranu z prvních dvou stavů, přičemž $z[0]$ je nedefinované, protože tuto zpětnou hranu nikdy nepoužíváme. Dále postupně simulujeme výpočet automatu nad slovem bez prvního znaku a po každém kroku se dozvíme novou zpětnou hranu. *Krokem* automatu pak není nic jiného než vnitřek (3. a 4. bod) naší hledací procedury. To, kam jsme se dostali, pak zaznamenáme jako zpětnou funkci z k. Čili pouštíme automat na jehlu bez prvního písmenka, provedeme vždy jeden krok automatu (přes další písmenko jehly) a zapamatujeme si, jakou zpětnou funkci jsme zrovna dostali. Díky pozorováním navíc víme, že zpětné hrany konstruujeme správně, nikdy nepoužijeme zpětnou hranu, kterou jsme ještě nesestrojili a víme i to, že celou konstrukci zvládneme v lineárním čase s délkou jehly.

Věta: Algoritmus KMP najde všechny výskyty v čase $O(J + S)$.

Důkaz: Lineární čas s délkou jehly potřebujeme na postavení automatu, lineární čas s délkou sena pak potřebujeme na samotné vyhledání.

Rabinův-Karpův algoritmus

Nyní si ukážeme ještě jeden algoritmus na hledání jedné jehly, který nebude mít v nejhorším případě lineární složitost, ale bude ji mít průměrně. Bude daleko jednodušší a ukáže se, že je v praxi daleko rychlejší. Bude to algoritmus založený na hashování.

Představme si, že máme seno délky S a jehlu délky J, a vezměme si nějakou hashovací funkci, které dáme na vstup J-tici znaků (tedy podslova dlouhá jako jehla). Tato hashovací funkce nám je pak zobrazí do množiny $\{0, \ldots, N-1\}$ pro nějaké dost velké N. Jak nám toto pomůže při hledání jehly? Vezmeme si libovolné „okénko“ délky J a než budeme zjišťovat, zda se v něm jehla vyskytuje, tak si spočítáme hashovací funkci a porovnáme ji s hashem jehly. Čili ptáme se, jestli je hash ze sena od nějaké pozice I do pozice $I+J$ roven hashi jehly – formálně: $h(\sigma[I : I+J]) = h(\iota)$. Teprve tehdy, když zjistíme, že se hodnota hashovací fce shoduje, začneme doopravdy porovnávat řetězce.

Není to ale nějaká hloupost? Může nám vůbec takováto konstrukce pomoci? Není to tak, že na spočítání hashovací funkce z J znaků, potřebujeme těch J znaků přečíst, což je stejně rychlé, jako rovnou řetězce porovnávat? Použijeme trik, který bude spočívat v tom, že si zvolíme šikovnou hashovací funkci. Uděláme to tak, abychom ji mohli při posunutí „okénka“ o jeden znak doprava v konstantním čase přepočítat. Chceme umět z $h(x_1 \dots x_j)$ spočítat $h(x_2 \dots x_{j+1})$. Na začátku si tedy spočítáme hash jehly a první J-tice znaků sena. Pak již jenom posouváme „okénko“ o jedna, přepočítáme hashovací funkci a když se shoduje s hashem jehly, tak porovnáme. Budeme přitom věřit tomu, že pokud se tam jehla nevyskytuje, pak máme hashovací funkci natolik rovnoměrnou, že pravděpodobnost toho, že se přesto strefíme do hashe jehly, je $1/N$. Jinými slovy jenom v jednom z řádově N případů budeme porovnávat falešně – tedy provedeme porovnání a vyjde nám, že výsledek je neshoda. V průměrném případě tedy můžeme stlačit složitost až téměř k lineární.

Podívejme se teď na průměrnou časovou složitost. Budeme určitě potřebovat čas na projití jehly a sena. Navíc strávíme nějaký čas nad falešnými porovnáními, kterých bude v průměru na každý N-tý znak sena jedno porovnání s jehlou – tedy SJ/N, přičemž N můžeme zvolit dost velké na to, abychom tento člen dostali pod nějakou rozumnou konstantu… Nakonec budeme potřebovat jedno porovnání na každý opravdový výskyt, čemuž se nevyhneme. Připočteme tedy ještě $J \cdot \sharp$*výskytů*. Dostáváme tedy: $\mathcal{O}(J + S + SJ/N + J \cdot \sharp$*výskytů*$)$.

Zbývá maličkost – totiž kde vzít hashovací funkci, která toto vše splňuje. Jednu si ukážeme. Bude to vlastně takový hezký polynom:

$$h(x_1 \dots x_j) := \left(\sum_{I=1}^{J} x_I \cdot p^{J-I} \right) \bmod N.$$

Jinak zapsáno se tedy jedná o:

$$(x_1 \cdot p^{J-1} + x_2 \cdot p^{J-2} + \dots + x_J \cdot p^0) \bmod N.$$

Po posunutí okénka o jedna chceme dostat:

$$(x_2 \cdot p^{J-1} + x_3 \cdot p^{J-2} + \dots + x_J \cdot p^1 + x_{J+1} \cdot p^0) \bmod N.$$

Když se ale podíváme na členy těchto dvou polynomů, zjistíme, že se liší jen o málo. Původní polynom stačí přenásobit p, odečíst první člen s x_1 a naopak přičíst chybějící člen x_{J+1}. Dostáváme tedy:

$$h(x_2 \dots x_{J+1}) = (p \cdot h(x_1 \dots x_J) - x_1 \cdot p^J + x_{J+1}) \bmod N.$$

Přepočítání hashovací funkce tedy není nic jiného, než přenásobení té minulé p, odečtení nějakého násobku toho znaku, který vypadl z okénka, a přičtení toho znaku, o který se okénko posunulo. Pokud tedy máme k dispozici aritmetické operace v konstantním čase, zvládneme konstantně přepočítávat i hashovací funkci.

Tato hashovací funkce se dokonce nejen hezky počítá, ale dokonce se i opravdu „hezky“ chová (tedy „rozumně“ náhodně), pokud zvolíme vhodné p. To bychom měli zvolit tak, aby bylo rozhodně nesoudělné s N – tedy $NSD(p, N) = 1$. Aby se nám navíc dobře projevilo modulo obsažené v hashovací funkci, mělo by být p relativně velké (lze dopočítat, že optimum je mezi $2/3 \cdot N$ a $3/4 \cdot N$). S takto zvoleným p se tato hashovací funkce chová velmi příznivě a v praxi má celý algoritmus takřka lineární časovou složitost (průměrnou).

Hledání více řetězců najednou

Nyní si zahrajeme tutéž hru, ovšem v trochu složitějších kulisách. Podíváme se na algoritmus, který si poradí i s více než jednou jehlou. Mějme tedy jehly $\iota_1 \ldots \iota_n$, a jejich délky $J_i = |\iota_i|$. Dále budeme potřebovat seno σ délky $S = |\sigma|$.

Předtím, než se pustíme do vlastního vyhledávacího algoritmu, možná bychom si měli ujasnit, co vlastně bude jeho výstupem. U problému hledání jedné jehly to bylo jasné – byla to nějaká množina pozic v seně, na kterých začínaly výskyty jehly. Jak tomu ale bude zde? Sice bychom také mohli vrátit pouze množinu pozic, ale my budeme chtít maličko víc. Budeme totiž chtít vědět i to, která jehla se na které pozici vyskytuje. Výstup tedy bude vypadat následovně: $V = \{(i, j) \mid \sigma[i : i + J_j] = \iota_j\}$.

Zde se však skrývá jedna drobná zrada. Budeme se asi muset vzdát naděje, že najdeme algoritmus, jehož složitost je lineární v celkové délce všech jehel a sena. Výstup totiž může být delší než lineární. Může se nám klidně stát, že na jedné pozici v seně se bude vyskytovat více různých jehel – pokud bude jedna jehla prefixem jiné (což jsme nikde nezakázali), tak máme povinnost ohlásit oba výskyty. Vzhledem k tomu budeme hledat takový algoritmus, který bude lineární v délce vstupu plus délce výstupu, což je evidentně to nejlepší, čeho můžeme dosáhnout.

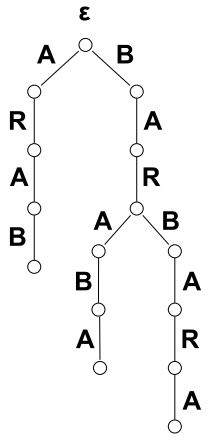
Algoritmus, který si nyní ukážeme, vymysleli někdy v roce 1975 pan Aho a paní Corasicková. Bude to takové zobecnění Knuthova-Morrisova-Prattova algoritmu.

Algoritmus Aho-Corasicková

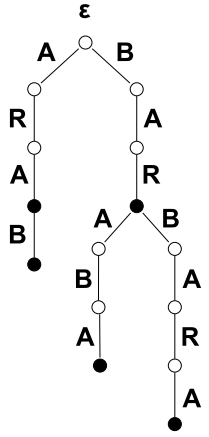
Opět se budeme snažit sestrojit nějaký vyhledávací automat a nějakým způsobem tento automat použít k procházení sena. Podívejme se nejprve na příklad. Budeme chtít vyhledávat tato slova: `ara`, `bar`, `arab`, `baraba`, `barbara`. Mějme tedy těchto pět jehel a rozmysleme si, jak by vypadal nějaký automat, který by tato slova uměl zatím jenom rozpoznávat. Pro jedno slovo automat vypadal jako cesta, zde už to bude strom. (viz obrázek).

Navíc budeme muset do automatu zanést, kde nějaké slovo končí. V původním automatu pro jedno slovo to bylo jednoduché – ono jedno jediné slovo odpovídalo poslednímu vrcholu cesty. Tady se však slova mohou vyskytovat vícekrát a končit nejenom v listech ale i v nějakém vnitřním vrcholu (což se stane tehdy, pokud je jedno hledané slovo prefixem jiného hledaného slova). Formálně to nebudeme dokazovat, ale snadno nahlédneme, že listy stromu odpovídají hledaným slovům, ale opačně to neplatí.

Dále bychom měli do automatu přidat zpětné hrany. Jejich definice bude úplně stejná jako u automatu pro hledání jednoho slova. Jinými slovy z každého stavu půjde



Vyhledávací automat – strom.

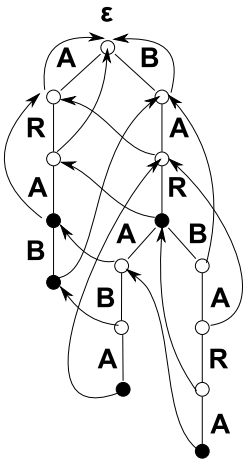


Vyhledávací automat s konci slov.

zpětná hrana do nejdelšího vlastního suffixu, který je stavem. Čili když budeme mít nějaké jméno stavu, budeme se ho snažit co nejméně (ale alespoň o znak) zkrátit zleva, abychom zase dostali jméno stavu. Z kořene – prázdného stavu – pak evidentně žádná zpětná hrana nepovede.

Zbývá nám ještě si rozmyslet, jakým způsobem bude náš automat hlásit výstup. Opět směřujeme k tomu, aby se automat po přečtení nějakého kusu textu nacházel ve stavu odpovídajícímu nejdelšímu možnému suffixu toho textu. Zatímco u hledání jediné jehly bylo hlášení výskytů jednoduché – kdykoliv jsme se dostali na konec „automatové cestičky“ tady to bude opět složitější.

První, co se nabízí, je využít toho, že jsme si označili nějaké vrcholy, kde hledaná slova končí. Co tedy zkusit hlásit výskyt tohoto slova vždy, když přijdeme do nějakého označeného vrcholu? Tento způsob však nefunguje, pokud se uvnitř



Vyhledávací automat se zpětnými hranami.

některé jehly skrývá jehla vnořená. Například po přečtení slova `bara`, nám náš současný automat neříká, že bychom měli nějaké slovo ohlásit, a přitom tam evidentě končí podřetězec `ara`. Stejně tak pokud přečteme `barbara`, už si nevšimneme toho, že tam končí zároveň i `ara`. Pouhé „hlášení teček“ tedy nefunguje.

Dále si můžeme všimnout toho, že všechna slova, která by se měla v daném stavu hlásit, jsou suffixy jména tohoto stavu. Přitom víme, že zpětná hrana jméno stavu zkracuje zleva. Takže speciálně všechny suffixy daného stavu, které jsou také stavy, se dají najít tak, že se vydáme po zpětných hranách do kořene. Nabízí se tedy vždy projít cestu po zpětných hranách až do kořene a hlásit všechny „tečky“. Tento způsob by nám však celý algoritmus značně zpomalil, protože cesta do kořene může být relativně dlouhá, ale teček na ní obvykle bude málo.

Mohli bychom také zkusit si pro každý stav β předpočítat množinu $cache(\beta)$, která by obsahovala všechna slova, která máme hlásit, když se ve stavu β nacházíme. Pokud pak do tohoto stavu vstoupíme, podíváme se na tuto množinu a vypíšeme vše, co v ní je. Výpis nám bude evidentně trvat lineárně k velikosti množiny, celkově tedy lineárně k velikosti výstupu. Problém je ale ten, že jednotlivé cache mohou být hodně velké, takže je nestihneme sestrojit v lineárním čase. (Rozmyslete si příklad slovníku, kdy se to stane.)

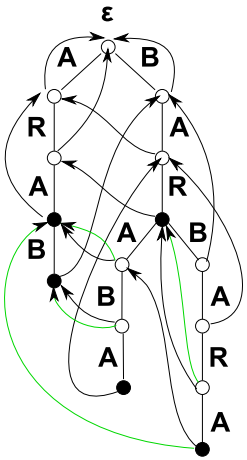
To, co nám ale již opravdu pomůže, bude zavedení zkratek. Všimli jsme si, že po zpětných hranách můžeme projít do kořene a hlásit všechny nalezené tečky. Vadilo nám ale, že se může stát, že budeme dlouho po cestě chodit a při tom žádné tečky nenalézat. Zavedeme si proto zkratky k nejbližší tečce.

Definice (zkratková hrana): Budeme mít tedy nějakou funkci $slovo(\beta)$:= slovo, které končí ve stavu β (nebo $\emptyset$, pokud žádné takové slovo není). Dále pak funkci $out(\beta)$:= nejbližší vrchol dosažitelný po zpětných hranách, čili nejdelší vlastní suffix stavu β, v němž je definovaná funkce *slovo*. Trochu lidštěji řečeno, ten nejbližší dosažitelný vrchol, ve kterém je tečka.

Po přidání těchto zkratkových hran již máme reprezentaci, ve které opravdu umíme v daném stavu vyjmenovat všechna slova, která máme vypsat, a to v čase lineárním s tím, kolik těch slov je.

Definice: Vyhledávací automat sestává ze stromu dopředných hran (vrcholy jsou prefixy jehel, hrany odpovídají rozšíření o písmenko), zpětných hran ($z(\beta) :=$ nejdelší vlastní suffix slova β, který je stavem) a zkratkových hran.

Automat pak bude na našem příkladu vypadat takto (zkratkové hrany jsou znázorněny zeleně):



Vyhledávací automat se zkratkovými hranami.

Nyní už nám zbývá jenom vlastní algoritmus – nejdřív popíšeme algoritmus, který bude hledat pomocí takového automatu, a potom se pustíme do toho, jak se takový automat staví.

Nejprve si nadefinujeme, jak vypadá jeden krok automatu. Bude to vlastně nějaká funkce, která dostane stav a písmenko. Ona nás pak pomocí tohoto písmenka posune po automatu. ($f(\alpha, x)$ bude dopředná hrana ze stavu α označená písmenem x)

Krok (α, x):

1. Dokud $f(\alpha, x) = \emptyset$ & $\alpha \neq$ *kořen:* $\alpha \leftarrow z(\alpha)$.
2. Pokud $f(\alpha, x) \neq \emptyset$: $\alpha \leftarrow f(\alpha, x)$.
3. Vrátíme výsledek.

Hledání:

1. $\alpha \leftarrow$ *kořen*.
2. Pro znaky x ze slova σ:
3. $\quad \alpha \leftarrow Krok(\alpha, x)$.
4. $\quad \beta \leftarrow \alpha$

5. Dokud $\beta \neq \emptyset$:
6. Je-li $slovo(\beta) \neq \emptyset$:
7. Ohlásíme $slovo(\beta)$.
8. $\beta \leftarrow out(\beta)$.

Algoritmus hledání vlastně není nic jiného, než prosté projití po zelených zkratkových hranách ze stavu α, ve kterém právě jsme, a ohlášení všeho, co po cestě najdeme.

V každém okamžiku se automat nachází ve stavu, který odpovídá nejdelšímu možnému suffixu toho, co jsme už přečetli. Důkaz tohoto invariantu je stejný jako u verze automatu pro hledání pouze jedné jehly, neboť vychází pouze z definice zpětných hran. Podobně nahlédneme, že časová složitost vyhledávací procedury je lineární v délce sena plus to, co spotřebujeme na hlášení výskytů. Nejprve na chvíli zapomeneme, že nějaké výskyty hlásíme a spočítáme jenom kroky. Ty mohou vést dopředu a zpátky. Krok dopředu prodlužuje jméno stavu o jedna, krok dozadu zkracuje aspoň o jedna. Tudíž kroků dozadu je maximálně tolik, co kroků dopředu a kroků dopředu je maximálně tolik, kolik je délka sena. Všechny kroky dohromady tedy trvají $\mathcal{O}(S)$. Hlášení výskytů pak trvá $\mathcal{O}(S + |V|)$. Celé hledání tedy trvá lineárně v délce vstupu a výstupu.

Zbývá nám už jen konstrukce automatu. Opět využijeme faktu, že zpětná hrana ze stavu β vede tam, kam by se dostal automat při hledání β bez prvního písmenka. Takže zase chceme něco, jako simulovat výpočet toho automatu na slovech bez prvního písmenka a doufat v to, že si vystačíme s tou částí automatu, kterou jsme už postavili. Tentokrát to však nemůžeme dělat jedno slovo po druhém, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi automatu. Mohlo by se nám tedy stát, že při hledání nějakého slova potřebujeme zpětnou hranu, která vede do jiného slova, které jsme ještě nezkonstruovali. Takže tento postup selže. Můžeme však využít toho, že každá zpětná hrana vede ve stromu alespoň o jednu hladinu výš. Můžeme tak strom konstruovat po hladinách. Lze si to tedy představit tak, že paralelně spustíme vyhledávání všech slov bez prvních písmenek a vždycky uděláme jeden podkrok každého z těch hledání, což nám dá zpětné hrany z dalšího patra stromu.

Konstrukce automatu:

1. Založíme prázdný strom, $r \leftarrow$ jeho kořen.
2. Vložíme do stromu slova $\iota_1 \ldots \iota_n$, nastavíme $slovo(*)$.
3. $z(r) \leftarrow \emptyset$, $out(r) \leftarrow \emptyset$.
4. Založíme frontu F a vložíme do ní syny kořene.
5. $\forall v \in F: \quad z(v) \leftarrow r, out(v) \leftarrow \emptyset$.
6. Dokud $F \neq \emptyset$:
7. Vybereme u z fronty F.
8. Pro všechny syny v vrcholu u:
9. $q \leftarrow Krok(z(u), písmeno\ na\ hraně\ uv)$.
10. $z(v) \leftarrow q$.

11. Pokud $slovo(q) \neq \emptyset$, pak $out(v) \leftarrow q$.
12. Jinak $out(v) \leftarrow out(q)$.
13. Vložíme v do fronty F.

To, že tento algoritmus zkonstruuje zpětné hrany jak má, vyplývá z toho, že neděláme nic jiného, než že spouštíme výpočty po hladinách na všechna hledaná slova bez prvního písmenka. Stejně tak to, že doběhne v lineárním čase, je taktéž důsledkem toho, že efektivně spouštíme všechny tyto výpočty. Jen někdy uděláme najednou krok dvou či více výpočtů (například `araba` a `arbara` se počítají na začátku, dokud jsou stejné, jen jednou). Časová složitost této konstrukce je tedy menší nebo rovna součtu časových složitostí výpočtů nad všemi těmi slovy. To už ale víme, že je lineární v celkové délce těchto slov. Konstrukce automatu tedy trvá nejvýše tolik, co hledání všech ι_i, což je $\mathcal{O}(\sum_i \iota_i)$.

Věta: Algoritmus Aho-Corasicková najde všechny výskyty v čase

$$\mathcal{O}\left(\sum_i \iota_i \;+\; S \;+\; \sharp\textit{výskytů}\right).$$

Ještě se na závěr zamysleme, jak bychom si takový automat ukládali do paměti. Určitě se nám bude hodit si stavy nějak očíslovat (třeba v pořadí, v jakém budou vznikat). Potom funkce pro zpětné a zkratkové hrany mohou být reprezentované polem indexovaným číslem stavu. Funkce *Slovo*, která říká, jaké slovo ve stavu končí, zase může být pole indexované stavem, které nám řekne pořadové číslo slova ve slovníku. Pro dopředné hrany v každém vrcholu pak můžeme mít pole indexované písmenky abecedy, které nám pro každé písmenko řekne, buď že taková hrana není, nebo nám řekne, kam tato hrana vede. Je vidět, že takovéto pole se hodí pro poměrně malé abecedy. Už pro abecedu A-Z bude velikosti 26 a z většiny bude prázdné, takže bychom plýtvali pamětí. V praxi se proto často používá hashovací tabulka. Případně bychom mohli mít i jen jednu velkou společnou hashovací tabulku, která bude reprezentovat funkci celou, ve které budou zahashované dvojice (stav, písmenko). Těchto dvojic je evidentně tolik, kolik hran stromu, čili lineárně s velikostí slovníku, a je to asi nejkompaktnější reprezentace.