Vyhľadávanie v texte

Broňa Brejová, KI FMFI UK

20. mája 2013

Toto sú poznámky z predmetu Vyhľadávanie v texte, ktoré z časti spísali študenti podľa prednášok v školských rokoch 2008/09 až 2012/13. Poznámky môžu obsahovať mnoho nepresností a navyše sa obsah predmetu z roka na rok mierne mení. Preto je ich použitie na vlastné nebezpečie.

Moja vďaka patrí nasledujúcim študentom, ktorí poznámky zapísali: Vladimír Boža, Viliam Dillinger, Marcel Ďuriš, Ivan Kováč, Michal Kováč, Marek Ludha, Ondrej Mikuláš, Martin Námešný, Michal Nánási, Milan Plžík, Rudolf Starovský, Lukáš Špalek, Rastislav Vaško a tiež ďalším, ktorí v nich našli chyby: Martin Ďuriš, Ján Hozza, Boris Kruľ, Eva Kubaščíková, Eva Lichnerová, Pavol Panák, Peter Perešíni, Martin Rejda, Juraj Stacho, Gabriel Ščerbák, György Tomcsányi.

Obsah

1 Obsah predmetu a motivácia

Základný problém (vyhľadávanie vzorky v texte, string matching): dané sú dva reťazce: vzorka (pattern) P a text T. Nájdite všetky výskyty P v T.

Príklad: $P =$"ma " $T =$"Ema ma mamu". Vzorka P má dva výskyty na pozíciách 2 a 5 reťazca T.

Príklady využitia: Vyhľadávacia funkcia v textových editoroch, internetových prehliadačoch a podobne. UNIXové nástroje ako grep. Práca s textovými atribútmi v relačných databázach, napríklad nasledujúci dotaz nájde apple, pineapple, apples atď.:

```
select name from fruits where name like "%apple%"
```

Ukážeme si niekoľko efektívnych algoritmov pre tento problém (Knuth-Morris-Pratt, Boyer-Moore a iné). Budeme tiež študovať aj ďalšie varianty tohto problému popísané nižšie.

Problém: V texte T chceme vyhľadať všetky výskyty niekoľkých vzoriek $P_1, P_2, \ldots, P_k$.

Príklad využitia: hľadanie vírusov, detekcia spamu, monitorovanie siete pred útočníkmi

Ukážeme si rozšírenia algoritmov zo základného problému, ktoré fungujú v tomto prípade (Ahov-Corasickovej algoritmus).

Problém: V texte T chceme postupne vyhľadať výskyty rôznych vzoriek $P1, P2, \ldots$ Chceme si text T predspracovať (zostaviť index), aby sme odpoveď pre každú vzorku vedeli nájsť rýchlejšie. Tento problém sa líši od predchádzajúceho v tom, že vzorky nemáme dané naraz, ale prichádzajú po jednej a pre každú máme nájsť odpoveď, kým príde ďalšia.

Príklad: internetové vyhľadávanie, knižničný katalóg

Ukážeme si efektívne dátové štruktúry na tento účel (sufixové stromy a polia). Tieto štruktúry tiež umožňujú efektívne riešiť problémy typu: nájdite najdlhšie podslovo, ktoré sa nachádza v daných dvoch reťazcoch.

Problém: V texte T nájdite všetky podslová, ktoré sú podobné na vzor P (majú od P napr. Hammingovu alebo editačnú vzdialenosť najviac k).

Príklad: vyhľadávanie v texte s preklepmi, detekcia plagiátorstva, detekcia príbuzných génov, ktoré vznikli zo spoločného predka mutáciami

Budeme sa venovať algoritmom na výpočet editačnej vzdialenosti medzi reťazcami a na nájdenie približných výskytov vzorky.

V poslednej časti semestra sa budeme venovať niekoľkým problémom, ktoré porovnávajú viacero textov a ktoré pochádzajú z bioinformatiky.

1.1 Označenie

Budeme používať štandardné označenie na prácu s reťazcami, ktoré poznáte napríklad z formálnych jazykov.

Abeceda Σ je konečná množina znakov (symbolov), jej veľkosť budeme označovať σ. Slovo (reťazec) nad abecedou Σ je postupnosť $s_0 s_1 \ldots s_{n-1}$. Dĺžku slova S označujeme $|S|$. Prázdne slovo (slovo dĺžky 0) označujeme ε. Pre slovo $S = s_0 s_1 \ldots s_{n-1}$ budeme $S[i..j]$ označovať podslovo $s_i s_{i+1} \ldots s_j$, kde $0 \leq i \leq j < n$. Ak $i > j$, $S[i..j]$ definujeme ako prázdny reťazec. $S[i]$ označuje i-ty znak slova S. Podslovo $S[i..j]$ je prefix slova S ak $i = 0$ a sufix, ak $j = |S| - 1$. Vlastné podslovo slova S je také podslovo, ktoré sa nerovná S (podobne vlastný sufix, prefix). Zreťazenie slov S a S' označujeme SS'. **Upozornenie:** niektoré časti textu číslujú pozície v texte od 1.

2 Vyhľadávanie kľúčových slov

Dokument – Sekvencia slov
Dotaz – Pre dané slovo w nájsť všetky dokumenty, ktoré ho obsahujú
Cieľ – vytvoriť index pre statickú množinu dokumentov, aby sme vedeli efektívne odpovedať na dotazy

Príklad
Dokument 0 : Ema ma mamu.
Dokument 1 : Mama ma Emu.
Dokument 2 : Mama sa ma. Ema sa ma.

Dotaz: Mama
Odpoveď: Document 1, Document 2

V praxi ako dokument uvažujeme web stránku, email, knihu, kapitoly, atď. Dokument môžeme predspracovať – lower/upper case, úprava na základný tvar, ako su oddelené slová, synonymá, atď. Ak máme veľa dokumentov, vzniká otázka, ako ich ohodnotiť.

2.1 Invertovaný index (inverted index)

Cieľom je vytvoriť utriedený zoznam slov a pri každom si pamätať, v ktorých dokumentoch sa nachádza (nezaujíma nás koľko krát).

Príklad
Ema : 0,2
Emu : 1
ma : 0,1,2
Mama : 1,2
mamu : 0
sa : 2

Statická implementácia pomocou polí

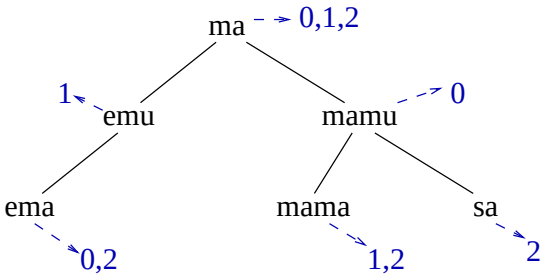
- Sadu dokumentov / url / súborov si namapujeme na čísla $(0,1,\ldots)$.
- Všetky slová si namapujeme na zoznamy ich výskytov v dokumentoch.
- Všetky výskyty slov si budeme pamätať v jednom veľkom poli, v druhom si budeme pamätať slová a index do poľa výskytov.

Príklad
Pole 1 = (0,2,1,0,1,2,1,2,0,2)
Pole 2 = (Ema - 0, Emu - 2, ma - 3, Mama - 6, mamu - 8, sa - 9, $ - 10)

Vyhľadávanie Hľadáme slovo dĺžky m, dohromady majme n slov a výsledok nám vráti k výsledkov. Čas potrebný na nájdenie slova bude:

- vyhľadáme slovo v zozname pomocou binárneho vyhľadávania $O(\log n)$.
- každé porovnanie ale trvá $O(m)$
- prejdeme k výsledkov

Dohromady to je $O(m \log n + k)$



Obr. 1: Príklad binárneho stromu

Tabuľka 1: Zložitosť vyhľadávania a predspracovane pre rôzne implementácie invertovaného indexu.

	Dotaz	Predspracovanie
Utriedené pole	$O(m \log n + p)$	$O(mN \log N)$
Vyhľ. strom	$O(m \log n + p)$	$O(mN \log n)$
Hašovanie - najh. prípad	$O(mn + p)$	$O(mNn)$
Hašovanie - priem. prípad	$O(m + p)$	$O(mN)$
Lex. strom	$O(m \log \sigma + p)$	$O(mN \log \sigma)$

Vytvorenie štruktúry Danú štruktúru vytvoríme nasledovne:

- v každom dokumente utriedime všetky slová
- v každom dokumente odstránime duplicitné slová
- všetky slová dáme dohromady pričom si zapamätáme, v ktorom dokumente boli
- tento velký zoznam ešte raz utriedime a jedným prechodom vytvoríme obe polia

Nech N je veľkosť vytvoreného poľa 1. Potom celková zložitosť bude $O(mN \log N)$

Pri tejto implementácii však vznikne problém, ak pridáme nový dokument. Musíme obidve polia kompletne poposúvať, čo je náročné.

Miesto poľa sa dá použiť na pamätanie si slov binárny vyhľadávací strom, ktorý v listoch bude mať spájané zoznamy výskytov. Ak zabezpečíme vyváženosť stromu, čas sa nezmení a problém s vkladaním sa odstráni. Prehľad zložitosti týchto dátových štruktúr sa nachádza v tabuľke 1.

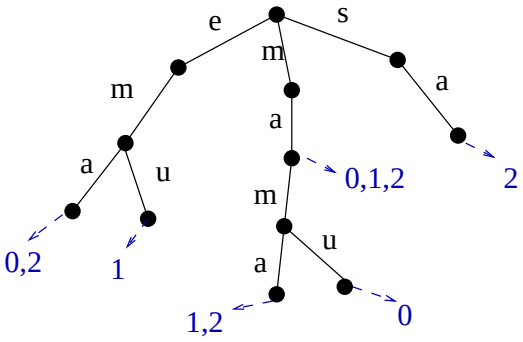
Keď je databáza textov veľká a musí byť uložená na disku, používajú sa B-trees.

2.2 Lexikografické stromy (trie)

Ďalšou možnosťou ako implementovať zoznam slov pre invertovaný index je použiť lexikografický strom, čo je stromová štruktúra špecificky určená na ukladanie množín reťazcov. Každá hrana lexikografického stromu zodpovedá nejakému znaku a cesta z koreňa do nejakého vrchola zodpovedá slovu na ceste.

Vyhľadávanie

Slovo w dĺžky m



Obr. 2: Trie (lexikografický strom)

```
1 node =root;
2 for (i=0; i<m; i++) {
3   node = node-> child[w[i]]
4   if (!node) return empty list
5 }
6 return list of node
```

čas – O(m) – nezáleží na tom, aký veľký je strom, čas vždy závisí iba od dĺžky slova

Pridanie nového slova

Pridávame slovo w, ktoré sa nachádza v dokumente D

```
1 node = root;
2 for(i=0; i<m; i++) {
3    if(! node->child[w[i]]) {
4      node->child[w[i]] = new node;
5    }
6    node = node->child[w[i]];
7 }
8 node->list.add(D)
```

čas - O(m)

Zložitosť pri veľkej abecede. Uvažujme niekoľko slov s dĺžkami $n_1 \dots n_z$, celková dĺžka N, veľkosť abecedy σ, dĺžka dotazu m. Lexikografické stromy sú dobré na reprezentáciu pri konštantnej abecede, pri veľkých abecedách musíme vyriešiť problém, ako uložiť smerníky na deti v každom vrchole tak, aby sme rýchlo vedeli pokračovať po hrane označenej určitým písmenom abecedy. Môžeme použiť napríklad takéto alternatívy:

- Pole detí indexované znakmi abecedy. Toto sa jednoducho implementuje a nájsť príslušné dieťa vieme v $O(1)$ čase. Môžeme však strácať pamäť, ak máme veľa prázdnych položiek. Celkové pamäť je $O(N\sigma)$, čas na vloženie slov $O(N\sigma)$, hľadanie slova $O(m)$.

- Pole dynamickej veľkosti, do ktorého uložíme iba skutočne existujúce deti a kvôli rýchlemu hľadaniu ich utriedime podľa abecedy. Pamäť je $O(N)$, vyhľadávanie beží v čase $O(m \log \sigma)$. Pridávanie jedného slova trvá $O(n_i \log \sigma + \sigma)$, lebo iba v poslednom existujúcom vrchole potrebujeme vkladať nový prvok do zoznamu v čase $O(\sigma)$. V ostatných krokoch buď nájdeme znak v zozname binárnym vyhľadávaním v čase $O(\log \sigma)$, alebo vytvoríme nový vrchol s jediným dieťaťom v čase $O(1)$. Všetky slová teda pridáme v čase $O(N \log \sigma + z\sigma)$.

- Vyvážené vyhľadávacie stromy, v ktorých klúčom sú znaky abecedy. Pamäť je $O(N)$, vyhľadávanie beží v čase $O(m \log \sigma)$ a vkladanie všetkých slov v čase $O(N \log \sigma)$.

2.3 Viacero kľúčových slov

Given several keywords, find documents that contain all of them (intersection, AND). Let us consider 2 keywords, one with m documents, other with n, m<n. Assume that for each keyword we have a sorted array of occurrences (sorted e.g. by document ID, pagerank,...)

Solution 1: Similar to merge in mergesort O(m+n):

```
1  i=0; j=0;
2  while(i<m && j<n) {
3    if(a[i]==b[j]) { print a[i]; i++; j++; }
4    else if(a[i]<b[j]) { i++; }
5    else { j++; }
6  }
```

Sometimes arrays are long, result short - can we improve? Rewrite a little bit:

```
1  j = 0;
2  for(i=0; i<m; i++) {
3       find smallest k>=j s.t. b[k]>=a[i]; (*)
4       if(k==n) { break; }
5       j=k;
6       if(a[i]==b[j]) {
7         print a[i];
8         j++;
9       }
10 }
```

How to do (*):

- linear search `k=j; while(k<n && b[k]<a[i]) { k++; }` the same algorithm as before

- binary search in b[j..n] $O(m \log n)$, faster for very small m

- doubling search/galloping search Bentley, Yao 1976 (general idea), Demaine, Lopez-Ortiz, Munro 2000 (application to intersection, more complex alg.), see also Baeza-Yates, Salinger (nice overview):

```
step = 1; k = j; k2=k;
while(k<n && b[k]<a[i]) {
    k2=k; k=j+step; step*=2;
}
binary search in b[k2..k];
```

If k=j+x, doubling search needs O(log x) steps. Overall running time of the algorithm: doubling search at most times, each time eliminates some number of elements, log(x1)+log(x2)+...log(xm) such that x1+x2+...+xm<=n and the sum is maximized. Happens for equally sized xi's, each n/m elements, $O(m\log(n/m))$ time. For constant m logarithmic, for large m linear. Also better than simple merge if different clusters of similar values in each cluster. If more than 2 keywords, iterate (possible starting with smallest lists), or extend this algorithm.

2.4 To do

Viac textu v casti o invertovanom indexe, preložiť a sTeXovať časť o viacerých kľúčových slovách.

3 Triviálny algoritmus pre vyhľadávanie vzorky v texte

Máme daný reťazec P dĺžky m a reťazec T dĺžky n. Úlohou je nájsť všetky pozície $\{i_1, i_2, \dots\}$ také, že $T[i_j..i_j+m-1] = P$.

Triviálny algoritmus skúsi každú potenciálnu pozíciu výskytu i a skontroluje, či $T[i] = P[0]$, $T[i+1] = P[1]$, ..., $T[i+m] = P[m]$. Ak áno, i je ďalšia pozícia výskytu. Pozície stačí skúšať po $n-m$, inak by sa nám P do T nezmestilo.

```
for(i=0; i<=n-m; i++) {
   j=0;
   while(j<m && P[j]==T[i+j]) { // (*)
      j++;
   }
   if(j==m) {
      print(i);
   }
}
```

Časová zložitosť v najhoršom prípade. Celkovú zložitosť zjavne môžeme odhadnúť zhora na $O(nm)$. Je to však tesný odhad? Celkový čas je úmerný počtu porovnaní medzi znakmi v riadku označenom hviezdičkou. Napríklad na vstupe P ="ma ", T ="Ema ma mamu" potrebujeme 15 porovnaní. V najhoršom prípade tento algoritmus potrebuje $m(n-m+1)$ porovnaní. Uvažujme napríklad reťazce $P = a^m$, $T = a^n$. Pre každé i porovnáme všetky znaky P.

V klasickej zložitosti uvažujeme čas ako funkciu celkovej dĺžky vstupu $N = m+n$. Aká je zložitosť tohto algoritmu v najhoršom prípade? Z hornej hranice $O(mn)$ dostávame triviálne $O(N^2)$. Ak nastavíme $m = N/4$ a $n = 3N/4$, dostávame, že hore-uvedený vstup vyžaduje

$N/4(N/2+1)$ porovnaní, takže zložitosť v najhoršom prípade je $\Theta(N^2)$. Pri niektorých aplikáciách sa spracovávajú veľmi veľké texty a prípadne aj veľké vzorky, takže takáto zložitosť nepostačuje.

Všimnite si však, že okrem pamäte potrebnej na uloženie P a T algoritmus potrebuje iba konštantnú ďalšiu pamäť. Tiež čas a pamäť nezávisia od veľkosti abecedy (za predpokladu, že jednotlivé znaky sa zmestia do jedného registra).

Na mnohých vstupoch sa však tento algoritmus nespráva až tak zle. Napríklad, ak sa prvé písmeno vzorky $P[0]$ nevyskytuje v texte T, urobíme iba $n-m+1$ porovnaní. Prípady, kde je počet porovnaní veľký, majú veľa opakujúcich sa znakov alebo ich skupín. Ak napríklad platí, že $P[0]$ sa nevyskytuje vo zvyšku vzorky, počet porovnaní bude lineárny, čo si dokážeme technikou, ktorá sa volá amortizovaná analýza zložitosti. Amortizovaná analýza sa používa vtedy, ak niektoré iterácie algoritmu (alebo niektoré operácie na dátovej štruktúre), trvajú dlho a iné krátko a chceme dokázať, že v najhoršom prípade bude súčet týchto krátkych a dlhých operácií pomerne malý. V našom prípade chceme nájsť horný odhad počtu porovnaní znakov, pričom predpokladáme, že $P[0]$ sa nevyskytuje vo zvyšku vzorky, ale inak môžu byť vzorka aj text ľubovoľné. Ak sa pri porovnávaní vzorky na pozícii i spraví $k>1$ porovnaní, znamená to, že $P[1..k-2] = T[i+1..i+k-2]$ a teda v $T[i+1..i+k-2]$ sa nevyskytuje znak $P[0]$. V každej z ďalších $k-2$ iterácií teda spravíme najviac 1 porovnanie. Máme dva typy iterácií: lacné, v ktorých spravíme iba 1 porovnanie a drahé, v ktorých spravíme viac porovnaní. Predstavme si, že každé porovnanie má cenu jedna. Za každý typ iterácie "zaplatíme" do pomyselnej kasičky sumu 2. Pri drahej iterácii táto suma zaplatí prvé porovnanie znaku $P[0]$ s $T[i]$ a posledné provnanie, v ktorom sa našla nezhoda. Zhody na pozíciách $P[1..k-2]$ zostanú zatiaľ nezaplatené. Každá z lacných iterácií zaplatí svoje jedno porovnanie a splatí jedno porovnanie z dlhu, ktorý narobila predchádzajúca drahá iterácia. Nakoľko tento dlh bol $k-2$ a počet lacných iterácií za drahou je tiež aspoň $k-2$, celý dlh splatia. Za celý prechod algoritmom sa teda splatia všetky porovnania, možno s výnimkou poslednej drahej iterácie, za ktorou sa už do reťazca nemusia "zmestiť" ďalšie lacné, takže potrebujeme ešte doplatiť najviac $m-2$. Celkovo teda zaplatíme $2(n-m+1)+m-2 \leq 2n$, a toto číslo je horný odhad na počet porovnaní v algoritme.

Časová zložitosť v priemernom prípade. Aký bude priemerný prípad? Uvažujme binárnu abecedu. Chceme spočítať priemerný čas (resp. počet porovnaní) cez všetky reťazce P a T dĺžky n a m. Nech $c_i(P,T)$ je počet porovnaní znakov v interácii i. Potom priemerný počet je

$$\begin{aligned} & \frac{1}{2^{n+m}} \sum_{P\in\{0,1\}^m, T\in\{0,1\}^n} \sum_{i=0}^{n-m} c_i(P,T) \\ = \; & \frac{1}{2^{n+m}} \sum_{i=0}^{n-m} 2^{n-m} \sum_{P\in\{0,1\}^m, T'\in\{0,1\}^m} c_0(P,T') \\ = \; & \frac{n-m+1}{2^{2m}} \sum_{P\in\{0,1\}^m, T'\in\{0,1\}^m} c_0(P,T') \end{aligned}$$

Počet dvojíc P a T dĺžky m, kde urobíme j porovnaní pre $j<m$ je $2^j 4^{m-j} = 2^{2m-j}$ (na prvých $j-1$ pozíciách majú ten istý, ale ľubovoľný znak a na pozícii j majú opačný znak, zvyšné pozície majú ľubovoľné). Počet reťazcov, kde spravíme m porovnaní, je $2^{m-1}4$. Teda dostávame

$$\sum_{P\in\{0,1\}^m, T\in\{0,1\}^m} c_0(P,T) = \sum_{j=1}^{m-1} j2^{2m-j} + m2^{m+1}.$$

Celkovo dostávame ako priemerný počet porovnaní sumu $(n-m+1)(\sum_{j=1}^{m-1} j/2^j + m/2^{m-1})$. Skúsme odhadnúť jej hlavnú časť (použijeme súčet geometrickej postupnosti $\sum_{i=0}^{n} q^i = (1-q^{n+1})/(1-q)$ a konkrétne pre $q = 1/2$ máme $2 - 1/2^n$):

$$\begin{aligned}
\sum_{j=1}^{m} j/2^j &= \sum_{j=1}^{m}\sum_{i=1}^{j} 1/2^j = \sum_{i=1}^{m}\sum_{j=i}^{m} 1/2^j \\
&= \sum_{i=1}^{m}(\sum_{j=0}^{m} 1/2^j - \sum_{j=0}^{i-1} 1/2^j) \\
&= \sum_{i=1}^{m}(2 - 1/2^m - (2 - 1/2^{i-1})) \\
&= \sum_{i=1}^{m}(1/2^{i-1} - 1/2^m) \\
&= -m/2^m + \sum_{i=0}^{m-1}(1/2^i) \\
&= -m/2^m + 2 - 1/2^{m-1}
\end{aligned}$$

Celkovo dostávame

$$\begin{aligned}
& (n-m+1)(\sum_{j=1}^{m-1} j/2^j + m/2^{m-1}) \\
= \;& (n-m+1)(\sum_{j=1}^{m} j/2^j + m/2^m) \\
= \;& (n-m+1)(2 - 1/2^{m-1}) \\
\leq \;& 2(n-m+1)
\end{aligned}$$

Takže časová zložitosť v priemernom prípade je lineárna.

Odvodenie priemernej zložitosti pomocou pravdepodobnosti. O niečo jednoduchšie je odvodiť tento vzťah pomocou pravdepodobnosti. Predstavme si náhodný proces, v ktorom generujeme jednotlivé znaky vo vzorke aj v texte nezávisle z rovnomerného rozdelenia na $\Sigma = \{0,1\}$ (t.j. $m+n$ krát si hodíme mincou a zapíšeme si výsledok ako binárne reťazce P a T). Nech X je náhodná premenná označujúca počet porovnaní, ak na takto vygenerovaných náhodných reťazcoch spustíme triviálny algoritmus. Potom priemerný počet porovnaní, ktorý sme počítali v predchádzajúcich odstavcoch, je rovný strednej hodnote premennej X, t.j. $E(X)$. Označme si ešte dve sady premenných: nech X_i je počet porovnaní v iterácii i (t.j. keď $P[0]$ zarovnáme s $T[i]$) a nech Y_i je dĺžka najdlhšieho spoločného prefixu reťazcov P a $T[i..n-1]$.

Potom $X = \sum_{i=0}^{n-m} X_i$. Stredná hodnota je lineárna a teda $E(X) = \sum_{i=0}^{n-m} E(X_i)$. Z definície strednej hodnoty máme $E(X_i) = \sum_{j=1}^{m} jP(X_i = j)$. Pre $k < m$ máme že $X_i = k$ práve vtedy, keď $Y_i = k-1$, lebo triviálny algoritmus skontroluje celý spoločný prefix a ešte aj prvý rozdielny znak. Podobne, $X_i = m$ práve vtedy keď $Y_i \geq m-1$. Môžeme teda písať:

$$\begin{aligned} E(X_i) &= \sum_{k=1}^{m} kP(X_i = k) \\ &= mP(Y_i \geq m-1) + \sum_{k=1}^{m-1} kP(Y_i = k-1) \\ &= mP(Y_i \geq m-1) + \sum_{k=1}^{m-1} k(P(Y_i \geq k-1) - P(Y_i \geq k)) \end{aligned}$$

Keď si súčet rozpíšeme, zistíme, že sčítance pre susedné hodnoty k zdieľajú jednu z pravdepodobností a ich odčítaním sa teda výraz značne zjednoduší (tzv. teleskopická suma)

$$\begin{aligned} E(X_i) &= 1(P(Y_i \geq 0) - P(Y_i \geq 1)) + 2(P(Y_i \geq 1) - P(Y_i \geq 2)) + 3(P(Y_i \geq 2) - P(Y_i \geq 3)) \\ &\quad + \cdots + (m-1)(P(Y_i \geq m-2) - P(Y_i \geq m-1)) + mP(Y_i \geq m-1) \\ &= \sum_{k=0}^{m-1} P(Y_i \geq k). \end{aligned}$$

Aby dĺžka spoločného prefixu Y_i bola aspoň k, musí sa prvých k znakov medzi P a $T[i..n-1]$ zhodovať, pričom každá dvojica sa zhoduje s pravdedobnosťou $1/2$ nezávisle od ostatných dvojíc. Takže máme $P(Y_i \geq k) = 2^{-k}$. Celkovo teda dostávame

$$E(X_i) = \sum_{k=0}^{m-1} 2^{-k} = 2 - 2^{-m+1}$$

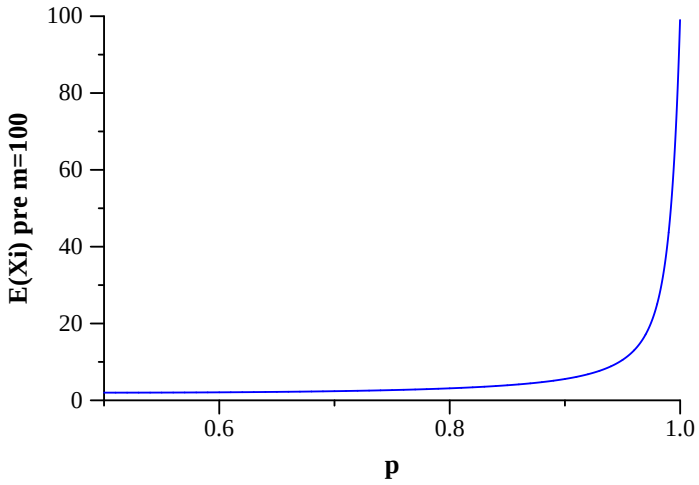
a po vynásobení $(n-m+1)$ pozíciami pre i dostávame pre $E(X)$ rovnaký vzťah ako predtým.

Iné rozdelenie pravdepodobnosti pre reťazce. Odvodenie priemernej zložitosti pomocou pravdepodobnosti má aj tú výhodu, že sa dá ľahko rozšíriť napríklad na prípad, keď 0 a 1 sa v texte vyskytujú nerovnomerne, t.j. $P(T[i] = 1) = P(P[j] = 1) = p$, kde $p > 0.5$. Intuitívne čím vyššie p, tým väčšia šanca že dva náhodné znaky budú oba 1 a teda sa budú zhodovať. Pravdepodobnosť, že $P[i] = T[j]$ je $p^2 + (1-p)^2$, čo označíme ako q. Aby mali dva reťazce spoločný prefix dĺžky aspoň k, musí nastať k takýchto nezávislých zhôd za sebou a teda $P(Y_i \geq k) = q^k$. Podobným postupom ako v prípade $p = 1/2$ dostávame

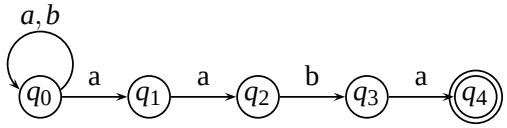
$$E(X_i) = \sum_{k=0}^{m-1} P(Y_i \geq k) = \frac{1-q^m}{1-q} = \frac{1-(p^2+(1-p)^2)^m}{1-p^2-(1-p)^2}.$$

Na obrázku 3 vidíme graf $E(X_i)$ pre $m = 100$ ako funkciu p. Pre $p = 1/2$ máme v priemere o niečo menej ako dve porovnania v každej iterácii. So zvyšujúcim sa p rastie šanca náhodného stretnutia dvoch jednotiek a teda aj priemerný počet porovnaní. V limite pre $p = 1$ dostaneme najhorší prípad, t.j. m porovnaní v každej iterácii.

Naopak, ak sú znaky rovnomerne rozdelené z nejakej väčšej abecedy, priemerný počet porovnaní klesá, lebo šanca zhody medzi dvoma znakmi je menšia.



Obr. 3: Graf $E(X_i)$ pre $m = 100$ ako funkcia p.



Obr. 4: Nedeterministický konečný automat akceptujúci jazyk L_P pre $P = aaba$.

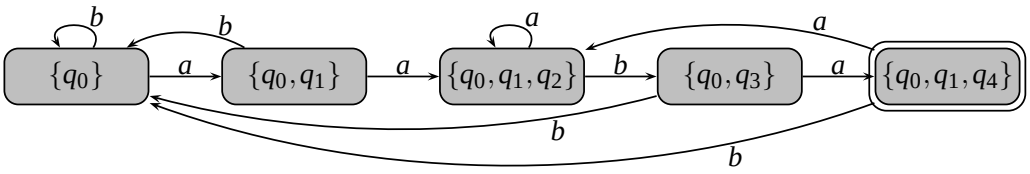
4 Konečné automaty a Knuth-Morris-Prattov algoritmus

Hľadáme výskyty vzorky P dĺžky m v texte T dĺžky n. Triviálny algoritmus má časovú zložitosť $O(mn)$ a v priemernom prípade $\theta(m+n)$. Teraz si ukážeme jeden z klasických algoritmov, ktoré pracujú v lineárnom čase $O(n+m)$.

Všimnime si najprv, prečo je náš triviálny algoritmus "zlý". Vezmime si vzorku $P = abcabd$ a hľadajme ju v texte $T = abcabcabd$. Triviálny algoritmus urobí v prvom kroku šesť porovnaní (zistí, že $P[0..4] = T[0..4]$ a $P[5] \neq T[5]$), potom dva krát po jednom porovnaní (zistí, že $P[0] \neq T[1]$ a $P[0] \neq T[2]$) a nakoniec znova šesť porovnaní (nájde jediný výskyt vzorky v texte). Spolu teda spraví 14 porovnaní. Ak by sme si však vopred predspracovali vzorku, vedeli by sme ju v texte hľadať rýchlejšie. Napríklad pomocou prvých 6 porovnaní zistíme, že $P[0..4] = T[0..4]$. Preto vieme, že sa nám neoplatí posunúť vzorku o jednu pozíciu, lebo hneď prvé porovnanie by porovnávalo $T[1] = P[1] = b$ s $P[0] = a$. Preto môžeme skočiť až na najbližší výskyt a v T, teda na pozíciu 3. Aj tu ešte môžeme ušetriť porovnania a to preto, že vieme, že že $P[0..4] = T[0..4]$ a súčasne $P[0..1] = P[3..4]$. Preto pri porovnávaní P a $T[3..8]$ stačí spraviť posledné 4 porovnania. Spolu tak ušetríme 4 porovnania. Takéto ušetrenie dosiahneme pomocou Knuth-Morris-Prattovho algoritmu, ktorý zavedieme pomocou konečných automatov.

4.1 Nedeterministický konečný automat

Majme slovo $P \in \Sigma^*$. Nech $L_P = \{xP | x \in \Sigma^*\}$. Tento jazyk je regulárny a môžeme preto zostrojiť nedeterministický konečný automat (NKA), ktorý ho rozpoznáva. Ako príklad budeme uvažovať vzorku $P = aaba$.



Obr. 5: Deterministický konečný automat akceptujúci jazyk L_P, ktorý bol vyrobený z nedeterministického konečného automatu z obrázku 4.

Pre jazyk L_P si zostrojíme NKA, tak ako na obrázku 4. Po načítaní reťazca T môže tento automat skončiť v stave q_i ak prefix P dĺžky i je sufixom T, t.j.

$$(q_0, T) \vdash^* (q_i, \varepsilon) \Leftrightarrow P[0..i-1] \text{ je sufix } T.$$

Vzorka P má teda výskyt v T na pozícii i práve vtedy, keď akceptujúci stav je dosiahnuteľný po načítaní prvých $i+|P|-1$ znakov T. Môžeme teda simulovať činnosť automatu na reťazci T (pre každú pozíciu si vypočítame množinu dosiahnuteľných stavov) a testujeme túto podmienku. Na jednej pozícii T však môže byť až m dosiahnuteľných stavov, takže aj takýto algoritmus by fungoval v čase $O(mn)$.

```
1  S = {0};
2  for(i=0; i<n; i++){
3    S1 = empty_set;
4    foreach state j in S {
5       add delta(j, T[i]) to S1;
6    }
7    if (m in S1){
8      print i-m+1;
9    }
10   S = S1;
11 }
```

Cvičenia: ako implementovať množinu stavov S resp. S_1 tak, aby algoritmus naozaj pracoval v čase $O(mn)$? Aký je vzťah medzi počtom porovnaní v triviálnom algoritme a súčtom veľkostí množín stavov S v tejto simulácii?

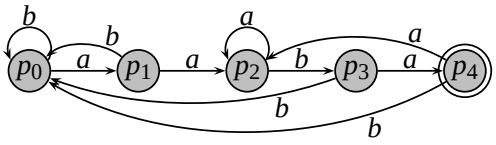
4.2 Deterministický konečný automat

Zostrojme teraz štandardnou podmnožinovou konštrukciu k tomuto automatu deterministický konečný automat. Výsledný automat je na obrázku 5. Pre tento automat platí nasledovné tvrdenie:

$$(\{q_0\}, T) \vdash^* (\{q_i \,|\, P[0..i-1] \text{ je } T\}, \varepsilon)$$

Stavy tohto automatu sú teda podmnožiny stavov NKA, pričom stav zodpovedá všetkým prefixom vzorky P, ktoré sú sufixami prečítaného slova T.

Všimnime si, že ak poznáme najdlhší prefix P, ktorý je sufixom T, vieme odvodiť aj všetky kratšie prefixy P, ktoré majú túto vlastnosť. Náš deterministický automat má teda zaručene presne $m+1$ stavov, jeden pre každú dĺžku prefixu. Tieto stavy pre jednoduchosť môžeme aj



Obr. 6: Deterministický konečný automat akceptujúci jazyk L_P z obrázku 5 po prečíslovaní stavov.

premenovať: namiesto množiny pôvodných stavov nazveme každý stav podľa dĺžky najdlhšieho prefixu. Stav $\{q_{a_i}, \ldots, q_{a_k}\}$ teda premenujeme na $p_{\max\{a_1 \ldots, a_k\}}$. V takto premenovanom DKA teda platí

$$(p_0, T) \vdash^* (p_i, \varepsilon) \iff P[0..i-1] \text{ je najdlhší prefix } P, \text{ ktorý je sufixom } T$$

Príklad takéhoto automatu je na obrázku 7. DKA vieme ľahko simulovať na vstupe T a ak sa dostaneme do akceptačného stavu, našli sme koniec výskytu vzorky P.

```
1 state = 0;
2 for(i=0; i<n; i++){
3   state= delta(state,T[i]);
4   if (state==m){
5     print i-m+1;
6   }
7 }
```

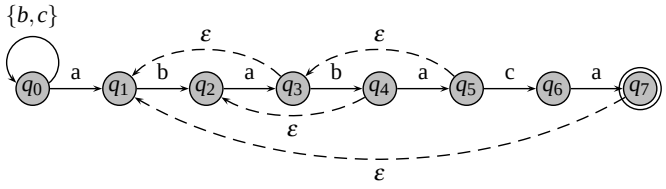
Ak už máme zostrojený automat, časová zložitosť hľadania výskytov vzoriek v texte je $O(n)$. Ako rýchlo vieme zostrojiť samotný automat? Pri množinovej konštrukcii by sme dostali čas $O(m^2\sigma)$ a ak by sme sa veľmi snažili, dal by sa dosiahnuť čas $O(m\sigma)$ (detaily neskôr). Zrýchliť sa to už nedá, lebo potrebujeme celú prechodovú funkciu automatu a tá má veľkosť $m\sigma$. Tiež si ju musíme celú pamätať a teda pamäťová zložitosť je tiež $O(m\sigma)$.

4.3 Morrisov-Prattov algoritmus 1970

Morrisov-Prattov algoritmus bol prvý známy algoritmus pracujúci v čase $O(m+n)$. Na to, aby sme mali menšiu pamäťovú zložitosť, je potrebné zmenšiť náš automat. Pridáme do neho epsilonové prechody, čím už síce nebude deterministický, ale po epsilonových prechodoch sa budeme hýbať len vtedy, ak sa nebudeme vedieť hýbať po neepsilonových prechodoch a preto bude vždy jasné, do ktorého stavu sa posunúť. Výsledný automat bude mať nasledovnú δ funkciu:

$$\delta(i,a) = \begin{cases} i+1 & \text{ak } a = P[i] \\ sp[i] & \text{inak, prechod na } \varepsilon \end{cases}$$

kde $sp[i]$ je dĺžka najdlhšieho vlastného sufixu $P[0..i-1]$, ktorý je súčasne prefixom P (pozri príklad na obrázku 7). Z každého stavu i okrem stavov 0 a m teda vychádzajú iba dva prechody: jeden do stavu $i+1$ a jeden do niektorého z predchádzajúcich stavov. Ak zo stavu i ideme po epsilových hranách, prechádzame postupne po všetkých vlastných sufixoch $P[1..i]$, ktoré sú súčasne prefixami P. Keď sa takto dostaneme do stavu j, z ktorého vychádza prechod označený aktuálnym znakom na vstupe, použijeme tento prechod.



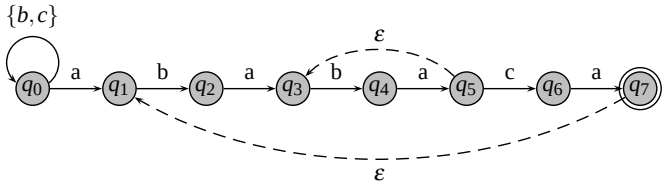
Obr. 7: Konečný automat pre MP algoritmus, vzorka $P = ababaca$. Epsilonové hrany idúce do q_0 nie sú v obrázku zakreslené.

Predpokladajme na chvíľku, že máme vypočítané $sp[i]$ pre každé i. Na nájdenie vzorky v texte môžeme simulovať činnosť upraveného automatu.

```
1  state=0;
2  for(i=0; i<n; i++){
3    while(state>0 && T[i]!=P[state]){ //epsilonove prechody
4      state=sp[state];
5    }
6    if(T[i]==P[state]){   //prechod do dalsieho stavu
7      state++;
8    }
9    if(state==m){   //sme v koncovom stave
10     print i-m+1;
11     state=sp[state];   //z posledneho stavu sa uz neda ist dalej
12   }
13 }
```

Tento algoritmus simuluje deterministický konečný automat. Po spracovaní každého znaku sa ocitne v tom istov stave, ako DKA. Intuícia za epsilonovými prechodmi je nasledovná. Predstavme si, že po spracovaní textu T je DKA v stave p_i, existuje teda reťazec α dĺžky i, ktorý je sufixom T a súčasne prefixom P. Ak teraz prečítame automatom ďalší znak a (spracovali sme teda reťazec Ta), potrebujeme nájsť najdlhšie β ktorá je sufixom Ta a prefixom P. Ak $\beta \neq \varepsilon$, musí končiť znakom a, t.j. $\beta = \beta' a$. Reťazec β' je sufixom T a teda aj sufixom α a súčasne prefixom P a teda aj prefixom α. Hľadáme teda najdlhšie β', ktoré je sufixom aj prefixom $\alpha = P[0..i-1]$ a pre ktoré je aj $\beta' a$ prefixom P. Ak ideme zo stavu i po epsilonových prechodoch až do stavu 0, prejdeme cez zoznam všetkých stavov, ktoré zodpovedajú sufixom $P[0..i-1]$, ktoré sú súčasne aj prefixami P. Prvý z nich, z ktorého ide šípka na a, je ten, ktorý hľadáme.

Tento algoritmus vyzerá kvadraticky, ale nie je. Ak nerátame while cyklus, časová zložitosť algoritmu je $O(n)$. Presnejšie, nech c_i je počet vykonaní riadku 4, potom zložitosť i-tej iterácie je $O(c_i + 1)$ a celková zložitosť algoritmu $O(n + \sum_{i=0}^{n-1} c_i)$. Vieme, že $0 \leq c_i < m$, ale ukážeme, že vysoké hodnoty nemôže nadobúdať príliš často. Každé vykonanie riadku 4 totiž zníži hodnotu premennej state aspoň o jedna. Hodnota tejto premennej sa zvýši v každej iterácii for cyklu najviac o 1 (v riadku 7). Zároveň hodnota premennej state nebude nikdy nižšia ako 0 a preto sa riadok 4 nemôže vykonať viackrát, ako riadok 7 a preto sa vykoná najviac n-krát a preto zložitosť celého algoritmu je $O(n)$.



Obr. 8: Konečný automat pre KMP algoritmus. Ak ide ε hrana do q_0, tak hrana v obrázku nie je zakreslená.

Vytvorenie automatu. Jediné, čo nám chýba, je vyrobenie automatu pre danú vzorku P. To sa nápadne podobá na samotné vyhľadávanie:

```
1  sp[0]=sp[1]=0;
2  j=0;
3  for(i=2; i<=m; i++){
4     // invariant: j=sp[i-1];
5     while(j>0 && P[i-1]!=P[j]) {
6        j=sp[j];
7     }
8     if(P[i-1]==P[j]) {
9        j++;
10    }
11    sp[i]=j;
12 }
```

Zdvôvodnenie je podobné ako pri vyhľadávaní: opäť poznáme najdlhší reťazec α dĺžky j, ktorý je vlastným sufixom $P[0..i-2]$ a sufixom P a hľadáme najdlhšiu $\beta = \beta' P[i-1]$, ktorá je sufixom aj prefixom $P[0..i-1]$.

Zhrnutie. Ako vidíme, MP algoritmus beží v čase $O(n+m)$ a s pamäťou $O(m)$. Samotný text nemusíme ukladať, môžeme ho čítať a spracovávať po jednom písmene (napr. pri spracovaní streamovaných médií). Jediná operácia so znakmi je ich porovnanie na rovnosť a algoritmus preto nie je príliš citlivý na veľkosť abecedy. Na druhej strane medzi spracovaním dvoch po sebe idúcich znakov môže prejsť čas až $O(m)$, pričom pri použití DKA je tento čas $O(1)$.

4.4 Knuth-Morris-Prattov algoritmus 1977

KMP algoritmus sa veľmi podobá na MP algoritmus, ale má v automate dlhšie epsilonové prechody, a to tak, že nasledujúce znaky sa nesmú rovnať. Takže $\delta'(i,\varepsilon)$ je dĺžka j najdlhšieho vlastného sufixu $P[0..i-1]$, ktorý je zároveň prefixom P a platí $P[j] \neq P[i]$ (pozri obr. 8). Napríklad pre $P = a^m$ KM algoritmus má $\delta(i,\varepsilon) = i-1$ a KMP algoritmus má $\delta'(i,\varepsilon) = 0$. Takže ak sa na vstupe objaví napr. písmeno b, KMP použije iba jeden epsilonový prechod, kým KM môže použiť až $m-1$ prechodov. O takto upravenom algoritme sa dá dokázať nasledovná veta:

Veta: Počet ε prechodov v jednej iterácii KMP je naviac $\log_\phi(m+1)$, kde $\phi = \frac{1+\sqrt{5}}{2}$ je zlatý rez.

Predtým, ako vetu dokážeme, uveďme niekoľko definícií a liem.

Definícia 1. Celé číslo x nazývame periódou slova S ak $1 \leq x \leq m$ a pre každé $i \in \{0,1,\ldots,|S|-x-1\}$ platí $S[i] = S[i+x]$.

Podľa tejto definície, dĺžka slova je vždy jeho periódou, slovo však môže mať aj kratšie periódy. Napríklad slovo abab má periódy 2 a 4, slovo aaba má periódy 3 a 4 a slovo aaaa má periódy 1,2,3,4.

Lema 1. Ak p a q sú periódy slova S a platí, že $p < q$ a $p+q \leq |S|$, tak aj $q-p$ je perióda S.

Dôkaz. Vezmime nejaké $i \in \{0,1,\ldots,|S|-(q-p)-1\}$, chceme dokázať, že $S[i] = S[i+(q-p)]$. Ak $i+q < |S|$, tak keďže q je perióda, máme, že $S[i] = S[i+q]$. Podobne, keďže p je perióda, platí, že $S[i+q-p] = S[i+q]$. Spojením týchto dvoch rovností dostaneme požadované tvrdenie.

Ak naopak $i+q \geq |S|$, potom $i-p \geq |S|-q-p \geq 0$ a teda keďže p a q sú periódy, máme $S[i] = S[i-p] = S[i-p+q]$. □

Nasledujúca lema sa už týka priamo KMP automatu pre vzorku P.

Lema 2. Nech $q_{j'} = \delta'(q_j, \varepsilon)$ a $q_{j''} = \delta'(q_{j'}, \varepsilon)$. Potom $j \geq j' + j'' + 1$.

Dôkaz. Lemu dokážeme sporom. Predpokladáme, že $j \leq j' + j''$.

Nech $p = j - j'$ a $q = j - j''$. Pre ich súčet z nášho predpokladu platí $p+q = j-j'+j-j'' \leq j-j+j = j$. Taktiež $q > p$ a p aj q sú periódy $P[0..j-1]$. Podľa predchádzajúcej lemy teda aj $q-p$ je perióda.

Z definície δ' tiež vieme, že $P[j''] \neq P[j']$, čo je ale spor, lebo $j' = j'' + (q-p)$ a keďže $q-p$ je perióda, tieto dva znaky by mali byť také isté. □

Dôkaz vety: Nech zo stavu q_j je k prechodov do stavu q_0. Dokážeme indukciou vzhľadom na k, že $j \geq F_{k+2} - 1$, kde F_i je i-te Fibonacciho číslo zadefinované rekurenciou $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$.

1. $k = 0$: $j \geq 0 = F_2 - 1$
   $k = 1$: $j \geq 1 = F_3 - 1$

2. $k \geq 2$

   $j \geq j' + j'' + 1 \geq F_{k+1} - 1 + F_k - 1 + 1 = F_{k+2} - 1$

Z definície Fibonacciho čísel sa dá dokázať, že $F_k \geq \phi^{k-2}$, z čoho vyplýva $j+1 \geq F_{k+2} \geq \phi^k$ a teda $k \leq \log_\phi(j+1) \leq \log_\phi(m+1)$. □

4.5 Cvičenia

Príklad 1. Nájdite najdlhší prefix P, ktorý sa vyskytuje v T. Riešenie: stačí si zapamätať najvyšší navštívený stav v MP/KMP algoritme.

Príklad 2. Nájdite najdlhší sufix P, ktorý sa vyskytuje v T. Riešenie: prevedieme P aj T na zrkadlové obrazy a hľadáme najdlhší prefix P^R v T^R.

Príklad 3. Máme spočítané hodnoty $sp[i]$ z MP algoritmu, chceme $\delta(i,a)$ z DKA. Riešenie: pre každé a v abecede spustíme

```
1  if(P[0]=='a') {
2    delta(0,'a') = 1;
3  }
4  else {
5    delta(0,'a') = 0;
6  }
7  for(i=1; i<=m; i++) {
8    if(i<m && P[i]=='a') {
9      delta(i,'a')=i+1;
10   }
11   else {
12     delta(i,'a')=delta(sp[i],a);
13   }
14 }
```

Príklad 4. Máme spočítané hodnoty $sp[i]$ z MP algoritmu, chceme sp_2 z KMP. Riešenie:

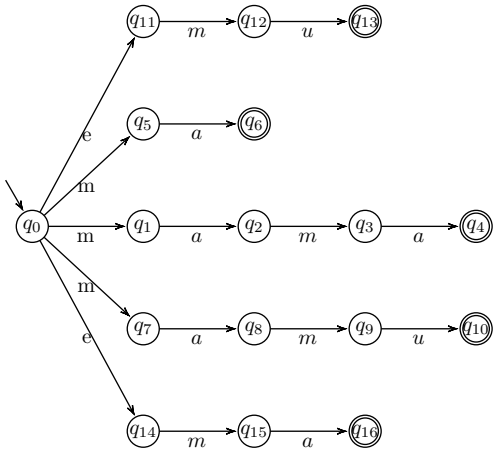
```
1  sp2[0]=0;
2  for(i=1; i<=m; i++) {
3    if(i==m || P[sp[i]]!=P[i]) {
4      sp2[i]=sp[i];
5    }
6    else {
7      sp2[i]=sp2[sp[i]];
8    }
9  }
```

TO DO

- Priebeh triviálneho a zlepšeného algoritmu pre hľadanie vzorky abcabd nejako krajšie znázorniť (obrázkom, napr. ako v prehľade od Charrasa a Lecroqa uvedenom v zozname literatúry)

5 Vyhľadávanie viacerých vzoriek a 2D vzorky

Máme $\mathscr{P} = \{P_1, P_2, \ldots, P_z\}$. Nech $m = \sum_i |P_i|$. Chceme nájsť všetky výskyty reťazcov z $\mathscr{P}$ v T. Triviálne riešenie tejto úlohy by bolo z-krát spustiť KMP, čím dostaneme časovú zložitosť



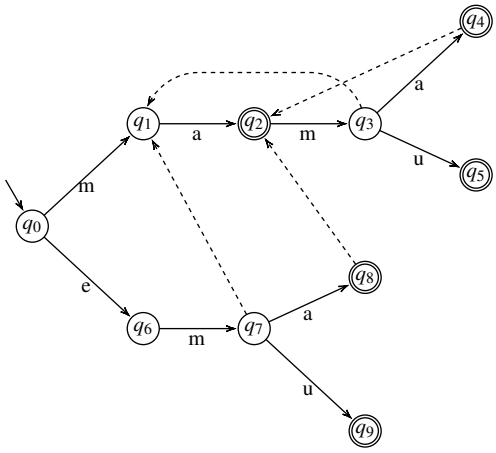
Obr. 9: Nedeterministický konečný automat rozpoznávajúci $L_{\mathscr{P}}$ pre $\mathscr{P} = \{mama, ma, emu, ema, mamu\}$.

$\sum_i O(|P_i| + n) = O(m + zn)$. Ak $m = N/2$ aj $z = N/2$ a $n = N/2$, tak celkový čas bude $O(N^2)$, kde $N = m + n$ je celková veľkosť vstupu.

5.1 Aho-Corasickovej algoritmus (1975)

Je to rozšírenie MP algoritmu na viac vzoriek, pričom dosahuje zrýchlenie oproti jednoduchému algoritmu popísanému vyššie. Podobne ako pri odvodení MP môžeme zostaviť nedeterministický konečný automat rozpoznávajúci jazyk $L_{\mathscr{P}} = \{xy | x \in \Sigma^*, y \in \mathscr{P}\}$ (obr. 9).

Z nedeterministického automatu chceme spraviť automat podobný na automat pre MP algoritmus. Najskôr zostrojíme lexikografický strom (pozri časť 2.2). Do lexikografického stromu pridáme epsilonové hrany tak, že z vrcholu pre slovo u ide hrana do vrcholu pre slovo v, ktoré je najdlhším vlastným sufixom u, ktorý je prefixom niektorej vzorky (príklad na obrázku 10).

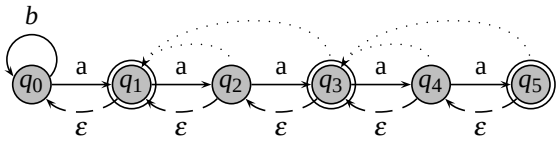


Obr. 10: AC automat pre vzorky $\{mama, ma, emu, ema, mamu\}$, epsilové prechody sú čiarkovane, epsilonové prechody do q_0 nie sú znázornené.

Na danom texte môžeme tento automat simulovať podobne ako MP automat a v každom kroku bude stav predstavovať najdlhší sufix načítaného reťazca, ktorý je súčasne prefixom

niektorej vzorky. Musíme však zmeniť spôsob vypisovania, lebo niektoré vzorky môžu byť podreťazcom iných vzoriek a teda na danej pozícii potrebujeme vypísať výskyty všetkých sufixov, ktoré sú vzorkami v $\mathscr{P}$. Jedna možnosť by bola v každom stave prejsť po epsilonových hranách až do q_0 a vypísať všetky stavy na tejto ceste, ktoré zodpovedajú niektorej vzorke. Aby sme ale nemuseli prechádzať aj cez stavy, ktoré nevypisujeme, pridáme si špeciálne výstupné hrany. Z vrcholu pre slovo u ide výstupná hrana do vrcholu pre najdlhšiu vzorku, ktorá je vlastným sufixom u, alebo má hodnotu null, ak taká vzorka neexistuje. Výstupné hrany teda tvoria spájaný zoznam výskytov, ktoré treba vypísať. Je to podmnožina spájaného zoznamu tvoreného epsilonovými hranami.

Príklad: V automate na obrázku 10 pôjdu výstupné hrany z q_4 a q_8 do q_2 a zo všetkých ostatných vrcholov do null. Na obrázku 11 je automat pre množinu vzoriek $\{a, a^3, a^5\}$ aj s výstupnými hranami.



Obr. 11: Automat pre vzorky $\{a, a^3, a^5\}$, bodkované hrany sú výstupné.

Vrchol v lexikografickom strome môže mať napríklad nasledovnú štruktúru:

- *parent*: smerník na rodič vrchola
- $child(a)\ \forall a \in \Sigma$: pole smerníkov na deti
- *last*: posledný znak slova pre tento vrchol, t.j. $a \in \Sigma$ také, že $parent.child[a] = this$
- *id*: číslo vzorky/null
- *epsilon*: epsilonová hrana
- *output*: výstupná hrana

Predspracovanie sa bude skladať z nasledovných krokov:

1. Vytvorenie lexikografického stromu
2. Nájdenie epsilonových hrán
3. Nájdenie výstupných hrán

Potom môže nasledovať samotný Aho-Corasickovej vyhľadávací algoritmus:

```
1 v=root;
2 for (i=0; i<n; i++) {
3   // hladame aktualny stav po epsilonovych hranach
4   while (v!=root && v.child(T[i])==null) {
5     v=v.epsilon;
6   }
```

```
7     // ak sa da, posunieme sa o jedno dopredu
8     if (v.child(T[i])!=null) {
9       v=v.child(T[i]);
10    }
11    // vypis vyskytov po vystupnych hranach
12    u=v;
13    while (u!=null) {
14      if (u.id!=null) {
15        print u.id, i-|P[u.id]|+1;
16      }
17      u=u.output;
18    }
19  }
```

Zložitosť. Samotný algoritmus vykoná najviac n iterácií vo vonkajšom cykle, zaujíma nás zložitosť vnútornej časti – tu sa najskôr nájde po epsilonových hranách prvý uzol, ktorý má potomka pre aktuálne písmeno, a potom (ak je to možné) sa vykoná jeden krok na toto písmeno. Dôležité je, že spätných epsilonových krokov sa počas celého algoritmu vykoná najviac toľko, koľko sa vykoná krokov do potomkov uzlov – a keďže týchto sa vykoná najviac n, aj spätných krokov bude najviac n. Celkovo dostávame pre tieto časti $O(n)$.

Posledná dôležitá časť je druhý *while*-cyklus, ktorý postupne prechádza po výstupných hranách a vypisuje výskyty vzoriek. Počas celého výpočtu sa po výstupných hranách prejde k-krát, kde k je celkový počet nájdených výskytov. Zložitosť je teda $O(n+k)$.

Výpočet ε-prechodov. Najskôr nastavíme koreňu epsilonový prechod do samého seba a potom pre všetky ostatné vrcholy zavoláme funkciu findepsilon(v), pričom postupujeme v poradí prehľadávania do šírky, takže najskôr spracujeme všetky vrcholy v hĺbke 1, potom všetky vrcholy v hĺbke 2 atď.

```
1   void findepsilon(node v) {
2     // Predpokladame u.epsilon je zname
3     // pre vsetky slova u kratsie ako v
4     a = v.last;
5     u = v.parent.epsilon;
6     while (u.child(a)==null && u!=root) {
7       u=u.epsilon;
8     }
9     if (u.child(a)!=null) {
10      u=u.child(a);
11    }
12    v.epsilon=u;
13  }
```

Zložitosť. Uvažujme cestu v lexikografickom strome z koreňa po list pre konkrétnu vzorku P_i a sčítajme čas potrebný na volania findepsilon pre všetky vrcholy na tejto ceste. Uvažujme

hĺbku vrchola u. Na začiatku cesty je táto hodnota 0 a vždy je nezáporná. V každom volaní funkcie sa hĺbka zvýši najviac o 1, takže celkový počet zvýšení je najviac $|P_i|$. Hĺbka vrchola u sa zníži aspoň o jedna každou iteráciou while cyklu, takže týchto iterácií je tiež najviac $|P_i|$. Každú cestu, ktorá končí v liste pre P_i teda spracujeme v čase $O(|P_i|)$. Celkový čas teda je $O(m)$, lebo v algoritme síce spracovávame striedavo vrcholy z rôznych ciest, každý vrchol je však aspoň na jednej ceste, takže jeho čas výpočtu aspoň raz započítame.

Výpočet výstupných hrán. Pre každý vrchol v chceme nájsť prvý epsilonový prechod, ktorý sa má vypísať.

```
1  void find_output(node v) {
2    // Predpokladame w.output je znamy pre vsetky slova
3    // kratsie ako v.
4    u=v.epsilon;  // prvy epsilonovy prechod
5    if (u.id!=null) { // ak je vzorka, ukazujeme nanho
6      v.output=u;
7    } else {          // ak nie je vzorka, jeho najblizsi vystup
8      v.output=u.output;
9    }
10 }
```

Ako sme videli v časti 2.2, lexikografické stromy pri veľkej abecede potrebujú veľa pamäte na uloženie poľa detí, v ktorom bude väčšinou veľa prázdnych položiek. Ak zarátame aj čas na inicializáciu tejto pamäte, algoritmus bude bežať v čase $O(m\sigma+n+k)$ a v pamäti $O(m\sigma)$. Ak použijeme nejakú zložitejšiu štruktúru, napríklad vyvážený binárny vyhľadávací strom, dostaneme algoritmus so zložitosťou $O((m+n)\log\sigma+k)$ a pamäťou $O(m+n)$.

Počet výskytov. Pre abecedu konštantnej veľkosti má Aho-Corasickovej algoritmus časovú zložitosť $O(m+n+k)$, kde k je počet výskytov. Aké veľké môže byť k ako funkcia N? Ukážeme, že Aho-Corasickovej algoritmus má v najhoršom prípade zložitosť $\Theta(N\sqrt{N})$. Najskôr ukážeme dolný odhad, t.j. že existuje vstup, na ktorom $k=\Omega(N\sqrt{N})$. Uvažujme množinu vzoriek $\{a,a^2,\cdots,a^{\sqrt{n}}\}$ a text $T=a^n$. Súčet dĺžiek vzoriek je $\Theta(n)$, teda $N=\Theta(n)$. Vzorka a^i má výskyt na každej pozícii od 1 po $n-i+1$, čo je aspoň $n-\sqrt{n}+1$. Celkový počet výskytov všetkých vzoriek je teda $\Theta(n\sqrt{n})=\Theta(N\sqrt{N})$.

Pre horný odhad musíme dokázať, že k nikdy nebude viac ako $O(N\sqrt{N})$. Vzorky rovnakej dĺžky majú vždy neprekrývajúce sa výskyty a preto ľubovoľný počet vzoriek dĺžky i má spolu najviac $n-i+1$ výskytov. Stačí nám teda uvažovať vzorky typu a^i v texte a^n. Takisto pre dané z a n dosiahneme najvyššie k ako funkciu N, ak máme čo najkratšie vzorky, lebo dlhšie vzorky znižujú k a zvyšujú N. Stačí teda uvažovať vzorky $\{a,a^2,\cdots,a^z\}$. Potom $N=n+z(z+1)/2$ a $k=z(n+1)-z(z+1)/2$. Aby sme overili, že k je najviac $O(N\sqrt{N})$, stačí do vzťahu pre k dosadiť $n=N-z(z+1)/2$, zderivovať ako funkciu od z a nájsť maximum.

5.2 Cvičenia

Čas Aho-Corasickovej algoritmu je $O(n+m+k)$, čo je horšie než lineárne pre veľké k. Závislosť od k je nevyhnutná, ak chceme vypísať všetky výskyty, ale ak nám stačí iná informácia, vieme algoritmus upraviť, aby bežal v čase $O(n+m)$.

Príklad: Modifikácia Aho-Corasickovej algoritmu, aby v čase $O(m+n)$ zistil, či je v texte výskyt aspoň jednej vzorky.

Riešenie: Namiesto vypisovania všetkých výskytov skontrolujeme len či aktuálny vrchol v je vzorka alebo či $v.output$ je vzorka.

Príklad: Celkový počet výskytov k.

Riešenie: V každom vrchole je navyše $v.count$ – počet sufixov v, ktoré sú z P.

Príklad: Nájsť všetky vzorky, ktoré majú aspoň jeden výskyt.

Riešenie: V každom vrchole v mám ísť po ceste výstupných hrán až do koreňa. Poznačím si, či som konkrétny vrchol už vypísal a akonáhle nájdem vrchol, ktorý som už vypísal, skončím. Po každej linke prejdem iba raz. Liniek je m, práca behania po výstupných hranách je $O(m)$.

Príklad: Chceme vektor dĺžky m, v ktorom bude uložený počet výskytov každej vzorky v texte.

Riešenie: Počítadlo v každom vrchole, ktoré hovorí, koľko ráz sa cezeň prešlo. Nakoniec chceme pre každý vrchol spočítať jeho počet aj počet pre všetky stavy, z ktorých sa do aktuálneho stavu vieme dostať po epsilonových hranách.

```
1  for each v in reverse BFS order { // od najhlbsich vrcholov
2    w=v.epsilon;
3    w.passes+=v.passes;
4  }
```

5.3 Bakerov-Birdov algoritmus pre vyhľadávanie matíc

Vzorka P aj text T sú vo forme matice, ktorej prvky sú z abecedy Σ. Hľadáme súradnice ľavého horného rohu všetkých výskytov P v T. Rozmery m a n udávajú celkový počet prvkov v jednotlivých maticiach, t.j. $m = m_1 m_2$ a $n = n_1 n_2$, kde $m_1 \times m_2$ a $n_1 \times n_2$ sú rozmery matíc P a T.

Baker-Birdov algoritmus využíva Ahov-Corasickovej algoritmus na vyhľadávanie viacerých vzoriek. Predpokladajme najskôr, že všetky riadky matice P sú navzájom rôzne. Algoritmus najskôr vytvorí automat pre všetky riadky vzorky P, a následne ním vyhľadávame postupne na všetkých riadkoch vstupu T, pričom vypĺňame maticu A nasledovne:

$$A[i,j] = \begin{cases} k & \text{ak } k\text{-ty riadok } P \text{ sa vyskytuje v matici } T \text{ na} \\ & \text{riadku } i \text{ od pozície } j, \\ -1 & \text{inak.} \end{cases}$$

Na takto vyplnenej matici A môžeme teraz po stĺpcoch spustiť napríklad KMP, ktorý vyhľadáva postupnosť čísel, ktorá zodpovedá vyhľadávanej vzorke (teda slovo $1,2,3,\ldots,m_1$).

Pokiaľ túto postupnosť v niektorom stĺpci nájdeme, našli sme zároveň výskyt pôvodnej vzorky P. Všimnime si, že tu ide o vyhľadávanie nad veľkou abecedou, to však nezhoršuje zložitosť KMP algoritmu.

Ak sa riadky v P opakujú, zitíme to pri budovaní lexikografického stromu a všetkým zhodným riadkom pripradíme to isté číslo.

Príklad: Majme maticu

$$P = \begin{pmatrix} aa \\ ba \\ aa \end{pmatrix}.$$

V Aho-Corasickovej algoritme hľadáme vzorky $P_1 = aa, P_2 = ba$. Potom pomocou KMP hľadáme vzorku 1,2,1 v nejakom stĺpci matice A.

Celková zložitosť tohto algoritmu pre konštantnú abecedu je $O(n+m)$. Člen k sa nevyskytuje, lebo výskyty vzoriek pre Aho-Corasickovej algoritmus sa neprekrývajú a teda ich je spolu najviac n.

6 Boyerov-Moorov algoritmus

Boyerov-Moorov (BM) algoritmus (1977) kontroluje vzorku odzadu a občas vie niektoré časti textu úplne preskočiť. Preto je v praxi často o niečo rýchlejší ako KMP algoritmus. Ako takmer pri každom algoritme na spracovanie textu najskôr vzorku predspracuje a potom ju vyhľadáva v texte. Podobá sa na triviálny algoritmus s kontrolou odzadu:

```
1 i=0;
2 while(i<=n-m){
3    j=m-1;
4    while(j>=0 && P[j]==T[i+j]){
5       j--;
6    }
7    if(j<0){ print i; }
8    i+=skip;
9 }
```

Ak skip $= 1$, máme štandardný triviálny algoritmus (akurát porovnávame vzorky odzadu). Ak však zoberieme text $T = bbbbbb$ a vzorku $P = baa$, tak by sme mohli v obidvoch krokoch nastaviť skip na 2 (pozri obr. 12). Pre všeobecnú vzorku a všeobecný text si nemôžeme dovoliť vždy skákať o 2. Namiesto toho BM algoritmus skáče podľa potreby. Presnejšie podľa nasledovných dvoch pravidiel (skáče podľa toho, ktoré pravidlo povie viac):

- Pravidlo zlého znaku (bad character rule)
- Pravidlo dobrého sufixu (good suffix rule)

6.1 Pravidlo zlého znaku

Nech $R[x]$ je index posledného (najpravejšieho) výskytu písmena $x \in \Sigma$ vo vzorke P, alebo -1, ak x vo vzorke P neexistuje. $R[x]$ vypočítame nasledovným algoritmom:

```
bbbbbbb   <-text
  # # #   <-porovnania
baa       <-vyskúšané posuny vzorky
  baa
    baa
```

Obr. 12: Ak porovnávame vzorku $P = baa$ s textom $T = bbbbbb$, môžeme vždy posunúť vzorku o dve, aby sa jediný výskyt 'b' vo vzorke dostal oproti práve nájdenému 'b' v texte. Celkovo sa pozrieme iba na 3 zo 7 znakov textu.

```
                  i
T  ......................xabcd.......
P                  ....x....yabcd
P                       ....x....yabcd
                           j
```

Obr. 13: Na pozícii j nastala nezhoda a preto môžeme vzorku posunúť na najbližšie x.

```
1 R={-1,-1,-1,...}
2 for(i=0; i<m; i++) {
3     R[P[i]]=i;
4 }
```

Táto časť trvá $O(m+\Sigma)$ a potrebujeme si pamätať pole o veľkosti $O(\Sigma)$.

Pravidlo zlého znaku: Ak $P[j+1..m-1] = T[i+j+1..i+m-1]$ a $P[j] \neq T[i+j] = x$ a $R[x] < j$, tak spravíme posun o $j - R[x]$. Ak $R[x] > j$, t.j. posledný výskyt x je až za súčasnou pozíciou j, posunieme sa o jedna. Pravidlo zlého znaku je ilustrované na obrázku 13.

Pri malých abecedách často dôjde iba k malým posunom, pri veľmi veľkých treba veľa času a pamäte na pole R, ale pri stredne veľkých je to dobrá heuristika.

6.2 Horspoolov algoritmus

Variant BM, ktorý nevyužíva pravidlo dobrého sufixu, iba mierne modifikované pravidlo zlého znaku. Na rozdiel od BM toto pravidlo neuplatňuje na znak, kde nastala nezhoda, ale vždy na najpravejší znak aktuálneho okna v texte, teda na znak $x = T[i+m-1]$. V poli R uložíme posledný výskyt každého znaku v $P[0..m-2]$, neuvažujeme teda posledný znak vzorky. Takto dostávame nasledujúci jednoduchý pseudokód:

```
1   R={-1, -1, -1, ...}
2
3   for(i=0; i<m-1; i++){
4     R[P[i]]=i;
5   }
6
```

```
T:  ..............xabcdefg.............
P:            efg..yabcdefg
P:                      efg..yabcdefg
```

Obr. 14: Príklad pravidla dobrého sufixu.

```
7    i=0;
8    while(i <= n-m){
9      j=m-1;
10     while(j>=0 && P[j]==T[i+j]){
11       j--;
12     }
13     if(j<0){ print i; }
14     i += m-1-R[T[i+m-1]];
15   }
```

V priemernom prípade algoritmus pracuje v čase $O(n/\sigma+m+\sigma)$, v najhoršom $O(mn+\sigma)$. Existuje aj (úplne iný) algoritmus s priemerným prípadom $O(n\frac{log_\sigma m}{m}+m)$ (Lecroq 1992).

6.3 Pravidlo dobréhu sufixu

Ak sme v niektorej iterácii našli zhodu medzi k poslednými znakmi P a príslušnou časťou T, chceme P posunúť tak, aby písmená z P, ktoré budú zarovnané s týmito k znakmi, boli tiež zhodné (pozri obrázok 14).

Budeme písať $P \sim Q$ práve vtedy, keď P je sufix Q alebo Q je sufix P. Ak nájdeme prvú nezhodu na pozícii j v P, posunieme i o $\gamma[j]$ definované takto:

$$\gamma[j] = m - \max\{k \mid 0 \le k < m \wedge P[j+1..m-1] \sim P[0..k-1]\}.$$

Pozrime sa, čo sa stane v dvoch extrémnych prípadoch algoritmu. Ak sa v danej iterácii nenašla žiadna zhoda medzi P a časťou T, premenná j bude mať hodnotu $m-1$, t.j. $P[j+1..m-1] = \varepsilon$. Pre ľubovoľný reťazec Q platí, že $Q \sim \varepsilon$ a teda k nadobudne hodnotu $m-1$, čiže posun bude 1. Naopak, ak nájdeme výskyt vzorky, máme $j = -1$ a teda $P[j+1..m-1] = P$. Hľadáme teda najdlhší vlastný prefix P, ktorý je aj sufixom P. Môže sa stať aj to, že jediný takýto prefix je ε; v tom prípade bude posun m.

Hodnoty $\gamma[-1\ldots m-1]$ si potrebujeme predspracovať, použijeme na to pole sp z Morris-Prattovho algoritmu. Hodnota $sp[i]$ je dĺžka najdlhšieho vlastného sufixu $P[0..i-1]$, ktorý je prefixom P. Použijeme aj pole spr, ktoré spočítame ako pole sp pre P^R, čo je zrkadlový obraz reťazca P. Hodnota $spr[i]$ je dĺžka najdlhšieho vlastného prefixu $P[m-i..m-1]$, ktorý je sufixom P.

Pri výpočte $\gamma[j]$ môžu nastať dve situácie. Prvá je, že $X = P[j+1..m-1]$ vyskytuje v P ešte raz a z týchto výskytov X chceme spočítať pravý koniec toho najpravejšieho. Pre každý výskyt X začínajúci na nejakej pozícii $m-i$ platí, že $spr[i] \ge m-j-1$ a navyše pre najpravejší výskyt platí rovnosť $spr[i] = m-j-1$, lebo pri nerovnosti máme zaručený ešte jeden výskyt viac v pravo, na začiatku sufixu P dĺžky $spr[i]$. Naopak, ak $spr[i] = m-j-1$, na pozícii $m-i$ začína výskyt X. Takže hľadáme najmenšie i, pre ktoré platí rovnosť $spr[i] = m-j-1$. Ak

```
T: ..................xabcd............
P:   ..zabcd..yabcd....yabcd
P:                    ..zabcd..yabcd....yabcd
```

Obr. 15: Silné pravidlo dobrého sufixu: hľadáme posledný výskyt $P[j+1..m]$ v P, pred ktorým ide iné písmeno ako $P[j]$.

X začína na pozícii $m-i$, končiť bude na pozícii $(m-i)+spr[i]-1$, teda za $\gamma[j]$ použijeme $m-(m-i)-spr[i] = i-spr[i]$.

Druhá možná situácia je, že X nemá ďalšiu kópiu v P. Hľadáme teda najdlhší sufix $P[j+1..m]$, ktorý je prefixom P a nakoľko celé X sa druhýkrát nevyskytuje, dĺžka tohto sufixu je $sp[m]$. V nasledujúcom kóde najskôr vypočítame γ podľa tohto pravidla, potom prepisujeme podľa výskytov X ktoré hľadáme zľava (t.j. od najväčších hodnôt i).

```
1 compute sp;
2 compute spr;
3 for(j=-1; j<m; j++) {
4   gamma[j] = m-sp[m];
5 }
6 for(i=m; i>=0; i--) {
7   j = m-spr[i]-1;
8   gamma[j] = i-spr[i];
9 }
```

Ak zoberieme vzorku $P = a^m$ a text $T = a^n$, tak nájdenie všetkých výskytov bude až kvadratické. Dokonca je možné spraviť takú vzorku a text, že vzorka sa v texte nebude nachádzať, ale napriek tomu bude čas behu BM algoritmu kvadratický. Kvôli tomu bolo spravené silné pravidlo dobrého sufixu, ktoré tento problém čiastočne vyrieši. Ak sa vzorka v texte nenachádza, časová zložitosť je $O(n+m)$. Silné pravidlo dobrého sufixu je ilustrované na obrázku 15.

Galil v roku 1979 spravil zlepšenie, ktoré beží v čase $O(n+m)$ v ľubovoľnom prípade. Dosiahol to tak, že ak preskočíme kus textu pomocou pravidla dobrého sufixu (silného), tak vieme, že istý kus v strede vzorky je rovnaký ako text a preto tam nemusíme porovnávať, čím si ušetríme prácu.

7 Rabin-Karp (1987)

Majme hashovaciu funkciu $H : \Sigma^m \to \{0,1,\ldots,p-1\}$. Na začiatku zahashujeme vzorku P a potom jej zahashovanú formu porovnávame so zahashovanými pozíciami v texte.

```
1 hp = H(P)       // hashujeme vzorku P
2 for(i=0; i<=n-m; i++) {
3   ht = H(T[i..i+m-1]);   // hashujeme potencialny vyskyt
4   if(hp == ht) {         // ak sa hashe rovnaju
5     if(P==T[i..i-m+1]) { // skontroluj, ci mame vyskyt
6       write i;
7     }
```

```
8     }
9 }
```

Aby sme dostali efektívny algoritmus, potrebujeme splniť nasledujúce podmienky:

1. Hash by sa mal počítať v čase O(1)
2. Chceme málo falošných zhôd takých, že $h_p = h_t \wedge P \neq T[i \ldots i-m+1]$

Slovo S dĺžky m reprezentujme ako číslo v sústave so základom σ: $\sum_{j=0}^{m-1} \sigma^{m-j-1} S[j]$. Napr. $(10011)_2 = 19$. Naša hashovacia funkcia bude $H(S) = S \bmod p$. Na počítanie budeme používať Hornerovu schému:

$$S = ((S[0]\sigma + S[1])\sigma + S[2])\sigma \ldots$$

```
1 int H(S,p,sigma){
2     h = 0;
3     for(i=0; i<|S|; i++) {
4       h = h*sigma mod p;
5       h = (h + S[i]) mod p;
6     }
7     return h;
8 }
```

Časová zložitosť tohto výpočtu je $O(|S|)$. Nech $H_i = H(T[i..i+m-1])$. Pre hľadanie v texte potrebujeme rýchlo vypočítať H_i z už vypočítanej hodnoty H_{i-1}. Dostaneme, že $H_i = (\sigma(H_{i-1} - \sigma^{m-1}T[i-1]) + T[i+m-1]) \bmod p$. Aby sme nemuseli pracovať s veľkými číslami, σ^{m-1} máme predpočítané tiež modulo p.

Celková časová zložitosť Rabinovho-Karpovho algoritmu je $O(n+m(1+k+f))$, kde k je počet výskytov a f je počet falošných zhôd, t.j. pozícií, kde $H(P) = H_i$, ale $P \neq T[i..i+m-1]$.

7.1 Randomizovaná verzia Rabinovho-Karpovho algoritmu

Dokážeme, že ak za p zvolíme náhodné prvočíslo z určitého rozsahu, priemerná hodnota f bude pre ľubovoľné P a T malá. Na zvolenie náhodného prvočísla môžeme použiť algoritmy z kryptografie používané na generovanie kľúčov.

Veta 1. Nech $\sigma = 2$ a $nm \geq 29$ a nech $K = nm^2$. Ak p je náhodné prvočíslo menšie ako K, pravdepodobnosť aspoň jednej falošnej zhody je najviac $2,52/m$ a priemerný čas Rabin-Karpovho algoritmu je $O(n+mk)$.

Dôkaz (nebrali sme, iba pre informáciu) Najskôr dokážme, že ak pravdepodobnosť aspoň jednej falošnej zhody je najviac $2,52/m$, stredná hodnota počtu zhôd f bude $O(n/m)$ a teda do časovej zložitosti prispeje členom $O(n)$:

$$E(f) = \sum_{i=0}^{n-m+1} i \cdot P(f = i) \leq n \cdot P(f > 0) = 2,52 \cdot \frac{n}{m}$$

Označme $\pi(n)$ počet prvočísel menších alebo rovných n. V dôkaze použijeme nasledujúce vlastnosti prvočísel:

1. $\frac{n}{\ln(n)} \leq \pi(n) \leq 1,26 \cdot \frac{n}{\ln(n)}$
2. pre $n \geq 29$ súčet prvočísel $\leq n$ je $\geq 2^n$
3. $n \geq 29$ tak každé $k \leq n2^n$ je deliteľné najviac $\pi(n)$ prvočíslami

Nech $\alpha_i = P - T[i..i+m-1]$. Keďže nepočítame modulo p, platí, že $\alpha_i = 0 \Leftrightarrow$ výskyt na pozícii i. Súčasne $\alpha_i < 2^i$.

$$S(P,T) = \prod_{i:\alpha_i \neq 0} \alpha_i \leq 2^{mn}$$

Preto $S(P,T)$ má najviac $\pi(mn)$ prvočíselných deliteľov. Vzorka má falošný výskyt v texte práve vtedy, keď α_i je deliteľné prvočíslom p. Z $\pi(K)$ prvočísel najviac $\pi(mn)$ má falošnú zhodu (t.j. delí $S(P,T)$).

$$P(f > 0) \leq \frac{\pi(m.n)}{\pi(K)} \leq 1,26 \cdot \frac{n.m.\log(n.m^2)}{n.m^2.\log(n.m)} \leq \frac{2.52}{m}$$

QED

Namiesto jedného prvočísla môžeme použiť aj L nezávislých prvočísel $p_1, p_2, \ldots, p_L$. Nech $f_{p,i}$ nám označuje udalosť, že náhodne zvolené prvočíslo p má falošnú zhodu na pozícii i. Falošná zhoda v tomto rozšírenom algoritme nastane ak pre nejaké i majú všetky zvolené prvočísla $f_{p,i} = 1$, t.j. pravdepodobnosť, že toto nastane je $P(\exists i \forall j f_{p_j,i})$.

Odhad 1

$$\begin{aligned} P(\exists i \forall j f_{p_{j,i}}) &\leq P(\forall j \exists i f_{p_{j,i}}) \\ &= \prod_j P(\exists i f_{p_{j,i}}) \\ &\leq (\frac{\pi(m.n)}{\pi(k)})^L \end{aligned}$$

Odhad 2 Budeme vychádzať z poznatku, že $\alpha_i < 2^m$ a teda najviac $\pi(m)$ prvočísel môže deliť α_i. Teda pre dané i a náhodné p máme $P(f_{p.i}) \leq \pi(m)/\pi(K)$.

$$P(\exists i \forall j f_{p_{j,i}}) \leq n \cdot P(\forall j f_{p_{j,i}}) = n \cdot P(f_{p_{j,i}})^L \leq n \cdot (\frac{\pi(m)}{\pi(k)})^L$$

Príklad Nech $m = 1000$ a $n = 100\,000$, potom $K = nm^2 = 10^{11} \sim 2^{36}$. Pre $L = 4$ dokázané odhady na pravdepodobnosť falošnej zhody vyzerajú nasledovne:

- pre 1 prvočíslo: $2,52.10^{-3}$
- odhad 1 pre L prvočísel: 4.10^{-11}
- odhad 2 pre L prvočísel: 5.10^{-21}

8 Využitie bitového paralelizmu

8.1 Shift-and algoritmus

Algoritmus bol publikovaný v roku 1992 Ricardom Baeza-Yatesom a Gastonom Gonnetom, avšak podobný algoritmus vynašiel vraj aj Balint Dömölki už v roku 1964. Algoritmus funguje pre krátke vzorky, ktorých dĺžka je menšia ako počet bitov jedného registra. Majme bitovú maticu M s riadkami $0,\ldots,m-1$ a stĺpcami $0,\ldots,n-1$, ktorá je definovaná nasledovne:

$$M(i,j) = 1 \Leftrightarrow P[0\ldots i] = T[j-i\ldots j]$$

V j-tom stĺpci si teda pamätáme stavy NKA z Morris-Prattovho algoritmu, ktoré sú dosiahnuteľné po načítaní $T[0..j]$ (t.j. všetky sufixy $T[0..j]$, ktoré sú prefixami P). Každý stĺpec vieme spočítať z predchádzajúceho podľa vzťahu

$$M(i,j) = (i = 0 \vee M(i-1,j-1)) \wedge P[i] = T[j]$$

pre $i \geq 0, j > 0$.

Pre každé $x \in \Sigma$, nech $U(x)$ je binárny vektor dĺžky m, taký že $U(x)[i] = 1 \Leftrightarrow P[i] = x$.

Jednotlivé stĺpce matice M a vektory $U(x)$ si budeme pamätať ako binárne čísla a robiť s nimi bitové operácie.

Napríklad pre P =aaba dostávame čísla $U(a) = 1011_2$, $U(b) = 0100_2$. Pozor, najľavejší znak P je vo vektoroch reprezentovaný vpravo, na pozícii najmenej významného bitu.

Dostávame takto nový vzťah pre výpočet j-teho stĺpca M z $j-1$:

$$M(\cdot,j) = ((M(\cdot,j-1) << 1|1) \& U(T[j])$$

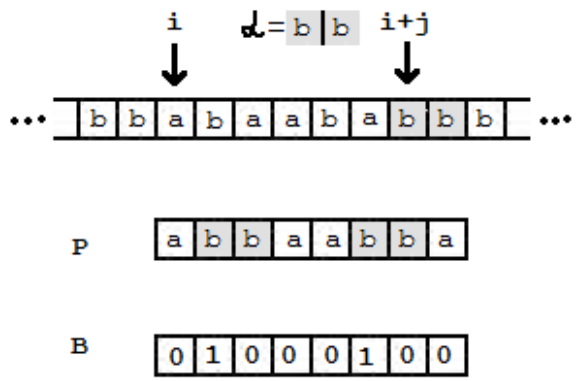
Tu využívame vzťah $P[i] = T[j] \Leftrightarrow U(T[j])[i] = 1$.

Ak m je menej ako dĺžka registra, časová zložitosť je $\Theta(n+m+\sigma)$ operácií s registrami.

8.2 BNDM: Backward non-deterministic DAWG matching

Je to algoritmus na hľadanie jednej vzorky v texte. Používa bitový paralelizmus, preto je vhodný pre krátke vzorky. Publikovali ho Navarro a Raffinot v roku 1998.

V každom kroku, keď máme zarovnanú vzorku na nejaké okno textu s posunom i, hľadáme najdlhšie α, ktoré je (vlastným) sufixom tohto okna a prefixom P. Potom posunieme okno tak, aby nájdené α boli zarovnané. Ako ale nájdeme α? Ideme premennou j postupne od konca okna, v každom kroku pridáme na začiatok α jeden znak. V každom kroku udržujeme všetky výskyty aktuálneho sufixu α v P — to sú kandidáti na najdlhší možný prefix P. Udržujeme si ich pomocou bitového vektora B, v ktorom je jednotka na začiatočnej pozícii každého výskytu α.




```
1  i=0;
2  while(i<=n-m){
3     j = m-1;
4     last_j = m-1;
5     B = (1<<m)-1;   //m krat 1
6     while(B>0 && j>=0){
7        B = (B >> 1) & U[T[i+j]];
8        if(B & 1){   // vyskyt podslova na zaciatku P?
9           if(j==0) print i;   // vyskyt podslova dlzky m?
10          else last_j = j;   // posunme posledny najdeny prefix
11       }
12       j--;
13    }
14    i += last_j;   //posunme sa na posledny najdeny prefix
15 }
```

Pole U obsahuje pre každý znak jeden bitový reťazec, ktorý má jednotky na výskytoch vo vzorke, podobne ako v Shift-and algoritme. To sa predpočíta v $O(m)$ jedným prejdením vzorky s tým, že ešte pole v čase $O(\sigma)$ inicializujeme na nuly.

Najhorší prípad je $O(mn+\sigma)$. Priemerný prípad $O(n\frac{\log_\sigma m}{m}+m+\sigma)$.

9 Prehľad algoritmov

V tabuľke 2 uvádzame prehľad všetkých preberaných algoritmov na hľadanie vzorky v texte.

10 Dolný odhad zložitosti vyhľadávania vzorky v texte porovnávaním

Uvažujme algoritmy, ktoré pristupujú k vzorke a textu len pomocou porovnaní znakov na rovnosť. Takýmto algoritmom je napríklad KMP, ale BM ním nie je, kvôli pravidlu zlého znaku.

Chceme dolný odhad počtu porovnaní, ktoré každý takýto algoritmus musí v najhoršom prípade urobiť. Majme vzorku $P = a^m$ a zatiaľ nešpecifikovaný text T dĺžky n nad abecedou

Algoritmus	najhorší prípad	priemerný prípad	porovnávanie	Poznámka
triviálny	$O(nm)$	$O(n+m)$	áno	jednoduchý
Gusfield	$O(n+m)$	$O(n+m)$	áno	zákl. predspracovanie
DKA	$O(n+\sigma m)$	$O(n+\sigma m)$	nie	real time
(K)MP	$O(n+m)$	$O(n+m)$	áno	
Aho Corasick	$O((n+m)\sigma+k)$	$O((n+m)\sigma+k)$	áno/nie	viac vzoriek
Boyer Moore	$O(nm), O(n+m)$	$O(\frac{n}{\sigma}+m)$	nie	
Shift-and	$O(n+m+\sigma)$	$O(n+m+\sigma)$	nie	pre m bitov v registri
BNDM	$O(nm+\sigma)$	$O(n\frac{\log_\sigma m}{m}+m+\sigma)$	nie	pre m bitov v registri
Rabin-Karp	$O(nm)$	$O(n+mk)$	nie	randomizovaný

Tabuľka 2: Prehľad prezentovaných algoritmov

$\{a,b\}$. Na tomto vstupe budeme simulovať algoritmus a na každé porovnanie na rovnosť odpovieme áno. Súčasne si zostavíme graf, v ktorom vrcholy budú jednotlivé znaky textu T a jeden vrchol pre celý reťazec P. Máme teda $n+1$ vrcholov.

Každá otázka spojí dva vrcholy, na ktoré sme sa pýtali, hranou. Po menej ako n otázkach nám zostanú aspoň dva komponenty súvislosti. Ak teda algoritmus spraví menej ako n porovnaní, vieme zostrojiť dva reťazce T, ktoré sedia so všetkými otázkami, ale dávajú rôzne odpovede, takže aspoň na jeden z nich by algoritmus zle odpovedal. Jeden z nich je $T = a^n$ a v druhom dáme a na všetky pozície, ktoré sú v tom istom komponente ako P a na zvyšné pozície dáme b.

Veta 2. Každý algoritmus, ktorý pristupuje k reťazcom P a T len pomocou porovnaní na rovnosť pre každé n a $m < n$ potrebuje v najhoršom aspoň n porovnaní a pracuje teda v čase $\Omega(n)$.

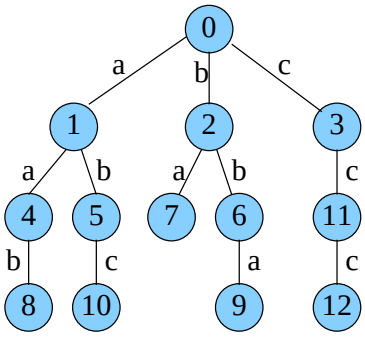
Cole a kol. 1995 pre vzorku $P = aba$ dokázali lepší dolný odhad $\frac{6}{5}n - o(n)$.

11 Hľadanie vzorky v komprimovanom texte

Pri veľkých súboroch ich chceme držať v komprimovanom stave a vyhľadávať v nich vzorku bez toho, aby ich bolo nutné dekomprimovať. Ako n teraz označíme veľkosť komprimovaného vstupu, u nekomprimovaného, chceme pokiaľ možno byť nezávislý od u, ideálne $O(n+m)$. Dĺžka vzorky m sa meria v nekomprimovanom stave.

11.1 Run length encoding, RLE

- Zapíš k po sebe idúcich kópií znaku a ako a^k. Dobré kódovanie napr. na binárne bitmapy, často väčšie oblasti bielej alebo čiernej. Text komprimovanej dĺžky n má teda tvar $t_1^{k_1}, \ldots, t_n^{k_n}$, kde $k_i \geq 1$ a $t_i \neq t_{i+1}$. Nezaujíma nás teraz počet bitov textu, ale počet run-ov.
- Jednoduchá idea vyhľadávania: považujeme a^k za znak (a,k) v špeciálnej abecede (nekonečnej veľkosti), zakódujeme tak aj vzorku a použime KMP algoritmus.
- Problém so začiatkom a koncom vzorky. Nech vzorka začína v RLE znakom (a,k) a končí (b,ℓ), pričom predpokladajme, že $a \neq b$. Potom vo vzorke aj texte všetky znaky



Text: aabbaabbabccccccc
Komprimovaný text: 1, 1, 2, 2, 4, 6, 5, 3, 11, 12

Obr. 16: Príklad LZW kompresie a vzniknutého lexikografického stromu

(a, j) pre $j > k$ nahradíme dvojicou znakov $(a, j-k)(a,k)$ a podobne znaky (b, j) pre $j > \ell$ nahradíme dvojicou $(b,\ell)(b, j-\ell)$. Teraz už môžeme naozaj použiť KMP algoritmus.

- Cvičenie: čo so vzorkami, ktoré začínajú aj končia na rovnaké písmeno?
- Zložitosť $O(n+m)$, komprimovaný text môže čítať a spracovávať po znakoch. Zložitosť zostáva $O(n+m)$ dokonca aj ak m by bola komprimovaná dĺžka vzorky.

11.2 LZW kompresia (Lempel–Ziv–Welch 1984)

Používaná vo formáte GIF, v minulosti populárna na UNIXe, ale mala aj problémy kvôli patentovanému algoritmu.

Algoritmus kompresie číta text a postupne buduje lexikografický strom. Vrcholy stromu sú číslované. Na začiatku zostavíme strom, v ktorom koreň má číslo 0 a list pre každý znak abecedy s číslami $1 \ldots \sigma$. Predpokladajme teraz, že sme už zakódovali určitú časť textu a vytvorili tým nejaký väčší strom. Podľa začiatku doteraz nezakóvaého textu algoritmus pokračuje po strome kým sa dá, potom do komprimovaného súboru vypíše číslo vrchola, v ktorom skončil. Pozrie sa na ďalšie písmeno textu a pridá ho ako dieťa aktuálneho vrcholu (toto písmeno však zatiaľ zostáva nekomprimované). Potom sa vráti do koreňa a opakuje. Príklad sa nachádza na obrázku 16. Ako n budeme označovať počet vypísaných čísel vrcholov.

Cvičenie: aký dlhý bude zápis pre a^n?

Dekompresia: vytvárame ten istý strom, ako pri kompresii. Nech komprimovaný text je $z_0, \ldots, z_{n-1}$. Na začiatku začneme strom s prvkami abecedy. Po prečítaní čísla vrchola z_i zo vstupu vypíšeme reťazec pre tento vrchol stromu. Pozrieme sa na ďalšie číslo z_{i+1} zo vstupu a prvý znak z reťazca pre tento vrchol stromu pridáme ako nový vrchol pod vrchol z_i. Môže sa ale stať, že vrchol z_{i+1} ešte v strome nie je, práve ho ideme pridať. Vtedy použijeme prvý znak slova pre vrchol z_i, ktorý bude súčasne aj prvým znakom pre z_{i+1}. Ak by sme chceli tento strom iba vytvárať a nevypisovať dekomprimovaný text, dá sa to v čase $O(n)$ (v každom vrchole si potrebujeme pamätať aj prvé písmeno slova pre tento text).

Na vyhľadávanie vzorky v komprimovanom texte si ukážeme ideu algoritmu od autorov Amir, Benson, Farach 1994, ktorý funguje v čase $O(n+m^2)$, aj keď v našej verzii skôr $O(n+m^3)$. Navyše budeme hľadať iba prvý výskyt vzorky. Predpokladáme konštantnú abecedu a budeme simulovať, čo by robil deterministický konečný automat na dekomprimovanom texte, ale chceme vždy v konštantom čase spracovať celý úsek textu reprezentovaný jedným číslom vrchola zo vstupu. Po načítaní čísla vrchola z_q chceme v konštantnom stave zistiť, v akom

stave bude DKA na konci textu reprezentovanom týmto vrcholom. Pripomeňme si, že tento stav zodpovedá najdlhšiemu sufixu textu, ktorý je prefixom vzorky P. Sú dva prípady:

- Tento sufix je celý vo vnútri reťazca reprezentovaného vrcholom z_q. Pre každý vrchol lexikografického stromu teda chceme vedieť, aký najdlhší sufix jeho slova je prefixom vzorky. Túto hodnotu si budem ukladať priamo vo vrchole a spočítame ju ľahko pri vytváraní vrcholu aplikovaním jedného kroku DKA na stav v jeho rodičovi.

- Tento sufix obsahuje celý reťazec reprezentovaný vrcholom z_q a navyše zasahuje aj do textu reprezentovaného $z_0 \dots z_{q-1}$. To znamená, že vrchol z_q reprezentuje nejaký podreťazec $P[j..k]$ vzorky P. Takých vrcholov je v strome $O(m^2)$. Nech stav DKA po prečítaní z_{q-1} bol i, teda zodpovedá prefixu $P[0..i-1]$. Predpočítame si vopred tabuľku veľkosti $O(m^3)$, ktorá pre každé i, j a k vráti najdlhší sufix v $P[0..i-1]P[j..k]$, ktorý je prefixom P. Indexy j a k budeme mať uložené vo vrchole a spočítame ich hoci aj triviálnym vyhľadávaním v P, ale len pre vrcholy, kde rodič bol podreťazcom P.

Ak po niektorom čísle dostaneme hodnotu stavu m, našli sme výskyt. Môže sa však stať, že DKA by nadobudol stav m niekde uprostred úseku reprezentovaného jedným číslom a výpočet týchto stavov sme preskočili. Prvý výskyt vzorky nikdy nebude celý vo vnútri úseku reprezentovaného jedným číslom, lebo to číslo zodpovedá reťazcu, ktorý sme už museli predtým nájsť v texte. Teda $P[i..m-1]$ bude na začiatku úseku kódovaného číslom z_q a $P[0..i-1]$ bude v predchádzajúcich úsekoch. Pritom stav po prečítaní z_{q-1} musí byť nejaké $j \geq i$. Pre každý vrchol reprezentujúci slovo u si pamätáme aj dĺžku najdlhšieho prefixu u, ktorý je sufixom vzorky. Toto si tiež skopírujeme z rodiča a zmeníme len ak celé u je sufixom, čo sa stane iba m krát v niektorých z $O(m^2)$ vrcholoch, ktoré sú podslovami P. Ak teda cez hranicu medzi z_{q-1} a z_q ide výskyt vzorky, musí sa nachádzať vo vnútri $P[0..i-1]P[k..m-1]$. Pre každé i a k si predpočítame, či tento reťazec vzorku obsahuje a ak áno, tak kde.

12 Regulárne výrazy

Aj keď by mali byť čitateľovi týchto poznámok regulárne výrazy dôverne známe, nezaškodí pripomenúť si ich definíciu.

Definícia 2. Regulárny výraz definujeme rekurzívne takto:

- ε je regulárny výraz generujúci prázdny jazyk $L(\varepsilon) = \{\varepsilon\}$
- Ak $a \in \Sigma$, tak potom je a regulárny výraz generujúci jazyk $L(a) = \{a\}$
- Ak R_1 a R_2 sú regulárne výrazy aj
  - (R_1) a generuje jazyk $L((R_1)) = L(R_1)$
  - $(R_1.R_2)$ a generuje jazyk $L((R_1.R_2)) = L(R_1).L(R_2)$
  - $(R_1|R_2)$ a generuje jazyk $L((R_1|R_2)) = L(R_1) \cup L(R_2)$
  - (R_1^*) a generuje jazyk $L((R_1)^*) = L(R_1)^* = \cup_{i=0}^{\infty} L(R_1)^i$

Regulárne výrazy nájdu široké uplatnenie v každodennom živote. Napríklad regulárnym výrazom `[0-9][0-9][0-9] [0-9][0-9]` môžeme overovať slovenské poštové smerovacie čísla. Regulárne výrazy používa aj unixový program `grep`. Taktiež si môžeme všimnúť, že regulárne výrazy generujú práve množinu regulárnych jazykov rovnako ako napríklad konečné automaty alebo regulárne gramatiky.

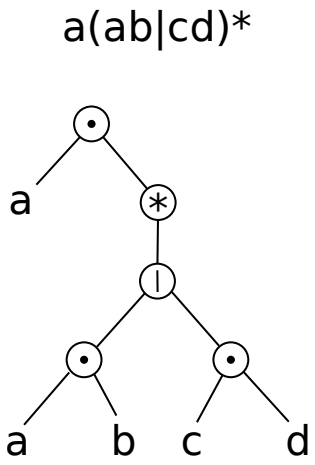
Príklad: Majme regulárny výraz $a(((a.b)|(cd))^*)$ ktorý po vynechaní zátvoriek môžeme zapísať ako $a(ab|cd)^*$. Tento regulárny výraz generuje jazyk
$L(R) = \{a, aab, acd, aabab, aabcd, acdab, acdcd, \ldots\}$

Pri vyhľadávaní regulárnych výrazov dostaneme na vstup text T dĺžky n a regulárny výraz R dĺžky m. Našou úlohou bude nájsť všetky výskyty slov z jazyka $L(R)$.

12.1 Thompsonov algoritmus (1968)

Pri spracovávaní regulárnych výrazov by nám mohla prísť vhod aj šikovnejšia reprezentácia regulárneho výrazu ako len obyčajný reťazec. Jednou z takýchto reprezentácii je aj tzv. parse tree:

- Listami stromu sú prvky abecedy Σ alebo prázdne slovo ε.
- Vnútornými uzlami stromu sú reprezentované operátory $|$ a $.$ s dvoma synmi a operátor $*$ s jediným synom.

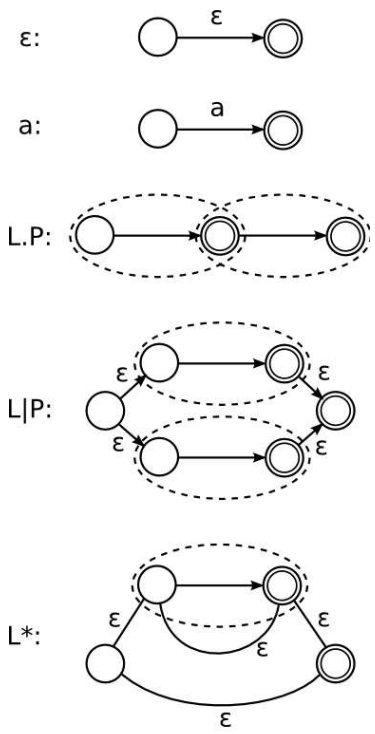


Obr. 17: Strom regulárneho výrazu

Úlohou Thompsonovho algoritmu je z regulárneho výrazu R za pomoci vyššie popísaného stromu skonštruovať nedeterministický konečný automat akceptujúci rovnaký jazyk ako R. Tento automat si z parsovacieho stromu zostrojíme jednoducho smerom zdola nahor nasledujúcim rekurzívnym postupom:

- Pre jednotlivé uzly budeme konštruovať automaty s jedným počiatočným a akceptačným stavom.

- Automat pre ε bude mať počiatočný a akceptačný stav, spojené budú epsilónovým prechodom.
- Automat pre *a* bude mať počiatočný a akceptačný stav, spojené budú prechodom na znak *a*.
- Automat pre konkatenáciu vyrobíme z jednoduchších automatov tak, že spojíme akceptačný stav prvého s počiatočným stavom druhého automatu.
- Automat pre zjednotenie vytvoríme tak, že najprv vytvoríme nový počiatočný a akceptačný stav. Z počiatočného stavu urobíme ε prechody do počiatočných stavov jednoduchších automatov. Z akceptačných stavov jednoduchších automatov povedieme ďalšie prechody na ε do nového akceptačného stavu.
- Automat pre iteráciu zostrojíme tak, že nový počiatočný a akceptačný stav prepojíme prechodom na ε alebo viacerými prechodmi cez jednoduchší automat.
- Do počiatočného stavu výsledného automatu nakoniec pridáme slučku na každé písmeno abecedy, aby sme mohli prečítať ľubovoľne dlhý nezaujímavý úsek textu pred výskytom regulárneho výrazu.



Obr. 18: Ilustrácia rekurzívneho spájania automatov pre jednotlivé operátory regulárneho výrazu

Takouto konštrukciou vieme zostrojit nedeterministický automat, ktorý bude mať pre *m* uzlov parsovacieho stromu najviac 2*m* stavov a 4*m* prechodov. Vieme totiž zaručiť, že každý stav má najviac dve vstupné a dve výstupné hrany.

Zostrojenie parsovacieho stromu nám zaberie najviac $O(m)$ času. Výsledný automat vieme simulovať v čase $O(mn)$ Thompsonovým algoritmom. V bitovom poli `B` si budeme pamäť čísla dosiahnuteľných stavov a postupne ich doplňať o ďalšie.

```
1  B[0] = 1; B[1..k-1]=(0,0,...);
2  epsilon(B);
3  for(int i=0; i<n; i++) {
4    B2[0..k-1] = 0;
5    foreach transition a->b labeled T[i] {
6      if(B[a]==1) { B2[b]=1; }
7    }
8    B=B2;
9    epsilon(B);
10   if(B[k-1]) { print occurrence ending at i }
11 }
12
13 void epsilon(B) {
14   for(int i=0; i<k; i++) {
15     if(B[i]) { search(B,i); }
16   }
17 }
18 void search(B,i) {
19   foreach epsilon transition i->j {
20     if(! B[j]) {
21       B[j]=1; search(B,j);
22     }
23   }
24 }
```

Cvičenie: Thompsonov algoritmus nájde konce výskytov, ako by sme našli začiatky, prípadne pre každý koniec najbližší začiatok?

12.2 Iné prístupy

Pre vyhľadávanie regulárnych výrazov vieme použiť aj mierne obmeny predchádzajúceho algoritmu. Môžeme sa v čase $O(m^2 2^m \sigma)$ pokúsiť zostrojit deterministický konečný automat s exponenciálnym počtom stavov, ktorý ale bude pracovať v čase $O(n)$. Iným prístupom by bolo vytvoriť viacero malých deterministický automatov, ktoré by sme prepojili do jedného veľkého automatu.

Vyhľadávanie si tiež môžeme zjednodušiť filtrovaním vzorov, o ktorých vieme, že sa v regulárnom výraze musia vyskytnúť. Zrýchlenie by sme mohli dosiahnuť použitím bitového paralelizmu.

12.3 Vzorky so žolíkmi

Môžu nastať situácie, keď nepotrebujeme presne určiť niektoré symboly podslova, ktoré chceme nájsť, ale vystačíme si s ľubovoľným znakom. Vytvoríme si nový symbol $*$, ktorý bude zastu-

povať ľubovoľný iný symbol z abecedy Σ. Ide teda o špeciálnu podtriedu regulárnych výrazov (pozor, znak $*$ už neznamená Kleeneho uzáver ale žolík).

Príklad: Regulárnemu výrazu `aa*b` vyhovujú texty `aaab`, `aabb`,`aacb`,`aadb` atď.

Na vyhľadávanie takýchto výrazov si vieme ľahko upraviť triviálny algoritmus. Stačí rozšíriť podmienku, ktorou sme overovali rovnosť znakov napríklad takto:

```
if (P[j] == '*' || P[j] == T[i, j] ...
```

Obdobne vieme upraviť aj algoritmy BNDM a Shift-and.

Časová zložitosť upraveného triviálneho algoritmu ostane $O(nm)$. Upravený algoritmus Shift-and bude pracovať v čase $O(n+m+\sigma)$. Pomocou sufixových stromov si neskôr ukážeme aj algoritmus pracujúci v čase $O(kn)$, kde k je počet hviezdičiek v regulárnom výraze. Teraz sa však pozrieme ako sa dá využiť rýchla Fourierova transformácia (Fast Fourier Transform - FFT) na zkonštruovanie algoritmu pracujúceho v čase $O(n \log m)$.

12.4 Využitie Fast Fourier Transform na hľadanie vzoriek

Fourierova transformácia sa hojne používa pri analýze signálu alebo v oblasti počítačového videnia. Pomocou Fourierovej transformácie si vieme rozložiť zložitý signál na jednotlivé zložky. Efektívny algoritmus na výpočet Fourierovej transformácie sa však dá použiť na násobenie polynómov. Ukážeme si, ako pomocou takéhoto násobenia polynómov využitím FFT dokážeme vyhľadávať regulárne výrazy v texte. Najprv sa však rozpamätajme, čo to vlastne násobenie polynómov je.

Násobenie polynómov. K daným polynómom

$$\begin{aligned} A(x) &= \Sigma_{k=0}^{n-1} a_k x^k \\ B(x) &= \Sigma_{k=0}^{n-1} b_k x^k \end{aligned}$$

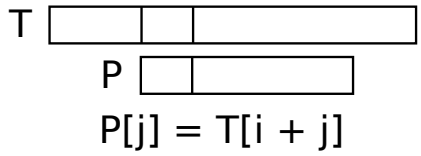
chceme vypočítať polynóm

$$\begin{aligned} C(x) &= A(x)B(x) \\ C(x) &= \Sigma_{k=0}^{2n-1} c_k x^k \text{kde} \\ c_k &= \Sigma_{j=0}^{n-1} a_j b_{k-j} \end{aligned}$$

V prípade potreby si môžeme nedefinované koeficienty položiť rovné nule. Triviálnym algoritmom vypočítame súčin polynómov v čase $O(n^2)$, Fourierovou transformáciou to však zvládaneme v čase $O(n \log n)$. Ukážme si ale najprv, ako vyhľadať výskyt obyčajnej textovej vzorky výrazu pomocou násobenia polynómov.

Hľadanie vzorky v texte. Pozrime sa na znaky našej abecedy na chvíľu ako na čísla. Náš text tak bude nad abecedou $\Sigma = \{1, \ldots, m+1\}$. Pre každé $i = 0, \ldots n-m$ si vypočítajme výraz

$$a_i = \Sigma_{j=0}^{m-1} (P[j] - T[i+j])^2$$



Obr. 19: Reprezentácia hodnoty a_i

Čomu ale takýto výraz zodpovedá? Môžeme si ho predstaviť ako akési ohodnotenie zarovnania textu a regulárneho výrazu začínajúceho na pozícii i.

Ďalej si treba uvedomiť, že jednotlivé členy sumy sú rovné nule práve vtedy, keď nastáva zhoda na príslušnom písmene textu a vzorky. Celá suma bude teda rovná nule, ak nastala zhoda na všetkých pozíciach vzorky a prehľadávaného textu. K ďalším záverom dôjdeme, ak si túto sumu rozpíšeme.

$$\begin{aligned} a_i &= \Sigma_{j=0}^{m-1}(P[j]-T[i+j])^2 \\ &= \Sigma_{j=0}^{m-1}P[j]^2 - 2\Sigma_{j=0}^{m-1}P[j]T[i+j] + \Sigma_{j=0}^{m-1}T[i+j]^2 \end{aligned}$$

Môžeme si všimnúť, že prvý člen súčtu nezávisí na pozícii vzorky v texte, preto sa dá predpočítať. Poslednú sumu si síce predpočítať nevieme, ale vieme ju rátať rýchlo pomocou prefixových súm.

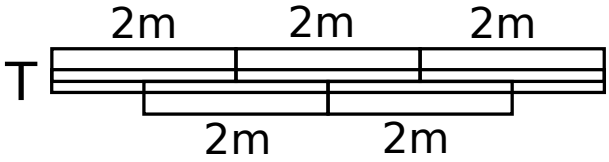
$$\begin{aligned} g_i &= \Sigma_{j=0}^{i-1}T[j]^2 \\ g_{i+1} &= g_i + T[i]^2 \end{aligned}$$

A teda posledný člen súčtu je pre a_i rovný $g_{i+m} - g_i$. Ostáva nám vypočítať druhý člen. Tu si pomôžeme trikom a prehľadávaný text reverzneme. Výsledkom bude výraz, ktorý sa viac podobá na násobené polynómy spomínané vyššie.

$$\begin{aligned} \Sigma_{j=0}^{m-1}P[j]T[i+j] &= \Sigma_{j=0}^{m-1}P[j]T^R[n-1-i-j] \\ &= \Sigma_{j=0}^{m-1}P[j]T^R[k-j] \end{aligned}$$

Tento výraz už vieme vypočítať pomocou FFT v čase $O(n\log n)$ pre reverznutý text a pôvodný vzor, t.j. pre $T^R.P$. Túto časovú zložitosť však vieme zlepšiť použitím ďalšieho triku. Vzorový text si rozdelíme do $\frac{n}{m}$ okienok dĺžky $2m$, pričom nasledujúce okienka sa budú prekrývať o m symbolov. Vyššie naznačeným výpočtom nájdeme vzory v jednotlivých okienkach v čase $O(m\log m)$. Keďže okienok je $\frac{n}{m}$, výsledný čas bude $O(n\log m)$, ktorý je o čosi lepší, pretože $m < n$.

Tento algoritmus zrejme nebudeme chcieť použiť na nájdenie obyčajnej vzorky, pretože máme rýchlejšie algoritmy. Je však veľmi jednoducho rozšíriteľný na vyhľadávanie vzoriek so žolíkmi.



Obr. 20: Rozdelenie prehľadávaného textu.

Využitie Fourierovej transformácie na hľadanie vzoriek s žolíkmi. Ak si uvedomíme, že zhodu na jednotlivých znakoch sme v predchádzajúcom algoritme rozoznávali tak, že jeden člen zložitej sumy je nulový, ľahko vymyslíme spôsob, ako testovať rovnosť na žolíka. Pre naše potreby bude rozumné, ak si symbol $*$ budeme reprezentovať hodnotou 0. Výslednú sumu tak upravíme nasledovne:

$$\begin{aligned} a_i &= \Sigma_{j=0}^{m-1}(P[j]-T[i+j])^2 T[i+j]P[j] \\ &= \Sigma_{j=0}^{m-1}P[j]^3 T[i+j] - 2\Sigma_{j=0}^{m-1}P[j]^2 T[i+j]^2 + \Sigma_{j=0}^{m-1}P[j]T[i+j]^3 \end{aligned}$$

Výskyt žolíkov povolíme aj v texte, čo nájde uplatnenie pri prehľadávní neúplných alebo poškodených textov, pri ktorých si nemôžeme byť istý všetkými ich symbolmi. Ak sa na danej pozícii vyskytuje žolík, tak potom príslušný člen sumy bude nulový bez ohľadu na hodnotu toho druhého symbolu. Na počítanie tohto výrazu použijeme postupne tri FFT, konkrétne na súčiny polynómov T^3P, T^2P^2 a TP^2.

Ostáva nám ešte objasniť, ako pomocou Fourierovej transformácie vynásobiť dva polynómy.

Násobenie polynómov pomocou FFT. Pre lepšie pochopenie ďalšieho textu nám pomôže, ak si uvedomíme, že polynóm môžeme reprezentovať dvoma rôznymi spôsobmi, pomocou koeficientov pri jednotlivých x^k, alebo podľa aspoň $n+1$ bodov v rovine [1], ktorými prechádza graf daného polynómu. Medzi jednotlivými reprezentáciami vieme konkrétny polynóm prevádzať, body v rovine jednoducho určíme tak, že ich do polynómu dosadíme. Opačnou transformáciou je interpolácia.

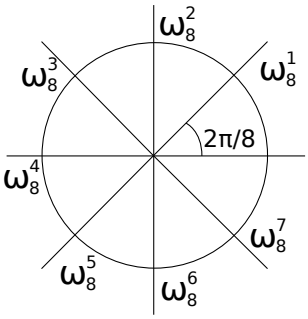
Reprezentácia pomocou bodov v rovine je pre nás výhodná, lebo pomocou nej vieme polynómy násobiť veľmi jednoducho. Vezmime si teda n bodov prislúchajúcich polynómu [2]:

$$\begin{aligned} (x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n) \\ y_i = A(x_i) \end{aligned}$$

Cez tieto body prechádza práve jedna krivka stupňa menej ako n. Súčin dvoch polynómov si takto vieme zrátať po bodoch. Jednoducho si vypočítame n bodov pre polynóm B v rovnakých súradniciach x_i ako pre polynóm A. Hodnoty y_i pre súčin polynómov $A(x)B(x)$ získame súčinom hodnôt y_i pre jednotlivé polynómy.

[1] kde n je stupeň polynómu

[2] n musí byť dosť veľké, aby sa doň zmestil aj stupeň súčinu polynómov.



Obr. 21: Vizualizácia ôsmych odmocnín komplexnej jednotky

Stupeň polynómov s ktorými počítame je však príliš malý, preto ho zvýšime [3] na n také, aby sa doň zmestil aj výsledný súčin. Ďalej je potrebné, aby tento stupeň bola mocnina dvojky.

Hodnoty y_i pre dané polynómy vieme určiť v $O(n^2)$ čase pomocou Hornerovej schémy ($O(n)$ čas na každé y_i). Transformácia v opačnom smere, t.j. ako z množiny bodov určiť koeficienty polynómu, ktorý cez ne prechádza by sa dala riešiť sústavu rovníc, kedy by nám každý pár (x_i, y_i) prispel jednou rovnicou. V oboch prípadoch je však rýchlejším riešením použitie Fourierovej transformácie.

Z teórie Fourierovej transformácie vyplýva, že jej výpočtom pre daný vektor koeficientov polynómu môžeme vyhodnotiť polynóm v n rôznych n-tých komplexných odmocninách jednotky. Nasleduje jednoduché násobenie po správanych súradniciach a spätný prevod do reprezentácie pomocou koefecientov využitím inverznej Fourierovej transformácie.

Potrebné komplexné odmocniny vypočítame pomocou nasledujúceho vzorca.

$$\omega_n \quad = \quad e^{2\pi i} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$$

Násobenie na jednotkovej kružnici v komplexnej rovine sa správa ako sčítanie uhlov, pripomína teda sčitovanie modulo n: $\omega_n^i \omega_n^j = \omega_n^{i+j \bmod n}$.

Pomocou tzv. rýchlej Fourierovej transformácie (Fast Fourier Transform - FFT) vypočítame hodnoty polynómu v komplexných odmocninách metódou rozdeľuj a panuj [4]. Koeficienty polynómu A si rozdelíme do dvoch skupín podľa toho, či sú pri párnej alebo nepárnej mocnine x a získame tak dva polynómy A_1 a A_2.

	x^0	x^1	x^2	$\ldots$	$x^{n/2-1}$	$\ldots$	x^{n-1}
A:	a_0	a_1	a_2	$\ldots$	$a_{n/2-1}$	$\ldots$	a_{n-1}
A_1:	a_0	a_2	a_4	$\ldots$	a_{n-2}		
A_2:	a_1	a_3	a_6	$\ldots$	a_{n-1}		

Pre takto rozdelený polynóm platí $A(x) = A_1(x^2) + xA_2(x^2)$

Využitím vzťahu $\omega_n^{2j} = \omega_{n/2}^j$ platného pre komplexné odmocniny zrátame výslednú hodnotu pomocou výsledkov dosiahnutých v rekurzívnom kroku. Hodnoty pre mocniny ω_n^{2j} už poznáme pre $j \leq \frac{n}{2} - 1$. Ak je $j > \frac{n}{2} - 1$, využijeme modulárnosť násobenia na kruhu:

$$\omega_{n/2}^j = \omega_{n/2}^{n/2}\omega_{n/2}^{j-n/2} = 1\omega_{n/2}^{j-n/2} = \omega_{n/2}^{j-n/2}$$

[3] Pre vysoké mocniny nastavíme nulové koeficienty

[4] Práve rozdeľovacia fáza tohto prístupu spôsobí, že potrebujeme polynóm, ktorého stupeň je mocnina dvojky

Teraz už môžeme smelo písať program na výpočet FFT.

```
1  complex FFT(A,n) {
2    if n=1, return (A[0]);
3    A1 = A[0,2,4,...,n-2];    Y1 = FFT(A1);
4    A2 = A[1,3,5,...,n-1];    Y2 = FFT(A2);
5    omega = cos(2*pi/n) + i* sin(2*pi/n);
6    x = 1;
7    for(int j=0; j<n/2; j++) {
8      Y[j] = Y1[j] + x*Y2[j];
9      Y[j+n/2] = Y1[j] - x*Y2[j];
10     x = omega*x;
11   }
12   return Y;
13 }
```

Tento výpočet bude pracovať v čase určenom rekurenciou $T(n) = 2T(n/2) + O(1)$ teda v čase $O(n \log n)$.

Celkové násobenie polynómov zostavíme postupom:

- nájdi $n = 2^k$ také, že $A(x)B(x)$ má stupeň $< n$
- pridaj nulové koeficienty, aby A a B mali dĺžku n
- spočítaj $A(\omega_n^j)$ pre $j = 0 \ldots n-1$ pomocou FFT
- spočítaj $B(\omega_n^j)$ pre $j = 0 \ldots n-1$ pomocou FFT
- spočítaj $C(\omega_n^j) = A(\omega_n^j)B(\omega_n^j)$ pre všetky j v $O(n)$
- preveď $C(x)$ späť na koeficienty pomocou FFT

Zostáva vysvetliť spätný prevod polynómu na koeficienty: TODO, len krátke poznámky:

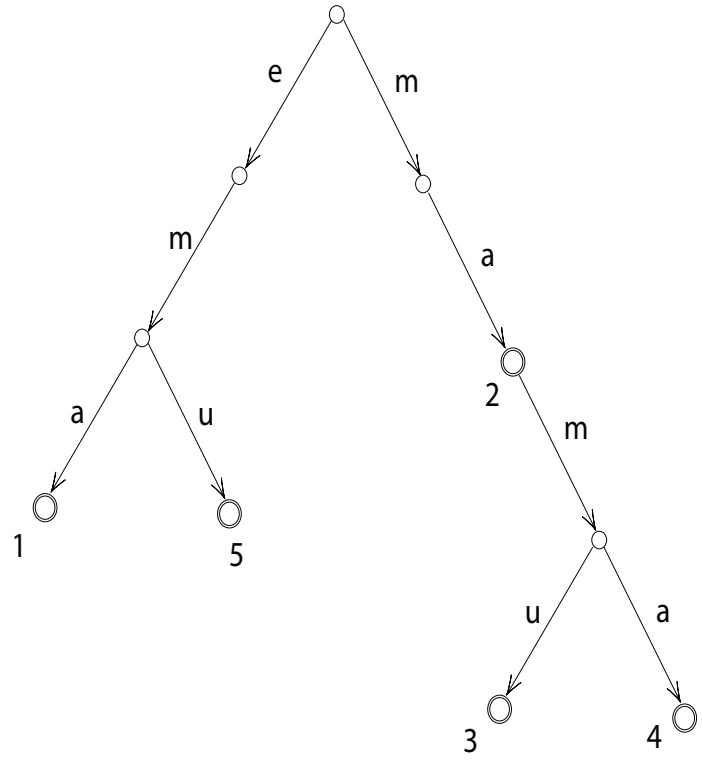
Given values of $y_j = A(\omega_n^j)$ for $j = 0 \ldots n-1$, compute A with degree $\leq n$. Coefficient $a_j = (1/n)\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$ (without proof). Compare with original FFT computing $y_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}$. Use FFT-like algorithm, use Y as A, use ω_n^{-1} instead of ω_n, multiply result by $1/n$.

12.5 TO DO

Spätný prevod na koeficienty pri FFT. Trochu podrobnejšie vysvetliť rozdeľuj a panuj algoritmus pre FFT.

13 Lexikografické stromy

Na jednej z predchádzajúcich prednášok sme si už spomínali dátovú štruktúru trie. Trie je strom, ktorého každá hrana je označená znakom z abecedy Σ, pričom hrany z jedného vrchola sú označené rôznymi znakmi. Slovo zodpovedajúce vrcholu dostaneme konkatenáciou znakov na ceste z koreňa. Vrcholy zodpovedajú prefixom slov zo vstupnej množiny $S_1, S_2, \ldots$. Ak vrchol zodpovedá S_i označíme ho indexom i (tak ako na obrázku č. 1).



Obr. 22: Trie pre $\{ema, ma, mamu, mama, emu\}$

13.1 Úlohy

Je Q v $\{S_1, ..., S_z\}$? Predpokladáme, že máme trie. Postupne prechádzame po hranách a môže nastať:

- Minieme Q a skončíme v akceptačnom vrchole $\Rightarrow$ vrátime "áno".
- Minieme Q a skončíme v neakceptačnom vrchole $\Rightarrow$ vrátime "nie".
- Neminieme Q a nemáme sa kam posunúť $\Rightarrow$ vrátime "nie".

Zložitosť je zjavne $O(m)$, kde $m = |Q|$.

Pridanie nového Q

- Minieme Q a skončíme v akceptačnom vrchole $\Rightarrow$ nerobíme nič, keďže dané slovo už v trie je.
- Minieme Q a skončíme v neakceptačnom vrchole $\Rightarrow$ označíme vrchol ako akceptačný.
- Neminieme Q a nemáme sa kam posunúť $\Rightarrow$ pridáme hrany pre zvyšok slova.

Zložitosť je znova $O(m)$.

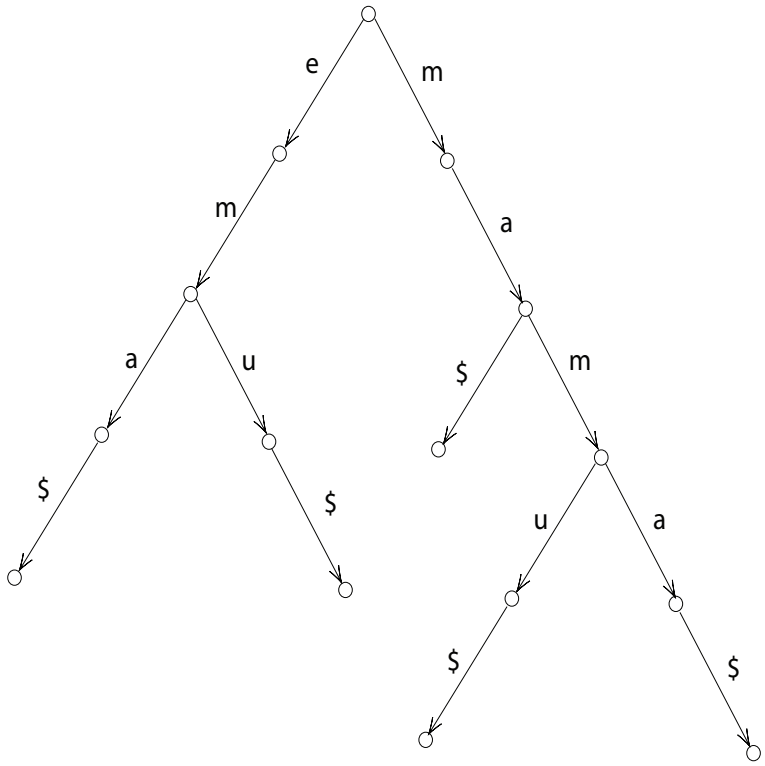
Zmazanie Q z existujúceho trie Nájdeme vrchol zodpovedajúci Q:

- Ak je list, tak mažeme cestu ku koreňu až kým nedojdeme po vrchol, ktorý je akceptačný (okrem seba samého) alebo rozvetvenie.
- Ak nie je list tak odznačíme, že je akceptačný.

Zložitosť je taktiež $O(m)$.

Vieme teda spraviť operácie search, insert a delete. Takže sa na trie môžeme pozerať ako na slovníkovú štruktúru. Keby sme mali napr. vyvážený vyhľadávací strom, tak by povedzme insert trval $O(m\log(z))$, zatiaľ čo v trie to je $O(m)$ (pokiaľ predpokladáme malú resp. konštantne veľkú abecedu).

Aby sme trochu zjednodušili algoritmy pracujúce s trie, pridáme za každé slovo v množine špeciálny symbol \$, ktorý sa nenachádza v Σ. Budujeme teda trie pre reťazce $S_1\$, \ldots, S_z\$$, ako je naznačené na obrázku 23. Po takejto úprave každému slovu z množiny zodpovedá jeden list v strome.



Obr. 23: Trie pre $\{ema\$, ma\$, mamu\$, mama\$, emu\$\}$.

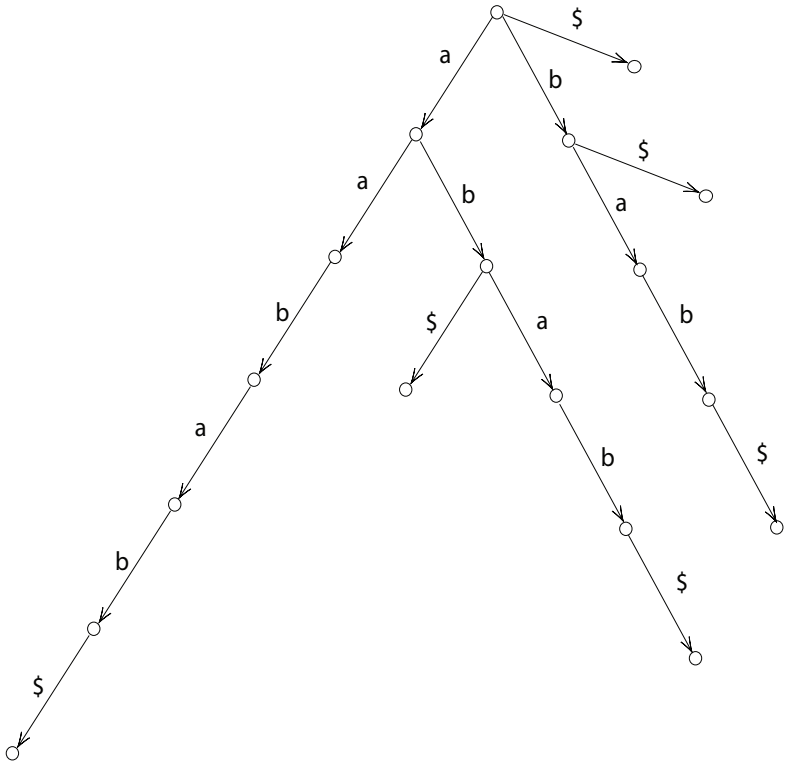
13.2 Príklady použitia lexikografického stromu

Trie môžeme použiť na riešenie rôznych úloh s množinami slov, ako napríklad:

- Spellchecking: zostavíme trie pre slová v slovníku, hľadáme slová v kontrolovanom texte.

- Rátanie frekvencií výskytu jednotlivých slov v texte - ku každému listu pridáme počítadlo; zložitosť je $O(n)$.
- Hľadáme najdlhšie slovo, ktoré je prefixom aspoň dvoch reťazcov z $S_1, ..., S_z$. Nájdeme najhlbší vrchol, ktorý má pod sebou aspoň dva listy.
- Pokiaľ by sme chceli hľadať najdlhšie slovo, ktoré je podslovom aspoň dvoch reťazcov z $S_1, ..., S_z$ tak na to sa trie príliš nehodí, pretože vrcholy zodpovedajú prefixom. Skúsime to vyriešiť rozšírením trie a to tak, že vybudujeme trie všetkých sufixov. Tým pádom vrcholy budú zodpovedať prefixom sufixov slov z $S_1, ..., S_z$ a to sú práve podslová. A na takto skonštruovanom trie hľadáme najhlbší vrchol, ktorý má pod sebou aspoň dva listy, ktoré sú z dvoch rôznych slov (tj. nie sufixy toho istého slova). Problém ale je, že veľkosť takejto trie a tým pádom aj zložitosť jej konštrukcie je $O(\sum_{i=1}^{z} |S_i|^2)$, pretože súčet dĺžok sufixov slova dĺžky m je $\Theta(m^2)$. Aby sme sa vyhli takejto kvadratickej zložitosti, budeme namiesto lexikografických stromov používať sufixové stromy.

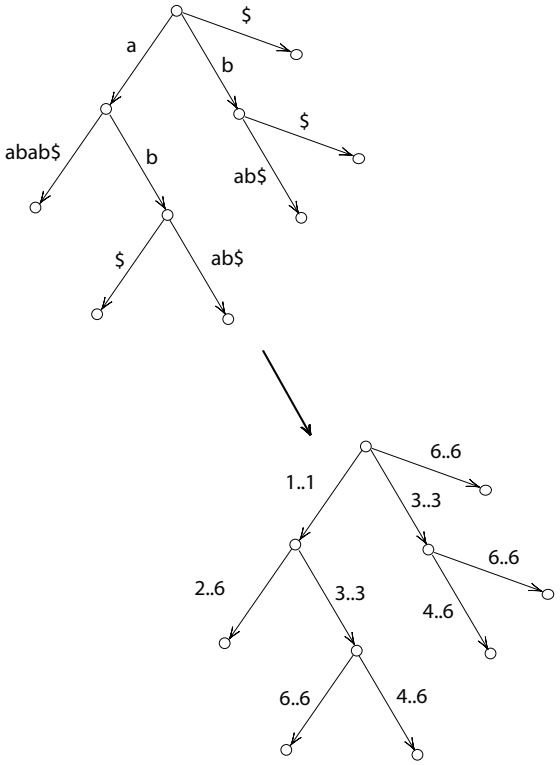
14 Sufixové stromy



Obr. 24: Lexikografický strom so všetkými sufixami *aabab*$

Sufixový strom dostaneme z lexikografického stromu tak, že nahradíme každú postupnosť k vrcholov bez vetvenia jednou hranou, ktorú označíme príslušným reťazcom dĺžky k. Napr. pre $S = aabab$ dostávame lexikografický strom sufixov ako na obrázku 24 a sufixový strom ako na obrázku 25 hore. Aby sme však naozaj ušetrili pamäť, nebudeme si pamätať samotné reťazce dĺžky k, ale len súradnice tohto reťazca v slove S (obr. 25, strom dole).

Teda v sufixovom strome sú hrany označené reťazcami. Reťazce pre hrany z jedného vrchola začínajú rôznymi písmenami. Každý list zodpovedá jednému sufixu S\$ a naopak. Každému vrcholu zodpovedá podslovo S\$, ale nie nutne naopak (kvôli skompaktneniu).

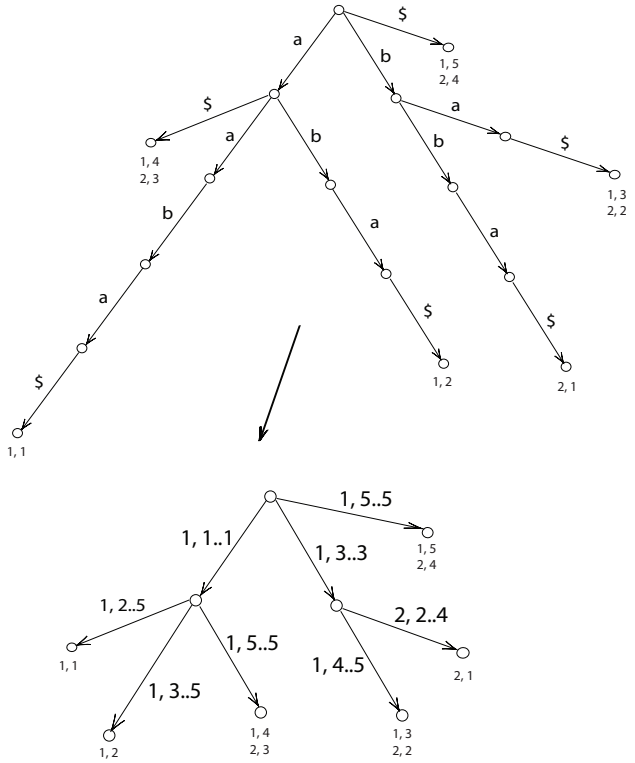


Obr. 25: Skompaktnený sufixový strom pre *aabab*\$ a pod ním daný strom so súradnicami namiesto reťazcov na hranách.

Koľko má vrcholov sufixový strom? Nech $|S| = m$. Potom sufixový strom má $m+1$ listov, jeden pre každý sufix. Potom m vnútorných vrcholov je najviac m, lebo každý má aspoň dve deti. Spolu máme teda $O(m)$ vrcholov. V každom vrchole si pamätáme smerníky do detí (prípadne na rodiča), teda $O(1)$ smerníkov (keď predpokladáme konštantnú abecedu), 2 indexy do S a v listoch ešte číslo sufixu (listy číslujeme od najdlhšieho sufixu ktorý reprezentujú, napr. pre strom na obr. 4 je list pri hrane *abab*\$ označený 1, list do ktorého ide hrana \$ z koreňa označený 6, atď.). Spolu teda $O(\log(m))$ bitov v každom vrchole respektíve $O(1)$ registrov. Pre celý strom to tvorí $O(m\log(m))$ bitov či $O(m)$ registrov. Čas na zostrojenie sufixového stromu je pri triviálnej implementácií až $O(m^2)$ ale pri použití pár „trikov" sa dostaneme k $O(m)$, pričom takýto algoritmus je opísaný v poznámkach nasledujúcej prednášky.

14.1 Zovšeobecnené sufixové stromy pre reťazce $S_1, S_2, \ldots, S_z$

Zovšeobecnený sufixový strom vytvoríme tak, že všetky reťazce ukončíme znakom \$, teda dostanem $S_1\$, S_2\$, \ldots, S_z\$$ (obr. 5). Potrebujeme si pamätať v každej hrane okrem indexov podslova aj z ktorého reťazca daný sufix je a v každom liste aj zoznam príslušných reťazcov (keďže môže byť daný sufix pre viac reťazcov).



Obr. 26: Zovšeobecnený sufixový strom pre $aaba\$, bba\$$ a pod ním skompaktnený zovšeobecnený sufixový strom s indexami pre tie isté vstupy

14.2 Príklady použitia sufixových stromov

Znovu ich je viacero, napríklad online vyhľadávanie vzoriek v statickom texte. Máme text T a postupne prichádzajú vzorky $P_1, P_2, \ldots$. Úlohou je nájsť pre každú vzorku výskyty. Doteraz sme to vedeli len cez KMP algoritmus v $(|P_i| + |T|)$ pre každú vzorku P_i. Pomocou sufixových stromov to zvládneme v $O(|P_i| + k)$, kde k je počet výskytov (po $O(|T|)$ predspracovaní na vybudovanie stromu). Pre vzorku P_i sledujeme cestu z koreňa, až kým:

- neminieme P_i a nedá sa pokračovať (môže sa stať aj uprostred hrany) $\Rightarrow$ nemá výskyt
- minieme P_i a skončíme vo vrchole v. Potom potrebujeme vypísať indexy sufixov všetkých listov v podstrome v. Zložitosť prezretia podstromu je $O(k)$, kde k je počet výskytov.
- minieme vzorku a skončíme uprostred hrany. Postupuje rovnako ako v predchádzajúcom prípade pre spodný vrchol hrany na ktorej sme skončili.

Iný príklad použitia je najdlhšie podslovo, ktoré sa v T vyskytuje aspoň dvakrát. Tento zodpovedá nejakému vrcholu (prečo?). Zaujíma nás reťazcová dĺžka vrchola, čo je dĺžka zodpovedajúceho reťazca pre daný vrchol. Hľadáme vnútorný vrchol s maximálnou reťazcovou hĺbkou.

Vráťme sa k nášmu motivačnému príkladu, t.j. nájsť najdlhšie slovo, ktoré je podslovom aspoň dvoch reťazcov z $S_1, \ldots, S_z$. Zostrojíme si zovšeobecnený sufixový strom pre tieto reťazce

a hľadáme najhlbší vrchol, ktorý má pod sebou listy zodpovedajúce sufixom aspoň dvoch rôznych S_i.

Ako ho nájdeme? Prehľadávame strom od listov a pre každý vrchol si pamätáme, či má pod sebou sufixy jedného slova alebo viacerých. Ak iba jedného, pamätáme si aj ktorého, aby sme v rodičovi vedeli tiež rýchlo vyhodnotiť túto podmienku. Celkovo vieme príklad riešiť v lineárnom čase pre konštantnú abecedu.

14.3 Hľadanie maximálnych opakovaní

Už sme videli ako nájsť najdlhšie podslovo s výskytom v dvoch rôznych textoch. Niečo podobné by sa nám hodilo pri detekcii plagiátorstva, avšak potrebovali by sme hľadať skôr približné než presné zhody. Ako uvidíme neskôr, hľadanie približných zhôd je výpočtovo náročnejšie, preto pre zrýchlenie môžeme chcieť nájsť aspoň viacero fragmetov presných zhôd. Nasledujúci problém takýto cieľ presnejšie formalizuje.

Maximálny pár v S je dvojica podslov $S[i..i+k]$ a $S[j..j+k]$ taká, že $S[i..i+k] = S[j..j+k]$ a $S[i-1] \neq S[j-1]$ a $S[i+k+1] \neq S[j+k+i]$. Inými slovami, pred a za rovnakými podslovami sú rozdielne znaky a teda sa rovnaká časť už nedá predĺžiť. Pre jednoduchosť si S predstavíme (resp. upravíme) na #$S$$, aby sa nám nestalo ze vypadneme z reťazca.

Príklad Nech $S = xabxabyabz$. Maximálny pár je *xab* na pozíciách 1 a 4, *ab* na pozíciách 2 a 8 a taktiež *ab* na pozíciách 5 a 8.

Maximálne opakovanie je reťazec, ktorý je časťou maximálneho páru. Pre predchádzajúcu ukážku to je *ab* a *xab*.

Chceme teraz pre reťazec S nájsť všetky maximálne opakovania. Zostavime sufixový strom pre S. Každé max. opakovanie zodpovedá nejakému vnútornému vrcholu. Tým pádom je najviac n max. opakovaní. Nech v je list, ktorý zodpovedá sufixu $S[i..n]$. Zavedieme si, že ľavý znak listu v bude $l(v) = S[i-1]$. Ďalej povieme, že vrchol stromu u je rôznorodý, ak v jeho podstrome sú aspoň dva rôzne $l(v)$. Potom tvrdíme, že vnútorný vrchol zodpovedá maximálnemu opakovaniu práve vtedy, keď je rôznorodý. Jedna implikácia je zjavná, pretože ak vnútorný vrchol zodpovedá max. opakovaniu, tak z definície sa vo vstupnom slove vyskytuje daný reťazec aspoň dvakrát a znaky naľavo od výskytu sa odlišujú. Opačnú implikáciu si ukážeme:

Majme reťazec α zodpovedajúci rôznorodému vnútornému vrcholu. Potom existujú dva znaky x a y také, že $x\alpha$, $y\alpha$ sú podslová S. Čo nasleduje za α v týchto výskytoch? Máme dve možnosti:

- rôzne $x\alpha p$ a $y\alpha r$ - a teda je to maximálny pár
- rovnaké $x\alpha p$ a $y\alpha p$. Z vrcholu α však ide aj ďalšia hrana začínajúca na nejaké $q \neq p$. Čo ide pred αq? Znova máme dve možnosti:
  - vždy x ($x\alpha q$) a teda mám $x\alpha q$ a $y\alpha p$
  - alebo $z \neq x$ ($z\alpha q$) a z toho dostávam $z\alpha q$ a $x\alpha p$

Ako hľadať rôznorodé vrcholy? Prehľadávame strom od listov v post-orderi. V každom vrchole si pamätáme buď jediné $l(v)$ ak nie je rôznorodý, alebo špeciálnu značku ak je. Toto sa ľahko spočíta pre daný vrchol prezretím všetkých detí.

Takže celkovo vieme vypísať všetky maximálne opakovania v $O(n)$ čase. Algoritmus sa dá ďalej rozšíriť na detekciu opakovaní, ktoré nie sú podreťazcami iných, resp. na výpis maximálnych dvojíc.

14.4 Zhrnutie

- Sufixové stromy sú kompaktná reprezentácia všetkých sufixov reťazca
- Dajú sa zostrojiť v lineárnom čase, ako uvidíme na budúcej prednáške
- Potrebujú však pomerne dosť veľa pamäte na každý znak reťazca, čo je problém pre dlhé reťazce
- Pomocou nich vieme v lineárnom čase odpovedať na mnohé zaujímavé problémy o rovnosti podslov

15 Ukkonenov algoritmus na konštrukciu sufixových stromov

Algoritmov na konštruovanie sufixových stromov je niekoľko, napríklad:

- Weiner 1973 (ktorý Knuth nazval „algoritmom roku ’73”)
- McCreigh 1976
- Ukkonen 1995

Bližšie si popíšeme Ukkonenov algoritmus. Najprv budeme vytvárať implicitný sufixový strom, bez koncového $ (v takomto strome niektoré sufixy môžu končiť uprostred hrany). Nech τ_i je implicitný sufixový strom pre $S[1..i]$ (prvých i písmen reťazca S). Budeme postupne vytvárať τ_i pre dlhšie a dlhšie prefixy a používame implicitný strom namiesto sufixového stromu, lebo by nebolo efektívne po každom kroku (tj. zväčšení i) pridávať a uberať $.

Samotný algoritmus vyzerá nasledovne (pozn.: `T(i)` $= \tau_i$):

```
zostav T(1)
for i = 2 to |S| + 1
  // nazveme to "fáza i" - prerob T(i-1) na T(i)
  for j = 1 to i
    // nazveme to "predĺženie j" - pridaj S[j..i] do stromu
    nájdi koniec cesty zodpovedajúcej S[j..i-1];
    pridaj k tejto ceste S[i] (*)
```

V kroku (*) môžu nastať 4 prípady:

- prípad **1** - cesta končí v liste, do ktorého ide hrana označená $\alpha \Rightarrow$ pridaj $S[i]$ k hrane do listu, tj dostanem hranu $\alpha S[i]$

- prípad **2a** - skončíme vo vnútornom vrchole a nedá sa pokračovať $S[i] \Rightarrow$ pridáme dieťa k vrcholu s hranou $S[i]$ a index listu bude j
- prípad **2b** - skončíme uprostred hrany a nedá sa ďalej pokračovať $S[i] \Rightarrow$ máme teda hranu $\alpha\beta$ a za α nemôžeme pokračovať, tak rozsekneme hranu na dve časti, kde prvá časť ostane hrana α za ktorou nasleduje vrchol s dvoma deťmi - jeden s hranou β a druhé dieťa s indexom j, kde hrana je označená $S[i]$
- prípad **3** - cesta pre $S[i..j]$ už je v strome $\Rightarrow$ netreba robiť nič

Takáto triviálna implementácia algoritmu má časovú zložitosť $O(n^3)$, čo nie je príliš dobré. Zlepšíme to trikmi medzi ktoré patrí odbúravanie pochodovania po cestách a (rozumným) preskočením väčšiny prípadov 1 až 3.

15.1 Trik 1 - sufixové linky

Ak v je vrchol pre reťazec $x\alpha$ ($x \in \Sigma$, α je reťazec), sufixová linka $S(v)$ je vrchol pre reťazec α. Udržujeme si $S(v)$ pre všetky vnútorné vrcholy.

Vytvorenie linky pre novo-vzniknutý vnútorný vrchol: Nové vnútorné vrcholy vznikajú iba v prípade 2b. Nech $S[j..i] = x\alpha y$ $(x,y \in \Sigma)$. Pridali sme vrchol pre reťazec $x\alpha$. α je už v strome. Vrchol pre $x\alpha$ má aspoň dve deti začínajúce na y,z. Vo fáze $i-1$ strom obsahoval $x\alpha z$ a teda aj αz (keďže obsahuje každý sufix). Po rozšírení $j+1$ fázy i určite bude existovať vnútorný vrchol pre α. V tom kroku tiež navštívime vrchol pre α a dorobíme linku z $x\alpha$.

Otázka znie, ako nám takéto sufixové linky pomôžu? V rozšírení j sme našli pozíciu $S[j..i]$. V ďalšom kroku hľadáme $S[j+1..i]$. Nájdeme najbližšieho predka $S[j..i]$, ktorý má $S(v)$ (nech to je $S[j..k]$) a cez jeho sufixovú linku sa dostaneme do $S[j+1..k]$ a zídeme dole do $S[j+1..i]$). Sufixová linka je najneskôr v praotcovi (príp. v koreni, pre ktorý sufixová linka nie je definovaná, ale to nám nevadí), pretože v strome je najviac jeden vnútorný vrchol (okrem koreňa), ktorý ešte nemá sufixovú linku a môže to byť práve otec novovytvoreného listu.

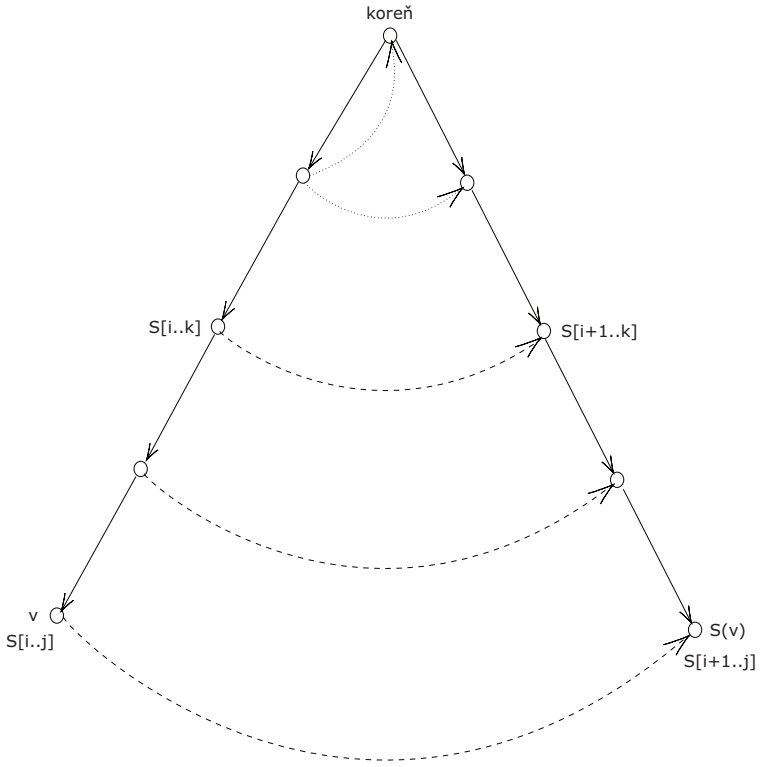
Lema: Ak hĺbka vrchola v je h, tak hĺbka $S(v)$ je aspoň $h-1$.

Pozn.: V leme je $h-1$, pretože syn koreňa nemusí mať vždy sufixovú linku na „vedľajší" vrchol, ale aj do koreňa.

Začneme v hĺbke 0. Dokopy vo fáze i stúpame najviac o $3n$ (najviac dve stúpania do praotca a jedno cez linku (z lemy)). Na konci fázy skončíme v hĺbke 1. Hĺbku teda zvýšime najviac $3n+1$-krát. Spolu teda prejdeme po $O(n)$ hranách resp. linkách.

15.2 Trik 2 - zrátaj a preskoč

Sme v stave, že sme dokončili spracovanie $x\alpha y$ a chceme spracovať αy a na to potrebujeme nájsť α. Vieme, že α je už niekde v strome. Trik je, že na každej hrane nám stačí skontrolovať len prvý znak a zvyšok už musí sedieť, pretože z každého vrchola musia hrany začínať iným znakom. Na každej hrane sa teda zdržím $O(1)$. Spolu teda jedna fáza trvá $O(n)$ a celý algoritmus tým pádom $O(n^2)$. Stále to však nie je to, čo chceme.



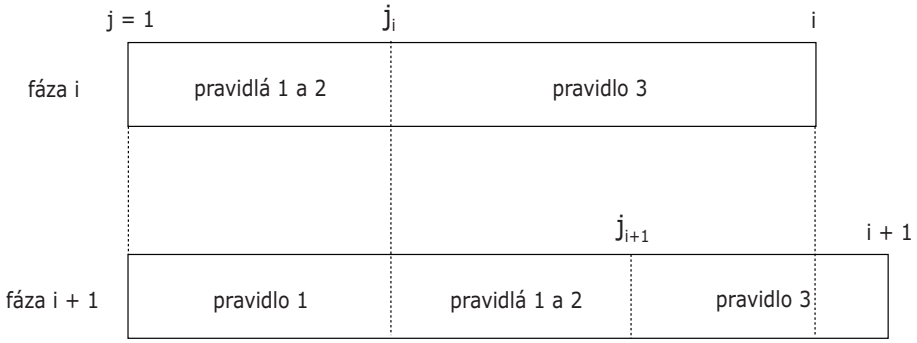
Obr. 27: Strom so sufixovými linkami. Prerušované čiary značia sufixové linky a bodkovaná predstavuje obe možnosti sufixovej linky.

15.3 Trik 3 - tri a dosť

Tretí trik znamená uvedomiť si, že ak sme v rozšírený fázy i a použijeme pravidlo **3**, tak vo zvyšku fázy i už budeme používať už len pravidlo **3**. Inými slovami, ak $S[j..i]$ už je v strome, tak aj $S[k..i]$ (pre $k > j$, teda všetky ďalšie sufixy) sú tiež v strome. Takže keď v algoritme použijeme pravidlo **3**, tak skončíme fázu.

15.4 Trik 4 - list zostane listom

Ak pre $S[j..i]$ vyrobíme list (to môžeme pravidlom **1**, **2a** alebo **2b**), tak pre $S[j..i+1]$ použijeme pravidlo **1** na tento list. Nech j_i je prvýkrát, keď vo fáze i použijeme pravidlo **3**.



Obr. 28: Pravidlá používané počas fáz i a $i+1$ s využitím trikov 3 a 4

V hranách do listov použijeme špeciálny index na koniec reťazca (napríklad nekonečno) reprezentujúci aktuálne i. Následne, vo fáze $i+1$ môžeme preskočiť prvých $j_i - 1$ rozšírení (obrázok 3). Preskočím len prvých $j_i - 1$, pretože j_i už je pravidlo **3**.

Po aplikovaní všetkých trikov vyzerá fáza i nasledovne:

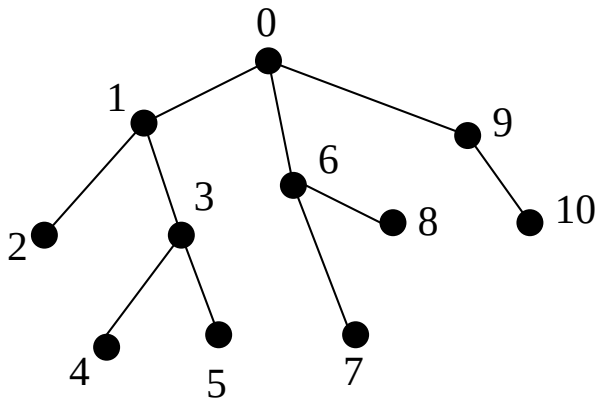
```
j = j_(i-1);
while (j <= i) {
  sprav rozšírenie j (podľa pôvodných pravidiel);
  ak sa použilo pravidlo 3 {
    j_i = j;
    break;
  }
}
```

Počet rozšírení je najviac $2n$ v celom algoritme. Čas na jedno rozšírenie je $O(1)$ + pochodovanie dole po hranách. Toto zlepšíme tak, že si vo fáze i zapamätáme posledný vrchol a tento bude prvý vo fáze $i+1$, takže v prvom kroku netreba pochodovať.

Koľko je teda možných pochodovaní? Spravíme najviac $2n$ rozšírení, pričom v každom stúpneme najviac o 3. Začíname v hĺbke 0 a po skončení sme najviac v $n+1$. Spolu teda najviac $7n+1$-krát klesneme ($2n*3+n+1$) a teda $O(n)$.

16 Najnižší spoločný predok

Budeme sa zaoberať hľadaním najnižšieho spoločného predka (lowest common ancestor) dvoch vrcholov v strome. Pod označením $z = lca(u,v)$ budeme rozumieť, že z je najnižším spoločným predkom vrcholov u a v.



Obr. 29: Strom v preorder očíslovaní

Na strome na obrázku 29 vidieť nasledujúce príklady *lca*.

$lca(2,4) = 1$
$lca(2,6) = 0$

$lca(9,10) = 9$

Triviálnym algoritmom vieme nájsť *lca* pre ľubovoľné dva vrcholy v čase $O(n)$. Stačí sledovať cestu od vrcholov do koreňa a prvý vrchol, ktorý je na oboch cestách spoločný bude *lca*.

Budeme sa zaoberať algoritmom, pomocou ktorého nájdeme *lca* pre ľubovoľné dva vrcholy v čase $O(1)$, pričom na predspracovanie budeme potrebovať čas $O(n)$. Takýto algoritmus vymysleli Harel a Tarjan 1984, neskôr zjednodušili Schieber a Vishkin 1988 a my si ukážeme ešte jednoduchšiu verziu podľa článku Bender a Farach-Coulton (2000).

Prehľadajme strom do hĺbky. Všimnime si prvý moment kedy objavíme vrchol u a prvý moment kedy objavíme vrchol v. Najbližšieho spoločného predka u,v spoznáme jednoducho. Je to najvyššie položený vrchol spomedzi vrcholov, ktoré sme prechádzali medzi objavením u a v.

Pomocou tejto myšlienky prevedieme úlohu hľadania najbližšieho spoločného predka na inú na prvý pohľad ľahšiu úlohu.

Prehľadajme náš strom do hĺbky. V každom kroku prehľadávania si zapamätáme hĺbku aktuálneho vrchola $D(t)$ a číslo aktuálneho vrchola $V(t)$. Navyše pre každý vrchol si zapamätáme kedy sme ho prvý krát objavili (pole R).

Nájdenie $lca(u,v)$ bude vyzerať nasledovne. Zistíme kedy boli objavené u,v, označme tieto časy t_u,t_v (bunv. nech $t_u < t_v$). Teraz nájdeme také t z intervalu $< t_u,t_v >$ aby $D(t)$ bolo čo najmenšie. Potom $lca(u,v) = V(t)$. Všimnime si, že polia D,V majú dĺžku $2n-1$, čo je stále $O(N)$.

Takto sme previedli úlohu hľadanie najbližšieho spoločného predka na úlohu rýchleho hľadania minima v úseku poľa (range minimum query - RMQ).

Napríklad pre strom na obrázku 29 dostaneme polia:

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V:	0	1	2	1	3	4	3	5	3	1	0	6	7	6	8	6	0	9	10	9	0
D:	0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0	1	2	1	0
R:	0	1	2	4	5	7	11	12	14	17	18										

Odpoveď v čase $O(1)$, predspracovanie v čase $O(N \lg N)$: Začnime najprv algoritmom, ktorý nám hľadanie minima v úseku poľa umožní v konštantnom čase, ale ešte nebude mať optimálny čas predspracovania a použitú pamäť.

Majme poľe $A[1..N]$ v ktorom chceme rýchlo vedieť odpovedať na otázky typu: Aké je minimum z čísel $A[i],A[i+1],\ldots,A[j-1]$?

Pre každý prvok poľa A a všetky zmysluplné k si zapamätáme nasledovné minimá:

$$B[i][k] = min(A[i],A[i+1],\ldots A[i+2^k-1])$$

Týchto hodnôt je $O(N \lg N)$ a spočítať ich vieme v rovnakom čase (treba počítať najprv pre $k = 0$, z nich potom pre $k = 1$, atď).

Odpoveď na otázku: $min(A[i],\ldots,A[j-1])$, potom vieme vypočítať nasledovne:

1. Nájdeme najväčšie k také, že $2^k \leq j-i$.
2. A potom platí:

$$min(A[i],\ldots,A[j-1]) = min(min(A[i],\ldots,A[i+2^k-1]),min(A[j-2^k],\ldots,A[j-1]))$$

$$min(A[i],\ldots,A[j-1]) = min(B[i][k], B[j-2^k][k])$$

Druhý krok vieme zjavne spraviť v konštantom čase. Na to, aby sme v takomto čase vedeli spraviť aj prvý krok si potrebujeme príslušné hodnoty k predpočítať (zaberie nám to $O(N)$ pamäte a rovnako veľa času na predpočítanie).

Vylepšenie na predspracovanie v $O(N)$ Nasekajme naše pôvodné pole na úseky dĺžky M. Potom pri odpovedi na otázku o minime v intervale $< i, j)$ môžu nastať dva prípady: buď celý interval leží v tom istom úseku, alebo vieme hľadaný interval rozdeliť na koniec úseku, kde leží i, niekoľko celých úsekov a začiatok úseku, kde leží j.

Úplne prvou vecou, čo môžeme spraviť je pre každý úsek spočítať minimum a tieto minimá uložiť do nového poľa A'. Toto vieme spraviť v čase $O(N)$ a nové pole bude mať dĺžku $O(N/M)$. V tomto poli budeme chcieť opäť hľadať minimum, použijeme na to predspracovanie popisané v predchádzajúcej časti. Toto predspracovanie bude mať zložitosť $O(N/Mlg(N/M))$. Ak $M = O(\lg N)$, tak toto predspracovanie má zložitosť lineárnu od N.

Ešte ale treba vyriešiť hľadanie minima v úsekoch. Tu si všimnime, že naše pole D v ktorom hľadáme minimá má jednu špeciálnu vlastnosť: jeho susedné hodnoty sa líšia o $+1$ alebo -1.

Zapíšme si každý úsek ako postupnosť $+,-$ (prvý znak môžeme odignorovať) podľa toho ako sa daný prvok líši od prechádzajúceho. Takto dostaneme niekoľko typov úsekov (ak bude M dostatočne malé, tak typov úsekov bude menej ako počet všetkých úsekov). Pre každý typ a každú dvojicu začiatok, koniec predpočítame pozíciu minima (tá je jednoznačne určená len z $+,-$).

Ako vhodná hodnota M sa ukazuje $M = \frac{1}{2}\lg N$. V tomto prípade máme $2^{D-1} = \sqrt{N}/2$ rôznych úsekov. V každom z nich potrebujeme nájsť odpoveď na $M^2 = \frac{1}{4}\lg^2 N$ otázok. To celé je teda rádovo $O(\sqrt{N}\lg^2 N)$ hodnôt a v rovnakom čase vieme aj tieto hodnoty spočítať.

Celý algoritmus sa dá teda zhrnúť nasledovne. Predpočítanie:

1. Prehľadaj strom do hĺbky a popritom generuj polia V, D, R.
2. Nasekaj pole D na úseky dĺžky $M = \frac{1}{2}\lg N$.
3. Pre každý úsek zisti jeho minimum to ulož do poľa D'. Nad poľom D predpočítať pole popísané v predoslej časti.
4. Pre každý úsek zisti jeho typ a pre každý typ predpočítaj príslušné odpovede.

Odpoveď na otázku $lca(u,v)$:

1. Nájdi v poli R príslušné hodnoty pre u, v: t_u, t_v.
2. Zisti v ktorých úsekoch sú hodnoty t_u, t_v.
3. Zisti príslušné minimum v prvom a poslednom úseku. Pomocou štruktúry z predchádzajúcej časti zisti minimum z úsekov medzi nimi.
4. Z týchto troch miním vyber najmenšie: t.
5. Odpoveď je $V(t)$.

17 Využitie najnižšieho spoločného predka na vyhľadávanie v texte

Na minulej prednáške sme ukázali, že hodnoty $LCA(u,v)$ vieme počítať v konštantnom čase s predspracovaním $\mathcal{O}(n)$. V sufixových stromoch, kde máme listy pre sufixy $S[i..n]$ a $S[j..n]$, je hodnotou $LCA(i,j)$ vrchol pre najdlhší spoločný prefix $S[i..n]$ a $S[j..n]$.

17.1 Hľadanie palindrómov

Lca sa dá využiť aj pri hľadaní palindrómov. Palindróm je reťazec, ktorý sa odpredu číta rovnako ako odzadu, teda $S = S^R$. Maximálny palindróm v S je podslovo $S[i..j]$, ktoré je palindróm a $S[i-1] \neq S[j+1]$, teda sa nedá rozšíriť.

Všetky maximálne palindrómy pre nejaký sufixový strom vieme nájsť v čase $O(n)$. Vytvoríme zovšeobecnený sufixový strom, kam vložíme S aj S_R. Budeme testovať všetky možné stredy palindrómu. Zvlášť budeme robiť palindrómy párnej a nepárnej dĺžky.

Nech má palindróm nepárnu dĺžku, teda $2j+1$. Nech je stred na pozícii i. Potom musí platiť: S[i+1 ... i+j] = S^R[n-i+2 ... n - i + j + 1]. Chceme nájsť najdlhší spoločný prefix S[i+1 ... n] a S^R[n - i + 2 ... n]. Stačí zavolať $lca(S_{i+1}, S^R_{n-i+2})$. Index j je dĺžka reťazca zodpovedajúceho vrcholu v. Podobne budeme postupovať aj pre párne dĺžky palindrómu.

17.2 Hľadanie približných výskytov P v T v Hammingovej vzdialenosti

Ďalej môžeme lca využiť aj v súvislosti s Hammingovou vzdialenosťou. Hammingova vzdialenosť $d_H(S_1,S_2)$ medzi reťazcami S_1 a S_2 rovnakej dĺžky je počet pozícií, takých že platí $S_1[i] \neq S_2[i]$.

Ukážeme ako pre dané T a P nájsť všetky i, pre ktoré platí $d_H(T[i..i+m-1],P) \leq k$, t.j. namiesto presných výskytov P v T hľadáme približné výskyty.

Prístup 1 Zostrojíme sufixový strom pre P a T a potom pre každú pozíciu v texte zisťujeme, či sa tam dá nájsť približný výskyt.

```
for(i = 1; i <= n - m + 1; i ++){
   j = 1;
   err = 0;
   while(j <= m){
      q = dĺžka max. spoločného prefixu T[i + j - 1 .. n] a P[j .. m]
      // nájdeme pomocou lca
      // P[j .. j + q - 1] = T[i + j - 1 .. i + j + q - 2]
      if (j + q  <= m){
         err ++;
         if (err > k){
            break;
         }
      }
      j = j + q + 1   //presunieme sa za chybu
   }
```

```
   if (err <= k){
     print i;
   }
}
```

Pretože chýb bude nanajvýš k, celková zložitosť tohto algoritmu bude $O(nk)$.

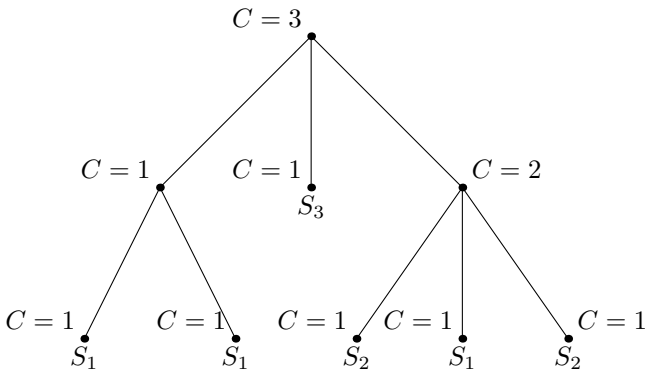
Prístup 2 Tento prístup nemá nič spoločné s LCA. Hodí sa, keď máme veľa vzoriek nad malou abecedou. Zostavíme si sufixový strom len pre T. Potom budeme skúmať všetky reťazce P' také, že $d_H(P,P') \leq k$. Budeme zisťovať, či P' je podslovo T (sledujeme cestu v strome). Prvý krok beží v čase $\mathcal{O}(n)$, posledný v čase $\mathcal{O}(m)$ pre každé P'. Počet rôznych P' vyrátame nasledovne: máme najviac $i \leq k$ chýb, musíme uvažovať i pozícií chýb a na každej nový znak, čo je

$$\sum_{i=0}^{k} \binom{m}{i} (\sigma - 1)^i,$$

čo je najviac $\mathcal{O}((m\sigma)^k)$. Takže celková zložitosť je $O(n + m^{k+1}\sigma^k)$. To je výhodné, ak máme veľa vzoriek a malú abecedu. Využíva sa to napríklad aj v bioinformatike, keď hľadáme (napr. v DNA) kúsok, na ktorý sa napr. viaže molekula rozpoznávajúca určitý motív (vzorku), v ktorej sa môže vyskytovať chyba, tj. nechceme presný výskyt. Napríklad pre ľudský genóm môžeme mať n napr. rádovo 10^9, veľkosť abecedy je 4 a môžeme uvažovať veľa vzoriek, napr. dĺžky cca $m = 10$ s $k = 2$ chybami.

17.3 Najdlhší podreťazec viacerých reťazcov

Ďalšiou aplikáciou *LCA* je zovšeobecnený sufixový strom pre z reťazcov $\{S_1, \ldots, S_z\}$. Úlohou je vyrátať hodnotu $C(v)$ – počet rôznych reťazcov S_i s listom v podstrome s koreňom v. Ak by sme túto hodnotu vedeli zrátať, vedeli by sme sa pýtať, aké je najdlhšie podslovo, ktoré sa vyskytuje v každom reťazci. (Pozrieme všetky vrcholy a hľadáme najhlbší taký, ktorý má ešte hodnotu z.) Môžeme sa tiež pýtať, aké je najdlhšie podslovo, ktoré sa vyskytuje v aspoň k reťazcoch.



Obr. 30: Príklad stromu s hodnotami $C(v)$.

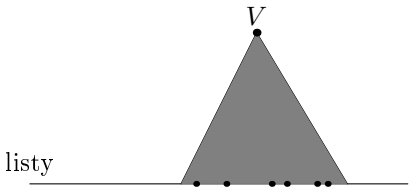
Jednoduchý algoritmus. Ukážeme si najprv jednoduchší spôsob – v každom vrchole si budeme pamätať zoznam S_i, potom hodnota $C(v)$ je dĺžka tohto zoznamu. Začneme od listov, ktorým priradíme im prislúchajúci reťazec. Problém je v tom, že časová zložitosť je $\mathcal{O}(nz)$, lebo otcovi rátame zoznam ako zjednotenie zoznamov jeho synov. Preto pre veľké z je tento algoritmus neefektívny.

Rýchlejší algoritmus Zostrojíme algoritmus s časovou zložitosťou $\mathcal{O}(n)$ pomocou funkcie *LCA*. Algoritmus najprv prejde strom do hĺbky a urobí zoznam listov v poradí, ako sa cez nich prechádzalo. (t.j. graficky zľava doprava) Rozhádžeme si tento zoznam na z zoznamov $L_1, \ldots, L_z$, pričom L_i obsahuje sufixy pre reťazec S_i.

Potom prejdeme cez každý zo zoznamov L_i a spočítame *LCA* pre každé dva prvky, ktoré idú za sebou v tomto zozname. Tieto zoznamy majú dokopy dĺžku n, *LCA* vieme rátať v konštatnom čase, takže zatiaľ je celý algoritmus lineárny. Navyše si pre každý vrchol počítame, koľkokrát bol ten vrchol *LCA*. (V premennej $h(v)$.) Nech $s(v)$ je súčet hodnôt $h(v)$ v podstrome s koreňom v. Spočítame ich zdola hore v lineárnom čase, lebo

$$s(v) = \sum_{w \in \text{child}(v)} s(w) + h(v).$$

Spočítame taktiež v premennej $l(v)$ počet listov v podstrome s koreňom v, čo môžeme tiež rátať zdola hore ako súčet pre všetky deti, pričom listy majú hodnotu 1. Potom tvrdíme, že $C(v) = l(v) - s(v)$ a výsledok dostaneme v $\mathcal{O}(n)$. Prečo je hodnota $C(v)$ takáto?



Obr. 31: Podstrom s koreňom v a listy, pre ktoré bude výsledok *LCA* v tomto podstrome.

Uvažujme vrchol v a reťazec S_i. Nech sa S_i v podstrome z koreňom v vyskytuje k-krát. (Nech k listov patrí k S_i.) Potom ak $k > 0$, tak chceme tento reťazec započítať jeden krát. V $l(v)$ je zarátaný k-krát a v $s(v)$ je zarátaný $k-1$ krát. (Keď sa pozrieme na tie listy, tvoria nejaký interval – sú pokope ako na obrázku 31 a pre každé dva vedľa seba zavoláme *LCA*. Hodnota $s(v)$ udáva počet, koľko krát výsledok *LCA* "spadol" do tohto podstromu, a to je pre každú dvojicu, ktorá je vo vnútri, tj $k-1$ dvojíc.) Preto je rozdiel $l(v) - s(v)$ rovný 1. Ak $k = 0$, tak aj $l(v) = 0$, aj $s(v) = 0$. (Každá dvojica je mimo podstromu, preto aj ich predok musí byť tiež mimo podstromu.)

17.4 TODO: Vypisovanie dokumentov

Preložit', spísať (S. Muthukrishnan, Efficient algorithms for document retrieval problems. SODA 2002: 657-666)

We have array A precomputed for RMQ.
For given i, j, x find all indices $k \in \{i, \ldots, j\}$ s.t. $A[k] \leq x$.

```
1 void small(i,j,x) {
2   if(j>i) return;
3   k = rmq(i,j);
4   if(a[k]<=x) {
5     print k;
6     small(i,k-1);
7     small(k+1,j);
8   }
9 }
```

cas vypoctu: O(p), kde p je pocet vypisanych indexov (kazdemu najdenemu k zapocitaj dve dcerske rekurzivne volania - celkovy pocet volani je najviac 2p+1)

Preprocess texts $\{S_1, \ldots, S_z\}$

Query: which documents contain pattern P?

We can do $O(m+k)$ where $k =$number of occurrences of P

Want $O(m+p)$ where $p =$number of documents containing P

Array of leaves L in DFS order

For leaf $L[i]$ let $A[i]$ be the index of previous leaf from the same S_j

Occurrences of P: subtree of corresponding to interval $[i, j]$ in L

Find all $k \in [i, \ldots, j]$ that have $A[k] < i$

Running time? Preprocessing?

18 Sufixové polia

Sufixové polia sú jednoduchšia štruktúra podobná sufixovým stromom. Problém pri sufixových stromoch je pamäťová náročnosť na ich uloženie. V počítači vieme uložiť n znakový reťazec nad binárnou abecedou v pamäti veľkosti $n/8\mathtt{B}$. Sufixový strom pre tento reťazec pri priamočiarej implementácii potrebuje pre každý z $2n-1$ vrcholov dva indexy do reťazca, pričom vo vnútorných vrcholoch navyše potrebujeme smerníky na dve deti a v listoch index začiatku sufixu, čo je $7n$ čísel alebo smerníkov. Na 32-bitovom počítači potrebujeme pamäť $28n\mathtt{B}$. Často si však ukladáme vo vrcholoch aj ďalšie informácie, napríklad smerník na rodiča, reťazcovú hĺbku, sufixovú linku a podobne.

Definícia 3. Sufixové pole reťazca S je pole všetkých sufixov reťazca v lexikografickom usporiadaní.

Príklad 5. V lexikografickom usporiadaní *ma*\$ < *mama*\$ < *mamu*\$. Znak \$ budeme pouvažovať za prvý v abecede. Sufixové pole pre $S =$ *mama* (označujeme SA) je pole $SA = (5,4,2,3,1)$. Tieto čísla zodpovedajú reťazcom \$, *a*\$, *ama*\$, *ma*\$, *mama*\$ a sú to indexy začiatku príslušného podreťazca.

Vidíme, že na uloženie sufixového poľa potrebujeme podstatne menej pamäte, ako na uloženie sufixového stromu.

18.1 Vyhľadávanie vzorky v sufixovom poli

Ak máme sufixové pole pre reťazec T, hľadáme výskyty vzorky P v T.

Algoritmus 1. Keďže je toto pole usporiadané, môžeme použiť binárne vyhľadávanie. Hľadáme teda úsek v poli od i po j, pre ktorý všetky sufixy $SA[i]\ldots SA[j]$ začínajú P.

```
1  search(X) {
2    //najdi max i take, ze T[SA[i]..n]<X
3    L = 0; R = n;
4    while(L < R){
5      k = (L + R + 1) / 2;
6      if(T[SA[k]..n]<X) { L = k; }
7      else { R = k - 1; }
8    }
9    return L;
10 }
```

Počet iterácií cyklu `while` v tomto algoritme je $\mathcal{O}(\log n)$. Po skončení vieme, že prvý výskyt môže byť na pozícii $i+1$. Ak teda chceme vedieť, či vzorka má nejaký výskyt, stačí porovnať P a $T[SA[i+1]..n]$. Celkový čas na vyhľadanie vzorky je počet iterácií krát $\mathcal{O}(m)$, lebo porovnanie reťazcov je v $\mathcal{O}(m)$.

Ak chceme poznať všetky výskyty alebo ich počet, môžeme spustiť binárne vyhľadávanie dvakrát: raz pre $X = P\$$ a raz pre $X = P\#$, kde # je špeciálny znak, v lexikografickom poradí za všetkými ostatnými znakmi abecedy. Ak prvé vyhľadávanie vrátilo index i a druhé index j, výskyty vzorky budú na pozíciách $SA[i+1],\ldots SA[j]$. Všetky výskyty teda nájdeme v čase $\mathcal{O}(m\log n+k)$, kde k je ich počet.

Algoritmus 2. Tento algoritmus vynecháva niektoré zbytočné porovnania. Nech $lcp(A,B)$ je dĺžka najdlhšieho spoločného prefixu reťazcov A a B (*longest common prefix*) (obr. 32).

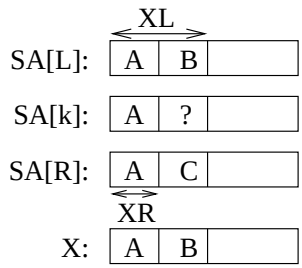
Budeme udržovať invariant:

$$XL = lcp(T[SA[L]..n], X)$$

$$XR = lcp(T[SA[R]..n], X)$$

```
1  L = 0; R = n;
2  XL = lcp(X, T[SA[L]..n]); XR = lcp(X, T[SA[R]..n]);
3  while(R - L > 1){
4    k = (L + R + 1) / 2;
5    h = min(XL, XR);
6    while(T[SA[k]+h]==X[h]) { h++; }
7    if(T[SA[k]+h]<X[h]){ L = k; XL = h; }
8    else { R = k; XR = h; }
9  }
10 // skoncime s intervalom dlzky 1 alebo 2
11 if (T[SA[R]..n] < X) {return R;}
12 else {return L;}
```

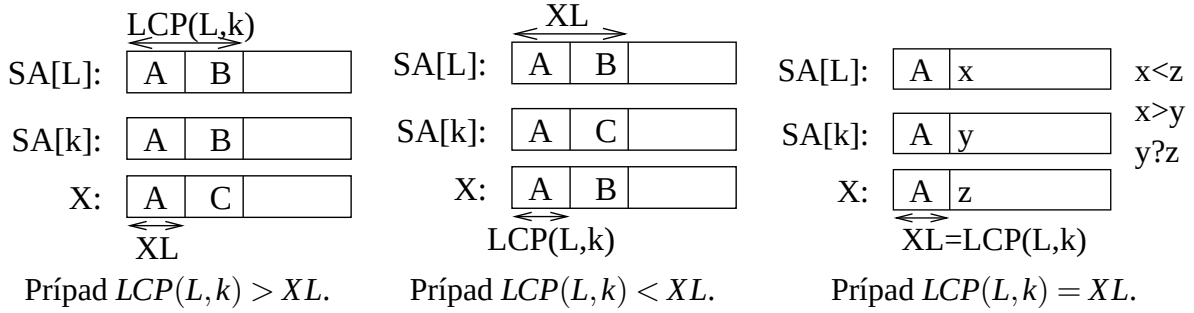
Tento algoritmus ale stále funguje v $\Theta(m\log n)$.



Obr. 32: Ilustrácia druhého algoritmu pre hľadanie vzorky v sufixovom poli, podprípad $XL > XR$.

Algoritmus 3. Tento algoritmus už má lepšiu zložitosť ako predchádzajúce dva. Podobá sa na algoritmus 2, ale v každej iterácii nechceme začať porovnávanie od pozície, ktorá je minimom XL a XR, ale od pozície, ktorá je ich maximom. Označme si ako $LCP(i, j)$ hodnotu $lcp(T[SA[i]..n], T[SA[j]..n])$. Predpokladajme, že vybrané hodnoty $LCP(i, j)$ poznáme (nezávisia od vzorky, len od T).

Predpokladajme, že v aktuálnej iterácii $XL \geq XR$. Prípad, že $XL < XR$ sa rieši symetricky podobným spôsobom. Môžu nastať nasledujúce tri prípady (pozr obr. 33:



Obr. 33: Tri prípady v treťom algoritme pre vyhľadávanie v sufixovom poli

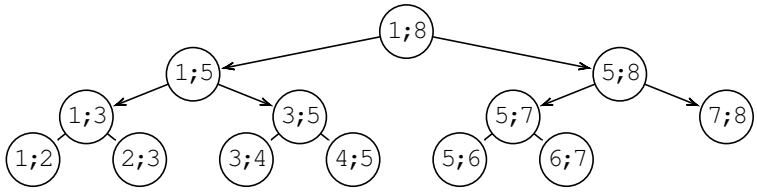
1. ak $LCP(L,k) > XL$, z tejto nerovnosti vieme, že $T[SA[k]..n] < X$ a teda $L = k$, XL zostáva také isté.

2. ak $LCP(L,k) < XL$, priamo z nerovnosti vieme, že $T[SA[k]..n] > X$, a teda $R = k$, $XR = LCP(L,k)$.

3. ak $LCP(L,k) = XL$, potom porovnávame X a $T[SA[k]..n]$ od $XL+1$.

Časová zložitosť Máme $O(\log n)$ iterácií, v každej $O(1)$ + porovnávanie znakov. $\max(XL, XR)$ neklesá. Nerovnosti znakov – $O(\log n)$. Rovnosť znakov posunie $\max(XL, XR)$ o 1. $\max(XL, XR) \leq m$, teda dokopy urobíme m porovnaní. Celková zložitosť je $O(m + \log n)$.

Keď robíme binárne vyhľadávanie, nepotrebujeme všetky lcp hodnoty. Konkrétne, v algoritme sme potrebovali $LCP(L,k)$ alebo $LCP(R,k)$, pričom k sa stalo novým L alebo R. Pozrime sa na rozhodovací strom na obrázku 34 – keďže sa pýtame len na intervaly z tohto stromu, a vieme, že strom má dokopy najviac $2n-1$ vrcholov, stačí nám spočítať a uložiť $2n-1$ hodnôt. Najprv spočítame hodnoty LCP pre listy (hodnoty $LCP(i, i+1)$).

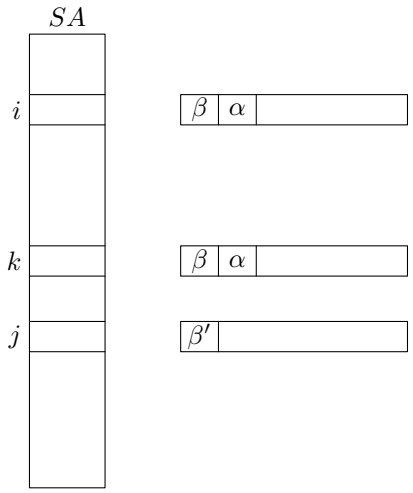


Obr. 34: Rozhodovací strom potrebných *lcp* hodnôt pri vyhľadávaní

Ak vieme spočítať $LCP(i,i+1)$, tak ostatné hodnoty *LCP* dopočítame ľahko, pretože pre $LCP(i,j)$ kde $j > i+1$ a pre každé $k \in \{i+1,\ldots,j-1\}$ platí

$$LCP(i,j) = \min\{LCP(i,k), LCP(k,j)\}.$$

Prečo? Môžeme to vidieť na obrázku 35. (Keby sa reťazce na pozíciách *i* a *j* zhodovali na viacerých znakoch ako reťazce na pozíciách *i* a *k*, tak potom by reťazce nemohli byť zoradené podľa abecedy, čo v sufixových poliach sú.) Z toho vyplýva, že ak vieme hodnoty pre listy, tak vieme lineárnym prechodom stromu spočítať pre všetky vrcholy. Takže potrebujeme už len spočítať *LCP* pre listy, čo sa dá v čase $O(n)$, ako ukážeme v časti 18.5.



Obr. 35: Odôvodnenie formuly na zrátanie $LCP(i,j)$

Zhrnutie V tabuľke 3 vidíme prehľad dátových štruktúr preberaných na prednáškach minulý týždeň. Vidíme, že sufixové pole je lepšie preto, lebo potrebujeme menej pamäte. S použitím *LCP* máme ešte nejaké urýchlenie – ak si predpočítame *LCP* pre niektoré dvojice, dosiahneme vyhľadávanie $\mathcal{O}(m+\log n)$ pri zachovaní pamäťovej náročnosti.

18.2 TO DO

Trochu učesať.

	konštrukcia	vyhľadávanie vzorky	pamäť (počet smerníkov)
sufixový strom	$\mathcal{O}(n\log\sigma)$	$\mathcal{O}(m\log\sigma)$	cca $7n$
sufixové pole	$\mathcal{O}(n)$	$\mathcal{O}(m\log n)$	n
sufixové pole + *LCP*	$\mathcal{O}(n)$	$\mathcal{O}(m+\log n)$	$3n$

Tabuľka 3: Porovnanie sufixových stromov, sufixových polí a sufixových polí s *LCP*.

18.3 Konštrukcia sufixového poľa

Klasickým triedením. Sufixové pole môžeme zostrojiť použitím klasického triedenia, napr. merge sortu. Ten urobí $\mathcal{O}(n\log n)$ porovnaní sufoxov, každé porovnanie trvá $\mathcal{O}(n)$, dokopy je to $\mathcal{O}(n^2\log n)$.

Triedením pomocou radix sortu. Trochu lepšie je použiť radix sort. Radix sort vo všeobecnosti triedi d ciferné čísla v k-árnej sústave, pričom každú cifru triedi pomocou counting sortu. Counting sort utriedi n čísel z množiny $\{0..k-1\}$ v čase $\mathcal{O}(n+k)$. Radix sort volá counting sort postupne d krát v poradí od najmenej významnej cifry po najvýzmanejšiu. Preto je celkový čas radix sortu $\mathcal{O}(d(n+k))$. V našom prípade predpokladajme, že abeceda je nejaká podmnožina množiny $\{0,..n-1\}$ a teda sufixy sú n-ciferné čísla v n-árnej sústave. Radix sort ich teda utriedi v čase $\mathcal{O}(n^2)$. Sufixové pole by sme ale chceli zostrojiť v lineárnom čase.

Pomocou sufixového stromu. Ďalšia možnosť je zostrojiť sufixový strom v lineárnom čase a potom ho prejsť do hĺbky tak, že v každom vrchole neprejdené hrany vyberáme podľa abecedy. Potom to poradie, v ktorom navštívime listy, je poradie, v akom majú byť sufixy v sufixovom poli. Takto vieme zostrojiť sufixové pole v lineárnom čase. Problém je v tom, že sme nechceli používať sufixové stromy kvôli veľkosti pamäte a takto ju aj tak budeme potrebovať.

Lineárny algoritmus na konštrukciu sufixového poľa. Existuje niekoľko lineárnych algoritmov na konštrukciu sufixového poľa, my si ukážeme jeden od autorov Karkkainen a Sanders (2003). Pracuje nad abecedou $\{1,\ldots,n\}$, kde n je dĺžka reťazca, pričom pracuje aj pre takúto veľkú abecedu v čase $O(n)$. Iné abecedy môžeme prečíslovať triedením znakov textu, čo však môže zhoršiť zložitosť. Za koniec reťazca pridáme niekoľko núl, ktoré budú simulovať špeciálny znak \$.

Prácu tohto algoritmu si ukážeme na príklade $S_p = fabbcabbd$. V tabuľke 4 môžeme vidieť prepísanie tohto slova do pracovnej abecedy.

i	0	1	2	3	4	5	6	7	8			
$S_p[i]$	f	a	b	b	c	a	b	b	d			
$S[i]$	5	1	2	2	3	1	2	2	4	0	…	0

Tabuľka 4: Prepísanie vstupu S_p do pracovnej abecedy $\{1\ldots n\}$

Krok 1: Utriedime sufixy $S[3i+k..n]$ pre $k = 1,2$ do poľa $SA_{1,2}$. Triedime teda dve tretiny zo všetkých sufixov pôvodného reťazca. Dosiahneme to tak, že vytvoríme nový reťazec S' nad abecedou Σ^3, t.j. znakmi budú trojice znakov z pôvodného reťazca, pričom

$$S' = S[1..3]S[4..6]\ldots S[3n'+1..3n'+3]S[2..4]S[5..7]\ldots S[3n'+2..3n'+4],$$

kde n' je najmenšie také, aby $S[3n'+3] = 0$. V našom príklade je to

$$S' = [abb][cab][bd0][bbc][abb][d00] = [1,2,2][3,1,2][2,4,0][2,2,3][1,2,2][4,0,0].$$

Sufixy v S' zodpovedajú vybraným sufixom S a sú aj v tom istom lexikografickom poradí. Stačí nám teda pre S' rekurzívne vytvoriť sufixové pole SA'. Predtým však ešte musíme prečíslovať znaky v S', aby sme dostali reťazec nad abecedou $\{1,\ldots,n\}$. To spravíme tak, že utriedime trojice znakov, ktoré tvoria znaky S' radix sortom v čase $O(n)$. Máme nový reťazec $S' = 1,4,3,2,1,5,0$, pričom $|S'| \approx 2/3|S|$. Teraz môžeme rekurzívne pustiť algoritmus na S', čím dostaneme sufixové pole SA'. V našom príklade dostaneme $SA' = (6,0,4,3,2,1,5)$.

Keď sme spočítali sufixové pole SA' pre S', prepočítame indexy tak, aby sa vzťahovali na pôvodný reťazec S a uložíme do poľa $SA_{1,2}$. Výsledné pole SA neskôr spravíme z poľa $SA_{1,2}$ povkladaním tých sufixov, ktoré sme neuvažovali. Príklad tejto transformácie vidíme v tabuľke 5.

S'	1	4	3	2	1	5	0
pozícia v S'	0	1	2	3	4	5	6
pozícia v S	1	4	7	2	5	8	
SA'	6	0	4	3	2	1	5
$SA_{1,2}$	–	1	5	2	7	4	8

Tabuľka 5: Polia SA' a $SA_{1,2}$ pre príklad z tabuľky 4.

Vytvoríme tiež v $O(n)$ čase pole *rank* definované nasledovne:

$$rank[i] = \begin{cases} 0 & i \geq n \\ - & i \bmod 3 = 0 \\ j & SA_{1,2}[j] = i \end{cases}$$

Ak teda chceme porovnať lexikograficky dva sufixy na pozíciách $3i+k$ a $3j+k'$ kde $k,k' \in \{1,2\}$, stačí nám porovnať príslušné hodnoty v poli *rank* (pozri príklad v tabuľke 6).

Poz	0	1	2	3	4	5	6	7	8	9
S	f	a	b	b	c	a	b	b	d	
	5	1	2	2	3	1	2	2	4	0
rank	–	1	3	–	5	2	–	4	6	0

Tabuľka 6: Pole rank pre príklad z tabuľky 4.

Krok 2: Utriedime sufixy tvaru $S[3i..n]$ do SA_0. Sufix $S[3i..n]$ reprezentujeme ako $(S[3i], rank[3i+1])$. V našom príklade dostávame $S[0..n]$ ako $(f,1)$, $S[3..n]$ ako $(b,5)$ a $S[6..n]$ ako $(b,4)$. Ak chceme porovnávať dva sufixy na pozíciách $3i$ a $3j$, stačí porovnávať ich reprezentáciu pomocou dvojíc, lebo

$$S[3i..n] < S[3j..n] \iff S[3i] < S[3j] \vee (S[3i] = S[3j] \wedge rank[3i+1] < rank[3j+1]).$$

Takto vzniknuté dvojice teda vieme utriediť lexikograficky radix sortom v čase $O(n)$. V našom príklade dostávame $SA_0 = (6,3,0)$.

Krok 3: Zlúčime triedené zoznamy sufixov SA_0 a $SA_{1,2}$ do výsledného sufixového poľa SA. Postupujeme podobne ako pri zlučovaní zoznamov v MergeSorte, chceme však každú dvojicu sufixov (jeden z SA_0 a jeden z $SA_{1,2}$) porovnať v konštantnom čase. Majme teda sufix $S[i..n]$ z poľa $SA_{1,2}$ s sufix $S[j..n]$ z SA_0.

ak $i \bmod 3 = 1$:
$$S[i..n] \leq S[j..n] \iff (S[i], rank[i+1]) \leq (S[j], rank[j+1])$$
ak $i \bmod 3 = 2$:
$$S[i..n] \leq S[j..n] \iff (S[i], S[i+1], rank[i+2]) \leq (S[k], S[j+1], rank[j+2])$$

Príklad niekoľkých porovnaní:

$$S[3..n] \leq S[7..n] \iff (b,5) \leq (b,6)$$
$$S[3..n] \leq S[4..n] \iff (b,5) \leq (c,2)$$
$$S[3..n] \leq S[2..n] \iff (b,c,2) \leq (b,b,5)$$
$$S[6..n] \leq S[2..n] \iff (b,b,6) \leq (b,b,5)$$

Zložitosť

- Krok 1: radixSort $O(n)$, rekurzia na S', $T(\frac{2}{3}n)$, vytvorenie $SA_{1,2}$, *rank* $O(n)$.
- Krok 2: radixSort v $O(n)$
- Krok 3: zlučovanie v $O(n)$

Celkový čas je teda $T(n) = T(\frac{2}{3}n) + O(n)$ (úplne správne by sme mali uvažovať aj drobný aditívny člen pri $\frac{2}{3}n$, to však pre jednoduchosť zanedbáme). Rekurencie tvaru $T(n) = aT(\frac{n}{b}) + f(n)$ vieme riešiť použitím Master theorem, pričom nás zaujíma prípad $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$, ktorý sa dá použiť iba ak navyše platí $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ pre nejaké $c < 1$. Potom dostávame $T(n) = \Theta(f(n))$.

V našom prípade máme $a = 1$, $b = \frac{3}{2}$, $\log_{\frac{3}{2}}(1) = 0$. Ľahko overíme aj dodatočnú podmienku, lebo $a \cdot f(\frac{n}{b}) = \frac{2}{3}n$. Dostávame teda, že čas nášho algoritmu je lineárny: $T(n) = \Theta(n)$.

Namiesto Master theorem môžeme spočítať čas strávený na každej úrovni rekurzie. Na vrchnej úrovni potrebujeme čas $c \cdot n$, a na každej ďalšej úrovni sa nám veľkosť vstupu zmenší na dve tretiny, na i-tej úrovni teda potrebujeme čas $c \cdot \left(\frac{2}{3}\right)^i n$. Sčítaním tejto geometrickej postupnosti dostávame

$$T(n) \leq \sum_{i=0}^{\infty} c \cdot n \left(\frac{2}{3}\right)^i = c \cdot n \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i = O(n).$$

18.4 Konštrukcia sufixového stromu zo sufixového poľa

Do stromu budeme pridávať sufixy v poradí, v akom sú v sufixovom poli. V každom vrchole budeme mať uloženú hodnotu $d(u)$ určujúcu sufixovú hĺbku vrcholu, t.j. dĺžku reťazca, ktorý mu zodpovedá. Predpokladajme, že už máme v strome sufixy $SA[0], SA[1], \ldots SA[i-1]$ a chceme pridať $T[SA[i]..n-1]$. Nech k je dĺžka najdlhšieho spoločného prefixu $T[SA[i-1]..n-1]$ a $T[SA[i]..n-1]$. So všetkými ostatnými sufixami, ktoré už máme v strome, má aktuálny sufix tiež spoločný prefix dĺžky najviac k. Nový sufix bude teda nový list, ktorý sa na súčasný strom napája vo vrchole reťazcovej hĺbky k ležiacom niekde na ceste z listu pre sufix $SA[i-1]$ do koreňa.

Začneme teda v liste pre sufix $SA[i-1]$ a posúvame sa smerom ku koreňu, kým nenájdeme prvý vrchol u s $d(u) \leq k$. Ak $d(u) = k$, nový sufix prilepíme nový list označený sufixom $SA[i]$. Ak $d(u) < k$, rozdelíme hranu z u do jeho dieťaťa novým vrcholom s reťazcovou hĺbkou k a pripojíme nový list. V ďalšej iterácii pre $i+1$ začneme opäť hľadať smerom hore z tohto listu.

Na prvý pohľad je tento algoritmus kvadratický, lebo na pridanie jedného sufixu môžeme potrebovať výjsť hore až po n hranách. Dá sa však dokázať, že celkový počet posúvania sa po strome je lineárny, podobne ako pri konštrukcii kartézskeho stromu. Pri vkladaní sufixu $SA[i]$ sa totiž posúvame zdola z listu pre sufix $SA[i-1]$. Každý vrchol, cez ktorý takto prejdeme a zistíme, že jeho reťazcová hĺbka je príliš veľká, už nikdy nenavštívime pri vkladaniach ďalších sufixov, lebo budeme začínať z nových listov, ktoré nezdieľajú túto časť cesty.

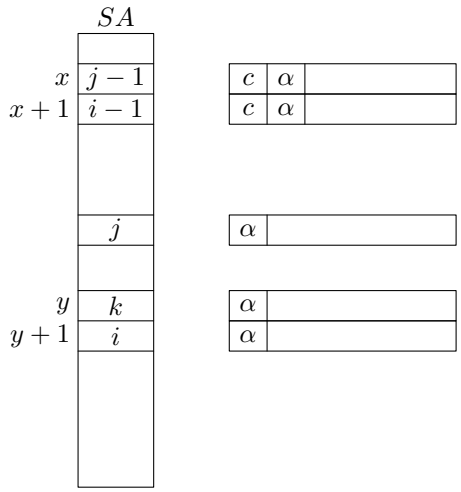
Zostáva nám len rýchlo určiť hodnotu najdlhšieho spoločného prefixu pre každé dva susedné sufixy v sufixovom poli. Toto vieme spočítať v lineárnom čase pomocou algoritmu, ktorý uvedieme v kapitole 18.5.

```
1  create roof and leaf w corresponding to SA[0]
2  v = w;
3  for(int i=1; i<=n; i++) {
4     while(v.parent.string_depth>L[i-1]) {
5       v = v.parent;
6     }
7     if(v.parent.string_depth<L[i-1]) {
8       split edge from v.parent to v with a new vertex
9       at string depth L[i-1]
10    }
11    attach new leaf w for SA[i] from v.parent;
12    v = w;
13 }
```

Ak teda spojíme lineárny algoritmus na konštrukciu sufixového poľa, lineárny výpočet spoločných prefixov a lineárny prevod so sufixového poľa na sufixový strom, dostávame $O(n)$ algoritmus na konštrukciu sufixového stromu, ktorý funguje v čase $O(n)$ aj pre abecedu $\{1, 2, \ldots, n\}$. Všimnite si, že deti každého vrcholu budú pridávané v abecednom poradí a teda ich môžeme pridávať na koniec utriedeného poľa dynamickej veľkosti.

18.5 Výpočet LCP hodnôt pre sufixové pole

Pripomeňme si, že $lcp(A,B)$ je dĺžka najdlhšieho spoločného prefixu reťazcov A a B a pre daný reťazec S definujeme $LCP(i,j) = lcp(T[SA[i]..n-1], T[SA[j]..n-1])$. V tejto časti uvedieme algoritmus, ktorý pre každý prvok sufixového poľa spočíta dĺžku najdlhšieho spoločného prefixu medzi $T[SA[i]..n-1]$ a $T[SA[i+1]..n-1]$, t.j. $LCP(i,i+1)$. Takúto hodnotu uložíme do nového poľa $L[i]$. Ako sme videli, tieto hodnoty sú potrebné pre rýchle vyhľadávanie vzoriek v sufixovom poli aj pre konštrukciu sufixového stromu zo sufixového poľa. Teraz uvedieme kľúčovú lemu algoritmu.



Obr. 36: Obrázok ilustrujúci lemu 3.

Lema 3. Majme x a y, pre ktoré platí $SA[x+1]+1 = SA[y+1]$. Potom $L[y] \geq L[x]-1$.

Dôkaz. Pozrime sa na pole SA (obrázok 36). Chceme zistiť hodnotu $LCP(y,y+1)$, pričom $SA[y+1] = i$ a $SA[y] = k$. Nájdeme si hodnotu $i-1$ na nejakej pozícii $x+1$ a pozrieme sa na hodnotu tesne pred ňou, $j-1$. Nech najdlhší spoločný prefix sufixov začínajúcich na pozíciách $i-1$ a $j-1$ je $c\alpha$. Sufixy začínajúce na pozíciách i aj j sa teda začínajú slovom α. Keďže $S[j-1..n]$ je v lexikografickom poradí pred $S[i-1..n]$, tak aj $S[j..n]$ musí byť pred $S[i..n]$ preto aj $S[k..n]$ začína na α. Vidíme, že $LCP(y,y+1) \geq LCP(x,x+1)-1$. □

Na využitie tejto lemy potrebujeme vedieť rýchlo zistiť, kde v poli SA je nejaká hodnota i. Na to použijeme inverzné pole *rank*, pre ktoré platí $rank[i] = x \iff SA[x] = i$. Toto pole vieme vypočítať jedným prechodom poľa SA v čase $O(n)$.

```
1 for(i=0;i<=n,i++){
2   rank[SA[i]] = i;
3 }
```

Samotný algoritmus postupuje od najdlhších sufixov po najkratšie a pre každý sufix spočíta jeho prefix s jeho predchodcom v poli SA.

```
1 h = 0;
2 for(i=0; i<=n; i++) {
3   if(rank[i]>0) {
```

```
4       k = SA[rank[i]-1];
5       // porovnavame sufixy S[i..n-1] a S[k..n-1]
6       // vieme vsak, ze maju aspon h znakov spolocnych
7       while (T[i+h]==T[k+h]) { h++; }
8       // nasli sme prvu poziciu, kde sa retazce nerovnaju
9       L[rank[i]-1] = h;
10      if(h>0) { h--; }
11    }
12  }
```

Ak v nejakej iterácii pre $i-1$ nájdeme hodnotu h, tak pre i bude hodnota LCP aspoň $h-1$. Najprv sa pozrieme, kde v poli SA je i, ak to nie je na začiatku, pozrieme sa, aký reťazec je pred ním, a potom ich porovnávame od h-tej pozície. Nakoniec upravíme h na $h-1$, lebo v nasledujúcej iterácii máme garantovaných $h-1$ zhôd.

Časová zložitosť. Budeme rátať počet porovnaní vo vnútornom cykle algoritmu. Počet iterácií, v ktorých porovnanie dopadne nerovnosťou, je najviac n, lebo v každej iterácii máme najviac jednu nerovnosť. Počet rovností už nie je taký jasný, ale vieme že každá rovnosť zvýši h o jedna. Keďže h začne na nule a dosahuje hodnotu najviac n, v každom kroku sa zmenší najviac o jeden, preto sa najviac n krát zníži. Takže počet rovností môže stúpnuť najviac $2n$ krát. Dokopy teda máme najviac $3n$ porovnaní.

18.6 Burrows-Wheelerova transformácia

Burrows-Wheeler transformácia (BWT) je transformácia textu využívaná pri kompresii (napríklad v programe bzip2), ale ako uvidíme neskôr, aj v úsporných dátových štruktúrach na hľadanie vzorky v texte. BWT transformuje vstupný reťazec S na reťazec $bw(S)$, pričom z reťazca $bw(S)$ vieme neskôr spätne získať reťazec S. Reťazec $bw(S)$ je permutáciou reťazca S, no táto permutácia sa často dá oveľa lepšie skomprimovať ako pôvodný reťazec.

Transformácia. BWT pozostáva z niekoľkých krokov.

- na koniec reťazca S pridáme špeciálny symbol \$, ktorý sa nevyskytuje v S
- vytvoríme maticu cyklických posunov reťazca S\$
- lexikograficky utriedime riadky matice
- $bw(S)$ je posledný stĺpec matice

Tretí krok vieme spočítať pomocou sufixového poľa: $bw(S)[i] = S[SA[i]-1]$, alebo $bw(S)[i] = S[n]$, ak $SA[i] = 0$.

Príklad. Zoberme vstupný reťazec $S =$ banana a pridajme nakoniec znak \$. Matica cyklických posunov a matica utriedených cyklických posunov:

b	a	n	a	n	a	$
a	n	a	n	a	$	b
n	a	n	a	$	b	a
a	n	a	$	b	a	n
n	a	$	b	a	n	a
a	$	b	a	n	a	n
$	b	a	n	a	n	a

$	b	a	n	a	n	**a**
a	$	b	a	n	a	**n**
a	n	a	$	b	a	**n**
a	n	a	n	a	$	**b**
b	a	n	a	n	a	**$**
n	a	$	b	a	n	**a**
n	a	n	a	$	b	**a**

$bw(S\$)$ = annb$aa

Spätná transformácia. Máme daný posledný stĺpec matice lexikograficky usporiadaných cyklických rotácií. Z neho vieme spočítať prvý stĺpec tejto matice jednoduchým triedením posledného stĺpca. Pre jednoduchosť označme prvý stĺpec matice F a posledný stĺpec L; $L = bw(S\$)$.

F		L
$	...	a
a	...	n
a	...	n
a	...	b
b	...	$
n	...	a
n	...	a

Platí, že $F[i]$ nasleduje po $L[i]$ v S. Preto symbolu $ v L prislúcha $S[0] = b$.

Následne môžeme najsť $S[0]$ v L a tým dostať $S[1]$ v F. Teda $S[1] = a$. Takto by sme mohli pokračovať ďalej, no ku písmenu a v L prislúchajú písmená z množiny $\{n, \$\}$. Nevieme teda jednoznačne určiť $S[2]$.

$	b	a	n	a	n	$\mathbf{a}_5$
$\mathbf{a}_5$	$	b	a	n	a	$\mathbf{n}_4$
$\mathbf{a}_3$	n	a	$	b	a	$\mathbf{n}_2$
$\mathbf{a}_1$	n	a	n	a	$	**b**
b	a	n	a	n	a	**$**
$\mathbf{n}_4$	a	$	b	a	n	$\mathbf{a}_3$
$\mathbf{n}_2$	a	n	a	$	b	$\mathbf{a}_1$

Môžeme si všimnúť, že relatívne poradie písmen a, alebo n je v L a F rovnaké. Pozrime sa napr. na písmená a v F. Ich poradie (5,3,1) je určené na základe kontextu na konci (banan, ban, b). Tento kontext určuje aj ich poradie v L.

Teraz už vieme jednoznačne vytvoriť pôvodný reťazec. Nájdeme $S[0]$ v L, dostaneme $S[1]$ v F ($S[1]$ = tretie a). Nájdeme tretie a v L, dostaneme $S[2]$ v F ($S[2]$ = druhé n, atď.

Na realizáciu použijeme nasledovné dátové štruktúry.

Reprezentácia pozícií písmen v L:

$: 4

a: 0, 5, 6

0: $_0$ a$_0$
1: a$_0$ n$_0$
2: a$_1$ n$_1$
3: a$_2$ b$_0$
4: b$_0$ $_0$
5: n$_0$ a$_1$
6: n$_1$ a$_2$

b: 3
n: 1, 2

Použitie BWT v kompresii

Riadky matice, v ktorej sú usporiadané cyklické reťazce, majú ako prefixy postupne prvky zo sufixového poľa. Prvky zo sufixovéhom poľa *SA*, ktoré majú spoločný prefix (v *SA* sú za sebou) majú v *S* často ako predchodcu rovnaké písmeno. Tieto písmená sú práve v poslednom stĺpci matice. V nasledujúcom príklade písmenu ' . ' predchádzajú písmená u,a, preto v $bw(S\$)$ je úsek obsahujúci iba tieto písmená.

$$\begin{aligned} S &= \texttt{ema.ma.mamu.mama.ma.emu.ema.sa.ma\$} \\ bw(S\$) &= \texttt{auaaaauaammsmmmmmm\$....ae.e..ea.mm} \end{aligned}$$

Môžeme použiť Move-to-front(MTF) recording, kde písmeno $S[i]$ nahradíme počtom rôznych písmen od posledného výskytu $S[i]$ v $S[0..i-1]$. Pred reťazec $bw(S\$)$ ešte pridáme všetky písmená z abecedy od najmenšieho po najväčší, aby sme vedeli skonštruovať postupnosť.

`$.aemsu|auaaaauaammsmmmmmm$....ae.e..ea.mm` je zakódovaný do postupnosti

`411000110303100000660004621101240`

Z tejto postupnosti vieme ľahko získať $bw(S\$)$ a z neho S. V tejto postupnosti sú malé čísla, veľa núl. Pre anglický text viac ako 50% tvoria nuly. Takáto postupnosť je ľahko komprimovateľná.

V príklade je entropia S rovná 2,37, entropia MTF reťazca $bw(S\$)$ je rovná 2,18. Napríklad pri použití aritmetického kódovania na dostatočne dlhom reťazci potrebujeme na zakódovanie jedného znaku v priemere počet bitov, ktorý sa približne rovná entropii. Bližšie informácie o BWT nájdu záujemcovia v článoku Manzini: The Burrows-Wheeler Transform: Theory and Practice. MFCS 1999, kapitola 2.

18.7 TO DO

V BWT miestami podrobnejšie vysvetliť, napr. použité dátové štruktúry.

19 Výpočet editačnej vzdialenosti

Hľadanie približných výskytov vzoriek v texte je dôležité z viacerých pohľadov. Za všetky spomenieme napríklad biológiu, kde môže vzniknúť napr. mutácia v DNA, chyby v texte alebo

odhaľovanie plagiatorstva. Videli sme už vyhľadávanie s Hammingovou vzdialenosťou d_H a algoritmus pre takéto vyhľadávanie s časovou zložitosťou $\mathcal{O}(nk)$.

19.1 Editačná vzdialenosť

Definícia 4. Editačná alebo Levenshteinova vzdialenosť d_E je vzdialenosť, ktorá dovoľuje robiť tri operácie na reťazcoch: (Zápis $u \to v$ znamená, že operácia zmení reťazec u na reťazec v.)

substitúcia $uav \to ubv$, kde $u, v \in \Sigma^*$ a $a, b \in \Sigma$,

inzercia $uv \to uav$, kde $u, v \in \Sigma^*$ a $a \in \Sigma$,

delécia $uav \to uv$, kde $u, v \in \Sigma^*$ a $a \in \Sigma$.

Vzdialebnosť je definovaná

$$d_E(S,T) = \text{najmenší počet operácií, ktoré transformujú } S \text{ na } T.$$

Príklad 6. Majme reťazec $S = strom$. Vykonáme nasledovné operácie: zmaž štvrý znak (dostávame *strm*), vlož *a* na tretie miesto (dostávame *starm*), zmeň piaty znak na *ý* (dostávame *starý*). Môžeme si tiež všimnúť, že slovo *strom* na slovo *starý* nevieme prerobiť na menej ako tri operácie. Preto $d_E(strom, starý) = 3$.

Zamyslíme sa, aká môže byť najväčšia editačná vzdialenosť medzi dvoma reťazcami. Môžu sa líšiť v každom písmene, napríklad pre a^n a b^m to bude maximum ich dĺžok. (Musíme substituovať všetky písmená kratšieho slova na písmena dlhšieho a doplniť písmenká na koniec.) Naopak najmenšia editačná vzdialenosť reťazcov s dĺžkami n a m bude absolútna hodnota rozdielu $|n-m|$, pretože v najlepšom prípade je kratší reťazec podreťazcom dlhšieho, preto stačí dopísať chýbajúce písmenká.

Definícia 5. Zarovnanie alebo alignment je dvojriadková tabuľka (matica) pozostávajúca z písmen reťazcov alebo pomlčiek. V prvom riadku je prvé slovo, v druhom druhé. (Pričom v oboch riadkoch môžu byť nejaké pomlčky.) Je zostavená tak, že pod seba napíšeme písmenká ktoré sa nezmenili alebo sa zmenili substitúciou. Tie písmená čo boli zmazané alebo vložené chceme mať v samostatných stĺpcoch, doplnené pomlčkami.

s	t	-	r	**o**	**m**
s	t	**a**	r	-	**ý**

Tabuľka 7: Zarovnanie pre operácie uvedené v príklade 6

Zodpovedajúce zarovnanie k príkladu 6 je zobrazené v tabuľke 7. Zo zarovnania sa ľahko vypíše postupnosť krokov, ako zmeniť prvé slovo na druhé. Tiež vieme ľahko zrátať počet operácií – tam kde sa naše dva riadky líšia. (V tabuľke 7 vyznačené hrubo.)

Na editačnú vzdialenosť sa môžeme pozerať aj ako na cenu najlpšieho zarovnania. iné zarovnanie tých slov je napríklad to zobrazené v tabuľke 8.

s	t	**r**	**o**	**m**
s	t	**a**	**r**	**ý**

Tabuľka 8: Iné zarovnanie reťazcov z príkladu 6

Ako spočítať editačnú vzdialenosť Použijeme dynamické programovanie. (Zrátame to pre podproblémy – podreťazce – a z toho pre celý reťazec, vypočítané hodnoty si budeme pamätať v tabuľke A.)

Podproblém $A[i,j] = d_E(S[1..i], T[1..j])$. Chceme zistiť $A[m,n]$, kde budeme mať uloženú editačnú vzdialenosť reťazcov S a T.

Začneme vypĺňať tabuľku od triviálnych prípadov. Platí $A[0,j] = j$, pretože do prvého reťazca musíme vložiť j znakov. Rovnako platí $A[i,0] = i$, z toho istého dôvodu. Všeobecný prípad, hodnotu na políčku $A[i,j]$, môžeme vyrátať formulou

$$A[i,j] = \begin{cases} \min(A[i-1,j]+1, A[i,j-1]+1, A[i-1,j-1]) & \text{ak } S[i] = T[j], \\ \min(A[i-1,j]+1, A[i,j-1]+1, A[i-1,j-1]+1) & \text{ak } S[i] \neq T[j]. \end{cases}$$

Túto formulu môžeme odvôvodniť nasledovne. Uvažujme najlacnejšie zarovnanie. Ak sú poslednom znaku dve písmená, tak je to tretia zložka predchádzajúcej formuly, (použijeme substitúciu alebo nič,) ak je tam pomlčka, tak jedna z prvých dvoch zložiek. (Buď delécia alebo inzercia.)

```
for(i=0;i<=n;i++){
  for(j=0;j<=n;j++){
    vypocitaj A[i,j] podla rekurencie;
  }
}
```

Výsledok je $d_E(S,T) = A[m,n]$.

Časová zložitosť Je zjavne $\mathcal{O}(mn)$.

V niektorých aplikáciách chceme nielen zistiť editačnú vzdialenosť, ale aj ako premeníme jeden reťazec na druhý. (tj. chceme nájsť minimálne zarovnanie.) Preto si pri $A[i,j]$ môžeme pamatať, ako sme to dostali. (Je to v prvom prípade inzercia, v druhom delécia, v treťom substitúcia.) Budeme to robiť napríklad v poli $B[i,j]$, potom postupnosť krokov získame "odzadu". Graficky môžeme pole B znázorniť ako šípky, pozri tabuľku 10.

Príklad 7. Majme reťazce $S = baab$, $T = abaa$. V tabuľke 9 a v tabuľke 10 vidíme hodnoty polí A a B. Minimálne zarovnanie dostaneme, ak pôjdeme po šípkach v poli B tak, že začneme na políčku $B[m,n]$. Pritom vodorovná šípka značí inzerciu, zvislá deléciu a diagonálna substitúciu alebo žiadnu operáciu. Preto reťazec T dostaneme z reťazca S vykonaním operácií: vlož a na pozíciu 1, zmaž b z poslednej pozície.

Vadí nám, že tento algoritmus používa kvadratickú pamäť. Výhodou je, že ak nás zaujíma iba hodnota editačnej vzdialenosti a nie zarovnanie, potrebujeme si pamätať len predchádzajúce hodnoty, takže nám stačí $\mathcal{O}(n)$ pamäť. Nabudúce bude algoritmus, ktorý bude asymptoticky rovnako rýchly, ale s lineárnou pamäťou aj pre výpis zarovnania. Na dynamické programovanie sa môžeme pozerať aj ako na hľadanie najkratšej cesty v acyklickom grafe (viď obr. 37, 38).

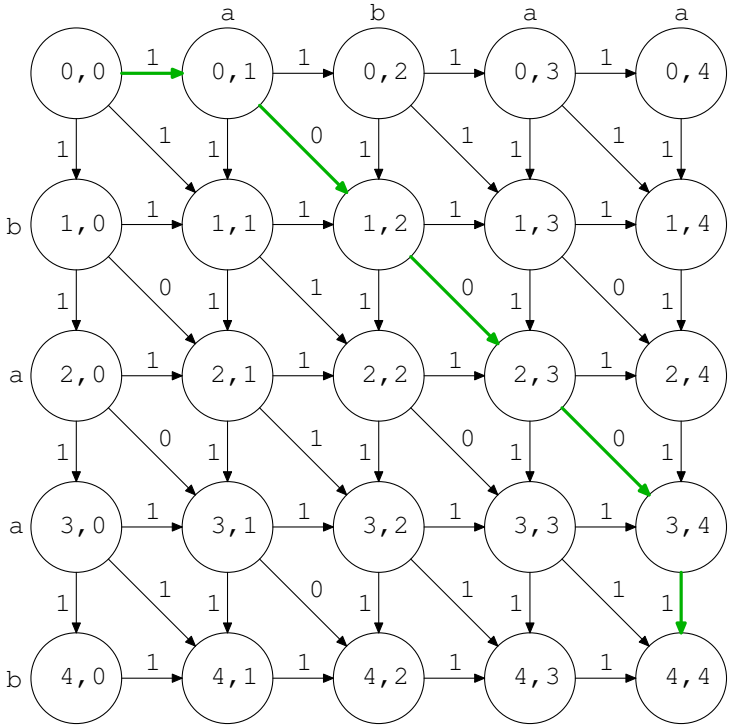
	-	a	b	a	a
-	0	1	2	3	4
b	1	1	1	2	3
a	2	1	2	1	2
a	3	2	2	2	1
b	4	3	2	3	2

Tabuľka 9: Pole A k príkladu 7.

	–	a	b	a	a
–	○	←	←	←	←
b	↑	↖	↖	←	←
a	↑	↖	↖	↖	↖
a	↑	↖	↖	↖	↖
b	↑	↑	↖	↑	↑

Tabuľka 10: Pole B k príkladu 7.

Obr. 37: Reprezentácia editačnej vzdialenosti grafom



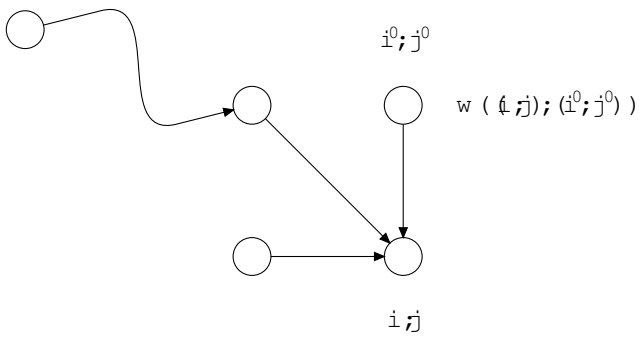
Najkratšie hrany v grafe sa dajú počítať napr. Dijkstrovým algoritmom. Čo ale skutočne hľadáme, je najkratšia cesta v orientovanom *acyklickom* grafe – Dijkstru nutne nepotrebujeme, stačí nám dynamické programovanie:

1. topologicky si zoradíme hrany
2. pre vrcholy i a j bude $A[i,j]$ dĺžka najkratšej cesty z $(0,0)$ do (i,j). Toto vieme spočítať dynamickým programovaním – pozeráme sa na predkov aktuálneho vrcholu, a hodnotu získame ako

$$A[i,j] = \min\{A[i',j'] + w((i',j'),(i,j)) \mid (i',j'),(i,j) \in E\}$$

.

Máme sľúbené dynamické programovanie, ktoré funguje v čase $O(mn)$ a pamäti $O(mn)$. Navyše, pokiaľ chceme len $d_E(S,T)$, stačí nám pamäť len $O(n+m)$ (pamätáme si iba 2 riadky A). Ak chceme vypísať zarovnanie (postupnosť operácií), potrebujeme tabuľku B o rozmeroch pôvodnej tabuľky A.



Obr. 38: Najkratšia cesta z $(0,0)$ do (i,j)

19.2 Hirschbergov algoritmus (1975)

Tento algoritmus sa od klasického výpočtu editačnej vzdialenosti líši tým, že potrebuje pamäť iba $O(n+m)$ namiesti $O(nm)$. Rozdiely oproti predchádzajúcemu algoritmu raz prejdem celú maticu a spočítam všetky A. Zapamätám si, kde moja cesta prejde cez stredný riadok matice – pričom ju počítam tak, že si pamätám iba dva riadky.

Stredný riadok matice označme k-ty. Zavedieme si zovšeobecnené pole $B_k[i,j]$ – najväčší index v riadku k, cez ktorý prechádza najkratšia cesta z $(0,0)$ do (i,j). Hodnoty $B_k[i,j]$ počítame podobne ako hodnoty B. Konkrétne pre $i > k$ máme

1. ak $A[i,j] = A[i-1,j-1] + w(S[i],T[j])$, potom $B_k[i,j] = B_k[i-1,j-1]$.
2. ak $A[i,j] = A[i-1,j] + 1$, potom $B_k[i,j] = B_k[i-1,j]$.
3. ak $A[i,j] = A[i,j-1] + 1$, potom $B_k[i,j] = B_k[i,j-1]$

Pre $i = k$ nastavíme $B_k[i,j] = j$.

Ak už poznáme $A[i-1,*]$ a $B_k[i-1,*]$, vieme spočítať $A[i,*]$ aj $B_k[i,*]$.

Nech $k' = B_k[m,n]$. Potom v optimálnom zarovnaní sa $S[1..k]$ zarovná s $T[1..k']$ a $S[k+1..m]$ s $T[k'+1..n]$. Toto použijeme na rekurzívny algoritmus na výpočet zarovnania:

```
1  optA(l1, r1, l2, r2) {
2      // zarovnaj S[l1..r1] a T[l2..r2]
3      if(r1-l1 <= 1 || r2-l2 <=1)
4          vyries pomocou dynamickeho programovania
5      else {
6          k=(r-l+1)/2;
7          for (i=0; i<=k; i++)
8              pocitaj A[i,*] z A[i-1,*]
9          for (i=k+1; i<=r-l+1; i++)
10             pocitaj A[i,*], B_k[i,*] z A[i-1,*], B_k[i-1,*]
11         k2=B_k[r1-l1-1,r2-l2-1];
12         optA(l1, l1+k-1, l2, l2+k2-1);
13         optA(l1+k, r2, l2+k2, r2);
14     }
15 }
```

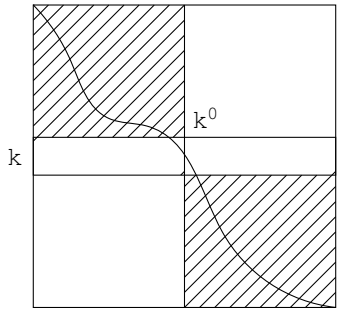
Označme si $N = nm$ (súčin dĺžky dvoch daných reťazcov). Na hornej úrovni rekurzie spúšťame dynamické programovanie pre celú maticu – čas bude teda $\leq c \cdot N$. Teraz vzniknú dva podproblémy – označme ich N_1 a N_2 (obr. 39):

$$N_1 = (k - l + 1)(k' - l' + 1)$$

$$N_2 = (r - k)(r' - k')$$

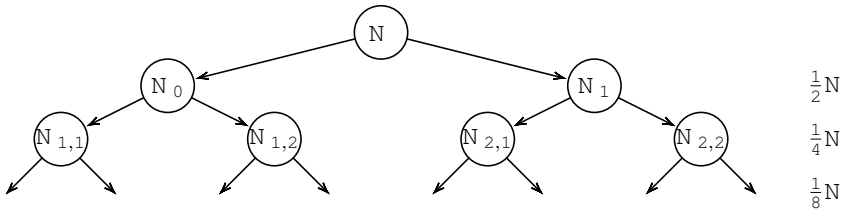
$$T(N) \leq T(N_1) + T(N_2) + c \cdot N$$

.



Obr. 39: Vzniknuté podproblémy pri výpočte hodnôt matice

Platí, že $N_1 + N_2 = \frac{1}{2}N$. Na hlavnej úrovni rekurzie máme problém veľkosti N, ktorý sa rozpadne na na menšie problémy (obr. 40).



Obr. 40: Strom rekurzie pre výpočet matice

Výška vzniknutého stromu bude $\log_2 m$.

$$T(n) = c \cdot N + \frac{1}{2}c \cdot N + \frac{1}{4}c \cdot N + \ldots \leq c \cdot N \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

$$= \underbrace{2 \cdot N}_{norm.dyn.prog}$$

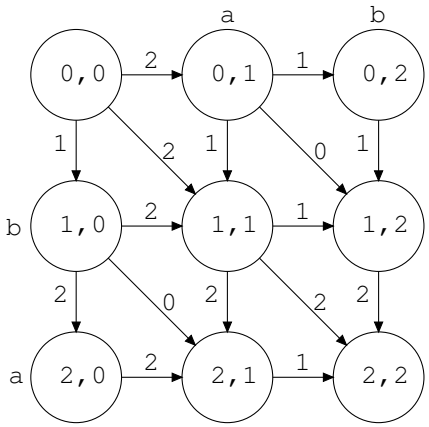
V priemere každé políčko vyplním dvakrát, teda čas $O(mn)$ je zhruba dvojnásobný. Pamäťová zložitosť – potrebujeme 2 riadky z oboch matíc A, B_k ($O(n)$), zásobník $O(\log m)$, vstup $O(n+m)$, výstup $O(n+m)$. Celková pamäť je $O(n+m)$.

19.3 Zovšeobecnené editačné vzdialenosti

Definícia *Zovšeobecnená editačná vzdialenosť*, označujeme d'_E, je editačná vzdialenosť, ktorá má ku každej operácii priradenú cenu:

- $w_s(a,b)$ – substitúcia a za b
- $w_d(a)$ – zmazanie a
- $w_i(a)$ – vloženie a

Ak máme dvojpísmenkové reťazce ab a ba, kde ceny budú $w_d(a) = w_i(a) = 1$, $w_d(b) = w_i(b) = w_s(a,b) = w_s(b,a) = 2$ (obr. 41).



Obr. 41: Graf pre dvojpísmenkové reťazce ab a ba

$$A[i,j] = \min \begin{cases} A[i-1,j-1] + w_s(S[i],T[j]) \\ A[i-1,j] + w_d(S[i]) \\ A[i,j-1] + w_i(T[i]) \end{cases}$$

$d_E(S,T)$ je metrika:

- $d_E(S,T) \geq 0$
- $d_E(S,T) = 0 \iff S = T$
- $d_E(S,T) = d_E(T,S)$
- $d_E(S,T) \leq d_E(S,U) + d_E(U,T)$

Príklad

1. $w_d(a) = 1$, $w_i(a) = 2$, $d'_E(\varepsilon,a) \neq d'E(a,\varepsilon)$
2. $w_s(a,b) = 1, w_s(b,c) = 1, w_s(a,c) = 3$. $a \to b \to c = 2$

	a a	a b	a- -a
skóre:	0	> 0	> 0

Voľba váhovacej funkcie na hľadanie optimálneho zarovnania. Pri váhovanej editačnej vzdialenosti sme hľadali zarovnanie s minimálnou cenou, pričom ceny stĺpcov boli nasledujúce:

Môžeme však namiesto vzdialenosti hľadať podobnosť a potom budeme hľadať zarovnanie s maximálnou cenou, kde ceny stĺpcov sú takéto:

	a a	a b	a- -a
skóre:	0	> 0	> 0

Tabuľka 11: príklad typického ohodnotenia

Uvažujme reťazec äcbaä äbcaä váhovaciu funkciu w spĺňajúcu podmienky:

- $w(x,x) = 0$
- $w(x,y) = w(y,x)$
- $w(x,y) \geq 0$

Optimálne zarovnania môžu vyzerať napríklad nasledovne:

$$\begin{matrix} a & c & b & a \\ a & c & b & a \end{matrix} \} 2w(b,c) \tag{1}$$

$$\begin{matrix} a & c & b & - & a \\ a & - & c & b & a \end{matrix} \} 2w(c,-) \tag{2}$$

$$\begin{matrix} a & c & b & a & - \\ - & a & b & c & a \end{matrix} \} 2w(a,-) + 2w(a,c) \tag{3}$$

Na uvedených príkladoch si môžeme všimnúť, že podľa nastavenia váh môžu byť rôzne zarovnania optimálne.

19.4 Podobnosť reťazcov

Ak sú reťazce tie isté, majú vysokú podobnosť.

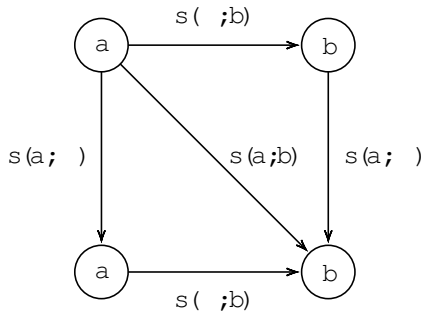
Definícia $s(a,b)$ – podobnosť znakov, kde $a,b \in \Sigma \cup \{-\}$ (tab. 12).

a	–	a
–	a	b
s(a,–)	s(–,a)	s(a,b)

Tabuľka 12: Príklad podobnosti reťazcov

Definícia *Cena zarovnania* je súčet podobností znakov v stĺpcoch. $s(S,T)$ je maximálna cena ich zarovnania.

Príklad Hľadáme najdlhšiu cestu v grafe z $(0,0)$ do (i,j) – v normálnom grafe je to ťažké, ale keďže náš graf je acyklický, môžeme použiť dynamické programovanie (obr. 42).



Obr. 42: Príklad grafu pri hľadaní podobnosti reťazcov

Keďže chceme podobnosť, podobnosť medzi rovnakými znakmi je zvyčajne 0, medzi rôznymi nižšia:

$$\begin{array}{rcl} s(a,a) & > & 0 \\ s(a,b) & < & 0, a \neq b \end{array}$$

Príklad

$$\begin{array}{rcl} s(a,a) & = & 1, a \in \Sigma \\ s(a,b) & = & -1, a \neq b, a, b \in \Sigma \cup \{-\} \end{array}$$

(tab. 13)

–	b	a	a	b
a	b	a	a	–
-1	1	1	1	-1

Tabuľka 13: Príklad zarovnania

Príklad

$$\begin{array}{rcl} s(a,a) & = & 0 \\ s(a,b) & = & -1 \\ s(S,T) & = & -d(S,T) \end{array}$$

Príklad

$$\begin{array}{rcl} s(a,a) & = & 1 \\ s(a,b) & = & 0 \end{array}$$

Pri tejto definícii dostávame $s(S,T) =$ dĺžka najdlhšej spoločnej podpostupnosti S a T.

20 Najdlhšia spoločná podpostupnosť

Definícia *Podpostupnosť* postupnosti $a_1a_2\ldots a_n$ je postupnosť $a_{i_1}a_{i_2}a_{i_3}\ldots a_{i_k}$, kde $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

Definícia: Postupnosť A je spoločná podpostupnosť postupností S a T, ak je podpostupnosťou S aj T. Dĺžku najdlhšej spoločnej podpostupnosti (longest common subsequence) označujeme $lcs(S,T)$.

```
S = EMA_MA_MA--MU
T = -MA-MA_MA_EMU
```

Obr. 43: Najdlhšia spoločná podpostupnosť $S =$ EMA_MA_MAMU a $T =$ MAMA_MA_EMU je MAMA_MAMU, teda $lcs(S,T) = 9$.

Problém hľadania najdlhšej spoločnej podpostupnosti sa dá redukovať na problém podobnosti reťazcov, ak hodnotiacu funkciu zvolíme nasledovne:

$$s(a,a) = 1$$
$$s(a,b) = s(a,-) = s(-,a) = 0$$

Výsledný algoritmus používa dynamické programovanie a má časovú zložitosť $O(mn)$, pamäťovú zložitosť vieme dosiahnuť $O(m)$. Špecifiká tejto hodnotiacej funkcie využíva nasledujúci algoritmus.

20.1 Algoritmus Hunt-Szymanski, 1977

Predstavme si pole $Z[1..m]$, ktoré bude pre každý výskyt písmena v S (na pozícii i) obsahovať zoznam indexov j do T, pre ktoré platí $T[j] = S[i]$ ("kde v poli T nájdem písmeno $S[i]$"). Utriedený bude od väčších j k menším.

Príklad:

$$\begin{aligned} Z[1] &= 9 \\ Z[2] &= 10,6,3,1 \\ Z[3] &= 7,4,2 \\ Z[4] &= 8,5 \\ Z[5] &= 10,6,3,1 \\ &\vdots \end{aligned}$$

Teraz spojíme všetky zoznamy $Z[i]$ do jedného zoznamu Z, pričom si pre každé číslo pamätáme z ktorej pozície ktorého $Z[i]$ pochádza (označme si $|Z| = r$):

$$Z = \underline{9,}\underline{10,6,3,1,}\underline{7,4,2,}\underline{8,5,}\underline{10,6,3,1,}\ldots$$

Ďalej využijeme nasledujúcu lemu.

Definícia: Rastúca podpostupnosť postupnosti Z je taká podpostupnosť $Z_{i_1}, Z_{i_2}, \ldots, Z_{i_k}$, pre ktorú platí $Z_{i_1} < Z_{i_2} < \cdots < Z_{i_k}$. Dĺžku najdlhšej rastúcej podpostupnosti Z (longest increasing subsequence) značíme $lis(Z)$.

Lema: $lcs(S,T) = lis(Z)$

Dôkaz vyplýva zo silnejšieho tvrdenia: medzi rastúcimi podpostupnosťami Z (IS) a spoločnými podpostupnosťami S a T (CS) existuje bijekcia (ktorá zachováva dĺžku). Pre každú IS, ktorá obsahuje $Z[i,j]$ totiž existuje CS, ktorá na danej pozícii obsahuje písmeno $S[i] = T[Z[i,j]]$. A naozaj, z každého zoznamu $Z[i]$ vyberieme najviac jedno číslo (keďže je klesajúci), takže výsledná CS bude podpostupnosť S. Navyše daná IS je rastúca postupnosť indexov do T, teda výsledná CS bude aj podpostupnosť T.

Algoritmus teda pozostáva z dvoch krokov:

1. konštrukcia Z
2. výpočet $lis(Z)$

Konštrukcia Z

- Pre malú abecedu použijeme pole $A[1..\sigma]$, kde $A[c]$ bude zoznam výskytov písmena c v T:

```
1 for (i=1; i≤n; i++)
2   pridaj i na zaciatok A[T[i]]
3 for (i=1; i≤m; i++)
4   Z[i] = A[S[i]];
```

  Časová zložitosť tohoto algoritmu je $O(n+m+\sigma+r)$.

- Pre veľkú abecedu uložíme A do hashovacej tabuľky alebo vyhľadávacieho stromu (kľúče budú písmená abecedy, hodnoty budú zoznamy). V druhom prípade budeme mať strom s najviac m prvkami, do ktorého vykonáme $n+m$ prístupov, čo nám dáva časovú zložitosť $O((n+m)\log m + r)$.

- Ak za abecedu považujeme riadky vstupných súborov tvorené písmenami z nejakej malej abecedy, potom je výhodné ako dátovú štruktúru pre A zvoliť lexikografický strom (kľúče budú riadky T). Časová zložitosť bude $O(n'+m'+r)$, kde n' (resp. m') je súčet dĺžok riadkov v súbore T (resp. S).

20.2 Výpočet $lis(Z)$

Najskôr ukážeme triviálne riešenie pomocou dynamického programovania, ktoré pobeží v čase $O(r^2)$. To neskôr zlepšíme na $O(r\log r)$.

Budeme vypĺňať tabuľku $A[1..r,1..r]$, kde na políčku $A[i,j]$ je najmenšie číslo x také, že x je posledný prvok nejakej rastúcej podpostupnosti dĺžky j postupnosti $Z_1 Z_2 \ldots Z_i$ (zoberieme

prvých i čísel postupnosti Z a pozrieme sa akým najmenším číslom môže končiť jej rastúca podpostupnosť dĺžky j). Okrajové prípady vyriešime $A[i,j] = \infty$ ak taká postupnosť neexistuje a $A[i,0] = -\infty$.

Príklad: Zoberme si prvých 9 čísel postupnosti $Z = 9,10,6,3,1,7,4,2,8,\ldots$:

- pre $j = 1$ máme podpostupnosti $(9),(10),(6),\ldots,(1),\ldots$, teda $A[9,1] = 1$
- pre $j = 2$ máme podpostupnosti $(9,10),(6,7),(6,8),\ldots,(1,2),\ldots$, teda $A[9,2] = 2$
- pre $j = 3$ končia všetky podpostupnosti $(6,7,8),(3,4,8),\ldots$ číslom 8, teda $A[9,3] = 8$

Ako teraz $A[i,j]$ vypočítať? Posledný prvok postupnosti reprezentovanej $A[i,j]$ môže byť

- Z_i, pričom pred ním je rastúca podpostupnosť dĺžky $j-1$ postupnosti $Z_1 Z_2 \ldots Z_{i-1}$. Z_i pritom na jej koniec vieme pripojiť len vtedy, ak existuje nejaká postupnosť dĺžky $j-1$, ktorá končí číslom menším ako Z_i, čo vieme ľahko zistiť z $A[i-1,j-1]$, kde máme informáciu o postupnosti s najmenším číslom na konci.
- Z_k pre nejaké $k < i$. V takom prípade Z_i nevyužijeme a výsledok pre prvých $i-1$ čísel máme v $A[i-1,j]$.

```
1  A[0,0] = -∞;
2  for (i=1; i≤r; i++) {
3    A[i,0] = -∞;
4    A[0,i] = ∞;
5  }
6  for (i=1; i≤r; i++) {
7    for (j=1; j≤r; j++) {
8      if (A[i-1,j-1] < Z_i)
9        A[i,j] = min(Z_i, A[i-1,j]);
10     else A[i,j] = A[i-1,j];
11   }
12 }
```

Z takto vyplnenej tabuľky potrebujeme zistiť najväčšie k, pre ktoré existuje k-prvková podpostupnosť, teda $lis(Z) = max\{k : A[r,k] < \infty\}$.

Poďme teraz zlepšiť časovú zložitosť. Zjavne platí $A[i,j] \leq A[i,k]$ pre $j < k$ (ináč by sme postupnosť zodpovedajúcu $A[i,k]$ skrátili o $k-j$ prvkov a dostali lepšiu j prvkovú postupnosť, rovnosť nastane len ak sú obidve hodnoty rovné ∞). Z toho vyplýva, že čísla v každom riadku A budú v neklesajúcom poradí. Pozrime sa teraz v koľkých číslach sa budú líšiť dva po sebe idúce riadky. Prvých niekoľko čísel riadku $i-1$ bude menších ako Z_i, označme si najvyšší index takéhoto čísla k. V prvých k iteráciach cyklu z riadku 7 bude platiť $A[i-1,j] < Z_i$, teda min na riadku 9 vyberie hodnoty z predchádzajúceho riadku tabuľky. V nasledujúcej iterácii už bude platiť $A[i-1,j-1] < Z_i \leq A[i-1,j]$, teda do $A[i,j]$ sa priradí hodnota Z_i. V ďalších iteráciach nebude splnená podmienka z 8. riadku, teda budeme len kopírovať hodnoty z predchádzajúceho riadku tabuľky. Z toho vyplýva, že jediná zmena medzi dvomi po sebe idúcimi

riadkami bude v $k+1$. stĺpci. Keďže čísla sú v riadkoch utriedené, index k môžeme hľadať binárnym vyhľadávaním.

```
1 A[0] = −∞;
2 for (i=1; i≤r; i++)
3   A[i] = ∞;
4 for (i=1; i≤r; i++) {
5   k = najvacsi index taky, ze A[k] < Z_i;
6   A[k+1] = Z_i;
7 }
```

Tento algoritmus nám teda spočíta dĺžku a posledný prvok najdlhšej rastúcej podpostupnosti. Ak potrebujeme zrekonštruovať celú podpostupnosť, použijeme pomocné pole ukazovateľov na dvojice $B[1..r]$. Prvý člen dvojice bude Z_i pre nejaké i, teda prvok podpostupnosti, ktorý patrí na príslušné miesto. Druhý člen bude ukazovateľ na zvyšok postupnosti. Zakaždým po priradení na riadku 6 si do $B[k+1]$ uložíme dvojicu (Z_i, kópia ukazovateľa z $B[k]$). Celú podpostupnosť potom máme uloženú ako spájaný zoznam v $B[lis(Z)]$ od posledného prvku.

V druhej fáze algoritmu sme r-krát volali binárne vyhľadávanie na r prvkoch. Sčítaním časových zložitostí obidvoch fáz teda dostávame odhad $O((n+m)\log m+r+r\log r)$. Bez ujmy na všeobecnosti budeme predpokladať $n > m$. Ďalej r je počet nejakých dvojíc, kde počet možností pre prvý člen je n a pre druhý člen m, teda môžeme použiť ohraničenie $0 \leq r \leq nm$. Potom platí aj $\log r \leq \log n^2 = 2\log n$. Z toho vyplýva odhad časovej zložitosti $O((n+r)\log n)$.

Pozrime sa teraz na očakávanú hodnotu r. Platí $P(S[i] = T[j]) = \frac{1}{\sigma}$, pretože k ľubovoľne zvolenému znaku $S[i]$ existuje práve jeden znak zo σ, ktorý sa s ním rovná (za predpokladu, že všetky znaky sú v T zastúpené rovnako). Potom

$$E(r) = \sum_{\forall i,j} P(S[i] = T[j]) = \frac{nm}{\sigma}$$

V priemernom prípade teda platí odhad časovej zložitosti $O(\frac{nm}{\sigma}\log n)$.

21 Zrýchlenia dynamického programovania na výpočet editačnej vzdialenosti

21.1 Ukkonenon algoritmus: Zrýchlenie pre veľmi podobné reťazce

Ukážeme Ukkonenov algoritmus z roku 1985, ktorý pracuje v čase $O(nD)$, kde $D = d_E(S,T)$. Uvažujeme základnú definíciu editačnej vzdialenosti, v ktorej má každá operácia cenu 1. V grafovej reprezentácii dynamického programovania má každá zvislá a vodorovná hrana cenu 1 a uhlopriečne hrany majú cenu 1 alebo 0 podľa toho, či sa príslušné znaky S a T zhodujú. Hľadáme najkratšiu cestu z $(0,0,)$ do (m,n). Ak $d_E(S,T) = D$, môžeme použiť najviac D vodorovných alebo zvislých hrán.

Očíslujme uhlopriečky v matici dynamického programovania (resp. v grafe) tak, že uhlopriečka k obsahuje políčka $A[i,j]$ pre $k = j - i$. Optimálne zarovnanie začína na uhlopriečke 0. Každá uhlopriečna hrana necháva číslo uhlopriečky rovnaké, každá zvislá a vodorovná ho mení o jedna. Nakoľko najkratšia cesta obsahuje najviac D vodorovných a zvislých hrán, bude sa celá vyskytovať v páse uhlopriečok $-D,\ldots,D$.

Rozhodovací problém. Najskôr vyriešime rozhodovací problém, v ktorom máme dané reťazce S a T a hodnotu k a chceme vedieť, či $d_E(S,T) \leq k$. Uvažujme všetky cesty z $(0,0)$ do (m,n), ktoré nevybočia z pruhu uhlopriečok $-k,\ldots,k$. Z týchto ciest vieme nájsť najkratšiu jednoduchou modifikáciou dynamického programovania, kde pre každý vrchol budeme v rekurencii uvažovať len tých z troch predchodcov, ktorí sú vo vnútri pruhu. V každom stĺpci matice počítame najviac $2k+1$ hodnôt, celková zložitosť je teda $O(kn)$.

Ak nakoniec dostaneme hodnotu $A[m,n] \leq k$, odpoveď na problém je áno, lebo sme našli aspoň jedno zarovnanie s cenou menšou ako k. Toto zarovnanie musí byť aj optimálne, lebo zarovnania, ktoré sme neuvažovali (tie, ktoré vybočia z nášho pruhu), majú cenu väčšiu ako k. Teda vieme, že $d_E(S,T) = A[m,n]$.

Ak naopak dostaneme hodnotu $A[m,n] > k$, odpoveď je nie, lebo všetky zarovnania (v pruhu aj mimo pruhu) majú cenu väčšiu ako k. Súčasne však nepoznáme cenu optimálneho zarovnania, vieme iba, že $k < d_E(S,T) \leq A[m,n]$, lebo optimálne zarovanie môže vyjsť mimo pruh.

Hľadanie hodnoty D. Ak vopred nepoznáme hodnotu k, ktorú by sme mohli použiť pri rozhodovacom probléme, môžeme ju nájsť postupom podobným na binárne vyhľadávanie. Začneme s hodnotou $k = 1$ a vždy keď odpoveď na rozhodovací problém je nie, hodnotu k zdvojnásobíme. Keď narazíme na prvé k^*, pre ktoré dostaneme odpoveď áno, môžeme skončiť, lebo zároveň sme spočítali aj $D = d_E(S,T)$ a optimálne zarovnanie. Vieme, že $k^*/2 < D \leq k^*$ a teda $k^* < 2D$. Ak čas potrebný na rozhodovací problém je najviac ckn pre nejakú konštantu c, čas pre hľadanie D bude

$$\sum_{i=0}^{\log_2 k^*} c2^i n < 2k^* cn < 4Dcn = O(Dn).$$

Hodnota D je najviac $\max(m,n)$ a teda čas tohto algoritmu nebude nikdy asymptotický horší ako čas základného dynamického programovania. Môže však byť oveľa lepší pre veľmi podobné reťazce.

21.2 Technika "štyroch Rusov"

Teraz predstavíme ďalšie zrýchlenie dynamického programovania na výpočet editačnej vzdialenosti. Základná myšlienka je rozdeliť problém na malé podproblémy, predpočítať riešenie každého možného podproblému a potom použiť tieto predpočítané riešenia na zrýchlenie algoritmu. Túto techniku vymysleli Arlazarov, Dinic, Kronrod and Faradzev v roku 1970, na editačnú vzdialenosť ju adaptovali Masek a Paterson v roku 1980. Bille and Farach-Colton (2008) tento prístup rozšírili pre veľké abecedy.

Predpokladajme, že vstupné reťazce sú rovnako dlhé $|S| = |T| = n$ a veľkosť abecedy je 2. Maticu A dynamického programovania rozdelíme na štvorce (bloky) veľkosti $t \times t$, tak aby sa susedné bloky prekrývali jedným stĺpcom a jedným riadkom. Každý blok bude tvoriť jeden podproblém. Uvažujme blok s ľavým horným rohom v políčku (i,j). Vstupom pre podproblém bude prvý riadok a prvý stĺpec bloku ako aj príslušné časti reťazcov S a T, t.j. dvojica vektorov $x_1 = (A[i,j..j+t-1], A[i+1..i+t-1,j])$ a $x_2 = (S[i+1..i+t-1], T[j+1..j+t-1])$. Výstupom je vektor $y = (A[i+t-1,j+1..j+t-1], A[i+1..i+t-2,j+t-1])$ obsahujúci

posledný riadok a stĺpec bloku. Uvedomme si, že vstup jednoznačne určuje výstup, lebo pri výpočte vnútra bloku potrebujeme len hodnoty z hranice bloku a podreťazce vstupných reťazcov v x_2.

Všetkých možných podproblémov je $(n+1)^{2t-1}2^{2t-2}$, lebo vektor x_1 má dĺžku $2t-1$ a každý jeho prvok je editačná vzdialenosť medzi nejakými prefixami S a T, čiže celé číslo od 0 do n. Podobne vektor x_2 obsahuje $2t-2$ znakov zo vstupnej binárnej abecedy. Tento počet je príliš velký a už pre $t=2$ by predpočítanie trvalo dlhšie ako bežné dynamické programovanie. Potrebujeme zredukovať veľkosť vstupu x_1.

Lema 4. Susedné políčka v stĺpci alebo riadku matice A sa líšia najviac o 1.

Dôkaz. Uvažujme dve susedné políčka v riadku. Z definície $A[i,j] \leq A[i,j-1]+1$, lebo existuje hrana dĺžky 1 z vrcholu $(i,j-1)$ do (i,j) a teda zo zarovnania $S[1..i]$ s $T[1..j-1]$ vieme vytvoriť zarovnanie $S[1..i]$ s $T[1..j]$. Podobne zo zarovnania $S[1..i]$ s $T[1..j]$ chceme vytvoriť zarovnanie pre $S[1..i]$ s $T[1..j-1]$. Odlíšime tri prípady podľa posledného stĺpca optimálneho zarovnania $S[1..i]$ s $T[1..j]$:

- Ak posledný stĺpec obsahuje $T[j]$ zarovnaný s medzerou. Tento stĺpec môžeme vynechať a dostaneme tak zarovnanie s cenou o jedna nižšou. Teda máme $A[i,j-1] \leq A[i,j]-1$.
- Ak posledný stĺpec obsahuje $S[i]$ a $T[j]$, vynecháme zo stĺpca $T[j]$. Tým cena vzrastie o 1 alebo ostane taká istá, podľa toho, či $S[i]=T[j]$ alebo nie. Teda máme $A[i,j-1] \leq A[i,j]+1$.
- Ak posledný stĺpec obsahuje $S[i]$ zarovnané s medzerou, nájdeme stĺpec obsahujúci $T[j]$ a z neho $T[j]$ zmažeme. Ak stĺpec ostane prázdny, zmažeme aj celý tento stĺpec. Táto zmena zase môže skóre buď nechať rovnaké, znížiť alebo zvýšiť o 1.

V každom prípade teda platí, že $A[i,j] \geq A[i,j-1]-1$. Podobne môžeme tvrdenie dokázať pre stĺpec.

Táto lema nám dáva návod ako úspornejšie zakódovať vektor x_1: pomocou hodnoty $A[i,j] \in \{0,1,\ldots,n\}$ a vektoru x'_1 hodnôt rozdielov medzi susednými hodnotami v prvom riadku resp. stĺpci $A[i,j+k]-A[i,j+k-1]$ a $A[i+k,j]-A[i+k-1,j] \in \{0,1,-1\}$. Teda možných vektorov x_1 je najviac $(n+1)3^{2t-2}$. Podobne zadefinujme vektor y' rozdielov na výstupe $A[i+t-1,j+k]-A[i+t-1,j+k-1]$, $A[i+k,j+t-1]-A[i+k-1,j+t-1]$.

Teraz zmeníme definíciu podproblému tak, že nebudeme z (x_1,x_2) počítať y, ale z (x'_1,x_2) budeme počítať y' nasledovným postupom:

- Do ľavého horného políčka bloku dáme hodnotu 0 (namiesto $A[i,j]$, ktoré nepoznáme).
- Z vektora rozdielov x'_1 spočítame prvý riadok a prvý stĺpec.
- Pomocou dynamického programovania vyplníme zvyšok bloku. Políčko $(i+p,j+q)$ bude obsahovať hodnotu $A[i+p,j+q]-A[i,j]$.
- Z posledného riadku a stĺpca spočítame vektor rozdielov y', v ktorých člen $A[i,j]$ vypadne.

Tento algoritmus trvá $O(t^2)$ pre každý blok daný vstupom (x'_1, x_2) a máme $3^{2t-2}2^{2t-2}$ možných blokov. Teda celkový čas predspracovania je $O(t^2 6^{2t})$. Výsledky pre jednotlivé bloky uložíme do poľa indexovaného vstupom prepočítaným na celé číslo. Pre daný vstup vieme v čase $O(t)$ spočítať index podproblému a nájsť riešenie v poli.

Po predspracovaní algoritmus funguje nasledovne:

- Vyplň nultý riadok a stĺpec matice, spočítaj rozdiely medzi susednými políčkami.
- Pre jednotlivé bloky matice nájdi vstupné rozdiely x'_1 spočítané v predchádzajúcich blokoch a v poli nájdi spočítané výstupné rozdiely.
- Na konci spočítaj z rozdielov hodnoty v poslednom stĺci matice a vypíš $A[m,n]$.

Počet blokov je n^2/t^2, na každý blok potrebujeme čas $O(t)$. Teda celkový čas je $O(n^2/t + t^2 6^{2t})$, čo pre $t = \log_6(n)/2$ je $O(n^2/\log n)$.

Poznámky. Ak vieme vstup pre jeden blok uložiť do registra, nájdenie indexu riešenia sa dá spraviť v čase $O(1)$ a celková zložitosť bude $O(n^2/\log^2 n)$. Algoritmus sa dá použiť aj pre abecedy veľkosti väčšej ako 2, ale dostávame zložitosť $O(n^2/\log_\sigma n)$. Naopak, algoritmus sa bez zmeny nedá použiť, ak operáciam v definícii editačnej vzdialenosti priradíme všeobecné váhy.

21.3 Prehľad algoritmov na výpočet editačnej vzdialenosti

Algoritmus	Problém	Čas	Poznámka
Základné dyn. prog.	l'ub. váhy	$O(mn)$	
Hirschbergov alg.	l'ub. váhy	$O(mn)$	$O(n)$ pamäť
Hunt-Szymanski	lcs	$O((n+r)\log n)$	$0 \le r \le mn$
Ukkonenov alg.	d_E/lcs	$O(nD)$	$0 \le D \le \max(m,n)$
Four Russians	d_E/lcs	$O(mn/\log n)$	$\sigma = O(1)$

Algoritmy Hunt-Szymanski a Ukkonenov sú adaptívne, lebo sa adaptujú na vlastnosti vstupu: hoci v najhoršom prípade ich čas je rovnaký alebo ešte horší ako čas základného algoritmu, pre niektoré triedy vstupov bežia oveľa rýchlejšie.

22 Hľadanie približných výskytov vzorky podľa editačnej vzdialenosti

Hľadáme pozície i také, že existuje $j \le i$ t.ž. $d_E(P, T[j,i]) \le k$

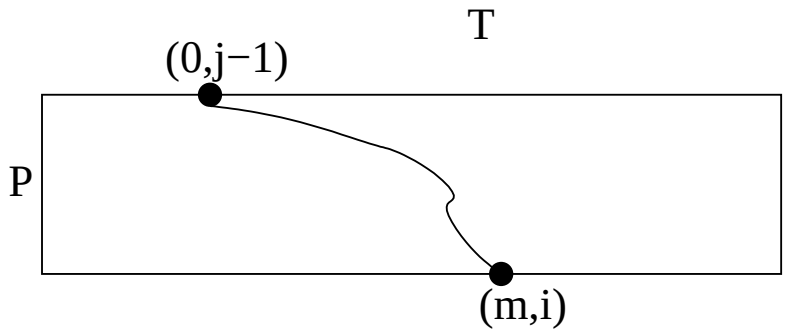
Doteraz sme si predstavili riešenia podobných problémov:

$d_E(P, T[i-m+1..i]) = 0$	$O(n)$	KMP
$d_H(P, T[i-m+1..i]) \le k$	$O(nk)$	sufixový strom + lca

Možné riešenia nášho problému:

- dynamické progamovanie $O(mn)$
- kombinované dynamické progamovanie, sufixové stromy a lca $O(nk)$ (nebudeme robiť)

Pripomeňme si, že hľadanie zarovnania pri výpočte editačnej vzdialenosti môžeme reprezentovať aj ako hľadanie najkratšej cesty v grafe z vrcholu $(0,0)$ do (m,n). Tentokrát ale chceme zarovnať reťazec P ku krátkemu kusu T, zmeníme teda aj graf.



- pridáme na začiatok vrchol s
- pridáme hrany $(s,(0,j))\forall j = 0 \ldots n$ s cenou 0
- cesta zo s do (m,i) zodpovedá zarovnaniu $T[j \ldots i]$, kde j je taká, že cesta začína hranou $(s,(0,j-1))$
- spustíme dynamické programovanie, ako sme mali predtým, až na prvý riadok, lebo tam použijeme hrany $(s,(0,j))$ s cenou 0
- vypíšeme i také, že $A[m,i] \leq k$

Takto budeme vypisovať konce, ak chceme nájsť začiatky, tak spustíme tento algoritmus na reverz slova

Patologický prípad

$T = a^n$, $m = n/2$

$P = a^m$, $k = n/2$

Výskytom je každá dvojica (i,j), z toho vyplýva kvadratický počet výskytov. Toto pre nás nie je efektívne a čo koby sme pre každý koniec vypísali najlepší začiatok?

$B_0[i,j] = l$, t.ž. l je posledný vrchol grafu 0, kde prechádza min, cestou z s do (i,j). Pokiaľ nás zaujíma aj konkrétne zarovnanie a nie len jeho cena, bude v poli $A[i,j]$ cesta z s do (i,j), v $B[i,j]$ budeme ukladať poslednú hranu (m,j) pre $A[n,j] \leq k$.

Nepotrebujeme si pamätať celú tabuľku so všetkými vypočítanými hodnotami, ale stačia nám posledné 2 riadky, z ktorých vieme dopočítať zvyšné nasledujúce. Teda pamäťová zložitosť je $O(n+m)$.

23 Úvod do bioinformatiky

Bioinformatika je odvetvie na hranici informatiky a biológie, ktoré skúma matematické modely skúmaných biologických problémov.

1. reťazce = sekvencie
2. stromy
3. veľa pravdepodobnosti a štatistiky

23.1 Reťazce DNA

$\Sigma = \{A, C, G, T\}$

DNA je molekula s genetickou informáciou. L'udský genóm(DNA) pozostáva z 24 reťazcov s celkovou dĺžkou $\sim 10^9$. DNA = digitálny zápis, ktorý obsahuje predpisy na výrobu proteínov. Sú uložené v tabuľke, ktorú naývame **genetický kód**: $\{A, C, G, T\}^3 \to$ aminokyselina

Príklad: $ATG \to M$

DNA okrem proteínov hovorí aj kedy, kde a v akých množstvách sa má ktorý proteín tvoriť. Táto oblasť je stále predmetom skúmania, lebo stále ešte nepoznáme do detailov, ako tento mechanizmus funguje.

23.2 RNA

$\Sigma = \{A, C, G, U\}$

Má dvojakú funkciu, buď je to enzým alebo potom môže mať funkciu dočasne uchovávať informáciu z DNA

23.3 Proteíny

Proteín je reťazec nad 20 prvkovou abecedou - 20 rôznych aminokyselín. V l'udskom organizme je asi 20000 rôznych proteínov.

Dĺžka jedného proteínu je $\sim 100 - 1000$. Proteíny by sme mohli nazvať ako HW bunky.

Majú rozličné ulohy, napr. enzýmy - katalyzátory, dôležité pre štruktúru bunky, signalizácia a iné.

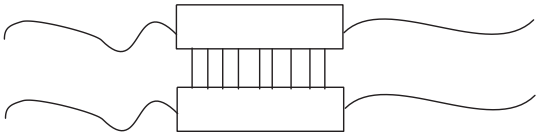
23.4 Evolúcia

Mutácia mení DNA a z toho pri evolúcii vznikajú rozdiely medzi rôznymi druhmi. Keďže je mutácia veľký zásah do organizmu, uchytia sa len tie, ktoré nrerobia nič zlé, alebo sú pozitývne pre jedinca.

Operácie ktoré menia DNA:

- substitúcia, insercia, delécia
- duplikácia, preusporiadanie veľkých blokov

Keď porovnáme DNA dvoch ľudí, zistíme, že sa líšia približne v 0.1% znakov.
človek a šimpanz sa líšia v 1% znakov $\rightarrow$ 6mil. rokov
človek a myš sa líšia v 30% znakov $\rightarrow$ 75mil. rokov



Z tohto vidíme, že existujú ostrovčeky podobností v mori iných sekvencií. Vidíme že v tých miestach pravdepodobne došlo k veľkej zmene. Taktiež z podobnosti môžeme dedukovať, že funkcie, ktoré sú veľmi dôležité pre život bunky nie sú zmutované a nachádzame aj 200 znakov dlhé rovnaké sekvencie.

$$\begin{array}{cccccc} A & T & C & G & G & A \\ A & - & C & G & G & A \end{array} \tag{4}$$

Jedna z hypotéz je, že napr. reťazce 4 vznikli zo spoločného predka inserciou / deléciou. Z tohto príkladu si môžeme uvedomiť, že **zarovnanie** = **hypotéza o evolúcii**. Preto najdôležitejší problém, ktorý rieši bioinformatika je **lokálne zarovnanie**.

24 Lokálne zarovnanie

Dané sú S a T, nájdi zarovanie medzi podslovami $S[i \dots j]$ a $T[p \dots q]$ s čo najväčšou podobnosťou k.

Použitie lokálneho zarovnania:

- hypotéty o evolúcii
- určovanie funkcií na základe podobnosti
- na základe podobnosti nájsť zachované dôležité regióny DNA

Toto zarovnanie zodpovedá ceste z $(i-1, p-1)$ do (j,q). Ako modifikovať graf?
Pridáme hrany $(s,(i,p))\forall i = 0 \dots m, \forall p = 0 \dots n$ s cenou 0. Naša výsledná rekurencia:

$$A = \begin{cases} 0 \\ A[i-1, j-1] + w(S[i], T[i]) \\ A[i, j-1] + w(-, T[j]) \\ A[i-1, j] + w(S[i], -) \end{cases} \tag{5}$$

Konce zarovnaní (j,q) t.ž. $A[j,q] \geq k$.

Náš cieľ: Chceme vypísať neprekrývajúce sa rozdielne zarovnania matice.

24.1 Lokálne zarovnania s dlhými inzerciami a deléciami

Uvedomme si, že po istej sekvencie mohol byť vložený iný úsek genetickej informácie a teda mali by sme to brať ako jednu inzerciu. Preto musíme ešte pridať hrany:

$$(i,j) \to (i,k)\, j < k, \text{ váha } 1$$

$$(i,j) \to (k,j) i < k, \text{ váha } 1$$

Čo bude teraz rekurencia?

$$A[i,j] = \max \begin{cases} A[i-1,j-1] + w(S[i],T[j]) \\ \max_{k<i} A[k,j] - 1 \\ \max_{k<j} A[i,k] - 1 \end{cases} \tag{6}$$

Časová zložitosť je O(m.n(m+n))

Lepšie riešenie: majme 3 kópie každého vrcholu $(i,j)_u$ - uhlopriečka, $(i,j)_v$ - vodorovne, $(i,j)_z$ - zvislo.

Kam sa dostaneme z (i,j)?

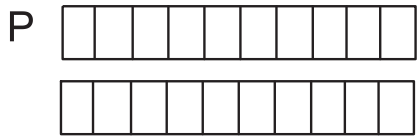
$$\begin{array}{llll} (i,j)_u & \to & (i,j+1)_v & -1 \quad \text{začiatok insercie alebo delécie} \\ (i,j)_z & \to & (i,j+1)_v & -1 \quad \text{začiatok insercie alebo delécie} \\ (i,j)_{u|v} & \to & (i+1,j)_z & -1 \quad \text{začiatok insercie alebo delécie} \\ (i,j)_v & \to & (i,j+1)_v & 0 \quad \text{pokračovanie} \\ (i,j)_z & \to & (i+1,j)_z & 0 \quad \text{pokračovanie} \\ (i,j)_{u|v|z} & \to & (i+1,j+1)_u & w(S[i],T[i]) \end{array} \tag{7}$$

Potom počítame maximumum z 9 možností a teda dostávame časovú zložitosť: O(m.n) lebo máme $9mn$ hrán.

25 Heuristické zrýchlenia pre editačnú vzdialenosť

Vypočet editačnej vzdialenosti slúži na hľadanie lokálneho zarovnania. Základný algoritmus využíva dynamické programovanie a beží v čaovej zložitosti O(m.n). V niektorých prípadoch vieme pôvodný algoritmus vylepšiť. Musíme si uvedomiť, že editačnú vzdialenosť budeme rátať pre dlhé reťazce, a teda zložitosť O(n.m) je pre nás neprijateľná. Viď tabuľka **??**

n	čas
100	0.0008 s
1000	0.08 s
10000	8 s
100000	13 min
10^6	22 hod
10^7	3 mesiace
10^8	25 rokov



Obr. 44: Zarovanie

25.1 Baeza-Yates & Perleberg 1992

Približné výskyty vzorky s $d_E \leq k$

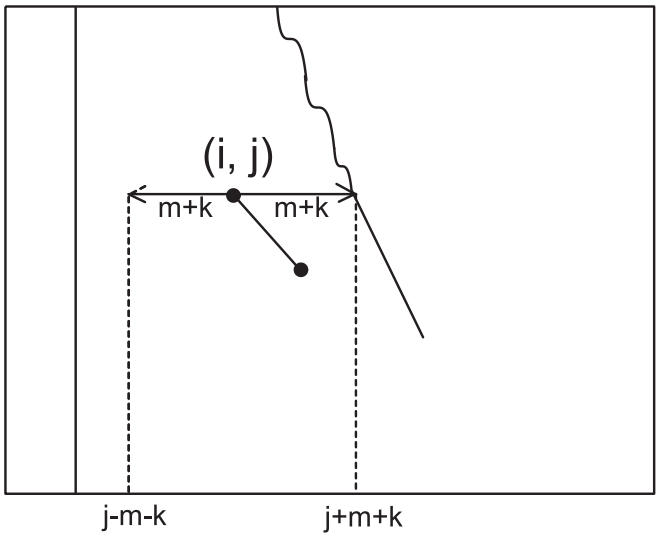
Rozdelíme P na úseky dĺžky $\lfloor \frac{m}{k+1} \rfloor = r$

P= mno6ina kúskov

Každé zarovnanie má aspoň 1 kúsok bez zmeny. Zoberieme si množinu kúskov a nájdeme si všetky ich výskyty v **P** použijeme Aho - Corasickov algoritmus.

Časová zložitosť: O($n+m+|I|$)

Vylepšeie: Budeme rátať dyn. programovaním len tam, kde sa môže vyskytovať.



Zoberieme kúsok začínajúci na (i, j). Pre každý výskyt $(i, j) \in I$ hľadáme približné výskyty P v $T[j-m-k, j+m+k]$ dyn. progamovaním v čase O(m^2)

Zložitosť spolu O($n+m^2|I|$) $\Rightarrow$ budeme analyzovať

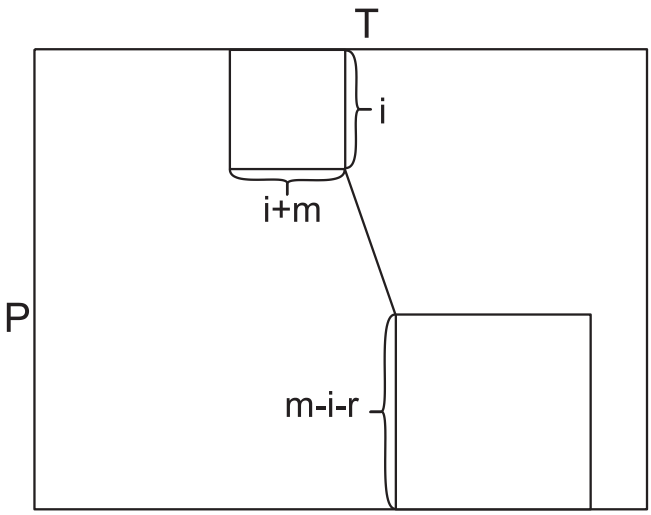
Praktické zlepšenia: zlučovanie prekrývajúcich sa intervalov.

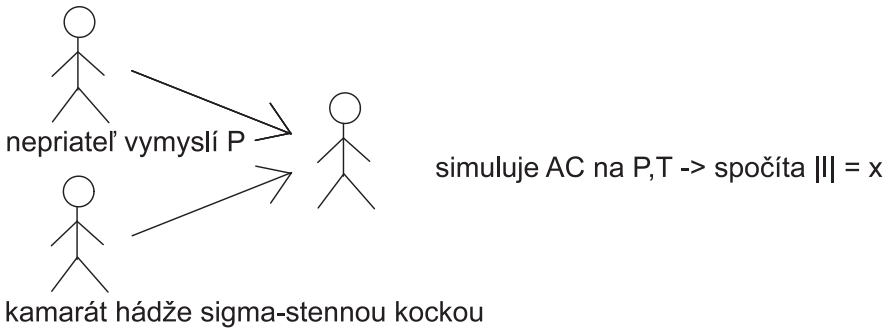
Týmto sme dosiahli, že nám stačí prehľadať menšie časti. Avšak po použití zlúčenia máme časovú zložitosť O($n+m^2$), to je taká istá ako má dynamické programovanie.

Budeme analyzovať $|I|$ v priemernom prípade:

$$\frac{1}{\sigma^n} \sum_{T \in \Sigma^n} |I_{(P.T)}|$$

Možeme ich použiť viackrát a tak dostaneme **E**(X). Prečo práve takto? Lebo v takomto príde je to zadefinované aj keď máme nerovnomerne rozdelenú pravdepodobnosť na $\Sigma^y n$. Mô-





žeme použiť linearitu **E**(X+Y) = **E**(X)+**E**(Y) $\mathbf{E}(X) = \sum_{x \text{ z oboru hodnôt } X} P(X = x)x$

$$X_{i,p} = \begin{cases} 1 & \text{ak } P = T[i \ldots i+p-1], p \in P \\ 0 & \text{inak} \end{cases}$$

$$X = \sum_{i=1}^{n} \sum_{p in P} X_{ip}$$

$$E(X) = \sum_{i} \sum_{p} E(X_{i,p})$$

$$E(X_{ip}) = P(X_{ip} = 0).0 + P(X_{ip} = 1).1$$

$$P(X_{ip} = 1) = P(T[i] = p[i] \wedge T[i+1] = p[i-1] \wedge [i+j-1] = p[r]) =$$

$$= \prod_{j=1}^{r} \underbrace{P(T[i+j-1] = p[j])}_{\frac{1}{\sigma}} \leq \sigma^{-r}$$

výsledok: $E(X) = \sigma^{-r}.n.(k-1)$

priemerný prípad je $O(n + n(k+1)\sigma^{-r}m^2)$ pravdepodobnosť, že $k < m/(6log_{\sigma}m)$ $r = \lfloor \frac{m}{k+1} \rfloor \geq \frac{m}{2k} \geq \frac{m}{2m/6\log_{\sigma} m} = 3.\log_{\sigma} m$

$\sigma^{-r} \leq \sigma^{-3\log_{\sigma} m = \frac{1}{m^3}}$

Celkový čas v priemernom prípade je O(n). $10, T = 4$

$$\begin{array}{lll} 6m = & 10 & T = 4 \quad k \leq 1 \\ & 100 & \quad k \leq 5 \\ & 1000 & \quad k \leq 33 \end{array} \tag{8}$$

$$\begin{array}{l} O(m.n) \\ O(k.n) \end{array} \} \text{v najhoršom prípade} \tag{9}$$

hľadáme nejaké zarovnania: $w(a,a) = 1; w(a,b) = -1; w(a,-) = w(-,a) = -\infty$
lokálne zarovnanie s cenou $\geq k$.
Hľadáme hrany dĺžky w, ktoré sa zhodujú:

$$\begin{array}{llll} (i,j) & S[i \ldots i+w-1] & = & S[j \ldots j+w-1] \\ \underbrace{1-11}_{1} & \underbrace{1-11}_{1} & & \underbrace{1-11}_{1} \end{array} \tag{10}$$

$$\begin{array}{ccccc} A & G & T & C & A \\ A & G & T & C & G \end{array} \tag{11}$$

Nájdeme množinu dvojíc (i, j), ktoré sa zhodujú (v jadrách) $S[i \ldots i+w-1] = S[j \ldots j+w-1]$. Pokúsime sa ísť po uhlopriečke a rozšíriť zarovnanie. S rozšírením skončíme, keď je cena rozšírenia menšia alebo rovná ako cena doterajšieho najlepšieho rozšírenia. Ak je skoré vačšie ako k vypíšeme.

Ako to môžeme robiť?

- sufixové stromy O$(n+m.|I|)$
- Aho-Corasick algoritmus O$(m.n+n)$

Keď budeme analyzovať O$(|I|.\min(m,n))$ zistíme, že v praxi O$(|I|)$ skôr závisí od T.

Prečo nenájdeme zarovnanie so skóre $\geq k$?

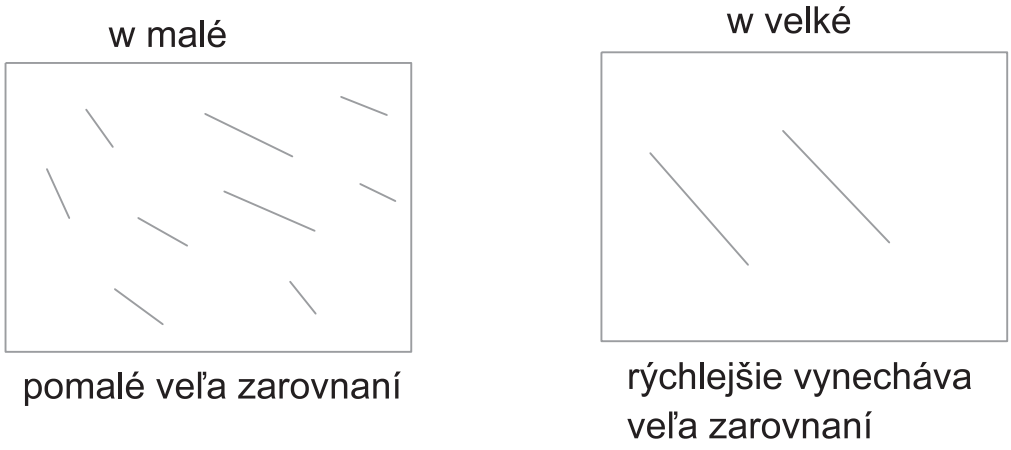
- lebo neobsahuje jadro w
- obsahuje jadro, ale skončíme rozširovanie priskoro
- závisí od T

25.2 BLAST

Algoritmus bol vtvorený Stephenom Altschulom a spoločníkmi v roku 1990. Vytvoríme náhodné zarovnanie dĺžky L.

skóre v sĺpci:

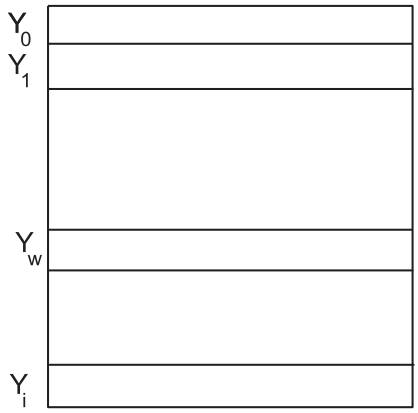
- $i+1$ s pravdepodobnosťou p
- $i-1$ s pravdepodobnosťou $(1-p)$



Hodíme L-krát mincou a tým získame zarovnanie. Potom zistíme, či sa v získanom zarovnaní nachádza za sebou w jednotiek. Následne vypočítame pravdepodobnosť P(odpoveď je áno) = 9L . Algoritmus využíva dynamické programovanie.

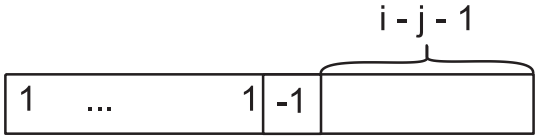
Y = počet + 1 na začiatku postupnosti $Y = \{0,1,\ldots,L\}$

$$q_i = \begin{cases} 0 & i < w \\ p^w & i = w \\ p^w + \sum_{j=0}^{w-1} p(1-p)q_i - j - 1 & i > w \end{cases} \tag{12}$$



Obr. 45: Spočítame si v každom prúžku, koľko z nich obsahuje jadro

$$\begin{aligned} q_i &= \textstyle\sum_{j=0}^{i}((Y=i).P(\text{ jadro } |Y=j) = \\ &= \textstyle\sum_{j=0}^{w-1} P(Y=i).P(\text{ jadro } |Y=j) + P(Y \geq w).P(\text{ jadro } |Y \geq w) = \\ &= \textstyle\sum_{j=0}^{w-1} p^i(1-p).q_{i-j-1} + p^w E(|I|) = m.n.P((i,j) \in I) = m.\sigma^{-w} \end{aligned} \tag{13}$$



Obr. 46: na i - j - 1 môže byť jadro

26 Viacnásobné zarovnanie

Základný problém: dané sú reťazce: $S_1, S_2, ..., S_k$, $n = \sum_{i=1}^{k} |S_i|$. Nájdite zarovnanie s optimálnou cenou.

Príklad: Zarovnanie troch reťazcov

$$\begin{pmatrix} A & C & A & G & T & - & A \\ A & - & A & C & T & - & C \\ G & C & A & C & T & G & A \end{pmatrix}$$

Motivácia: bioinformatika

- hypotéza o evolúcií: rovnaké znaky v jednom stĺpci pravdepodobne pochádzajú od spoločného predka
- zistiť, či ide o inzerciu alebo deléciu
- nájsť pozície, ktoré sa málo menia (takéto pozície majú často dôležitú biologickú funkciu a mutácia znamená vyhynutie organizmu, teda prírodný výber ich udržuje viac-menej nemenné)

Cena zarovnania: Cenu $w(A)$ viacnásobného zarovnania A, môžme určiť rôznymi spôsobmi, my si uvedieme tieto tri:

1. Sum-of-pairs(SP) - cenu určíme ako súčet cien všetkých dvojíc riadkov viacnásobného zarovnania. Presnejšie
$$w(A) = \sum_{i<j} d_A(S_i, S_j)$$
kde d_A je cena zarovnania[5] indukovaného viacnásobným zarovnaním A. Cena príkladu $w(A) = d_A(S_1,S_2) + d_A(S_1,S_3) + d_A(S_2,S_3) = 3+3+4 = 10$.

2. Porovnanie s konsenzom. Konsenzus C_A viacnásobného zarovnania A je reťazec, s dĺžkou rovnakou ako zarovnania, ktorého znaky sú najčastejšie sa vyskytujúce znaky v danom stĺpci zarovnania. Potom cenu definujeme ako:
$$w(A) = \sum_{i=1}^{k} d_A(S_c, S_i)$$

[5] d_A je editačná vzdialenosť a cena substitúcie, inzercie, delécie je 1, v prípade dvoch pomlčiek cenu dodefinujeme ako 0

Konsenzus príkladu je C_A =ACACT-A a cena zarovnania $w(A) = d_A(C_A,S_1)+d_A(C_A,S_2)+d_A(C_A,S_3) = 1+2+2 = 5$

3. Využitie fylogenetického stromu - predpokladajme, že máme strom T o vývoji organizmov. Ku každej vstupnej sekvencii je priradený list stromu T. V každom vnútornom vrchole u chceme zostrojiť reťazce S_u, také že cena

$$w(A) = \sum_{(u,v)\in T} d_E(S_u,S_v)$$

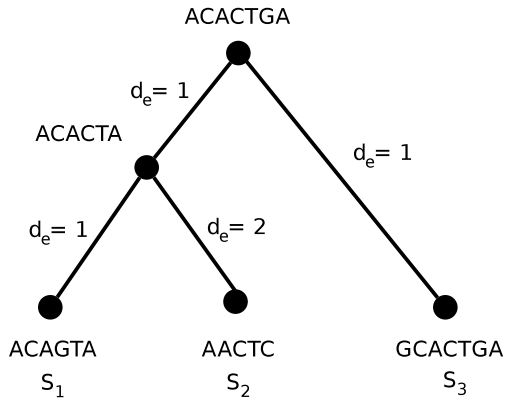
je minimálna.

Túto sumu budeme nazývať fylogenetická vzdialenosť - minimálny počet operácií, ktoré sú nutné aby sme upravili reťazec z nejakého neznámeho koreňa na listy.

Všimnime si, že v poslednej definícii sme zadefinovali cenu pre určitú množinu reťazcov, ale nie samotné zarovnanie. Ak chceme zostrojiť aj zarovnanie, použijeme nasledovný pomocný pojem.

Definícia: Nech T je strom. Vrcholu u priradíme reťazec S_u. Hovoríme, že zarovnanie A reťazcov S_u je konzistentné s T ak pre všetky hrany $(u,v) \in T$ platí $d_E(S_u,S_v) = d_A(S_u,S_v)$

Lema: Pre každý strom existuje konzistentné zarovnanie.



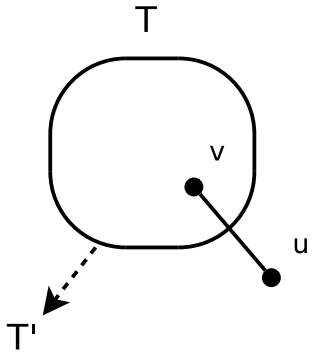
Obr. 47: Príklad stromu s editačnými vzdialenosťami susedných vrcholov

Ak teda poznáme strom a reťazce vo vnútroných vrcholoch z definície fylogenetickej vzdialenosti, podľa lemy zostrojíme zarovnanie reťazcov pre všetky vrcholy (vnútorné aj listy) a zmažeme reťazce pre vnútorne vrcholy, čím dostávame zarovnanie pre vstupné reťazce.

Dôkaz lemy: indukciou vzhľadom na počet vrcholov (reťazcov v zarovnaní)

1. báza: $k = 2$ 2 vrcholy - zoberieme optimálne zarovnanie dvoch reťazcov vzhľadom na editačnú vzdialenosť

2. $k > 2$ Máme strom T. Odoberme mu list u. $T' = T - u$. Z IP vyplýva, že existuje konzistentné zarovnanie pre T', nazvime ho A'. Označme v suseda u v T a A_p optimálne zarovnanie S_u a S_v. Teraz po stĺpcoch spojíme A' a A_p do A.



Obr. 48: Stromy T a T' v druhom kroku dôkazu lemy

$\binom{x}{y} \in A_p$	y vložíme pod x v A
$\binom{x}{-} \in A_p$	$-$ vložíme pod x v A
$\binom{-}{y} \in A_p$	y vložíme stĺpec $-$ do A a pod neho y

Stĺpce v A' s pomlčkou v S_v majú pomlčku v S_u. Takto zostrojené zarovnanie má cenu $w(A') + d_E(S_u, S_v)$

Poznámka: Konštruktívny dôkaz dáva návod na algoritmus v polynomiálnom čase.

26.1 Algoritmus pre SP

Dynamické programovanie: Tak ako v prípade zarovnania dvoch reťazcov tak aj v tomto prípade môžeme použiť dynamické programovanie. Zadefinujme si podúlohu pre zarovnanie troch reťazcov:

$$A[i,j,k] = \text{cena optimálneho zarovnania } S_1[1..i], S_2[1..j], S_3[1..k]$$

Taktiež si zadefinujme funkciu na ohodnotenie stĺpca v zarovnaní:

$v(a,b,c) = \text{cena stĺpca} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$, $a,b,c \in \Sigma \cup \{-\}$. Napríklad $v(A,A,C) = 2$, $v(-,-,A) = 2$.

Rekurencia:

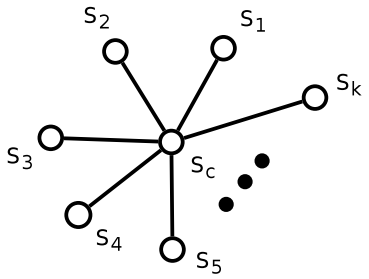
$$A[i,j,k] = min \begin{cases} A[i-1,j-1,k-1] + v(S_1[i], S_2[j], S_3[k]) \\ A[i-1,j,k-1] + v(S_1[i], -, S_3[k]) \\ A[i-1,j-1,k] + v(S_1[i], S_2[j], -) \\ A[i,j-1,k-1] + v(-, S_2[j], S_3[k]) \\ A[i,j,k-1] + v(-,-,S_3[k]) \\ A[i,j-1,k] + v(-, S_2[j], -) \\ A[i-1,j,k] + v(S_1[i], -, -) \end{cases}$$

Vo všeobecnom prípade pre k reťazcov dostávame $2^k - 1$ možností. Celková zložitosť algoritmu je $O((2(n+1))^k)$. V roku 2006 Elias ukázal, že viacnásobné zarovnanie je NP-ťažké, pre všetky 3 typy cien.

V praxi sa používajú rôzne heuristiky:

- orezávanie dynamického programovania
- progresívne algoritmy(rekurzívne rozdelím na dve časti a potom zarovnávam zarovnania[6])
- použitím kotvy - dostatočne dlhý spoločný reťazec - potom sa zarovnávajú reťazce samostatne pred kotvou a za ňou.

Aproximačný algoritmus pre SP cenu Určíme centrálny reťazec $S_c \in \{S_1, S_2, \ldots S_k\}$ taký, že $\sum_{i=1}^{k} d_E(S_c, S_i)$ je minimálna.



Obr. 49: Strom s centrálnym reťazcom S_c, z ktorého zostavíme konzistentné viacnásobné zarovnanie

Existuje zarovnanie konzistentné so stromom. V polynomiálnom čase zostrojíme viacnásobné zarovnanie A.

Pre indukovanú vzdialenosť d_A tiež platí trojuholníková nerovnosť

$$d_A(S_i, S_j) \leq d_A(S_i, S_c) + d_A(S_c, S_j)$$

Označme A^* optimálne zarovnanie. Potom pre aproximačný faktor platí:

$$\frac{w(A)}{w(A^*)} = \frac{\sum_{i \neq j} d_A(S_i, S_j)}{\sum_{i \neq j} d_{A^*}(S_i, S_j)}$$

Pre čitateľ platí

$$\sum_{i \neq j} d_A(S_i, S_j) \leq \sum_{i \neq j} (d_E(S_i, S_c) + d_E(S_c, S_j)) = 2(k-1) \sum_{i=1}^{k} d_E(S_i, S_c)$$

pre menovateľ

$$\sum_{i \neq j} d_{A^*}(S_i, S_j) \geq \sum_{i \neq j} d_E(S_i, S_j) \geq \sum_{i,j} d_E(S_i, S_c) = k \sum_{i=1}^{k} d_E(S_i, S_c)$$

z toho

$$\frac{w(A)}{w(A^*)} \leq \frac{2(k-1) \sum_{i=1}^{k} d_E(S_i, S_c)}{k \sum_{i=1}^{k} d_E(S_i, S_c)} = \frac{2(k-1)}{k} < 2$$

[6]Tiež NP-ťažký problém

27 Zostavovanie DNA sekvencií

27.1 Najkratšie spoločné nadslovo (shortest common superstring)

Vstup: $S_1, S_2, \ldots, S_k \in \Sigma^*$

Výstup: Najkratší reťazec S taký, že každé S_i je podslovo S.

Príklad: { ema, ma, mamu, mama, emu } a S =emamamuemu

Predpokladáme, že žiadne S_i nie je podslovo S_j ak $i \neq j$. Pretože, také slová môžme vynechať.

Uvedený problém je NP-ťažký. Aby sme to mohli ukázať, zadefinujme si k nemu rozhodovací problém:

Vstup: $S_1, S_2, \ldots S_k \in \Sigma^*$ a t

Výstup: Existuje spoločné nadslovo slov $S_1, S_2, \ldots S_k$ dĺžky najviac t? (áno/nie)

Veta 3. Rozhodovací problém pre najkratšie spoločné nadslovo je NP-ťažký.

Dôkaz: 1980 Galant, Maier, Storer

V dôkaze použijeme redukciu z nasledujúcej verzie Hamiltonovskej cesty:

Hamiltonovská cesta:

Vstup: Orientovaný graf s vrcholmi $1 \ldots n$, každý vrchol, okrem vrcholu n má aspoň 2 výstupné hrany.

Výstup: Existuje cesta 1 do n, ktorá prechádza cez každý vrchol práve raz?

Veta 4. Hamiltonovská cesta je NP-úplný problém.

Dôkaz vety 3 Musíme zostrojiť polynomiálny algoritmus, ktorý na vstupe dostane graf G a transformuje ho na slová $S_1, S_2, \ldots S_k$ a t také, že slová $S_1, S_2, \ldots S_k$ majú nadslovo dĺžky t práve vtedy keď G má hamiltonovskú cestu.

Algoritmus bude pracovať nad abecedou $\Sigma = \{1, 2, \ldots n\} \cup \{\overline{1}, \overline{2}, \ldots \overline{n-1}\} \cup \{¢, \$, \#\}$. Pre každý vrchol $v \in G$ a pre každú hranu z v do $w_0, w_1, \ldots w_{k-1}$ zostrojíme slová tvaru $\overline{v}w_i\overline{v}$ a $w_i\overline{v}w_{i+1 \bmod k}$. Presnejšie pre každý vrchol v zostrojíme množinu slov

$$A_v = \{\overline{v}w_i\overline{v} : i = 0 \ldots k-1\} \cup \{w_i\overline{v}w_{i+1 \bmod k} : i = 0 \ldots k-1\}$$

kde k je počet odchádzajúcich hrán z v. Definujme si štandardné nadslovo

$$S_{v,w_i} = \overline{v}w_i\overline{v}w_{i+1}\overline{v} \ldots \overline{v}w_{i-1}\overline{v}w_i$$

Vidíme, že slovo S_{v,w_i} je nadslovo množiny slov A_v. Pre každý vrchol z množiny $v \in \{2, 3, \ldots n-1\}$ definujme konektor

$$C_v = \{v\#\overline{v}\}$$

a nakoniec množinu terminálov

$$T = \{¢\#\overline{1}, n\#\$\}$$

Výstupná množina slov bude

$$L = (\cup_{v=1\ldots n-1} A_v) \cup (\cup_{v=2\ldots n-1} C_v) \cup T$$

a hľadaná dĺžka slova $t = 2m + 3n$, kde m je počet hrán a n je počet vrcholov grafu G.

cesta $\Rightarrow$ nadslovo Máme hamiltonovskú cestu v G, ktorá prechádza vrcholmi v poradí $u_1, u_2, \ldots u_n$ ($u_1 = 1, u_n = n$). K tejto ceste zostrojme reťazec $S = ¢\#S_{u_1,u_2}\#S_{u_2,u_3}\#\ldots\#S_{u_{n-1},u_n}\#\$$. Ukážme, že takýto reťazec S je nadslovo slov z množiny L a jeho dĺžka je $2m + 3n$.

- Keďže S_{1,u_2} začína na $\overline{1}$ tak reťazec $¢\#\overline{1}$ je podreťazec S. Podobne aj S_{u_{n-1},u_n} končí na n, teda $n\#\$$ je podreťazec S.
- Hamiltonovská cesta obsahuje všetky vrcholy, potom S obsahuje všetky reťazce $S_{i,i+1}$ pre $i = 1, 2, \ldots n-1$. Vieme, že $S_{i,i+1}$ je nadslovo pre množinu A_i, teda S je nadslovo pre všetky A_i kde $i = 1, 2, \ldots n-1$.
- Je ľahké nahliadnuť, že konektor $u_{i+1}\#\overline{u_{i+1}}$ sa nachádza v S medzi $S_{u_i,u_{i+1}}\#S_{u_{i+1},u_{i+2}}$ a z toho vyplýva že aj konektor z C_i pre $i = 2, 3, \ldots n-1$ je podslovom S.

Dĺžka štandardného nadslova pre vrchol v s $k = d_{\text{out}}(v)$ výstupnými hranami je $2k + 2$[7] Dĺžka S je

$$\begin{aligned} |S| &= \sum_{i=1}^{n-1} (2d_{\text{out}} + 2) + (n-2) + 4 \\ &= 2m + 2(n-1) + (n+2) \\ &= 2m + 3n \end{aligned}$$

nadslovo $\Rightarrow$ cesta V L je $2m + n$ reťazcov a každý z nich má dĺžku 3. Keď ich poskladáme do nadslova môžu sa prekrývať najviac o 2 znaky, inak by boli zhodné. Nech sa prekrývajú o dva potom dĺžka nadreťazca je aspoň $2m + n + 2$. Ale konektory a terminály sa nemôžu prekrývať o 2, pretože nemajú na kraji mrežu. Teda sa môžu najviac o jeden. V množine A_v sú možné prekryvy iba v týchto prípadoch:

1. $w_i\overline{v}w_{i+1}$ a $\overline{v}w_{i+1}\overline{v}$
2. $\overline{v}w_i\overline{v}$ a $w_i\overline{v}w_{i+1}$

Konektory sa môžu prekrývať iba o jeden znak a to v týchto prípadoch:

1. $v\#\overline{v}$ a $\overline{v}w_i\overline{v}$
2. $w_i\overline{v}w_{i+1}$ a $w_{i+1}\#\overline{w_{i+1}}$

[7] Index pri w v štandardnom nadslove beží od i po $k-1$ a potom od 0 po i, čo je dokopy $(k-1-i+1)+(i-0+1) = k+1$.

Minimálna dĺžka nadslova bezohľadu na graf je $2m+2n+\underbrace{2(n-2)}_{konektory}+2 = 2m+3n$. Ale my máme slovo dĺžky $2m+3n$ a teda musí mať tvar

$$¢\#\ldots\#\underbrace{\ldots}_{S_{v,w_i}}\#\underbrace{\ldots}_{S_{w_i,z}}\#\ldots\#\$$$

Medzi mrežami musí byť S_{v,w_i} pretože za mrežou je bezprostredne $\overline{v}$ a dvojica $\#\overline{v}$ sa už nemôže nikde inde vyskytovať, lebo konektor tohto typu je len jeden. S_{v,w_i} musí byť v kuse, pretože inak by narástla dĺžka slova. Analogicky za blokom S_{v,w_i} musí nasledovať blok $S_{w_i,z}$. Nadslovo S_{v,w_i} reprezentuje hranu $(v,w_i) \in E$ a celý reťazec reprezentuje hamiltonovskú cestu v grafe.

27.2 Aproximácie

Greedy algoritmus

1. nájdi najväčší prekryv medzi dvoma slovami, t.j. najdlhšiu α takú, že $S_i = \alpha\beta$ a $S_j = \gamma\alpha$.
2. spoj S_i a S_j do $\gamma\alpha\beta$
3. opakuj 1. krok, kým nezostane iba jedno slovo

hľadanie α Vybudujme sufixový strom pre S_j. Hľadáme najdlhší prefix na ceste S_i v sufixovom strome, kde končí najdlhší sufix S_j.(najnižší taký vrchol). To vieme nájsť v $O(n)$.

Aby sme to však nemuseli robiť pre každé i a j zvlášť, majme zovšeobecnený sufixový strom pre $S_1, S_2, \ldots S_k$. Pre dané S_i hľadáme najhlbší prefix S_i, ktorý sufixom nejakého $S_j (j \neq i)$. Hľadáme teda najdlhší dolár, ktorý nepatrí reťazcu S_i. Celú iteráciu vieme urobiť v $O(|S_i|)$ a celý algoritmus v čase $O(n)$.

dolný odhad faktoru Dolný odhad pre aproximačný faktor predchádzajúceho algoritmu je 2. Nech $L_k = \{c(ab)^k, (ba)^k, (ab)^k c\}$. Potom

alg.	nadslovo	dĺžka nadslova
greedy	$c(ab)^k c(ba)^k$	$4k+2$
opt.	$c(ab)^{k+1}c$	$2k+4$

Aproximačný faktor $\frac{4k+2}{2k+4} \to 2$ pre $k \to \infty$. Je otvorený problém, či je to vždy najviac 2. Dá sa ukázať, že uvádzaný algoritmus má faktor 2 pri optimalizácii počtu ušetrených znakov $(\sum |S_i|) - c$ a tiež faktor najviac 4 pri optimalizácii dĺžky najkratšieho nadslova.

Maximalizácia ušetrených znakov: Preformulujme úlohu tak, že maximalizujeme počet ušetrených znakov c, potom dĺžka nadslova bude $(\sum |S_i|) - c$. V optimálnom prípade sa takáto dĺžka bude rovnať dĺžke najkratšieho spoločného nadslova, inak by sme mohli ešte niečo ušetriť. Teraz môžme ukázať, že takáto preformulovaná úloha sa dá aproximovať faktorom 2 pomocou už spomenutého greedy algoritmu.

27.3 TO DO

Dokaz pre 4-apriximaciu usetrenych znakov.

28 Hľadanie motívov

Doteraz sme sa prevažne zaoberali problémami, kde sme v texte chceli nájsť nejakú nami zvolenú vzorku. Dnešný problém je trochu opačný: na základe textov sa snažíme nájsť zaujímavú vzorku, ktorá sa v nich často vyskytuje.

28.1 Biologická motivácia

DNA obsahuje gény (úseky kódujúce proteíny, RNA molekuly) a sekvencie regulujúce, kedy sa ktorý gén má transkribovať (kopírovať do RNA). Na spustenie transkripcie sa na DNA viažu proteíny zvané transkripčné faktory, ktoré "prilákajú" RNA polymerázu k začiatku génu, aby sa mohla začať transkripcia. Chceli by sme vedieť, ktorý transkripčný faktor sa kam viaže v genóme a ktoré gény reguluje. Každý faktor rozpoznáva určitý motív v DNA sekvencii. Napríklad faktor E2F1 sa viaže na reťazec TTTCGCGC, pripúšťa však určité obmeny, napr. 4. a 5. pozícia môžu byť C alebo G a jedno z prvých troch T môže byť zmenené na iný znak.

Máme teda jeden konkrétny transkripčný faktor a chceme zistiť, na aký motív v sekvencii sa viaže. Môžeme použiť experimentálne metódy, ktoré vedia nájsť približne miesta, kde sa daný faktor viaže, s presnosťou napr. asi 500 znakov. Na základe takéhoto experimentu dostaneme veľa približne 500-znakových úsekov genómu a chceme nájsť krátky motív, nachádzajúci sa v každom alebo aspoň vo väčšine z nich. Dostávame teda výpočtové problémy nasledujúceho typu:

Máme dané reťazce $S_1, S_2, \ldots, S_k$ dĺžky n a dĺžku motívu $L < n$. Chceme nájsť motív dĺžky L, ktorý sa vyskytuje v každom, alebo v čo najviac reťazcoch S_i.

Skúma sa niekoľko formulácií tohto problému, ktoré sa líšia definíciou motívu a jeho výskytov.

28.2 Problém 1: presné výskyty reťazca

V tomto prípade motív bude jednoducho reťazec dĺžky L a budeme uvažovať jeho presné výskyty v texte. Hľadáme teda také slovo dĺžky L, ktoré sa vyskytuje v čo najviac zo vstupných reťazcov $S_1, S_2, \ldots, S_k$. Triviálne môžeme tento problém riešiť tak, že zoberieme každé podslovo dĺžky L zo vstupných reťazcov a spočítame jeho výskyty v ostatných reťazcoch. Ak použijeme triviálny algoritmus aj na hľadanie výskytov, dostaneme zložitosť $O(n^2k^2L)$. Videli sme však, ako sa dá tento problém riešiť aj v lineárnom čase $O(nk)$, ak použijeme sufixové stromy a štruktúry na hľadanie najnižšieho spoločného predka.

28.3 Problém 2: Consensus Pattern Problem, nepresné výskyty reťazca

V tomto probléme je motív stále obyčajný reťazec, dovoľujeme však približné výskyty, pričom meriame Hammingovu vzdialenosť. Formálne hľadáme reťazec S dĺžky L a pozície výskytu $i_1, i_2, \ldots i_k$ také, že súčet vdialeností od S k jednotlivým výskytom $\sum_{j=1}^{k} d_H(S, S_j[i_j..i_j + L - 1])$ je minimálny.

Poznámka: Ak by sme namiesto Hammingovej použili editačnú vzdialenosť, išlo by o lokálnu verziu viacnásobného zarovnania.

Tento problém je opäť NP-ťažký, obsahuje však dva podproblémy, ktoré sa dajú efektívne riešiť. Ak poznáme výskyty $i_1, \ldots i_k$, vieme optimálne S nájsť tak, že na každej pozícii zoberieme znak s najväčším počtom výskytov. Naopak, ak poznáme S, vieme nájsť $i_1, \ldots i_k$ tak, že v každej sekvencii vyberieme podslovo dĺžky L s najmenšou Hammingovou vzdialenosťou od S.

Tieto dva algoritmy sa dajú spojiť do jednoduchej heuristiky: začneme s nejakým zoznamom výskytov, pre ne nájdeme najlepší motív S a pre tento motív znova nájdeme výskyty. To opakujeme, kým dochádza k zmenám. V každej iterácii sa cena riešenia zlepší, alebo ostane rovnaká.

Druhá heuristika zvolí za S nejaké podslovo reťazca S_i dĺžky L, potom nájde výskyty. Ak toto spravíme pre všetky podslová všetkých vstupných reťazcov a zvolíme najlepšie, dostávame 2-aproximačný algoritmus.

Dôkaz: Nech α_i je optimálny výskyt optimálneho motívu S v reťazci S_i, nech $d_i = D_H(\alpha_i, S)$ a nech p je index najmenšieho d_i. Potom keď skúšame v našom algoritme motív α_p, dostávame podľa trojuholníkovej nerovnosti, ktorá pre Hammingovu vzdialenosť platí, nasledujúci vzťah:

$$d_H(\alpha_p, \alpha_i) \leq d_H(\alpha_p, S) + d_H(S, \alpha_i) \leq 2d_H(\alpha_i, S).$$

Teda cena motívu α_p je najviac dvojnásobok ceny S. Náš algoritmus nájde riešenie aspoň tak dobré ako α_p (môže zvoliť iné výskyty alebo iný motív, ale iba ak sú aspoň tak dobré).

Tento algoritmus môžeme ďalej zovšeobecniť tak, že nebudeme začínať z jedného podslova, ale z r podslov, kde r je konštanta zvolená užívateľom. Budeme skúšať všetky r-tice podslov dĺžky L zo vstupných reťazcov s opakovaním (t.j. to isté podslovo môžeme použiť aj viackrát v tej istej r-tici). Pre každú r-ticu nájdeme jej konsenzus S (najpočetnejšie písmeno v každom stĺpci) a pre S potom nájdeme najlepší výskyt v každom reťazci. Celkovo zvolíme S, ktoré malo najlepšiu cenu.

Príklad: Uvažujme reťazce abaab, baaba, aabab, bbaaa, aabba a $L = 5$, t.j. celý reťazec bude výskytom. Optimálny motív je aaaaa s cenou 10 (od každého reťazca sa líši o 2). Pre $r = 1$ zvolíme vždy za motív jeden reťazec. Zhodou okolností nám pre každý výjde cena 12, t.j. aproximačný faktor 1,2. Pre $r = 3$ napr. pre trojicu abaab, baaba a aabab dostaneme konsenzus aaaab, ktorý má cenu 11, t.j. aproximačný faktor 1,1.

Li, Ma and Wang (1999) dokázali, že tento algoritmus má aproximačný faktor najviac $1 + \frac{4\sigma-4}{\sqrt{e}(\sqrt{4r+1}-3)}$ v čase $O((nk)^{r+1}L)$. Ide teda o polynomiálnu aproximačnú schému (PTAS): pre ľubovoľné ε si vieme zvoliť r tak, aby chyba bola menšia ako $1+\varepsilon$. Avšak čas rastie exponenciálne s r a nedostávame tak veľmi praktický algoritmus.

28.4 Problém 3: Closest substring, nepresné výskyty reťazca

Tento probléme sa líši od predchádzajúceho iba definíciou ceny riešenia. Namiesto súčtu vzdialeností výskytov od motívu budeme uvažovať maximum: $\max d_H(S, S_j[i_j..i_j+L-1])$. Ak tento problém preformulujeme ako rozhodovací, pýtame sa, či pre existuje motív, ktorý má v každom vstupnom reťazci výskyt s Hammingovou vzdialenosťou najviac k.

Tento problém je ešte o niečo ťažší ako predchádzajúci, lebo aj keby sme poznali výskyty, nevieme ľahko nájsť motív. Nasledujúci problém je totiž tiež NP-úplný. Closest string prob-

	1	2	3	4	5	6	7	8
A	0.1	0	0	0	0	0	0	0
C	0	0.1	0	0.5	0.4	1	0	1
G	0	0	0	0.5	0.6	0	1	0
T	0.9	0.9	1	0	0	0	0	0

Obr. 50: Príklad pravdepodobnostného profilu.

lem: pre dané reťazce $S_1, S_2, \ldots, S_k$ rovnakej dĺžky n nájdi reťazec, ktoré je od každého z nich vo vzdialenosti najviac k.

Rovnako ako predtým však aj tu môžeme dostať 2-aproximačný algoritmus, ak budeme ako S skúšať všetky podslová. Takisto sa v praxi používa prehľadávanie s orezávaním, ktoré často vie nájsť optimálne riešenie na realistických inštanciách.

28.5 Problém 4: motív ako regulárny výraz

Motív tentokrát bude jednoduchý regulárny výraz, v ktorom okrem normálnych znakov, dovolíme aj žolíky, ktoré reprezentujú jeden z danej množiny znakov. Napríklad motív pre transkripčný faktor E2F1 sa dá zapísať ako `TTT[CG][CG]CGC`, kde na štvrtej a piatej pozícii dovoľujeme C alebo G. Niekedy sa skúmajú aj motívy s flexibilnými medzerami, napr. `TTT-N(2,3)-CGC` reprezentuje motív, ktorý začína TTT, potom obsahuje ľubovoľné 2 alebo 3 písmená (N zvykne v DNA označovať ľubovoľný alebo neznámy znak) a potom CGC. Pri regulárnych výrazoch zvyčajne za výskyt považujeme podslovo, ktoré spĺňa tento výraz, t.j. už nepovoľujeme ďalšie substitúcie.

Pre daný motív chceme uvažovať počet jeho výskytov, ale aj jeho špecifickosť (motív s veľa žolíkmi sa bude často vyskytovat aj čisto náhodou). V tejto oblasti existuje zaujímavá kombinaorická teória, ktorá definuje akúsi lineárne veľkú bázu motívov, z ktorej sa dajú odvodiť všetky ostatné motívy s aspoň dvoma výskytmi v daných reťazcoch bez toho, aby sme sa pozerali na tieto reťazce.

28.6 Problém 5: motív ako pravdepodobnostný profil

Regulárny výraz môže dovoľovať na určitej pozícii výber z viacerých alternatív, nevie však zašpecifikovať, že niektoré z týchto alternatív sa vyskytujú menej často a niektoré častejšie. Toto sa dá dosiahnuť pravdepodobnostným profilom. Ten je určený maticou A rozmerov $\sigma \times L$, kde $A[x,i]$ je pravdepodobnosť, že na i-tej pozícii v motíve bude znak x. Súčet každého stĺpca matice je 1. V príklade na obrázku 50 tretia, šiesta, siedma a ôsma pozícia obsahujú vždy jedno fixné písmeno s pravdepdobnosťou 1, kým na ostatných pozíciách sú možné vždy dve rôzne písmená potenciálne s rôznou pravdepodobnosťou.

Aby sme určili, či nejaký reťazec α dĺžky L je výskytom motívu, spočítame jeho pravdepodobnosť ako súčin $p_\alpha = \prod_{i=1}^{L} A[i, \alpha[i]]$ (predpokladáme teda, že jednotlivé pozície motívu sú nezávislé). Môžeme potom obmedziť výskyty ako reťazce s pravdepodbnosťou nad určitou hranicou.

Ak však hľadáme nový motív, zvyčajne neurčujeme jednoznačný prah pravdepodobnosti, ale hľadáme tabuľku A a výskyt v každej sekvencii $\alpha_1, \ldots, \alpha_k$, ktoré maximalizujú súčin pravdepdobností jednotlivých výskytov $\prod_{j=1}^{k} p_{\alpha_j}$. Ak už poznáme sadu výskytov a chceme zosta-

viť motív, ktorý by maximalizoval tento súčin, spočítame na každej pozícii relatívne frekvencie jednotlivých znakov medzi výskytmi a tie dáme do tabuľky. Všimnite si, že toto je podobné na Consensus pattern problem, kde sme do motívu dali vždy najčastejší znak.

Jedným z možných heuristických prístupov na riešenie tohto problému je stochastický algoritmus Gibbs Sampling. Tento na začiatku zvolí náhodné výskyty $\alpha_1, \ldots, \alpha_k$ a z nich zostaví profil. Potom v každej iterácii vyberie jeden zo vstupných reťazcov S_i a v ňom zvolí náhodne jednu pozíciu ako nový výskyt α_i pričom každú pozíciu v sekvencii zvolí s pravdepodobnosťou úmernou p_{α_i} pre túto pozíciu. Po zvolení nového výskytu prepočítame profil a celý postup veľa krát opakujeme.

Tento iteratívny algoritmus sa podobá na jednoduchú heuristiku, ktorú sme videli pre Consensus pattern problem, ale líši sa v dvoch aspektoch: v každej iterácii meníme polohu len jedného výskytu, nie všetkých a ako nový výskyt neberieme ten najlepší, ale náhodne, pričom lepšie výskyty majú väčšiu pravdepodobnosť.

29 Úsporné dátové štruktúry

Pozor, táto časť obsahuje iba zopár útržkovitých poznámok, kombinácia prezentácie a prípravy na hodinu.

29.1 Úvod do úsporných štruktúr, úsporná štruktúra pre binárny strom a pre rank a select

Pozri prednášku 17, Erik Demaine, MIT
`http://courses.csail.mit.edu/6.851/spring12/lectures/`

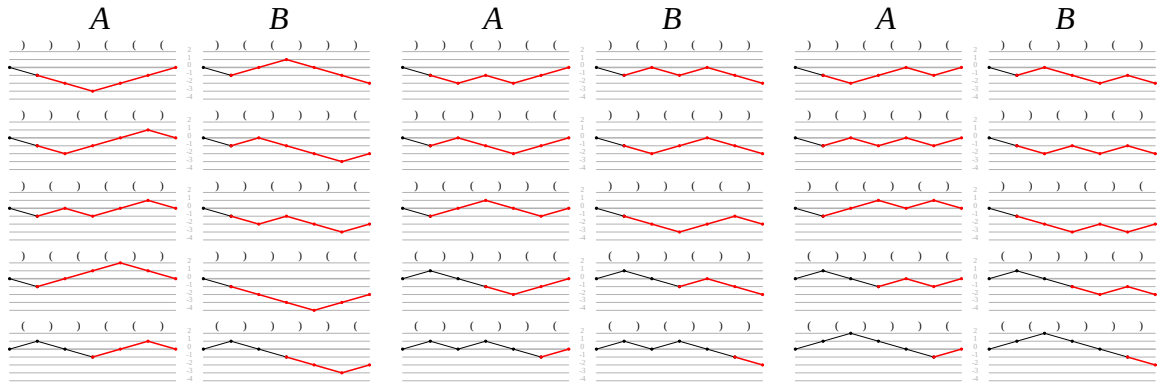
29.2 Zhrnutie rank/select

Rank:

- trivialne riesenie n log n bitov, pamatam si rank pre kazde i
- prakticke riesenie: pamatam si rank iba pre i*t pre nejake t, zvysok dopocitam
- teoreticke riesenie: n+o(n) bitov: pamatam si rank kazdych t1 bitov, v ramci oblasti kratsi rank v ramci t2 bitov, predpocitana tabulka pre rank v ramci useku dlzky t2
- poznamka: ak chceme spocitat $\text{rank}_0(i)$, pouzijeme i+1-$\text{rank}_1(i)$

Select:

- teoreticke riesenie: n+o(n) bitov, zlozitejsie ako rank
- bin vyhladavanie so selectom O(log n) cas



Obr. 51: Ilustrácia dôkazu o Catalánových číslach. Bijekcia medzi prvkami množín $A = X(3,3,-\infty) \setminus X(3,3,0)$ a $B = X(2,4,-\infty)$. Grafy zobrazujú prefixové súčty. Vedľa seba sú vždy zodpovedajúce si prvky z A a B, prevrátená časť je zobrazená červenou.

29.3 Binarny lexikograficky strom

Na Erikovej prednaske sa vieme pohybovat po strome, ale nevieme zistot, ci vnut. vrchol zodpoveda slovu v mnozine

Jednoduchy fix s 3n bitmi: bitovy vektor pre vsetky vrcholy stromu, poznac si, ktore vrcholy zodpovedaju slovu z mnoziny

Ktory sme ozajstny vrchol zisti pomocou rank

Pre tie slova, ktore su v mnozine mozeme mat dalsie pole indexovane cislom slova, ktore opat ziskame pomocou rank v tomto dalsom vektore, v tomto poli potom mozeme mat dalsie udaje, ako cisla dokumentov

Urcite existuju aj lepsie datove struktury, napr. nepotrebujeme explicitne ukladat bit pre listy

29.4 Catalánove čísla

Budeme uvažovať nasledujúce problémy:

- Koľko je binárnych stromov s n vrcholmi?
- Koľko je dobre uzátvorkovaných výrazov s n pármi zátvoriek?
- Koľko je postupností, ktoré obsahujú n krát číslo +1 a n krát -1 a pre ktoré sú všetky prefixové súčty nezáporné?

Už sme videli, že prvé dva z týchto problémov sú navzájom ekvivalentné. Druhé dva sú tiež ekvivalentné, lebo stačí každú ľavú zátvorku nahradiť +1 a každú pravú zátvorku -1. Aby bol výraz dobre uzátvorkovaný, v každom jeho prefixe musí byť aspoň toľko ľavých zátvoriek ako pravých a teda v postupnosti +1 a -1 máme nezáporný prefixový súčet.

Ukážeme teraz, že odpoveďou na všetky tri uvedené problémy je Catalánove číslo $C_n = \binom{2n}{n}/(n+1)$ ($C_0 = 1, C_1 = 1, C_2 = 2, C_3 = 5, \ldots$)

Nech $X(n,m,k)$ je množina vsetkých postupností dĺžky $n+m$ obsahujúcich n krát číslo +1, m krát číslo -1 takých, že každý prefixový súčet je aspoň k. My chceme spočítať veľkosť množiny $X(n,n,0)$.

Ak by sme nekládli žiadne požiadavky na prefixové súčty, dostali by sme množinu $X(n,m,-\infty)$. Veľkosť tejto množiny je $|X(n,m,-\infty)| = \binom{n+m}{n}$, lebo z celkových $n+m$ pozícií v postupnosti musíme zvoliť n, na ktorých bude hodnota +1, na ostatných bude -1.

Uvažujme teraz množinu $A = X(n,n,-\infty) \setminus X(n,n,0)$, t.j. množinu všetkých "zlých" postupností, v ktorých prefixový súčet aspoň raz klesne pod nulu. Ukážeme, že táto množina je rovnako veľká ako množina $B = X(n-1,n+1,-\infty)$. V množine B sú prefixové súčty ľubovoľné a teda jej veľkosť vieme spočítať. Našim cieľom je nájsť zobrazenia $f : A \to B$ a $g : B \to A$, ktoré sú navzájom inverzné a teda ide o dve bijekcie, potvrdzujúce rovnakú veľkosť množín A a B.

Vezmime teda nejakú postupnosť $a_1 \dots a_{2n}$ z A a nájdime najmenší index i, že súčet $a_1 + \cdots + a_i$ je -1. Postupnosť $f(a_1 \dots a_{2n}) \in B$ zostavíme tak, že od indexu i napravo zmeníme znamienka na opačné (obr. 51). Nakoľko súčet celej postupnosti $a_1, \dots, a_{2n}$ je 0 a súčet jej prvých i členov je -1, súčet zvyšných členov je +1. Keď im zmeníme znamienka na opačné, ich súčet bude -1 a teda súčet celej zmenenej postupnosti bude -2, musí teda obsahovať $n-1$ krát $+1$ a $n+1$ krát -1. Ide teda naozaj o postupnosť z množiny B.

Naopak hodnotu zobrazenia g pre nejakú postupnosť $b_1 \dots b_{2n} \in B$ zostavíme tak, že opäť nájdeme prvý index i, kde prefixový súčet klesne na -1 (taký musí existovať, lebo celkový súčet je -2). Opäť všetkým členom postupnosti napravo od i zmeníme znamienka na opačné. Takto zjavne dostávame zobrazenie inverzné k f.

Dokázali sme teda, že $|A| = |B| = \binom{2n}{n-1}$. Počet dobre uzátvorkovaných postupností je teda $|X(n,n,0)| = |X(n,n,-\infty)| - |X(n-1,n+1,-\infty)| = \binom{2n}{n} - \binom{2n}{n-1}$. Ľahkou úpravou výrazov môžeme overiť, že tento rozdiel je rovný $\binom{2n}{n}/(n+1)$.

Iný spôsob ako odvodiť vzorec pre počet dobre uzátvorkovaných výrazov je najskôr zapísať tento počet rekurenciou. Nech teda $T(n)$ je počet dobre uzátvorkovaných výrazov s n pármi zátvoriek. Pre $n = 0$ uvažujeme prázdny výraz za dobre uzátvorkovaný, máme teda $T(0) = 1$. Pre $n > 0$ uvažujme prvú ľavú zátvorku a jej zodpovedajúcu pravú zátvorku. Nech i je počet párov zátvoriek medzi týmito dvoma. Vnútri tohto najľavejšieho páru môžeme zátvorky rozmiestniť $T(i)$ spôsobmi a zvyšné zátvorky sú napravo od tohto páru a môžu tam byť rozmiestnené $T(n-i-1)$ spôsobmi. Dostávame teda rekurenciu $T(n) = \sum_{i=0}^{n-1} T(i)T(n-i-1)$. Ak túto rekurenciu vyriešime metódami kombinatorickej analýzy, dostávame opäť Catalánove čísla. Na riešenie je možné použiť napríklad generujúce funkcie.

29.5 Wavelet tree: Rank nad väčšou abecedou

- Namiesto bitového poľa máme reťazec S nad väčšou abecedou Σ
- Chceme podporovat $\text{rank}_a(i)$, co je pocet vyskytov pismena a v retazci $S[0..i]$
- OPT je $n \log_2 \sigma$
- Pouzijeme rank na binarnych retazcoch nasledovne:
- rozdelime abecedu na dve casti Σ_0 a Σ_1
- Vytvorime bin. retazec B dlzky n: B[i]=1 iff S[i] je v Σ_1
- Vytvorime retazce S_0 a S_1, kde S_i obsahuje pismena z S, ktore su v Σ_i, sucet dlzok je n

- Rekurzivne pokracujeme v deleni abecedy a retazcov, az kym nedostaneme abecedy velkosti 2

rank$_a(i)$ v tejto strukture:

- ak $a \in \Sigma_j$, i=rank$_j(i)$ v retazci B
- pokracuj rekurzivne s vypoctom rank$_a(i)$ v lavom alebo pravom podstrome

Velkost: binarne retazce $n\log_2\sigma$ + struktury pre rank nad bin. retazcami (mozem skonkatenovat na kazdej urovni a dostat $o(n\log_2\sigma)$

Cas $O(\log\sigma)$, robim jeden rank na kazdej urovni. Extrahovanie S[i] tiez v case $O(\log\sigma)$

Príklad: $\Sigma_0 = \{\$,.,a\}, \Sigma_1 = \{e,m,u\}$

```
i    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
T[i] e  m  a  .  m  a  .  m  a  m  u  .  m  a  m  a  .  m  a  .  e  m  u  $
B[i] 1  1  0  0  1  0  0  1  0  1  1  0  1  0  1  0  0  1  0  0  1  1  1  0
T0   a.a.a.aa.a.$
T1   emmmmummmemu
```

29.6 FM index

- Ferragina and Manzini 2000
- hladanie vyskytov P v T pomocou BWT a ranku
- T dlzky n, pripojime $ na poziciu T[n]
- zostrojime sufixove pole SA pre T a $T' = BWT(T)$
- zostrojime wavelet tree pre T', vieme ratat rank$_a(i)$ v T'
- spocitame $C[a] = \sum_{b\in\Sigma, b<a}$ pocet vyskytov b v T
- nech L(S) = najmensi index i t.z. S je prefix T[SA[i]..n]
- nech R(S) = najvacsi index ...
- pre S = epsilon, mame L(S)=0, R(S)=n
- ak aS je podslovo T, $L(aS) = C[a] + rank_a(L(S)-1)$
- ak aS je podslovo T, $R(aS) = C[a] + rank_a(R(S)) - 1$
- navyse ak S je podslovo T, tak aS je podslovo T prave vtedy ak L(aS)<=R(aS)
- Opakujeme pre stale dlhsie sufixy P, nakoniec dostaneme interval SA, kde sa nachadzaju vyskyty - spatne hladanie
- vieme teda v O(|P|) testovat, ci ma vyskyt a kolko ich je, nepotrebujeme ani SA, iba wavelet tree
- existuju aj techniky na kompaktnejsie ulozenie SA

Priklad

```
i      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
T[i]   e  m  a  .  m  a  .  m  a  m  u  .  m  a  m  a  .  m  a  .  e  m  u  $
SA[i] 23 19 16  3 11  6 18 15  2  5 13  8  0 20 17 14  1  4 12  7 21  9 22 10

T'[i]  u  a  a  a  u  a  m  m  m  m  m  m  $  .  .  a  e  .  .  .  e  a  m  m
r$(i)  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1
r.(i)  0  0  0  0  0  0  0  0  0  0  0  0  0  1  2  2  2  3  4  5  5  5  5  5
ra(i)  0  1  2  3  3  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  6  6  6
re(i)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  2  2  2  2
rm(i)  0  0  0  0  0  0  1  2  3  4  5  6  6  6  6  6  6  6  6  6  6  6  7  8
ru(i)  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2


x      $  .  a  e  m  u
C(x)   0  1  6 12 14 22


hladam .ama
L(epsilon) = 0
R(epsilon) = 23
L(a) = C(a) + ra(L(epsilon)-1) = 6 + ra(-1) = 6
R(a) = C(a) + ra(R(epsilon))-1 = 6 + ra(23)-1 = 11
L(ma) = C(m) + rm(L(a)-1) = 14 + rm(5) = 14
R(ma) = C(m) + rm(R(a))-1 = 14 + rm(11) - 1 = 19
L(ama) = C(a) + ra(L(ma)-1) = 6 + ra(13) = 10
R(ama) = C(a) + ra(R(ma))-1 = 6 + ra(19) - 1 = 10
L(.ama) = C(.) + r.(L(ama)-1) = 1 + r.(9) =  1
R(.ama) = C(.) + r.(R(ama))-1 = 1 + r.(10)-1 =  0


 0 23 $
 1 19 .emu$
 2 16 .ma.emu$
 3  3 .ma.mamu.mama.ma.emu$
 4 11 .mama.ma.emu$
 5  6 .mamu.mama.ma.emu$
 6 18 a.emu$
 7 15 a.ma.emu$
 8  2 a.ma.mamu.mama.ma.emu$
 9  5 a.mamu.mama.ma.emu$
10 13 ama.ma.emu$
11  8 amu.mama.ma.emu
12  0 ema.ma.mamu.mama.ma.emu$
13 20 emu$
14 17 ma.emu$
15 14 ma.ma.emu$
16  1 ma.ma.mamu.mama.ma.emu$
17  4 ma.mamu.mama.ma.emu$
18 12 mama.ma.emu$
19  7 mamu.mama.ma.emu$
20 21 mu$
21  9 mu.mama.ma.emu$
22 22 u$
23 10 u.mama.ma.emu$
```

29.7 TODO

Prerobiť na normálne poznámky, pridať niečo z MIT prednášky.