Vyhledávání v textu algoritmem Aho-Corasick

Aleš Novák, ADS II., 13.1.2012

Stojíme-li před úkolem vyhledat výskyty N slov celkové délky M v textu délky L, můžeme použít „hloupý“ algoritmus, který pro každý znak vstupu vyzkouší, zda od něj nezačíná některé z hledaných slov. Každý znak vstupu se v nejhorším případě může porovnávat se všemi znaky hledaných slov, tento algoritmus má tedy časovou složitost $\mathcal{O}(L * M)$. Chytřejší algoritmus, který poprvé popsal Alfred V. Aho a Margaret J. Corasick (-ová), používá konečný stavový automat sestavený z vyhledávaných slov. Každý znak vstupu se čte pouze jednou a způsobí změnu stavu automatu.

Základem automatu bude *trie*, tedy prefixový strom. Hrany tohoto základu budou odpovídat písmenům hledaných slov. Stavy obsahují příznak, že se jedná o stav, ve kterém končí nějaké slovo. Kořen stromu představuje *nulový* stav. Ostatní stavy představují prefix jednoho nebo více slov.

Stojím-li v nějakém stavu automatu, a načtu ze vstupu písmeno, na které neexistuje přechod, je jasné, že musím tuto větev automatu opustit. Kdybych se ale přesunul do kořenu stromu, nemohl bych zjistit situaci, kdy slova, která končí v podstromech současného stavu, obsahují prefix jiných slov. Pro tento účel strom rozšíříme na graf pomocí přidání zpětných hran. Zpětná hrana ze stavu, který představuje nějaký sled znaků, vede do stavu, který představuje suffix tohoto řetězce. Takových stavů může být více, zpětná hrana povede do takového, který odpovídá nejdelšímu suffixu. Vidíme, že zpětná hrana vede nejméně o jednu úroveň níže.

Nejprve nastíním, jakou reprezentaci stavů budu v příkladech používat:

```
1: struct stav:
2:     slovo = NULL
3:     zpetna_hrana = koren
4:     zkratkova_hrana = NULL
5:     hrany [VELIKOST_ABECEDY] = {NULL... NULL}
```

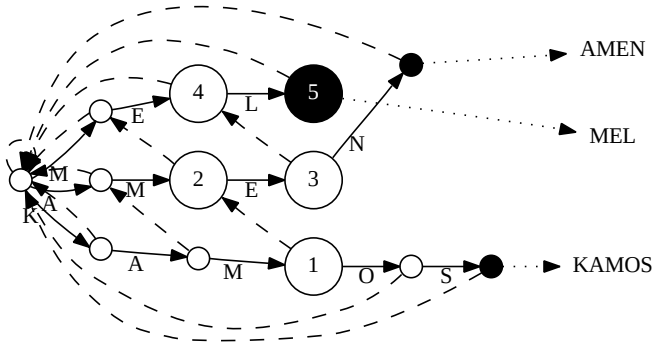
Procedura pro přechod ze stavu *stav* na písmeno *znak* vypadá tedy takto:

```
1: def prejdi(stav, znak):

2:     if stav.hrany [znak] != null:
3:         return stav.hrany [znak]
4:     else if stav != koren:
5:         return prejdi (stav.zpetna_hrana, znak)
6:     return koren
```

Ukažme si, jak vypadá automat pro tři slova: `KAMOS`, `AMEN`, `MEL`. Hrany vedoucí ze stavu na některý znak, jsou černé plné, zpětné hrany jsou černé čárkované, stavy, ve kterých končí některé slovo, jsou vybarveny černě a vede z nich tečkovaná hrana k dotyčnému slovu.

Slova začínají na různá písmena, mají tedy zcela disjunktní větve. Jak vypadá načítání vstupu "`KAMEL`"? Nejdříve se přímočaře přes přechody na písmena `K-A-M` dostaneme do stavu, který je na obrázku označen 1. Z toho ale nevede hrana na znak `E` a proto se přejde zpětnou hranou do stavu označeného 2, ze kterého již taková hrana vede, takže přes ní projde do stavu označeného 3. Načte se další znak `L`, na který ze současného stavu není přechod. Opět se použije zpětná hrana,



Obrázek 1: Automat pro `KAMOS`, `AMEN`, `MEL`

vedoucí do stavu 4, ze kterého přechod na tento znak existuje, tím se dostáváme do stavu 5, který odpovídá slovu `MEL`.

Musíme se vypořádat ještě s jedním problémem, který představuje zjišťování, zda ve stavu, ve kterém jsme, nekončí nějaké slovo, pokud je dané slovo podřetězcem tohoto stavu. Např. pokud máme automat tvořený slovy `KAMEN`, `AMEN`, `MEN` a načteme řetězec `KAMEN`, musíme v posledním stavu ohlásit všechny tři řetězce.

Pro ten účel zavedeme ještě jeden typ hrany, kterou Martin Mareš[1] pojmenoval „zkratkovou“. Hrana ze stavu vede do nejdelšího slova, které je suffixem řetězce, kterému odpovídá současný stav.

Následující obrázek představuje vyhledávací automat pro slova `KAMENOLOM`, `LOMIKAMEN`, `KAM`, `AMEN`, `LOM`. Zkratkové hrany jsou vyznačeny červeně.

Konstrukce automatu

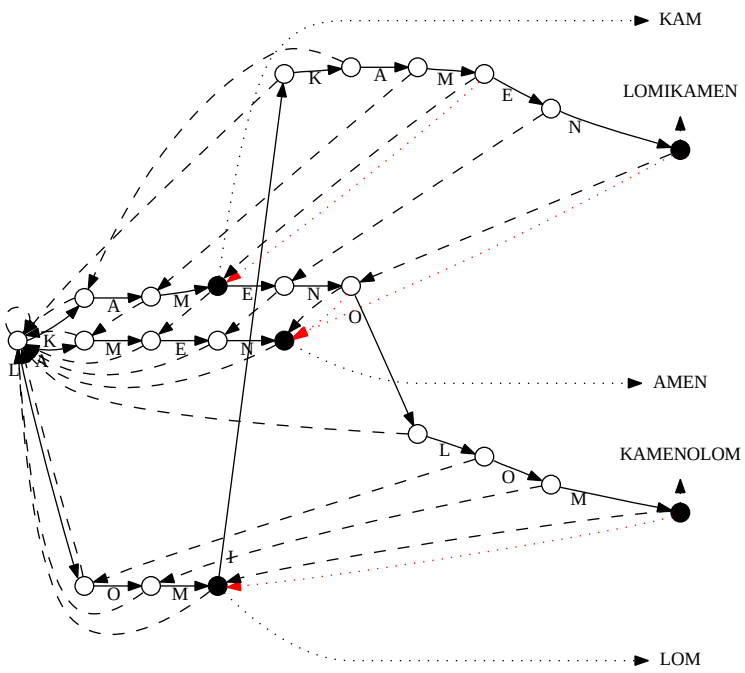
Zbývá popsat, jak takový automat zkonstruujeme. Nejdříve vytvoříme prefixový strom ze vstupních slov.

```
1: foreach slovo in slovnik:
2:     stav = koren
3:     foreach znak in slovo:
4:         if stav.hrany [znak] == NULL:
5:             stav.hrany [znak] = new stav ()
6:         stav = stav.hrany [znak]
7:     stav.slovo = slovo
```

Tento algoritmus bere jednou každý znak každého z hledaných slov, tedy má viditelně časovou složitost $\mathcal{O}(M)$.

Teď strom projdeme do šířky a nastavíme zpětné a zkratkové hrany. Na začátku mají všechny stavy automatu „rezignovaně“ nastavenou zpětnou hranu do kořene.

Stojíme ve stavu s, jehož zpětná hrana ukazuje na stav q odpovídající nejdelšímu suffixu s, a nastavujeme zpětnou hranu pro stav d, do kterého se ze s dostaneme přes znak z. Pokud ze stavu q vede také hrana na znak z (řekněme do stavu e), je jasné, že „shoda pokračuje“, máme již o jedna



Obrázek 2: Složitější automat

delší stejný suffix, takže zpětná hrana z d má ukazovat do stavu e. Pokud hrana z q na z nevede, třeba existuje takový stav odpovídající nejdelšímu suffixu q, ze kterého vede? Nebo-li pojedeme po zpětných hrany z q, dokud nenarazíme na stav, ze kterého vede hrana na z, nebo na kořen.

Zpětná hrana z d tedy ukazuje na stav e, který odpovídá nejdelšímu jeho suffixu. Je tedy jasné, že když se dostaneme do d, měli bychom oznámit bychom oznámit všechna slova končící nejen v d, ale i v e.

Algoritmus vypadá takto:

```
 1: fronta = [ koren ]
 2: while fronta.size > 0:
 3:     stav = fronta.pop_first ()
 4:     foreach z in 0..VELIKOST_ABECEDY-1:
 6:         stavs = stav.hrany [z]
 5:         if stavs == NULL: continue
 6:         stavz = prejdi (stav.zpetna_hrana, z)
 7:         stavs.zpetna_hrana = stavz

 8:         if stavz.slovo != NULL:
 9:             stavs.zkratkova_hrana = stavz
10:         else:
11:             stavs.zkratkova_hrana = stavz.zkratkova_hrana

12:         fronta.push_back (stavs)
```

Tento algoritmus bere jednou každý stav prefixového stromu, kterých může být nejvíce M.

Časová složitost

Pro každý načtený znak, kdy stojíme ve stavu hloubky q, můžeme buď udělat jeden krok a posunout se do hloubky $q+1$ (ze současného stavu existuje hrana pro načtený znak), nebo nejvýše q kroků po zpětných hranách („vyskákat“ až do kořene, uvažíme-li např. automat pro slova `ABC`, `BC`, `C`). Z toho vidíme, že průchod má časovou složitost $\mathcal{O}(L)$.

Algoritmus sestávající z konstrukce automatu ze slov o celkové délce M, průchod automatem pro vstup délky L s K výskyty hledaných slov tedy bude pracovat v čase $\mathcal{O}(M+L+K)$.

Implementace v C

Přikládám jednoduchou implementaci v C. Vstupem programu je soubor, kde v první části je na každém řádku jedno hledané slovo, následuje prázdný řádek a dále text, ve kterém se vyhledává. Na výstup se nejdříve vypíše sestavený vyhledávací automat a potom nalezená slova s pozicí. Automat se vypisuje ve formátu DOT, který slouží jako vstup pro (takřka geniální) program Graphviz[1], který vytváří co nejvíce rovinné nakreslení. Ta byla použita i v tomto textu.

Při implementaci je třeba řešit funkci, která pro zadaný stav a znak určí, zda z něj existuje hrana s tímto znakem. Nejpřímočarejší řešení, které jsem zvolil i já, je pole velikostí odpovídající rozsahu hodnot znaků. Paměťová náročnost je ale $\mathcal{O}(M * rozsah\_hodnot)$.

Reference

[1] Zápisy z ADS II. - Martin Mareš
http://mj.ucw.cz/vyuka/1112/ads2/

[2] Set matching and Aho-Corasick Algorithm - Pekka Kilpeläinen
http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf

[1] Domovská stránka projektu je http://www.graphviz.org