Rozhledy matematicko-fyzikální

Zdeněk Dvořák

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VII. část

Rozhledy matematicko-fyzikální, Vol. 83 (2008), No. 4, 16–26

Persistent URL: http://dml.cz/dmlcz/146091

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VII. část*)

Zdeněk Dvořák, Martin Mareš, Petr Škoda, MFF UK Praha

V tomto dílu programátorské kuchařky se budeme zabývat vyhledáváním slov v textu. Konkrétně mějme seznam slov a *hodně dlouhý* text a budeme chtít nalézt všechny výskyty těchto slov v textu. Tato úloha a její varianty mají mnoho praktických aplikací, například v textových editorech, internetových vyhledávačích nebo databázových systémech. Ukážeme si její řešení, kterému stačí jeden průchod textem a lineární čas na předzpracování seznamu slov. To je zjevně asymptoticky optimální, nicméně poznamenejme, že v této oblasti je známo mnoho dalších výsledků (toto řešení lze zychlit v některých speciálních případech; běžná je situace, kdy se text nemění a vyhledává se v něm opakovaně apod.)

Pro začátek si zavedeme několik pojmů:

- *Abeceda* Σ je libovolná konečná množina, jejímž prvkům budeme říkat *znaky*. Klidně si představujte klasickou latinskou abecedu, ale může to být například i množina $\{0,1\}$.
- Σ^* je množina všech *slov*, která lze z naší abecedy utvořit. To jsou všechny konečné posloupnosti znaků z Σ. Slovo z latinské abecedy je například „`lama`“, slovo z abecedy $\{0,1\}$ například posloupnost 01101. Slova budeme značit řeckými písmenky a zvláštní postavení mezi nimi má *prázdné slovo* ε. Budeme používat značení $\alpha[i]$ pro i-té písmeno slova α. Například, je-li $\alpha = \texttt{lama}$, pak $\alpha[3] = \texttt{m}$.
- $|\alpha|$ pro $\alpha \in \Sigma^*$ je délka slova, tedy počet jeho znaků – $|\texttt{lama}| = 4$, $|01101| = 5$, $|\varepsilon| = 0$.
- $\alpha\beta$ pro $\alpha, \beta \in \Sigma^*$ je zřetězení slov α a β, tedy slovo, které vznikne zapsáním slov α a β za sebe.
- γ^k je slovo vzniklé k-násobným zopakováním slova γ. Tedy $\gamma^0 = \varepsilon$, $\gamma^{k+1} = \gamma^k\gamma$.

*) Informace o Korespondečním semináři z programování pořádaného Matematicko-fyzikální fakultou Univerzity Karlovy v Praze lze nalézt na webové stránce `http://ksp.mff.cuni.cz`.

- Slovo α nazveme *podslovem* slova β, pokud je α obsaženo v β, čili pokud $\beta = \gamma\alpha\delta$ pro nějaká (případně i prázdná) slova γ a δ. Podslovo slova β začínající a-tým a končící b-tým znakem označme $\beta[a \ldots b]$. Například $\beta[2 \ldots |\beta|]$ je slovo β bez prvního písmene.
- Řekneme, že slovo α je *prefixem* slova β, pokud slovo β začíná slovem α, čili $\beta = \alpha\delta$ pro nějaké slovo δ.
- Podobně α je *sufixem* slova β, pokud β končí slovem α, tedy $\beta = \delta\alpha$ pro nějaké slovo δ.
- Každé slovo je prefixem i sufixem sebe sama. Je-li α pre-/sufixem slova β a $\alpha \neq \beta$, říkáme, že α je *vlastní* pre-/sufix β.
- Všimněte si, že prázdné slovo je podslovem, prefixem i sufixem každého slova včetně prázdného slova.

Po tomto teoretickém úvodu se konečně zamyslíme nad vlastním vyhledáváním. Nejprve si úlohu trochu zjednodušme a zkoumejme případ, kdy hledáme všechny výskyty jednoho slova $\alpha \in \Sigma^*$ o délce $|\alpha| = p$ v textu $\beta \in \Sigma^*$, $|\beta| = n$.

Asi první algoritmus, který nás napadne, je procházet text β od začátku až do konce a pro každou pozici v textu zkontrolovat, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provedeme až p porovnání znaků, čili celkem až np porovnání. Jde to lépe?

Všimněme si, že porovnávání slova s textem může skončit dvěma způsoby. Buď zjistíme, že se slovo s textem shoduje celé, nebo najdeme v textu znak, který ve slově není. Tehdy nestačí pokračovat novým vyhledáváním od místa, kde jsme skončili: např. pro slovo `instinkt` a text `instinstinkt` by algoritmus u druhého `s` zjistil, že se text liší, a pokud by pokračoval dále, již by nenalezl skutečný výskyt slova. Proto se vždy musíme vrátit o kousek zpět, v předchozím algoritmu jsme se vraceli vždy těsně za místo, kde se text začal se slovem shodovat.

Na druhou stranu, když se takto vrátíme, začneme znovu zpracovávat text, který už jsme jednou četli, takže je vlastně předem dáno, jak to dopadne. Pojďme toho využít. Říkejme *stavy* prefixům slova α. Pro každou pozici i v textu si označme $r[i]$ nejdelší stav, který je obsažen v textu tak, že v něm končí na pozici i, tj. nejdelší sufix slova $\beta[1 \ldots i]$, který je stavem. Zřejmě $r[i] = \alpha$ právě tehdy, když se α vyskytuje v β na pozici $i - p$.

Předpokládejme, že již známe stav $r[i]$, a chceme určit stav $r[i+1]$. Jestliže znak $\beta[i+1]$ prodlouží $r[i]$ na delší prefix slova α, získáme nový nejdelší stav $r[i+1]$ (rozmyslete si, že nemůže existovat delší). Zajímavější

je případ, kdy prefix není možné prodloužit. Nahlédněme, že useknutím posledního písmenka stavu získáme zase stav, takže useknutím posledního písmenka stavu $r[i+1]$ získáme nějaký sufix stavu $r[i]$. Stav $r[i+1]$ tedy vznikne prodloužením co možná nejdelšího sufixu stavu $r[i]$ o písmenko $\beta[i+1]$ (některé sufixy prodloužit nejdou, vezměme nejdelší, který jde). Pro předchozí příklad a prefix `instin` to bude sufix `in`. Samozřejmě je také možné, že nejde prodloužit žádný sufix, například pokud se znak $\beta[i+1]$ ve vyhledávaném slově vůbec nevyskytuje. V tomto případě bude stav $r[i+1]$ prázdné slovo.

Jelikož nový stav získáme ze sufixů předchozího stavu, nemusíme vědět vůbec nic o předcházejících písmenech textu. Postačí nám předpočítat si pro každý stav σ jeho nejdelší vlastní sufix, který je také stavem – ten si označíme $f(\sigma)$ a funkci f budeme říkat *zpětná funkce*. Přechod od $r[i]$ k $r[i+1]$ budeme provádět tak, že zkusíme $r[i]$ prodloužit o znak $\beta[i+1]$ a když to nepůjde, zkrátíme si $r[i]$ pomocí zpětné funkce a opět zkusíme přidat tentýž znak; pokud to stále nejde, zkracujeme dál opětovným zavoláním zpětné funkce, dokud se nám prodloužení nezdaří nebo dokud nedostaneme prázdné slovo.

Aby se nám se stavy v programu pohodlně pracovalo, očíslujeme si je – j-tý stav bude prefixem slova α o délce j. Zpětná funkce pak bude přiřazovat číslům čísla, takže si ji můžeme pamatovat v obyčejném jednorozměrném poli.

Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? Jak spočítat zpětnou funkci? Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až p-krát. Při každém volání však klesne délka aktuálního stavu alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků lineární v délce textu.

Konstrukci zpětné funkce provedeme malým trikem. Všimněte si, že $f(i)$ je přesně stav, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\alpha[2 \ldots i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní sufix daného stavu, který je také stavem, zatímco $r[i]$ označuje nejdelší sufix textu, který je stavem. Tyto dvě definice se liší jen v tom, že ta druhá připouští i nevlastní sufixy, a právě tomu zabráníme odstraněním

prvního znaku. Takže f získáme tak, že spustíme vyhledávání na část samotného slova w. Jenže k vyhledávání zase potřebujeme funkci f. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $f(1) = \varepsilon$. Pokud již máme $f(i)$, pak výpočet $f(i+1)$ odpovídá spuštění automatu na slovo délky i a při tom 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, jelikož $(i+1)$-ní prefix je prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celý řetězec $\alpha[2 \ldots p]$ a sledovat, jakými stavy bude procházet, a to budou přesně hodnoty zpětné funkce. Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $p-1$, a proto poběží v čase $O(p)$. Časová složitost celého algoritmu tedy bude $O(n+p)$. Dodáme už jen, že tento algoritmus poprvé popsali Knuth, Morris a Pratt [1] a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
var
  Slovo: array[1..P] of Char;
  Text: array[1..N] of Char;
  F: array[1..MaxS] of Integer; { zpětná fce }
  function Krok(I: Integer; C: Char): Integer;
  begin
    if (I < P) and (Slovo[I+1] = C) then Krok := I+1
    else if I > 0 then Krok := Krok(F[I], C)
    else Krok := 0;
  end;
var I, R: Integer; { pomocné proměnné }
begin { konstrukce zpětné funkce }
  F[1]:= 0;
  for I:= 2 to P do F[I]:= Krok(F[I-1], Slovo[I]);
  { procházení textu }
  R:= 0;
  for I:= 1 to N do
  begin
    R:= Krok(R, Text[I]);
    if R = P then writeln(I);
  end;
end.
```

Tento algoritmus můžeme také formálně popsat pomocí automatů:

Konečný automat nad abecedou Σ si můžeme představit jako stroj, kterému dáme slovo ze Σ^* a on ho buď odmítne nebo přijme. V průběhu práce je vždy v právě jednom *stavu* z nějaké pevné konečné množiny stavů. Slovo zpracovává po jednotlivých znacích a podle přečteného znaku se rozhodne, do jakého stavu přejde. K tomu slouží *přechodová* funkce g, která dvojicím (aktuální stav, znak) přiřazuje nové stavy. Když takto zpracujeme celé vstupní slovo, automat podle toho, v jakém stavu se právě nachází, odpoví, zda je slovo přijato nebo odmítnuto.

Konečný automat lze formálně nadefinovat jako čtveřici (Q, g, q_0, F), kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která pro daný stav automatu a znak na vstupu řekne, do jakého stavu má automat přejít;
- $q_0 \in Q$ je *počáteční stav,* v němž je automat na počátku výpočtu;
- $F \subset Q$ je množina *přijímacích stavů.*

Výpočet konečného automatu pak probíhá následovně:

1. Nastav aktuální stav s_0 na počáteční stav q_0.
2. Postupně čti znaky $x[i]$ ze vstupu a po přečtení každého přejdi ze stavu s_{i-1} do stavu $s_i = g(s_{i-1}, x[i])$.
3. Pokud skončíš v přijímacím stavu ($s_n \in F$), pak slovo přijmi.

Příklad 1. Mějme automat nad abecedou $\Sigma = \{0, 1\}$ se třemi stavy s_1, s_2, s_3, počátečním stavem $q_0 = s_1$, jedním přijímacím stavem $F = \{s_1\}$ a přechodovou funkcí g dle tabulky:

$$\begin{array}{ll} g(s_1, 0) = s_3 & g(s_2, 1) = s_3 \\ g(s_1, 1) = s_2 & g(s_3, 0) = s_3 \\ g(s_2, 0) = s_1 & g(s_3, 1) = s_3 \end{array}$$

Tento automat přijímá právě slova ve tvaru $(10)^k$, $k \geq 0$, tedy např. 101010 a prázdné slovo přijme, zatímco 1010101 odmítne.

Konečné automaty celkem dobře popisují chod našeho algoritmu – ten také zpracovává text po znacích a přechází podle právě přečteného znaku mezi stavy. Jsou zde ale ještě některé rozdíly: Předně KMP neodpovídá ano/ne, ale hlásí jednotlivé výskyty. K tomu můžeme automat upravit například tak, že množinu přijímacích stavů bude používat nejen na konci vstupu, ale v každém kroku. Druhá odlišnost tkví v tom,

že přechodová funkce KMP (ta odpovídá prodlužování prefixu o další písmeno) není definována všude. Tam, kde definována není, nastupuje místo ní zpětná funkce, která nás přesouvá mezi stavy tak dlouho, než přechodová funkce definována je.

Tomuto rozšíření se obvykle říká *vyhledávací automat* a definuje se jako pětice $(Q, g, f, q_0, \text{out})$, kde:

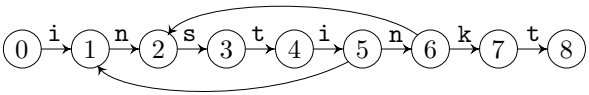
- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která je definovaná pouze pro některé dvojice (stav, znak);
- $f : Q \to Q$ je *zpětná funkce,* která říká, do jakého stavu se má automat přesunout, pokud přechodová funkce není definována;
- $q_0 \in Q$ je *počáteční stav,* v němž se automat nachází na začátku výpočtu;
- $\text{out} : Q \to \mathcal{P}(\Sigma^*)$ je *výstupní funkce,* která každému stavu přiřazuje, jaký se v něm má ohlásit výstup, což bude množina nalezených slov. V případě KMP byla tato množina vždy buď prázdná nebo jednoslovná; až budeme za chvíli hledat více slov, bude bohatší.

Výpočet dle vyhledávacího automatu probíhá následovně:

1. Nastav aktuální stav s na počáteční stav q_0.
2. Pro každý znak $c = x[i]$ vstupního textu proveď:
3. *Dokud* je $g(s, c)$ nedefinovaná, přejdi zpět do stavu $s \leftarrow f(s)$.
4. Přejdi do nového stavu $s \leftarrow g(s, c)$.
5. Vypiš všechna slova z $\text{out}(s)$.

Ještě doplňme, že aby se algoritmus vždy zastavil, musí pro každý znak $c \in \Sigma$ a každý stav s existovat stav s' dosažitelný z s opakovanou aplikací funkce f takový, že $g(s', c)$ je definováno. V našem případě to bude zaručeno tím, že $g(q_0, c)$ bude definováno pro každý znak $c \in \Sigma$. Pro znak c, kterým nezačíná žádné z hledaných slov, bude $g(q_0, c) = q_0$.

Příklad 2. Pro slovo `instinkt` by vyhledávací automat vypadal takto (zpětnou funkci jsme kreslili pouze tam, kde nevede do stavu 0):

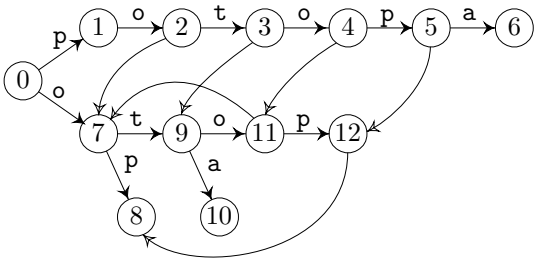


Nyní algoritmus KMP rozšíříme, aby uměl hledat více slov. Mějme slovník K, což je konečná množina slov nad abecedou Σ, a prohledávaný

text β. Vytvoříme vyhledávací automat, jehož výstupem bude výpis nalezených slov a jejich pozic v textu. Jeho stavy budou odpovídat prefixům všech slov ze slovníku a očíslujeme si je přirozenými čísly, počáteční stav $q_0 = 0$ bude odpovídat prázdnému prefixu. Výstupní funkce out pro prefix α ohlásí všechna slova ze slovníku, která jsou sufixem slova α.

Příklad 3. Jak takový vyhledávací automat může vypadat, si ukážeme pro latinskou abecedu a slovník

$$K = \{\texttt{potopa}, \texttt{op}, \texttt{ota}, \texttt{otop}\}.$$



Rovnými čarami je zobrazena přechodová funkce, kroucenými zpětná funkce. Nejsou zakresleny šipky do 0 u přechodové ani u zpětné funkce. Výstupní funkce je dána následující tabulkou:

$$\begin{array}{ll} \text{out}(5) = \{\texttt{otop}, \texttt{op}\} & \text{out}(10) = \{\texttt{ota}\} \\ \text{out}(6) = \{\texttt{potopa}\} & \text{out}(12) = \{\texttt{otop}, \texttt{op}\} \\ \text{out}(8) = \{\texttt{op}\} & \text{out}(\text{ostatní}) = \emptyset \end{array}$$

Vyhledávání pomocí tohoto automatu bude probíhat stejně jako u KMP, $r[i]$ opět bude nejdelší stav, na který končí právě přečtená část textu, složitost vyhledávání bude opět $O(n)$ až na vypisování výskytů, které poběží v čase $O(\text{počet výskytů})$, což může být více než lineárně, ale lépe to určitě nejde.

Pro pořádek dokážeme, že automat doopravdy vyhledává všechny výskyty: (i) Každé slovo, které oznámíme jako nalezené, se v textu opravdu vyskytuje ($r[i]$ se v textu vyskytuje podle své definice a všechna oznámená slova jsou sufixy $r[i]$). (ii) Všechny výskyty opravdu oznámíme. Pokud se na pozici i vyskytuje slovo $\alpha \in K$, pak je zajisté α jedním ze stavů, na něž $\beta[1 \ldots K]$ končí a $r[i]$ musí být buďto tento stav nebo nějaký ještě delší, jehož je α sufixem.

Teď se podíváme na to, jak vyhledávací automat pro daný slovník sestrojit. Provedeme to ve dvou krocích. Nejprve sestrojíme množinu stavů Q, přechodovou funkci g a částečnou výstupní funkci o. Ve druhém kroku vytvoříme zpětnou funkci f a rozšíříme o na výstupní funkci out.

V prvním kroku založíme počáteční stav 0, postupně projdeme celý slovník K a každé slovo σ ze slovníku do automatu přidáme. To provedeme tak, že začneme ve stavu 0 a pustíme automat na σ. Jakmile ale v některém stavu s pro znak $\sigma[i]$ nebude přechodová funkce definována, přidáme nový stav q, nastavíme přechodovou funkci $g(s, \sigma[i]) = q$, přejdeme do stavu q a pokračujeme. Tím v lineárním čase vytvoříme strom stavů. Pokaždé, když dojdeme na konec slova, nastavíme také částečnou výstupní funkci $o(q)$ na $\{\sigma\}$.

Popíšeme tuto část formálně:

1. Začni s množinou stavů $Q \leftarrow \{0\}$.
2. Pro každé slovo σ ze slovníku K proveď kroky 3–7:
3. Nastav aktuální stav s na 0.
4. Pro každé písmeno $\sigma[i]$ slova σ proveď 5–6:
5. Pokud je $g(s, \sigma[i])$ nedefinované, založ nový stav q, nastav $Q \leftarrow Q \cup \{q\}$ a polož $g(s, \sigma[i]) \leftarrow q$.
6. Přejdi do nového stavu: $s \leftarrow g(s, \sigma[i])$.
7. Nadefinuj částečnou výstupní funkci: $o(s) \leftarrow \{\sigma\}$.

Zpětnou funkci vytvoříme podobně jako pro jedno slovo tak, že pustíme ještě nehotový automat na část vyhledávaného slova. Opět chceme využít toho, že je funkce definovaná pro všechna kratší slova. Vezměme si náš příklad. Při přidávání slova `potopa` bychom nastavili $f(1) = 0$, $f(2) = 7$, $f(3) = 9$, ale u druhého `o` bychom chtěli použít zpětnou funkci $f(9)$, která ještě není definovaná. Proto budeme zpracovávat prefixy všech slov ze slovníku současně v pořadí podle jejich rostoucí délky.

Ještě vyřešíme výstupní funkci. Označme $\sigma(s)$ slovo, jehož cesta vede do stavu s. Pokud pro stav s platí $f(s) = 0$, znamená to, že neexistuje žádný nevlastní (neprázdný) sufix, který by byl prefixem některého ze slov ve slovníku. Proto v tomto stavu může skončit pouze slovo $\sigma(s)$. Nastavíme $\mathrm{out}(s) = o(s)$. Pokud $f(s) \neq 0$, končí v tomto stavu také všechna slova, které jsou sufixem slova $\sigma(s)$. Tehdy:

$$\mathrm{out}(s) = o(s) \cup \mathrm{out}(f(s)).$$

Opět formálně:

1. Založ frontu F, zatím prázdnou.
2. Nastav $f(0) \leftarrow 0$ a $\mathrm{out}(0) \leftarrow \emptyset$.
3. Pro každý znak $c \in \Sigma$ proveď následující krok:
4. Pokud je stav $s \leftarrow g(0,c) \neq 0$, pak nastav $f(s) \leftarrow 0$, $\mathrm{out}(s) \leftarrow o(s)$ a zařaď s na konec fronty F.
5. Dokud je nějaký stav ve frontě, prováděj následující:
6. Odeber první stav r z fronty F.
7. Pro každý znak $c \in \Sigma^*$, pokud je $a \leftarrow g(r,c) \neq 0$, proveď:
8. Buď $s \leftarrow f(r)$. Dokud $g(s,c) = 0$, přiřazuj $s \leftarrow f(s)$.
9. Nastav $f(a) \leftarrow g(s,c)$.
10. Nastav $\mathrm{out}(a) \leftarrow o(a) \cup \mathrm{out}(g(s,c))$.
11. Zařaď a na konec fronty F.

Aby algoritmus fungoval rychle, musíme zvolit šikovnou reprezentaci výstupní funkce. Kdyby si každý stav pamatoval svou vlastní množinu, mohly by tyto množiny dohromady být víc než lineárně velké (zkuste vymyslet příklad slovníku, pro který tomu tak je) a museli bychom se vzdát naděje, že stihneme automat zkonstruovat v lineárním čase. Proto použijeme trik: všimneme si, že $\mathrm{out}(s)$ je pro každý stav buďto rovna $\mathrm{out}(f(s))$ nebo se od ní liší přidáním slova $o(s)$. Stačí si proto pamatovat $o(s)$ a ještě nějakou funkci $z(s)$, která řekne, ve kterém stavu máme najít zbytek množiny $\mathrm{out}(s)$. Krok 9 proto upravíme takto:

9. Pokud je $o(f(s)) = \emptyset$, polož $z(s) \leftarrow z(f(s))$, jinak $z(s) \leftarrow f(s)$.

Podobně upravíme vypisování nalezených slov: vypíšeme $o(s)$ a pokud je $z(s) \neq 0$, pokračujeme ve vypisování ve stavu $z(s)$.

Ještě se zamysleme nad časovou složitostí. Označme P velikost celého slovníku. První část algoritmu provede maximálně $O(P)$ kroků, pokud považujeme velikost abecedy za konstantu. Ve druhé fázi se každý stav dostane do fronty právě jednou, takže vše je lineární až na průchody zpětnou funkcí. Podobně jako u KMP i zde spouštíme vyhledávací automat na všechna hledaná slova bez prvního písmene, až na to, že místo jedno po druhém je zpracováváme na přeskáčku a že společné části výpočtů (než se strom rozvětví) počítáme jen jednou. Celkem to tedy bude trvat nejvýše tolik, kolik vyhledání všech slov dohromady, což je $O(P)$.

Celkově tedy vyhledávací algoritmus běží v čase $O(P + n + v)$, kde n je délka textu, P celková velikost slovníku a v počet nalezených výskytů.

Na závěr dodejme, že tento algoritmus vymysleli Aho a Corasicková [2], a předveďme program:

```
var
  N: Integer; { délka textu }
  W: Integer; { počet slov }
  Slova: array[1..MaxW, 1..MaxK] of Char;
  Delky: array[1..MaxW] of Integer;
  Text: array[1..MaxN] of Char;
  Q: Integer; { počet stavů }
  { přechodová a zpětná funkce: }
  g: array[1..MaxQ, Char] of Integer;
  f: array[0..MaxQ] of Integer;
  { obě části výstupní funkce: }
  o: array[0..MaxQ] of Integer;
  z: array[0..MaxQ] of Integer;

  procedure NulujStav(State: Integer);
  var C: Char;
  begin
    o[State]:= 0;
    for C:= #0 to #255 do g[State,C]:=0;
  end;

  function Krok(S: Integer; C: Char): Integer;
  begin
    while (S>0) and (g[S,C]=0) do S:= f[S];
    Krok:= g[S,C];
  end;

var
  S, I, J, L: Integer;
  C: Char;
  FI, FC: Integer;
  Fronta: array[1..MaxQ] of Integer;
begin
  { vložíme všechna slova }
  Q:= 0;
  NulujStav(Q);
  for I:= 1 to W do
  begin
    S:= 0;
    for J:= 1 to Delky[I] do
    begin
      if g[S, Slova[I,J]] = 0 then
      begin
        Q:= Q + 1;
        NulujStav(Q);
        g[S, Slova[I,J]]:= Q;
      end;
      S:= g[S, Slova[I,J]];
```

```
    end;
    o[S]:= I;
  end;
  { zkonstruujeme zpětnou a výstupní funkci }
  f[0]:= 0; z[0]:= 0;
  FC:= 1; FI:= 1;
  Fronta[FC]:= 0;
  while FI <= FC do
  begin
    for C:= #0 to #255 do
    if g[Fronta[FI], C] <> 0 then
    begin
      S:= g[Fronta[FI], C];
      I:= Krok(f[Fronta[FI]], C);
      if Fronta[FI] = 0 then f[S]:= 0 else f[S]:= I;
      if o[f[S]] <> 0 then z[S]:= f[S]
             else z[S]:= z[f[S]];
      FC:= FC + 1;
      Fronta[FC]:= S;
    end;
    FI:= FI + 1;
  end;
  { hledáme }
  S:= 0;
  for I:= 1 to N do
  begin
    S:= Krok(S, Text[I]);
    L:= S;
    while L <> 0 do
    begin { hlásíme výskyty }
      if o[L] <> 0 then
      begin
        write(I, ': ');
        for J:= 1 to Delky[o[L]] do write(Slova[o[L],J]);
        writeln;
      end;
      L:= z[L];
    end;
  end;
end.
```

Literatura

[1] Knuth, D., Morris, J. H., Pratt, V.: Fast pattern matching in strings. *SIAM Journal on Computing* **6** (1977), s. 323–350.

[2] Aho, A. V., Corasick, M. J.: Efficient string matching: An aid to bibliographic search. *Communications of the ACM* **18** (1975), s. 333–340.