1. Vyhledávání v textu

V této kapitole se budeme věnovat příslovečnému hledání jehly v kupce sena. *Seno* bude představovat nějaký text σ délky S. Budeme v něm chtít najít všechny výskyty *jehly* – podřetězce ι délky J.

Kupříkladu v seně `bananas` se jehla `ana` vyskytuje hned dvakrát, přičemž výskyty se překrývají. V seně `anna` se tatáž jehla nevyskytuje vůbec, protože hledáme souvislé podřetězce, a nikoliv vybrané podposloupnosti.

Senem přitom nemusí být jenom obyčejný text. Podobné problémy potkáváme třeba v bioinformatice při zkoumání genetického kódu, nebo v matematice, kde pomocí řetězců kódujeme grafy a jiné kombinatorické struktury.

1.1. Řetězce a abecedy

Aby se nám o řetězcových algoritmech lépe vyprávělo, uděláme si nejprve pořádek v terminologii okolo řetězců.

Definice:

- *Abeceda* Σ je nějaká konečná množina, jejím prvkům budeme říkat *znaky* (někdy též *písmena*).
- Σ^* je množina všech *slov* neboli *řetězců* nad abecedou Σ, což jsou konečné posloupnosti znaků ze Σ.

Příklady: Abeceda může být tvořena třeba písmeny `a` až `z`, bity `0` a `1` nebo nukleotidy `C`, `T`, `A`, `G`. Potkáme ovšem i rozlehlejší abecedy: například mezinárodní znaková sada Unicode má $2^{16} = 65\,536$ znaků, v novějších verzích dokonce 1 114 112 znaků. Ještě extrémnějším způsobem používají řetězce lingvisté: na český text se někdy dívají jako na řetězec nad abecedou, jejíž znaky jsou česká slova.

Velikost abecedy se obvykle považuje za konstantu. My budeme navíc předpokládat, že abeceda je dostatečně malá, abychom si mohli dovolit ukládat do paměti pole indexovaná znakem. Později se tohoto předpokladu zbavíme.

Značení:

- *Slova* budeme značit malými písmenky řecké abecedy α, β, ...
- *Znaky* abecedy označíme malými písmeny latinky x, y, ... Konkrétní znaky budeme psát `psacím strojem`. Znak budeme používat i ve smyslu jednoznakového řetězce.
- *Délka slova* $|\alpha|$ udává, kolika znaky je slovo tvořeno.
- *Prázdné slovo* značíme písmenem ε, je to jediné slovo délky 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 do $|\alpha| - 1$.

- $\alpha[k : \ell]$ je *podslovo* začínající k-tým znakem a končící těsně před ℓ-tým. Tedy $\alpha[k : \ell] = \alpha[k]\alpha[k+1]\ldots\alpha[\ell-1]$. Pokud $k \geq \ell$, je podslovo prázdné. Pokud některou z mezí vynecháme, míní se $k = 0$ nebo $\ell = |\alpha|$.
- $\alpha[: \ell]$ je *prefix* (předpona) tvořený prvními ℓ znaky řetězce.
- $\alpha[k :]$ je *suffix* (přípona) od k-tého znaku do konce řetězce.
- $\alpha[:] = \alpha$.

Dodejme ještě, že každé slovo je podslovem sebe sama a prázdné slovo je podslovem každého slova. Pokud budeme hovořit o *vlastním* podslovu, budeme tím myslet podslovo různé od celého slova. Analogicky pro prefixy a suffixy.

1.2. Knuthův-Morrisův-Prattův algoritmus

Vraťme se nyní zpět k původnímu problému hledání podřetězců. Na vstupu jsme dostali seno σ a jehlu ι. Na výstupu chceme oznámit všechny výskyty; snadno je popíšeme například množinou všech indexů k takových, že $\sigma[k : k + |\iota|] = \iota$.

Kdybychom postupovali podle definice, zkoušeli bychom všechny možné pozice v seně a pro každou z nich otestovali, zda tam nezačíná nějaký výskyt jehly. To je funkční, nicméně pomalé: možných začátků je řádově S, pro každý z nich porovnáváme až J znaků jehly. Celková časová složitost je tedy $\Theta(JS)$.

Zkusme jiný přístup: nalezneme v seně první znak jehly a od tohoto místa budeme porovnávat další znaky. Pokud se přestanou shodovat, přepneme opět na hledání prvního znaku. Jenže odkud? Pokud od místa, kde nastala neshoda, selže to třeba při hledání jehly `kokos` v seně `clanekokokosu` – neshoda nastane za `koko` a zbylý `kos` nás neuspokojí. Nebo se můžeme vrátit až k výskytu prvního znaku a pokračovat těsně za ním, ale to zase trvá $\Theta(JS)$.

Nyní ukážeme algoritmus, který je o trochu složitější, ale nalezne všechny výskyty v čase $\Theta(J + S)$. Později ho zobecníme, aby uměl hledat více různých jehel najednou.

Inkrementální algoritmus

Na hledání podřetězce půjdeme *inkrementálně*. Tím se obecně myslí, že chceme postupně rozšiřovat vstup a přepočítávat, jak se změní výstup. V našem případě vždy přidáme další znak na konec sena a započítáme případný nový výskyt jehly, který končí tímto znakem.

Abychom toho dosáhli, budeme si průběžně udržovat informaci o tom, jakým nejdelším prefixem jehly končí zatím přečtená část sena. Tomu budeme říkat *stav algoritmu*. A jakmile bude tento prefix roven celé jehle, ohlásíme výskyt.

V našem „kokosovém“ příkladě se tedy po přečtení sena `clanekoko` nacházíme ve stavu `koko`, následují stavy `kok`, `koko` a `kokos`.

Představme si nyní obecně, že jsme přečetli řetězec σ, který končil stavem α. Pak vstup rozšíříme o znak x na σx. V jakém stavu se teď máme nacházet? Pokud to nebude prázdný řetězec, musí končit na x, tedy ho můžeme napsat ve tvaru $\alpha' x$.

Všimneme si, že α' *musí být suffixem slova* α*:* Jelikož $\alpha' x$ je prefix jehly, je α' také prefix jehly. A protože $\alpha' x$ je suffixem σx, musí α' být suffixem σ. Tedy jak α, tak α' jsou suffixy slova σ, které jsou současně prefixy jehly. Ovšem stav α jsme vybrali jako nejdelší slovo s touto vlastností, takže α' musí být nejvýše tak dlouhé, a tedy je suffixem α.

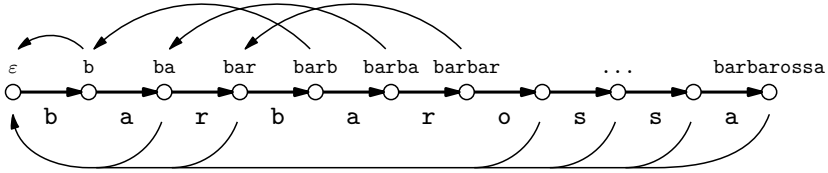
Stačilo by proto probrat všechny suffixy slova α, které jsou prefixem jehly, a vybrat z nich nejdelší, který po rozšíření o znak x stále je prefixem jehly.

Abychom ale nemuseli suffixy procházet všechny, předpočítáme si *zpětnou funkci* z. Ta nám pro každý prefix jehly řekne, jaký je jeho nejdelší vlastní suffix, který je opět prefixem jehly. To nám umožní procházet rovnou kandidáty na nový stav: probereme řetězce α, $z(\alpha)$, $z(z(\alpha))$, ... a použijeme první z nich, který lze rozšířit o znak x. Pokud nepůjde rozšířit ani jeden z těchto kandidátů, novým stavem bude prázdný řetězec.

Na této myšlence je založen následující algoritmus, objevený v roce 1974 Donaldem Knuthem, Jamesem Morrisem a Vaughanem Prattem.

Knuthův-Morrisův-Prattův algoritmus

Algoritmus se opírá o *vyhledávací automat.* To je orientovaný graf, jehož vrcholy (*stavy* automatu) odpovídají prefixům jehly. Vrcholy jsou spojeny hranami dvou druhů: *dopředné* popisují rozšíření prefixu přidáním jednoho písmene, *zpětné* vedou podle zpětné funkce, čili z každého stavu do jeho nejdelšího vlastního suffixu, který je opět stavem.



Obr. 1.1: Vyhledávací automat pro slovo `barbarossa`

Reprezentace automatu bude přímočará: stavy očíslujeme od 0 do J, dopředná hrana povede vždy ze stavu s do $s+1$ a bude odpovídat rozšíření prefixu o příslušný znak jehly, tedy o $\iota[s]$. Zpětné hrany si zapamatujeme v poli Z: prvek $Z[s]$ bude říkat číslo stavu, do nějž vede zpětná hrana ze stavu s, případně bude nedefinované, pokud taková hrana neexistuje.

Kdybychom takový automat měli, mohli bychom pomocí něj inkrementální algoritmus z předchozího oddílu popsat následovně:

Procedura KmpKrok (jeden krok automatu)
Vstup: Jsme ve stavu s, přečetli jsme znak x

1. Dokud $\iota[s] \neq x \ \wedge \ s \neq 0 : s \leftarrow Z[s]$. ◁ *zpětné hrany*

2. Pokud $\iota[s] = x$, pak $s \leftarrow s + 1$. ◁ *dopředná hrana*

Výstup: Nový stav s

Algoritmus KmpHledej (spuštění automatu na řetězec σ)

Vstup: Seno σ, zkonstruovaný automat

1. $s \leftarrow 0$
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow \text{KmpKrok}(s, x)$
4. $\quad$ Pokud $s = J$, ohlásíme výskyt.

Invariant: Stav algoritmu s v každém okamžiku říká, jaký nejdelší prefix jehly je suffixem zatím přečtené části sena. (To už víme z úvah o inkrementálním algoritmu.)

Důsledek: Algoritmus ohlásí všechny výskyty. Pokud jsme právě přečetli poslední znak nějakého výskytu, je celá jehla suffixem zatím přečtené části sena, takže se musíme nacházet v posledním stavu.

Jen musíme opravit drobnou chybu – těsně poté, co ohlásíme výskyt, se algoritmus zeptá na dopřednou hranu z posledního stavu. Ta přeci neexistuje! Napravíme to jednoduše: přidáme fiktivní dopřednou hranu, na níž je napsán znak odlišný od všech skutečných znaků. Tím zajistíme, že se po této hraně nikdy nevydáme. Stačí tedy vhodně dodefinovat $\iota[J]$.[⟨1⟩]

Lemma: Funkce KmpHledej běží v čase $\Theta(S)$.

Důkaz: Výpočet funkce můžeme rozdělit na průchody dopřednými a zpětnými hranami. S dopřednými je to snadné – pro každý z S znaků sena projdeme po nejvýše jedné dopředné hraně. To o zpětných hranách neplatí, ale pomůže nám, že každá dopředná hrana vede o právě 1 stav doprava a každá zpětná o aspoň 1 stav doleva. Proto je všech průchodů po zpětných hranách nejvýše tolik, kolik jsme prošli dopředných hran, takže také nejvýše S. □

Mimochodem, předchozí lemma nám vlastně říká, že jeden krok automatu má konstantní amortizovanou složitost. A důkaz v sobě skrývá přímočaré použití potenciálové metody z oddílu **??**: roli potenciálu zde hraje číslo stavu.

Konstrukce automatu

Hledání tedy pracuje v lineárním čase, zbývá domyslet, jak v lineárním čase sestrojit automat. Stavy a dopředné hrany získáme triviálně, se zpětnými budeme mít trochu práce.

Podnikneme myšlenkový pokus: Představme si, že automat už máme hotový, ale nevidíme, jak vypadá uvnitř. Chtěli bychom zjistit, jak v něm vedou zpětné hrany, ovšem jediné, co umíme, je spustit automat na nějaký řetězec a zjistit, v jakém stavu skončil.

⟨1⟩ V jazyce C můžeme zneužít toho, že každý řetězec je ukončen znakem s nulovým kódem.

Tvrdíme, že pro zjištění zpětné hrany ze stavu α stačí automatu předložit řetězec $\alpha[1:]$. Definice zpětné funkce je totiž nápadně podobná invariantu, který jsme o funkci KmpHledej dokázali. Obojí hovoří o nejdelším suffixu daného slova, který je prefixem jehly. Jediný rozdíl je v tom, že v případě zpětné funkce uvažujeme pouze vlastní suffixy, zatímco invariant připouští i ty nevlastní. To ovšem snadno vyřešíme „ukousnutím“ prvního znaku jména stavu.

Pokud bychom chtěli objevit všechny zpětné hrany, stačilo by automat spouštět postupně na řetězce $\iota[1:1]$, $\iota[1:2]$, $\iota[1:3]$, atd. Jelikož funkce KmpHledej je lineární, stálo by nás to dohromady $\mathcal{O}(J^2)$. Pokud si ale všimneme, že každý ze zmíněných řetězců je prefixem toho následujícího, je jasné, že stačí spustit automat jen jednou na řetězec $\iota[1:]$ a jen zaznamenávat, kterými stavy jsme prošli.

To je zajímavé pozorování, řeknete si, ale jak nám pomůže ke konstrukci automatu, když samo hotový automat potřebuje? Pomůže pěkný trik: pokud hledáme zpětnou hranu z i-tého stavu, spouštíme automat na slovo délky $i-1$, takže se můžeme dostat pouze do prvních $i-1$ stavů a vůbec nám nevadí, že v tom i-tém ještě není zpětná hrana hotova.⟨2⟩

Při konstrukci automatu tedy nejdříve sestrojíme dopředné hrany, načež rozpracovaný automat spustíme na řetězec $\iota[1:]$ a podle toho, jakými stavy bude procházet, doplníme zpětné hrany. Jak už víme, vyhledávání má lineární složitost, takže celá konstrukce potrvá $\Theta(J)$.

Hotový algoritmus pro konstrukci automatu můžeme zapsat následovně:

Algoritmus KmpKonstrukce
Vstup: Jehla ι délky J

1. $Z[0] \leftarrow$ *nedefinováno*, $Z[1] \leftarrow 0$
2. $s \leftarrow 0$
3. Pro $i = 2, \ldots, J$:
4. $\quad s \leftarrow \textsc{KmpKrok}(s, \iota[i-1])$
5. $\quad Z[i] \leftarrow s$

Výstup: Pole zpětných hran Z

Výsledky můžeme shrnout do následující věty:

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

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

⟨2⟩ Konstruovat nějaký objekt pomocí téhož objektu je osvědčený postup, který si už vysloužil svůj vlastní název. V angličtině se mu říká *bootstrapping* a z toho také vzniklo bootování počítačů, protože při něm operační systém zavádí do paměti sám sebe. Kde se toto slovo vzalo? Bootstrap znamená česky *štruple* – to je takové to očko na patě boty, které usnadňuje nazouvání. A v jednom z příběhů o baronu Prášilovi slyšíme barona vyprávět, jak se uvíznuv v bažině zachránil tím, že se vytáhl za štruple. Krásný popis bootování, není-liž pravda?

Cvičení

1. Naivní algoritmus, který zkouší všechny možné začátky jehly v seně a vždy porovnává řetězce, má časovou složitost $\mathcal{O}(JS)$. Může být opravdu tak pomalý, uvážíme-li, že porovnávání řetězců skončí, jakmile najde první neshodu? Sestrojte vstup, na kterém algoritmus poběží $\Theta(JS)$ kroků, přestože nic nenajde.
2. *Rotací* řetězce α o K pozic nazýváme řetězec $\alpha[K :]\alpha[: K]$. Jak o dvou řetězcích zjistit, zda je jeden rotací druhého?
3. Jak v lineárním čase zrotovat řetězec, dostačuje-li paměť počítače jen na uložení jednoho řetězce a $\mathcal{O}(1)$ pomocných proměnných?
4.* Navrhněte algoritmus, který v lineárním čase nalezne tu z rotací zadaného řetězce, jež je lexikograficky minimální.
5. Je dáno slovo. Chceme nalézt jeho nejdelší prefix, který je současně suffixem.

1.3. Více řetězců najednou: algoritmus Aho-Corasicková

Nyní si zahrajeme tutéž hru v trochu složitějších kulisách. Tentokrát bude jehel vícero: $\iota_1, \ldots, \iota_N$, jejich délky označíme $J_i = |\iota_i|$. Dostaneme nějaké seno σ délky S a chceme nalézt všechny výskyty jehel v seně.

Opět si nejdřív musíme ujasnit, co má být výstupem. Dokud byla jehla jedna jediná, bylo to zřejmé – chtěli jsme nalézt množinu všech pozic v seně, na kterých začínaly výskyty jehly. Jak tomu bude zde? Chceme se dozvědět, která jehla se vyskytuje na které pozici. Jinými slovy vypsat všechny dvojice (k, i) takové, že $\sigma[k : k + J_i] = \iota_i$.

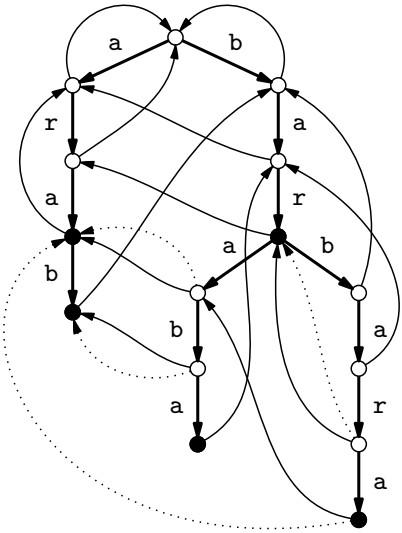
Těchto dvojic může být poměrně hodně. Pokud je totiž jedna jehla suffixem druhé, na jedné pozici v seně mohou končit výskyty obou. Celková velikost výstupu tak může být větší než lineární v délce vstupu (viz cvičení 1). Budeme proto hledat algoritmus, který bude lineární v délce vstupu plus délce výstupu, což je evidentně to nejlepší, čeho můžeme dosáhnout.

Algoritmus, který si nyní ukážeme, objevili v roce 1975 Alfred Aho a Margaret Corasicková. Je elegantním zobecněním Knuthova-Morrisova-Prattova algoritmu pro více řetězců.

Opět se budeme snažit sestrojit *vyhledávací automat,* jehož stavy budou odpovídat prefixům jehel a dopředné hrany budou popisovat rozšiřování prefixů o jeden znak. Hrany tedy budou tvořit strom orientovaný směrem od kořene (písmenkový strom pro daný slovník, který už jsme potkali v oddílu **??**).

Každý list stromu bude odpovídat některé z jehel, ale jak je vidět na obrázku, některé jehly se mohou vyskytovat i ve vnitřních vrcholech (pokud je jedna jehla prefixem jiné). Výskyty jehel ve stromu si tedy nějak označíme, příslušným stavům budeme říkat *koncové.*

Dále potřebujeme zpětné hrany (na obrázku tenké šipky). Jejich definice bude úplně stejná jako u automatu KMP. Z každého stavu půjde zpětná hrana do jeho nejdelšího vlastního suffixu, který je také stavem. Čili se budeme snažit jméno



Obr. 1.2: Vyhledávací automat pro
slova `ara`, `bar`, `arab`, `baraba`, `barbara`

stavu zkracovat zleva tak dlouho, než dostaneme jméno dalšího stavu. Z kořene – prázdného stavu – pak evidentně žádná zpětná hrana nepovede.

Funkce pro hledání v seně bude vypadat stejně jako u KMP: začne v počátečním stavu (to je kořen stromu) a postupně bude rozšiřovat seno o další písmenka. Pokaždé zkusí jít dopřednou hranou a pokud to nepůjde, bude se vracet po zpětných hranách. Přitom se buďto dostane do vrcholu, kde vhodná dopředná hrana existuje, nebo se nový znak nehodí ani v kořeni, a tehdy je zahozen.

Stejně jako u KMP nahlédneme, že procházení sena trvá $\Theta(S)$ a že platí analogický invariant: v každém okamžiku se nacházíme ve stavu, který odpovídá nejdelšímu suffixu zatím přečteného sena, který je prefixem některé jehly.

Hlášení výskytů

Kdy ohlásíme výskyt jehly? U KMP to bylo snadné: kdykoliv jsme dospěli do posledního stavu, znamenalo to nalezení jehly. Nabízí se hlásit výskyt, kdykoliv dojdeme do stavu označeného jako koncový. To ale nefunguje: pokud náš ukázkový automat přečte seno `bara`, skončí ve stavu `bara`, který není koncový, a přitom by zde měl ohlásit výskyt jehly `ara`. Stejně tak přečteme-li `barbara`, nevšimneme si, že na témže místě končí i `ara`.

Platí ale, že všechna slova, která bychom měli v daném stavu ohlásit, jsou suffixy jména tohoto stavu. Mohli bychom se tedy vydat po zpětných hranách až do kořene a kdykoliv projdeme přes koncový vrchol, ohlásit výskyt. To ovšem trvá příliš dlouho – jistě by se stávalo, že bychom podnikli dlouhou cestu do kořene a nenašli na ní vůbec nic.

Další, co se nabízí, je předpočítat si pro každý stav β množinu slov $M(\beta)$, jejichž výskyty máme v tomto stavu hlásit. To by fungovalo, ale existují množiny jehel, pro které bude celková velikost množin $M(\beta)$ superlineární (viz cvičení 3). Museli bychom se tedy vzdát lákavé možnosti stavby automatu v lineárním čase.

Jak to tedy vyřešíme? Zavedeme zkratky (na obrázku vyznačeny tečkovaně):

Definice: *Zkratková hrana* ze stavu α vede do nejbližšího koncového stavu $\zeta(\alpha)$ dosažitelného z α po zpětných hranách (a různého od α).

Jinými slovy, zkratka $\zeta(\alpha)$ nám řekne, jaký je nejdelší vlastní suffix slova α, který je jehlou. Pokud takový suffix neexistuje, žádná zkratková hrana ze stavu α nepovede. Pomocí zkratkových hran můžeme snadno vyjmenovat všechny výskyty. Budeme postupovat stejně, jako bychom procházeli po všech zpětných hranách, jen budeme dlouhé úseky zpětných hran, na nichž není nic k hlášení, přeskakovat v konstantním čase.

Reprezentace automatu

Vyhledávací automat sestává ze stromu dopředných hran, ze zpětných hran a ze zkratkových hran. Rozmysleme si, jak vše uložit do paměti. Stavy očíslujeme, třeba podle toho jak vznikaly, a pro každý stav s si budeme pamatovat:

- *Zpět*(s) – číslo stavu, kam vede zpětná hrana (nebo $\emptyset$, pokud ze stavu s žádná nevede),
- *Zkratka*(s) – kam vede zkratková hrana (obdobně),
- *Slovo*(s) – zda tu končí nějaké slovo (a pokud ano, tak které),
- *Dopředu*(s, x) – kam vede dopředná hrana označená písmenem x (pro malé abecedy si to můžeme pamatovat v poli, pro velké viz cvičení 5).

Celý algoritmus pro zpracování sena automatem pak bude vypadat takto:

Procedura AcKrok (jeden krok automatu)
Vstup: Jsme ve stavu s, přečetli jsme znak x

1. Dokud *Dopředu*$(s, x) = \emptyset \ \wedge \ s \neq$ *kořen*: $s \leftarrow$ *Zpět*(s).
2. Pokud *Dopředu*$(s, x) \neq \emptyset$: $s \leftarrow$ *Dopředu*(s, x).

Výstup: Nový stav s

Algoritmus AcHledej (spuštění automatu na daný řetězec)
Vstup: Seno σ, zkonstruovaný automat

1. $s \leftarrow$ *kořen*
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow$ AcKrok(s, x)
4. $\quad j \leftarrow s$
5. $\quad$ Dokud $j \neq \emptyset$:
6. $\quad\quad$ Je-li *Slovo*$(j) \neq \emptyset$:
7. $\quad\quad\quad$ Ohlásíme *Slovo*(j).

8. $j \leftarrow \mathit{Zkratka}(j)$

Stejným argumentem jako u KMP zdůvodníme, že všechny kroky automatu dohromady trvají $\Theta(S)$. Mimo to ještě hlásíme výskyty, což trvá $\Theta(\textit{počet výskytů})$. Zbývá ukázat, jak automat sestrojit.

Konstrukce automatu

Opět se inspirujeme algoritmem KMP a nahlédneme, že zpětná hrana ze stavu β vede tam, kam by se automat dostal při hledání slova β bez prvního znaku. Chtěli bychom tedy začít sestrojením dopředných hran a pak spouštěním ještě nehotového automatu na jednotlivé jehly doplňovat zpětné hrany, doufajíce, že si vystačíme s už sestrojenou částí automatu.

Kdybychom však automat spouštěli na jednu jehlu po druhé, dostali bychom se do úzkých, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi stromu. Mohlo by se nám tedy stát, že bychom při hledání potřebovali zpětnou hranu, která dosud nebyla vytvořena.

Budeme tedy zpětné hrany raději konstruovat po hladinách. Každá taková hrana vede alespoň o jednu hladinu výš, takže se při hledání vždy budeme pohybovat po té části stromu, která už je bezpečně hotová. Můžeme si představit, že paralelně spustíme vyhledávání všech slov bez prvních písmenek a vždy uděláme jeden krok každého z těchto hledání, což nám dá zpětné hrany v dalším patře stromu.

Navíc kdykoliv vytvoříme zpětnou hranu, sestrojíme také zkratkovou hranu z téhož vrcholu: Pokud vede zpětná hrana ze stavu s do stavu z a $\mathit{Slovo}(z)$ je definováno, musí vést zkratka z s také do z. Pokud v z žádné slovo nekončí, musí zkratka z s vést do téhož vrcholu, kam vede zkratka ze z.

Algoritmus AcKonstrukce

Vstup: Slova $\iota_1, \ldots, \iota_n$

1. Založíme strom, který obsahuje pouze kořen r.
2. Vložíme do stromu slova $\iota_1 \ldots \iota_n$, nastavíme *Slovo* ve všech stavech.
3. $\mathit{Zpět}(r) \leftarrow \emptyset$, $\mathit{Zkratka}(r) \leftarrow \emptyset$
4. Založíme frontu F a vložíme do ní syny kořene.
5. Pro všechny syny s kořene: $\mathit{Zpět}(s) \leftarrow r$, $\mathit{Zkratka}(s) \leftarrow \emptyset$.
6. Dokud $F \neq \emptyset$:
7. Vybereme i z fronty F.
8. Pro všechny syny s vrcholu i:
9. $z \leftarrow \textsc{AcKrok}(\mathit{Zpět}(i), \text{písmeno na hraně } is)$
10. $\mathit{Zpět}(s) \leftarrow z$
11. Pokud $\mathit{Slovo}(z) \neq \emptyset$: $\mathit{Zkratka}(s) \leftarrow z$.
12. Jinak $\mathit{Zkratka}(s) \leftarrow \mathit{Zkratka}(z)$.
13. Vložíme s do fronty F.

Výstup: Strom, pole *Slovo*, *Zpět* a *Zkratka*

Pro rozbor časové složitosti si uvědomíme, že konstrukce zpětných hran hledá všechny jehly, jen kroky jednotlivých hledání vhodným způsobem střídá (jakoby je prováděla paralelně). Časovou složitost tedy můžeme shora omezit součtem složitostí hledání jehel, což, jak už víme, je lineární v délce jehel.

Chování celého algoritmu shrneme do následující věty:

Věta: Algoritmus Aho-Corasicková najde všechny výskyty v čase $\Theta\left(\sum_i J_i + S + V\right)$, kde $J_1, \ldots, J_n$ jsou délky jednotlivých jehel, S je délka sena a V počet výskytů.

Cvičení

1. Nalezněte příklad jehel a sena, v němž je asymptoticky více než lineární počet výskytů. Přesněji řečeno ukažte, že pro každé n existuje vstup, v němž je součet délek jehel a sena $\Theta(n)$ a počet výskytů není $\mathcal{O}(n)$.
2. Uvažujme zjednodušený algoritmus AC, který nepoužívá zkratkové hrany a vždy projde po zpětných hranách až do kořene. Ukažte vhodnými příklady vstupů, že tento algoritmus je asymptoticky pomalejší.
3. Jednoduchý způsob, jak si poradit s hlášením výskytů, je předpočítat si pro každý stav s množinu $M(s)$ slov k ohlášení. Dokažte, že tyto množiny není možné sestrojit v lineárním čase s velikostí slovníku, protože součet jejich velikostí může být pro některé vstupy superlineární.
4. Rozmyslete si, že množiny $M(s)$ z předchozího příkladu by bylo možné reprezentovat jako srůstající spojové seznamy – tedy takové, kde si každý prvek pamatuje ukazatel na svého následníka, který ovšem může ležet v jiném seznamu. Přesvědčte se, že námi zavedené zkratkové hrany lze interpretovat jako ukazatele ve srůstajících seznamech.
5. Upravte algoritmy z této kapitoly, aby si poradily s velkými abecedami.
6. Co kdybychom chtěli pro každou pozici v seně hlásit jenom jeden výskyt jehly? Mohl by to být třeba ten nejdelší, který na dané pozici končí. Ukažte, jak to zařídit bez vyjmenování všech výskytů. Jak by se situace změnila, kdybychom místo nejdelšího hledali nejkratší?
7. Mějme seno a jehly. Popište algoritmus, který v lineárním čase pro každou jehlu spočítá, kolikrát se v seně vyskytuje. Časová složitost by neměla záviset na počtu výskytů – ten, jak už víme, může být superlineární.
8. Cenzor dostane množinu zakázaných podřetězců a text. Vždy najde nejlevější výskyt zakázaného podřetězce v textu (s nejlevějším koncem; pokud jich je více, tak nejdelší takový), vystřihne ho a postup opakuje. Ukažte, jak text cenzurovat v lineárním čase. Chování algoritmu si vyzkoušejte na textu $\mathtt{a}^n\mathtt{b}^n$ a zakázaných slovech $\mathtt{a}^{n+1}$, $\mathtt{b}$.

1.4. Rabinův-Karpův algoritmus

Na závěr ukážeme ještě jeden přístup k hledání jehly v seně, založený na hešování. Časová složitost v nejhorším případě sice bude srovnatelná s hledáním hrubou silou, ale v průměru bude lineární a v praxi tento algoritmus často překoná KMP.

Představme si, že máme seno délky S a jehlu délky J. Pořídíme si nějakou hešovací funkci H, která J-ticím znaků přiřazuje čísla z množiny $\{0, \ldots, N-1\}$ pro nějaké dost velké N. Budeme posouvat okénko délky J po seně, pro každou jeho polohu si spočteme heš znaků uvnitř okénka, porovnáme s hešem jehly a pokud se rovnají, porovnáme okénko s jehlou znak po znaku.

Pokud je hešovací funkce „kvalitní“, málokdy se stane, že by se heše rovnaly, takže místo času $\Theta(J)$ na porovnávání řetězců si vystačíme s porovnáním hešů v konstantním čase. Jenže ouha, čas $\Theta(J)$ potřebujeme i na vypočtení heše pro každou polohu okénka. Jak z toho ven?

Pořídíme si hešovací funkci, kterou lze při posunutí okénka o pozici doprava v konstantním čase přepočítat. Tyto požadavky splňuje třeba polynom

$$H(x_1, \ldots, x_J) = (x_1 P^{J-1} + x_2 P^{J-2} + \ldots + x_{J-1} P^1 + x_J P^0) \bmod N,$$

přičemž písmena považujeme za přirozená čísla a P je nějaká vhodná konstanta – potřebujeme, aby byla nesoudělná s N a aby P^J bylo řádově větší než N. Posuneme-li nyní okénko z $x_1, \ldots, x_J$ na $x_2, \ldots, x_{J+1}$, heš se změní takto:

$$\begin{aligned} H(x_2, \ldots, x_{J+1}) &= (x_2 P^{J-1} + x_3 P^{J-2} + \ldots + x_J P^1 + x_{J+1} P^0) \bmod N \\ &= (P \cdot H(x_1, \ldots, x_J) - x_1 P^J + x_{J+1}) \bmod N. \end{aligned}$$

Pokud si mocninu P^J předpočítáme, proběhne aktualizace heše v konstantním čase.

Celý algoritmus pak bude vypadat následovně:

Algoritmus RabinKarp
Vstup: Jehla ι délky J, seno σ délky S

1. $j \leftarrow H(\iota)$ ◁ *heš jehly*
2. $h \leftarrow H(\sigma[:J])$ ◁ *heš první pozice okénka*
3. Zvolíme P a N a předpočítáme $P^J \bmod N$.
4. Pro i od 0 do $S - J$: ◁ *možné pozice okénka*
5. Je-li $h = j$:
6. Pokud $\sigma[i : i+J] = \iota$, ohlásíme výskyt na pozici i.
7. Pokud $i < S - J$: ◁ *přepočítáme heš*
8. $h \leftarrow (P \cdot h - \sigma[i] \cdot P^J + \sigma[i+J]) \bmod N$

Pojďme prozkoumat složitost algoritmu. Inicializace algoritmu a počítání hešů okének trvají celkem $\mathcal{O}(J+S)$. Pro každou polohu okénka ovšem můžeme strávit čas $\mathcal{O}(J)$ porovnáváním řetězců. To může celkem trvat až $\mathcal{O}(JS)$. Abychom ukázali, že průměr je lepší, odhadneme pravděpodobnost porovnání.

Pokud nastane výskyt, určitě porovnáváme. Nenastane-li, heš jehly se shoduje s hešem okénka s pravděpodobností $1/N$ (za předpokladu dokonale náhodného chování hešovací funkce, což jsme o té naší nedokázali; blíže viz cvičení 1).

V průměru tedy spotřebujeme čas $\mathcal{O}(J + S + VJ + S/N \cdot J)$, kde V je počet nalezených výskytů. Pokud nám bude stačit najít první výskyt a zvolíme $N > SJ$, algoritmus poběží v průměrném čase $\mathcal{O}(J + S)$.

Dodejme, že tento algoritmus objevili v roce 1987 Richard Karp a Michael Rabin. Později se podobná myšlenka stala základem metod na detekci podobnosti souborů, které můžete ochutnat ve cvičení 2.

Cvičení

1. Polynomiální hešovací funkce nejsou dokonale náhodné, ale kdybychom zvolili prvočíselné N a náhodné P, mohli bychom využít poznatků o univerzálním hešování z oddílu **??**. Spočítejte pomocí cvičení **??**, kolik v průměru nastane kolizí, a pomocí toho stanovte průměrnou časovou složitost vyhledávání.
2. Bob a Bobek si povídají po telefonu a pojali podezření, že každý z nich používá trochu jinou verzi softwaru pro kouzelný klobouk. Bob navrhuje rozdělit soubor s programem na 32 KB bloky, každý z nich zahešovat do 64-bitového čísla a výsledky si říci. Bobek oponuje, že tak by snadno poznali pár změněných bytů, ale vložení jediného bytu by mohlo změnit všechny heše. Poradíme jim, aby soubor prošli „okénkovou“ hešovací funkcí a kdykoliv je nejnižších B bitů výsledku nulových, začali nový blok. Rozmyslete si, že toto dělení je odolné i proti vkládání a mazání bytů. Jak zvolit B a parametry hešovací funkce, aby průměrná velikost bloku zůstala 32 KB?

1.5. Další cvičení

1. Jak zjistit, zda je zadané slovo α periodické? Tím myslíme zda existuje slovo β a číslo $k > 1$ takové, že $\alpha = \beta^k$ (zřetězení k kopií řetězce β).

2.* Navrhněte datovou strukturu pro dynamické vyhledávání v textu. Jehla je pevná, v seně lze průběžně měnit jednotlivé znaky a struktura odpovídá, zda se v seně právě vyskytuje jehla.

3. *Pestrý* budeme říkat takovému řetězci, jehož všechny rotace jsou navzájem různé. Kolik existuje pestrých řetězců v Σ^n pro konečnou abecedu Σ a prvočíslo n?

4.** Vyřešte předchozí cvičení pro obecné n.

5.* Substituční šifra funguje tak, že zpermutujeme znaky abecedy: například permutací abecedy `abcdeo` na `dacebo` zašifrujeme slovo `abadcode` na `dadecoeb`. Zašifrovaný text je méně srozumitelný, ale například vyzradí, kde v originálu byly stejné znaky a kde různé. Buď dáno seno zašifrované substituční šifrou a nezašifrovaná jehla. Najděte všechny možné výskyty jehly v originálním seně (tedy takové pozice v seně, pro něž existuje permutace abecedy, která přeloží jehlu na příslušný kousek sena).

6. Definujme Fibonacciho slova takto: $F_0 = \mathtt{a}$, $F_1 = \mathtt{b}$, $F_{n+2} = F_n F_{n+1}$. Jak v zadaném řetězci nad abecedou $\{\mathtt{a}, \mathtt{b}\}$ najít nejdelší Fibonacciho podslovo?

7.* Pokračujme v předchozím cvičení. Dostaneme řetězec nad nějakou obecnou abecedou, chceme nalézt jeho nejdelší podřetězec, který je isomorfní s nějakým Fibonacciho slovem (liší se pouze substitucí jiných znaků za `a` a `b`).

8.* Je dán text a číslo K. Jak zjistit, který podřetězec délky K se v textu vyskytuje nejčastěji?

9.* Opět je dán text, tentokrát hledáme nejdelší podřetězec, který se vyskytuje alespoň dvakrát.

10.* Ukažte, jak pro dané dva řetězce najít jejich nejdelší společný podřetězec.