Miroslav Beneš, 2001
Informatika I/2/31

Vyhledávání v textu

Obsah

I. Úvod

V dnešní době, kdy se většina dokumentů (ať už na úřadech nebo v běžných podmínkách) vede na počítačích, a tedy v elektronické podobě, je potřeba míti dostatečně silnénástroje k tomu, aby se v daných dokumentech vyhledal nějaký text (slovo, věta apod.)v co nejkratším možném čase. Proto se řadu let vyvíjí různé algoritmy, které tento úkols odlišnými úspěchy plní.
Tyto algoritmy jsou běžnému uživateli počítače schovány v jiných větších celcích jako jsourůzné textové editory, kde je občas nutné nějaké to slovo v textu vyhledat (potažmo zaměnitza jiné). Dalším příkladem mohou být vyhledávací systémy v knihovnách, vyhledávání v databázích, všelijaké vyhledávací servery apod. Principy vyhledávání vzorků v textu se také používají i v jiných oborech než v informatice, příkladem může být napříkladhledání "vzorků" v DNA šroubovici.
Z toho vyplývá, že vyhledávání v textu je poměrně rozsáhlý problém o jehož užitečnostijistě není sporu. Proto je určitě užitečné uvést si nějaké příklady algoritmů, které tentoproblém řeší, popsat je a porovnat.

Algoritmy se mohou dělit do několika možných kategorií podle toho za jakým účelem se používají nebo podle svého původu. Můžeme se tedy setkat s algoritmy, které vyhledávajípouze jeden vzorek v daném textu, v jiných případech je možno vyhledávat množinu vícevzorků. Rozdíl je také v tom, že některé algoritmy naleznou pouze první výskyt vzorkuv textu, jiné naleznou všechny výskyty.Existují algoritmy, které se inspirovaly i jinými obory informatiky jako je např. teorieautomatů.

V tomto článku bych se rád věnoval zejména dvěma algoritmům. Prvním je Knuth-Morris-Prattův algoritmus (KMP), druhým je Boyer-Mooreův algoritmus (BM). Společným rysem obou jeskutečnost, že vyhledávají všechny výskyty jednoho vzorku v daném textu. Na KMP se dánahlížet jako na upravený konečný (v tomto případě vyhledávací) automat. BM se na druhoustranu jeví jako naivní vyhledávací algoritmus (viz dále) se dvěmidůležitými heuristikami.
Představitelem skupiny algoritmů, které vyhledávají celou množinu vzorků je např.algoritmus Aho-Corasickové, který je založen na poznatcích právě z teorie automatů.
V dalším odstavci bych rád zmínil další dva zajímavé a relativně nové algoritmy: Baeza-Yates-Gonnet, který je založen na vyhledávání pomocí bitových masek, a algoritmusQuicksearch, jehož autorem je D.M. Sunday.

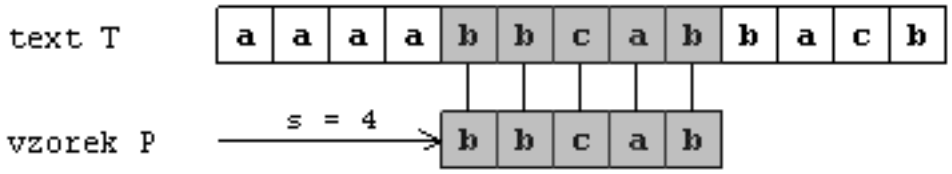
Nejprve si však zaveďme několik důležitých pojmů a definic, které se budou hodit k pozdějšímu popisu a zkoumání algoritmů.

>>>Obsah

II. Základní pojmy

I přesto, že většina lidí v principu chápe, co vyhledávání v textu je, definujme sitento pojem alespoň trochu přesněji a formálněji.
Nejprve si vysvětlíme některé základní pojmy potřebné pro popis daného problému. **Abecedou** Σ rozumíme konečnou množinu znaků. Například Σ ={0,1} nebo Σ ={a,b,...,z}. Prvky této množiny se častonazývají znaky, písmena nebo symboly. **Slovem** v abecedě Σ je míněnakonečná posloupnost znaků z této abecedy. Prázdným slovem se rozumí posloupnost délky0 a obvykle se značí ϵ (ϵ nepatří do Σ).
Množina všech slov v abecedě Σ se značí Σ^*. Na této množině je definovánaoperace **skládání (konkatenace)**, která dvěma slovům x a y délek m a n přiřadí slovo xy délky m+n. Tato operace je asociativní (to znamená, že (xy)z je to saméslovo jako x(yz)) a pro více než jednoprvkovou abecedu nekomutativní(v případě jednoprvkové abecedy: aa"=" aa, pokud Σ ={a}; v případě víceprvkové např. xy není rovno yx pro x, y navzájem různá). Roli jednotkového prvku této operace hraje prázdné slovoϵ, tedy xϵ =ϵ x=x pro každé x z ϵ.
Nechť x je prvkem Σ^*. Potom délkou slova x rozumímepočet znaků v x. Zapisujeme |x|. Tedy jak bylo poznamenáno |ϵ |=0.
Řekneme, že slovo x z Σ^* je **předponou (prefixem)**slova y z Σ^*, existuje-li takové slovo u zΣ^*, že xu=y. Takové u zřejmě existuje nejvýše jednoa je-li neprázdné, říkáme, že **xvlastní předpona**. Obdobně, řekneme, že slovo x z Σ^* je **příponou (suffixem)**slova y z Σ^*, existuje-li takové slovo v zΣ^*, že vx=y. Opět takové v existuje nejvýše jednoa je-li neprázdné, mluvíme o **vlastní příponě**.Zřejmě platí, že ϵ je vlastní příponou a předponou každého slova z Σ^*. Podobně každé slovo je svou jedinou nevlastní příponou i předponou.
Pro příklad si uveďme, že slovo ab je předponou slova abbdad a to předponou vlastní. Zároveň je vlastní příponou slova abbab.
Pro další účely si zaveďme ještě jeden jednoduchý pojem, **prefix o k znacích.** Prefix vzorku P[1..m] o k znacích, tedy P[1..k],budeme značit P_k. Podobně budeme mít tento pojem i pro prohledávaný text(T_k).
Nyní se vraťme k formulaci problému vyhledávání v textu. Nechť máme danou abecedu Σ a tím i množinu Σ^*. Předpokládejme, že máme dány dva textové řetězce (nejlepší je

představit si je jako pole jednotlivých znaků). Řetězec P = $p_1 \ldots p_m$ (nebo jako pole znaků P[1..m]) budeme nazývat **vzorek**. Jeho délka je m. Řetězec T = $t_1 \ldots t_n$ (T[1..n]) bude prohledávaný textdélky n. Oba řetězce jsou slova z Σ^*.
Říkáme, že vzorek P se v textu T **nachází s posunutím** s (jinými slovy řečeno, nachází se v textu T **na pozici** s+1), jestliže 0<=s<=n-m a zároveň T[s+1..s+m]=P[1..m] (pro všechna j 1<=j<=m $t_s + j = p_j$). Pokud se vzorek P v prohledávaném textu T nachází nazýváme **splatným posunem** . Jinak je tento posun **neplatný**. Problém vyhledávání jednoho vzorku v textu lze tedy formulovat jako problém nalezení všech platných posunů, se kterými se vzorek P nachází v textu T.



Tento obrázek znázorňuje předchozí definici o vyhledávání v textu. Vzorek P=bbcabse nachází v textu T s platným posunem 4, neboli na 5. pozici.

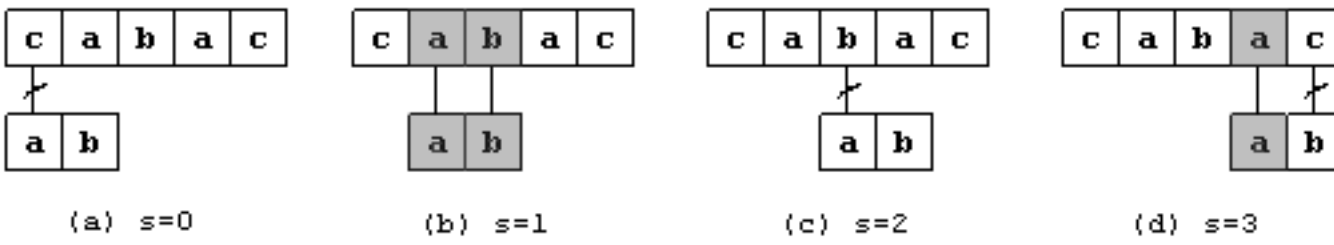
>>>Obsah

III. Algoritmy

Po vysvětlení všech potřebných pojmů a formulaci daného problému se můžeme začít zabývat jednotlivými algoritmy. Nejprve se zlehka podíváme na naivní vyhledávací algoritmus a jehočasovou složitost, aby se určitým způsobem zdůvodnila potřeba lepších a efektivnějšíchvyhledávacích algoritmů. Poté probereme již zmíněné dva algoritmy (Knuth-Morris-Pratt aBoyer-Moore).

1. Naivní algoritmus

Tento algoritmus je v podstatě výsledkem první myšlenky, která každého napadne, když dostaneza úkol navrhnout algoritmus na vyhledávání v textu. Jednoduše řečeno spočívá v prozkoumánívšech možností (ne tedy doslova, ale v principu ano). Jak tomu u takových algoritmů bývá,jeho časová složitost není příliš dobrá.
Myšlenka je jednoduchá. Budeme procházet zadaný text a na každé pozici zkontrolujeme, zdatu nezačíná daný vzorek. Princip znázorňuje následující obrázek.



Jak je z obrázku vidno, vzorek se jakoby postupně posouvá pod daným textem a na

každém místěse kontroluje, zda se zde nenachází příslušný vzorek. V tomto příkladu byl vzorek nalezen nadruhé pozici (s posunem 1, případ (b)). V ostatních případech vždy nastala na určitém místě vzorku kolize.
Neefektivnost tohoto algoritmu spočívá právě ve skutečnosti, že pokud narazím na neshodu, posunuvzorek pouze o jedno místo doprava a začnu porovnávat znovu, a to až do konce prohledávanéhotextu. Tímto postupem se tedy nevyužívají informace, které byly získány v předchozím kroku.Například u našeho obrázku po tom, co jsem našel vzorek v případě (b), nemusím kontrolovatpozici o jednu vpravo, ale mohu přistoupit až k nákresu pod písmenem (d).
Tyto informace, které závisí právě na zadaném vzorku se snaží využívat efektivnější algoritmy, které si zde ukážeme.
Naivní algoritmus by v pseudokódu mohl vypadat asi následovně:

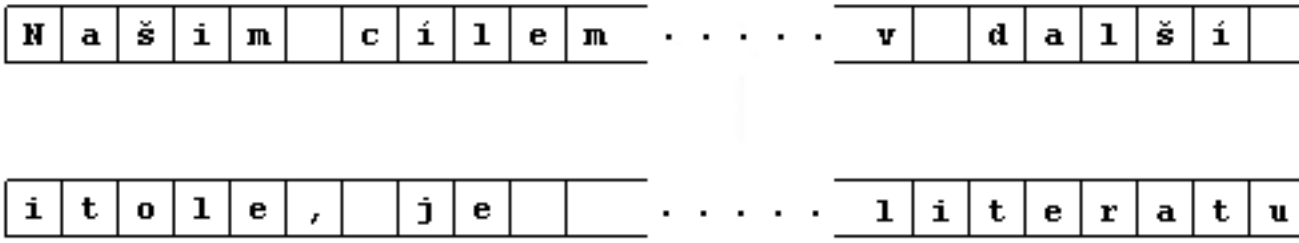
```
(1)  n = length(T)
(2)  m = length(P)
(3)  for s = 1 to n-m+1
(4)   if T[s..s+m-1] == P[1..m]
(5)    then print("Vzorek se nachází na pozici", s)
```

Nyní se kód rozeberme a podívejme se na časovou složitost algoritmu. První dvě řádky obstarávajípouze uložení délky obou textových řetězců do proměnných `n` a `m`.Posouvání vzorku pod textem zajišťuje cyklus, který začíná na řádce (3). Provede se přesně(n-m+1)-krát, což je počet pozic, na kterých se může vzorek vyskytovat. Řádka (5) jen informujeo nalezení vzorku v textu. Pro určení časové složitosti je klíčová řádka číslo 4. Zde seprovádí porovnávání vzorku s daným textem. Tento pseudozápis může ve skutečnosti být while cyklus, který porovnává jednotlivé znaky dokud nenarazí na neshodu nebo na konec vzorku, v tomtopřípadě se vykoná řádka (5). Je snadné nahlédnout, že v nejhorším případě (to je že vždy projduvšechny znaky ve vzorku) se while cyklus provede m-krát. Časová složitost v nejhorším případěje tedy *O((n-m+1)*m)*.

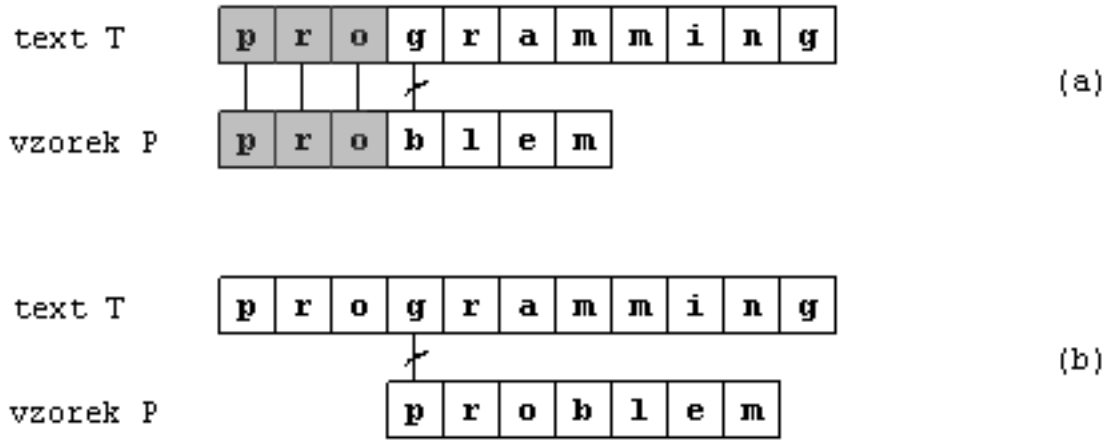
>>>Obsah

2. Knuth-Morris-Prattův algoritmus (KMP)

Mezi prvními, kteří si uvědomili, že informace, které získává naivní algoritmus svým porovnávánímznak po znaku, mohou být velmi cenné pro návrh efektivního algoritmu, byl právě Knuth se svýmispolečníky Morrisem a Prattem. Jejich nápad spočíval v tom, že pokud se tyto informace využijísprávným způsobem, může se vzorek nad prohledávaným textem posouvat i o více než pouze o jedenznak doprava. Tím se významně zkrátí doba potřebná k prohledání textu. Také je zbytečné sev prohledávaném textu vracet ke znakům, které již byly analyzovány tak jak to činí naivní algoritmus. Toto vracení spočívá ve skutečnosti, že pokud při porovnávání vzorku s daným textemnarazím na neshodu, vrátím se zpět na začátek vzorku a ten posunu o jedno místo doprava. Tato činnost je zřejmě zbytečná, neboť já již mám informaci o předchozích znacích, stačí jipouze dostatečně využít. Vracení se v textu může přinést i další problém, který není na první pohled zřejmý. Při zpracovávání delšího textu, určitě není tento text v paměti počítače celý.Ze souboru se načítá po kusech do nějakého bufferu v paměti, se kterým se poté pracuje.Podívejme se na následující příklad.

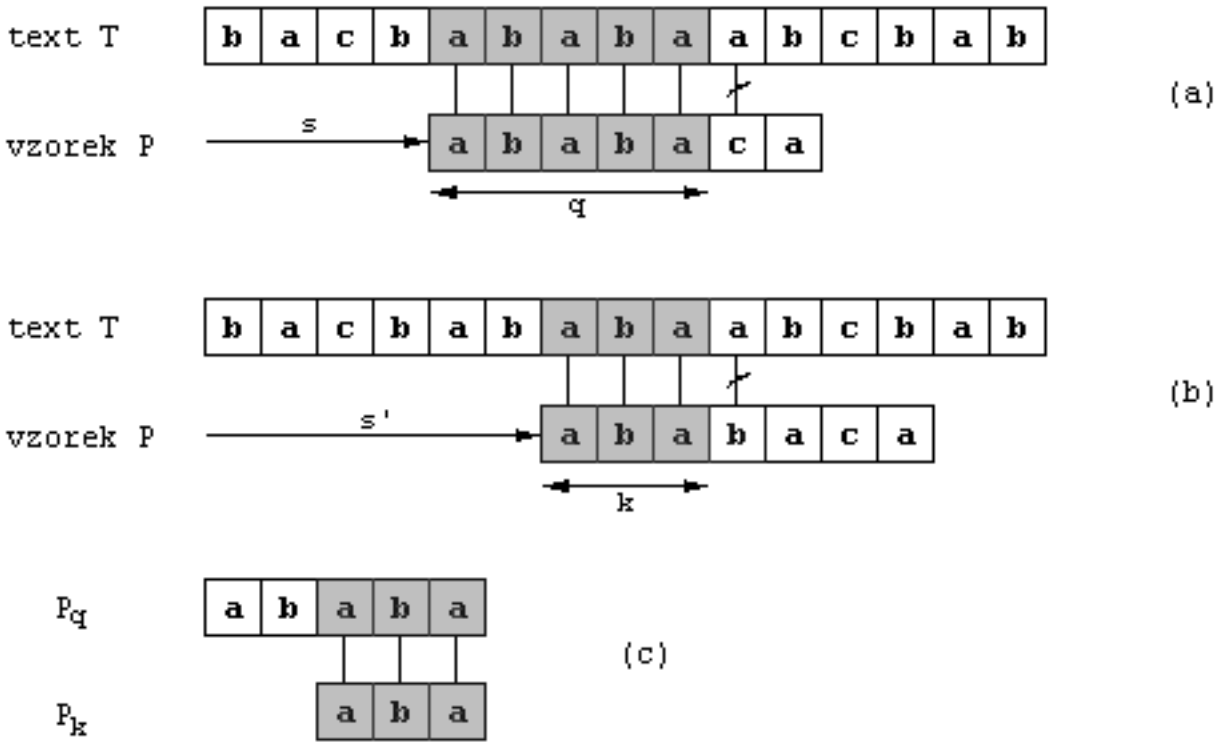


Dejme tomu, že v takovémto textu hledáme výskyt slova **kapka**. Dva řádky v obrázkupředstavují dva kusy textu tak, jak jsou načteny do paměti (bufferu). Naivní algoritmusprochází prvním bufferem a na konci s podezřením, že se nachází uprostřed vzorku načte novýkus textu a tento zahodí. Ovšem hned u prvního znaku zjistí neshodu a vzhledem ke své funkci munezbude nic jiného než se v textu vrátit. To ale představuje další přístup na disk, neboť se musínačíst předchozí kus textu. Princip algoritmu KMP zajistí, že se nic takového nestane.



Tento obrázek je příkladem výpočtu KMP algoritmu. Porovnávání vzorku s textem začíná jako obvykleu prvního znaku zleva (vzorek je zarovnán s textem). Algoritmus postupuje dokud nenarazí naneshodu na čtvrté pozici mezi znaky **b** a **g** (obrázek (a)).Z předchozích znaků okamžitě víme, že posun vzorku o jeden nebo dva znaky nemá význam. Posun o tři znaky ale může splnit účel. Tím se vzorek zarovná s textem nad znakem, kde nastala neshoda.Odtud dále může pokračovat porovnávání. Jak vidno na této pozici se hned první písmeno vzorkuneshoduje se znakem v textu (obrázek (b)), vzorek se poté posune o jedno místo doprava.Velikost takového posunu (o tři v prvním nebo o jeden znak v dalších případech) závisí pouzena charakteru a formě každého vzorku. Posun je nezávislý na prohledávaném textu. Jeho velikosturčuje tzv. **prefixová funkce**.
Díky prefixové funkci si před spuštěním vlastního vyhledávacího algoritmu předpočtu hodnotyposunů pro jednotlivé pozice ve vzorku do nějaké tabulky. Mnoho efektivních vyhledávacích algoritmů používá podobné předpočítané tabulky, které se později v průběhu vyhledávání používají.Tedy jak je patrné algoritmus KMP bude mít dvě fáze. V první fázi si z daného vzorku vypočítámepotřebné hodnoty posunů. Druhá fáze bude uskutečňovat vlastní vyhledávání.

Nejdříve se tedy věnujme první fázi a prefixové funkci. Tato funkce vyjadřuje chování vzorku vzhledem k posunům k sobě samému. Uveďme si ještě jeden příklad.



Na obrázku (a) vidíme, že při takovémto zarovnání (s posunem `s`), se prvních pětpísmen vzorku shoduje s pěti písmeny v textu (`q=5`), přičemž na znaku šestém došlok neshodě. Z informace, že pět písmen se shodovalo, můžeme okamžitě vyvodit, které to byly, neboť je to prvních pět písmen ve vzorku, a také můžeme zjistit příslušný posun. Je možné určitposuny, o kterých již teď mohu prohlásit, že jsou neplatné, a tím je v budoucnu přeskočit. Zdeje na první pohled jasné, že posun o jedno políčko doprava (tedy posun `s+1`) jeneplatný, neboť první písmeno ve vzorku (**a**) by bylo zarovnáno k písmenu v textu,o kterém již máme informaci, že se shodovalo s druhým písmenem ve vzorku (**b**).Posun `s+2` na obrázku (b) naopak dává jistou naději, že by vzorek mohl být nalezenna tomto místě, neboť tři znaky se shodují. K navržení kódu, který vypočítá dané hodnoty je tedy třeba vyřešit následující problém. Nechťmáme dány znaky `P[1..q]`, o kterých víme, že se shodují s těmito znaky v textu `T[s+1..s+q]` (v našem příkladu je to slovo **ababa**, na obrázku vybarvená políčka). Jaký je nejmenší posun `s'>s` takový, aby platilo `P[1..k]=T[s'+1..s'+k]`, přičemž `s'+k=s+q`. Slovy řečeno to znamená přesněto, o co se snažíme. Jak moc můžeme vzorek posunout doprava tak, aby se předpona vzorku kratší než počet znaků (**aba**ba), které se předtím shodovaly (ababa, tedy 5), shodovalas příponou slova v textu (ab**aba**), které bylo středem shody (ababa). Pokud takovápředpona není (`k=0`) posuneme vzorek o počet znaků, které se shodovaly (`q=5`).
Tedy posun `s'` je nejmenší takový, že je větší než `s` a není nezbytněneplatný. Jak bylo řečeno v nejlepším případě je nový posun `s'` roven `s+q`. V každém případě již nemusíme porovnávat prvních `k` znaků vzorku,neboť jejich shodu v textem máme zaručenou.
Tyto informace se dají spočítat porovnáváním vzorku se sebou samým, jak je hrubě naznačenona obrázku (c). Víme, že `T[s'+1..s'+k]` (ab**aba**) je část známéhotextu, a tedy to musí být

přípona jisté části P, která také byla prozkoumána.Je to přesně přípona P_q. Nyní můžeme přesně formulovat požadavek na posun s' a to pro každou pozici ve vzorku.
Nechť mám vzorek P[1..m], potom prefixová funkce (udává posuny s')vypadá následovně:

π : \{1,...,m\} -$>$ \{0,...,m-1\}, taková že

π[q]= max\{k: k$<$q and P$_k$ je příponou P$_q$\}.

Pro prefixovou funkci dále platí, že pro každé q z {1,...,m} je π[q]< q.

Pozn. π[q] tedy představuje délku nejdelší předpony P, která je příponou P_q.

Nyní se podívejme, jak bude algoritmus KMP fungovat jako celek. Tedy první fází je výpočet prefixové funkce, který jsme si v principu popsali. Následuje fáze druhá, vyhledávání.Princip je velice jednoduchý. Na začátku zarovnám vzorek vůči textu a začnu porovnávat. Pokudnarazím na neshodu, podívám se do tabulky prefixové funkce s indexem, který odpovídá pozici vevzorku, kde došlo k neshodě. S tabulky zjistím číslo, které udává znak vzorku (číslo určujejeho pozici ve vzorku) který se, když příslušně zarovnám vzorek k textu, bude nacházet přesněnad znakem v textu, který byl příčinou neshody (vzorek se tedy posune doprava). Vzorek budu tímtozpůsobem posouvat doprava dokud nenarazím na jeho začátek nebo nebudu moci pokračovat od danépozice dalším porovnáváním. Takto pokračuji dokud nenarazím na konec textu. Pokud vzorek v textu najdu, oznámím to a pokračuji dále (dá se totiž uvažovat, že u posledního znaku ve vzorku došlo k neshodě).
Nyní si již můžeme uvést zápis algoritmu v pseudokódu (na vstupu je text T a vzorek P).

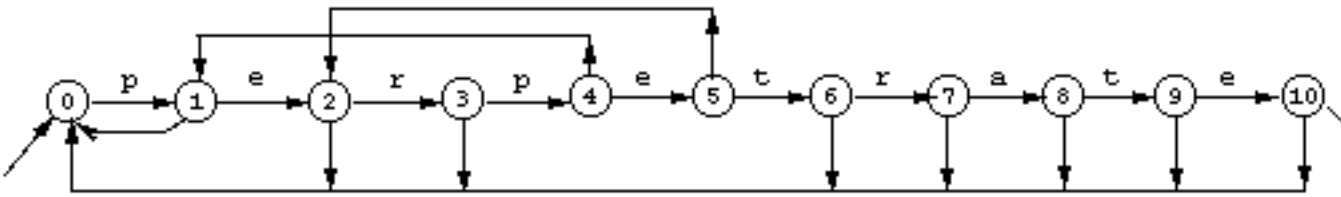
```
 (1)  n = length(T)
 (2)  m = length(P)
 (3)  $\pi$ = Prefix\_func(P)
 (4)  q = 0
 (5)  for i = 1 to n
 (6)    while (q $>$ 0 \&\& P[q+1] != T[i]) q = $\pi$[q]
 (7)    if P[q+1] == T[i] then q = q+1
 (8)    if q = m then
 (9)      print("Vzorek se nachází na pozici", i-m+1)
(10)      q = $\pi$[q]

Prefix\_func(P)
 (1)  m = length(P)
 (2)  $\pi$[1] = 0
 (3)  k = 0
 (4)  for q = 2 to m
 (5)    while (k $>$ 0 \&\& P[k+1] != P[q]) k = $\pi$[k]
 (6)    if P[k+1] == P[q] then k = k+1
 (7)    $\pi$[q] = k
 (8)  return $\pi$
```

Nyní si určíme časovou složitost algoritmu. Nejprve výpočet prefixové funkce. Základem je,že vnitřní while cyklus se provede nejvýše tolikrát jako cyklus vnější. Platí totiž, že každýmprůchodem vnitřním cyklem se proměnná k sníží, neboťπ[k]< k. Současně se proměnná

`k` může zvýšit nejvýšejednou v každém kroku vnějšího cyklu (díky řádce (6)). Z toho evidentně plyne, že počet průběhůvnitřním cyklem je menší roven počtu kroků vnějšího cyklu. Odtud již plyne, že časová složitostvýpočtu prefixové funkce je *O(m)*.
Obdobnou úvahou dojdeme k tomu, že časová složitost vyhledávací fáze je *O(n)*. Časová složitost celého algoritmu je tedy *O(m+n)*, což je výrazně lepší než u naivního algoritmu.

V úvodu celého referátu bylo řečeno, že KMP algoritmus do jisté míry souvisí s konečnými automaty. Jak? Je to velice jednoduché. Předpokládejme, že máme vzorek `P` délky `m`. Definujeme si tedy konečný automat, který bude mít `m+1` stavů. Přechody mezi jednotlivými stavy budou postupně určené jednotlivými písmeny vzorku. Tedy např.přechod mezi nultým a prvním stavem bude podle písmena p_1, přechod meziprvním a druhým stavem podle p_2 atd. Zbylé přechody (tedy jakési chybové)bude určovat právě prefixová funkce. Vstupním stavem bude stav 0 a výstupním stav `m`.Samotné vyhledávání bude realizováno jako práce takového automatu se vstupem, který odpovídázadanámu textu. Je zde pouze rozdíl v tom, že pokud se pomocí prefixové funkce vrátím do některého předešlého stavu, okamžitě zkusím přes to samé písmeno (které vlastně způsobilo neshodu) přejít do následujícího stavu, jinak se vracím dále.
Uveďme si příklad.



Na obrázku je konečný automat pro vzorek **perpetrate**. Nyní si ukážeme, jak budevypadat výpočet automatu pro dvě různá slova. Výpočet je znázorněn jako posloupnost stavů, mezikterými jsou písmena, přes která se mezi stavy přechází.

(a) budeme prohledávat text **perperpetrate**: $0^p1^e2^r3^p4^e5^r2^r3^p4^e5^t6^r7^a8^t9^e10$
(b) nyní to bude text **perpespetrate**: $0^p1^e2^r3^p4^e5^s2^s0^p1^e2^r3^p4^e5^t6^r7^a8^t9^e10$
>>>Obsah

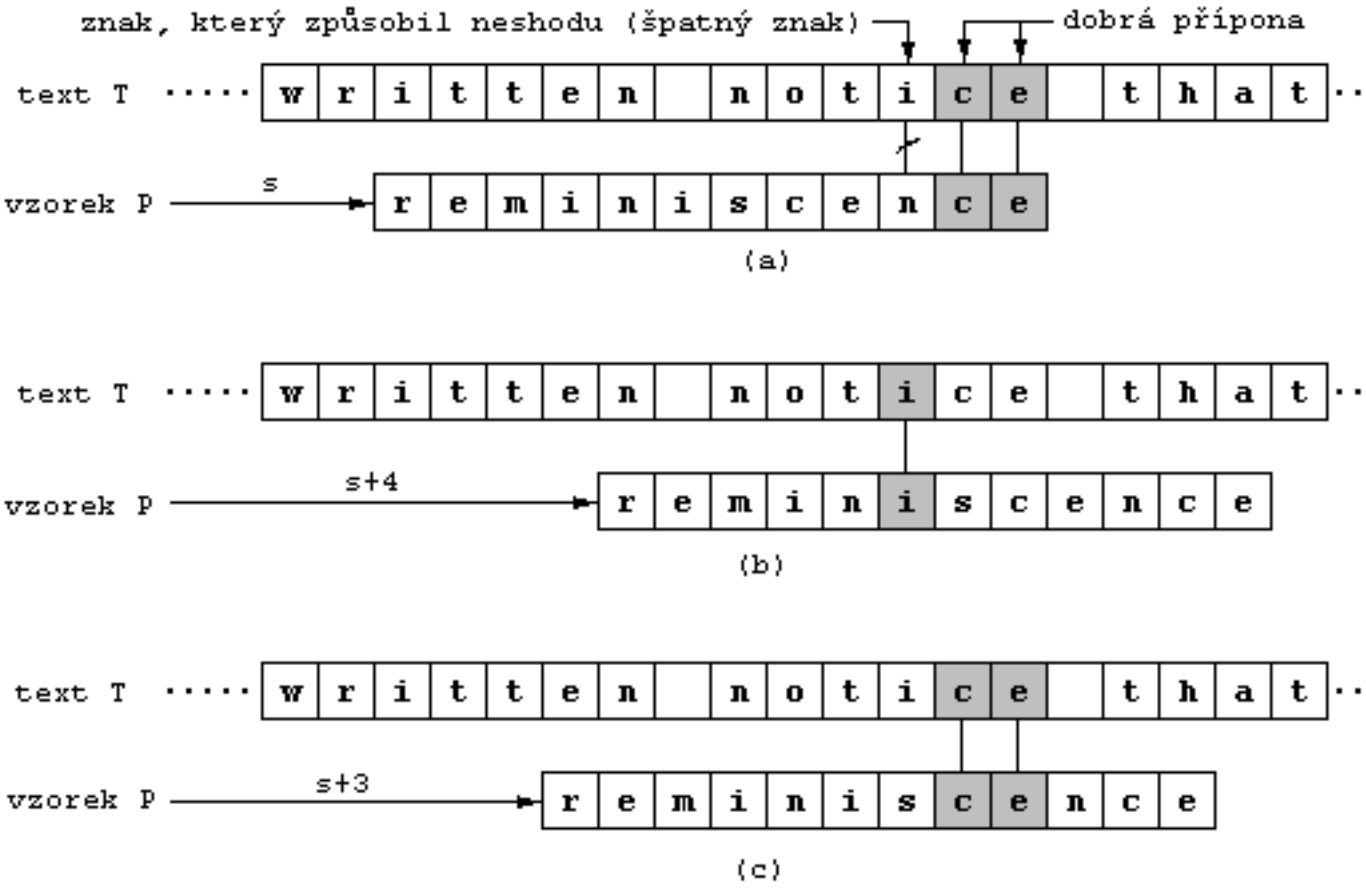
3. Boyer-Mooreův algoritmus (BM)

V této kapitole si předvedeme další chytrý a efektivní algoritmus jehož autory jsou S. Boyer aJ. Strother Moore. Jak bylo řečeno v úvodu, tento algoritmus se příliš zásadně neliší od naivního algoritmu, který jsme si ukázali. Je zde pouze několik odlišností. Abychom si jeukázali, předvedeme si ihned, jak Boyer-Mooreův algoritmus vypadá. Na vstupu je text `T`, vzorek `P` a abeceda Σ.

```
(1)  n = length(T)
(2)  m = length(P)
(3)  $\lambda$ = Last\_Occur\_func(P,m,$\Sigma$)
(4)  $\gamma$ = Good\_suff\_func(P,m)
(5)  s = 0
```

```
 (6)  while (s $<$= n-m)
 (7)    j = m
 (8)    while (j $>$ 0 \&\& P[j] == T[s+j]) j = j-1
 (9)    if j == 0
(10)      then print("Vzorek se nachází na pozici", s+1)
(11)           s = s+$\gamma$[0]
(12)      else s = s+max($\gamma$[j], j-$\lambda$[T[s+j]])
```

V čem se tedy tento algoritmus podobá a v čem se liší od naivního algoritmu? Podobnost je jakve struktuře (což zas tak podstatné není) tak ve skutečnosti, že se vzorek opět porovnávás daným textem a v případě neshody se vzorek posune doprava. Odlišnosti jsou v tom, ževzorek se s textem porovnává zprava doleva, tedy odzadu (u naivního algoritmu se porovnáváníprovádí zleva doprava). Pokud narazím na začátek vzorku, je jasné že jsem v textu našel jeho výskyt. Zde je další rozdíl, neboť při takovém nálezu neposunu vzorek o jedno místo doprava,ale o nějakou hodnotu $\gamma[0]$. Pokud narazím na neshodu opět posunu vzorek, ale posun nemusímít nutně velikost jedna jako u naivního algoritmu. Ve skutečnosti je tento posun mnohdy mnohemvětší. Další odlišností je, že v případě naivního algoritmu (a vlastně i v případě Knuth-Morris-Prattova algoritmu) se zpracoval každý znak prohledávaného textu aspoň jednou (u naivního algoritmu i mnohokrát). U Boyer-Mooreova algoritmu se díky tomu, že vzorek procházímod konce, a díky tomu, že vzorek v případě neshody posunu mnohdy o více než jedno písmeno doprava, na některé znaky v prohledávaném textu vůbec nedostane (přeskočí se).
Aby se tohoto úspěchu dosáhlo, používá algoritmus dvě heuristiky (v kódu jsou reprezentoványzatím záhadnými symboly γ a λ). Jelikož jde o heuristiky dá se očekávat, žese časová složitost v nejhorším případě oproti naivnímu algoritmu příliš nezlepší. Naštěstíjsou tyto heuristiky tak efektivní a úspěšné, že v běžné praxi dosahují velmi dobrých výsledků.Jak bylo řečeno výše, spousta znaků v textu se díky těmto heuristikám může jednoduše přeskočitaniž by byly nějakým způsobem zpracovány. Na následujícím obrázku si ukážeme, jaká ja základnímyšlenka obou heuristik. Jen pro zajímavost v angličtině se nazývají "bad-character heuristic"(tedy něco jako heuristika špatného znaku. Z toho se dá odvodit, že heuristika bude nějakýmzpůsobem souviset se znakem, který v textu způsobil neshodu.) a "good-suffix heuristic" (tomu v češtině odpovídá asi heuristika dobré přípony. Opět je zjevné, že bude souviset s příponami vzorku).



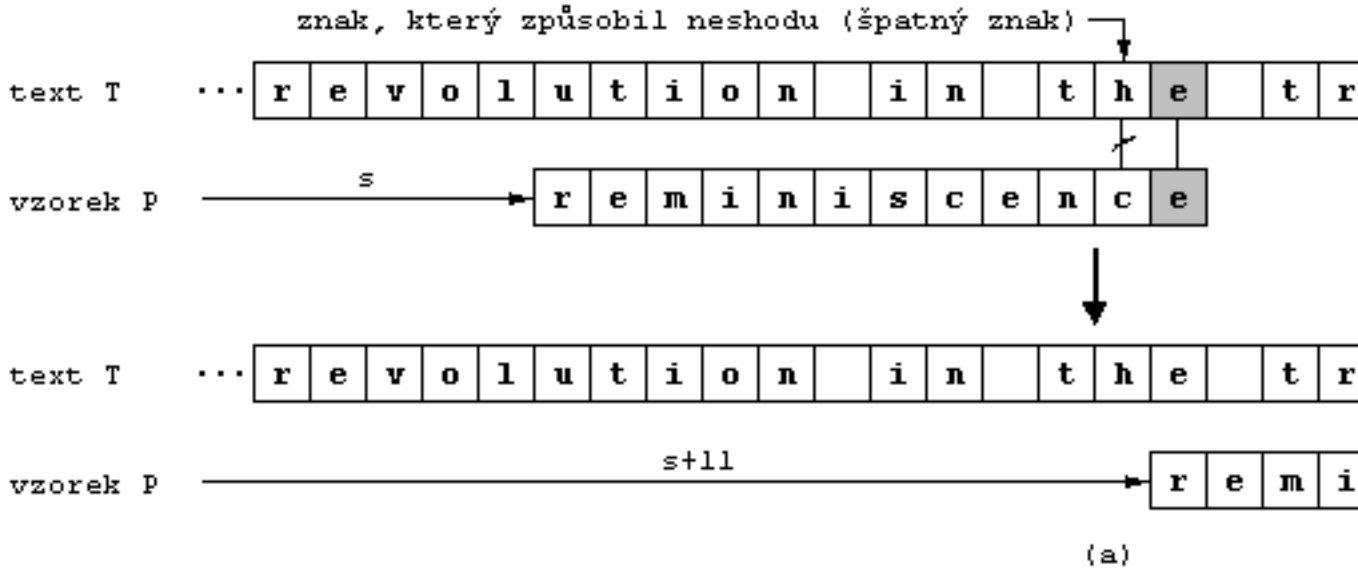
Na obrázku (a) vidíme, že hledáme vzorek **reminiscence** v textu T, z kterého vidíme pouze část. Daný posun **s** je neplatný, neboť na třetím znakuod konce došlo k neshodě (přípomínám, že se vzorek prochází odzadu). Šedě je vyznačena tzv.dobrá přípona **ce**. Je to část vzorku odzadu, která se shoduje s jistou částítextu, vzhledem ke kterému je vzorek zarovnán (jak uvidíme později tato část se dá velmi jednoduše určit a z ní se dá spočítat příslušný posun). Znak **i**, který způsobilneshodu (ve vzorku se na stejném místě vyskytuje písmeno **n**) je již dříveproklamovaný tzv. špatný znak.
Na obrázku (b) je načrtnuto, jak si s neshodou poradí "bad-character heuristic". Ta provedeposun vzorku o tolik pozic doprava, aby špatný znak v textu byl zarovnán k nejpravějšímu výskytustejného znaku ve vzorku. Zde je špatný znak **i**, a jelikož se **i**vyskytuje ve vzorku na sedmé pozici od konce, musím vzorek posunout o čtyři pozice doprava, aby se dosáhlo požadovaného výsledku. U této heuristiky však existují dvě výjimky. Pokud sešpatný znak ve vzorku vůbec nevyskytuje, vzorek se posune o takový počet míst, aby první písmenovzorku bylo zarovnáno k písmenu, které následuje přímo po znaku, který způsobil neshodu.V podstatě jde o nejlepší možný případ a je logické, že pokud je v textu znak, který se v danémvzorku nenachází nemusím se tou částí textu zabývat. Druhou výjimkou je, pokud je nejpravějšívýskyt špatného znaku napravo od aktuální pozice, kde byla odhalena neshoda (vzorek by se tedy posouval doleva). V tomto případě neposkytuje heuristika žádnou možnost.
Na obrázku (c) je zobrazeno chování "good-suffix" heuristiky v případě, že se narazí na neshodu.Tato heuristika provede posun vzorku doprava o nejmenší počet znaků, který zaručí,

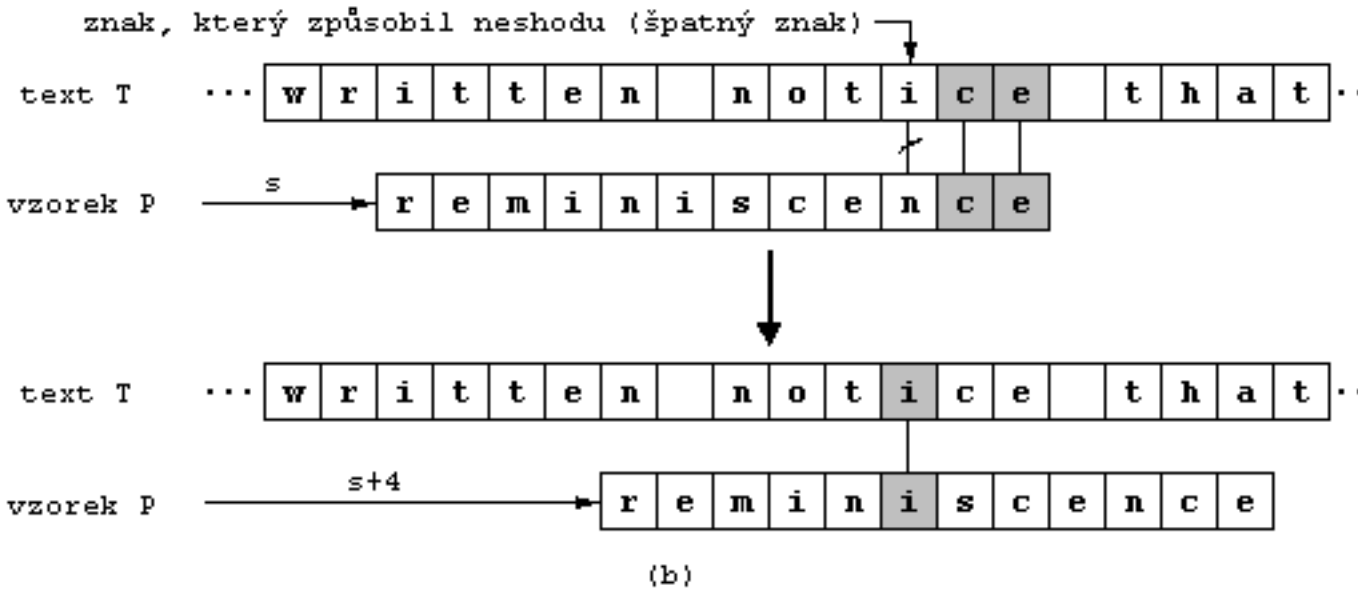
že znakyve vzorku, které se po posunu budou nacházet pod dobrou příponou bude stejné jako znaky v tétopříponě, tedy **ce**. V našem příkladu je to posun o tři pozice doprava.
Když Boyer-Mooreův algoritmus narazí na neshodu, dostane v lepším případě dvě doporučení oddvou heuristik, o kolik je znaků je možno vzorek bezpečně posunout (v lepším případě, nebotheuristika "bad-character" někdy doporučení neposkytne). Algoritmus si tedy logicky vyberevětší posun (v našem příkladě posun vzorku o čtyři pozice doprava).

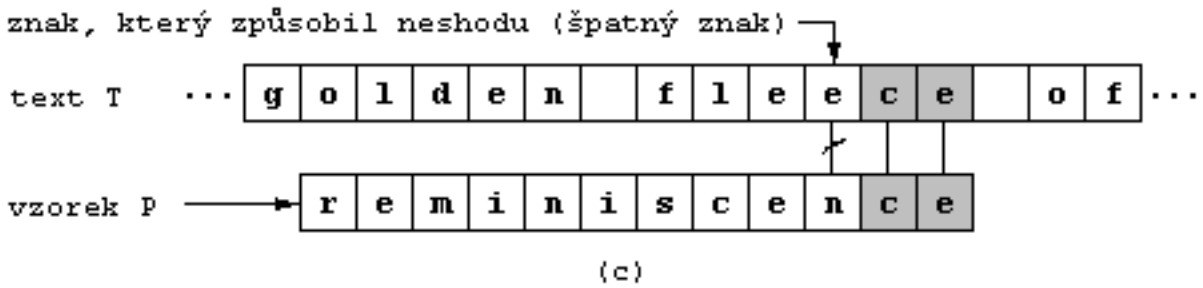
Tuto skutečnost (míněno vybírání a vůbec užití heuristik) je v pseudokódu reflektováno na řádce(12) v případě, že byl nalezen výskyt vzorku, nebo na řádce (13) v případě, že došlo k neshodě. Zde se vybere větší číslo z `j-λ[T[s+j]]` (poskytnuto heuristikou "bad-character") a γ`[j]` (poskytnuto "good-suffix" heuristikou), o které sezvýší posun `s`.
Nyní se podíváme, jak jednotlivé heuristiky přesně fungují a jak se dají spočítat posuny, kteréposkytují. Už nyní je zřejmé, že posuny závisí pouze na vzorku, případně na abecedě Σ.Na prohledávaném textu opět příliš nezáleží.

Heuristika špatného znaku

Už bylo poznamenáno, že u této heuristiky se využívá znalost nejpravějšího výskytu znaku ve vzorku, který způsobil neshodu v textu (`T[s+j]`). Z toho se poté odvodí početznaků, o který se vzorek může posunout doprava. Je zřejmé, že v nejlepším případě, kdy dojdek neshodě hned na prvním porovnávaném znaku (tedy posledním znaku vzorku) a tento špatný znakse ve vzorku nevyskytuje, je možné posunout vzorek doprava o celou jeho délku. Pokud k tomudochází při prohledávání opakovaně, porovná se ve skutečnosti pouhý zlomek celkového počtupísmen, které prohledávaný text obsahuje. Heuristika "bad-character" tedy zajišťuje velmivýrazné urychlení vyhledávacího procesu a to i díky faktu, že porovnávání vzorku s textemse provádí zprava doleva.
Jak tedy heuristika ve skutečnosti pracuje? Nebude škodit, když k odpovědi použijeme trochuformálnějšího zápisu.
Nechť při porovnávání došlo k neshodě. To znamená, že `P[j]!=T[s+j]` pro nějaké `j`, pro které platí `1<=j<=m`. Potom `k` buď největšíčíslo takové, že `1<=k<=m` a zároveň `P[k]==T[s+j]`, pokud takové `k` existuje. Pokud neexistuje, buď `k=0`. Jedná se tedy o nejpravějšívýskyt špatného znaku ve vzorku. Vzorek tedy můžeme bezpečně posunout o `j-k` znaků.V důkazu tohoto tvrzení se rozlišují tři možné případy podle velikosti `k`, které jsou znázorněny na následujícím obrázku.







Na obrázku (a) je ilustrován první případ, kdy se špatný znak `T[s+j]` (v našempříkladě je to písmeno **h**) ve vzorku na jiném místě vůbec nevyskytuje. Vzorekmůžeme tedy bezpečně

posunout o j míst aniž bychom vynechali možnost výskytuvzorku v textu (na obrázku o 11 pozic). Vzorek se zarovná pod písmeno v textu, které následujepřesně za znakem, který způsobil neshodu. Tvrzení v tomto případě platí, neboť k=0a vzorek posuneme o j pozic, což je přesně j-k míst.
Na dalším obrázku (b) je zobrazen další případ, kdy k<j. Nejpravější výskytšpatného znaku ve vzorku je vlevo od místa j, kde došlo k neshodě. Tudíž j-k>0 a vzorek mohu o tento počet míst bezpečně přemístit doprava. Výskytšpatného znaku ve vzorku se potom zarovná ke špatnému znaku v textu. Posun je bezpečný, protože k je index nejbližšího znaku, který se shoduje se špatným, vzhledem k posunu. To znamená, že všechny posuny o velikost menší než j-k jsou neplatné a posunprávě o j-k je v tuto chvíli platný (je možné, že se vyloučí hned v dalším kroku).Na obrázku máme situaci, kdy k=6 a j=10, špatný znak je **i**. Vzorek tedy posunu o čtyři pozice a **i** budou zarovnaná podsebou.
Poslední obrázek (c) znázorňuje i poslední možnost postavení špatného znaku ve vzorku, kdy k>j. Potom by platilo, že j-k<0, což by znamenalo posunutívzorku směrem doleva (návrat zpátky). Tato možnost se v průběhu algoritmu automaticky podchytí,neboť druhá heuristika vždy zaručí posun alespoň o jedno místo a jelikož algoritmus vybírámaximum z obou čísel, vždy se v takovémto případě vybere číslo poskytnuté heuristikou "good-suffix". V našem příkladu je špatný znak **e**, j=10 a k=12.

Nyní si uvedeme jednoduchý pseudokód funkce, která heuristiku "bad-character" realizuje. Funkcedostane na vstup vzorek P, jeho délku m a abecedu Σ, protožeposun se musí spočítat pro každý znak, který se může vyskytnout jako špatný.

```
Last\_Occur\_func(P,m,$\Sigma$)
  (1)  for každý znak a z abecedy $\Sigma$
  (2)    $\lambda$[a]=0
  (3)  for j=1 to m
  (4)    $\lambda$[P[j]]=j
  (5)  return $\lambda$
```

Funkce vrací pole λ, kde λ[a] představuje pozici nejpravějšího výskytu znaku a ve vzorku a to pro všechny znaky z abecedy Σ. V případě,že se znak ve vzorku nevyskytuje je hodnota rovna nule. λ se nazývá last-occurence function čili něco jako funkce posledního výskytu.
Určení časové složitosti je jednoduché. Řádka (2) se provede tolikrát, kolik má abeceda Σznaků, tedy —Σ—-krát. Řádka (4) se provede přesně m-krát. Časová složitostje tudíž *O(—Σ—+m)*.

Heuristika dobré přípony

V tomto odstavci si ukážeme, jak vypočítat posuny doporučované druhou heuristikou, heuristikou"good-suffix". Pro tento účel si definujme relaci Q~R pro dva textové řetězce Q a R, pro které platí, že buď Q je příponou R nebo R je příponou Q. Tato relace neznamená nic jinéhonež, že pokud oba řetězce zarovnáme pod sebe podle pravého okraje, budou se ve znacích podsebou shodovat. Zároveň platí, že Q~R právě tehdy, když R~Q.

Další vztah:
Jestliže Q je příponou R a zároveň S je příponou R, potom Q~S. Slovy řečeno to znamená, že pokud je Qpříponou R a nějaké S je také příponou R, tak je jasné,že řetězce Q a S mají určitý počet znaků stejných. Tedy buď Q je příponou S nebo S je příponou Q.To je ale Q~S podle definice ~.

Nechť při porovnávání došlo k neshodě na j-tém místě vzorku (tedy P[j]!=T[s+j]), pro nějaké j<m. Potom heuristika "good-suffix" říká,že vzorek mohu bezpečně posunout o

vzdálenost

```
$\lambda$[j]=m-max\{k: 0$<$=k$<$m \& P[j+1..m]\textasciitilde{}P$_k$\}
```

Tedy λ[j] je nejmenší vzdálenost, o kterou můžeme vzorek posunout, anižbychom způsobili nějakou neshodu dobré přípony T[s+j+1..s+m] vůči odpovídajícímznakům nově posunutého vzorku. Tuto situaci si můžeme ukázat na obrázku (b) s předchozí kapitoly. K neshodě došlo na třetím znaku vzorku od konce, tedy j=3. Dobrá příponaje tedy slovo **ce**, poslední dvě písmena vzorku **reminiscence**.Z definice λ hledáme největší k, které splňuje, že P[j+1..m]~P_k. V našem případě je k=9, neboť P[j+1..m] je slovo **ce** (dobrá přípona) a nejdelší předpona vzorku P končící **ce** je slovo **reminisce**, jehož délka jedevět. Vzorek tedy můžeme posunout o m-k=12-9=3 pozice doprava.
Ještě poznamenejme, že funkce λ je dobře definována pro všechna j, neboť P[j+1..m]~P_0 pro všechna j (prázdný řetězec je v relacise vším). λ se nazývá good-suffix function, v překladu funkce dobré přípony.
Jelikož je naše definice této funkce pro účely výpočtu na počítači poněkud nevhodná, provedeměnekolik relativně jednoduchých úprav, abychom dostali ekvivalentní definici, ale ve tvaru,v kterém půjde přepsat do našeho pseudokódu.
Nejprve si ukážeme, že platí vztah λ[j]<=m-π[m] pro všechna j, kde π je prefixová funkce, kterou jsme použili u KMP algoritmu. Položme w=π[m]. Z definice prefixové funkce máme, že musí platit, že P_w je příponou vzorku P. Protože P[j+1..m] je také příponou P, dostaneme ze vztahu uvedeného výše, že nutně P[j+1..m]~P_w. Podle definice λ platí, že λ[j]<=m-w (neboť mám w, které splňuje požadavky, ale nemusí to být maximální takové číslo, proto je možné, že λ[j] bude menší než m-w). Jelikož máme w=π[m], plyne odtud rovnou vztah λ[j]<=m-π[m] pro všechna j, což jsme chtěli dokázat.Díky tomu můžeme naši definici funkce λ přepsat do následující podoby.

```
$\lambda$[j]=m-max\{k: $\pi$[m]$<$=k$<$m \& P[j+1..m]\textasciitilde{}P$_k$\}
```

Tuto definici jsme z předchozího dostali následovně.

```
(1)  $\lambda$[j]=m-max\{k: 0$<$=k$<$m \& ...\} $<$= m-$\pi$[m]
(2)  $\pi$[m] $<$= max\{k: 0$<$=k$<$m \& ...\}
(3)  k $>$= $\pi$[m]
```

Úprava mezi (1) a (2) je triviální. Vztah (3) plyne z (2), neboť největší k budevždy větší než π[m], proto takhle omezené k mohu hledat od začátku.

Pokračujme v úpravách naší nové definice λ. Z podmínky P[j+1..m]~P_k vyplývá, že buď P[j+1..m] je příponou P_k nebo P_k je příponou P[j+1..m]podle definice ~. Druhá možnost přímo implikuje, že P_k je příponoucelého vzorku P (P_k je příponou P[j+1..m],což je ale přípona P. Je tedy jasné, že i P_k je příponou P). Odtud dostaneme vztah, že k<=π[m] z definice π(máme, že P_k je příponou P a zároveň π[q]=max{k: k<q & P_k je přípona P_q}. Spojením těchtodvou faktů dojdeme ke vztahu π[m]=max{k: k<m & P_k je přípona P}.Odtud plyne, že π[m]>=k, neboť π[m] se rovná maximu, tedypro ostatní k je jistě větší.). Z této nerovnosti plyne λ[j]>=m-π[m] a to následovně.

```
k $<$= $\pi$[m]
m-$\pi$[m] $<$= m-k   pro všechna k
m-$\pi$[m] $<$= m-max\{k: ...\}=$\lambda$[j]
m-$\pi$[m] $<$= $\lambda$[j]
```

Definici λ můžeme dále upravit.

```
$\lambda$[j]=m-max(\{$\pi$[m]\} sjednoceno s \{k: $\pi$[m]$<$k$<$m \& P[j+1..m] je přípo
```

Odtud plyne významná skutečnost, λ[j]>0 pro všechna j (z definice plyne, že buď bude λ[j]=m-π[m] a to je určitě kladné(víme, že π[m]<m), nebo λ[j]=m-k (k je maximem z druhémnožiny), v tomto případě je ale λ[j] také kladné, neboť k<m). To je věc, kterou jsme potřebovali, protože zaručí, že BM algoritmusbude posunovat vzorek stále doprava (i v případě, že první heuristika vrátí záporné číslo).
Pokračujme v naší snaze zjednodušit definici funkce λ dále. Pro další účely si zavedemeobrácený vzorek P' vzorku P a tomu odpovídající prefixovou funkci π'. Potom P'[i]=P[m-i+1] pro i=1,...,m a π'[t] jenejvětší u takové, že u<t a zároveň P'$_u$je příponou P'$_t$.
Nechť k je největší číslo takové, že P[j+1..m] je příponou P$_k$, potom π'[l]=m-j, kde l=(m-k)+(m-j).
Z toho, že P[j+1..m] je příponou P$_k$, plyne m-j<=k (P[j+1..m] jako přípona P$_k$ nemůžebýt delší než P$_k$ a délka P[j+1..m] je m-j)a l<=m (neboť l=(m-k)+(m-j) a předchozí nerovnost).
Také platí, že j<m a k<=m, z čehož plyne l>=1(l=(m-k)+(m-j). První závorka je díky první nerovnosti kladná, druhá závorka jedíky druhé nerovnosti nezáporná. Celé je to tedy větší nebo rovno jedné.).Jelikož 1<=l<=m je funkce π' dobře definována.
Nyní si dokážeme tvrzení π'[l]=m-j. Jelikož P[j+1..m] je příponou P$_k$, máme také P'$_m - j$ je příponou P'$_l$ (pouhé obrácení a přeindexování). Odtud dostaneme π'[l]>=m-j (neboť m-j vyhovuje definici π', hodnota všakmůže být díky maximalizaci i větší.). Pro spor předpokládejme, že p>m-j, kde p=π'[l]. Podle definice π' máme, že P'$_p$ je přípona P'$_l$. To se však dá napsat také jako P'[1..p]=P'[l-p+1..l].Přepisem vzhledem k původnímu vzorku získáme P[m-p+1..m]=P[m-l+1..m-l+p]. Pokud teď použijeme substituci l=2m-k-j, dostaneme P[m-p+1..m]=P[k-m+j+1..k-m+j+p]. Tedy P[m-p+1..m] je přípona P$_k$−m+j+p.
Protože p>m-j, pak j+1>m-p+1,a tedy P[j+1..m] je přípona P[m-p+1..m]. Celkem máme fakt, že P[j+1..m] je přípona P$_k - m + j + p$ (to plyne z tranzitivity"operace suffixování" (A je přípona B a B je přípona C, potom A je příponou C)).
Protože p>m-j, máme k'>k, kde k'=k-m+j+p. Jelikož k' splňuje definici π' a dokonce k'>k, docházíme ke sporu s tím, že k je největší číslo splňující definici π'.
Tedy p=π'[l]=m-j a tvrzení je dokázáno.

Díky tvrzení máme π'[l]=m-j, z toho plyne j=m-π'[l] a dosazenímdo l=(m-k)+(m-j) dostaneme k=m-l+π'[l]. Díky tomu můžeme lépe přepsat definici λ.

```
$\lambda$[j]=m-max(\{$\pi$[m]\} sjednoceno s \{m-l+$\pi$'[l]: 1$<$=l$<$=m \& j=m-$\pi$'[
    =min(\{m-$\pi$[m]\} sjednoceno s \{l-$\pi$'[l]: 1$<$=l$<$=m \& j=m-$\pi$'[l]\})
```

Tato definice je již natolik dobře formulována, že se dá přímo přepsat do pseudokódu. Funkcedostane na vstup vzorek P a jeho délku m.

```
Good\_suff\_func(P,m)
  (1)  $\pi$=Prefix\_func(P)
  (2)  P'=Obrat(P)
  (3)  $\pi$'=Prefix\_func(P')
  (4)  for j = 0 to m
  (5)    $\gamma$[j]=m-$\pi$[m]
  (6)  for l = 1 to m
  (7)    j = m-$\pi$'[l]
  (8)    if $\gamma$[j] $>$ l-$\pi$'[l]
```

```
 (9)      then $\gamma$[j] = l-$\pi$'[l])
(10)  return $\gamma$
```

Časová složitost této procedury je *O(m)*. Časová složitost v nejhorším případě celého BM algoritmu je tedy *O((n-m+1)*m+—Σ—)* (složitost obou heuristik dohromady je *O(m+—Σ—)* a vyhledávací fáze je v podstatě naivní algoritmus). Ve skutečnosti se v běžné praxi dosahuje mnohem lepších výsledků a tento algoritmus je velmi používaný.
Postupem času se objevilo několik úprav tohoto algoritmu a dosáhlo se lineární složitosti iv nejhorším případě.

>>>Obsah

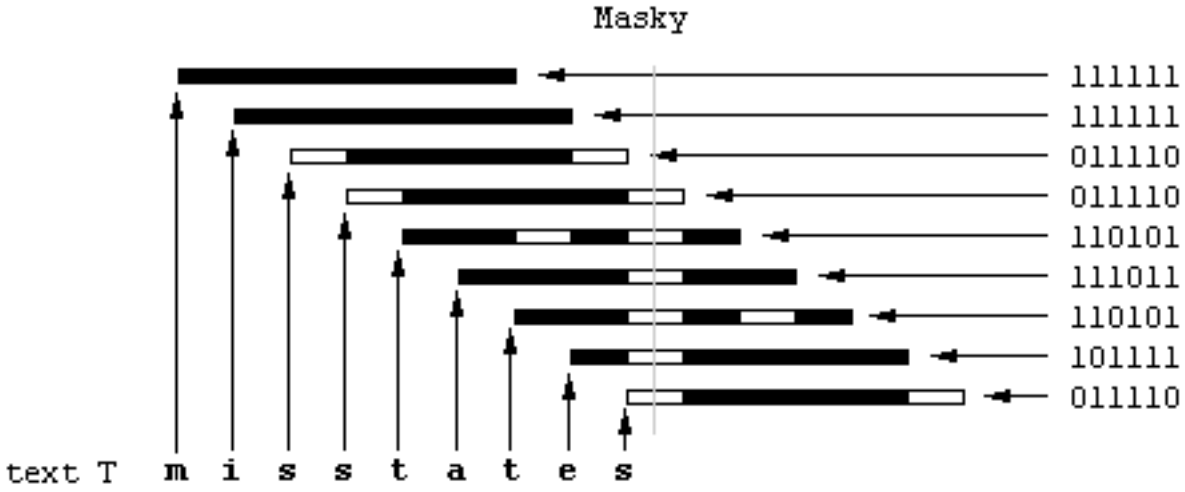
4. Baeza-Yates-Gonnetův algoritmus (BYG)

Nyní si předvedeme algoritmus, který se od předchozích liší už svým přístupem k vyhledávánív textu. Byl publikován v roce 1992 a autory jsou R. Baeza-Yates a G.H. Gonnet. Narozdíl od dvou algoritmů, které jsme si již objasnili, a kde se vyhledává pomocíporovnávání znaků ve vzorku a v textu, algoritmus BYG přišel s ideou vyhledávání pomocí bitovýchmasek. Princip je celkem jednoduchý. I v tomto algoritmu se používá předpočítaná tabulka, nyníje to však tabulka bitových vektorů pro každý znak ze vstupní abecedy Σ. Každá bitovápozice v daném vektoru pro dané písmeno odpovídá pozici tohoto písmena ve vzorku. Z toho plyne,že každý vektor musí být tak dlouhý jako je daný vzorek. Vektor je posloupnost jedniček, alepokud se znak odpovídající tomuto vektoru vyskytuje na k-té pozici ve vzorku, jena k-té pozici ve vektoru číslo nula.
Zde existuje několik možných variant, jak se vektory konstruují. Buď se pozice vektoru číslujíodleva nebo odprava. Někdy jsou vektory nulové a výskyt znaku ve vzorku je značen jedničkou.Nejběžnější je však způsob, který jsme si ukázali a to je číslování pozic odprava a výskyt je značen nulou. Ukažme si na příkladu jak taková tabulka vektorů vypadá pro slovo **states** za předpokladu klasické abecedy Σ ={a,b,...,z}.

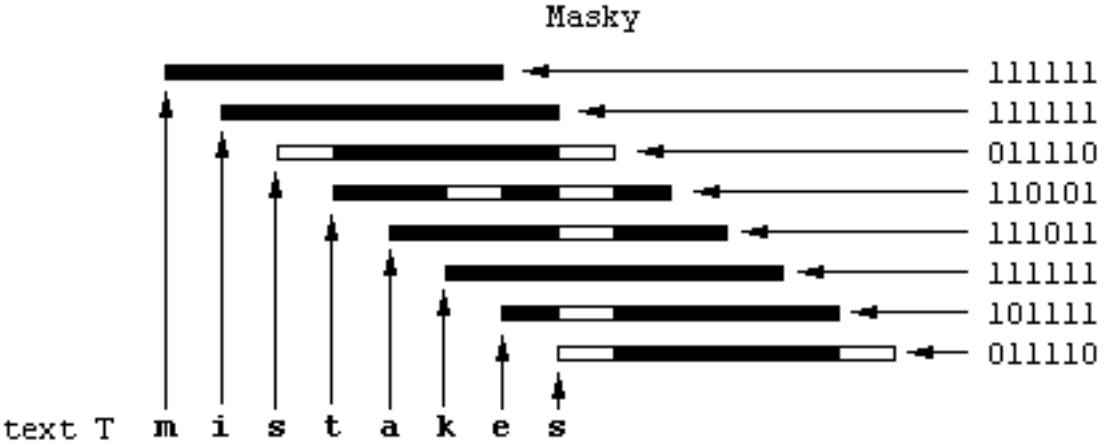
Znak	Pozice ve vzorku 654321
a	111011
b	111111
c	111111
d	111111
e	101111
f	111111
...	...
r	111111
s	011110
t	110101
u	111111
...	...

Například písmeno **s** se ve vzorku vyskytuje na první a šesté poslední pozici, v odpovídajícím vektoru je tudíž první a šestý bit nastaven na nulu.
Na všechny tyto vektory se můžeme podívat jako na masky, kde nula je transparentní ("průhledná")a jednička netransparentní. Tyto masky potom můžeme zarovnat k danému prohledávanému textu.Uveďme si příklad. Nechť máme text **misstates**. Potom když k písmenům zarovnámeodpovídající

masky dostaneme následující obrázek.



Na obrázku je každá maska zarovnána k odpovídajícímu písmenu. To znamená maska pro **m** je zarovnána k písmenu **m** v textu, maska pro **i** je zarovnána k **i** atd. Jelikož se v textu vzorek **states** nachází je na obrázku seřazeno šest transparentních buněk pod sebou.Šedá čára znázorňuje paprsek světla, kterým by se daly jednotlivé buňky prosvítit z jedné stranyna druhou.
Abychom viděli rozdíl mezi tím, kdy se vzorek v textu najde a mezi okamžikem, kdy dojde k neshodě, ukážeme se další příklad. Tentokrát je prohledávaným textem slov **mistakes**.



Na obrázku je vidět, že maska pro písmeno **k** zamezuje prosvícení blokutransparentních bitů. To znamená, že v textu se vzorek na tomto místě nenachází.
Nyní si ukážeme, jak by mohl vypadat kód pro hledání vzorku v textu algoritmem BYG (následujícíkód vyhledá pouze první výskyt, pro hledání všech výskytů je potřeba kód částečně upravit).Nejprve však několik poznámek. Zmíněné prosvícení buněk se v programu provádí, ovšem digitálněpomocí bitových operací. Jednotlivé masky se shromažďují v proměnné `work`, jejížpočáteční hodnota je `-1` (v dvojkovém doplňku jsou to samé jedničky). S každýmnovým znakem v textu se tato proměnná posune o jeden bit doleva (vynásobí se dvěma), cožodpovídá odsazování masek na obrázcích. Maska pro nový znak se k proměnné `work`přidá pomocí operace OR. V každém cyklu se poté tato proměnná testuje na příslušném bituna nulovost

(neboť operace OR zachovává nulovost bitů u znaků, které se shodují se vzorkem).Pokud se nula v bitu vyskytuje je vzorek nalezen, v opačném případě se pokračuje dále.
Program dostává na vstup text `T` a vzorek `P`. Nejprve se musí vytvořittabulka bitových masek a definovat bit, který se bude testovat na nulovost (je to vlastně bits indexem délky vzorku).

```
 (1)  n = length(T)
 (2)  m = length(P)
 (3)  masks = Comp\_masks(P, $\Sigma$)
 (4)  testbit = 2\textasciicircumm
 (5)  i = 1
 (6)  found = false
 (7)  while (not found \&\& i $<$ n)
 (8)    while (i $<$ n \&\& T[i] != P[1])  i = i+1
 (9)    work = -1
(10)    while (work != -1 \&\& not found)
(11)      work = (work shl 1) or masks[T[i]]
(12)      if (work and testbit = 0)
(13)        then found = true
(14)             print("Vzorek byl nalezen na pozici", i+1-m)
(15)        else i = i+1
```

Cyklus na řádce (8) prochází text do doby než najde počáteční znak vzorku, odtud začíná hledatvzorek podle masek. V cyklu (10)-(15) se provádí vlastní činnost, tedy posouvání a OR-ovánímasky spolu s testem na přítomnost vzorku.
Abychom si lépe ukázali, jak algoritmus funguje (podle kódu, který jsme se sestavili a ne podlejakési představy z předchozích obrázků), probereme další příklad. Budeme hledat vzorek **state** v textu **misstates**. Program tedy podle řádky (8) projdetext až narazí na znak **s**, což je počáteční písmeno našeho vzorku. Potom seproměnná `work` nastaví na výchozí hodnotu -1. Následně se proměnná posune o jeden bit doleva (nejpravější bit bude nula) a přioruje se maska pro písmeno **s**.Jelikož tato maska obsahuje nulu na nejpravějším bitu, operace OR tuto hodnotu zachová.Abychom viděli, co se bude dít dál, uvedeme si tabulku. Hodnota Posun bude vyjadřovat proměnnou `work` posunutou o jeden bit doleva, políčko Maska představuje masku pro danépísmeno a políčko Výsledek je výsledkem operace OR na předchozí dvě hodnoty. Bit, který se testuje na nulovost je zvýrazněn.

Vstupní znaky	Bitové hodnoty
s	1111111111111110 Posun 1111111111111110 Maska 1111111111111110 Výsledek
s	1111111111111100 Posun 1111111111111110 Maska 1111111111111110 Výsledek
t	1111111111111100 Posun 1111111111110101 Maska 1111111111111101 Výsledek
a	1111111111111010 Posun 1111111111111011 Maska 1111111111111011 Výsledek
t	1111111111110110 Posun 1111111111110101 Maska 1111111111110111 Výsledek
e	1111111111101110 Posun 1111111111101111 Maska 1111111111101111 Výsledek

Dále algoritmus nepokračuje, protože vzorek byl v tomto okamžiku odhalen otestováním příslušnéhobitu na nulu. Nyní si ukážeme obdobnou tabulku pro text, který vzorek neobsahuje, např.**mistakes**.

Vstupní znaky	Bitové hodnoty
s	1111111111111110 Posun 1111111111111110 Maska 1111111111111110 Výsledek
t	1111111111111100 Posun 1111111111110101 Maska 1111111111111101 Výsledek
a	1111111111111010 Posun 1111111111111011 Maska 1111111111111011 Výsledek
k	1111111111110110 Posun 1111111111111111 Maska 1111111111111111 Výsledek

Program v tuto chvíli skončí vnitřní cyklus, neboť `work=-1`, což znamená, žebyla nalezena neshoda. Dále by program hledal první znak vzorku (**s**) v textu.Nyní si ukažme jednoduchý kód, který předpočítá masky pro všechny znaky abecedy Σ.Procedura dostane na vstup vzorek `P` a abecedu Σ.

```
(1)  for každý znak a z abecedy $\Sigma$
(2)    masks[a] = -1
(3)  j = 1
(4)  for i = 1 to m
(5)    masks[P[i]] = masks[P[i]] and not j
(6)    j = j shl 1
(7)  return masks
```

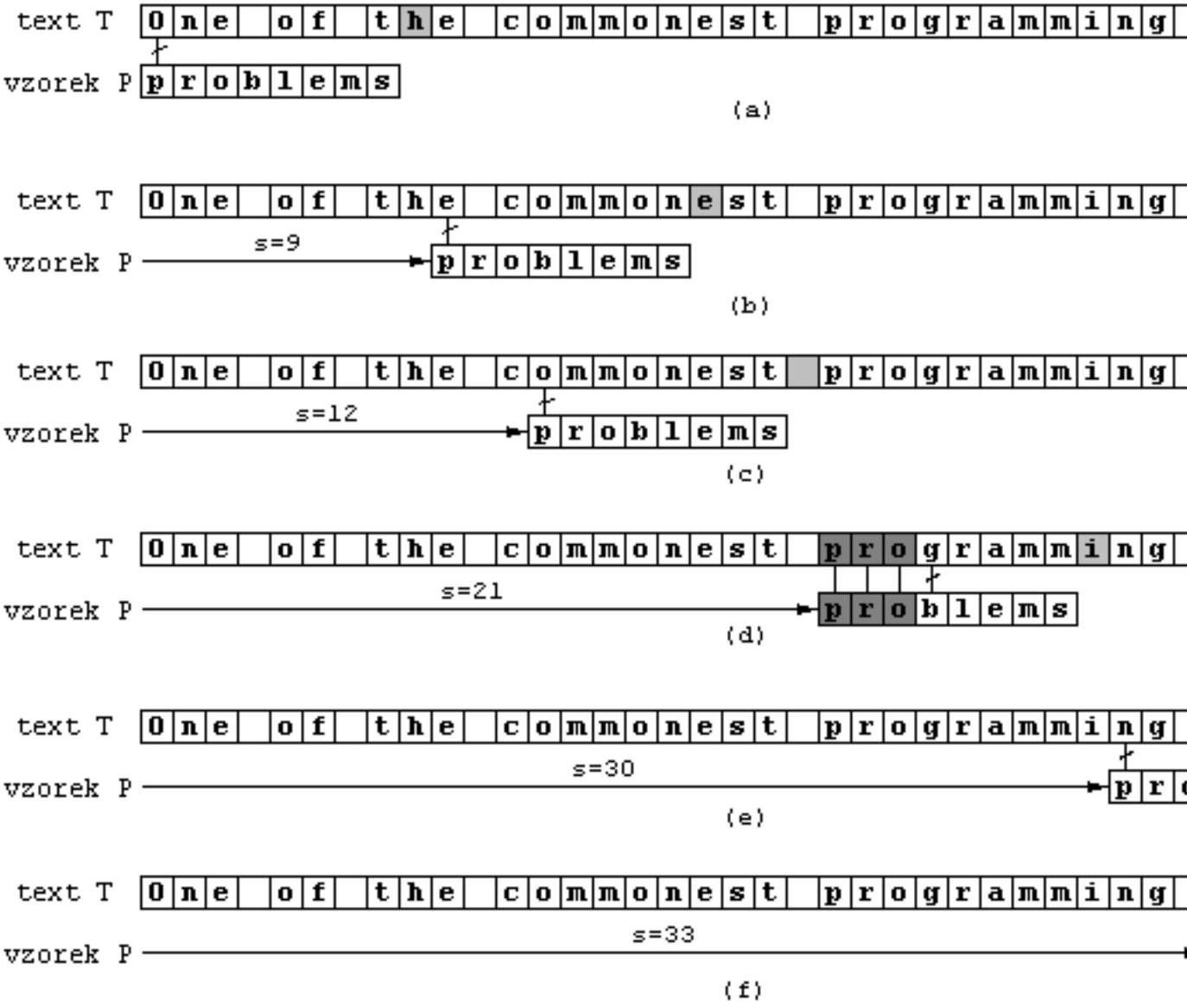
Proměnná `j` slouží k vyznačování nul v příslušných vektorech. Počáteční hodnotouje jednička a s každým průběhem cyklu se hodnota zdvojnásobí (posun o jeden bit doleva).Například pokud `j=4`, což je v bitovém zápisu `0000000000000100`. Tím,že operaci AND použijeme na masku, kterou máme, a na dvojkový doplňek `j`(`1111111111111011`), vyznačíme nulu na třetím bitu masky.
Časová složitost této procedury je *O(m)*, protože cyklus se vykoná pro každý znak `m`-krát (cykly jsou sice dva, ale složitost *O(2*m)* odpovídá *O(m)*z definice *O*). Časová složitost samotného vyhledávání je *O(n)* v nejhorším případě. Celková složitost je tedy *O(n+m)*.

>>>Obsah

5. Quicksearch

V roce 1990 publikoval člověk jménem D.M. Sunday algoritmus Quicksearch, který se od předchozíchvýznamně liší ve dvou věcech. Je rychlejší a mnohem jednodušší. Některé jeho rysy jsou podobnéjako u Boyer-Mooreova algoritmu. U BM algoritmu totiž v nejlepším případě přeskočíme tolik znakůkolik je délka samotného vzorku. U Quicksearch se, jak uvidíme později, nejčastěji přeskakuje `m+1` znaků (kde `m` je délka vzorku). Znamená to, že časová složitostv průměrném případě je méně než *O(n)* a blíží se k *O(n/(m+1))*. To je důležitézvlášť pro delší vzorky, kde je urychlení opravdu markantní. U Boyer-Mooreva algoritmu sedíky tomu, že vzorek porovnáváme odzadu, v textu vracíme, což může způsobit problémy s paměťovýmbufferem, které byly popsány v úvodu Knuth-Morris-Prattova algoritmu. Quicksearch nic takovéhonedělá, takže se jeví jako bezproblémový. Má však jiné nevýhody, které jsou popsané v dalšíkapitole.
Myšlenka tohoto algoritmu je opravdu velice jednoduchá. Na začátku jako obvykle zarovnáme vzorekk prohledávanému textu. Stejně jako u naivního algoritmu budeme text a vzorek porovnávat znakpo znaku. Postup se ale liší v případě, že objevíme neshodu mezi jednotlivými písmeny. V tomtookamžiku se podíváme na znak, který se nachází v textu přímo za koncem vzorku (testový znak).Pokud se tento znak ve vzorku na žádném místě neobjevuje, žádný

posun, který by umístil jakékolipísmeno vzorku nad testový znak, nebude platný. S klidným svědomím tedy můžeme celý vzorek přemístit až za testový znak. To představuje posun o `m+1` znaků, kde `m`je velikost vzorku. Tato vzdálenost je mnohem lepší než v případě předchozích algoritmů (včetně BM algoritmu, kde byl posun v tomto případě pouze `m`).
Pokud se testový znak ve vzorku na nějakém místě nachází (případně na více místech), posunemevzorek o nejmenší vzdálenost takovou, že se testový znak bude shodovat se znakem v nově posunutém vzorku, který je zarovnán k testovému znaku. Většinou to bude představovat posuno více než jeden znak. Dalším porovnáním zjistíme, jestli byl posun spravný a vzorek se na této pozici již nachází. Jinak se posuneme stejným způsobem dále. Pro lepší ilustraci, jaktento algoritmus v textu vyhledává, si uvedeme příklad. V následujícím textu budeme hledt výskytvzorku **problems**.



Na obrázku (a) je vyobrazena výchozí situace. Hned první znak textu způsobuje neshodu. Testový znak je písmeno **h**. Jelikož se takové písmeno ve vzorku nevyskytuje, posunemevzorek

o jeho délku zvětšenou o jedna (tedy o devět znaků). To znamená, že se vzorek posuneaž za písmeno **h**.
Situace je ilustrována na obrázku (b). Tentokrát je testový znak písmeno **e**, které se ve vzorku vyskytuje. Proto musíme vzorek posunout tak, aby byly dvě **e**zarovnány pod sebe. Tato vzdálenost musí být o jednu větší než je **e** vzdálenood konce vzorku. Vzorek tedy posuneme o `8-6+1=3` znaky.
Na obrázku (c) opět došlo k neshodě na prvním porovnávaném znaku. Testový znak je prázdné políčko, které se ve vzorku nevyskytuje. Znovu posuneme vzorek o devět míst.
Následuje situace z obrázku (d). Zde se stejně jako u naivního algoritmu porovnají tři znaky(**pro**). Na čtvrtém písmenu dojde k neshodě. Protože testový znak **i** se ve vzorku nevyskytuje, přemístíme vzorek až za tento testový znak, tedyopět o devět míst.
Obrázek (e). Zde je opět testovým znakem písmeno **e** a stejně jako v případě (b),posuneme vzorek o devět míst.
Na obrázku (f) je zobrazena konečná situace. Vzorek je nalezen a bylo k tomu potřeba pouhýchpět posunů a celkem 16 porovnání.

Nyní se už můžeme uvést jak bude algoritmus vypadat zapsán v pseudokódu. Na vstup proceduradostane text `T`, vzorek `P` a abecedu Σ. Procedura opět vyhledá pouze první výskyt pro nalezení všech výskytů jsou však třeba jen drobné úpravy. Tabulku posunů `shift` je nutno před samotným vyhledáváním předpočítat.

```
 (1)  n = length(T)
 (2)  m = length(P)
 (3)  shift = Comp\_shift(P,$\Sigma$)
 (4)  pat = 1
 (5)  s = 0
 (6)  while (pat $<$= m \&\& pat+s $<$= n)
 (7)    if P[pat] == T[pat+s]
 (8)      then pat = pat+1
 (9)      else s = s+shift[T[s+m+1]]
(10)           pat = 1

Comp\_shift(P,$\Sigma$)
 (1)  for každý znak a z abecedy $\Sigma$
 (2)    shift[a] = m+1
 (3)  for i = 1 to m
 (4)    shift[P[i]] = m-i+1
 (5)  return shift
```

Kód je pouze přepisem faktů, které tu byly vysvětleny. Snad jen poznámka k funkci Comp_shift.V prvním cyklu se všem znakům přiřadí hodnota maximálního posunu a teprve poté se provádí proznaky, které se ve vzorku vyskytují, přesnější úprava.
V nejlepším případě se neshoda objeví pokaždé hned u prvního porovnávaného znaku (u prvního myšleno jako prvního po každém posunu). To znamená, že posun bude pokaždé o `m+1` znaků. Časová složitost je potom asi *O(n/(m+1))*. Kompletní analýza časové složitosti tohoto algoritmu zatím nebyla poskytnuta, ale Sunday tvrdí, že není horší než *O(n)*.

>>>Obsah

IV. Srovnání algoritmů

V této kapitole si řekneme výhody a nevýhody algoritmů, které zde byly popsány. Na jaké účely se hodí a na jaké ne.
První algoritmus, kterému jsme se věnovali, byl tzv. naivní algoritmus (odkaz). Myslím, že nemá cenu se tomuto algoritmu věnovat moc dlouho, neboť svou časovou složitostí není předurčen k příliš velkému použití. Na druhou stranu pro relativně krátké texty (řádově o stovkách maximálně tisících znacích) a krátké vzorky (20 písmen) je tentoalgoritmus asi dobrou volbou, a to už jenom z důvodu, že ostatní efektivní algoritmy si pro své potřeby předpočítávají různé tabulky, což tento algoritmus nedělá, a proto se u krátkých textů jeho pomalost neprojeví a vzhledem ke své jednoduchosti implementace je mnohdy používán (implementace v assembleru, použití v jednoduchých textových editorech apod.). Jeho nevýhodou je, že při použití pro vyhledávání v souborech může dojít k problému s paměťovým bufferem, který je popsán v úvodu kapitoly věnované Knuth-Morris-Prattovu algoritmu.
Dalším algoritmem, který jsme si ukazovali, je právě Knuth-Morris-Prattův algoritmus (odkaz). Tento algoritmus odstraňuje hlavní nevýhody naivního algoritmu. Je to především vícenásobné testování jednoho znaku a vracení se v textu. Díky tomu je vhodný pro vyhledávání v textových souborech, neboť nevznikají problémy s paměťovým bufferem (pravděpodobnost, že se tento problém vyskytne, je sice malá, ale přihodit se může). Nevýhodou ale je, že se stále porovnávají všechny znaky textu (i když jak uvidíme později, pro jisté speciální účely je to nezbytné). Algoritmus se hodí pro vyhledávání vzorků, o kterýchnejsou k dispozici žádné informace, a tedy není jisté, jestli by bylo výhodnější použít obyčejný naivní algoritmus.
V další kapitole jsme se věnovali algoritmu Boyera a Moorea (odkaz). Tento algoritmus je asi všeobecně nejlépe použitelný i přes svou relativní složitost implementace.Je zvláště vhodný pro vzorky větší délky a pro relativně velkou abecedu znaků, kde se oběheuristiky mohou plně uplatnit. Narozdíl od předchozích algoritmů, tento prozkoumá pouzezlomek všech znaků v textu (to se ale v některých případech může nevyplatit). Bohužel díkyzpětnému porovnávání vzorku s textem může dojít stejně jako u naivního algoritmu k problémůms paměťovým bufferem, který vyžaduje další režii výpočtu. I přes nepříliš dobrou časovou složitost je v průměru velmi dobrý.
Následuje algoritmus Baeza-Yates-Gonnet (odkaz). Tento algoritmus je velmispecifický, a proto jsou s ním spjatá i jistá omezení co do implementace. Prvním omezením jepožadavek na schopnost programovacího jazyka (a počítače) provádět bitové operace OR, AND a bitový posun, na číslech typu `integer`. Druhým a po pravdě řečeno asi drastičtějším omezením je délka bitových masek, které se v algoritmu používají. Tyto masky musí mít stejnou délku jako daný vzorek. Dnešní počítače počítají vše buď v 32-bitové nebo v 64-bitové aritmetice, což je omezení pro kompilátory programovacích jazyků a tím i pro délku bitové masky. Proto v případě, že chceme hledat vzorek delší než 32 (potažmo 64) bitů, není tento algoritmus i přes svou rychlost tou správnou volbou. Na druhou stranu, pokud chceme sestrojit algoritmus, který nebude case-sensitive (tedy nebude rozlišovat velikost písmen), není problém upravitBaeza-Yates-Gonnetův algoritmus tak, aby tento požadavek bez problémů řešil. Stačí pouze vyrobit masky zvlášť pro velká a malá písmena a trochu upravit vyhledávací část.
Tím se dostáváme k poslednímu algoritmu, který zde byl popsán, Quicksearch (odkaz). Nejdříve dvě fakta. Za prvé tento algoritmus se stejně jako KMP a BYG v textu nevrací zpět. Za druhé, stejně jako Boyer-Mooreův algoritmus i tento přeskakuje velké množství neporovnaných znaků. Jak bylo řečeno, je tato skutečnost výhodou co do urychlení algoritmu, ale ve speciálních případech není toto přeskakování žádoucí. Takovým příkladem může být situ-

ace, kdy při každém nalezení vzorku v textu chceme, aby program nahlásil číslo řádky, kde se vzorek vyskytuje. V případě, kdy program počítá každý znak pro novou řádku a tento znak se posunutím vzorku o několik pozic přeskočí, dochází při nahlášení výskytu vzorku ke zkreslení informace o čísle řádky. Proto je při výběru algoritmu nutné uvažovat, k čemu bude sloužit.Testy ukázaly (na textech o délce přibližně 200000 znaků), že Quicksearch si velmi dobře počíná v situacích, kdy je vzorek relativně delší. Při nejčastějších délkách vzorků (od šesti do osmipísmen) porovná algoritmus pouze přibližně jednu šestinu všech znaků. Záleží však i na tom, jakvzorek vypadá. Existují dvě upravené verze (maximal shift algorithm a optimal shiftalgorithm), které dosahují o pět procent lepších výsledků než základní algoritmus. Vzhledemk tomu, že před vlastním vyhledáváním předpočítávají spoustu různých věcí, jsou ale vhodné pouze pro texty délky řádově o stotísících znacích.

>>>Obsah

V. Závěr

V tomto dokumentu jsme popsali několik algoritmů zabývajících se vyhledáváním vzorků v textu.Všechny mají jednu vlastnost společnou, vyhledávají jeden vzorek v textu. Samozřejmě existujíi algoritmy, které vyhledávají celé množiny vzorků (algoritmus Aho-Corasickové, algoritmusCommentz Walterové) i množiny zadané pomocí regulárních výrazů (B-algoritmus). I přesto jsoutyto algoritmy velice užitečné a svou vzájemnou rozdílností je spektrum jejich použitelnostipoměrně široké. Dále jsme si uvedli výhody a nevýhody jednotlivých algoritmů, k čemu sehodí a k čemu. U každého je vedle podrobného vysvětlení i pseudokód, podle kterého by nemělbýt problém daný algoritmus naprogramovat.
Tento dokument je tedy jakýsi úvod do problematiky vyhledávání v textu, neboť tato úlohaje velice rozsáhlá a zasahuje do mnoha oborů teoretické informatiky.

>>>Obsah

VI. Literatura

Zde jsou uvedeny použité materiály.

- T.H.Cormen, C.E.Leiserson, R.L.Rivest: Introduction to Algorithms, MIT Press 1990, kapitola34 - String matching
- Ivana Vovsová: Vyhledávání vzorků v textu, diplomová práce, MFF 1994
- String Searching, článek z neznámé knihy od neznámého autora
- A. Koubková, J. Pavelka: Úvod do teoretické informatiky, MFFPress 1998

>>>Obsah