VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

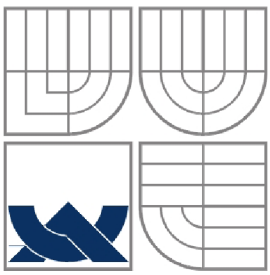
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYHLEDÁVÁNÍ PŘIBLIŽNÝCH PALINDROMŮ V DNA SEKVENCÍCH

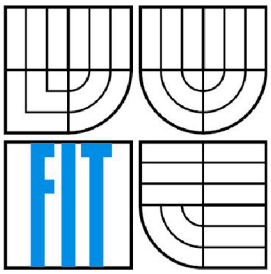
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE *RICHARD REMIÁŠ*
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYHLEDÁVÁNÍ PŘIBLIŽNÝCH PALINDROMŮ V DNA SEKVENCÍCH

FINDING APPROXIMATE PALINDROMES IN DNA SEQUENCES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RICHARD REMIÁŠ
AUTHOR

VEDOUCÍ PRÁCE — ING. TOMÁŠ MARTÍNEK
SUPERVISOR

BRNO 2010

Abstrakt

Práca sa zaoberá problematikou vyhľadávania presných a približných palindrómov. V súvislosti s vyhľadávaním presných palindrómov analyzuje naivný postup vyhľadávania ako aj postupy založené na sufixových stromoch, ktorých konštrukcia je tiež rozobraná. Vyhľadávanie približných palindrómov je realizované za pomoci princípov dynamického programovania. Samotné vyhľadávanie je rozdelené na tri časti: vyhľadanie palindrómov, filtrácia výsledkov a ich rekonštrukcia. Každá časť je popísaná algoritmom a implementovaná programom v prílohe práce.

Abstract

This work discusses problematics of exact and approximate palindrome searching. In relation with exact palindrome searching, native algorithm and algorithm using suffix trees, which construction is also analyzed, is presented. Approximate palindrome search is implemented with application of dynamic programming principles. The search itself is dividded into three parts: palidnrome search, result filter and reconstruction of palindromes. Each part is described by an algorithm and implemented in a program, contained in attachment.

Klíčová slova

Palindrom, přibližný palindrom, přesný palindrom, vyhledávání palindromů, sufixový strom, dynamické programování.

Keywords

Palindrome, approximate palindrome, exact palindrome, palindrome searching, suffix tree, dynamic programming.

Citace

Richard Remiáš: Vyhledávání přibližných palindromů v DNA sekvencích, bakalářská práce, Brno, FIT VUT v Brně, 2010

Vyhľadávanie približných palindrómov v DNA sekvenciách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Tomáše Martínka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Richard Remiáš
15. květen 2010

Poděkování

Chcel by som poďakovať môjmu vedúcemu, pánovi Inžinierovi Tomášovi Martínkovi za to, že mal tú vôľu a diskutoval somnou všetky vážne problémy na ktoré som narazil.

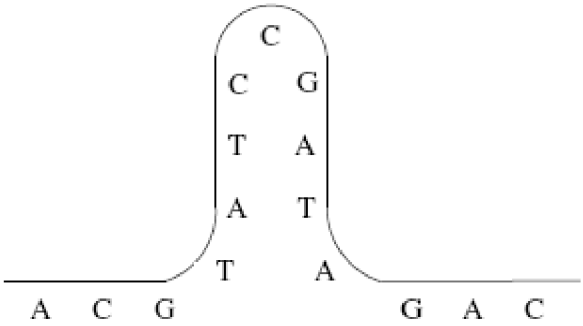
Obsah

1 Úvod

Vyhľadávanie reťazcov v texte je, v dnešnej dobe, jedna z najčastejších aplikácií informatiky. Riešením problematiky vyhľadávania sa zaoberalo nespočet odborníkov a vymysleli množstvo algoritmov, ktoré tento problém riešia. Základné požiadavky na vyhľadávanie sú rýchlosť a presnosť. Práve za účelom zvyšovania rýchlosti sa vymýšlajú nové algoritmy a štruktúry pre reprezentáciu dát.

Vyhľadávanie je v súčasnosti vo veľkej miere využívané aj v biológií, konkrétne vo vetve, ktorá sa zaoberá lúštením genetického kódu. Jeden typ reťazcov, ktorý je potrebné vyhľadávať sú palindrómy. Aby bolo možné úspešne vyhľadávať tieto reťazce, je potrebné nájsť odpovede na tri základné otázky: Čo je to palindóm? Aké dátove štruktúry sú ideálne na reprezentáciu dát, v ktorých palindrómy vyhľadávame? Aké algoritmy je potrebné použiť, aby sme sa dopracovali k výsledku v rozumnom čase?

Odpoveď na otázku „Čo je to palindróm?“ je pomerne jednoduchá. Palindróm, neodborne povedané, je reťazec, ktorý sa číta rovnako odpredu, ako aj odzadu. Aj v bežnej konverzácí môžeme naraziť na slová-palindrómy ako **rotor, radar** alebo **madam**. Všetky vymenované slová sú palindrómy nepárne, bez medzery. Od palindrómov, ktoré sa nachádzajú v reťazcoch DNA sa líšia v niekoľkých podstatných rysoch. Množina znakov vyskytujúcich sa v reťazcoch DNA má iba štyroch členov, písmená A, T, G a C. Sú to začiatočné písmená látok, ktoré tvora genetický kód: **adenín**, **tymín**, **cytozín** a **guanín**. Tieto látky sa navzájom viažu vodíkovími väzbami na základe komplementarity. Komplementárne bázy sú adenín – tymín a cytozín – guanín, čo znamená, že sa nevyskytujú väzby A-C, A-G, T-C alebo T-G. Táto skutočnosť určuje zásadnú odlišnosť medzi klasickými palindrómami a palindrómami v reťazcoch DNA, ktorá je demonštrovaná na obrázku 1.



Obr. 1 Presný nepárny palindróm v reťazci DNA

Na obrázku je znázornený nepárny palindróm **TATCCGATA**. Znak C na piatom mieste v reťazci tvorí stred palindrómu, môžeme ho nazvať aj medzerou s dĺžkou jeden. Reťazce TATC a ATAG si na prvý pohľad neodpovedajú, ale je potrebné si uvedomiť, že reťazec ATAG tvorí komplement k reťazcu TATC, tým pádom je možné danú postupnosť znakov vyhodnotiť ako palindróm, konkrétne presný palindróm.Viac ako presné palindrómy a v reťazcoch DNA vyskytujú palindrómy približné. Približný palindróm obsahuje chyby, ktoré spôsobujú, že jeho prvá polovica nie je úplne komplementárna s jeho druhou polovicou.

Druhá odlišnosť je dĺžka medzery palindrómu. Doteraz boli uvedené len palindrómy s medzerou dĺžky jeden. Túto medzeru tvoril stredový znak palindrómu. V praxi sa často vyskytujú palindrómy s medzerou, ktorú tvorí niekoľko znakov. Pri vyhľadávaní týchto palindrómov je dôležité si určiť istú hranicu dĺžky medzery. Práve chybovosť palindrómov a existencia medzery sú skutočnosti, ktoré výraznou mierou pridávajú na zložitosti vyhľadávacích algoritmov, ale zároveň ich

nie je možné ignorovať. Za účelom zníženia časovej náročnosti sú používané nielen špeciálne dátové štruktúry a algoritmy, ale aj hardvér určený na akceleráciu výpočtov, prebiehajúcich pri vyhľadávaní.

Táto bakalárska práca sa zameriava na softvérovú časť vyhľadávania približných palindrómov v DNA reťazcoch. Je delená na tri hlavné časti: prvá časť obsahuje delenie a definície rôznych typov palindrómov a chýb, ktoré sa v nich môžu vyskytovať. Druhá časť sa zaoberá reprezentáciou dát, resp. reťazcov, v ktorých sú palindrómy vyhľadávané a spôsobmi vyhľadávania presných palindrómov. Tretia časť je zameraná na spôsoby vyhľadávania palindrómov s chybami pomocou metódy dynamického programovania a jej implementáciu.

2 Palindrómy

Definícia 2.1 *Nech máme reťazec* w *s dĺžkou* n *znakov, v tvare* $w_1, w_2, w_3, \ldots w_{n-1}, w_n$. *Pod pojmom reverzovaná forma reťazca* w *budeme rozumieť reťazec* w^R *v tvare* $w_n, w_{n-1}, w_{n-2}, \ldots . w_2, w_1$.

Definícia 2.2 *Nech máme reťazec* w *tvaru* $w_1, w_2, w_3, \ldots w_{n-1}, w_n$ *s dĺžkou* n .

- *Reťazec tvaru* $ww^R = w_1, w_2, \ldots w_{n-1}, w_n, w_n, w_{n-1}, \ldots w_2, w_1$ *nazývame párnym palindrómom dĺžky* $2n$.
- *Reťazec tvaru* $wcw^R = w_1, w_2, \ldots w_{n-1}, w_n, c, w_n, w_{n-1}, \ldots w_2, w_1$ *nazývame nepárnym palindrómom dĺžky* $2n+1$ *so stredom* c .

Definícia 2.3 *Nech máme reťazec* w *tvaru* $w_1, w_2, w_3, \ldots w_{n-1}, w_n$ *s dĺžkou* n *a reťazec* b *tvaru* $b_1, b_2, b_3, \ldots b_{m-1}, b_m$ *s dĺžkou* m *, pre ktorý platí* $b_1 \neq b_m$.

- *Reťazec tvaru* $wbw^R = w_1, w_2, \ldots w_{n-1}, w_n, b_1, b_2, \ldots b_{m-1}, b_m, w_n, w_{n-1}, \ldots w_2, w_1$ *nazývame palindrómom s medzerou.*
- *Medzerou rozumieme nesymetrický reťazec* b *tvaru* $b_1, b_2, b_3, \ldots b_{m-1}, b_m$ *, pričom* $|b| = m \geqslant 2$.

Definícia 2.4 *Nech máme reťazec* $w = w_1, w_2, w_3, \ldots w_{n-1}, w_n$ *, jeho reverzovanú formu* w^R *a párny palindróm* $P = ww^R$ *. Rozlišujeme tri druhy chýb v palindróme* P *:*

- $P = w_1, w_2, \ldots w_{n-1}, w_n, w_n, w_{n-1}, \ldots w_{i+1}, w_{i-1}, \ldots . w_2, w_1 \quad 1 < i < n$ *nazývame palindróm s chybou typu DELETE.*
- $P = w_1, w_2, \ldots w_{n-1}, w_n, w_n, w_{n-1}, \ldots w_i, a, w_{i-1}, \ldots . w_2, w_1 \quad 1 < i < n$ *nazývame palindróm s chybou typu INSERT.*

- $P = w_1, w_2, \ldots w_{n-1}, w_n, w_n, w_{n-1}, \ldots w_{i+1}, a, w_{i-1}, \ldots . w_2, w_1 \quad 1 < i < n$ *nazývame palindróm s chybou typu MISMATCH.*

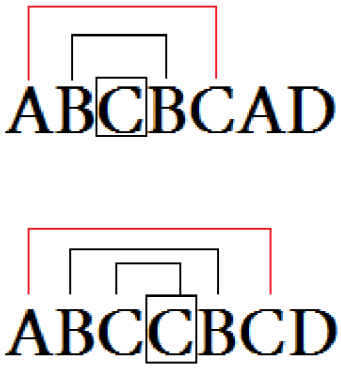
- *Palindróm, ktorý obsahuje aspoň jednu z týchto chýb nazývame palindrómom približným.*

Definícia 2.5 *Pod pojmom maximálny palindróm rozumieme palindróm, ktorý pri rozšírení o jeden znak na pravú a ľavú stranu nevytvorí nový palindróm.*

3 Hľadanie presných palindrómov

3.1 Naivná metóda

Naivná metóda na hľadanie palindrómov [1] využíva na uloženie reťazca pole a sekvenčne ho prehľadáva na výskyt palindrómov. Princíp tejto metódy spočíva v tom, že každý znak uvažuje ako stred palindrómu a sekvenčne prechádza reťatzcom oboma smermi, za účelom odhalenia symetrických častí. Tento prístup vyžaduje dva priechody sekvenciou znakov, pri prvom priechode sa zisťuje výskyt nepárnych palindrómov a pri druhom priechode sa hľadajú párne palindrómy. Pri vyhľadávaní párnych palindrómov sa znak naľavo od uvažovaného stredu vyhodnocuje ako symetrická časť stredu palindrómu, ostatné znaky sa porovnávajú rovnakým spôsobom pri oboch priechodoch.



Obr 3.1: Ukážka hľadania palindrómov v texte naivnou metódou

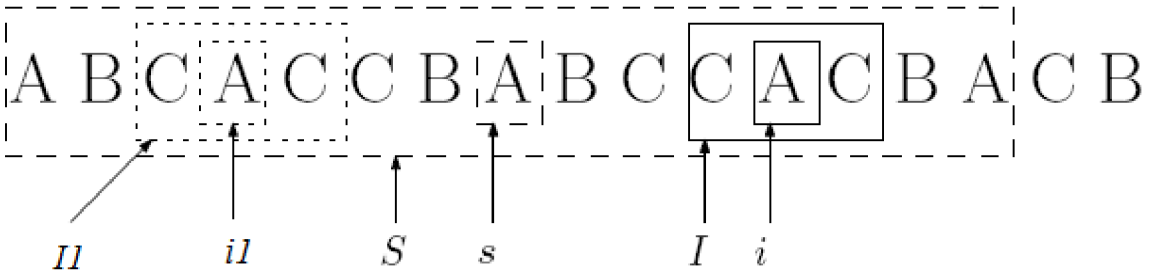
V najlepšom prípade, keď všetky porovnávané znaky tvoria palindróm je zložitosť lineárna $\Omega(2n-3)$, v najhoršom prípade, ak sa na vstupe nevyskytuje žiadny palindróm je zložitosť kvadratická $\Omega(\frac{n^2-n}{2})$ [1].

Z uvedeného vyplýva, že táto metóda v predvedenej forme kvôli svojej zložitosti nie je vhodná na vyhľadávanie palindrómov. K tejto metóde existuje optimalizácia, ktorá uvedenú metódu výrazne urýchluje. Optimalizácia využíva pomocné pole *p*, s dĺžkou odpovedajúcou dĺžke vstupného reťazca. Obsah pola tvorí pre každý znak vstupu dĺžka symetrického okolia daného znaku.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
r[i]	A	C	C	A	C	C	B	A	B	C	C	A	C	B	A	C	B
p[i]	0	0	0	2	0	0	0	5	0	0	0	1	0	0	0	0	0

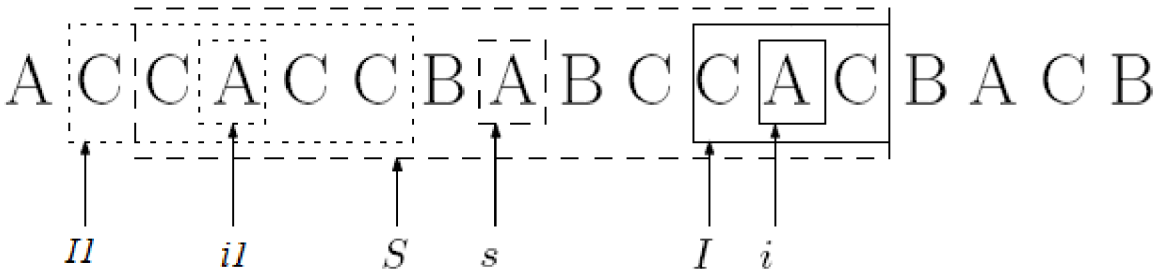
Tabuľka 3.1: Reťazec *r* s poľom *p*

Uvažujme *i* stred, okolo ktorého hľadáme palindróm a *I* jeho symetrické okolie. Už vyhľadávaný stred, ktorého okolie zasahuje najviac doprava označíme *s* a jeho okolie *S*. Stred symetrický podľa *s* so stredom *i* označme *i1* a jeho symetrické okolie *I1*. Optimalizácia nastane, keď sa vyhľadávaný stred *i* nachádza v *S*. Potom je možné zistiť, či stred *i* má nejaké symetrické okolie. Nastáva jeden z dvoch možných prípadov:



Obr. 3.2: Dĺžka *I* a *I1* je rovnaká [1]

1. Medzi ľavým okrajom *I1* a ľavým okrajom *S* je nenulový počet znakov. V takomto prípade je dĺžka *I* rovná dĺžke *I1*.



Obr. 3.3: Dĺžka *I* je neznáma [1]

2. Ľavý okraj *I1* zasahuje ďalej ako *S*, tým pádom vieme určiť minimálnu dĺžku symetrickej časti *I*, ktorá je rovná spoločnej časti *I1* a *S* naľavo od *i1*. Ostatné znaky napravo od *i* je nutné testovať na zhodu.

Uvedená optimalizácia znižuje časovú náročnosť algoritmu na úroveň lineárnu [1].

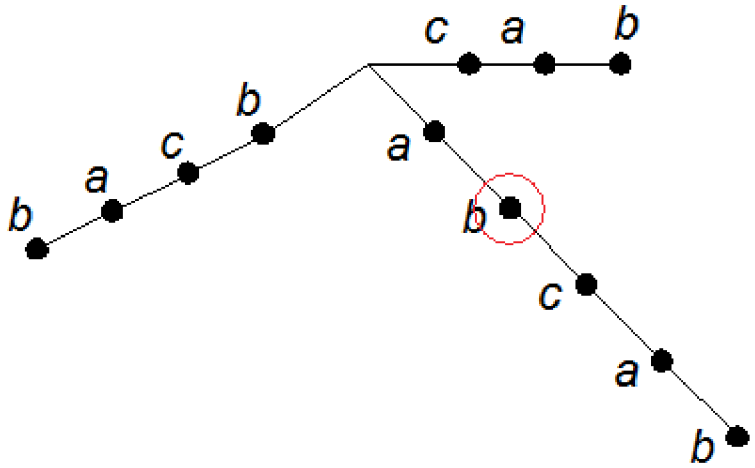
3.2 Suffixové stromy

Definícia 3.1: Nech je $t = t_1 t_2 \ldots t_n \in \Sigma^n$ vstupný text. Potom orientovaný strom $T_t = (V, E)$ s koreňom *r* nazývame jednoduchým sufixovým stromom pre text *t* ak splňuje podmienky:

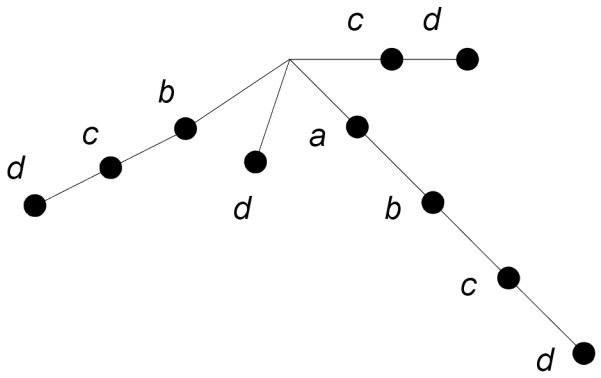
1. Strom má práve *n* listov označených 1 …. n
2. Hrany stromu sú označené znakmi z abecedy Σ
3. Všetky hrany vychádzajúce z uzlu smerom k synovským uzlom sú označené rozlišnými znakmi
4. Cesta od koreňa k uzlu *i* je označená ako $t_i \ldots t_n$ (Cestu chápeme ako postupnosť hrán od koreňa k uzlu *i*)

Sufixové stromy [2] slúžia na predspracovanie textu pred začiatkom vyhľadávania. Proces tvorby takéhoto stromu, ako aj vyhľadávanie v ňom prebieha, ako bude neskôr ukázané, v lineárnom čase. Pre praktické použitie tejto dátovej štruktúry je nutné zistiť, či je možné sufixový strom v uvedenej forme vytvoriť pre ľubovolný reťazec. Sufixový strom podľa definície 3.1 nie je možné vytvoriť pre každý reťazec. Problém nastáva, ak je nejaký sufix zároveň aj prefixom iného sufixu v reťazci.

Príklad takéhoto reťazca je reťazec *abcab*. Sufix *ab* je zároveň prefixom sufixu *abcab* . Podľa definíce 3.1, by mala cesta vychádzajúca z koreňa pre sufix *ab* mala končiť so znakom *b* ako listom, ale znak *b* je taktiež súčasťou cesty pre sufix *abcab*. Táto skutočnosť je rozporom medzi podmienkami 3 a 4 definície 3.1. Na obrázku 3.4 je tento prípad znázornený. Na obrázku 3.5 je znázonený sufixový strom pre valídny reťazec *abcd*.



Obr 3.4: Nesprávny sufixový strom pre reťazec *abcab*



Obr. 3.5: Sufixový strom pre reťazec *abcd*

Riešenie problému konštrukcie sufixového stromu pre ľubovolný reťazec spočíva v pridaní znaku $\$ \notin \Sigma$ na koniec vstupného reťazca - *t$*. Vloženie nového symbolu $ na koniec reťazca zaručuje, že žiadny sufix nebude prefixom iného sufixu (Znázornené na obrázku 3.6).

Tvrdenie 3.1: Nech $t = t_1 t_2 \ldots t_n \in \Sigma^n$ je reťazec a $\$ \notin \Sigma$, potom algoritmus 3.1 vytvorí jednodchý sufixový strom pre *t$*.

Dôkaz: Dokázaním platnosti podmienok 1 až 4 definície 3.1 sa potvrdí tvrdenie 3.1. Strom vytvárame postupným pridávaním uzlov k už existujúcemu podstromu, pričom v každom kroku do stromu pridáme jeden sufix reťazca. Dokázanie podmienok:

1. Sufixy sa do stromu pridávajú postupne so znižujúcou sa dĺžkou. Tým je zaručené, že v kroku *i* nie je uzol, kam je nová cesta pripojená, listom. Vzhľadom na skutočnosť, že žiadny sufix nie je prefixom iného sufixu, je nová cesta pre každé *i* neprázdna. V každom z krokov 1 až n je pridaný práve jeden list, čo znamená, že je podmienka 1 splnená.

2. Podmienka 2 je zjavne dodržaná pre abecedu $\Sigma \cup \$$.

3. Podmienka 3 vyplýva z faktu, že algoritmus v kroku *i* hľadá najdlhšiu cestu začínajúcu v koreňovom uzle, a na jej koncový uzol pripojí novú cestu. Keby algoritmus pripojil novú cestu cez hranu označenú znakom $a \in \{\Sigma \cup \$\}$ na uzol, z ktorého hrana označená znakom *a* vedie, nová cesta by nebola maximálna.

4. Podmienka 4 vyplýva z konštrukcie algoritmu 3.1

Algoritmus 3.1:

Vstup: Reťazec $t = t_1 t_2 \ldots t_n \in \Sigma^n$

Nech $t' := t\$$ pre symbol $\$ \notin \Sigma, \Sigma' = \Sigma \cup \{\$\}$

Inicializuj T koreňom r a prázdnou množinou hrán

for $i := 1$ **to** n **do**

// vlož nový sufix $t_i \ldots t_n \$$ do stromu

Začni v koreni r a hľadaj v T cestu označenú maximálnym prefixom $t_i \ldots t_{j_i}$

končiacu uzlom x_i // táto cesta je jedinečná a nekončí listom.

Pridaj cestu $x_i, y_{j_i+1}, \ldots y_{i_n}, y_{i_n+1}$ označenú $t_{j_i+1}, \ldots t_n \$$ na strom T, pričom

$y_{j_i+1}, \ldots y_{i_n}, y_{i_n+1}$ sú novo-pripojené uzly.

Označ nový list y_{i_n+1} znakom i.

Výstup: Sufixový strom T pre t'

Takýto sufixový strom nie je vhodný pre praktické použitie, pretože jeho dĺžka môže dosiahnuť až veľkosti $\Omega(|t|^2)$. Tento problém je možné odstrániť efektívnejšou reprezantáciou sufixového stromu. Prvá optimalizácia spočíva v odstránení všetkých uzlov, z ktorých vychádza menej ako dve hrany (za predpokladu že umožníme aby boli hrany označené reťazcami). Drhuhá optimalizácia vychádza z faktu, že všetky reťazce v sufixovom strome sú podreťazcami vstupného reťazca. Tým pádom nemusíme označovať hrany znakmi, ale pozíciami vo vstupnom reťazci. Týmto postupom je možné zmenšiť priestorovú náročnosť sufixového stromu na $O(n \log n)$.

Definícia 3.2: Nech je $t = t_1, t_2, \ldots t_n \in \Sigma^n$ vstupný text. Orientovaný strom $T = (V, E)$ s koreňom r nazývame kompaktný sufixový strom, ak:

1. Strom má práve n listov označených $1 \ldots n$.
2. Každý vnútorný uzol stromu má najmenej dva dcérske uzly.
3. Hrany stromu sú značené podreťazcami vstupného textu t, každý podreťazec dĺžky k je reprezantovaný svojou počiatočnou a konečnou pozíciou v texte t ak $k \geq 2 * \log_2(n - |\Sigma|)$.
4. Všetky reťazce značiace hrany vychádzajúce z uzlu do jeho potomkov majú odlišný počiatočný znak.
5. Každá cesta od koreňa k listu i je označená ako $t_i \ldots . t_n$ (Značenie cesty chápeme ako spojenie jednotlivých označení hrán na ceste).

Algoritmus 3.2 popisuje konštrukciu kompaktného sufixového stromu. Aby bolo možné daný algoritmus použiť, je potrné najprv nadefinovať nasledujúce pojmy:

Definícia 3.3: Nech je $t = t_1, t_2, \ldots t_n \in \Sigma^n$ vstupný reťazec a $t' := t\$$ pričom $\$ \notin \Sigma$. Ďalej nech je $T = (V, E)$ sufixový strom pre t' s funkciou ohodnocujúcou hrany $E \to \Sigma$. Nech *pathlabel(x)* je konkatenácia označení hrán na ceste do uzlu x. Nech *depth* je suma dĺžok hrán

vedúcich do uzlu *x* (Dĺžka *pathlabel(x)*). Ak je *T* jednoduchý sufixový strom, definujeme *Pos(x)* pre každý uzol *x* ako minimálne označenie listu podstromu, ktorý má koreň v uzle *x*.

V predchádzajúcej definící uvažujeme označenia hrán na ceste k uzlu *x* ako neskracované.

Algoritmus 3.2 [2]:

Vstup: Reťazec $t = t_1, t_2, \ldots t_n \in \Sigma^n$

1. Nech $t' := t\$$ pre symbol $\$ \notin \Sigma, \Sigma' = \Sigma \cup \{\$\}$. Vytvoríme suffixový strom *T = (V,E)* podľa algoritmu 2.1.
2. Eliminácia uzlov s 1 potomkom:
   Nech je *X* množina všetkých uzlov s jedným potomkom
   while $X \neq 0$ **do**
   Vyber prvok *x* z množiny *X*, nech *y* je rodič uzlu *x* a z jeho potomok. Odstráň hrany *(y,x)* a *(x,z)* a nahraď ich jedinou hranou *(y,z)* s ohodnotením *label(y,z) = label(y,x)label(x,z)*. Odstráň uzol *x*.
3. Kompresia dlhých značení:
   for all $e = (x, y) \in E$ **do**
   if $|label(e)| \geq 2 * \log_2(n - \Sigma \cup \{\$\})$ **then**
   //nahradenie označenia pomocou znaku označením pomocou pozící v texte
   $label'(e) := [Pos(y) + depth(x), Pos(y) + depth(x) + |label(e)| - 1]$
   else
   $label'(e) := label(e)$

Výstup: Kompaktný sufixový strom

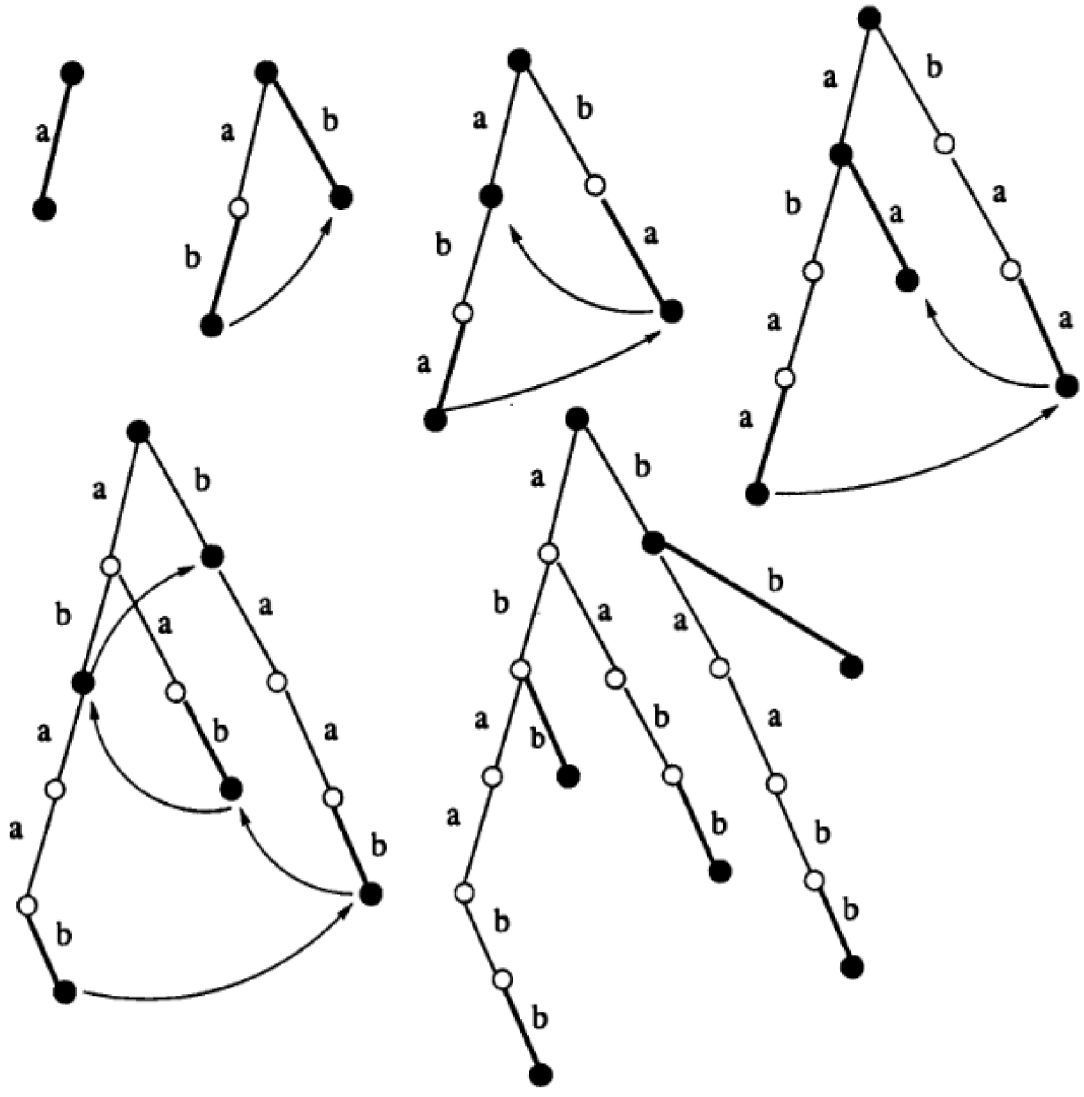
Časová zložitosť uvedeného algoritmu sa blíži až k hodnote $O(n^2)$. Existujú aj rýchlejšie algoritmy na výrobu sufixového stromu, napr. Ukkonenov algoritmus popísaný v nasledujúcej kapitole.

3.2.1 On-line konštrukcia sufixového stromu

Tento typ konštrukcie sufixového stromu využíva Ukkonenov algoritmus [3]. Princíp spočíva v konštrukcí sufixových stromov pre jednotlivé prefixy vstupného textu. V nasledujúcom výklade chápeme označenie p^i ako prefix s dĺžkou *i*. Jedným zo základných konceptov rýchlej tvorby sufixových stromov sú *sufixové spoje*.

Definícia 3.4: Nech máme reťazec *x*, ktorému odpovedá uzol *u* v sufixovom strome. Sufixový spoj *suf[u]* je uzol, ktorý odpovedá reťazcu *x* s odobraným prvým znakom. Ak taký uzol neexistuje, potom *suff[u] = NULL*, pre koreňový uzol *r* platí *suff[r] = r*.

On-line metódou vytvoríme v každom kroku $i \in 1 \ldots n$ sufixový strom pre pre p^i . Vytvorené stomy neukladáme samostatne, ale strom $T(p^i)$ je v i-tom kroku vytvorený zo stomu $T(p^{i-1})$. Strom sa tvorí postupným pridávaním znakov vstupného textu k *esenciálnym uzlom*. Esenciálny uzol je uzol zodpovedajúci sufixu práve spracovávaného prefixu vstupného textu. Napríklad ak tvoríme sufixový strom pre reťazec *abaabb* v kroku číslo 2 spracovávame prefix *ab*, esenciálne uzly budú uzly zodpovedajúce sufixom *b* a *ab*. Kompletný postup tvorby sufixového stromu pre reťazec *abaabb* jezobrazený na obrázku 3.6. Jednotlivé esenciálne uzly (plné krúžky v grafe) sú pospájané sufixovými spojmi na základe definície 3.4. Pre správnu funkčnosť algoritmu je nutné si v každom kroku uchovávať tabulku sufixových spojov, podľa ktorej postupne pridávame uzly.



Obr 3.6: Postup tvorby on-line stromu pre *abaabb* [3]

V prvom kroku vezmeme prvý prefix reťazca *abaabb* $p^1 = a$. Ten má jeden sufix *a*, koreňový uzol považujeme taktiež za esecnciálny uzol. V druhom kroku spracujeme prefix o jeden znak dlhší

$p^2 = ab$. Tu sa vyskytujú dva sufixy: *b, ab*. Sufix *b* vznikne zo sufixu *ab* odstránením prvého znaku, tým pádom môžeme spojiť uzly značiace tieto dva sufixy sufixovým spojom, ktorý uložíme do tabulky sufixových spojov. Analogicky pokračujeme, kým sú na vstupe nespracované prefixy.

Tvrdenie 3.2: Algoritmus on-line tvorby sufixových stromov vybuduje sufixový strom v čase zodpovedajúcom jeho veľkosti.

Dôkaz: Vyplýva z toho že práca vykonaná v jednej iterácí algoritmu 3.3 zodpovedá počtu pridaných uzlov.

3.2.2 Ukkonenova metóda tvorby sufixových stromov

Ukkonenova metóda [3] vytvorí kompaktnú verziu on-line stromu. Tento strom bude mať lineárnu veľkosť, tým pádom (tvrdenie 3.2) bude časová náročnosť jeho tvorby taktiež lineárna.

Algoritmus 3.3 [3]:

Vstup: Reťazec $t = t_1, t_2, \ldots t_n \in \Sigma^n$

Vytvor strom T(t[1]) so sufixovými spojmi

for $i := 2$ **to** *n* **do**

$a_i = t[i]$

$v_{i-1} := najhlbší\ list\ stromu\ T\ (p^{i-1})$

$k := min\{k : potomok(suf^k[v_{i-1}], a_i) \neq nil\}$

vytvor potomkov cez hranu a_i *pre* $v_{i-1}, suf[v_{i-1}], \ldots suf^k[v_{i-1}]$

a sufixové spoje medzi nimi

Výstup: On-line sufixový strom pre *t*

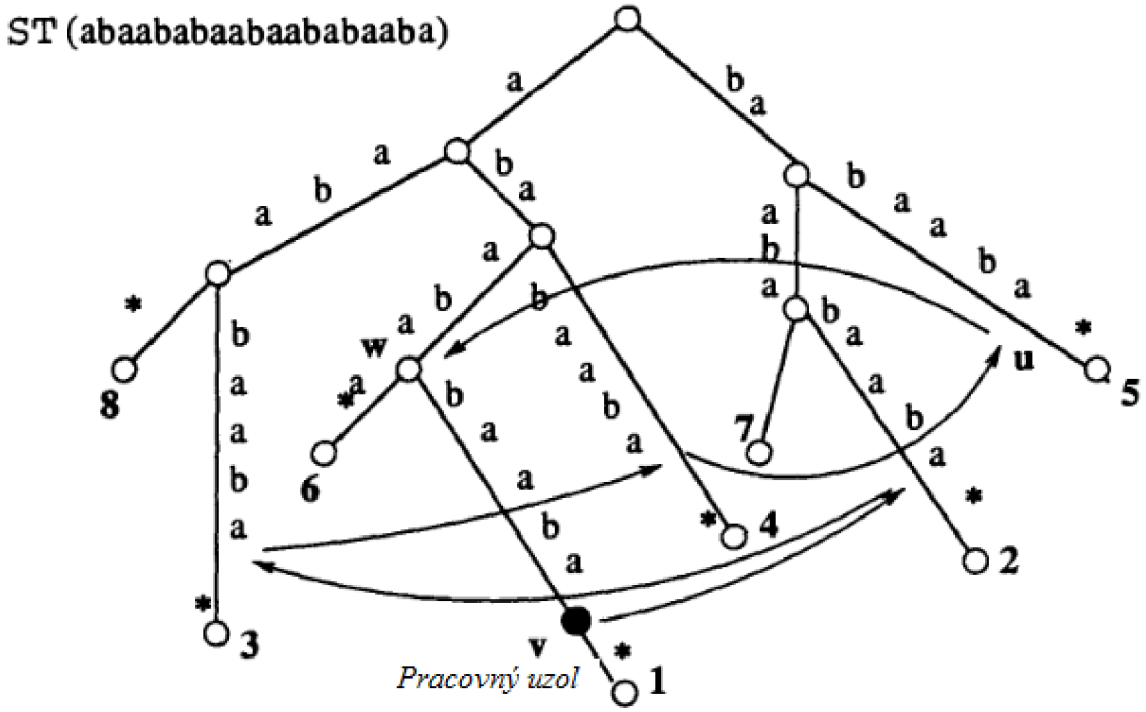
Ukkonenov algoritmus vytvára stromy „v pravom slova zmysle“, narozdiel od stromov (často označovaných ako *trie*), ktorých príklad je v predošlej kapitole. Rozdiel medzi stromami *trie* a *tree* je práve v efektívnejšej reprezentácí. V strome *tree* neexistuje každý uzol zo stromu *trie*. Ak niektorý zo uzol z *trie* zodpovedá nejakému uzlu z *tree* nazveme ho *reálnym*. Ostatné uzly nazývame *implicitné*.

Definícia 3.5: Dvojicu *(u, a)* nazývame implicitným uzlom v strome T (tree), ak *a* je prefixom označenia hrany na ceste od uzlu *u* k ľubovolnému z jeho potomkov. Ak je *a* prázdny reťazec, uzol *u* je uzlom reálnym

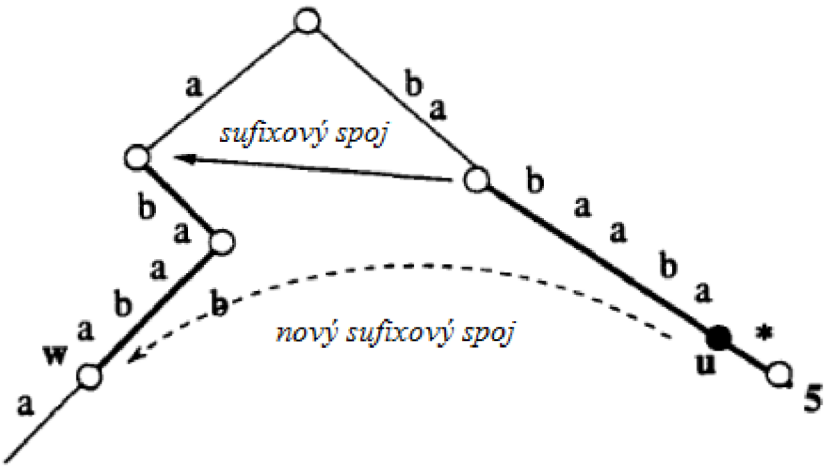
V Ukkonenovom algoritme uchovávame iba najhlbšie vnútorné esenciálne uzly. Esenciálne uzly, z ktorých vystupuje iba jedna hrana, podobne ako v algoritme na začiatku kapitoly, odstraňujeme. Späť ich získame po poslednom kroku pridaním ukončovacieho znaku $\$ \notin \Sigma$. Aj v tomto algoritme sa uplatní metóda skrátenia označenia hrán pomocou pozící vo vstupnom texte. Navyše, pri listových uzloch, používame znak * ako indikáciu, že hrana označená *(l,r)* sa v tomto kroku rozšírila o jeden znak *(l, r+1)*. Znak * slúži na skrátenie tohto zápisu a značí poslednú spracovanú pozíciu vstupného textu. Ak máme sekvenciu esenciaálnych uzlov $v_i, v_{i-1}, \ldots v_0$ a vieme že uzly

$v_i, v_{i-1}, \ldots v_k$ sú listami, môžeme ich spracovanie preskočiť práve na základe metódy značenia s *. Tým pádom začneme pracovať s uzlom v_{k-1}, ktorý nazývame *pracovný uzol*, a ktorý je zároveň najhlbší vnútorný esenciálny uzol (pracovný uzol zodpovedá nejakému sufixu vstupného textu obr. 3.7).
Ďalším dôležitým pojmom sú *implicitné sufixové spoje – imsuf*. Implicitné sufixové spoje sa tvoria pre implicitné uzly a ukazujú na iný implicitný uzol. Pre vytvorenie implicitného sufixového spoja pre uzol *u* je nutné sledovať sufixový spoj jeho otcovského uzlu *v*, a od cieľa tohto sufixového spoju postupovať rovnakou cestou ako od uzlu *v* k uzlu *u* (obrázok 3.8) .

Tvrdenie 3.3: Ukkonenov algoritmus vytvára kompaktný strom *ST(text)* spôsobom on-line konštrukcie v lineárnom čase (na ustálenej abecede).



Obr. 3.7: Znázornenie pracovného uzlu počas tvorby stromu Ukkonenovým algoritmom [3]



Obr. 3.8: Tvorba nových implicitných sufixových spojov

Algoritmus 3.4:

Vstup: Reťazec $t = t_1 t_2 \dots t_n \in \Sigma^n$

Vytvor strom T(t[1]) so sufixovými spojmi

$v := koreň$

for $i := 2$ **to** n **do**

$a_i = t[i]$

if $potomok(v, a_i) \neq nil$ **then** $v := potomok(v, a_i)$

else

$k := min\{k : potomok(imsuf^k[v_{i-1}], a_i) \neq nil\}$

vytvor potomkov cez hranu a_i *pre* $v_{i-1}, suf[v_{i-1}], \dots suf^k[v_{i-1}]$

a imsuf spoje medzi nimi

$v := potomok(imsuf^{k-1}(v), a_i)$

// v je najhlbší vnútorný esenciálny uzol stromu $T(p_{i-1})$

Výstup: On-line kompaktný sufixový strom pre *t*

3.2.3 Hľadanie palindrómov v sufixových stromoch

Pre metódu hľadania palindrómov v sufixových stromoch [4] musíme sufixový strom vytvoriť nielen pre vstupný reťazec $t = t_1 t_2 \dots t_n \in \Sigma^n$ ale aj pre jeho reverzovanú formu $t' = t_n, t_{n-1}, \dots t_1 \in \Sigma^n$. Na konie reťazcov musíme pripojiť znaky, ktoré sa nenachádzajú

v abecede $\$, \$' \notin \Sigma$, aby sme sa vyhli problému nejednoznačnosti sufixových stromov. Ďalej musíme strom doplniť o dodatočné údaje:

1. pre listové uzly index $i_f(i_r)$, ktorý udáva pozíciu sufixu, v *t(t')*

2. pre každý uzol indexy, ktoré udávajú miesto výskytu podreťazca odpovedajúceho danému uzlu vo vstupoch *t, t'* v grafe *(*index_t | *index_t')*

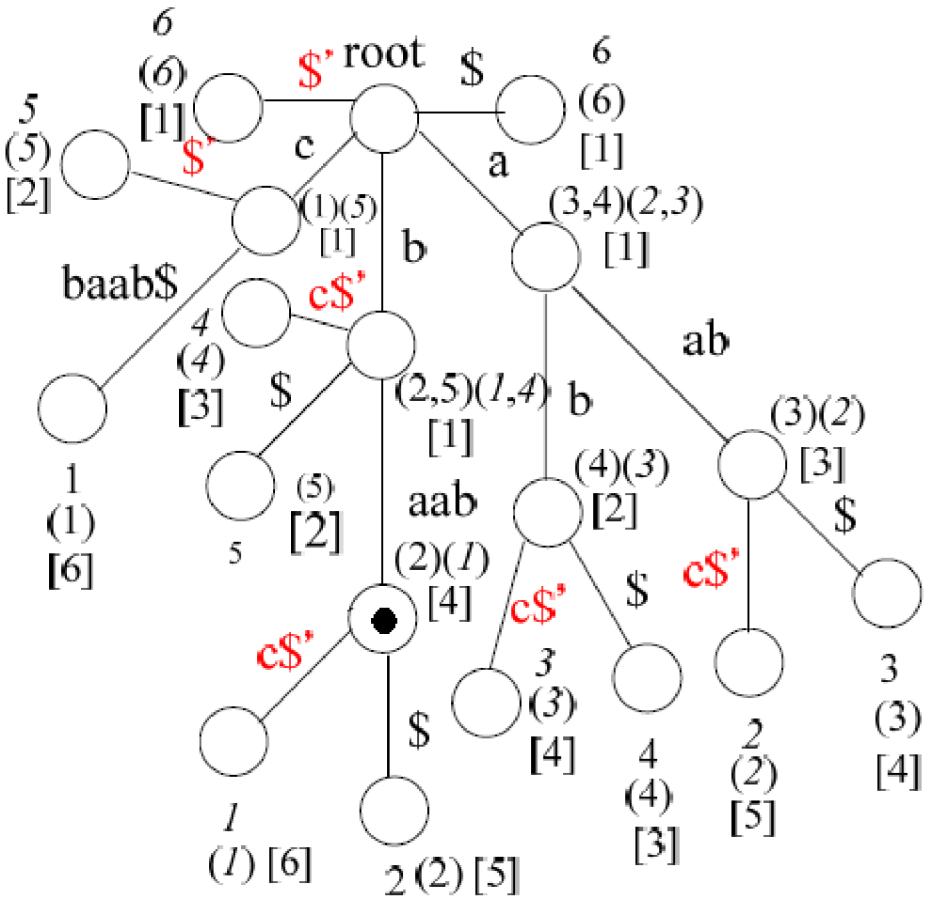
3. pre každý uzol jeho vzdialenosť od koreňa označenú *D(v)*, v grafe v []

Vzorec 3.1: $i_f + i_r = (n - D(v) + 2)$

Vzorec 3.1 udáva podmienku pre výskyt palindrómu v sufixovom strome. Ak táto podmienka platí pre nejaký uzol, reťazec odpovedajúci tomuto uzlu je maximálny palindróm.

Dôkaz: Nech máme maximálny palindróm na pozící i_f s dĺžkou *k* vo vstupnom reťazci *t* s dĺžkou *n*. Potom musí existovať maximálny palindróm na pozící i_r v *t'*. Potom môžeme povedať, že $i_f - 1 + k + i_r - 1 = n$. V našom sufixovom strome *k = D(V)*. Po úprave a dosadení dostávame vzorec $i_f + i_r = (n - D(v) + 2)$.

Príklad je uvedený na obrázku 3.9 na vzorovom reťazci *cbaab*. Uzol vyznačený bodkou spĺňa podmienku výskytu palindrómu, $i_f = 2 \quad i_r = 1 \quad n = 5 \quad D(v) = 4 \quad 2 + 1 = 5 - 4 + 2$.



Obr. 3.9: Sufixový strom pre reťazec *t=cbaab* a *t'* s vyznačeným palindrómom [4]

Algoritmus 3.5 [4]:

Vstup: Reťazec $t = t_1 t_2 \dots t_n \in \Sigma^n$

Vytvor sufixový strom pre *t$* a *t'$'*

Pre každý uzol vypočítaj *D(v)*, $i_f(i_r)$ a pozíciu v *t(t')*

Pre každý uzol otestuj podmienku $i_f + i_r = (n - D(v) + 2)$, ak ju spĺňa, vráť reťazec odpovedajúci sanému uzlu

Výstup: Množina maximálnych palindrómov

4 Vyhľadávanie približných palindrómov

Približné palindrómy, na rozdiel od presných obsahujú chyby. Budeme uvažovať tri druhy chýb: insertion, deletion, substitution uvedených v kapitole 1. Týmito chybami sa od seba odlišujú symetrické polovice palindrómov. Aby sme zistili, či je daný reťazec približným palindrómom, musíme zistiť, či je možné ho pomocou postupnej eliminácie chýb transformovať na presný palindróm. Takto by sme mohli teoreticky ľubovolný reťazec transformovať na palindróm, preto je nutné si určiť maximály počet chýb.

4.1 Princíp dynamického programovania

Na riešenie problému vyhľadávania približných palindrómov je možné aplikovať princípy dynamického programovania (DP) [5]. Základné pravidlá DP sú:

1. Nájdi štruktúru pre optimálne riešenie
2. Rekurzívne definuj hodnotu optimálneho riešenia
3. Vypočítaj hodnotu optimálneho riešenia metódou zdola-nahor
4. Zostroj optimálne riešenie pomocou hodnôt vypočítaných v kroku 3

Kroky 1-3 tvoria riešenie problému pomocou DP a akrok 4 predstavuje spätné vyhľadanie optimálneho riešenia, ak je vyžadované. Z bodov č. 2 a 3 môžeme odvodiť, že DP rozkladá problém na ekvivalentné, rýchlo riešiteľné podproblémy.

4.2 Aplikácia princípov DP na hľadanie približných palindrómov

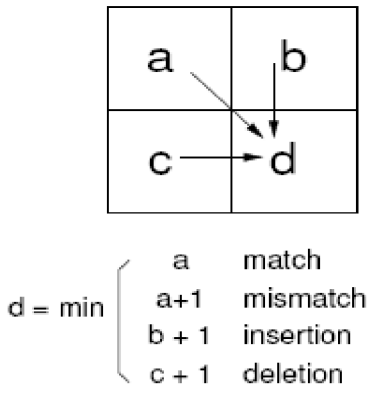
Podľa pravidiel DP v predošlej podkapitole, je nutné najprv nájsť optimálnu štruktúru pre riešenie problému. Pre náš problém – vyhľadávanie palindrómov – predstavuje ideálnu štruktúru **DP matica** [6]. Takáto matica má stĺpce indexované znakmi reťazca, v ktorom palindrómy hľadáme a riadky má indexované znakmi reverzovanej formy tohto reťazca. V DP matici sa stred každého palinrómu nachádza na hlavnej antidiagonále – pre nepárne palindrómy, alebo antidiagonále, ktorá s ňou bezprostredne susedí – pre párne palindrómy (obr. 4.1). Problém, ktorý je nutné vyriešiť je nájdenie pozície v tejto matici, kde sa nejaký palidróm končí. Aby sme túto pozíciu našli, musíme sa vedieť po matici pohybovať. Pohyb po DP matici predstavuje chyby, resp. zhody v palindróme. Na začiatku kapitoly sme si uviedli tri druhy chýb – insetrion, deletion, substitution – každej z nich prísluší pohyb v matici jedným smerom. Pri chybe typu insertion sa pohybujeme smerom dolu, pri chybe deletion smerom doprava a pri chybe substitution, rovnako ako aj pri zhode smerom doprava-

dolu. Prečo je tomu tak? Je treba si uvedomiť, že pri pohybe maticou sa pohybujeme aj cez prehľadávaný reťazec aj cez jeho reverzovanú formu. Ak sa pohneme doprava, zmeníme pozíciu v prehľadávanom reťazci, ale nie v jeho reverzovanej forme. Vo svojej podstate je tento krok podobný odstráneniu jedného znaku z prehľadávaného reťazca. Pri pohybe dolu je situácia obrátená – pohneme sa v reverzovanej forme reťazca, čo sa podobá vymazaniu znaku z reverzovanej formy. Ak reverzujeme aj operáciu vymazania znaku, dostaneme operáciu pridania znaku – insertion. Zmenou pozície smerom doprava a nadol zároveň (smer hlavnej diagonály) zmeníme pozíciu v prehľadávanom reťazci aj jeho reverzovanej forme. Ak sa indexované znaky na novej pozící nezhodujú, jdená sa o zámenu znaku, chybu substitution, avšak ak sa znaky zhodujú, žiadna chyba nenastala.

Jednoduchý pohyb po matici ale žiadny problém nevyrieši. Každú zmenu políčka je treba nejako ohodnotiť. Táto potreba vyplýva aj z toho, že inak by sme chybu substitution neboli shopný odlíšiť od zhody znakov. V tomto prípade ohodnotíme každú chybu jedným bodom – jeden bod bude predstavovať jednu chybu – a zhodu ohodnotíme nula bodmi.

	M	I	S	S	I	S	S	I	P	P	I
I										0	0
P									0	0	
P								0	0		
I							0	0			
S						0	0				
S					0	0					
I				0	0						
S			0	0							
S		0	0								
I	0	0									
M	0										

Obr. 4.1: DP matica reťazca mississippi



Obr. 4.2: Krok DP [6]

Keď už vieme ako sa pohybovať, je otázne, ktorým smerom. Našim cieľom je nájsť čo najdlhšie palindrómy, s čo najmenším počtom chýb. To znamená, že sa v matici musíme dostať čo najďalej, na políčko s čo najmenším ohodnotením. Táto úloha sa dá rozložiť na menšie podúlohy – dostať sa na ďalšie políčko s väčším indexom tak, aby sme ho ohodnotili najmenším možným počtom chýb. Na políčko je možné sa dostať pohybom zhora, zľava, a kombináciou pohybov zhora a zľava. Výber z týchto troch možností nazývame **krok DP** (obr. 4.2). Jeden takýto krok postupne učiníme pre každé políčko v DP matici pod hlavnou diagonálou. Výsledok po jednej sérií DP krokov bude vyzerať takto:

	M	I	S	S	I	S	S	I	P	P	I
I										0	0
P									0	0	1
P								0	0	0	
I							0	0	1		
S						0	0	1			
S					0	0	0				
I				0	0	1					
S			0	0	1						
S		0	0	0							
I	0	0	1								
M	0	1									

Obr. 4.3: DP matica po jednej sérií DP krokov

DP kroky aplikujeme až dovtedy, kým nevyplníme celú spodnú časť DP matice (obr. 4.4). Takto vyplnená DP matica obsahuje začiatočné a konečné pozície všetkých presných aj približných palindrómov. Kde sa nachádzajú? Začiatky sa nachádzajú – ako už bolo vyššie spomenuté – na hlavnej resp. vedlajšej antidiagonále a konce palindrómov teoreticky všade. Každé políčko môže predstavovať palindróm s počtom chýb zodpovedajúcim hodnote políčka. Nás však nezaujímajú všetky palindrómy, ale iba tie najdlhšie s najmenším počtom chýb. Presné palidrómy – tie čo majú na celom svojom priebehu ohodnotenie 0 – sa v našej matici nachádzajú štyri a sú vyznačené na obrázku 4.4 modrou farbou. Všetky najdlhšie približné palindrómy môžeme potom nájsť ďalším pohybom po matici od konečných pozící presných palindrómov – vyznačené šedou farbou.

Teraz máme nájdené všetky zaujímavé palindrómy v DP matici. V niektorých prípadoch stačí vedieť počet nájdených palindrómov, ale čo ak chceme vedieť ako vyzerajú? Presnú podobu palindrómov získame spätným priechodom ohodnotenej matice. Začneme na koncovej pozící palindrómu a postupujeme smerom nahor. V každom kroku spätného prehladania sa presunieme na políčko o index nižšie s najmenším ohodnotením. Takto vieme získať presnú podobu všetkých palindrómov.

Späť k princípom dynamického programovania. Najprv sme našli ideálnu štruktúru pre reprezentáciu nášeho problému: DP maticu. Následne sme si určili, že riešením problému hľadania približných palindrómov sú pozície v DP matici s najvyšším indexom a najmenším ohodnotením. Potom sme rozložili náš problém na sériu DP krokov a metódou zdola-nahor vyplnili DP maticu. Nakoniec sme spätne prešli maticu a získali optimálne riešenie – dodržanie všetkých základných pravidiel DP. Tento spôsob však nie je bez optimalizácí veľmi efektívny. Pred zhodnotením jeho nárokov a aplikáciou optimalizácí je vhodné vysvetliť pojem sufixové polia.

	M	I	S	S	I	S	S	I	P	P	I
I										0	0
P									0	0	1
P								0	0	0	1
I							0	0	1	1	0
S						0	0	1	1	2	1
S					0	0	0	1	2	2	2
I				0	0	1	1	0	1	2	3
S			0	0	1	0	1	1	1	2	3
S		0	0	0	1	1	0	1	2	2	3
I	0	0	1	1	0	1	1	0	1	2	2
M	0	1	1	2	1	1	2	1	1	2	3

Obr. 4.4: Vyplnená DP matica

4.3 Najdlhší spoločný prefix a sufixové polia

V minulej kapitole sme si ukázali, ako nájsť pomocou metód DP palindrómy v reťazci. Problém DP prístupu predstavuje veľký počet krokov, ktoré je treba vykonať. Existujú však metódy ako počet krokov zredukovať. Metóda [7], ktorá tu bude uvedená využíva sufixové polia na vyhľadávanie najdlhšieho spoločného prefixu (*lognest common prefix* - *lcp*) dvoch reťazcov. Prečo nám vyhľadávanie lcp pomôže? Symetrické časti palindrómov v DP matici sú vlastne lcp sufixov reťazca a jeho reverzovanej formy, ktoré začínajú na indexoch políčka DP matice. Ak by sme mohli zistiť dĺžku týchto úsekov v konštantnom čase významne by to urýchlilo prechod DP maticou.

4.3.1 Sufixové pole

Sufixové pole reťazca *S* je zoradené pole *Pos(1..n)* sufixov reťazca *S*. Pole *Pos* obsahuje sufixy lexikograficky zoradené za sebou a platí $Pos[k]=i$, pričom S_i je sufix začínajúci v reťazci *S* na pozícii *i* a lexikograficky patrí na *k-te* miesto. Sufixové pole je svojou vyjadrovacou schopnosťou ekvivalentné sufixovému stromu, avšak s lepšou priestorovou náročnosťou. Pole *Pos* je v podstate lexikograficky zoradený zoznam listov sufixového stromu. Na obrázku 4.5 je znázornené sufixové pole reťazca mississippi.

1 *i*
2 *ippi*
3 *issippi*
4 *ississippi*
5 *mississippi*
6 *pi*
7 *ppi*
8 *sippi*
9 *sissippi*
10 *ssippi*
11 *ssissippi*

Obr. 4.5: Sufixové pole reťazca mississippi

4.3.2 Najdlhší spoločný prefix

Pre efektívny výpočet lcp pomocou sufixových polí zavádzame dve dodatočné polia. Pole *Height* obsahujúce *lcp* dvoch susediacich sufixov v poli *Pos:* $Height[k]=lcp(Pos[k-1], Pos[k])$ pričom *k* patrí do rozmedzia $2 \leq k \leq n$. Pomocou pola *Height* je možné vypočítať *lcp* pre ktorúkoľvek dvojicu sufixov v lineárnom čase. Pole *Rank* je pomocné pole, ktoré slúži na výpočet hodnôt pola *Height*. Platí, že ak $Pos[i]=k$ tak $Rank[k]=i$ - pole *Pos* udáva pozíciu *k* lexikograficky *i*-teho sufixu vo vstupnom reťazci, naopak pole *Rank* udáva pozíciu *i* sufixu, ktorý sa nachádza vo vstupnom reťazci na *k*-tom mieste, v poli *Pos*. Na obrázku 4.6 je znázornené sufixové pole reťazca mississippi aj s polami *Height* a *Rank*.

i	*pos[i]*	*rank[i]*	*height[i]*
1	11	5	0
2	8	4	1
3	5	11	1
4	2	9	4
5	1	3	0
6	10	10	0
7	9	8	1
8	7	2	0
9	4	7	2
10	6	6	1
11	3	1	3

Obr. 4.6: Polia *Pos, Rank* a *Height* reťazca mississippi

Za riešenie problému nájdenia *lcp* dvoch sufixov budeme považovať vypočítanie hodnôt poľa *Height*. Aby ho bolo možné efektívne doplniť, je nutné si uvedomiť niekoľko faktov ohladne *lcp* v súvislosti so sufixovým polom. Platí, že *lcp* dvoch sufixov v poli *Pos* je minimom z *lcp* všetkých susediacich dvojíc sufixov medzi nimi. Tento fakt vyplýva z usporiadania sufixov v poli *Pos*.

Príklad:

i	*Sufix*	*Height*
1	sippi	0
2	sissippi	2
3	ssippi	1
4	ssissippi	3

V tabulke sú uvedené štyri sufixy reťazca missippi: *sippi, sissippi, ssippi, ssissippi*. Ak chceme zistiť *lcp* medzi sufixami 1 a 4 stačí nájsť minimum dĺžok medzi nimi v poli *Height*, čo je 1 a to predstavuje aj dĺžku *lcp* pre tieto dva sufixy.

Z tohto tvrdenia je možné ďalej odvodiť, že *lcp* dvoch susedných sufixov je rovnako dlhý alebo dlhší ako *lcp* susedných dvojíc sufixov, ktoré skúmané sufixy obkolesujú.

Príklad:

i	*Sufix*	*Height*
1	i	0
2	ippi	1
3	issippi	1
4	ississipps	4
5	mississippi	0
6	pi	0

Tvrdenie platí pre každý riadok tabulky.

Ak je *lcp* dvoch susedných prefixov väčší ako 1, ak odstránime prvý znak z oboch sufixov, ich vzájomné lexikografické poradie sa nezmení. Zároveň odstránením prvého znaku skrátime aj *lcp* týchto dvoch sufixov o jeden znak.

Príklad:

Nech máme sufixy *sippi* a *sissippi*. Ich *lcp* má dĺžku 2. Po odstránení prvého znaku dostávame sufixy *ippi* a *issippi*. Ich lexikografické poradie je zachované a ich *lcp* má dĺžku 1.

Tieto skutočnosti je možné využiť pri riešení nasledujúceho problému: Chceme vypočítať *lcp* nejakého sufixu S_i a jeho susediaceho sufixu v poli *Pos*, pričom poznáme dĺžku *lcp* sufixu S_{i-1} a jeho susediaceho prefixu v poli *Pos*. Vieme že sufix S_i je možné získať zo sufixu S_{i-1} odstránením prvého znaku. Za predpokladu, že dĺžka *l* najdlhšieho spoločného prefixu sufixu S_{i-1} a jeho susediaceho sufixu je väčšia ako 1, môžeme na základe vlastností *lcp* tvrdiť, že dĺžka *lcp* sufixu S_i a jeho susediaceho sufixu je rovnaká, alebo väčšia ako $l-1$, čiže $Height[i] \geq Height[i-1]-1$.

Príklad:

Máme reťazec $S = mississippi$. Chceme zistiť dĺžku *lcp* sufixu $S_3 = ssissippi$ a jeho susediaceho sufixu $S_{Pos[10]} = ssippi$ a vieme, že sufix $S_2 = ississippi$ so svojím susedným sufixom $S_{Pos[3]} = issippi$ má dĺžku *lcp* 4. Po aplikovaní pravidla $Height[i] \geq Height[i-1]-1$ zistíme, že najmenšia dĺžka *lcp* pre S_3 je 3, takže prvé tri znaky nemusíme porovnávať a preskočíme ich. Pri porovnávaní ďalšieho odhalujeme že sa odlišuje, tým pádom zisťujeme, že dĺžka *lcp* pre sufix S_3 je 3 po jedinom porovnaní.

4.4 Implementácia vyhľadávania palindrómov pomocou DP princípov

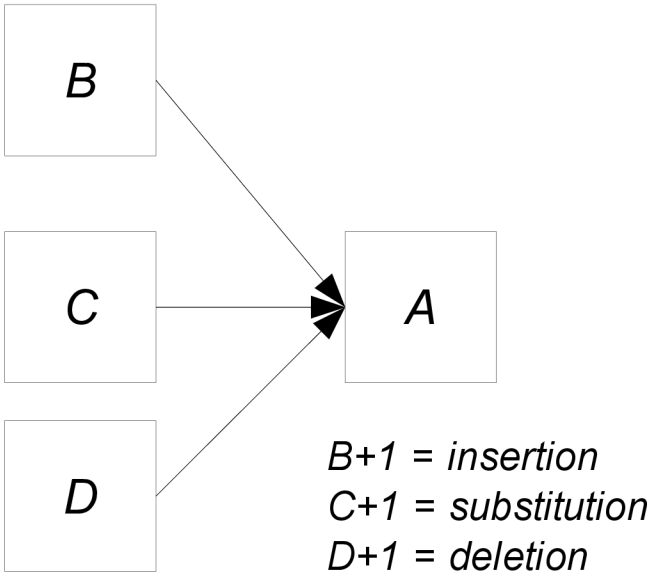
V podkapitole 4.2 sme si uviedli spôsob ohodnotenia DP matice pomocou počtu chýb. Pri implementácí bola využitá metóda[8], ktorá vytvára ekvivalentnú DP maticu, ktorej políčka nie sú ohodnotené počtom chýb, ale dosiahnutou dĺžkou palindrómu (obr. 4.6). V tejto matici sú riadky

indexované poradovým číslom diagonály a stĺpce počtom chýb, ktoré palindróm obsahuje. Potom hodnota políčka s indexami *i* a *j* hovorí, že palidróm na diagonále *i* pri počte chýb *j* dosahuje dĺžku $DP[i, j]$.

Dôvodov na prechod na tento typ matice je hneď niekoľko. Prvým dôvodom je **priestorová náročnosť**. Pôvodná metóda má priestorovú náročnosť $\frac{n^2-2}{2}$ (*n* je dĺžka prehľadávaného reťazca), za predpokladu, že použijeme iba spodnú časť matice. Aby sme presne určili priestorovú náročnosť alternatívnej metódy je nutné ju presne zanalyzovať. Bez akejkoľvek optimalizácie, by jej veľkosť rástla s počtom chýb, čiže by odpovedala vzorcu $(2n-3)*e$, pričom *n* je dĺžka prehľadávaného reťazca a *e* je počet chýb. Výsledná zložitosť by bola pri veľkom počte chýb ešte horšia ako u pôvodnej metódy. Otázka: „Potrebujeme uchovávať informáciu o každom kroku?“ Odpovede sú dve. Ak chceme spraviť spätný prechod maticou, tak áno, ale ak chceme iba nájsť palindrómy a rekonštrukciu vykonáme oddelene, odpoveď znie nie. Táto skutočnosť vyplýva zo štruktúry DP kroku (obr. 4.7). Napriek tomu, že sa výzor trochu zmenil, funkcia zostáva rovnaká. Na výpočet hodnoty políčka DP matice je potrebné poznať hodnoty okolitých políčok. V našej matici sú všetky potrebné hodnoty obsiahnuté v predošlom stĺpci. Tým pádom potrebujeme iba dva stĺpce matice: aktuálny a stĺpec predošlého kroku. Týmto spôsobom sa zredukuje priestorová zložitosť na $(2n-3)*2$.

	0	1	2	3
1	0	1	1	1
2	0	1	1	1
3	2	2	2	2
4	0	2	2	2
5	0	1	3	3
6	0	1	2	3
7	0	1	3	4
8	0	2	3	4
9	2	3	5	5
10	0	2	4	5
11	0	4	5	5
12	3	4	4	4
13	0	4	4	4
14	0	2	3	3
15	2	3	3	3
16	0	2	2	2
17	0	1	2	2
18	0	1	1	1
19	0	1	1	1

Obr. 4.7: Alternatívna DP matica pre reťazec mississippi



Obr. 4.8: DP krok alternatívnej matice

Druhým dôvodom pre zmenu prístupu je prípadná **aplikácia optimalizácie**. V predošlej podkapitole sme si uviedli princípy výpočtu *lcp* pomocou sufixových polí. Za použitia správnych dátových štruktúr je možné potom získať *lcp* dvoch ľubovolných sufixov v konštantnom čase. Túto hodnotu je potom možné po každom DP kroku pripočítať k ohodnoteniu políčka matice.

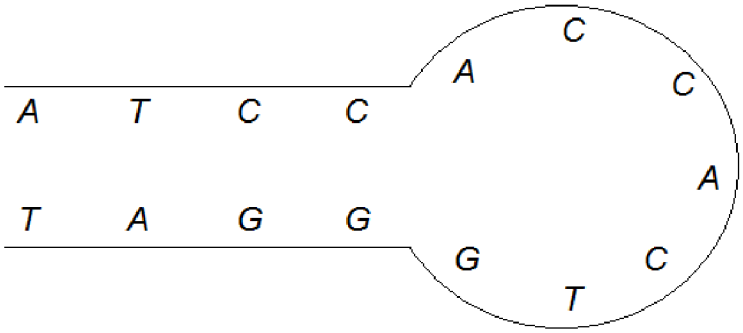
Vlastný proces vyhľadania palinrómov je rozdelený do troch programov – vyhľadávací program, filter výsledkov a program rekonštrukcie výsledkov, ktoré sú prepojené pomocou rúr (pipes) a ich spoločný výstup je množina nájdených zrekonštruovaných palindrómov. Každému programu je ďalej venovaná jedna sekcia.

4.4.1 Implementačné detaily vyhľadávacieho programu

Vstup programu tvorí súbor, zadaný ako parameter programu, ktorý obsahuje prehľadávaný reťazec. Výstup programu je štvorica hodnôt udávajúca poradové číslo diagonály, na ktorej sa palindróm končí, dĺžku palindrómu, počet chýb v palindróme a dĺžku asymetrickej časti palinrómu (jej problematika bude bližšie rozobratá neskôr v tejto podkapitole), pre každý nájdený palindróm.

Program najprv načíta vstupný reťazec s ukončovcím znakom ('\0') a uloží ho do pamäti. Potom k reťazcu pripojí jeho reverzovanú formu ukončenú ukončovacím znakom. Posledný inicializačný krok pred samotným výpočtom je alokovanie kompaktnej formy DP matice.

Pojem asymetrická časť reťazca označuje úsek asymetrický úsek palindrómu medzi jeho symetrickými časťami. V DNA reťazcoch vytvára „sľučku“ v strede palindrómu (obr. 4.8). Všetky asymetrické časti palindrómov sa v DP matici vyskytujú na začiatku diagonál. Existujú dva prístupy ako spracovať tieto úseky. Prvý prístup vychádza z **globálneho zarovnania**. Celá asymetrická časť sa ohodnotí, ako keby to bol iba reťazec chýb. Kvalita palindrómu (vysvetlené pri časti exportovania palindrómov) sa tým pádom zhorší o daný počet chýb. Uvedené správanie nie je ideálne – došlo by k neobjektívnemu hodnoteniu palindrómov. Druhý prístup sa opiera o **lokálne zarovnanie**. Chyby asymetrickej časti sa rátajú ako zhody, tým pádom nedojde k zhoršeniu kvality palindrómu. Na druhej strane je v tomto prípade palindróm nadhodnotený.



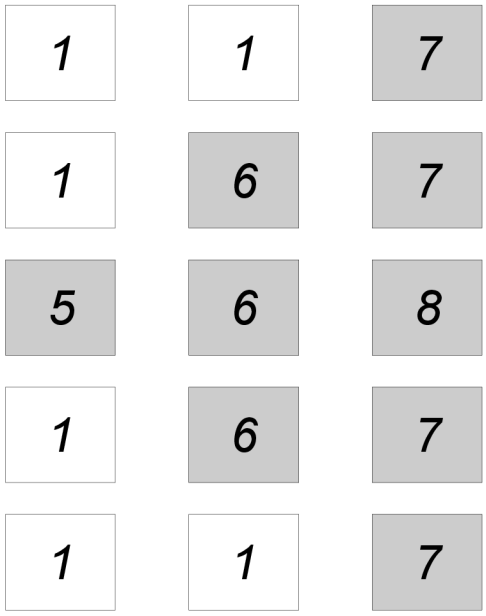
Obr. 4.9: Asymetrická časť palindrómu

Ideálne riešenie vzniknutého problému je nájsť asimetrické časti na všetkých diagonálach ešte pred začatím ohodnocovania matice a uložiť si ich do pomocného poľa. Ak od dĺžky palindrómu (lokálne zarovnanie) alebo počtu chýb (globálne zarovnanie) odpočítame položku pomocného pola, dostaneme skutočný počet chýb palindrómu. Spomínané riešenie ešte nie je konečné. Vzhľadom na to, že informácia o asymetrickej časti je viazaná na diagonálu a nie na palindróm, nastáva problém pri chybách typu *deletion* a *insertion*, keď palindróm preskočí medzi diagonálami. Vtedy je nutné zaistiť, aby sa spolu s palindrómom preniesla aj hodnota asymetrickej časti. Operácia prenosu hodnoty dĺžky asymetrickej časti nie je taká jednoduchá, ako sa môže na prvý pohľad zdať. Pri kroku DP sa jednoducho vyberie maximálna hodnota z troch možných a tá sa priradí ohodnocovanému políčku. Nie je potrebné vedieť, z ktorej diagonály sa hodnota preniesla. Aby sme mohli preniesť aj hodnotu dĺžky asymetrickej časti, musíme presne vedieť, z ktorej diagonály palindróm preskočil. Potrebný výpočet nie je nijak zložitý, ale obsahuje vetvenie, ktoré sa nachádza v hlavnej sľučke programu – narastá výpočetná náročnosť.

Vyhľadávené palindrómy už majú správne vypočítanú kvalitu, ale čo je **kvalita palindrómu**? Jednoducho povedané počet chýb v pomere k dĺžke palidrómu, čiže $q = n/e$, pričom q je kvalita palindrómu, n je jeho dĺžka a e je počet chýb. Kvalita je dôležitá v zmysle výstupu ako aj v zmysle ukončenia behu programu. Kedy by mal program skončiť? Vtedy, ak už neexistuje palindróm, ktorý by spĺňal požiadavky na kvalitu. Kedy by sa mal palindróm exportovať? Ak by sme exportovali všetky palindrómy na ktoré narazíme v priebehu ohodnocovania matice, zátaž na filtrovací program by bola obrovská. Zároveň by sme vyviezli aj veľa palindrómov s rovnakým základom, ktoré vznikli vetvením. Preto je lepšie skontrolovať, či v blízkom okolí palindrómu – okolie jednej diagonály – sa nenachádza iný palindróm s rovnakým alebo lepším ohodnotením. Odôvodnenie spomenutého opatrenia je priblížené v nasledujúcom príklade.

Príklad:

Nech máme stĺpec DP matice a jeho rozvinutie po dvoch sériách DP krokov.



Obr. 4.10: Rozširovanie palindrómu

Vidíme, že v prvej sérí sa vyskytuje dobré ohodnotenie palindrómu iba na jednej diagobále. V druhej sérí sa podľa princípu DP kroku prenesie dobré ohodnotenie aj na vedľajšie diagonály. V tretej sérí sa situácia opakuje. Takto vzniklo päť približných palindrómov, ktoré majú spoločný základ a sú dostatočne kvalitné na export. Ak ale zavedieme pravidlo, že sa bude palindróm exportovať iba vtedy, ak sa na vedlajších diagonálach nevyskytujú lepšie, alebo rovnako ohodnotené palindrómy, exportuje sa iba jeden palindróm.

Na uvedenom príklade je znázornená dôležitosť merania kvality palindrómu a tým pádom aj správneho hodnotenia asymetrickej časti.

Algoritmus 4.1: Vyhľadávanie palindrómov pomocou DP matice

Vstup: Prehľadávaný reťazec

načítaj reťazec *r*

vytvor reverzovanú r' formu a spoj s reťazcom *r*

vytvor sufixové pole pre konkatenáciu rr'

alokuj DP maticu *reach*, pomocné pole *rank* a pole asymetrických častí *asym*

for $i = 0$ **to** length(*reach*) **do**

while ($r[ind1] \neq r'[ind2]$) **do**

$asym[i] = asym[i] + 1$

$ind1 = ind1 + 1$

$ind2 = ind2 + 1$

$reach[i] = asym[i] + lcp(rr', ind1, ind2)$

while existuje palindróm dostatočnej kvality **do**

$e = e + 1$

for $i = 0$ **to** length(*reach*) **do**

$reach[e][i] = max(reach[e-1][i-1], reach[e-1][i], reach[e+1][i])$

$reach[e][i] = reach[e][i] + lcp(rr', ind1, ind2)$

if ($reach[e][i]$ vyhovuje kvalite)

exportuj $reach[e][i]$

uvoľni zdroje

Výstup: Množina palidrómov

Algoritmus neukazuje ošetrenie hraníc a výpočet indexov. Pre presnejšie vyjadrenie pozrite prílohy so zdrojovými kódmi.

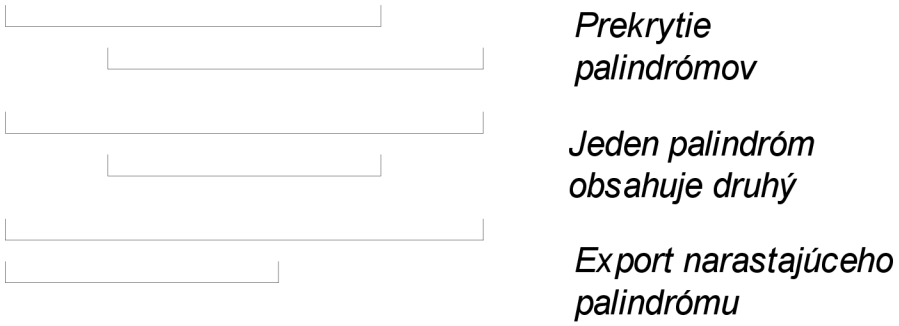
4.4.2 Analýza náročnosti vyhľadávacieho programu

Program dosahuje lineárnu pamäťovú zložitosť, čo je esenciálny fakt pre všetky programy pracujúce s veľkým množstvom dát. Presné vyjadrenie zložitosti je $11n-10$, pričom n je dĺžka prehľadávaného reťazca. Časová zložitosť programu závisí na vstupných dátach. Hlavný cyklus programu sa ukončí, ak už neexistuje palindróm požadovanej kvality. Tým pádom počet iterácií obvplyvňuú nielen vlastnosti vstupného reťazca, ale aj uživatelský vstup – určenie požadovanej kvality. Algoritmus taktiež obsahuje okrem cyklu samotného výočtu aj inicializačný cyklus. V inicializačnom cykle sa hľadá asymetrická časť palindrómov. Ak by bol na vstupe reťazec bez palindrómov, v uvedenej forme by cyklus prebehol $2*(\sum_{i=1}^{n-1} n/2-i)+n/2$ krát – najhorší prípad. V implementácí bol uvedený horný limit asymetrickej časti 200 znakov. Hlavný výpočetný cyklus programu prebehne $E*(2n-3)$ krát, pričom E je maximálny počet chýb vyhovujúceho palindrómu. Výpočet kroku DP a *lcp* neprebehne vždy. Táto skutočnosť je spôsobená rôznou dĺžkou diagonál. Ak index presiahne dĺžku diagonály, výpočet sa preskakuje a novému políčku v DP matici sa pripadí hodnota z predošlého kroku. Vzhľadom na to, že v programe neboli použité žiadne optimalizáce pre vyhľadanie *lcp*, najhoršia zložitosť programu je $\frac{n^2-2}{2}$. Výsledky experimentov s programom je možné nájsť v prílohe.

4.4.3 Implementačné detaily filtru výsledkov

Vyhľadávací program nájde veľké množstvo palindrómov, ktoré sa nejakým spôsobom prekrývajú. Najčastejšie sa stáva, že palindróm na dlhom úseku spĺňa požiadavku na kvalitu a v každej iterácí sa exportuje jeho dlhšia verzia. Pre tieto prípady platí, že neskôr exportovaný palindróm v sebe obsahuje celý palindróm exportovaný v predošlom kroku. Iná možnosť je, že palindrómy na rôznych diagonálach sa prekrývajú. V prípade prekrytia je nutné rozhodnúť, ktorému palindrómu patrý zdielaná časť. Posledná často vyskytujúca sa možnosť je, že palindróm obsahuje v sebe iný palindróm exportovaný z odlišnej diagonály. Túto situáciu lepšie objasní príklad.

Príklad:
Nech máme reťazec *mississippi*. Exportované palindrómy s nula chybami budú: *issi* na pozící 2 vstupného reťazca – onačenie *p1*, *issi* na pozící 5 vstupného reťazca – označenie *p2*, *ippi* na pozící 8 vstupného reťazca – označenie *p3* a nakoniec *ississi* na pozící 2 vstupného reťazca – označenie *p4*. Vidíme že palindróm *p4* obsahuje prvé dva exportované palindrómy a prekrýva sa s palindrómom *p3* jedným znakom. Všetky palindrómy pritom začínajú na rôznych diagonálach (obr. 4.4). Podľa správnosti by sa palindrómy *p1* a *p2* nemali exportovať a palindróm *p3* podľa toho, či prekrytie o jeden znak povolíme alebo nie.



Obr. 4.11: Znázornenie prekrytia palindrómov

Pri tvorbe správne fungujúceho filtru je nutné všetky predvedené prípady odstrániť. V posledných dvoch situáciách znázornených na obrázku 4.11 je postup jednoduchý. Palindróm, ktorý má začiatočnú pozíciu s väčším indexom a koncovú pozíciu s menším indexom jako ktorýkoľvek doteraz akceptovaný palindróm neprejde filtrom. V prvej znázornenej situácí je třeba rozhodnúť, ktorý palindróm bude lepšie vybrať. Aby sme boli schopný sa rozhodnúť, je nutné si určiť kritérium, na základe ktorého budeme palindrómy hodnotiť a lepšie ohodnotený palindróm prejde. Pre implementáciu bolo zvolené kritérium kvality palindrómu v zmysle pomeru počtu chýb na dĺžku. Všetky potrené informácie sú obsiahnuté vo výstupe prvého programu a výpočet je pomerne jednoduchý. Pojem kvalita palindrómu je bližšie popísaný v predošlej podkapitole.

Algoritmus 4.2:

Vstup: informácie o palindróme (diagonále, dĺžka, počet chýb, asymetrická časť)

```
while údaje na vstupe do
    zisti počiatočný index, koncový index a kvalitu palindrómu
    for každý palindróm vyhovujúci filtru do
        if jeden palindróm obsahuje druhý do
            vyber väčší palindróm
        else if palindrómy sa prekrývajú do
            vyber palindróm s lepšou kvalitou
        else do
            vyber oba palindrómy
        ulož vybrané palindrómy do pola vyhovujúcich palidrómov
```

Výstup: Množina vyhovujúcich palindrómov

4.4.4 Analýza náročnosti filtru

Podobne ako pre vyhľadávaci program, aj vlastnosti filtru sú obvplyvnené z veľkej časti vlastnosťami vstupu. Ak vstupný reťazec obsahuje veľké množstvo prekrývajúcich-sa palindrómov, filter prepustí, a tým pádom bude porovnávať, menšiu množinu palindrómov. Toto tvrdenie sa vsťahuje na priestorovú aj na časovú zložitosť. Časovú náročnosť je možné vyjadriť vsťahom $\sum_{i=1}^{n} p_i$, pričom n je počet vstupných palindrómov a p_i je počet prvkov množiny vyhovujúcich palindrómov po i analyzovaných vstupných palindrómoch. Celková pameťová náročnosť je potom p_n . Príklad chovania a efektívnosti filtru je uvedený v prílohe.

4.4.5 Implementačné detaily rekonštrukcie palindrómov

Rekonštrukcia nájedných palindrómov je posledným krokom ich získavania. V niektorých prípadoch stačí poznať iba pozíciu a veľkosť palindrómu, ale niekedy je potrebné poznať presnú štruktúru približného palindrómu aj so všetkými zmazanými alebo vloženými znakmi. V programe vyhľadávania palindrómov, za predpokladu uchovania celej DP matice by bola možná okamžitá rekonštrukcia získaných palindrómov. Problémom by bola veľkosť matice. Ak by sme analyzovali reťazec o dĺžke desať miliónov znakov a najdlhší palindróm by mal napríklad sto chýb, iba DP matica by obsahovala $(10000000-3)*100=999999700$ políčok. Za predpokladu že jedno políčko je položka typu *long* s veľkosťou 8 bytov, celková veľkosť zabraného priestoru by bola viac ako 7GB. Do tejto hodnoty nie sú zarátané žiadne ďalšie pomocné premenné. Dané požiadavky sú na obyčajné počítače príliš veľké, preto je lepšie krok rekonštrukcie separovať.

Oddelenie rekonštrukcie však zo sebou prináša iný problém. Je treba vypočítať DP maticu odznova. Matica sa však nemusí počítať celá. Je postačujúce aby sme vypočítali DP maticu pre samostatné palindrómy. Týmto spôsobom bude pameťová náročnosť zodpovedať veľkosti DP matice pre najväčší palindróm. Otázne je jako majú vyzerať jednotlivé PD matice pre palindrómy. Jedna z možností je vytvoriť kompletnú DP maticu pre palindróm indexovanú jeho prvou a druhou symetrickou časťou (obr. 4.12). Narozdiel od vyhľadávacej matice v prvom programe, musíme vypočítať kompletný obsah matice, nie iba jej spodnú časť. Tento prístup však ignoruje fakt, že nevieme kde palindróm začína. Vieme iba index diagonály na ktorej končí, jeho dĺžku a počet chýb. Z týchto údajov je možné si spočítať pozíciu konca, ale nie je možné zistiť, aké chyby sa v palindróme vyskytujú.

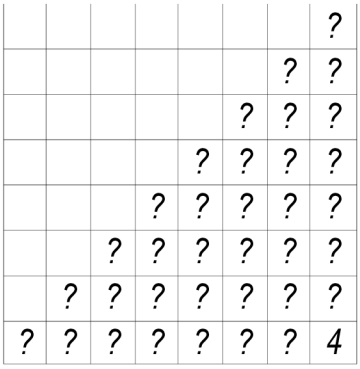
	I	S	A	S	I
I	0	1	2	3	3
S	1	0	1	2	3
S	2	1	1	1	2
I	2	2	2	2	1

Obr. 4.12: Rekonštručná DP matica pre palindróm *isasissi*

Príklad:

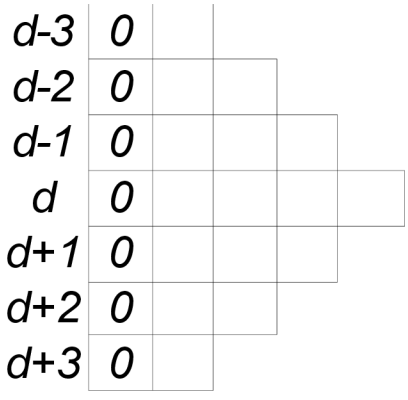
Máme palindróm, ktorý končí na diagonále *d*, má dĺžku *n* a obsahuje *e* chýb. Ak by boli všetky chyby typu insertion, palindróm by začínal na diagonále *d*, ak by boli všetky chyby typu insertion,

palindróm by začínal na diagonále $d+e$, naopak, ak by boli všetky chyby typu deletion, palindróm by začínal na diagonále $d-e$. Chyby môžu byť ľubovolne kombinovateľné, takže dostávame 3^e možností a $2e+1$ možných začiatočných pozící. To je veľmi veľká neurčitosť.



Obr. 4.13: Neurčitosť postupu

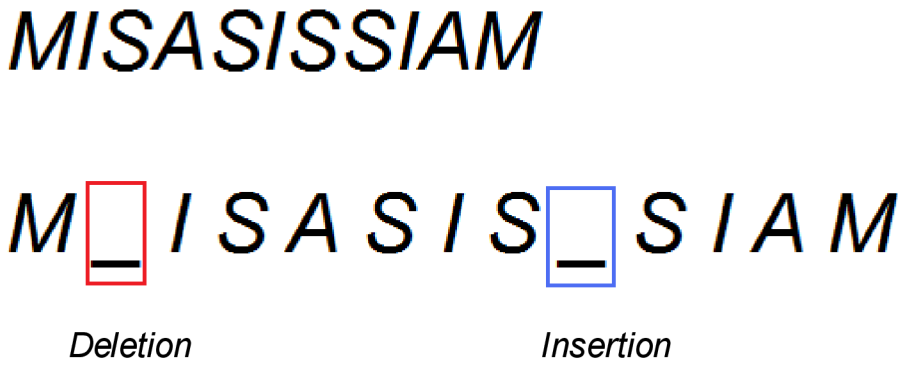
Druhý spôsob rekonštrukcie sa s podobným problémom nestretáva. Za prepokladu, že poznáme prehľadávaný reťazec, si vytvoríme výsek DP matice. Pre palindróm dĺžky *n* s *e* chybami na diagonále *d* bude výsledok zahŕňať diagonály indexov $d-e$ až $d+e$, pričom dĺžky jednotlivých riadkov budú $n/2$ pre diagonálu *d*, $n/2-1...e$ pre diagonály $d-1...d-e$, $n/2+1...e$ pre diagonálu $d+1...d+e$. Pre vyplnenie matice môžeme zvoliť prítup popísaný na začiatku kapitoly, ale aj alternatívu použitú v prvom programe. Pre implementáciu bol vybraný prvý uvedený spôsob – značí sa počet chýb, každá chyba je hodnotená jedným bodom a zhoda nula bodmi (obr. 4.14 a 4.15). Rekonštrukcia potom prebieha od posledného políčka na diagonále *d* a postupuje smerom k začiatku matice. V každom kroku sa volí posun k najlepšie ohodnotenému políčku. Pri rekonštrukcí nemusíme uvažovať asymetrické časti. Môžu byť ohodnotené chybami, pritom sa správnosť algoritmu nenaruší. Posledný problém predstavuje spôsob rekonštrukcie chýb typu *insertion* a *deletion*. Základ tvorí vkladanie znakov, ktoré sa nenachádzajú v abecede vstupného reťazca do výstupného palindrómu. Miesto vloženia závisí od indexovania pôvodnej DP matice. Ak boli stĺpce indexované pôvodným reťazcom a riadky jeho reverzovanou formou, znaky pre insertion vkladáme do pravej symetrickej časti a znaky pre deletion do ľavej symetrickej časti (obr. 4.16).



Obr. 4.14: Príklad rekonštrukčnej matice

d-3	0	1			
d-2	0	1	2		
d-1	0	0	0	0	
d	0	1	2	2	1
d+1	0	1	2	3	
d+2	0	1	2		
d+3	0	1			

Obr. 4.15: Príklad vyplnenej rekonštrukčnej matice



Obr. 4.16: Ukážka reťazca obsahujúceho chyby *deletion* a *insertion*

Algoritmus 4.3:

Vstup: množina odfiltrovaných palindrómov a analyzovaný reťazec

načítaj vstupný reťazec
while je palindróm na vstupe **do**
  alokuj a inicializuj DP maticu
  for i=1 **to** dĺžka palindrómu **do**
    for j=0 **to** počet chýb palindrómu **do**
      DP krok
  for i=dĺžka palindrómu **to** 0 **do**
    invertovaný DP krok
  uvoľni zdroje

Výstup: množina zrekonštruovaných palindrómov

4.4.6 Analýza náročnosti rekonštrukčného programu

Rekonštrukčný program pracuje v jednej chvíli iba s jedným palindrómom. Jeho priestorová náročnosť nikdy nebude vyššia, ako nároky na najväčší palindróm. Maximálna aj minimálna priestorová zložitosť sa dá vyjadriť vsťahom $n+2*\sum_{i=1}^{e}(l/2-i)+l/2+l+e+1$, pričom n je dĺžka analyzovaného reťazca, e je počet chýb palindrómu, l je dĺžka palindrómu. Časová zložitosť je možné približne vyjadriť vsťahom $i*(e*(l/2)+l+e+1)$, pričom i je počet palindrómov na vstupe. Približná zložitosť je to z dôvodu, že vnútro cyklu prebehne iba pre indexy do matice, ktoré nie sú za hranicou dĺžky jednotlivých diagonál. Reálna časová zložitosť je nižšia oproti uvedenej. Analýzu výstupu je možné nájsť v prílohe.

5 Záver

Algoritmy pre vyhľadávanie približných palindrómov sú perfektným príkladom dôležitosti správneho spojenia výpočetných algoritmov a dátových štruktúr. Použitím matice dynamického programovania bolo možné pomocou opakovania elementárnej operácie výberu maxima resp. minima (podľa zvoleného prístupu) dosiahnuť nájdenie všetkých približných palindrómov v prehľadávanom reťazci. Implementovaný program má lineárnu priestorovú a, v najhoršom prípade, kvadratickú časovú zložitosť. Na program je možné v budúcnosti aplikovať optimalizáciu vyhľadávania najdlhšieho spoločného prefixu. V súčasnosti sú známe postupy, ktoré umožnujú čas potrebný na vyhľadanie najdlhšieho spoločného prefixu skrátiť z lineárneho na konštantný. Ďalšie optimalizácie by bolo potrebné aplikovať na proces filtrovania nájdených palindrómov. Experimentami bolo zistené, že práve filtrovací program spotrebuje najväčšie množstvo času. Jendou možnou nepriamou optimalizáciou, je sprísnenie kritérí exportovania palindrómov vo vyhľadávacom programe. Inou možnosťou je sprísniť kritériá pre palindrómy. Možnými kritériami sú pravdepodobnosť výskytu v náhodnej postupnosti znakov, alebo pravdepodobnosť výskytu v súvislosti s chemickými a biologickými väzbami medzi bázami DNA reťazcov. Zlepšovanie rýchlosti algoritmov pracujúcich s veľkými množstvami dát je základom pre ďalšie možné napredovanie analýzy týchto dát, a práve analýza reťazcov DNA, či už zvieracích alebo ľudských, je jedna z potrebných a rozvíjajúcich sa oblastí informatiky a biológie. Každý predsa chceme vedieť o sebe maximum, aby sme mohli poznať naše slabosti, silné stránky, alebo z čistej ľudskej zvedavosti.

Literatúra

[1] Jirkovský, V. *Vyhledávání palindromů.* Praha: Fakulta elektrotechnická ČVUT, január 2008. Diplomová práca.

[2] Bockenhauer, H.-J. a Bongartz, D. *Algorithmic Aspects of Bioinformatics (Natural computing series)*. Rozenberg, G. et al. edit. Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-71912-0.

[3] Crochemore, M. Rytter, W. *Jewels of stringology*. Singapore: Worl Scientific Publishing, 2002. ISBN 981-02-4782-6.

[4] Shih Jang Pang a Lee R.C.T. Looking for all palindromes in a string. In *The 23rd Workshop on Combinatorial Mathematics and Computation theory*. Department of Computer Science and Information Engineering, National Chi-Nan University, Puli, Nantou Hsien, 545, Taiwan, 2006. S. 166 – 169.

[5] Buhmann, J. H. *Wissenschaftliches rechnen – kernfach.* [online]. [cit. 28. apríl 2010]. Dostupné na <http://www.inf.ethz.ch/personal/vroth/wrk/stma4_6_volker.pdf>.

[6] Martínek, T. a Lexa, M. *Hardware Acceleration of Approximate Palindromes Searching*. Fakulta informačných technológí VUT Brno, Božetěchova 2, Brno, 612 66, Fakulta informatiky Masarykova univerzita, Botanická 68a, Brno, 602 00, Česká republika.

[7] Kasai, T. Arimura, H. et al. *Linear-Time Longest-Common-Prefix Computation in Sufix Arrays and Its Applications.* Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan, School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea, PRESTO, Japan Science and Technology Corporation, Japan.

[8] Allison, L. *Finding Approximate Palindromes in Strings Quickly and Simply.* School of Computer Science and Software Engineering, Monash University, Clayton, Victoria, Australia 3800, 24. Nov. 2004.

Zoznam príloh

Príloha 1.

Výsledky testovania programu na prvých 853455 znakoch chromozómu č. 19 potkana rodu *Rattus norvegicus* obsiahnuté v CD prílohe.

Vyhľadávací program:

kvalita palindrómu	*5*
inicializačné iterácie	*6100749*
iterácie hlavného cyklu	*18775955*
porovnávaní	*32977292*
palindrómov	*139442*
kvalita palindrómu	*10*
inicializačné iterácie	*6100749*
iterácie hlavného cyklu	*6827620*
porovnávaní	*17391862*
palindrómov	*1195*
kvalita palindrómu	*20*
inicializačné iterácie	*6100749*
iterácie hlavného cyklu	*3413810*
porovnávaní	*12939582*
palindrómov	*54*
kvalita palindrómu	*30*
inicializačné iterácie	*6100749*
iterácie hlavného cyklu	*6827620*
porovnávaní	*12939582*
palindrómov	*0*

Filter:

vstupný počet palindrómov	*54*
výstupný počet palindrómov	*6*
porovnaní	*255*
vstupný počet palindrómov	*1195*
výstupný počet palindrómov	*660*
porovnaní	*415065*
vstupný počet palindrómov	*24536*
výstupný počet palindrómov	*12070*
*porovnaní **	*190756621*

* program bol predčasne ukončený po 15 minútach behu

Rekonštrukčný program:

počet palindrómov na vstupe	*6*
iterácie hlavného cyklu	*6*
iterácie DP cyklu	*327*
iterácie backtrack cyklu	*110*
počet palindrómov na vstupe	*660*
iterácie hlavného cyklu	*660*
iterácie DP cyklu	*25769*
iterácie backtrack cyklu	*8597*