VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

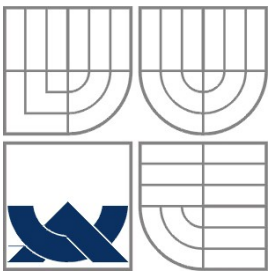
# PRIBLIŽNÉ VYHĽADÁVANIE REŤAZCOV V PREDSPRACOVANÝCH DOKUMENTOCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

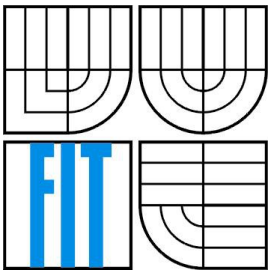
AUTOR PRÁCE
AUTHOR

Bc. RÓBERT TOTH

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# PŘIBLIŽNÉ VYHLEDÁVÁNÍ ŘETĚZCŮ V PŘEDSPRACOVANÝCH DOKUMENTECH

APPROXIMATE STRING MATCHING IN PREPROCESSED DOCUMENTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RÓBERT TOTH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KAŠTIL

BRNO 2014

# Abstrakt

Tato práce se zabývá problémem přibližného vyhledávání řetězců, označovaným též jako vyhledávání s chybami. Práce se zaměřuje na oblast offline algoritmů, které umožňují po jednorazovém předspracování textu velmi rychlé vyhledávaní díky indexu, který si nad textem vytvoří. Nejprve bude definován problém samotný a demonstrována rozmanitost jeho využití, následována krátkým shrnutím rozdílných přístupů k této problematice. Poté budou detailně probrány některé algoritmy založené na použití suffixových stromů a představen nový hybridní algoritmus. Algoritmy budou implementovány v jazyce C a jejich výkonnost detailně otestována v sérii experimentů se zaměřením na určení reálného přínosu nového algoritmu do této oblasti.



# Abstract

This thesis deals with the problem of approximate string matching, also called string matching allowing errors. The thesis targets the area of offline algorithms, which allows very fast pattern matching thanks to index created during initial text preprocessing phase. Initially, we will define the problem itself and demonstrate variety of its applications, followed by short survey of different approaches to cope with this problem. Several existing algorithms based on suffix trees will be explained in detail and new hybrid algorithm will be proposed. Algorithms wil be implemented in C programming language and thoroughly compared in series of experiments with focus on newly presented algorithm.



# Klíčová slova

přibližné vyhledávání řetězců, vyhledávání s chybami, suffixové stromy, indexování textu, indexační algoritmy, offline algoritmy



# Keywords

approximate string matching, inexact searching, searching allowing errors, suffix trees, suffix arrays, text indexing, text indexing algoritms, offline algorithms



# Citace

Róbert Toth: Približné vyhľadávanie reťazcov v predspracovaných dokumentoch, diplomová práce, Brno, FIT VUT v Brně, 2014

# Približné vyhľadávanie reťazcov v predspracovaných dokumentoch

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Róbert Toth
15. května 2014

## Poděkování

Rád by som sa poďakoval Ing. Janu Kaštilovi za jeho vedenie při práci a za to, že svojím vlastným nasadením ešte väčšmi podnietil môj záujem o túto problematiku. Ďakujem tiež celej svojej rodine, Kolégiu Antona Neuwirtha za neopakovateľné študijné podmienky a všetkým priateľom, ktorí ma posúvajú ďalej na ceste životom.

# Obsah

# 1 Úvod

*„Na vyhľadávaní je úžasné to, že ho v najbližšej dobe určite nedovedieme k dokonalosti. Je tu toľko úskalí a toľko nedostatkov. Mám pocit, že s tým, čo je potrebné urobiť, nebudeme nikdy hotoví.“*

Larry Page, spoluzakladateľ spoločnosti Google

Človek od nepamäti túži po poznaní a pochopení sveta okolo seba. Už od vzniku prvých písiem sa pritom snažil všetky svoje vedomosti zaznamenať pre budúce generácie aj písomne. Postupom času tak boli založené knižnice, v ktorých sa poznatky ľudstva zbierali a následne triedili a zaraďovali. Azda práve preto sa objavila aj potreba jednoduchého *vyhľadávania* podľa autora či názvu diela, takže vznikali rozličné zoznamy a kartotéky, ktoré pomáhali toto hľadanie urýchliť.

S nástupom moderných technológií začali byť k tomuto účelu využívané počítače. Tie vyhľadávanie nielen mnohonásobne urýchlili a rozšírili o ďalšie kritériá, ale so vzrastajúcou výpočtovou rýchlosťou umožnili aj vyhľadávanie v celých textoch. Časom tak vznikli mnohé algoritmy na vyhľadávanie reťazcov v texte (tzv. *string matching algorithms*). Ich aplikácia je pritom veľmi široká: od bežnej kancelárskej práce, cez vyhľadávanie súborov na pevných diskoch, až po hľadanie kníh v databázach knižníc. V praxi sa však väčšinou stretávame len s presným vyhľadávaním textu (*exact string matching*).

Táto práca pojednáva o približnom vyhľadávaní reťazcov (*approximate string matching*), ktoré je generalizáciou presného vyhľadávania a umožňuje v texte nájsť aj reťazce určitým spôsobom podobné hľadanému. I keď je takéto vyhľadávanie výpočtovo náročnejšie ako presné, poskytuje užívateľovi nespornú výhodu v podobe rýchlejšej a príjemnejšej práce, pretože mu umožňuje nájsť hľadaný reťazec aj v prípade výskytu pravopisných chýb, preklepov či iných poškodení.

Práca sa zameriava na oblasť tzv. *indexačných* alebo *offline algoritmov*, ktoré umožňujú jednorazovo predspracovať text do určitých dátových štruktúr, vďaka ktorým je potom dosahované veľmi rýchle nájdenie vyhľadávaných pojmov. Cieľom práce je zmapovať existujúce riešenia a navrhnúť vylepšený algoritmus. Ten bude následne spolu s vybranými už existujúcimi algoritmami implementovaný a ich výkonnosť dôkladne porovnaná v sérii testov.

V kapitole 2 sa zoznámime s problémom približného vyhľadávania ako takého, uvedieme jeho možné aplikácie a definujeme základné pojmy. Následne predstavíme základný algoritmus na približné vyhľadávanie a uvedieme krátky prehľad rozličných prístupov k tejto problematike a tiež odkazy na ďalšie prehľadové zdroje v tejto oblasti.

Cieľom kapitoly 3 je v krátkosti predstaviť vybrané offline algoritmy a demonštrovať tak základné princípy používané pri indexovaní textu pre účely vyhľadávania. Kapitola 4 potom obsahuje podrobný popis nového hybridného algoritmu, ktorého implementácia spolu s ostatnými algoritmami je popísaná v kapitole 5 a podrobne otestovaná v 6. kapitole. Záver obsahuje celkové vyhodnotenie práce a návrhy na jej ďalšie možné pokračovanie.

# 2 Približné vyhľadávanie reťazcov

Problém približného vyhľadávania reťazcov môžeme chápať ako úlohu nájsť v zadanom texte všetky pozície hľadaného reťazca, umožňujúc pritom výskyt určitého počtu chýb v každej zhode. Každý druh chyby môže mať svoju vlastnú váhu, ktorá určuje jej závažnosť, alebo môžu byť všetky chyby váhované uniformne. Miera odlišnosti zadaného reťazca od nájdeného sa potom určí ako súčet váh všetkých chýb nachádzajúcich sa v zhode.

Presné vyhľadávanie tak môžeme považovať za špeciálny prípad, kedy sa povolený počet chýb —a teda aj miera odlišnosti zadaného reťazca od nájdeného—rovná nule.

## 2.1 Aplikácia

Použitie klasických algoritmov pre vyhľadávanie reťazcov v širšom spektre úloh je limitované nutnosťou mať k dispozícii dáta bez akýchkoľvek poškodení či chýb. V reálnom aj digitálnom svete sa však prirodzene vyskytujú údaje v určitej miere poškodené alebo nepresné.

Či už sú chyby v údajoch spôsobené informačným šumom, zlou interpretáciou údajov, nepresným prepisom, zaokrúhľovaním hodnôt alebo chybou parity, presné vyhľadávanie v týchto dátach nie je možné bez ich sanitácie. Tá však často vyžaduje neúmerné množstvo času alebo ju v zásade nie je možné zautomatizovať.

Práve umožnenie výskytu chýb pri približnom vyhľadávaní rozširuje možnú aplikáciu takto upravených algoritmov na mnohé ďalšie oblasti:

- **výpočtová biológia:** vyhľadávanie DNA sekvencií po možných mutáciách, rozpoznávanie evolučne príbuzných sekvencií, či rekonštrukcia fylogenetického stromu pomocou ohodnotenia miery „rozdielnosti“ sekvencií;
- **spracovanie signálov:** získavanie originálneho signálu po jeho prenose stratovým kanálom, rozpoznávanie hlasu, či detekcia slov v hlasovom prejave;
- **analýza a spracovanie textu:** korekcia preklepov v písanom texte, optické rozpoznávanie znakov (OCR skenery), porovnávanie súborov (`*NIX` utilita `diff`), či rozpoznávanie ručne písaného textu;
- **analýza sieťových tokov:** popri presnej detekcii vírusových sekvencií v routeroch a firewalloch je možná aj heuristická;
- **… a mnohé iné,** napríklad detekcia spamu v spamových filtroch alebo identifikácia hudobnej nahrávky na základe jej (poškodenej) časti.

Široký prehľad možných využití približného vyhľadávania aj s odkazmi na pôvodné práce je možné nájsť v [16] a [6].

## 2.2 Definícia problému

Približné vyhľadávanie reťazcov môžeme formálne definovať nasledovne:

> Nech $\Sigma$ je konečná *abeceda* veľkosti $|\Sigma|$.
>
> Nech $T \in \Sigma^*$ je *text* dĺžky $n$, kde $T_{1...n}$ sú jednotlivé *znaky* textu.
>
> Nech $P \in \Sigma^*$ je *vzorka* (ďalej len *pattern*) dĺžky $m$, kde $P_{1...m}$ sú jednotlivé *znaky* patternu.
>
> Nech $k \in \mathbb{R}$ je maximálna povolená *editačná vzdialenosť* (edit distance) medzi hľadaným a nájdeným reťazcom. Ďalej označme $\alpha = k/m$ ako *pomer chyby* (error ratio).
>
> Nech $\delta$ je množina *editačných operácií* v tvare $\delta(z, w) = t$, kde $t \geq 0$ je *cena operácie*.
>
> Nech $\mathrm{d}(x, y)$ je *funkcia editačnej vzdialenosti* (distance function).
>
> Potom pri zadaných $T$, $P$, $k$ a $\mathrm{d}()$ je úlohou nájsť všetky také podreťazce $q \in T$, že $\mathrm{d}(P, q) \leq k$.

Máme teda nájsť pozície všetkých reťazcov $q$ v texte $T$, ktorých editačná vzdialenosť (ďalej len vzdialenosť) od zadaného patternu $P$ je menšia alebo rovná povolenej vzdialenosti $k$.

Funkcia editačnej vzdialenosti $\mathrm{d}(P, q)$ ohodnocuje vzdialenosť ako minimálnu cenu sekvencie editačných operácií (ďalej len operácií) $\delta$ transformujúcich $P$ na $q$. Táto vzdialenosť je teda súčtom cien všetkých aplikovaných operácií, rovnajúca sa $\infty$ v prípade, že taká sekvencia neexistuje (inak povedané, daná transformácia nie je možná).

Množina operácií v $\delta$ závisí od charakteru riešeného problému a chýb, ktoré je potrebné brať do úvahy. Cena každej operácie vyjadruje váhu daného druhu chyby. Podľa množiny povolených operácií potom rozlišujeme viaceré *modely editačných vzdialeností*.

### 2.2.1 Modely editačných vzdialeností

Medzi základné editačné operácie patria:

- Inzercia $\delta(\varepsilon, a)$, teda vloženie znaku $a$.
- Delécia $\delta(a, \varepsilon)$, teda vymazanie znaku $a$.
- Substitúcia alebo editácia $\delta(a, b)$ pre $a \neq b$, teda nahradenie znaku $a$ za $b$.
- Transpozícia $\delta(ab, ba)$ pre $a \neq b$, teda výmena susedných znakov $a$ a $b$.

Jednotlivé modely sa líšia v podpore týchto operácií. Výber konkrétneho modelu sčasti determinuje aj škálu použiteľných algoritmov, nakoľko väčšina z nich umožňuje implementovať len niektoré z modelov. Tabuľka 2.1 demonštruje rozdiely medzi modelmi na príkladoch vzdialeností niektorých patternov od reťazca.

**Levenshteinova vzdialenosť**

Prvýkrát definovaná v [10], ďalšie definície v [28] a [29]. Umožňuje inzercie, delécie a substitúcie a je zároveň najčastejšie používaným modelom. V literatúre sa často označuje aj ako „vyhľadávanie reťazcov $k$ odlišnosťami“ (*string matching with $k$ differences*) alebo len „editačná vzdialenosť“[1]. V často používanej zjednodušenej verzii (tzv. *unit-cost edit distance*) má každá operácia rovnakú cenu `1`.

**Damerau–Levenshteinova vzdialenosť**

Odlišuje sa od Levenshteinovej vzdialenosti pridanou podporou transpozícií. [4] definuje túto vzdialenosť a udáva, že 80% pravopisných chýb je možné opraviť pomocou tohoto modelu jedinou operáciou.

**Hammingova vzdialenosť**

Umožňuje len substitúciu. Pôvodne definovaná v [7] pre detekciu bitových chýb („substitúcií“ `0→1` a `1→0`) pri dátových prenosoch, neskôr používaná aj pri približnom vyhľadávaní [12]. V literatúre nazývaná aj „vyhľadávanie reťazcov s $k$ nezhodami“ (*string matching with $k$ mismatches*).

**Ďalšie vzdialenosti**

Medzi ďalšie používané vzdialenosti patrí „epizódna vzdialenosť“ (*episode distance*), povoľujúca len inzercie v cene `1`, alebo „vzdialenosť najdlhšiej spoločnej podsekvencie“ (*longest common subsequence distance*). Povoľuje len operácie inzercie a delécie, obe v cene `1`. Vzdialenosť sa v tomto prípade rovná množstvu „nespárovateľných“ znakov.

| Model | Pattern (vzdialenosti od reťazca „ALPHABET“) | | |
|---|---|---|---|
| | ALPABET | ALPHIBBET | APLAHBET |
| Levenshtein | 1 | 2 | 3 |
| Damerau–Levenshtein | 1 | 2 | 2 |
| Hamming | ∞ | ∞ | 4 |
| Epizódna vzdialenosť | 1 | ∞ | ∞ |
| Spol. podsekvencia | 1 | 3 | 4 |

*Tabuľka 2.1: Porovnanie vzdialenosti niektorých patternov od reťazca „`ALPHABET`“ v jednotlivých modeloch editačných vzdialeností.*

V tejto práci budeme ďalej štandardne uvažovať zjednodušený Levenshteinov model vzdialenosti s jednotnou cenou operácií `1`. V prípade použitia iného modelu bude tento fakt explicitne uvedený v texte.

1 Hoci pojem *editačná vzdialenosť* slúži na všeobecné označenie vzdialenosti dvoch reťazcov, často je ním označovaný práve model Levenshteinovej vzdialenosti.

## 2.3 Základný algoritmus

Prvý algoritmus na približné vyhľadávanie reťazcov v texte, tzv. *Sellersov algoritmus*, bol uverejnený v roku 1980 [20]. Je založený na použití tzv. dynamickej programovacej tabuľky (*dynamic programming table*), čo je matica veľkosti $(m+1)\times(n+1)$, kde každá hodnota $C_{i,j}$ reprezentuje vzdialenosť $P_{1\ldots i}$ od $T_{1\ldots j}$. Viac informácií o dynamickej matici je možné nájsť v [6], prípadne v [13]—druhý zdroj okrem popisu obsahuje aj Java applet s voliteľnými parametrami, zobrazujúci výpočet matice po jednotlivých krokoch pre ľubovoľný pattern a text.

### 2.3.1 Výpočet

Jednotlivé hodnoty v matici sú počítané podľa vzorca 2.1, v ktorom uvažujeme jednotnú cenu pre všetky operácie. Algoritmus najprv podľa prvého a druhého vzťahu inicializuje hodnoty v nultom stĺpci a riadku. Vďaka nulám v nultom riadku je každá pozícia v texte potenciálnym začiatkom zhody s patternom—inak povedané, na každej pozícií textu sa v editačnej vzdialenosti 0 vyskytne zhoda s prázdnym patternom $\varepsilon$.

$$\begin{aligned} C_{i,0} &= i && \text{pre } 0 \le i \le m \\ C_{0,j} &= 0 && \text{pre } 0 \le j \le n \\ C_{i,j} &= \begin{array}{l} \text{if } (P_i = T_j) \text{ then } C_{i-1,j-1} \\ \text{else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) \end{array} && \text{pre } 1 \le i \le m, 1 \le j \le n \end{aligned} \tag{2.1}$$

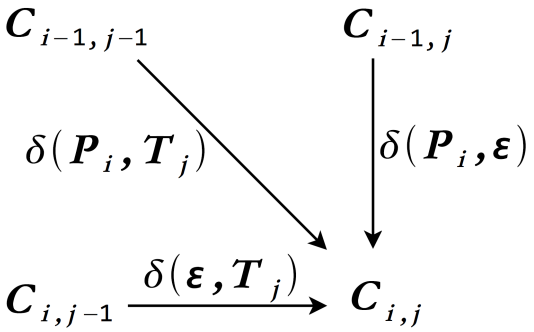
Následne sa podľa posledného vzťahu prechádza zvyšok matice. Pre výpočet každej hodnoty $C_{i,j}$ je nutné poznať tri susedné hodnoty (vľavo hore, hore a naľavo od $C_{i,j}$), preto prebieha výpočet po riadkoch vždy zľava doprava (prípadne je možné prechádzať maticu aj po stĺpcoch zhora dole).

Ak sa znaky $P_i$ a $T_j$ zhodujú, algoritmus použije hodnotu z $C_{i-1,\,j-1}$, čo zodpovedá posunu vyhľadávania o jeden znak v patterne aj texte.

Ak sa však znaky nezhodujú, je nutné „opraviť“ pattern najvýhodnejšou editačnou operáciou. V tomto prípade sa použije najnižšia z troch susedných hodnôt a navýši sa o cenu zodpovedajúcej operácie podľa obrázku 2.1.

Použitie každej hodnoty teda zodpovedá aplikácii konkrétnej editačnej operácie na pattern v danej pozícii:

- $C_{i,\,j-1}$ **inzercii** $T_j$ medzi $P_{i-1}$ a $P_i$ (posun o jeden znak v texte, ale nie v patterne),
- $C_{i-1,\,j}$ **delécii** znaku $P_i$ (posun o jeden znak v patterne, ale nie v texte),
- a $C_{i-1,\,j-1}$ **substitúcii** znaku $P_i$ za $T_j$ (posun o jeden znak v patterne aj v texte).




*Obrázok 2.1: Použitie susedných hodnôt pri editačných operáciách.*

### 2.3.2 Odlišné ceny operácií

Z obrázku 2.1 vyplýva, že v prípade odlišných cien jednotlivých operácií stačí inicializovať nultý stĺpec matice podľa ceny delécie. Pri počítaní matice sa potom použije tretí vzťah zo vzorca 2.1 upravený tak, aby tieto ceny zohľadňoval. Po tejto úprave dostávame vzťah 2.2:

$$C_{i,j} = \textit{if } P_i = T_j \textit{ then } C_{i-1,j-1} \textit{ else } \min \begin{cases} C_{i-1,j} & + \delta(P_i, \varepsilon) \\ C_{i,j-1} & + \delta(\varepsilon, T_j) \\ C_{i-1,j-1} & + \delta(P_i, T_j) \end{cases} \quad (2.2)$$

### 2.3.3 Výsledná matica

Príklad matice po prebehnutí celého algoritmu (s cenou operácií `1`) na konkrétnom patterne a texte znázorňuje tabuľka 2.2. V matici je dôležitý hlavne posledný riadok, jeho hodnoty označujú editačnú vzdialenosť patternu od textu na danej pozícii. Každá hodnota $C_{m,j}$ menšia alebo rovná vzdialenosti $k$ označuje konečnú pozíciu $j$ približnej zhody patternu $P$ v texte $T$. Povolená vzdialenosť bola v tomto prípade `2` a ku zhode došlo na pozíciách `6` a `14` (obe vo vzdialenosti `2`).

| $C_{i,j}$ | | s | a | m | p | l | e | _ | s | t | e | e | p | l | e |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| s | 1 | **0** | 1 | 1 | 1 | 1 | 1 | 1 | **0** | 1 | 1 | 1 | 1 | 1 | 1 |
| t | 2 | **1** | **1** | 2 | 2 | 2 | 2 | 2 | 1 | **0** | **1** | 2 | 2 | 2 | 2 |
| a | 3 | 2 | **1** | **2** | 3 | 3 | 3 | 3 | 2 | 1 | **1** | **2** | 3 | 3 | 3 |
| p | 4 | 3 | 2 | 2 | **2** | 3 | 4 | 4 | 3 | 2 | 2 | 2 | **2** | 3 | 4 |
| l | 5 | 4 | 3 | 3 | 3 | **2** | 3 | 4 | 4 | 3 | 3 | 3 | 3 | **2** | 3 |
| e | 6 | 5 | 4 | 4 | 4 | 3 | **2** | 3 | 4 | 4 | 3 | 3 | 4 | 3 | **2** |

*Tabuľka 2.2: Dynamická programovacia matica pre vyhľadávanie patternu „`staple`“ v texte „`sample steeple`“ s dvomi chybami. Tučným písmom sú zvýraznené cesty k nájdeným zhodám.*

### 2.3.4 Vety o dynamickej matici

Vzhľadom ku svojej štruktúre a spôsobu výpočtu má dynamická matica viaceré vlastnosti, ktorých objavenie nielenže umožnilo urýchliť niektoré existujúce algoritmy, ale aj vytvoriť mnohé nové, založené práve na ich využití. Pre správne pochopenie ďalšieho textu uvedieme aspoň tri základné z nich (v prípade potreby je možné ďalšie vlastnosti nájsť v [6] alebo [22]):

**Veta o neklesajúcej sekvencii diagonál**
Nech $C_{i,j}$ je ľubovoľná hodnota v dynamickej matici, kde $1 \leq i \leq m, 1 \leq j \leq n$. (2.3)
Potom platí, že $C_{i,j} \in \{C_{i-1,j-1}, C_{i-1,j-1}+1\}$ (prevzatá z [22], dôkaz tamtiež).

**Veta o stĺpcových diferenciách**
Nech $C_{i,j}$ je ľubovoľná hodnota v dynamickej matici, kde $1 \leq i \leq m, 0 \leq j \leq n$. (2.4)
Potom platí, že $C_{i,j} - C_{i-1,j} \in \{-1, 0, 1\}$ (prevzatá z [23], dôkaz tamtiež).

**Veta o riadkových diferenciách**
Nech $C_{i,j}$ je ľubovoľná hodnota v dynamickej matici, kde $0 \leq i \leq m, 1 \leq j \leq n$. (2.5)
Potom platí, že $C_{i,j} - C_{i,j-1} \in \{-1, 0, 1\}$ (prevzatá z [27], odvodenie tamtiež z viet 2.4, 2.5 a vzorca dynamickej matice 2.1)

### 2.3.5 Časová a priestorová zložitosť

Pri analýze časovej a priestorovej zložitosti algoritmov budeme vždy uvádzať horné ohraničenie, tzv. *worst-case*. V prípade označenia inej (napr. očakávanej) zložitosti bude tento fakt explicitne spomenutý v texte.

Časová zložitosť základného algoritmu je očividne $O(m * n)$, keďže algoritmus počíta maticu týchto rozmerov. Priestorová zložitosť je zdanlivo rovnaká. Bližší pohľad na algoritmus však ukazuje, že k výpočtu stačí mať uložený vždy len jeden stĺpec matice a každý ďalší počítať priamo do neho. Skutočná priestorová zložitosť algoritmu je teda $O(m)$.

## 2.4 Klasifikácia algoritmov – online a offline prístup

Doba vyhľadávania klasických algoritmov v rozsiahlych textoch môže aj pri použití najlepších techník dosiahnuť vysokých hodnôt—ich efektivita je totiž vždy limitovaná dĺžkou textu, ktorý postupne po znakoch prechádzajú. Z tohoto dôvodu sa algoritmy používané k približnému vyhľadávaniu začali deliť na dve základné skupiny podľa toho, či je pred vyhľadávaním známy pattern alebo text, resp. ktorý z nich je pred vyhľadávaním predspracovaný.

*Online algoritmy*, používané v prípade predom neznámeho textu, si pred samotným vyhľadávaním väčšinou predspracujú pattern, aby bolo jeho následné vyhľadanie čo najrýchlejšie.

Na opačnej strane stoja *offline algoritmy*, v literatúre nazývané tiež *indexovacie*, pretože pred vyhľadávaním vytvoria nad textom určitú dátovú štruktúru (tzv. index), ktorá potom slúži na rýchle vyhľadanie patternu v texte.

Vzhľadom k obmedzenému rozsahu práce nie je možné poskytnúť kompletný prehľad všetkých techník a používaných prístupov v oboch skupinách. V prípade potreby detailnejších informácii je však možné nahliadnuť do podkapitoly 2.4.3, ktorá obsahuje odkazy na ďalšie zdroje mapujúce prístupy a algoritmy pre približné vyhľadávanie.

Ako už bolo uvedené, táto práca sa zameriava na algoritmy určené pre offline vyhľadávanie a jej cieľom je navrhnúť a implementovať vylepšenie práve takéhoto algoritmu. Vybrané offline algoritmy budú detailne prebrané v kapitole 3, vylepšený algoritmus bude predstavený v kapitole 4 a jeho implementácia a otestovanie v porovnaní s pôvodnými algoritmami potom v kapitolách 5 a 6.

### 2.4.1 Online algoritmy

Online algoritmy sú vo všeobecnosti pomalšie ako offline, pretože—či už majú alebo nemajú predspracovaný pattern—musia pri vyhľadávaní prejsť celú dĺžku textu, čím vzniká minimálna časová zložitosť vyhľadávania $\boldsymbol{O(n)}$. Sú väčšinou používané v jednom z troch prípadov:

- text nie je pred vyhľadávaním známy, prípadne sa mení tak rýchlo, že je jeho indexácia neefektívna;
- objem textu je taký, že vytvorenie akéhokoľvek indexu jednoducho nie je možné;
- text ako taký nie je konečný, tj. algoritmus neustále prijíma nové znaky, v ktorých vyhľadáva výskyt patternu.

Posledný menovaný prípad sa týka napríklad routerov alebo firewallov, ktoré vyhľadávajú podozrivé patterny v sieťových tokoch dát. V týchto prípadoch je text prakticky nekonečný, pretože prichádza v rozličných intenzitách takmer bez prestania.

Všetky online algoritmy môžeme ďalej rozdeliť podľa konkrétnej metódy riešenia na viaceré základné skupiny, ich rozbor však necháme na iné zdroje (viď podkapitolu 2.4.3).

**Algoritmy založené na dynamickej matici**

Vôbec najstaršia skupina online algoritmov využíva pre výpočet dynamickú maticu. Vďaka viacerým vlastnostiam dynamickej matice (podkapitola 2.3.4) vzniklo v tejto kategórii azda najviac algoritmov. Medzi najrýchlejšie z nich patria tzv. diagonálne tranzitívne algoritmy (*diagonal transition algorithms*) využívajúce k výpočtu vetu 2.3. Medzi algoritmy založené na dynamickej matici patrí aj pôvodný algoritmus od Sellersa [20], ktorý bol vysvetlený v podkapitole 2.3.

**Konečné automaty**

Algoritmy založené na *deterministických konečných automatoch* (ďalej len DKA) sú zaujímavé tým, že pri vyhľadávaní dosahujú hranicu časovej zložitosti online algoritmov $\boldsymbol{O(n)}$. Ich použiteľnosť je ale limitovaná kvôli vysokému počtu stavov, ktoré sa pri vytváraní automatu musia vygenerovať. Použitím *nedeterministických konečných automatov* (ďalej len NKA) je možné počet stavov značne znížiť, ale za cenu straty minimálnej časovej zložitosti. Ďalším riešením je použitie

tzv. *lazy automatov*, ktoré priamo počas vyhľadávania generujú potrebné stavy, opäť však za cenu nižšej rýchlosti výpočtu.

**Bitový paralelizmus**

Algoritmy z tejto kategórie využívajú fakt, že počítač je schopný pracovať s celým dátovým typom `word` naraz. Jeho veľkosť je závislá od architektúry počítača, v dnešných počítačoch má teda `word` `32` alebo `64` bitov. Akákoľvek bitová operácia (`and`, `or`, `xor`, `shift` a ďalšie) je vykonaná nad každým bitom `word`u paralelne, takže pri vhodnom využití týchto operácií je možné daný algoritmus urýchliť až 32 či 64krát. V praxi sa paralelizácia uplatňuje najmä na NKA a pri použití dynamickej programovacej matice.

**Algoritmy filtrujúce text**

Špecifickou skupinou algoritmov sú filtračné algoritmy, ktoré pracujú v dvoch fázach. V prvej fáze najprv odfiltrujú časti textu, v ktorých sa hľadaný pattern nemôže nachádzať. V druhej fáze je potom použitý akýkoľvek klasický algoritmus, ktorý vyhľadá približné zhody s patternom vo zvyšnom (neodfiltrovanom) texte. Filtračné algoritmy patria medzi najrýchlejšie, pretože využívajú fakt, že je jednoduchšie nájsť časti textu, v ktorých sa pattern nachádzať nemôže, ako vyhľadať v texte jeho približné zhody.

Ich nevýhoda je naopak tá, že k vyhľadaniu približných zhôd potrebujú ďalší algoritmus, ktorý musí zvyšný text prejsť. Okrem toho sú veľmi závislé na pomere chyby $\alpha$—pri vyšších hodnotách sa filtračné algoritmy stávajú neefektívnymi, pretože odfiltrujú príliš malé množstvo textu.

### 2.4.2 Offline algoritmy

Hlavnou výhodou offline algoritmov je fakt, že vďaka indexu nemusia pri hľadaní prechádzať celý text, takže vyhľadávanie máva časovú zložitosť nižšiu ako $O(n)$. Nevýhodou je naopak potreba udržovania celého (často veľmi veľkého) indexu v pamäti a tiež nutnosť (pri prevažnej väčšine algoritmov) vytvoriť nový pri každej zmene textu.

Typickým príkladom použitia týchto algoritmov môže byť napríklad fulltextové vyhľadávanie v databáze kníh. Text databázy sa mení len vtedy, keď do nej pribudnú nové knihy, takže je použitie indexu nad takýmto textom veľmi výhodné.

Rovnako ako online algoritmy, aj indexovacie môžeme podľa metód riešenia rozdeliť na viacero kategórií (kategorizácia prevzatá z [17]). Ich grafické porovnanie zobrazuje obrázok 2.2.

**Generácia okolia (Neighborhood generation)**

Myšlienka tejto metódy je, že vygenerujeme všetky reťazce, ktoré majú od patternu $P$ vzdialenosť maximálne $k$, teda tzv. $k$-okolie ($k$-*neighborhood*) patternu. K nájdeniu približných zhôd s textom nám potom stačí vyhľadať v texte všetky tieto reťazce *presným* vyhľadávaním.

Hoci je táto metóda principiálne veľmi jednoduchá, počet vygenerovaných reťazcov so zväčšujúcimi sa hodnotami $m$ a $k$ rapídne stúpa, čo často limituje jej použitie len na dostatočne malé hodnoty týchto parametrov. Práca sa sústredí práve na túto kategóriu, pričom sa zameriava

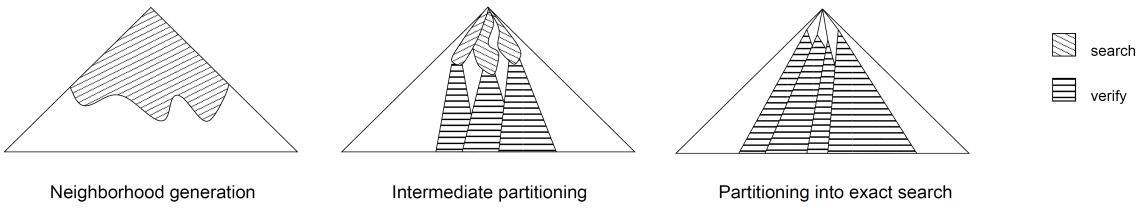
na algoritmy využívajúce k tomuto vyhľadávaniu suffixové stromy. Vylepšený hybridný algoritmus je preto taktiež z tejto kategórie.

**Rozklad na presné vyhľadávanie (Partitioning into exact search)**

Algoritmy z tejto kategórie hľadajú časti patternu v texte *presným* vyhľadávaním. Tento postup je založený na fakte, že aj pri približnom vyhľadávaní sa vždy musia aspoň niektoré časti patternu vyskytnúť v texte bez chýb. Jedná sa teda o akúsi variantu filtrácie, kedy sa po nájdení časti patternu v texte skúmajú okolité znaky, aby sa zistil prípadný výskyt približnej zhody.

**Rozdelenie na stred (Intermediate partitioning)**

Táto metóda je akousi kombináciou oboch vyššie uvedených metód. Tak ako v rozklade na presné vyhľadávanie, aj v tejto vyhľadávame časti patternu v texte. Tieto časti sú však väčšie ako v pôvodnej metóde a môžu sa vyskytnúť s chybami. Preto ich vyhľadávame síce stále len približne, ale s *menším* počtom povolených chýb. To nám potom umožňuje použiť na približné vyhľadávanie metódu generácie okolia.




*Obrázok 2.2: Grafické porovnanie metód offline algoritmov (obrázok prevzatý z [17]).*

### 2.4.3 Ďalšie zdroje

Prehľad algoritmov zakončíme krátkym zoznamom zdrojov, ktoré detailnejšie mapujú taxonómiu algoritmov na približné vyhľadávanie. V prípade potreby bližších informácií o prístupoch alebo konkrétnych algoritmoch je potom možné nahliadnuť do nich.

Veľmi široký a podrobný prehľad online algoritmov sa nachádza v [16]. Prehľad niektorých starších algoritmov aj s implementáciou je v [8]. Podrobný prehľad a nová implementácia online algoritmov založených na konečných automatoch sa nachádza v predchádzajúcej práci autora [21].

Krátke zhrnutie offline algoritmov sa nachádza v [17].

V [15] sú detailne prebrané nielen online a offline algoritmy, ale aj niektoré algoritmy pre špeciálne druhy vyhľadávania (napríklad slovníkové vyhľadávanie alebo paralelné vyhľadávanie viacerých patternov).

# 3 Offline algoritmy pre približné vyhľadávanie

Cieľom tejto kapitoly je prehľadovo pokryť rozličné prístupy k offline vyhľadávaniu predstavením konkrétnych algoritmov z každej skupiny. Nakoľko však navrhovaný vylepšený algoritmus (kapitola 4) spadá pod kategóriu algoritmov generujúcich $k$-okolie patternu, pre obmedzený rozsah semestrálneho projektu vynecháme algoritmy spadajúce pod ostatné kategórie. Tieto budú pridané až do diplomovej práce, ktorá na tento účel poskytuje dostatočný rozsah. Spolu s nižšie uvedenými algoritmami budú tiež implementované pre účely porovnania ich efektívnosti s navrhovaným algoritmom.

## 3.1 Používané dátové štruktúry

Kritickou súčasťou každého offline algoritmu je jednak dátová štruktúra, ktorú na indexovanie textu využíva, a tiež samotný spôsob, akým je nad ňou vyhľadávanie vykonávané. Kým pri online algoritmoch je väčšinovo používanou dátovou štruktúrou jednoduché pole či hashovacia tabuľka, offline algoritmy často využívajú zložitejšie štruktúry, s ktorými musí byť čitateľ zoznámený pred predstavením samotných algoritmov. Tie najdôležitejšie z nich preto v skratke uvedieme spolu s odkazmi na ich formálnejšie definície.

### 3.1.1 Suffixové stromy

Suffixový strom je stromová štruktúra veľmi často využívaná pre indexáciu textu. Umožňuje rýchle nájdenie ľubovoľného suffixu nachádzajúceho sa v texte, pretože pre každý z nich obsahuje strom samostatnú vetvu vedúcu priamo z koreňového uzlu, pričom jednotlivé hrany sú označené vždy znakom, ktorým je možné sa dostať z nadradeného uzlu do synovského. Definujme suffixový strom nasledovne:

> Nech $ST(T) = (Q, \Sigma, \texttt{goto}, \texttt{root}, \texttt{position})$ je *suffixový strom* reprezentujúci všetky suffixy textu $T$, kde:
>
> $Q$ je množina všetkých *uzlov stromu* $s \in Q$;
>
> $\Sigma$ je konečná *abeceda* textu $T$;
>
> $\texttt{goto}(s, t) = s'$ je *tranzitívna funkcia stromu*, ktorá pre ľubovoľný *uzol* $s \in Q$ a znak textu $t \in \Sigma$ vráti buď nový *uzol* $s'$, tzn. urobí *prechod* v rámci stromu na nový uzol $s \xrightarrow{t} s'$, alebo nedefinovanú hodnotu, pokiaľ daný prechod v strome nie je možný;

***root*** je *koreňový uzol* stromu;

*listový uzol* $r$ stromu $T$ je taký uzol, pre ktorý nie je hodnota funkcie `goto` definovaná pre žiadne $t$, tj. $\forall\, t \in \Sigma$: `goto`$(r, t) =$ ***undefined***;

`position`$(s)$ je *pozičná funkcia stromu*, ktorá pre ľubovoľný *listový uzol* $s \in Q$ vráti pozíciu v texte $T$, na ktorej začína reťazec vytvorený konkatenáciou znakov všetkých prechodov od uzlu ***root*** po uzol $s$, a pre ľubovoľný nelistový uzol vráti najnižšiu hodnotu z pozícií všetkých listových uzlov, ktoré sa nachádzajú v jeho podstrome.

Potom pri prechode z *koreňového uzlu* ***root*** stromu $T$ dosiahneme existujúci *listový uzol* $r$ vtedy a len vtedy, ak konkatenácia znakov všetkých prechodov od uzlu ***root*** po $r$ tvorí *platný suffix textu* $T$ na pozícii `position`$(r)$.

Ďalej označme *hĺbkou uzlu* `depth`$(s)$ dĺžku reťazca potrebného k prechodu z uzlu ***root*** do uzlu $s$.

Taktiež nazvime uzol $s$ *synovským uzlom* uzlu $r$ práve vtedy, ak `goto`$(r, t) = s$.

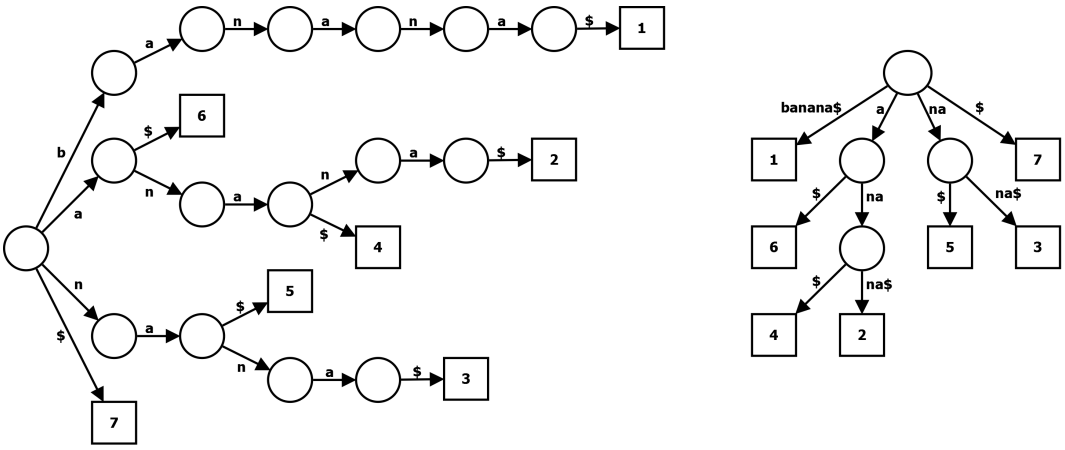
Každý listový uzol $s$ v strome teda predstavuje konkrétny unikátny suffix textu $T$ a obsahuje index jeho začiatku v texte `position`$(s) = T_j$. Z definície taktiež vyplýva, že hĺbka listového uzlu sa vždy rovná dĺžke suffixu.

Pre praktické použitie je suffixový strom vytváraný vždy tak, že každý listový uzol je dosiahnuteľný špeciálnym znakom $\$$, ktorý sa nevyskytuje nikde inde v texte a označuje koniec reťazca. Aby však hĺbka listových uzlov naďalej spĺňala definíciu, tento posledný prechod nezvyšuje hĺbku uzlu a tiež nie je explicitne zahrnutý v algoritmoch, keďže ho vnútorne rieši algoritmus pre vytváranie suffixového stromu.

Komprimovaný suffixový strom (*suffix tree*) sa od základného, nekomprimovaného stromu (*suffix trie*) líši tým, že hrany nemusí mať označené len jedným znakom, ale celým reťazcom, ako ukazuje Obrázok 3.1. Keďže základnou nevýhodou limitujúcou praktické použitie nekomprimovaných stromov je ich pamäťová náročnosť $O(n^2)$, v praxi sa používajú stromy komprimované, v ktorých sú všetky uzly s len jedným synovským uzlom zlúčené s uzlom rodičovským.

Vytvorenie komprimovaného suffixového stromu má časovú aj pamäťovú náročnosť $O(n)$. V práci budeme na ich vytvorenie najčastejšie používať algoritmus od Ukkonena [25][2], keďže súčasťou ním vytvoreného stromu sú aj tzv. suffixové odkazy (*suffix links*), ktoré väčšina offline algoritmov vyžaduje k svojej práci. [5] obsahuje Java applet demonštrujúci tvorbu stromu krok za krokom.

2 Pôvodný článok od Ukkonena obsahuje viaceré chyby a nepresnosti, kvôli ktorým je veľmi problematické algoritmus správne implementovať. [19] obsahuje veľmi dobrý návod k jeho správnemu pochopeniu.




*Obrázok 3.1: Porovnanie nekomprimovaného (vľavo) a komprimovaného (vpravo) suffixového stromu vytvoreného nad textom „banana“.*

### 3.1.2 Suffixové polia

Suffixové pole (*suffix array*) je alternatívou k suffixovým stromom, ktoré má menšiu pamäťovú náročnosťou a o $O(\log n)$ pomalšie vyhľadávanie. V poli samotnom sú uložené len indexy jednotlivých lexikograficky zoradených suffixov textu, tak ako to zobrazuje Tabuľka 3.1. V kombinácii s tzv. *LCP poľom* má suffixové pole rovnakú vyjadrovaciu silu ako suffixový strom; viď [26] pre ich porovnanie a postup vytvorenia.

| Suffix | Index | Lexikografické poradie |
|---|---|---|
| banana$ | 1 | 5 |
| anana$ | 2 | 4 |
| nana$ | 3 | 7 |
| ana$ | 4 | 3 |
| na$ | 5 | 6 |
| a$ | 6 | 2 |
| $ | 7 | 1 |

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A[i] | 7 | 6 | 4 | 2 | 1 | 5 | 3 |

*Tabuľka 3.1: Tabuľka suffixov pre text „banana“ (vľavo) a zodpovedajúce LCP pole (vpravo)*

### 3.1.3 Suffixový automat

Suffixový automat (*suffix automaton*) je konečný automat postavený nad suffixovým stromom, zložený z jednotlivých stavov (uzlov stromu) a prechodovej funkcie. Jeho definícia je nasledovná:

Nech $SA(T) = (Q, \Sigma, \texttt{goto}, \texttt{root}, f)$ je *suffixový automat* prijímajúci všetky suffixy textu $T$, kde:

$Q$ je množina všetkých *stavov automatu* $s \in Q$;

$\Sigma$ je konečná *abeceda* textu $T$;

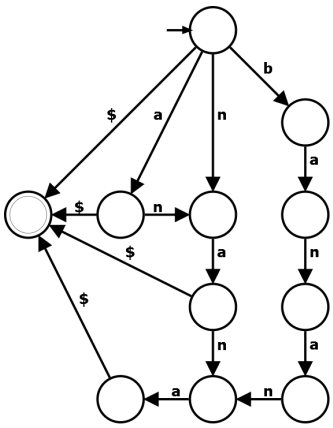
$\texttt{goto}(s, t) = s'$ je *tranzitívna funkcia automatu*, ktorá pre ľubovoľný *stav* $s \in Q$ a znak textu $t \in \Sigma$ vráti nový *stav* $S'$, tzn. urobí *prechod automatu* $S \xrightarrow{t} S'$;

*root* je *začiatočný stav automatu*, zhodný s koreňovým uzlom pôvodného suffixového stromu;

a $f$ je *konečný stav automatu*., pričom do konečného stavu vedú len prechody cez špeciálny znak *$*.

Potom pri prechode textom $T$ dosiahne automat *konečný stav* $f$ po prečítaní znaku $T_j$ vtedy a len vtedy, ak konkatenácia znakov všetkých prechodov od stavu *root* po $f$ *tvorí platný suffix textu* $T$.

Postup konštrukcie minimálneho suffixového automatu zo suffixového stromu je popísaný napr. v [3], [18] obsahuje Java applet umožňujúci po krokoch sledovať priebeh konštrukcie automatu (v ruštine). Obrázok 3.2 zobrazuje suffixový automat vytvorený nad textom „banana“.




*Obrázok 3.2: Suffixový automat vytvorený nad textom „banana“.*

Hoci to z definícií predchádzajúcich dátových štruktúr priamo nevyplýva, všetky tri majú rovnakú vyjadrovaciu silu a preto sú z pohľadu použitia v offline algoritmoch ekvivalentné a je možné medzi nimi robiť bezstratový prevod.

## 3.2 Vyhľadávanie v suffixových stromoch

Predtým, ako prejdeme na konkrétne offline algoritmy k približnému vyhľadávaniu, je pre ujasnenie najprv potrebné ukázať, ako v týchto štruktúrach prebieha presné vyhľadávanie. Hoci to z definícií dátových štruktúr v predchádzajúcej podkapitole priamo nevyplýva, všetky tri majú rovnakú vyjadrovaciu silu a preto sú z pohľadu použitia v offline algoritmoch ekvivalentné a je možné medzi nimi robiť bezstratový prevod. Presné vyhľadávanie preto objasníme len na nekomprimovaných suffixových stromoch, keďže väčšina offline algoritmov v notácii pracuje práve s nimi.

Ako už bolo spomenuté v definícii suffixového stromu (podkapitola 3.1.1), každý listový uzol predstavuje unikátny suffix textu $T$, nad ktorým je strom vytvorený. Presné vyhľadávanie ľubovoľného suffixu $S$ teda prebieha postupným prechodom stromom z koreňového uzlu do synovských, pričom sa pre prechod z aktuálneho uzlu $r$ používa vždy znak $P_{i+1}$, kde index $i$ sa rovná hĺbke uzlu $\texttt{depth}(s)$.

Vyhľadávanie ľubovoľného podreťazca textu je implicitne umožnené štruktúrou suffixového stromu, pretože každý podreťazec musí byť zároveň prefixom nejakého suffixu textu. Pokiaľ teda pri vyhľadávaní patternu $P$ existuje v strome kontinuálna cesta $\texttt{goto}$-prechodov od koreňového uzlu pre každý znak patternu $P_1 \ldots P_m$, je vyhľadávaný reťazec prefixom aspoň jedného suffixu a vyskytuje sa v texte.

Rozšírme pre účely ďalšieho textu definíciu tranzitívnej prechodovej funkcie stromu na celé reťazce takto:

> $\texttt{goto}(s, P) = s'$ je *tranzitívna funkcia stromu*, ktorá pre ľubovoľný *uzol* $s \in Q$ a reťazec znakov textu $P \in \Sigma^*$ vráti buď nový *uzol* $s'$, tzn. urobí *sériu prechodov* $\texttt{goto}(s_i, P_i)$ pre $i = 1 \ldots m$, kde $s_1 = s$ a $s_i$ pre $i > 1$ je výstupom predchádzajúceho prechodu, alebo nedefinovanú hodnotu, pokiaľ v ľubovoľnom kroku daný prechod v strome nie je možný.

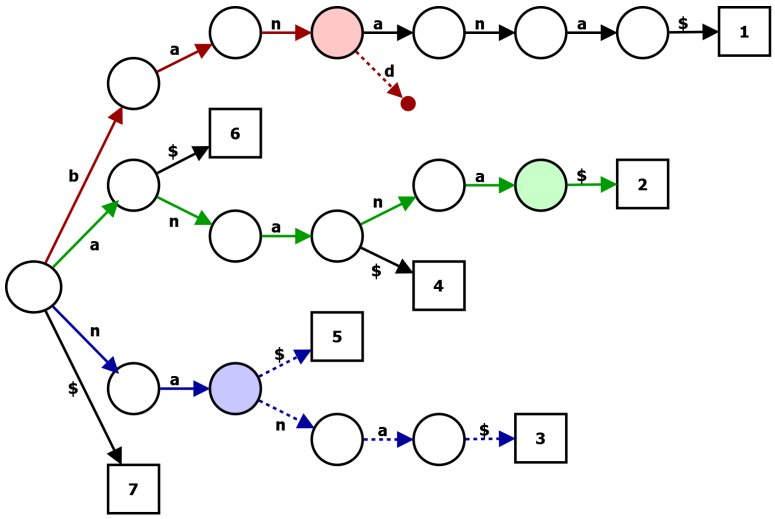
Algoritmus pre presné vyhľadávanie textu v suffixovom strome ukazuje Pseudokód 1. Najprv je v cykle (3-8) vykonaný prechod pre každý znak patternu. Pokiaľ sa všetky prechody úspešne uskutočnili, prejde algoritmus k nahlasovaniu zhody (9-13). Tu sa situácia odlišuje podľa toho, či je posledný uzol listový alebo nie. Ak ide o listový uzol, znamená to, že bol vyhľadávaný unikátny suffix textu (9-10). V opačnom prípade bol vyhľadávaný pattern len vlastným prefixom jedného alebo viacerých suffixov. Z tohoto dôvodu je potrebné prejsť celý podstrom aktuálneho uzlu a nahlásiť zhodu pre každý listový uzol, ktorý sa v podstrome nachádza (11-13).

**Pseudokód 1 — Presné vyhľadávanie v suffixovom strome**

```
1  s = root                              # začiatok vyhľadávania v koreňovom uzle
2  dpt = depth(s)                        # nastavenie počiatočnej hĺbky na nulu
3  while dpt ≤ m:                        # kým je hĺbka menšia ako dĺžka patternu…
4    dpt = dpt + 1
5    s = goto(s, P[dpt])                 # prechod na nový uzol
6    if s == undefined:                  # kontrola existencie prechodu
```

```
7       print "Pattern sa nenašiel"
8       exit
9   if isLeaf(s):                          # ak je uzol listový, nahlás zhodu
10    print "Zhoda na pozícii " + position(s) + "…" + (position(s) + depth(s))
11  else:                                  # inak nahlás zhodu
12    for s2 in subtreeLeafs(s):           #   so všetkými listovými uzlami podstromu
13      print "Zhoda na pozícii " + position(s) + "…" + (position(s) + depth(s))
```

Modelovú situáciu prehľadne znázorňuje obrázok 3.3. Vyhľadávaný podreťazec „na“ sa v texte „banana“ vyskytuje na pozíciách 3 a 5, pričom k zhode dôjde na uzle s hĺbkou 2.




*Obrázok 3.3: Znázornenie neúspešného vyhľadávania patternu „band“ (červená) a úspešného vyhľadávania suffixu „anana“ (zelená) a podreťazca „na“ s dvoma výskytmi v texte (modrá).*

## 3.3 Základný Suffix–tree algoritmus

Základný suffix–tree algoritmus predstavuje priamu aplikáciu výpočtu dynamickej programovacej matice na štruktúru suffixového stromu. Tento jednoduchý algoritmus nebýva v praxi často používaný, pretože má hornú časovú zložitosť vyhľadávania horšiu ako základný algoritmus pre výpočet pomocou dynamickej matice (viď podkapitolu 2.3.5); navyše je potrebné indexovať text do suffixového stromu. Tarhio a Ukkonen ho len okrajovo spomenuli vo svojej práci [9], v ktorej predstavili prvý efektívny offline algoritmus využívajúci suffixový strom. Vysvetlenie jeho princípu nám však poslúži ako referenčný bod pre objasnenie ďalších algoritmov, ktoré z neho vychádzajú.

Algoritmus si pri indexácii textu vytvára suffixový automat, ktorý následne pri vyhľadávaní patternu prechádza, pričom v každom uzle stromu počíta jeden stĺpec dynamickej programovacej matice. Dôležitým rozdielom oproti základnému Sellersovmu algoritmu (podkapitola 2.3) je spôsob, akým je zabezpečené, že každá pozícia v texte môže byť potenciálnym začiatkom zhody s patternom. Kým v základnom algoritme bolo potrebné nultú hodnotu každého stĺpca dynamickej

matice inicializovať na nulu ($C_{0,j} = 0$ pre $0 \leq j \leq n$; viď podkapitolu 2.3.1), v suffixovom strome už je táto funkčnosť obsiahnutá priamo v jeho štruktúre existenciou cesty z koreňového uzlu pre každý suffix textu. Pre výpočet editačných vzdialeností v rámci offline algoritmov preto budeme používať *editačnú hodnotu* $D_{i,j}$ určenú vzorcom 3.1:

$$\begin{aligned} D_{i,0} &= i && \text{pre } 0 \leq i \leq m \\ D_{0,j} &= j && \text{pre } 0 \leq j \leq n \\ D_{i,j} &= \begin{array}{l} \textit{if } (P_i = T_j) \textit{ then } D_{i-1,j-1} \\ \textit{else } 1 + \min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) \end{array} && \text{pre } 1 \leq i \leq m, 1 \leq j \leq n \end{aligned} \tag{3.1}$$

Po tejto úprave teda poslednou otázkou ostáva, ako aplikovať princíp približného vyhľadávania na prechod suffixovým stromom. Pred uvedením samotného algoritmu si zaveďme ešte jedno značenie:

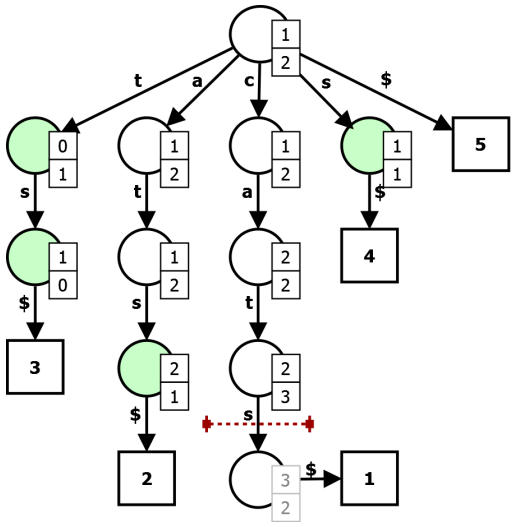
> Nech $s$ je ľubovoľný uzol suffixového strom $ST(T)$ nad textom $T$ a $w \in \Sigma^*$ podreťazec textu $T$ taký, že $\texttt{goto}(root, w) = s$.
>
> Potom označme ako $\texttt{dcol}(s)$ *stĺpec editačných hodnôt* $D_{1\ldots m,j}$, kde index $j$ je pozícia podreťazca $w$ v texte $T$.

Vzhľadom k tomu, že potrebujeme brať do úvahy editačné operácie, nemôžeme pattern vyhľadávať len prechodom jednou cestou tak, ako pri presnom vyhľadávaní (podkapitola 3.2). Algoritmus preto musí z každého uzlu vyskúšať všetky existujúce prechody, overiť, či je posledná editačná hodnota $D_m$ menšia alebo rovná povolenej editačnej vzdialenosti $k$ a v kladnom prípade nahlásiť približnú zhodu. Vyhľadávanie takto pokračuje až do maximálnej hĺbky uzlov $m+k$, pretože za touto hranicou nemôže dôjsť k približnej zhode.[3] Modelovú situáciu pri vyhľadávaní patternu „`ts`“ v texte „`cats`“ s editačnou vzdialenosťou `1` znázorňuje obrázok 3.4. Prechod vetvou stromu zodpovedajúcou suffixu „`cats`“ sa zastaví na uzle s hĺbkou `3`, pretože za touto hĺbkou budú pre každý uzol $s$ všetky hodnoty v stĺpci editačných hodnôt $\texttt{dcol}(s)$ vždy vyššie ako $k$.

### 3.3.1 Kanonický výstup približných zhôd

Obrázok 3.4 poukazuje aj na ďalší fakt, ktorý treba pri vyhľadávaní vziať do úvahy: rovnaká pozícia zhody môže byť v strome nájdená viackrát s odlišnou editačnou vzdialenosťou. Ako vidíme na zeleno zvýraznených uzloch, ku zhode patternu s podreťazcom textu došlo na pozícii 3 pre podreťazec „`t`“ a `4` pre podreťazce „`ts`“, „`ats`“ a „`s`“. Definujme preto pre takéto situácie korektný výstup približných zhôd všetkých algoritmov preberaných v tejto práci nasledovne:

3 Tento záver vychádza z jednoduchého pozorovania, že ku $k$-približnej zhode nemôže dôjsť medzi dvomi reťazcami, ktorých rozdiel dĺžok je väčší ako $k$ znakov, pretože každá inzercia (resp. delécia) znaku zvyšuje ich vzájomnú editačnú vzdialenosť o `1`.




*Obrázok 3.4: Približné vyhľadávanie patternu „`ts`“ v texte „`cats`“ s povolenou editačnou vzdialenosťou `1`. Zelenou sú zvýraznené uzly, na ktorých došlo ku zhode s podreťazcom textu. Vedľa jednotlivých uzlov sú znázornené zodpovedajúce stlpce editačných vzdialeností.*

Nech $\boldsymbol{M}$ je množina všetkých približných zhôd patternu $\boldsymbol{P}$ v texte $\boldsymbol{T}$ s maximálnou editačnou vzdialenosťou $\boldsymbol{k}$.

Nech $\texttt{dis}(\boldsymbol{m})$ je funkcia, ktorá pre ľubovoľné $\boldsymbol{m} \in \boldsymbol{M}$ vracia jeho editačnú vzdialenosť a $\texttt{pos}(\boldsymbol{m})$ funkcia, ktorá vracia pozíciu konca zhody $\boldsymbol{m}$ v texte $\boldsymbol{T}$.

Potom *kanonický výstup približných zhôd* je podľa funkcie $\texttt{pos}()$ vzostupne usporiadaná množina $\boldsymbol{M}_c \subseteq \boldsymbol{M}$ taká, že $\forall\, \boldsymbol{m} \in \boldsymbol{M}:\ \forall\, \boldsymbol{x} \in \boldsymbol{M}, \boldsymbol{x} \neq \boldsymbol{m}:\ \texttt{pos}(\boldsymbol{x}) \neq \texttt{pos}(\boldsymbol{m}) \vee \texttt{dis}(\boldsymbol{m}) < \texttt{dis}(\boldsymbol{x}) \vee \texttt{depth}(\boldsymbol{m}) < \texttt{depth}(\boldsymbol{x})$.

Kanonický výstup reflektuje fakt, že v prípade viacerých možných sekvencií editačných operácií chceme vyberať vždy takú, ktorá povedie k najmenšej editačnej vzdialenosti, a v prípade viacerých zhôd s rovnakou vzdialenosťou vyberáme tú s najmenším možným podreťazcom textu. Pokiaľ nebude explicitne uvedené inak, budeme ďalej v tejto práci pri výstupe každého algoritmu uvažovať takto definovaný kanonický výstup.[4]

### 3.3.2 Vyhľadávanie

Po vytvorení suffixového stromu prebieha vyhľadávanie patternu tak, ako to ukazuje Pseudokód 2. Algoritmus najprv nastaví potrebné hodnoty pre koreňový uzol a následne

4 Pre účely prehľadnosti toto filtrovanie zhôd nebudeme zahŕňať do pseudokódov, no v reálnej implementácii algoritmov je zahrnuté.

na riadkoch 3-17 „hrubou silou“ prechádza všetky nepreskúmané uzly stromu, vyhodnocuje ich editačné hodnoty (9-14) a hlási nájdené zhody (16-17). Následne nový uzol zaradí do nepreskúmaných (18-19), pokiaľ nebola dosiahnutá maximálna hĺbka.

**Pseudokód 2 Základný Suffix-Tree algoritmus: vyhľadávanie**

```
1  new = {root}                              # množina nepreskúmaných uzlov
2  dcol(root) = [0, 1, …, m]                 # nastav editačné hodnoty
3  while not new.empty():                    # kým existujú nepreskúmané uzly…
4    s = new.pop()                           # vyber uzol
5    dOld = dcol(s)
6    foreach t where (s, t) in goto:         # pre každý prechod z uzlu…
7      s2 = goto(s, t)                       # prejdi na nový uzol
8      dNew = [ dOld[0]+1 ]                  # vypočítaj nové hodnoty dcol (nultá
9      for i = 1 to m:                       #   hodnota sa zvyšuje podľa vzorca 3.1)
10       if P[i] == t:
11         dNew[i] = dOld[i-1]
12       else:
13         minValue = min( dOld[i-1], dOld[i], dNew[i-1] )
14         dNew[i] = 1 + minValue
15     dcol(s2) = dNew                       #   a ulož ich
16     if dNew[m] ≤ k:                       # ak sa našla zhoda, nahlás jej pozíciu
17       print "Zhoda na pozícii " + pos(s2) + ", vzdialenosť " + dNew[m]
18     if depth(s2) < m + k:                 # ak sme nedosiahli maximálnu hĺbku,
19       new.add(s2)                         #   pridaj nový uzol do nepreskúmaných
```

### 3.3.3 Optimalizácia: ukončenie výpočtu nevalídnych stĺpcov

V základnom algoritme je nový uzol pridaný do nepreskúmaných vtedy, ak je jeho hĺbka menšia ako $m+k$. Korektnosť takéhoto výpočtu vychádza z upraveného vzorca (3.1) pre počítanie stĺpcov editačných hodnôt v suffixovom strome.

Bližšie skúmanie výpočtu nových stĺpcov však ukazuje, že výpočet je možné ukončiť už vtedy, keď sú všetky hodnoty aktuálneho uzlu väčšie ako $k$. Žiadny prechod z takéhoto uzlu totiž nemôže viesť do uzlu, v ktorom by ktorákoľvek hodnota nového dynamického stĺpca bola menšia alebo rovná $k$, keďže jednotlivé hodnoty môžu oproti predchádzajúcemu stĺpcu len stúpať (pri editačnej operácii) alebo ostať konštantné (pri zhode znaku patternu s textom). Pre úpravu algoritmu potrebujeme nasledovné notácie:

Označme *valídnou hodnotou* každú položku $D_i \leq k$ v stĺpci editačných hodnôt $\texttt{dcol}(s)$ pre uzol $s$.

Ďalej nech funkcia $\texttt{lastValid}(s)$ pre ľubovoľný uzol $s$ vracia index poslednej valídnej položky v stĺpci $\texttt{dcol}(s)$.

Potom *valídna časť stĺpca editačných hodnôt* uzlu $s$ je pole $D_{1\ldots i}$, kde $i = \texttt{lastValid}(s)$.

Následne môžeme algoritmus upraviť tak, ako to ukazuje Pseudokód 3. Algoritmus na začiatku nastaví poslednú valídnu hodnotu koreňového uzlu (2). Pri výpočte nových stĺpcov počíta len toľko hodnôt, koľko je reálne potrebné vzhľadom k indexu poslednej valídnej hodnoty starého stĺpca (8), a priebežne aktualizuje poslednú valídnu hodnotu stĺpca nového (10-11). Detekcia zhody prebieha tiež pomocou poslednej valídnej hodnoty (13) a nový uzol je zaradený do nepreskúmaných len vtedy, ak je aspoň jedna hodnota jeho stĺpca valídna (15-16).

**Pseudokód 3 Základný Suffix-Tree algoritmus: optimalizované vyhľadávanie**

```
1  ...                                    # inicializácia
2  lastValid(root) = k                    # posledná valídna hodnotu koreňového uzlu
3  while not new.empty():                 # kým existujú nepreskúmané uzly…
4    ...
5    stopAt = min( m, lastValid(s) + 1 )  # rozsah počítaných hodnôt cyklu
6    foreach t where (s, t) in goto:      # pre každý prechod z uzlu…
7      ...
8      for i = 1 to stopAt:               # optimalizovaný výpočet dcol
9        ...
10       if dNew[i] ≤ k:                  # ak je vypočítaná hodnota valídna,
11         lastValid(s2) = i              #   aktualizuj poslednú valídnu hodnotu uzlu
12     dcol(s2) = dNew                    # ulož nový stĺpec
13     if lastValid(s2) == m:             # ak sa našla zhoda, nahlás jej pozíciu
14       ...
15     if lastValid(s2) > 0               # ak je aspoň jedna hodnota stĺpca valídna,
16       new.add(s2)                      #   pridaj nový uzol do nepreskúmaných
```

### 3.3.4 Optimalizácia: zavedenie výpočtu dĺžky zhody

Algoritmus pre presné vyhľadávanie (podkapitola 3.2) dokázal určiť nielen pozíciu zhody, ale aj jej dĺžku, tj. dĺžku podreťazca v texte, v rámci ktorého došlo k približnej zhode, priamo z hĺbky uzlu, na ktorom došlo k zhode. Pri približnom vyhľadávaní toto nie je možné, pretože delécia znaku z patternu zmenšuje dĺžku podreťazca textu, na ktorom dochádza k zhode s patternom.

Dĺžku podreťazca textu, na ktorom dochádza k približnej zhode, je teda potrebné udržovať pri výpočte stĺpcov editačných vzdialeností. Zavedieme si preto k tomuto účelu nasledovnú definíciu:

> *Dĺžka podreťazca textu* $\boldsymbol{L_{i,j}}$ pre $1 \le i \le m$, $1 \le j \le n$ je dĺžka najkratšieho suffixu $w$ textu $T_{1\ldots j}$, pre ktorý platí, že $\mathtt{d}(P_{1\ldots i}, T_{j-|w|+1\ldots j}) = D_{i,j}$.
>
> Ďalej nech $s$ je ľubovoľný uzol suffixového strom $ST(T)$ nad textom $T$ a $w \in \Sigma^*$ podreťazec textu $T$ taký, že $\mathtt{goto}(root, w) = s$.
>
> Potom označme ako $\mathtt{lcol}(s)$ *stĺpec dĺžok podreťazcov textu* $L_{1\ldots m,j}$, kde index $j$ je pozícia konca podreťazca $w$ v texte $T$.

Hodnota $L_{i,j}$ teda označuje najkratší možný podreťazec textu končiaci na pozícii $T_j$, ktorého editačná vzdialenosť od časti patternu $P_{1\ldots i}$ je $D_{i,j}$. Jej hodnoty je možné určiť priamo pri výpočte

editačných hodnôt podľa vzorca 3.2. Vidíme, že keďže nám ide o dĺžku *najkratšieho* možného suffixu časti textu $T_{1\ldots j}$, snažíme sa vybrať vždy najprv hornú hodnotu, potom diagonálnu a až ako poslednú možnosť hodnotu zľava (tieto v poradí zodpovedajú delécii z patternu, zhode resp. substitúcii a konečne inzercii do patternu).

$$L_{0,j} = 0$$
$$L_{i,j} = \begin{cases} L_{i-1,j} & \text{ak } D_{i,j} = D_{i-1,j} + 1 \\ L_{i-1,j-1}+1 & \text{ak } D_{i,j} = D_{i-1,j-1} + (\text{if } (P_i = T_j) \text{ then } 0 \text{ else } 1) \\ L_{i,j-1}+1 & \text{inak} \end{cases} \quad (3.2)$$

Výsledný algoritmus (zahŕňajúci aj optimalizáciu z predchádzajúcej podkapitoly) zobrazuje Pseudokód 4. Okrem počiatočného nastavenia stĺpca `lcol` pre koreňový uzol (3) je zaujímavá najmä časť s výpočtom nových stĺpcov `dcol` a `lcol` (13-21). Vzhľadom ku spôsobu výpočtu hodnôt $L_{i,j}$ (vzorec 3.2) musí algoritmus skúšať jednotlivé editačné operácie podľa ich priority, inak by dĺžky nájdenej zhody neboli vypočítané správne. Pri nájdení zhody sa dĺžka jej odpovedajúcemu podreťazcu v texte $T$ určí z poslednej hodnoty v stĺpci `lcol` (28).

**Pseudokód 4 — Základný Suffix-Tree algoritmus: vyhľadávanie**

```
1  new = {root}                              # množina nepreskúmaných uzlov
2  dcol(root) = [0, 1, …, m]                 # nastav editačné hodnoty
3  lcol(root) = [0, …, 0]                    #   a dĺžky najkratších suffixov
4  while not new.empty():                    # kým existujú nepreskúmané uzly…
5    s = new.pop()                           # vyber uzol
6    dOld = dcol(s)
7    lOld = lcol(s)
8    stopAt = min( m, lastValid(s) + 1 )     # rozsah počítaných hodnôt cyklu
9    foreach t where (s, t) in goto:         # pre každý prechod z uzlu…
10     s2 = goto(s, t)                       # prejdi na nový uzol
11     dNew = [ dOld[0]+1 ]                  # nastav nultú hodnotu dcol podľa vzorca 3.1
12     lNew = [ 0 ]                          #   a lcol podľa vzorca 3.2
13     for i = 1 to stopAt:                  # optimalizovaný výpočet nových stĺpcov
14       dNew[i] = dNew[i-1] + 1             # delécia z patternu
15       lNew[i] = lNew[i-1]
16       if dOld[i-1] + (P[i] != t) < dNew[i]: # zhoda alebo substitúcia
17         dNew[i] = dOld[i-1] + (P[i] != t)
18         lNew[i] = lOld[i-1] + 1
19       if dOld[i-1] + 1 < dNew[i]:         # inzercia do patternu
20         dNew[i] = dOld[i] + 1
21         lNew[i] = lOld[i] + 1
22       if dNew[i] ≤ k:                     # ak je vypočítaná hodnota valídna,
23         lastValid(s2) = i                 #   aktualizuj poslednú valídnu hodnotu uzlu
24     dcol(s2) = dNew                       # uloženie nových stĺpcov
25     lcol(s2) = lNew
26     if lastValid(s2) == m:                # ak sa našla zhoda, nahlás jej pozíciu
27       end = lNew[m]                       # vypočítaj koniec zhody
28       print "Zhoda na pozícii " + pos(s2) + "…" + end + ", vzdialenosť " + dNew[m]
29     if lastValid(s2) > 0:                 # ak je aspoň jedna hodnota stĺpca valídna,
```

```
30        new.add(s2)                    #   pridaj nový uzol do nepreskúmaných
```

### 3.3.5 Časová a priestorová zložitosť

**Časová zložitosť**

Časová zložitosť indexácie textu do suffixového stromu je pri použití vhodného algoritmu ([25]) $O(n)$. Pomocné informácie potrebné pri vyhľadávaní (hĺbka uzlu, pozícia zodpovedajúceho suffixu v texte, …) je možné jednoducho uložiť ku každému uzlu stromu v konštantnom čase $O(1)$, preto zložitosť nezvyšujú.

Pôvodný algoritmus pri vyhľadávaní prechádza všetkými uzlami stromu až do hĺbky $m+k$ a v každom počíta stĺpec editačných hodnôt dĺžky $m$. Keďže z každého uzlu môže viesť maximálne $|\Sigma|$ prechodov, je časová zložitosť vyhľadávania pri pôvodnom algoritme $O(|\Sigma| * m^2)$.[5]

Optimalizácia z podkapitoly 3.3.3 zmenšuje hornú časovú zložitosť algoritmu na $O(|\Sigma| * m * k)$ a priemernú zložitosť na $O(|\Sigma| * k^2)$, keďže výpočet zastavuje v momente, keď sú všetky hodnoty stĺpca editačných hodnôt väčšie ako $k$ a každý stĺpec je zároveň počítaný len po poslednú valídnu hodnotu predchádzajúceho stĺpca.

**Priestorová zložitosť**

Určiť priestorovú náročnosť algoritmu je jednoduché. Jedinou štruktúrou závislou od dĺžky textu je suffixový strom používaný ako jeho index. Tento má priestorové nároky $O(n)$. Keďže pomocné informácie potrebné pri vyhľadávaní sú jednoduché skalárne typy, je možné ich jednoducho uložiť ku každému uzlu stromu a zjavne nezvyšujú priestorovú zložitosť algoritmu.

## 3.4 Ukkonenov algoritmus

V roku 1993 publikoval Ukkonen nový, vylepšený algoritmus [24] založený na suffixových stromoch. V článku sa nachádzajú tri jeho varianty, pričom všetky pracujú na rovnakom princípe, len s odlišnou mierou sofistikovanosti. V našej práci sa budeme sústrediť na poslednú verziu, ktorú autor v článku odporučil ako prakticky najpoužiteľnejšiu.

Algoritmus je vylepšením predchádzajúceho offline algoritmu Jokinena a Ukkonena [9]. Tento bol publikovaný v roku 1991 ako vôbec prvý, ktorý úspešne využil suffixový strom k približnému vyhľadávaniu s hornou časovou zložitosťou menšou ako pri základnom Sellersovom algoritme. Stavia na základnom suffix–tree algoritme, no vylepšuje jeho časovú zložitosť na $O(n * m)$ (v najlepšom prípade dosahuje časovú zložitosť $O(m)$), čím spĺňa prirodzenú požiadavku na minimálne takú rýchlosť, akú je možné dosiahnuť základným online algoritmom.

Vylepšenie časovej zložitosti algoritmu Jokinena a Ukkonena je dosiahnuté tak, že sa vyhľadávanie obmedzuje len na tzv. *užitočný podstrom* celého stromu. Ten je určený ako množina

5 Jokinen a Ukkonen uvádzajú v pôvodnej práci zložitosť $O(n * m^2)$, keďže každý znak textu môže byť potenciálne unikátny. Nakoľko sa však zložitosť odvíja od maximálneho počtu prechodov z jedného uzlu, vyjadrenie pomocou veľkosti abecedy presnejšie určuje časovú zložitosť algoritmu.

uzlov, ku ktorým z počiatočného uzlu vedú len užitočné cesty, a ktorá je spojená s koreňovým uzlom najdlhšou z takýchto ciest. *Užitočná cesta* $a_1 \dots a_h$ je pritom taká, že všetky jej podcesty $a_1 \dots a_i$ pre $i \leq h$ obsahujú suffix $a_k \dots a_i$ taký, že editačná vzdialenosť ľubovoľného prefixu patternu $P$ od suffixu je menšia alebo rovná $k$ a zároveň dĺžka tohoto suffixu je väčšia alebo rovná dĺžke najkratšej cesty k uzlu $s = \texttt{goto}(root, a_1 \dots a_i)$, ku ktorému podcesta vedie.

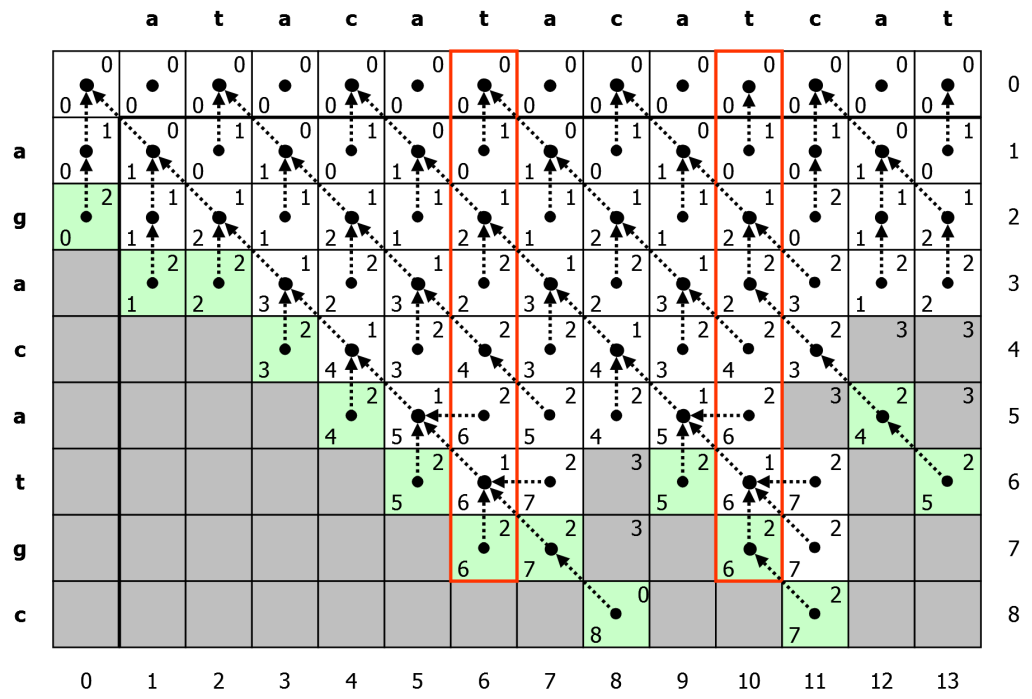
Nakoľko je však Ukkonenov algoritmus z roku 1993 prepracovaním a zjednodušením tohoto algoritmu, nebudeme ho detailne preberať a pre jeho detailný popis odkazujeme čitateľa na pôvodný článok autorov ([9]).

Na rozdiel od predchádzajúceho algoritmu sa Ukkonenov algoritmus snaží optimalizovať časové nároky vyhľadávania tým, že vynecháva opakované počítanie stĺpcov editačných vzdialeností s rovnakými hodnotami, ktoré už raz boli určené. Pre podrobné vysvetlenie tohoto konceptu však potrebujeme nasledovné definície:

> *Realizovateľný* $k$*-približný prefix* $Q_{i,j}$ (*viable* $k$*-approximate prefix*) pre každé $1 \leq i \leq m$, $1 \leq j \leq n$ je taký podreťazec textu $T_{1 \dots j}$, pre ktorý platí, že $|Q_{i,j}| = L_{i,j}$.
>
> Označme ďalej ako *maximálny realizovateľný* $k$*-približný prefix* $Q_j$ také $Q_{i,j}$, kde $i$ označuje poslednú validnú hodnotu $D_{i,j}$, teda $Q_j = Q_{\texttt{lastValid}(s),j}$ pre výpočet daného uzlu $s$ suffixového stromu.
>
> Nech ďalej $\lambda(T_j)$ pre ľubovoľné $1 \leq j \leq n$ je *dĺžka maximálneho realizovateľného prefixu* $Q_j$, teda $\lambda(T_j) = L_{i,j}$ kde $i = \texttt{lastValid}(s)$. Nakoľko každý uzol predstavuje konkrétny podreťazec textu, budeme ďalej používať aj konvenciu $\lambda(s)$ pre označenie dĺžky maximálneho realizovateľného prefixu ľubovoľného uzlu suffixového stromu $s$.

Realizovateľný prefix $Q_{i,j}$ teda vyjadruje podreťazec textu, na ktorom dochádza k približnej zhode s časťou patternu $P_{1 \dots i}$, a $Q_j$ označuje taký podreťazec, ktorý determinuje celú validnú časť stĺpca editačných hodnôt $D_{1 \dots \texttt{lastValid}(s),j}$ uzlu $s$. Vidíme teda, že ide o formalizáciu intuitívne použitého pojmu „podreťazec textu, na ktorom dochádza k približnej zhode“, ktorý bol predtým použitý v podkapitole 3.3.4. Tamtiež definovaný stĺpec dĺžok podreťazcov textu $\texttt{lcol}(s)$ odteraz budeme nazývať aj *stĺpcom dĺžok realizovateľných prefixov* v zhode s definíciou realizovateľného prefixu.

Ako už bolo predtým spomenuté, vychádzajúc z vlastností dynamickej matice vieme s určitosťou povedať, že každá zhoda patternu môže závisieť len na podreťazci textu s dĺžkou maximálne $m + k$. Situáciu prehľadne znázorňuje tabuľka 3.2. V tabuľke sú realizovateľné prefixy znázornené pre každú validnú hodnotu $D_{i,j}$ pomocou editačnej cesty vedúcej do nultého riadku.




*Tabuľka 3.2: Dynamická matica pre text „`atacatacatcat`“ a vyhľadávaný pattern „`agacatgc`.“ Čísla v horných rohoch znázorňujú editačné vzdialenosti $\boldsymbol{D}_{i,j}$, v spodných dĺžky realizovateľných prefixov $\boldsymbol{L}_{i,j}$. Zelenou sú zvýraznené posledné valídne hodnoty a oranžovou príklady ekvivalentných stĺpcov editačných hodnôt $\boldsymbol{D}_{*,6}$ a $\boldsymbol{D}_{*,10}$ (prevzaté z [2] a doplnené).*

Príklad rovnakých realizovateľných prefixov je znázornený oranžovým podfarbením. Všimnime si, že keď majú dva stĺpce editačných vzdialeností rovnaké realizovateľné prefixy v každom svojom riadku, musia byť rovnaké aj všetky ich jednotlivé editačné hodnoty, ktoré majú editačnú vzdialenosť maximálne $\boldsymbol{k}$ (tj. valídna časť stĺpcov). Taktiež je zrejmé (a vyplýva to zo vzorca 3.2), že stĺpec dĺžok realizovateľných prefixov tvorí neklesajúcu sekvenciu hodnôt. Z toho môžeme vyvodiť nasledujúce vety:

**Veta o ekvivalencii stĺpcov editačných hodnôt a realizovateľných prefixov**

Nech $\boldsymbol{D}_{*,i}$ a $\boldsymbol{D}_{*,j}$ sú ľubovoľné dva stĺpce editačných hodnôt a $\boldsymbol{L}_{*,i}$ a $\boldsymbol{L}_{*,j}$ zodpovedajúce stĺpce dĺžok podreťazcov textu.

Potom platí, že páry stĺpcov $(\boldsymbol{D}_{*,i}, \boldsymbol{L}_{*,i})$ a $(\boldsymbol{D}_{*,j}, \boldsymbol{L}_{*,j})$ sú ekvivalentné, tj. $(\boldsymbol{D}_{*,i}, \boldsymbol{L}_{*,i}) \equiv (\boldsymbol{D}_{*,j}, \boldsymbol{L}_{*,j})$, ak valídne časti stĺpcov $\boldsymbol{D}_{*,i}$ a $\boldsymbol{D}_{*,j}$ majú rovnaké hodnoty a korešpondujúce položky stĺpcov $\boldsymbol{L}_{*,i}$ a $\boldsymbol{L}_{*,j}$ sú tiež zhodné (prevzatá z [24], odvodenie tamtiež). (3.3)

**Veta o neklesajúcej sekvencii realizovateľných prefixov**

Nech $L_{*,j}$ je ľubovoľný stĺpec realizovateľných prefixov $L_{0\ldots m,j}$.

Potom platí, že $L_{i,j} \leq L_{i+1,j}$ pre ľubovoľné $0 \leq i < m$ (prevzatá z [24], odvodenie tamtiež). (3.4)

Veta o ekvivalencii stĺpcov je fundamentálnym základom pre Ukkonenov algoritmus: kedykoľvek je počas priebehu algoritmu navštívený uzol v suffixovom strome, ktorého ekvivalentné stĺpce už boli vypočítané predtým, je nový výpočet preskočený. Každý takto vypočítaný stĺpec je preto uložený v suffixovom strome do uzlu, ktorý je možné z počiatočného uzlu dosiahnuť jeho realizovateľným prefixom, aby v budúcnosti nemusel byť znovu prepočítavaný. Optimalizácia spočíva tiež v tom, že keďže na nevalídnych hodnotách stĺpcov nezáleží, takto ekvivalentných stĺpcov bude tým viac, čím je vyhľadávaný pattern dlhší v porovnaní s povolenou editačnou vzdialenosťou.

### 3.4.1 Suffixové odkazy

Pred uvedením presného postupu algoritmu potrebujeme ešte definovať koncept suffixových odkazov:

Nech $s$ je ľubovoľný uzol suffixového strom $ST(T)$ nad textom $T$ a $w \in \Sigma^*$ podreťazec textu $T$ taký, že $\texttt{goto}(root, w) = s$.

Nech $\texttt{path}(s) = w$, tj. funkcia vracia pre ľubovoľný uzol $s$ reťazec znakov potrebných k prechodu z koreňového uzlu do neho: $\texttt{goto}(root, \texttt{path}(s)) = s$.

Potom $\texttt{f}(s) = s'$ je *suffixová funkcia stromu*, ktorá pre ľubovoľný uzol $s \in Q, s \neq root$ dostupný z koreňového uzlu reťazcom $aw$, kde $a \in \Sigma, w \in \Sigma^*$, vráti nový uzol $s'$ taký, že $\texttt{path}(s') = w$, tj. $\texttt{path}(s) = a \mid \texttt{path}(s')$, kde symbol „|“ značí konkatenáciu reťazcov.

Definujme pre ďalšie použitie v práci suffixový strom nasledovne:

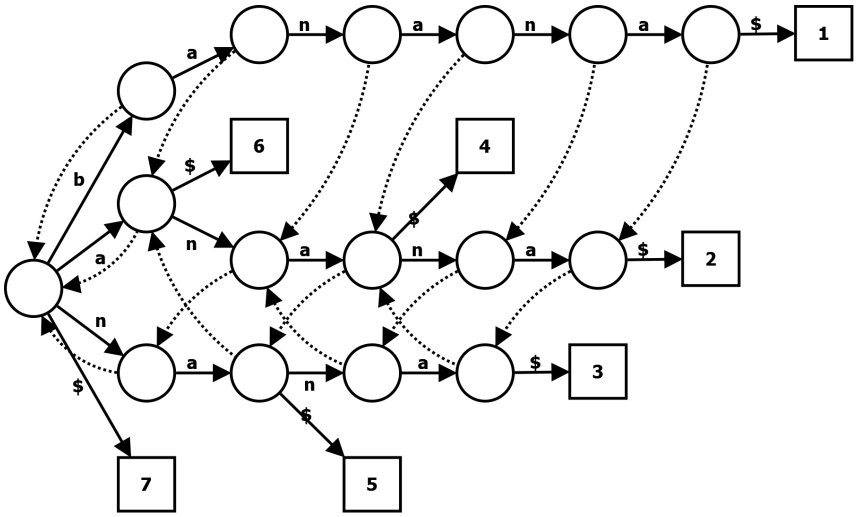
*Suffixový strom obohatený o suffixové odkazy* $ST_f(T) = (ST(T), \texttt{f})$, kde:

$ST(T)$ je ľubovoľný suffixový strom tak, ako bol definovaný v podkapitole 3.1.1;

$\texttt{f}(s) = s'$ je vyššie definovaná *suffixová funkcia stromu* pre každý uzol $s \in Q, s \neq root$.

Obrázok 3.5 zobrazuje ukážku suffixového stromu nad textom „banana“. Vidíme, že z každého nelistového uzlu $s$ vedie suffixová cesta až do koreňového uzlu $root$, pričom sa každým prechodom

pomocou suffixovej funkcie `f` skracuje uzlu zodpovedajúci podreťazec textu o prvý znak tohoto podreťazca.



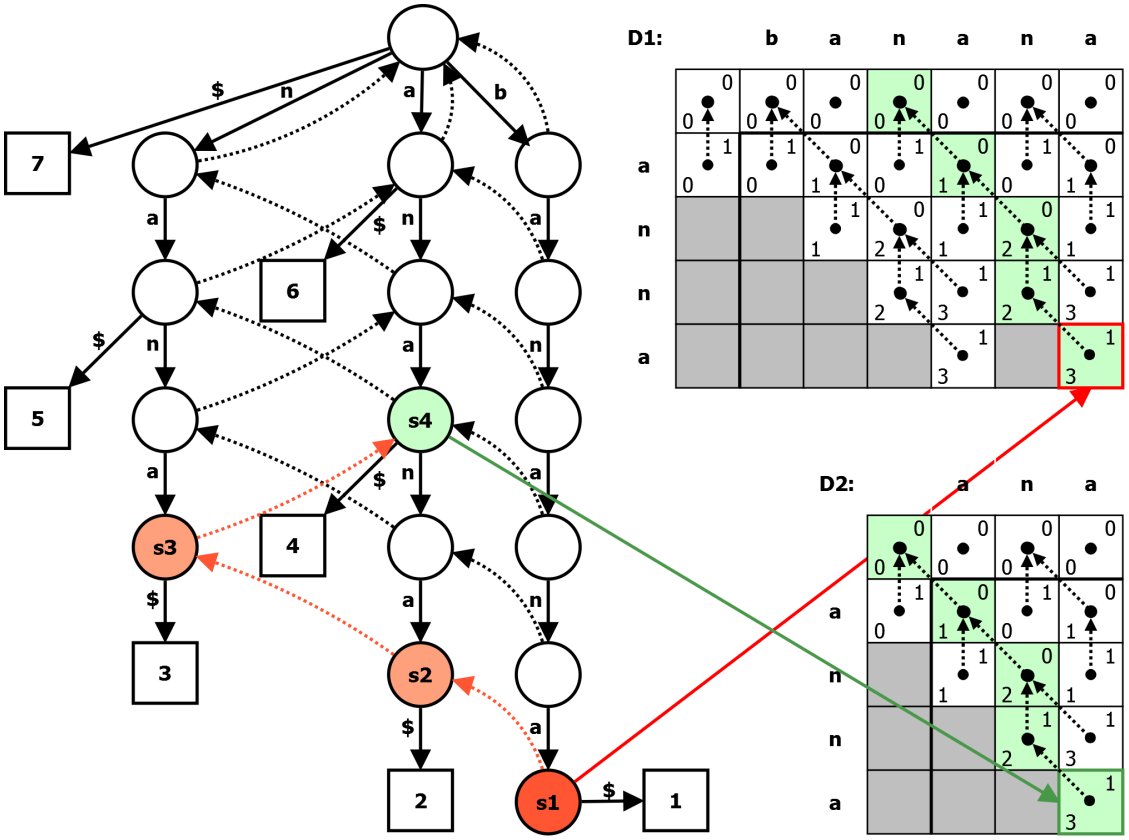

*Obrázok 3.5: Nekomprimovaný suffixový strom s pridanými suffixovými odkazmi vytvorený nad textom „banana“.*

Princíp algoritmu spočíva v tom, že prechádza suffixový strom do hĺbky (depth-first traversal), pri každom ešte neoznačenom uzle vypočíta dvojicu stĺpcov $\texttt{dcol}(s)$ a $\texttt{lcol}(s)$ a označí ho. Následne prechádza suffixovými odkazmi do nadradených uzlov (pričom každý z nich tiež označuje) až dovtedy, kým sa hĺbka uzlu $s'$ nerovná poslednej valídnej hodnote $\texttt{lcol}(s)$, teda $\texttt{depth}(s') = \lambda(T_j)$. K tomuto uzlu uloží vypočítané hodnoty $\texttt{dcol}(s)$ a $\texttt{lcol}(s)$ a nechá ho neoznačený.

Dôvod tohoto postupu je ten, že všetky takto prejdené uzly sú navzájom ekvivalentné podľa vety 3.3—ich valídne časti stĺpcov $\texttt{dcol}(s)$ a $\texttt{lcol}(s)$ sú teda rovnaké a netreba ich počítať opakovane. Praktický dôsledok označenia všetkých uzlov okrem posledného počas prechádzania suffixových odkazov je, že implicitne ukazujú na tento (po suffixovej ceste prvý) uzol s uloženými hodnotami $\texttt{dcol}(s)$ a $\texttt{lcol}(s)$.

Časť priebehu algoritmu pri vyhľadávaní patternu „anna“ v texte „banana“ ukazuje obrázok 3.6. Algoritmus sa nachádza na uzle $s_1$, v ktorom vypočíta hodnoty posledného stĺpca dynamickej programovacej matice $D_1$. Keďže $\lambda(s_1) = 3$, približná zhoda nájdená v poslednom stĺpci dynamickej matice má pozíciu v texte $T_{j-3+1\ldots j}$—závisí teda len na tejto časti textu a znaky „ban“ počítané v prvých troch stĺpcoch dynamickej matice sú irelevantné. Algoritmus teda prejde pomocou suffixových odkazov cez uzly $s_2$ a $s_3$ na uzol $s_4$ zodpovedajúci relevantnému podreťazcu textu „ana“ (poznamenajme, že toto platí práve z dôvodu, že $\texttt{path}(s_4) = ana$). V tomto uzle uloží vypočítané hodnoty $\texttt{dcol}(s_1)$ a $\texttt{lcol}(s_1)$ a pokračuje z neho ďalej všetkými definovanými `goto`-prechodmi.

Korektnosť takéhoto postupu je viditeľná pri porovnaní dynamických matíc $\boldsymbol{D}_1$ a $\boldsymbol{D}_2$: zelenou farbou zvýraznené hodnoty tvoriace editačnú cestu maximálneho realizovateľného prefixu majú rovnaké hodnoty a sú neovplyvnené odstránením prvých troch stĺpcov v tabuľke $\boldsymbol{D}_2$, preto sú posledné stĺpce oboch matíc ekvivalentné.




*Obrázok 3.6: Časť priebehu Ukkonenovho algoritmu pri vyhľadávaní patternu „anna“. Dĺžka maximálneho realizovateľného prefixu v stave $s_1$ je 3, preto algoritmus prejde pomocou suffixových odkazov stavmi $s_2$ a $s_3$ na stav $s_4$ s hĺbkou 3, kam uloží vypočítané hodnoty.*

### 3.4.2 Vyhľadávanie

Kompletný priebeh vyhľadávania Ukkonenovho algoritmu ukazuje Pseudokód 5. V hlavnom cykle (4-42) sú postupne prechádzané všetky nepreskúmané uzly. Vnútorný cyklus prechádza všetky možné `goto`-prechody (8-42) a pokiaľ nájdený uzol ešte nie je označený (eliminovaný), vypočíta jeho stĺpce `dcol`$(s)$ a `lcol`$(s)$ (11-21). Následne sa pomocou suffixových odkazov hľadá uzol, ktorý je v hĺbke zodpovedajúcej dĺžke realizovateľného prefixu (22-25). Pokiaľ nájdený uzol ešte nie je eliminovaný, sú vypočítané stĺpce uložené (27-28), uzol vložený do zoznamu nepreskúmaných (30) a program skontroluje, či neprišlo k približnej zhode (31-33). Posledná časť algoritmu (34-42) označuje na suffixovej ceste až po koreňový uzol všetky také uzly, do ktorých vedú suffixové odkazy len z uzlov, ktoré už sú označené (tj. aktuálny uzol bol posledný neoznačený,

z ktorého takýto odkaz viedol). Tento princíp je v pôvodnom článku označovaný ako *eliminácia uzlu pokrytím*, kedy je podreťazec reprezentovaný daným uzlom suffixom reťazcov len takých uzlov, ktoré sú už eliminované—uzol „pokrytý“ len eliminovanými uzlami sa teda sám stáva eliminovaným.

Dôležité je poznamenať, že aby algoritmus fungoval korektne, musí sa zoznam nepreskúmaných uzlov správať ako zásobník, tj. vloženie nového uzlu na riadku 30 ho musí vložiť na začiatok zoznamu a výber nového uzlu (5) musí vždy odstrániť vrchný (prvý) uzol. Tento spôsob zabezpečí, aby bol strom prechádzaný prechodom do hĺbky, nie do šírky.

**Pseudokód 5** **Ukkonenov algoritmus: vyhľadávanie**

```
1  new = {root}                          # množina nepreskúmaných uzlov
2  dcol(root) = [0, 1, …, m]             # nastav editačné hodnoty
3  lcol(root) = [0, …, 0]                #   a dĺžky najkratších suffixov
4  while not new.empty():                # kým existujú nepreskúmané uzly…
5    s = new.pop()                       # vyber uzol
6    dOld = dcol(s)
7    lOld = lcol(s)
8    foreach t where (s, t) in goto:     # pre každý prechod z uzlu…
9      s2 = goto(s, t)                   # prejdi na nový uzol
10     if eliminated(s2) continue        # ak je uzol už eliminovaný, preskoč výpočet
11     dNew = [ 0 ]                      # nastav nultú hodnotu dcol podľa pôvodného
12     lNew = [ 0 ]                      #   vzorca a lcol podľa vzorca 3.2
13     for i = 1 to m:                   # výpočet nových stĺpcov
14       dNew[i] = dNew[i-1] + 1         # delécia z patternu
15       lNew[i] = lNew[i-1]
16       if dOld[i-1] + (P[i] != t) < dNew[i]: # zhoda alebo substitúcia
17         dNew[i] = dOld[i-1] + (P[i] != t)
18         lNew[i] = lOld[i-1] + 1
19       if dOld[i-1] + 1 < dNew[i]:     # inzercia do patternu
20         dNew[i] = dOld[i] + 1
21         lNew[i] = lOld[i] + 1
22     length = lNew[lastValid(s2)]      # dĺžka maximálneho realizovateľného prefixu
23     while depth(s2) > length and not eliminated(s2):  # prechod suffixovou cestou
24       eliminate(s2)                                   #   a označenie prejdených
25       s2 = f(s2)                                      #   uzlov
26     if depth(s2) == length and not eliminated(s2): # ak je nájdený uzol neoznačený
27       dcol(s2) = dNew                 # uloženie nových stĺpcov
28       lcol(s2) = lNew
29       eliminate(s2)                   # označ uzol
30       new.push(s2)                    #   a pridaj ho do zoznamu nepreskúmaných
31       if dNew[m] ≤ k:                 # ak sa našla zhoda, nahlás jej pozíciu
32         end = pos(s2) + length        # vypočítaj koniec zhody
33         print "Zhoda na pozícii " + pos(s2) + "…" + end + ", vzdialenosť " + dNew[m]
34       s2 = suffixLink(s2)
35       while s2 != root:                       # označuj všetky uzly na suffixovej
36         p = path(s2)                          #   ceste, pokiaľ všetky do nich
37         foreach t where (root, t+p) in goto:  #   vedúce suffixové odkazy vedú
38           co-s = goto(root, t+p)              #   z už označených uzlov
39           if not eliminated(co-s):            # inak skonči
40             break 2
```

```
41          eliminate(s2)
42          s2 = f(s2)
```

Otázne možno ostáva, prečo je pri výpočte nultej hodnoty $\texttt{dcol}(s)$ na riadku 11 používaný pôvodný vzorec (pri ktorom je nastavovaná vždy na `0`) namiesto toho, aby bola inkrementálne zvyšovaná tak, ako sa zvyšuje hĺbka uzlu. Dôvod je ten, že na rozdiel od základného suffixového algoritmu (podkapitola 3.3) sa vypočítaný stĺpec neukladá automaticky do uzlu, pre ktorý je počítaný, ale prechodom suffixovými odkazmi je nájdený taký uzol, pri ktorom je *celý* jemu zodpovedajúci podreťazec súčasťou približnej zhody, nie len jeho suffix. Vidíme to na dynamických programovacích maticiach na obrázku 3.6. Aby tieto vypočítané hodnoty zodpovedali uzlu nájdenému cestou po suffixových odkazoch, je preto nutné umožniť približnú zhodu na každej pozícii textu.

### 3.4.3 Optimalizácia: minimalizácia výpočtov

Na Ukkonenov algoritmus je možné aplikovať základnú optimalizáciu výpočtu na odstránenie zbytočného počítania nevalídnych častí stĺpcov editačných hodnôt a realizovateľných prefixov podobne, ako je to pri základnom algoritme (podkapitola 3.3.3). Kompletnú optimalizáciu zobrazuje Pseudokód 6.

Vidíme, že na rozdiel od základného suffix–tree algoritmu však nie je možné aplikovať optimalizáciu do tej miery, že sa do nepreskúmaných uzlov pridajú len také, ktorých aspoň jedna hodnota je valídna. Toto je zapríčinené vyššie spomenutou vlastnosťou algoritmu, ktorá vyžaduje, aby boli nulté hodnoty každého stĺpca $\texttt{dcol}(s)$ inicializované na `0`.

**Pseudokód 6 — Ukkonenov algoritmus: minimalizácia výpočtov**

```
1   ...                                     # inicializácia
2   lastValid(root) = k                     # posledná valídna hodnotu koreňového uzlu
3   while not new.empty():                  # kým existujú nepreskúmané uzly…
4     ...
5     stopAt = min( m, lastValid(s) + 1 )   # rozsah počítaných hodnôt cyklu
6     foreach t where (s, t) in goto:       # pre každý prechod z uzlu…
7       ...
8       for i = 1 to stopAt:                # optimalizovaný výpočet dcol
9         ...
10        if dNew[i] ≤ k:                   # ak je vypočítaná hodnota valídna,
11          lastValid(s2) = i               #   aktualizuj poslednú valídnu hodnotu uzlu
12    ...
13    if depth(s2) == length and not eliminated(s2): # ak je nájdený uzol neoznačený
14      ...
15      if lastValid(s2) == m:              # ak sa našla zhoda, nahlás jej pozíciu
16        ...
17      ...
```

### 3.4.4 Časová a priestorová zložitosť

**Časová zložitosť**

Rovnako ako pri základnom suffix–tree algoritme je zložitosť indexácie textu do suffixového stromu $O(n)$. Určiť celkový počet uzlov $q$, ktoré budú algoritmom prejdené pri vyhľadávaní, nie je presne možné, keďže závisí od dĺžky patternu $m$, editačnej vzdialenosti $k$, štruktúry textu (počet opakovaní a dĺžka unikátnych podreťazcov) a veľkosti abecedy textu $\Sigma$. Je zrejmé, že $q$ dosahuje hodnôt $\langle 0, n \rangle$, pričom autor uvádza presnejšie ohraničenie $q = O(\min(n, m^{k+1} * |\Sigma|^k))$ (odvodenie v pôvodnom článku [24]). Nakoľko však algoritmus nie je optimálny pri výbere uzlu s realizovateľným prefixom s najmenšou editačnou vzdialenosťou, celkový počet navštívených uzlov je $O(m * q)$, keďže dĺžka realizovateľného prefixu a teda aj počet jeho suffixov, ktoré môže algoritmus vybrať pred navštívením správneho uzlu, je $O(m)$.

Výpočet stĺpcov každého takto navštíveného uzlu trvá $O(m)$ a suffixová cesta z neho do koreňového uzlu je tiež obmedzená na $O(m)$, z čoho vychádza konečná časová zložitosť vyhľadávania algoritmu $O(m^2 * q)$ pre $q = O(\min(n, m^{k+1} * |\Sigma|^k))$.

Optimalizovaný výpočet z podkapitoly 3.4.3 znižuje časovú zložitosť na $O(m * k * q)$, keďže každý stĺpec je počítaný len po poslednú valídnu hodnotu predchádzajúceho stĺpca.

**Priestorová zložitosť**

Priestorová zložitosť algoritmu pre indexáciu textu je $O(n)$ potrebných pre komprimovaný suffixový strom. Priestorová zložitosť algoritmu závisí od faktu, že v každom navštívenom uzle (virtuálne nekomprimovaného) stromu je potrebné uložiť skalárnu informáciu o jeho eliminácii a vypočítané stĺpce veľkosti $O(m)$. Z toho plynie celková priestorová zložitosť $O(m^2 * q)$.

# 4 Vylepšený hybridný algoritmus

V predchádzajúcej kapitole sme uviedli dva základné prístupy k offline približnému vyhľadávaniu pomocou suffixových stromov. V tejto kapitole bude podrobne predstavený vylepšený hybridný algoritmus, ktorý je predmetom tejto diplomovej práce a ktorý pracuje na podobnom princípe ako tieto už uvedené algoritmy, no zlepšuje rýchlosť vyhľadávania pomocou kombinácie online a offline prístupu k približnému vyhľadávaniu nad suffixovým stromom. Algoritmus samotný je za určitých podmienok možné úspešne použiť v kombinácii s inými offline algoritmami založenými na použití suffixových stromov. Tieto podmienky budú podrobnejšie prebrané v samostatnej podkapitole 6.7 časti o implementácii algoritmov.

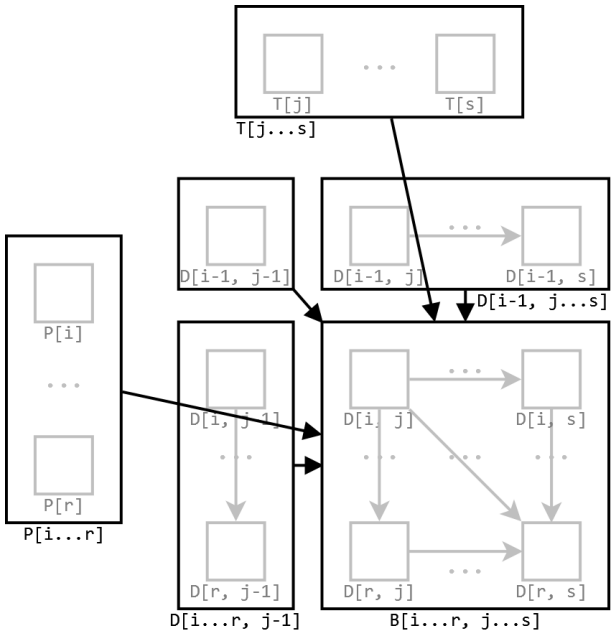
Offline algoritmy používajúce k indexácii textu suffixové stromy (resp. ich ekvivalentné alternatívy, viď podkapitolu 3.1) môžeme podľa spôsobu, akým sa snažia optimalizovať vyhľadávanie, približne rozdeliť do dvoch kategórií:

- **obmedzenie prechádzaného podstromu** v celom suffixovom strome určením rozlične definovaného „užitočného podstromu“ (Základný suffix–tree algoritmus v 3.3, Jokinen–Ukkonen [9]);
- a **obmedzenie zbytočného počítania stĺpcov dynamickej matice** či už vo forme opakovaných výpočtov alebo výpočtov, ktoré nemôžu viesť k približnej zhode (Ukkonenov algoritmus v 3.4, Cobbsov algoritmus [2]).

Obe kategórie pritom vyžadujú relatívne náročné vyhodnocovacie postupy a heuristiky. Optimalizácia vyhľadávania v našom algoritme sa bude zameriavať na urýchlenie samotného výpočtu stĺpcov dynamickej matice. Technika, ktorú budeme využívať, je nazývaná „Four Russians“ a bola prvýkrát predstavená v roku 1970 [1]. Tá je založená na tom, že výpočet dynamickej matice neprebieha po jednotlivých hodnotách $\boldsymbol{D}_{i,j}$, ale po celých blokoch, ktoré sú vytvorené ešte pred samotným vyhľadávaním.

Metóda je jednou z často využívaných optimalizácií v online algoritmoch (napríklad [11] a [27], implementácia v [21]), no podľa našich informácií nebola nikdy použitá na urýchlenie offline vyhľadávania. Bližšie informácie s podrobnou analýzou o nej je možné nájsť v [6] (str. 302-307); [14] obsahuje okrem popisu aj Java applet, umožňujúci po zvolení patternu a textu zobraziť priebeh výpočtu po jednotlivých krokoch.

Počítanie dynamickej matice metódou Four Russians šikovne využíva tú vlastnosť dynamickej matice, že každá hodnota $\boldsymbol{D}_{i,j}$ závisí len od troch predchádzajúcich hodnôt $\boldsymbol{D}_{i,j-1}$, $\boldsymbol{D}_{i-1,j}$ a $\boldsymbol{D}_{i-1,j-1}$ a konkrétnych znakov patternu $\boldsymbol{P}_i$ a textu $\boldsymbol{T}_j$. Počítanie po blokoch je potom generalizáciou tohoto postupu v tom zmysle, že na určenie bloku $\boldsymbol{B}_{i\ldots r,j\ldots s}$ potrebujeme hodnotu $\boldsymbol{D}_{i-1,j-1}$, vektory editačných vzdialeností $\boldsymbol{D}_{i-1,j\ldots s}$ a $\boldsymbol{D}_{i\ldots r,j-1}$ a znakov patternu $\boldsymbol{P}_{i\ldots r}$ a textu $\boldsymbol{T}_{j\ldots s}$, ako to prehľadne znázorňuje obrázok 4.1.




*Obrázok 4.1: Nahradenie výpočtu jednotlivých hodnôt dynamickej matice výpočtom po blokoch.*

## 4.1 Problém blokového výpočtu v suffixových stromoch

Kombinácia blokového výpočtu stĺpcov dynamickej programovacej matice s prechodom suffixového stromu nie je jednoduchá. Medzi základné požiadavky, ktoré máme na úspešnú aplikáciu tohoto prístupu v offline vyhľadávaní patrí:

- nezvýšiť časovú zložitosť prípravnej fázy indexácie textu $O(n)$;
- v prípade, že je potrebná fáza predspracovania patternu, musí mať táto časovú zložitosť nižšiu ako vyhľadávanie samotné;
- vyhľadávanie s použitím blokového výpočtu musí mať časovú zložitosť nižšiu ako vyhľadávanie bez neho; a
- vyhľadávanie musí byť naďalej možné s akýmkoľvek vyhľadávaným patternom $P$ a zvolenou editačnou vzdialenosťou $k$.

Pre výber najvhodnejšieho prístupu boli preskúmané tri články, ktoré sa blokovému výpočtu dynamickej matice venovali: pôvodná práca z roku 1970, podľa autorov ktorej je technika „Four Russians“ pomenovaná [1], algoritmus Masek–Paterson z roku 1980 [11] a článok z roku 1996, ktorú napísali Wu, Manber a Myers [27].

Ako najvhodnejší algoritmus pre kombináciu s prechodom cez suffixový strom bol nakoniec vybraný posledný menovaný. Prvé dva algoritmy sa totiž v pôvodných prácach pri testovaní ukázali pomalšie ako základný Sellersov algoritmus (viď napríklad testy vykonané v [27]), ich prínos bol teda viac teoretický ako praktický. Naviac sa prechod stromovou štruktúrou v kombinácii

s niekoľko-stĺpcovými blokmi ukázal ako obzvlášť komplikovaný, nakoľko v každom uzle sa počíta práve jeden stĺpec matice, blok veľkosti $b$ by teda vyžadoval posun práve a vždy o $b$ `goto`-prechodov. Toto sa stáva problémom vždy, keď z uzlu vedie viac `goto`-prechodov pre rozličné $t \in \Sigma$, čo je v suffixovom strome bežným javom. Vybraný algoritmus namiesto blokov používa len takzvané *regióny*, čo sú vlastne bloky so šírkou `1` a výškou zvolenou podľa potrieb a pamäťových možností prostredia, na ktorom je vyhľadávanie vykonávané. Takýto postup pri použití v online vyhľadávaní ([21], [27]) viedol k urýchleniu počítania stĺpcov dynamickej matice z $O(m)$ na $O(m/r)$ (kde $r$ bola zvolená veľkosť regiónu) a k časovej zložitosti prípravy blokov $O(n)$, čo je pre náš účel ideálne.[6]

V nasledujúcej podkapitole detailne predstavíme vybraný blokový algoritmus tak, ako by bol použitý pri aplikácii na základný Sellersov algoritmus. V podkapitole 4.3 potom uvedieme navrhovaný hybridný algoritmus, pozostávajúci z úpravy blokového algoritmu pre použitie so základným suffix–tree algoritmom z podkapitoly 3.3.

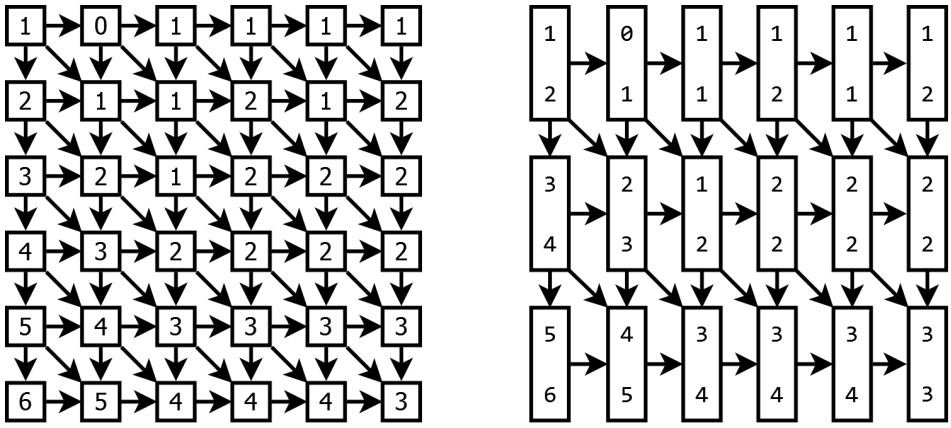
## 4.2 Online algoritmus Wu, Manber a Myers

V roku 1996 publikovali Wu, Manber a Myers prácu [27], v ktorej navrhli nové riešenie veľkého počtu stavov deterministických automatov, ktoré sa často používajú na online približné vyhľadávanie (viď podkapitolu Konečné automaty).

Vo všetkých predchádzajúcich algoritmoch postavených na deterministických automatoch bol za stav považovaný konkrétny stĺpec dynamickej matice. Základný problém tohoto poňatia je, že počet možných stĺpcov dynamickej matice môže byť podľa vety 2.4 až $3^m$. Veľkosť abecedy $\Sigma$ a povolená editačná vzdialenosť $k$ síce znižujú hranicu maximálneho počtu stavov, ale pri ich vysokých hodnotách môže počet stavov skutočne dosiahnuť pôvodný horný limit $3^m$ (bližšia analýza v [21]). Ich počet je preto pri väčších dĺžkach patternu hlavným faktorom limitujúcim použitie deterministických automatov. Wu, Manber a Myers vytvorili algoritmus (ďalej len „algoritmus WMM“), ktorý toto obmedzenie eliminuje a je teoreticky aplikovateľný na akýkoľvek algoritmus pre približné vyhľadávanie založený na výpočte dynamickej matice.

V algoritme WMM je každý stĺpec dynamickej matice rozdelený na viac častí, tzv. regiónov. Ich presná definícia bude podaná neskôr, zatiaľ je pre nás postačujúce, že tieto sú potom pri výpočte dynamickej matice počítané namiesto jednotlivých hodnôt, ako to zobrazuje obrázok 4.2. Ich veľkosť je naviac voliteľná, takže je možné algoritmus prispôsobiť konkrétneho systému, resp. jeho pamäťovým možnostiam.

6 V závislosti od zvolenej veľkosti regiónu. V pôvodnej práci autori používali veľkosť regiónu $r = 5$ a inkrementálnym výpočtom jednotlivých regiónov dosiahli časovú zložitosť $O(n)$.

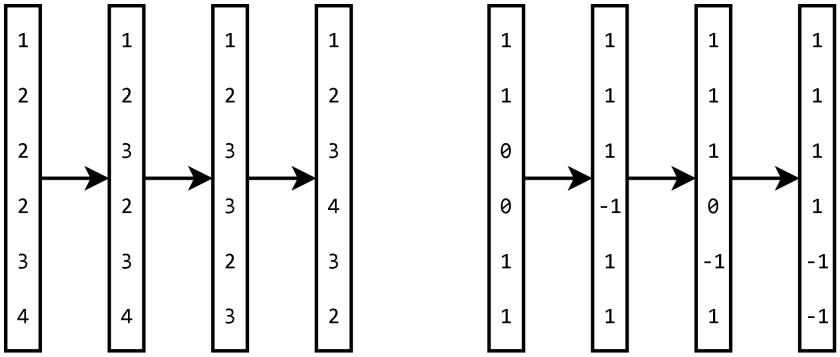



*Obrázok 4.2: Transformácia výpočtu pôvodnej dynamickej matice (vľavo) na výpočet regiónov algoritmu WMM (vpravo).*

Postup vytvárania regiónov preberieme v jednej z nasledujúcich podkapitol, predtým ale musíme objasniť použitie tzv. *diferencií*, ktoré budú v algoritme neskôr potrebné.

### 4.2.1 Diferencie v dynamickej matici

Podľa vety 2.4 vieme, že dve hodnoty editačných vzdialeností, ktoré sa nachádzajú v rovnakom stĺpci dynamickej matice priamo nad sebou, sa môžu líšiť maximálne o hodnotu `±1`. Využitím tohoto faktu môžeme upraviť základný Sellersov algoritmus tak, aby jednotlivé stĺpce dynamickej matice obsahovali namiesto reálnych hodnôt len tzv. *diferencie*, teda rozdiely medzi hodnotami nachádzajúcimi sa nad sebou. Obrázok 4.3 znázorňuje rozdiel medzi pôvodnými a novými stĺpcami dynamickej matice, vyjadrenými pomocou diferencií.




*Obrázok 4.3: Porovnanie stĺpcov dynamickej matice vyjadrených pomocou pôvodných hodnôt (vľavo) a pomocou diferencií (vpravo).*

Táto úprava samozrejme samotný výpočet matice nezrýchľuje, nakoľko v podstate meníme len značenie hodnôt vnútri stĺpcov. Bude však potrebná neskôr pri vytváraní regiónov, ktoré ju umožnia počítať rýchlejšie. Definujme si teraz formálne niektoré nové pojmy potrebné pre ďalšiu prácu:

Nech $I_{i,j}$ označuje[7] *diferenciu* $I_{i,j} = C_{i,j} - C_{i-1,j}$ pre $1 \le i \le m, 0 \le j \le n$.

Nech $cin_{i,j}$ označuje *carry-in hodnotu* $C_{i,j} - C_{i,j-1}$. Z uvedenej definície diferencie môžeme ľubovoľnú carry-in hodnotu vypočítať aj ako $cin_{i,j} = cin_{i-1,j} + I_{i,j} - I_{i,j-1}$ (odvodenie sa nachádza v [27]).

Ďalej označme $s[i,t] = \begin{cases} 0 & ak\ P_i = t \\ 1 & inak \end{cases}$ pre $1 \le i \le m$ a pre všetky $t \in \Sigma$.

Carry-in hodnoty sú potrebné vo výpočte k uloženiu rozdielu medzi hodnotou aktuálneho a starého stĺpca dynamickej matice a $s[i,t]$ slúži k porovnaniu daného znaku patternu s ľubovoľným znakom $t$ v abecede $\Sigma$.

Výpočet dynamickej matice pomocou diferencií sa potom riadi vzťahom 4.1. Ten je odvodený z pôvodného vzorca pre výpočet dynamickej matice 2.1; odvodenie a dôkaz správnosti sa nachádzajú v pôvodnom článku [27].

$$\begin{aligned} I_{i,0} &= 1 && pre\ 0 \le i \le m \\ I_{0,j} &= 0 && pre\ 0 \le j \le n \\ I_{i,j} &= \min \begin{cases} 1 \\ s[i,T_j] - cin_{i-1,j} \\ I_{i,j-1} - cin_{i-1,j} + 1 \end{cases} && pre\ 1 \le i \le m, 1 \le j \le n \end{aligned} \tag{4.1}$$

Môžeme teda povedať, že k výpočtu novej diferencie potrebujeme poznať tri hodnoty:

- $s[i,T_j]$, teda či je znak patternu $P_i$ zhodný so znakom textu $T_j$;
- diferenciu predchádzajúceho stĺpca $I_{i,j-1}$;
- a $cin_{i-1,j}$, čiže carry-in hodnotu získanú pri výpočte predchádzajúcej diferencie.

Poznamenajme, že pri výpočte matice postačuje mať uložený vždy len jeden stĺpec diferencií a nový počítať vždy do neho (tak, ako pri pôvodnom Sellersovom algoritme v podkapitole 2.3). Rovnako môžeme pristupovať aj ku carry-in hodnote, na ktorej uloženie stačí jednoduchá skalárna premenná (napr. `integer`). Pseudokód 7 demonštruje takto realizovaný algoritmus.

**Pseudokód 7 — Výpočet dynamickej matice pomocou diferencií**

```
1  for i = 1 to m:                    # inicializácia nultého stĺpca
2    I[i] = 1
3  for j = 1 to n:                    # pre každý znak textu…
4    cin = 0                          # carry-in hodnota je na začiatku 0
5    t = T[j]                         # prečítanie aktuálneho znaku textu
6    for i = 1 to m:                  # výpočet nového stĺpca...
7      oldI = I[i]                    # uloženie I[i, j-1]
```

7 V pôvodnom článku je diferencia označovaná ako $D_j[i]$, vzhľadom k použitiu značenia $D_{i,j}$ pre editačné hodnoty však budeme používať značenie $I_{i,j}$.

```
8       I[i] = min(1, s[i, t] - cin, I[i] - cin + 1)
9       cin += I[i] - oldI                  # aktualizácia carry-in hodnoty
```

Ďalším zaujímavým faktom je, že kým v pôvodnej dynamickej matici bol každý stĺpec tvorený $m+1$ hodnotami, pri použití diferencií nultý riadok potrebný nie je. Dôvod je ten, že pre výpočet $I_{1,j}$ nie je potrebná predchádzajúca diferencia $I_{0,j}$, pretože je vo vzťahu zastúpená carry-in hodnotou $cin_{0,j}$. Nultá carry-in hodnota má pre ľubovoľný stĺpec hodnotu $0$ (vychádzajúc z toho, že všetky hodnoty dynamickej matice v nultom riadku sa rovnajú nule), takže môže byť nastavená ešte pred samotným cyklom na výpočet nového stĺpca (4).

### 4.2.2 Vytvorenie regiónov

Po definovaní diferencií môžeme pokračovať popisom postupu vytvárania regiónov. Ako už bolo spomenuté, po rozdelení pôvodných stĺpcov dynamickej matice na viac regiónov bude výpočet každého regiónu závislý od troch predchádzajúcich regiónov (ako to bolo zobrazené na obrázku 4.2). Túto závislosť teraz bližšie preskúmame, najskôr si však definujme niektoré nové pojmy:

Nech $p$ je *index regiónu* veľkosti $r$, kde $1 \leq p \leq m/r$. Pre jednoduchosť zatiaľ predpokladajme, že $r$ delí $m$ bezo zvyšku, opačná situácia bude vysvetlená neskôr v podkapitole 4.2.6.

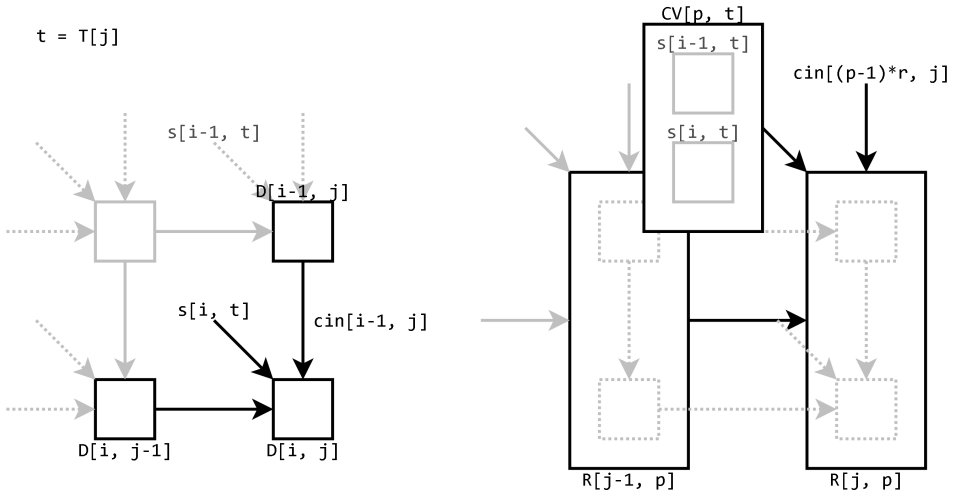
Nech $CV[p,t]$ je *charakteristický vektor* dĺžky $r$, ktorý je tvorený hodnotami $s[(p-1)*r+1,\ t],\ s[(p-1)*r+2,\ t],\ \ldots\ s[p*r,\ t]$.

Nech $I_j$ je *vektor diferencií* $I_{1\ldots m,j}$ pre $0 \leq j \leq n$.

Potom *región* $R[j,p]$[8] je pre každý index $p$ podvektor vektoru diferencií $I_j[(p-1)*r+1,\ \ldots,\ p*r]$.

Výpočet regiónov bude prebiehať veľmi podobne ako pri dynamickej matici vyjadrenej diferenciami, len s tým rozdielom, že jeden krok algoritmu vypočíta celý región, nielen jednu hodnotu. Situáciu znázorňuje obrázok 4.4.

8 V pôvodnej práci je región označovaný ako $D_j\langle p\rangle$, pre odlíšenie od editačných hodnôt $D_{i,j}$ bolo však zvolené toto označenie.




*Obrázok 4.4: Porovnanie výpočtu dynamickej matice (vľavo) a regiónov algoritmu WMM (vpravo) pomocou diferencií.*

Podľa obrázku teda každý nový región $\boldsymbol{R}[\boldsymbol{j}, \boldsymbol{p}]$ závisí od troch premenných:

- regiónu predchádzajúceho stĺpca $\boldsymbol{R}[\boldsymbol{j}-1, \boldsymbol{p}]$;
- vstupnej carry-in hodnote $\boldsymbol{cin}_{(p-1)*r,j}$, získanej po výpočte regiónu $\boldsymbol{R}[\boldsymbol{j}, \boldsymbol{p}-1]$;
- a charakteristického vektoru $\boldsymbol{CV}[\boldsymbol{p}, \boldsymbol{T}_j]$.

Použitie diferencií nám slúži k tomu, aby regióny nemuseli obsahovať hodnoty v rozmedzí $\langle 0; k+1 \rangle$, ale len v rozsahu $\langle -1; 1 \rangle$. Keďže každý región je tvorený $r$ diferenciami, celkový počet vygenerovaných regiónov bude potom $3^r$. Vďaka tejto úprave môžeme každý región zakódovať do rozsahu $\langle 0; 3^r \rangle$ a uložiť ako bežný `integer`[9]. Podobne môžeme postupovať aj pri charakteristických vektoroch—s tým rozdielom, že obsahujú len hodnoty `0` a `1`, môžeme ich teda prekódovať do celých čísel v rozsahu $\langle 0; 2^r \rangle$. V prípade carry-in hodnoty nie je takéto prispôsobenie potrebné, keďže je to len skalárna celočíselná hodnota v rozsahu $\langle -1; 1 \rangle$.

Po týchto úpravách môžeme tranzitívnu funkciu regiónov definovať takto:

> Nech $\boldsymbol{R}$ je ľubovoľný región, $\boldsymbol{CV}$ charakteristický vektor (obe v podobe celých čísel) a $\boldsymbol{cin}$ carry-in hodnota.
>
> Potom $h[\boldsymbol{R}, \boldsymbol{CV}, \boldsymbol{cin}] = (\boldsymbol{R'}, \boldsymbol{cin'})$ je *tranzítívna funkcia regiónov*, ktorá pre každú kombináciu $\boldsymbol{R}$, $\boldsymbol{CV}$ a $\boldsymbol{cin}$ vráti nový región $\boldsymbol{R'}$ a novú hodnotu $\boldsymbol{cin'}$.

9 Diferencie majú tri možné hodnoty (`-1`, `0` a `1`), po posunutí do intervalu `<0…2>` je možné každú z nich považovať za jednu cifru čísla v trojkovej sústave. Po prevode celého regiónu do desiatkovej sústavy tak získame výsledné zakódovanie (index) regiónu.

Poslednou otázkou teda zostáva, ako presne vygenerovať charakteristické vektory a samotnú tabuľku tranzitívnej funkcie. Pseudokód 8 zobrazuje postup vytvorenia tranzitívnej tabuľky $h$.

**Pseudokód 8 WMM algoritmus: generovanie regiónov**

```
1  maxR = 3^r - 1                          # počet rozličných regiónov
2  maxCV = 2^r - 1                         # počet charakteristických vektorov
3  h = []
4  for R = 0 to maxR:                      # pre každý rozličný región…
5    decodedR = decode(R)
6    for CV = 0 to maxCV:                  # pre každý charakteristický vektor…
7      decodedCV = decode(CV)
8      for cin = -1 to 1:                  # pre každú carry-in hodnotu…
9        R2 = []
10       cin2 = cin
11       for i = 1 to r:                   # výpočet nového regiónu
12         R2[i] = min(1, decodedCV[i] - cin2, decodedR[i] - cin2 + 1)
13         cin2 += R2[i] - decodedR[i]
14       h[R, CV, cin] = [encode(R2), cin2] # uloženie prechodu
```

Algoritmus generuje tabuľku presne tak, ako to naznačuje jej definícia: vypočíta tranzíciu pre každú možnú kombináciu $R$, $CV$ a $cin$. Vonkajší cyklus (4-14) postupne prebieha pre každý možný región $R$. Funkcia `decode` (5) následne z celočíselného indexu regiónu získa pôvodný región vo forme diferencií, tie totiž budú potrebné pri počítaní nového regiónu (12). V strednom cykle (6-14) prechádza algoritmus každé možné číslo charakteristického vektoru $CV$ a rovnako aj z neho dekóduje pôvodný vektor (7). Vnútorný cyklus (8-14) je nakoniec vykonaný trikrát pre každú možnú carry-in hodnotu `-1`, `0` a `1`. Výpočet nového regiónu prebieha až v najvnútornejšom cykle (11-13). Po ňom algoritmus uloží prechod do tranzitívnej tabuľky $h$ (14), pričom je nový región $R_2$ pred uložením prekódovaný do celočíselného vyjadrenia funkciou `encode`.

### 4.2.3 Vytvorenie charakteristických vektorov

Nakoniec ostáva už len vygenerovať tabuľku charakteristických vektorov $CV$. Jej úlohou je pre každý prijatý znak textu $T_j$ v závislosti od aktuálneho indexu regiónu $p$ vrátiť prislúchajúci charakteristický vektor dĺžky $r$ prekódovaný do celého čísla. Pseudokód 9 zobrazuje celý algoritmus, prekódovanie vektoru opäť zabezpečuje funkcia `encode` (9). Hoci to z priebehu algoritmu vyplýva, poznamenajme, že ku vygenerovaniu charakteristických vektorov potrebujeme poznať nielen vyhľadávaný pattern, ale aj abecedu textu (1).

**Pseudokód 9 WMM algoritmus: vytváranie charakteristických vektorov**

```
1  for t in alphabet:                      # pre každý znak v abecede textu…
2    for p = 1 to m/r:                     # pre každý región…
3      vector = []
4      for j = 1 to r:                     # výpočet charakteristického vektoru
5        if t == P[(p-1)*r + j]            # zhoda znakov
6          vector[j] = 0
7        else                              # odlišnosť znakov
```

```
8           vector[j] = 1
9       CV[p, t] = encode(vector)          # uloženie vektoru
```

### 4.2.4 Vyhľadávanie pomocou regiónov

Po vytvorení tranzitívnej tabuľky a charakteristických vektorov prebieha samotné vyhľadávanie pomerne jednoducho (Pseudokód 10). Algoritmus najprv inicializuje všetky regióny tak, aby zodpovedali diferenciám nultého stĺpca dynamickej matice, teda samým `1` (1-2). Taktiež nastaví premennú `sum` na súčet diferencií nultého stĺpca (3)[10]. Následne prebieha algoritmus pre každý znak textu (4-11), kde sa vždy najprv nastaví premenná `cin` na `0` (6, vysvetlenie v závere podkapitoly 4.2.1). Po prechode do nových regiónov (7-8) je podľa poslednej carry-in hodnoty aktualizovaný aj súčet diferencií (9). V závere už zostáva len overiť, či tento súčet nepresiahol povolenú editačnú vzdialenosť $k$ (10-11).

**Pseudokód 10 WMM algoritmus: vyhľadávanie**

```
1   for p = 1 to m/r:                      # inicializácia regiónov na nultý
2     R[p] = 3^r – 1                       #   stĺpec (regióny zo samých jednotiek)
3   sum = m                                # začiatočný súčet diferencií stĺpca
4   for j = 1 to n:                        # pre každý znak textu…
5     t = T[j]
6     cin = 0                              # carry-in hodnota je na začiatku 0
7     for p = 1 to m/r:                    # pre každý región… (prechod)
8       R[p], cin = h[ R[p], CV[p, t], cin ]
9     sum = sum + cin                      # aktualizácia súčtu
10    if sum ≤ k:                          # našla sa zhoda?
11      print "Zhoda na pozícii " + j + ", vzdialenosť " + sum
```

### 4.2.5 Univerzálnosť tranzitívnej funkcie

Dôležitou vlastnosťou tranzitívnej funkcie $h$, ktorú bolo možné vypozorovať už z algoritmu pre jej vytvorenie (Pseudokód 8), je jej nezávislosť od konkrétneho patternu, textu či abecedy. Pri každom prechode je síce potrebný charakteristický vektor $CV[p, T_j]$ (14), ale jeho hodnoty vyjadrujú len to, či sa znaky patternu v danom rozsahu (podľa indexu regiónu $p$) zhodujú s určitým znakom textu. *Neobsahujú* však žiadnu informáciu o tom, o aké znaky sa jedná. Túto informáciu obsahuje tabuľka charakteristických vektorov, ktorá je vytvorená vždy pre daný pattern a abecedu textu tesne pred vyhľadávaním.

Z tohoto pozorovania vychádza dôležitý záver: regióny aj tranzitívnu tabuľku je potrebné vytvoriť len raz pri inicializácii algoritmu určením požadovanej veľkosti regiónov, pretože nie sú závislé na žiadnom inom údaji. Pri každom vyhľadávaní (nového) patternu už stačí vytvoriť len tabuľku charakteristických vektorov, ktorej vytvorenie je v porovnaní s vyhľadávaním samotným veľmi rýchle, nie celú tranzitívnu tabuľku.

---

10 Hoci to z definícií vyplýva, poznamenajme, že súčet diferencií stĺpca vždy zodpovedá poslednej hodnote v pôvodnej dynamickej matici, teda $I_{m,j}$. Z toho taktiež môžeme ľahko vyvodiť, že súčet diferencií všetkých regiónov jedného stĺpca sa tiež rovná tejto hodnote.

### 4.2.6 Nedeliteľnosť patternu veľkosťou regiónov

V definícii indexu regiónu (podkapitola 4.2.2) sme implicitne predpokladali deliteľnosť dĺžky patternu $m$ veľkosťou regiónu $r$ bezo zvyšku. V tejto podkapitole preto preberieme úpravy potrebné k správnemu fungovaniu algoritmu aj v opačných prípadoch. Zmeny sa týkajú tabuľky charakteristických vektorov a samotného vyhľadávania.

Charakteristické vektory musíme vytvárať tak, akoby bola dĺžka patternu o toľko väčšia, aby bola bezo zvyšku deliteľná veľkosťou regiónov. Označme si počet znakov, o ktoré takto „predĺžime“ pattern, ako *padding*. K výpočtu posledných hodnôt tých vektorov, ktoré prislúchajú koncu patternu, nám potom v patterne bude chýbať práve `padding` znakov. Nakoľko však na chýbajúcich znakoch nezáleží, nastavíme posledné hodnoty týchto vektorov na `0`, čo bude zodpovedať zhodnosti s akýmkoľvek znakom textu. Tým zaistíme, že tieto znaky nebudú mať na výpočet žiadny vplyv. Pseudokód 11 obsahuje upravenú podmienku pôvodného riadku (5) vo výpočte tabuľky charakteristických vektorov (Pseudokód 9).

**Pseudokód 11 WMM algoritmus: padding patternu**

```
1  if (p-1)*r + j > m  or  t == P[(p-1)*r + j]
```

Uvedený postup má už len niekoľko posledných chýb, ktoré musíme vyriešiť. Prvou je, že sme virtuálne predĺžili pattern, a tak sa teraz pri vyhľadávaní budú všetky približné zhody patternu s textom objavovať `padding` znakov za ich reálnou pozíciou. To vyriešime tak, že pri inicializácii túto hodnotu pripočítame k premennej `sum` (3) a pri vyhľadávaní ju naopak odčítame od pozície každej približnej zhody $j$. Kvôli predĺženiu patternu tiež nie sú pri konci textu (posledných `padding` znakov) detekované zhody s patternom. K oprave tejto chyby stačí predĺžiť o `padding` ľubovoľných znakov aj samotný text $T$.

### 4.2.7 Optimalizácia: transformácia tabuľky prechodov

V pôvodnom algoritme (Pseudokód 8) má tabuľka prechodov $h$ tri dimenzie. Jednoduchým preskúmaním ich rozsahov ($cin \in \langle -1; 1 \rangle$, $CV \in \langle 0; 2^r \rangle$ a $R \in \langle 0; 3^r \rangle$) však zistíme, že ak upravíme celočíselnú reprezentáciu každého charakteristického vektoru $CV$ na $3 * CV + 1$ a každého regiónu $R$ na $3*2^r*R$, môžeme pretransformovať tabuľku do jedinej dimenzie, čím docielime rýchlejšiu indexáciu jej hodnôt.

Po tejto úprave bude mať potom akýkoľvek prístup k tabuľke (či už pri vytváraní jej prechodov alebo vo fáze vyhľadávania) tvar $h[R + CV + cin]$ a tabuľka samotná tak bude mať rozsah $\langle 0; 3*6^r \rangle$.

### 4.2.8 Optimalizácia: minimalizácia výpočtov

V druhej časti pôvodnej práce [27] autori použili na svoj algoritmus podobnú optimalizáciu na vynechanie zbytočných výpočtov, ako bola použitá v základnom suffix–tree algoritme (podkapitola 3.3.3). Vzhľadom k tomu, že pracujeme s regiónmi, označme si ako *valídne regióny*

tie, ktoré obsahujú aspoň jednu valídnu editačnú hodnotu, a posledný z nich označme ako *posledný valídny región* (ten obsahuje poslednú valídnu hodnotu). Myšlienka je taká, že po prijatí znaku vo fáze vyhľadávania môžeme ukončiť výpočet jeden región za posledným valídnym regiónom z predchádzajúceho stĺpca, pretože ďalšie regióny nemôžu byť valídne (podľa vety 2.4).

K aplikácii optimalizácie je potrebná relatívne komplexná zmena algoritmu, preto si ho vysvetlíme priamo na zdrojovom kóde (Pseudokód 12). Na začiatku je potrebné určiť celkový počet regiónov (1), posledný valídny z nich (2) a všetky inicializovať (3-4). Súčet diferencií určíme len pre valídne regióny (5).

V hlavnom cykle budeme najprv počítať regióny po posledný valídny z predchádzajúceho stĺpca (9). Prechodová funkcia musí odteraz vracať aj novú hodnotu $rSum[p]$, ktorá obsahuje informáciu o súčte diferencií daného regiónu.

Len ak bude súčet regiónov rovný povolenej vzdialenosti $k$ (a ich počet menší ako celkový počet), budeme pre istotu počítať jeden ďalší región (11-14)[11]. Ak táto podmienka nie je splnená, algoritmus hľadá posledný valídny región v aktuálnom stĺpci (17-19). Keďže je však presné určenie výpočetne príliš náročné[12], región je hľadaný len približne: overí sa, či aktuálny súčet je vyšší od povolenej vzdialenosti aspoň o $r$, v takom prípade totiž región nemôže obsahovať žiadnu valídnu hodnotu. Na záver ostáva opäť už len overiť, či sa nenašla približná zhoda (20-21).

**Pseudokód 12 WMM algoritmus: optimalizované vyhľadávanie**

```
1  regions = (m + padding)/r              # získanie počtu regiónov
2  last = k / r                           # posledný valídny región
3  for p = 1 to last:                     # inicializácia regiónov na nultý
4    R[p] = 3^r - 1                       #   stĺpec (regióny zo samých jednotiek)
5  sum = last * r                         # začiatočný súčet diferencií stĺpca
6  for j = 1 to n + padding:              # pre každý znak textu…
7    t = T[j]
8    cin = 0                              # carry-in hodnota je na začiatku 0
9    for p = 1 to last:                   # pre každý valídny región…
10     R[p], cin, rSum[p] = h[ R[p] + CV[p, t] + cin ] # prechod
11   if sum == k and last < regions:      # treba počítať jeden ďalší región?
12     last += 1                          # zvýšenie počtu regiónov
13     R[last], cin = h[ (3^r-1) + CV[(last, t)] + cin ] # prechod
14     sum += r + cin                     # aktualizácia súčtu (s novým regiónom)
15   else:
16     sum += cin                         # aktualizácia súčtu
17     while sum > k + r:                 # hľadanie posledného valídneho regiónu
18       sum -= rSum[last]                # znižovanie súčtu o nevalídne regióny
19       last -= 1
20   if last == regions and sum ≤ k:      # našla sa zhoda?
21     print "Zhoda na pozícii " + (j - padding) + ", vzdialenosť " + sum
```

11 Podmienka je dostatočná, pretože súčet nemôže byť menší ako $k$, inak by bol v predchádzajúcom stĺpci vygenerovaný ďalší región a premenná `last` by bola o `1` vyššia.

12 V pôvodnej práci [27] autori najprv použili presný výpočet, no po testoch zistili, že približné určenie je efektívnejšie.

### 4.2.9 Časová a priestorová zložitosť

Určiť časovú a priestorovú zložitosť algoritmu nie je zložité, keďže počty regiónov, charakteristických vektorov aj prechodov tranzitívnej funkcie poznáme dopredu.

**Časová zložitosť**

Časová zložitosť výpočtu charakteristických vektorov je $O(|\Sigma| * m)$, pretože pre každý znak z abecedy musí algoritmus vypočítať $m$ hodnôt. Ďalej vieme, že počet tranzícií je po transformácii tabuľky prechodov (podkapitola 4.2.7) vždy $3*6^r$ a výpočet jednej trvá $O(r)$. Dostávame teda celkovú časovú zložitosť fázy predspracovania $O(r * 6^r)$.

Vo fáze vyhľadávania trvá výpočet jedného stĺpca dynamickej matice $O(m/r)$, pretože stĺpec dĺžky $m$ je počítaný po jednotlivých regiónoch. Celkový čas vyhľadávania je preto $O(n * m/r)$.

**Priestorová zložitosť**

Tabuľka charakteristických vektorov potrebuje miesto $O(|\Sigma| * m/r)$, pretože na každý znak pripadá $O(m/r)$ charakteristických vektorov zakódovaných do celého čísla.

Tabuľka prechodov potrebuje mať uložené dve celé čísla (nový región a carry-in hodnotu) pre každú vstupnú kombináciu regiónu, charakteristického vektoru a carry-in hodnoty. Keďže týchto kombinácií je spolu $3*6^r$, je celková priestorová zložitosť $O(2*3*6^r) = O(6^r)$.

## 4.3 Hybridný suffix–tree algoritmus s blokovým výpočtom

V tejto podkapitole podrobne vysvetlíme navrhovaný hybridný algoritmus, ktorý vznikne úpravou a prispôsobením algoritmu WMM z predchádzajúcej podkapitoly a jeho následnou kombináciou so základným suffix–tree algoritmom z podkapitoly 3.3.

Priebeh hybridného algoritmu môžeme rozdeliť rozdeliť na tri fázy: fázu generovania regiónov a tranzitívnej funkcie, fázu vytvorenia suffixového stromu po zadaní indexovaného textu a fázu samotného vyhľadávania po zadaní patternu a editačnej vzdialenosti. Generovanie charakteristických vektorov je pritom súčasťou poslednej fázy, keďže pre svoj výpočet vyžaduje znalosť vyhľadávaného patternu.

### 4.3.1 Generovanie regiónov a tranzitívnej funkcie

Vytváranie regiónov a tranzitívnej funkcie prebieha rovnako, ako v pôvodnom algoritme WMM (podkapitola 4.2.2). Po aplikácii optimalizácií na transformáciu tabuľky prechodov (podkapitola 4.2.7) a minimalizáciu výpočtov (podkapitola 4.2.8) vyzerá potom generovanie regiónov tak, ako to ukazuje Pseudokód 13. Vidíme, že transformácia tabuľky spôsobila, že indexy regiónov a charakteristických vektorov sa ukladajú do tranzitívnej funkcie multiplikované (18-19) o konštanty uvedené v podkapitole 4.2.7 (3-4). Optimalizácia výpočtov si zase vyžaduje ukladať do tranzitívnej funkcie aj súčet diferencií v každom regióne (17).

**Pseudokód 13** **Hybridný algoritmus: generovanie regiónov**

```
maxR = 3^r                              # počet rozličných regiónov
maxCV = 2^r                             # počet charakteristických vektorov
rMultiplier = maxCV*3                   # inkrementory regiónov a CV pre
cvMultiplier = 3                        #   transformáciu tranzitívnej funkcie
h = []
for R = 0 to maxR - 1:                  # pre každý rozličný región…
  decodedR = decode(R)
  for CV = 0 to maxCV - 1:              # pre každý charakteristický vektor…
    decodedCV = decode(CV)
    for cin = -1 to 1:                  # pre každú carry-in hodnotu…
      R2 = []
      cin2 = cin
      rSum = 0                          # súčet diferencií regiónov je na začiatku 0
      for i = 1 to r:                   # výpočet nového regiónu
        R2[i] = min(1, decodedCV[i] - cin2, decodedR[i] - cin2 + 1)
        cin2 += R2[i] - decodedR[i]     # aktualizácia hodnôt carry-in
        rSum += R2[i]                   #   a súčtu diferencií regiónu
      h[R*rMultiplier, (CV*cvMultiplier)+1, cin] # uloženie prechodu
        = [encode(R2)*rMultiplier, cin2, rSum]
```

### 4.3.2 Vytvorenie suffixového stromu a zistenie abecedy textu

Indexácia textu do suffixového stromu prebieha bezo zmeny tak, ako pri pôvodnom suffix–tree algoritme. Zmenou je spôsob určenia abecedy textu. V pôvodnom WMM algoritme sa implicitne predpokladalo, že abeceda textu bude zadaná kvôli počítaniu charakteristických vektorov spolu s patternom, keďže ide o online algoritmus, ktorý predpokladá neznalosť samotného textu predom.

V našom offline hybridnom algoritme je situácia opačná: text poznáme predom a pattern je zadaný až v momente, keď ho treba vyhľadávať. Abecedu textu je preto možné zistiť priamo z textu a vďaka štruktúre suffixového stromu je tento krok veľmi jednoduchý. Z koreňového uzlu ***root*** totiž vychádzajú hrany pre každý rozličný znak, ktorým začína niektorý suffix textu. Zistenie celej abecedy textu je teda otázkou prejdenia všetkých existujúcich `goto`-prechodov z koreňového uzlu a získanie zodpovedajúcich znakov (podreťazcov dĺžky `1`).

### 4.3.3 Generovanie charakteristických vektorov

Tabuľku charakteristických vektorov je možné vygenerovať až v poslednej fáze, keď je zadaný vyhľadávaný pattern. V pôvodnom algoritme (podkapitola 4.2.3) tvorila tabuľka dvojrozmerné pole o veľkosti $m/r * |\Sigma|$, kde $m/r$ značí počet regiónov (zaokrúhlený hore, pokiaľ je dĺžka patternu nedeliteľná veľkosťou regiónov). Pre urýchlenie prístupu môžeme rovnako ako pri tranzitívnej tabuľke transformovať do jednorozmerného poľa aj tabuľku charakteristických vektorov.

K tomu potrebujeme vytvoriť pomocnú tabuľku `charCodes`$(t) = i$, ktorá pre každý znak textu $t \in \Sigma$ vráti jeho unikátny index $i$. Algoritmus vytvorenia je intuitívny, pre ilustráciu ho uvádza Pseudokód 14.

**Pseudokód 14** **Hybridný algoritmus: vytvorenie indexu abecedy textu**

```
1  charCodes = []                          # inicializácia tabuľky kódov
2  index = 0                               # index prvého znaku
3  for t in alphabet:                      # pre každý znak v abecede textu…
4    charCodes[t] = index                  # ulož znak do indexu
5    index += r                            # inkrementácia indexu o veľkosť regiónu
```

S vytvorenou tabuľkou znakových kódov môžeme jednorozmernú tabuľku charakteristických vektorov vytvoriť tak, ako to ukazuje Pseudokód 15.

**Pseudokód 15** **Hybridný algoritmus: generovanie charakteristických vektorov**

```
1  for t in alphabet:                      # pre každý znak v abecede textu…
2    for p = 1 to m/r:                     # pre každý región…
3      vector = []
4      for j = 1 to r:                     # výpočet charakteristického vektoru
5        if t == P[(p-1)*r + j]:           # zhoda znakov
6          vector[j] = 0
7        else:                             # odlišnosť znakov
8          vector[j] = 1
9      CV[charCodes[t] + p] = encode(vector) # uloženie vektoru
```

### 4.3.4 Vyhľadávanie v suffixovom strome

V algoritme vyhľadávania dochádza oproti pôvodnému algoritmu k najväčším zmenám, keďže je potrebné vyhľadávanie algoritmu WMM injektovať do procesu priebehu suffixového stromu základného suffix–tree algoritmu. K tomu je potrebné o každom uzle v množine nepreskúmaných uzlov udržovať dodatočné informácie o zodpovedajúcich regiónoch. Celý priebeh (s aplikovanými optimalizáciami z podkapitol 4.2.7 a 4.2.8) zobrazuje Pseudokód 16.

Hneď na začiatku vidíme zmenu oproti pôvodnému algoritmu, keď pre koreňový uzol ukladáme hodnoty všetkých regiónov, index posledného validného regiónu a celkovú sumu diferencií (5-10). Pri prechádzaní nepreskúmaných uzlov postupne vykonáme každý možný `goto`-prechod (12) a tieto informácie vždy načítame (15-17). Následne urobíme `h`-prechod pre každý validný región a aktualizujeme potrebné dodatočné informácie (20-30). Overovanie približnej zhody spočíva v zistení, či sú všetky regióny validné a celková suma diferencií je menšia ako editačná vzdialenosť $k$ (31-32).

Zaujímavý je spôsob, akým je aplikovaná optimalizácia základného algoritmu z podkapitoly 3.3.3 pre ukončenie výpočtu v uzloch, ktorých všetky hodnoty sú nevalídne. Nie je možné priamo overiť, či je aspoň jeden región validný—museli by sme z celočíselnej reprezentácie regiónu spätne získavať jednotlivé diferencie, čo by bolo časovo náročné. Algoritmus preto overuje len hornú hranicu, pri ktorej je isté, že sú všetky hodnoty v regiónoch väčšie ako $k$ nezávisle od konkrétneho regiónu. Toto docielime tým, že uvažujeme najhorší možný región (diferencie sú samé `1`). Nový uzol je teda pridaný do nepreskúmaných vtedy, pokiaľ je súčet diferencií validných regiónov po sčítaní s hĺbkou uzlu menší alebo rovný súčtu editačnej vzdialenosti a maximálneho súčtu

validných regiónov (34). V takom prípade je potom pridanie uzlu sprevádzané uložením aktuálnych pomocných informácií k novému uzlu (34-37).

**Pseudokód 16 Hybridný algoritmus: vyhľadávanie**

```
1  rCount = ceil(m / r)                    # celkový počet regiónov
2  rMultiplier = (2^r)*3                   # inkrementor regiónov
3  maxRegion = (3^r - 1)*rMultiplier       # región zo samých jednotiek
4
5  new = {root}                            # množina nepreskúmaných uzlov
6  lastValid(root) = ceil(k / r)           # posledný validný región
7  sum(root) = lastValid(root) * r         # súčet diferencií validných regiónov
8  for p = 1 to lastValid(root):           # inicializácia regiónov na nultý
9    R[p] = maxRegion                      #   stĺpec (regióny zo samých jednotiek)
10 regions(root) = R                       #   a ich uloženie
11 while not new.empty():                  # kým existujú nepreskúmané uzly…
12   s = new.pop()                         # vyber uzol
13   foreach t where (s, t) in goto:       # pre každý prechod z uzlu…
14     s2 = goto(s, t)                     # prejdi na nový uzol
15     last = lastValid(s)                 # načítaj informácie patriace k uzlu
16     sum = sum(s)
17     R2 = regions(s)
18     charCode = charCodes[t]             # zisti kód znaku
19     cin = 0                             # carry-in hodnota je na začiatku 0
20     for p = 1 to lastValid(s):          # pre každý validný región…
21       R2[p], cin, rSum[p] = h[ R[p] + CV[charCode + p] + cin ] # prechod
22     if sum(s) == k and lastValid(s) < rCount: # treba počítať jeden ďalší región?
23       last += 1                         # zvýšenie počtu regiónov
24       R2[last], cin, _ = h[ maxRegion + CV[charCode + last] + cin ] # prechod
25       sum += r + cin                    # aktualizácia súčtu (s novým regiónom)
26     else:
27       sum += cin                        # aktualizácia súčtu
28       while sum > k + r:                # hľadanie posledného validneho regiónu
29         sum -= rSum[last]               # znižovanie súčtu o nevalídne regióny
30         last -= 1                       #   a ich počtu
31     if last == regionCount and sum ≤ k: # našla sa zhoda?
32       print "Zhoda na pozícii " + (pos(s2) - padding) + ", vzdialenosť " + sum
33     if sum + depth(s2) ≤ k + (last*r) + 1: # ak je aspoň jeden región validny
34       new.add(s2)                       #   pridaj nový uzol do nepreskúmaných
35       lastValid(s2) = last              #   a ulož k nemu zodpovedajúce informácie
36       sum(s2) = sum
37       regions(s2) = R2
```

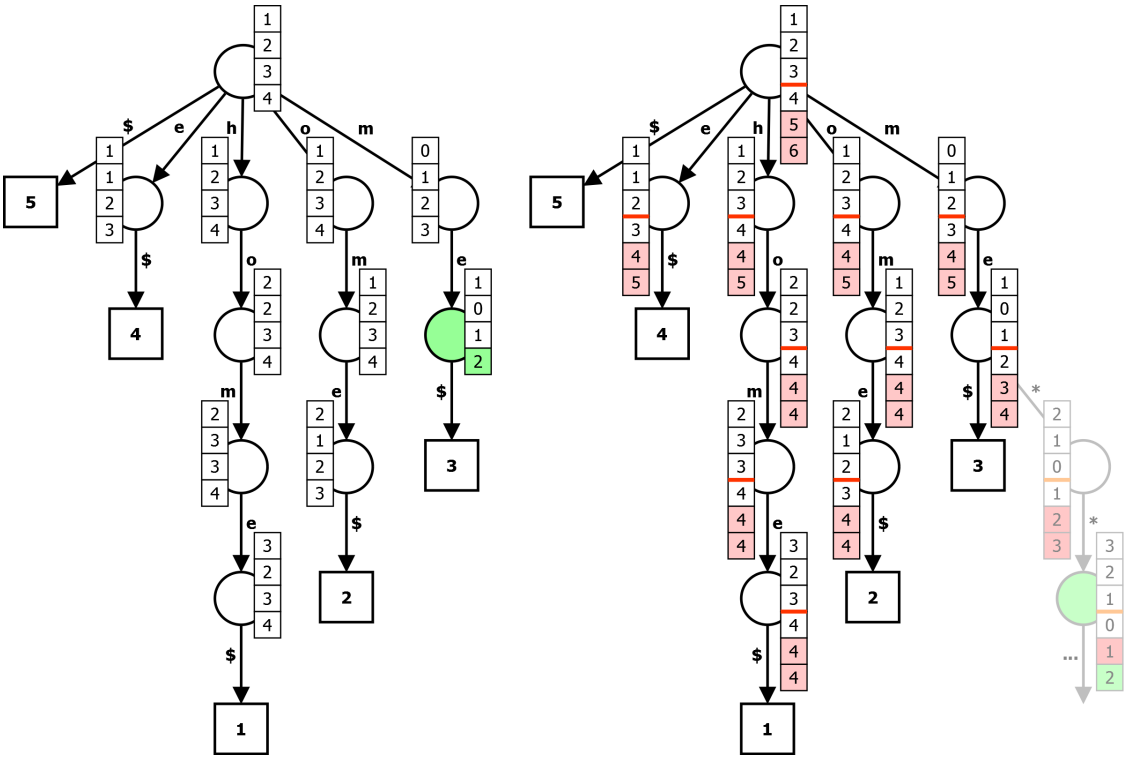
### 4.3.5 Detekovanie približných zhôd v listových uzloch

Predchádzajúci algoritmus obsahuje logickú chybu, kvôli ktorej môže vynechať niektoré približné zhody s editačnou vzdialenosťou väčšou ako `0`. Chyba sa prejaví len v prípade, že je na pattern potrebné použiť padding (tj. veľkosť regiónov nedelí pattern bezo zvyšku) *a zároveň* sa približná zhoda nachádza na konci textu. Modelovú situáciu znázorňuje obrázok 4.5. Surffixový strom je vytvorený nad textom „`home`“, vyhľadávaný pattern je „`men`“ a povolená editačná vzdialenosť je `2`.

V prípade základného suffix–tree algoritmu (na obrázku vľavo) by bola približná zhoda vo vzdialenosti `2` nájdená v uzle, ktorý predstavuje podreťazec textu „`me`“.

Pri použití nášho hybridného algoritmu s regiónmi veľkosti `3` musí byť ale pattern umelo predĺžený o `2` „ľubovoľné“ znaky. Zodpovedajúce charakteristické vektory sú síce predĺžené tak, aby v posledných `padding` riadkoch akceptovali akékoľvek znaky (podkapitola 4.2.6), no ku približnej zhode pri vyhľadávaní *aj tak* musí prísť až o `padding` znakov ďalej, kde bude pozícia zhody skorigovaná (Pseudokód 12, riadok 21).

Z tohoto dôvodu sme v algoritme WMM predlžovali o `padding` ľubovoľných znakov aj samotný text. V suffixovom strome by tento krok mal neželané následky pri vyhľadávaní (okrem toho by musel byť *každý* existujúci suffix textu predĺžený, čo by negatívne ovplyvnilo veľkosť stromu). Dôsledkom je, že na konci textu nie je možné vykonať posun o `padding` znakov (tj. `goto`-prechodov), ako to naznačuje šedo zaznačená časť suffixového stromu vpravo.



*Obrázok 4.5: Porovnanie správneho nájdenia približnej zhody patternu „`meta`“ v základnom suffix–tree algoritme (vľavo) a chybného výpočtu neopraveného hybridného algoritmu (vpravo). Oranžovou sú znázornené hranice regiónov, červenou hodnoty pridané paddingom, kvôli ktorému sa zhoda nenašla. Zošednutá časť ukazuje, prečo by sa chyba neprejavila, keby uzol nebol listový.*

Riešenie problému ukazuje Pseudokód 17. Úpravou postupu vyhľadávania môžeme po výbere každého nepreskúmaného uzlu najprv zistiť, či je listový. Implementačne netreba do uzlov pridávať žiadnu novú informáciu, stačí vždy overiť, či z uzlu vedie aspoň jeden `goto`-prechod. V prípade, že je strom interný, prebieha algoritmus podľa pôvodného postupu (Pseudokód 16). V opačnej situácii

algoritmus overí, či by ku približnej zhode na posledných `padding` pozíciách v texte došlo, keby nespôsobil navýšenie súčtov diferencií (8-10) a v kladnom prípade jednoducho nahlási zhody (11).

**Pseudokód 17 Hybridný algoritmus: vyhľadávanie s detekciou v listových uzloch**

```
1  ...
2  while not new.empty():                  # kým existujú nepreskúmané uzly…
3    s = new.pop()                         # vyber uzol
4    if not isLeaf(s):                     # nelistový uzol (s goto-prechodmi)
5      foreach t where (s, t) in goto:     # pre každý prechod z uzlu…
6        ...                               # zvyšok algoritmu
7    else:                                 # listový uzol
8      for offset = padding-1 downto 0:
9        realSum = sum(s) - padding + offset
10       if lastValid(s) == regionCount and realSum ≤ k: # našla sa zhoda?
11         print "Zhoda na pozícii " + (pos(s2) - offset) + ", vzdialenosť " + realSum
```

### 4.3.6 Optimalizácia: vylepšenie heuristiky prechodu stromu

Optimalizácia, ktorú v tejto podkapitole predstavíme, sa týka jednej z charakteristík základného suffix–tree algoritmu, ktorú sme uviedli hneď na začiatku jeho popisu (podkapitola 3.3). Inicializácia nultých hodnôt jednotlivých stĺpcov dynamickej matice sa nevykonávala pomocou hodnôt $C_{0,j} = 0$ zo vzorca 2.1, ale editačnými hodnotami $D_{0,j} = j$ (ako bolo uvedené vo vzorci 3.1). Dôvod bol ten, že v suffixovom strome nemusíme zabezpečovať potenciálny začiatok zhody na ľubovoľnej pozícii v texte tým, že nastavíme hodnoty prvého riadku dynamickej matice na `0`, pretože táto funkčnosť je obsiahnutá priamo v štruktúre stromu existenciou cesty z koreňového uzlu pre každý suffix textu.

Hoci to vtedy nebolo spomenuté, táto zmena v skutočnosti nebola nutná pre správne fungovanie algoritmu vďaka definovaniu kanonického výstupu približných zhôd (podkapitola 3.3.1). Ten v prípade viacerých zhôd na jednej pozícii textu uprednostňuje tie, ktoré majú najmenšiu editačnú vzdialenosť a v prípade viacerých takých zhôd zase tie, ktoré majú menšiu dĺžku zhody (tj. nájdeného podreťazca textu). V prípade inicializácie nultých hodnôt stĺpcov na `0` by preto jediným dôsledkom bolo, že algoritmus nahlási aj „zbytočne“ vzdialené či dlhé zhody, no tie by boli nakoniec správne vyfiltrované a konečný výstup by teda bol korektný.

Keď sa pozorne zadívame na priebeh hybridného algoritmu (generovanie regiónov v podkapitole 4.3.1 a vyhľadávanie v podkapitole 4.3.4), zistíme, že pracuje s pôvodným vzorcom pre inicializáciu nultých hodnôt $C_{0,j} = 0$. Jednotlivé regióny totiž neuchovávajú informáciu o skutočnej editačnej vzdialenosti, len jednotlivé diferencie (región $p$ „nevie“, na akej editačnej vzdialenosti skončil predchádzajúci región $p-1$). V samotnom vyhľadávaní sa potom hĺbka uzlu využíva len pri detekcii, či bude nový uzol pridaný do zoznamu nepreskúmaných. Hoci je však takýto výpočet korektný vďaka kanonickému výstupu, negatívne sa prejavuje na rýchlosti algoritmu, pretože sú zbytočne prechádzané uzly, ktoré sú v príliš veľkej hĺbke na to, aby mohli viesť k úspešnej zhode.

Riešením je vložiť penalizáciu za hĺbku uzlu do nultej carry-in hodnoty $cin_{0,j}$ jej nastavením na hodnotu `1` namiesto `0` (ako v pôvodnom algoritme). Keďže táto carry-in hodnota je používaná

ako vstup do výpočtu prvého regiónu stĺpca a značí diferenciu z (virtuálneho) predchádzajúceho regiónu, takáto zmena spôsobí zvýšenie celkového súčtu diferencií stĺpca s každým ďalším `goto`-prechodom. Upravené fragmenty algoritmu ukazuje Pseudokód 18. Poslednou zmenou, ktorú je potrebné urobiť, je ošetriť hľadanie posledného valídneho regiónu tak, aby v cykle nedošlo k prechodu do záporných hodnôt (12).

**Pseudokód 18 Hybridný algoritmus: vylepšenie heuristiky pri vyhľadávaní**

```
1  ...
2  while not new.empty():                # kým existujú nepreskúmané uzly…
3    s = new.pop()                       # vyber uzol
4    foreach t where (s, t) in goto:     # pre každý prechod z uzlu…
5      ...
6      cin = 1                           # nastav carry-in hodnotu na 1, nie 0!
7      ...
8      if sum(s) == k and lastValid(s) < rCount: # treba počítať jeden ďalší región?
9        ...
10     else:
11       sum += cin                      # aktualizácia súčtu
12       while sum > k + r and last > 0:  # hľadanie posledného valídneho regiónu
13         ...
```

### 4.3.7 Časová a priestorová zložitosť

Nakoľko je hybridný algoritmus kombináciou dvoch odlišných prístupov, je jeho časová aj priestorová zložitosť závislá od oboch algoritmov, ktoré kombinuje: základného suffix–tree algoritmu aj algoritmu WMM.

Priebeh hybridného algoritmu môžeme rozdeliť rozdeliť na tri fázy: fázu generovania regiónov a tranzitívnej funkcie, fázu vytvorenia suffixového stromu po zadaní indexovaného textu a fázu samotného vyhľadávania po zadaní patternu a editačnej vzdialenosti. Generovanie charakteristických vektorov je pritom súčasťou poslednej fázy, keďže pre svoj výpočet vyžaduje znalosť vyhľadávaného patternu.

**Časová zložitosť**

Časová zložitosť indexácie textu sa oproti základnému suffix–tree algoritmu nemení, je teda stále rovná $\boldsymbol{O}(\boldsymbol{n})$. Počet prechodov v tranzitívnej tabuľke je vždy $3*6^r$ a výpočet každého regiónu trvá $\boldsymbol{O}(\boldsymbol{r})$. Fáza generovania regiónov a tranzitívnej funkcie (ktorú stačí vykonať len raz, keďže je nezávislá od textu aj patternu) má teda celkovú časovú zložitosť $\boldsymbol{O}(\boldsymbol{r} * 6^r)$ a fáza vytvorenia suffixového stromu $\boldsymbol{O}(\boldsymbol{n})$, keďže zistenie abecedy textu má zložitosť $\boldsymbol{O}(|\boldsymbol{\Sigma}|)$ a vieme, že $|\boldsymbol{\Sigma}| \leq \boldsymbol{n}$.

Fáza vyhľadávania zahŕňa vytvorenie tabuľky znakových kódov, ktoré trvá $\boldsymbol{O}(|\boldsymbol{\Sigma}|)$. Následne sú v čase $\boldsymbol{O}(|\boldsymbol{\Sigma}| * \boldsymbol{m})$ vygenerované charakteristické vektory a vykonané samotné vyhľadávanie. To pri základnom suffix–tree algoritme (pri zahrnutí optimalizácie z podkapitoly 3.3.3) trvá $\boldsymbol{O}(|\boldsymbol{\Sigma}| * \boldsymbol{m} * \boldsymbol{k})$, keďže však výpočet v hybridnom algoritme prebieha po regiónoch, je konečná zložitosť $\boldsymbol{O}(|\boldsymbol{\Sigma}| * (\boldsymbol{m}/\boldsymbol{r}) * \boldsymbol{k})$.

**Priestorová zložitosť**

Priestorové nároky suffixového stromu sú rovnaké ako v základnom suffix–tree algoritme, teda $O(n)$. Tabuľka charakteristických vektorov potrebuje miesto $O(|\Sigma| * m/r)$ (na každý znak $O(m/r)$ charakteristických vektorov zakódovaných do celého čísla) a tranzitívna tabuľka $O(6^r)$, keďže celkový počet kombinácii regiónu, charakteristického vektoru a carry-in hodnoty je $3^r * 2^r * 3$ a pre každú kombináciu sú v tabuľke uložené len tri skalárne hodnoty (nový index regiónu, carry-in hodnota a súčet diferencií), ktoré zložitosť nezvyšujú. Pre rozumne zvolené $r$ (tj. $r \in O(\log_6(n))$) je teda celková priestorová zložitosť rovná $O(n)$.

# 5 Implementácia

Prvým krokom pred implementáciou algoritmov z kapitoly 3 a novým hybridným algoritmom, popísaným v kapitole 4, bol výber programovacieho jazyka. Hlavnými kritériami pri jeho voľbe boli:

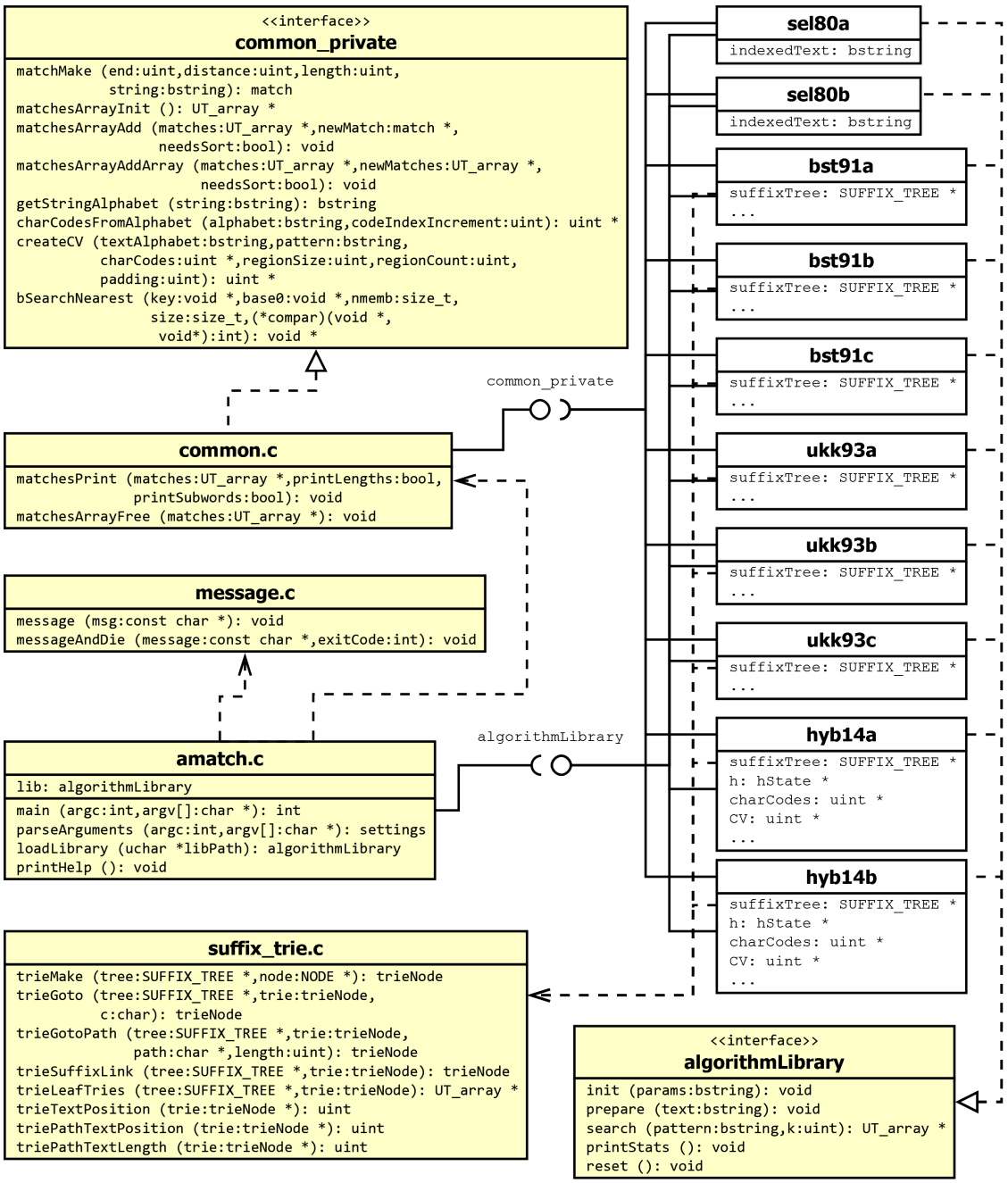
- vysoká rýchlosť;
- malá priestorová náročnosť dátových štruktúr;
- priamy prístup k pointerovej aritmetike;
- prístupnosť knižníc pre prácu s potrebnými abstraktnými dátovými typmi (najmä suffixovým stromom); a
- možnosť jednoduchým spôsobom merať pamäť alokovanú jednotlivými algoritmami.

Najlepšími kandidátmi pre implementáciu sa stali jazyky `C` a `Objective-C`. `Python`, ktorý bol použitý na implementáciu online algoritmov v predchádzajúcej práci autora [21], sa ukázal ako pamäťovo príliš náročný, hoci programovanie v ňom prebieha rýchlo a má natívnu podporu širokej škály abstraktných dátových typov. Naviac ide o interpretovaný jazyk, preto je ním dosahovaná rýchlosť výpočtu príliš nízka na reálne použitie.

Jazyk `Objective-C` sa javil ako ideálna voľba, pretože spája rýchlosť jazyka `C` s podporou pre objektovo orientované programovanie. Pre jeho problematické portovanie na rozličné architektúry (podporu má najmä na systéme `OS X`, na ostatných platformách je slabšia až žiadna) bol však po zvážení všetkých kritérií vybraný jazyk `C` (presnejšie `ANSI C99` bez `C++` nadstavby). Hoci v jazyku `C` chýba možnosť programovať objektovo a natívne poskytuje len základné dátové štruktúry, jeho obrovskou výhodou je, že programátor má absolútnu kontrolu nad akoukoľvek alokáciou údajov. Meranie pamäte použitej algoritmom je vďaka tomu v tomto jazyku úplne jednoduché. Implementácia bola testovaná na 64bitových verziách operačných systémov `Mac OS X 10.9.2 a Ubuntu 12.04.2` s použitím prekladača `gcc` vo verzii `4.6.3` (resp. `Apple LLVM 5.1`).[13]

Program samotný bol rozdelený do viacerých modulov (zdrojových kódov), ktoré si teraz popíšeme. Ich celkový prehľad a vzájomné závislosti zobrazuje diagram na obrázku 5.1.

13 Všeobecne stačí verzia prekladača podporujúca štandard `C99`.




*Obrázok 5.1: Diagram modulov a ich vzájomných závislostí programu amatch. Z dôvodu prehľadnosti nie sú na obrázku znázornené pomocné moduly a knižnice tretích strán (viď podkapitolu 5.2).*

## 5.1 Hlavný modul

Hlavný modul `amatch.c` je vstupným bodom programu, keďže obsahuje po spustení volanú funkciu `main`. Zjednocuje všetky ostatné moduly, obsahuje parser pre kontrolu zadaných parametrov a volá ďalšie potrebné funkcie.

Po zadaní parametrov je najprv zavolaná funkcia `parseArguments`, ktorá overí správnosť všetkých prepínačov a vráti nastavenia programu uložené v zozname. Podľa požadovaného algoritmu, ktorým má byť približné vyhľadávanie uskutočnené, je funkciou `loadLibrary` dynamicky načítaná zdieľaná knižnica z podadresáru `algorithms`. Hlavný modul potom postupne zavolá funkcie `init`, `prepare`, `search` a `reset` implementované v knižnici v rámci rozhrania `algorithmLibrary`. Metódou `prepare` predá hlavný modul algoritmu text pripravený na indexáciu a metódou `search` vyhľadávaný pattern a editačnú vzdialenosť, pričom návratovou hodnotou je zoznam nájdených približných zhôd. Tieto sú potom metódou `matchesPrint` z pomocného modulu `message.c` vypísané na obrazovku.

Ak bol parametrami vyžiadaný aj výpis štatistík, sú za zoznamom zhôd vypísané aj časy jednotlivých fáz algoritmu, informácie o parametroch a ďalšie pre algoritmus špecifické štatistiky, získané volaním funkcie `printStats` z rozhrania `algorithmLibrary`. Príloha A obsahuje návod k použitiu hlavného modulu a tiež popis všetkých jeho parametrov.

## 5.2 Externé moduly a knižnice tretích strán

Jednotlivé algoritmy extenzívne využívajú rozličné dátové štruktúry, ktoré v jazyku `C` nie sú natívne podporované. Z tohoto dôvodu boli pre ich podporu použité viaceré voľne šíriteľné knižnice, ktoré si teraz v krátkosti predstavíme.

### 5.2.1 Knižnica pre prácu so suffixovými stromami

Vzhľadom k tomu, že všetky algoritmy okrem `sel80a/b` využívajú ku svojej funkčnosti suffixové stromy, bolo potrebné pre tento účel nájsť knižnicu, ktorá by umožňovala vytvorenie suffixového stromu z textu v lineárnom čase a zároveň podporovala všetky operácie potrebné k plnej funkčnosti algoritmov (mimo iného podporu pre suffixové odkazy).

Z tohoto dôvodu bola zvolená `ANSI C` implementácia suffixových stromov v knižnici `suffix_tree`, vyvinutej na Ústave výpočetnej techniky Univerzity v Haife pod vedením Shlomo Yonu.[14] Implementácia je založená na Ukkonenovom algoritme pre vytváranie suffixových stromov v lineárnom čase ([25], lepšie vysvetlenie v [19]) a automaticky pri tvorbe stromu vytvára aj suffixové odkazy, čo je pre naše potreby ideálne.

Nakoľko však všetky algoritmy konceptuálne predpokladajú nekomprimovaný suffixový strom, bolo potrebné nad touto knižnicou ešte vytvoriť samostatnú vrstvu zapúzdrujúcu všetky operácie nad komprimovaným stromom tak, aby sa javil ako nekomprimovaný (viď podkapitolu 5.3.3).

14 Knižnica je voľne prístupná na stránke <http://yeda.cs.technion.ac.il/~yona/suffix_tree/>.

### 5.2.2 Knižnica pre prácu s reťazcami

Vzhľadom k tomu, že jazyk `C` poskytuje len veľmi obmedzenú natívnu podporu pre alokáciu a prácu s reťazcami, bola v programe použitá voľne šíriteľná knižnica `bstrlib`.[15] Táto sprehľadňuje a zjednodušuje kód a uľahčuje pokročilé operácie nad reťazcami v jednotlivých algoritmoch aj pri parsovaní argumentov vo funkcii `parseArguments`.

### 5.2.3 Knižnica na podporu hashovacích tabuliek a dynamických polí

Ďalším problémom základného jazyka `C` je chýbajúca podpora dynamických polí a hashovacích tabuliek. Pre ich časté použitie v algoritmoch bola použitá knižnica `uthash`[16] (a jej súčasť `utarray`), ktoré túto podporu pridávajú. „Knižnica“ je v skutočnosti tvorená len niekoľkými hlavičkovými súbormi, ktoré definujú makrá pre všetky potrebné operácie, vďaka čomu je ich vykonanie veľmi efektívne.

## 5.3 Pomocné moduly

Pre zjednodušenie štruktúry hlavného modulu boli viaceré pomocné funkcie presunuté do troch samostatných modulov.

### 5.3.1 Modul pre výpis správ

Veľmi jednoduchý modul `message.c` obsahuje len dve funkcie pre výpis chybových a iných správ na štandardný chybový výstup: `message` a `messageAndDie`. Obe majú ako parameter text správy, ktorú vypíšu, druhá spôsobí navyše ukončenie programu po výpise—preto má ešte ďalší parameter, umožňujúci špecifikovať jeho návratovú hodnotu.

### 5.3.2 Základný modul pre približné vyhľadávanie

Modul `common.c` obsahuje dve funkcie pre výpis zhôd (`matchesPrint`) a uvoľnenie ich zoznamu z pamäte (`matchesArrayFree`). Naviac definuje aj rozhranie `algorithmLibrary`, ktoré musia všetky algoritmy povinne implementovať.

Obsahuje tiež implementáciu rozhrania `common_private`, ktoré definuje základné funkcie využívané všetkými algoritmami. Rozhranie (ani jeho implementácia) nie je normálne viditeľné, koncept chránených (*protected*) funkcií je virtuálne dosiahnutý tým, že je rozhranie deklarované v samostatnom hlavičkovom súbore `common_private.h`, ktorý je direktívou `#include` vložený len do algoritmov, kým do ostatných zdrojových súborov je vložený súbor `common.h`.

Súčasťou tohoto rozhrania sú funkcie na inicializáciu novej zhody (`matchMake`), nového zoznamu zhôd (`matchesArrayInit`) a tiež funkcie zabezpečujúce pridávanie nových zhôd do zoznamu (`matchesArrayAdd`, `matchesArrayAddArray`) tak, aby bol zachovaný kanonický

15 Knižnica je prístupná na stránke <http://bstring.sourceforge.net/>.

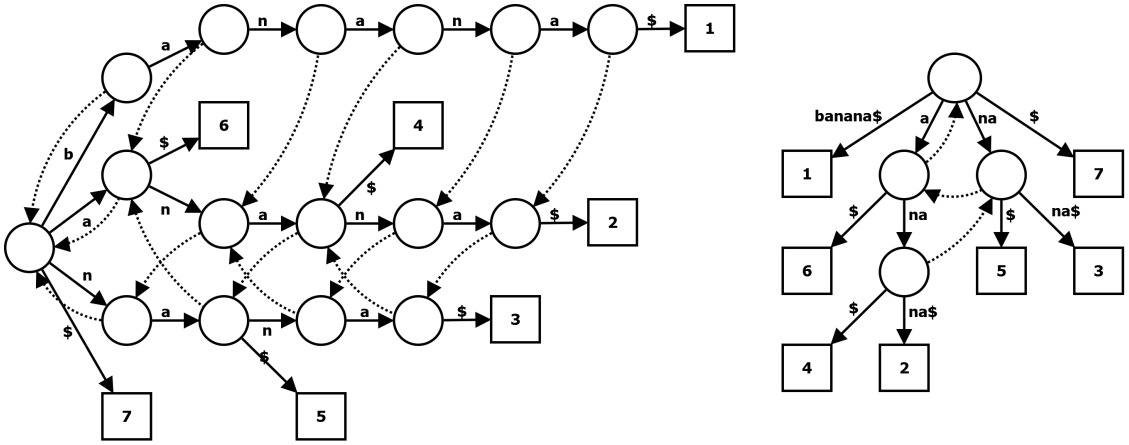
16 Knižnica je na stiahnutie na adrese <http://troydhanson.github.io/uthash/>.

výpis zhôd definovaný v podkapitole 3.3.1. K tomuto účelu bola vo funkcii `bSearchNearest` reimplementovaná štandardizovaná funkcia `bsearch` na binárne vyhľadávanie v poli.

Rozhranie ďalej obsahuje pomocné funkcie na zistenie abecedy v zvolenom reťazci (`getStringAlphabet`) a vytváranie tabuľky znakových kódov (`charCodesFromAlphabet`) a charakteristických vektorov (`createCV`), používaných v pôvodnom algoritme WMM a hybridnom algoritme `hyb14a/b`.

### 5.3.3 Modul na prácu s nekomprimovaným suffixovým stromom

Ako už bolo povedané, knižnica na prácu so suffixovým stromom indexuje text do komprimovaného stromu,[17] no používané algoritmy predpokladajú strom bez komprimácie. Toto spôsobuje komplikácie pri prechode jednotlivými uzlami stromu, zisťovaní ich hĺbky v strome, pozície v texte a najmä pri suffixových prechodov, keďže suffixové odkazy v komprimovanom strome sú vytvorené len pre niektoré uzly jeho nekomprimovaného obrazu, ako to ukazuje obrázok 5.2.




*Obrázok 5.2: Porovnanie suffixových odkazov v nekomprimovanom (vľavo) a komprimovanom (vpravo) suffixovom strome vytvorenom nad textom „`banana`“.*

Pre účely zapuzdrenia skutočnej implementácie suffixového stromu bol preto vytvorený modul `suffix_trie.c`, ktorý vytvára nad knižnicou `suffix_tree` implementačnú vrstvu poskytujúcu všetky potrebné operácie nad nekomprimovaným suffixovým stromom.

Funkcia `trieMake` vytvorí (virtuálny) uzol nekomprimovaného stromu zo zadaného uzlu komprimovaného stromu. Následne je pohyb v nekomprimovanom strome z tohoto uzlu umožnený funkciami `trieGoto` a `trieGotoPath` a funkcia `trieSuffixLink` vykoná suffixový prechod zo zadaného uzlu. Funkcia `trieLeafTries` vráti zoznam všetkých listových uzlov v podstrome a trojica pomocných funkcií (`trieTextPosition`, `triePathTextPosition`

17 Pripomeňme, že nekomprimované stromy majú priestorovú zložitosť $\boldsymbol{O}(\boldsymbol{n}^2)$, preto nie je vhodné s nimi pracovať a prakticky každá knižnica na vytváranie suffixových stromov ich automaticky komprimuje.

a `triePathTextLength`) slúži na získanie pozície daného uzlu v texte, cesty k nemu z koreňového uzlu a dĺžku tejto cesty.

## 5.4 Moduly algoritmov

Všetky moduly implementovaných algoritmov sú umiestnené v podadresári `./algorithms`. Napriek tomu, že v kapitole 3 boli prebrané len dva algoritmy a následne bol v kapitole 4 prezentovaný jeden nový hybridný algoritmus, implementovaných metód je viac. Pre každý algoritmus bolo totiž vytvorených viac verzií programu—tak, aby odpovedali jednotlivým optimalizáciám, ktoré boli uvedené v zodpovedajúcich podkapitolách. Takto bude možné v kapitole 6 otestovať nielen efektívnosť každého algoritmu ako takého, ale aj význam a prínos jeho jednotlivých optimalizácií zvlášť.

Medzi implementované metódy bol zaradený aj pôvodný algoritmus od Sellersa [20], vysvetlený v podkapitole 2.3. Ten bude pri testovaní slúžiť ako referenčný bod, s ktorým budeme môcť ostatné metódy porovnávať.

Implementácia prebiehala väčšinou presne podľa pseudokódov v kapitolách 3 a 4, prípadné menšie zmeny boli vykonané len pre urýchlenie výpočtu. Každý modul algoritmu musí implementovať všetky funkcie špecifikované v rozhraní `algorithmLibrary`, tj. `init`, `prepare` a `search`, slúžiace k vytvoreniu indexu a vyhľadávaniu, pomocnú funkciu `printStats`, ktorá v prípade potreby vypíše podstatné informácie o algoritme a funkciu `reset`, ktorá slúži k uvoľneniu všetkých údajov alokovaných v predchádzajúcich funkciách. Hlavné funkcie sú vždy volané postupne, čím je možné celý algoritmus „odkrokovať“.

*Inicializačná fáza* (funkcia `init`) obsahuje časti algoritmu, ktoré sa môžu vykonať hneď po spustení programu, tj. v dobe, keď ešte nie je známy pattern, text ani editačná vzdialenosť. Hybridný algoritmus `hyb14a/b` je takto schopný vytvoriť si univerzálne štruktúry ešte predtým, než budú vôbec použité.

*Fáza predspracovania* vo funkcii `prepare` je zavolaná vtedy, keď je známy text, v ktorom bude vyhľadávanie prebiehať, ale samotný vyhľadávaný pattern ešte nie. Počas nej prichádza k vytvoreniu indexu textu (v prípade implementovaných algoritmov vo forme suffixového stromu), takže väčšinou aj najdlhšie trvá.

Posledná metóda `search` zahajuje *fázu vyhľadávania* daného patternu s požadovanou editačnou vzdialenosťou v predspracovanom texte. Keďže boli všetky štruktúry vytvorené v predchádzajúcich dvoch fázach, väčšinou už nie sú potrebné žiadne rozsiahle prípravy a vyhľadávanie tak môže začať prakticky okamžite.

### 5.4.1 Sellersov algoritmus

Hoci tento algoritmus nepatrí medzi offline algoritmy (tj. nepredspracováva text), bol z vyššie uvedených dôvodov zaradený do práce. Pred vyhľadávaním si nevytvára žiadne ďalšie štruktúry, preto sa celý algoritmus nachádza v metóde `search`.

Modul je nazvaný **`sel80a.c`**.

### 5.4.2 Základný suffix–tree algoritmus

Základný suffix–tree algoritmus fázu inicializácie nevyužíva a vo fáze predspracovania indexuje textu do suffixového stromu. Implementované boli tri rozličné verzie:

- **`bst91a.c`**: základná verzia algoritmu, implementácia je prakticky totožná s pseudokódom v podkapitole 3.3.2 (Pseudokód 2);
- **`bst91b.c`**: základný algoritmus po aplikovaní optimalizácie na ukončenie výpočtu nevalídnych stĺpcov z podkapitoly 3.3.3;
- **`bst91c.c`**: algoritmus `bst91b.c` s aplikovanou optimalizáciou pre výpočet dĺžky zhody z podkapitoly 3.3.4.

### 5.4.3 Ukkonenov algoritmus

Ukkonenov algoritmus má podobný priebeh ako základný suffix–tree algoritmus, no vo fáze vyhľadávania využíva heuristiku na zamedzenie opakovaného výpočtu dynamických stĺpcov s rovnakým realizovateľným prefixom. Implementované boli tri rozličné moduly:

- **`ukk85a.c`**: základná verzia algoritmu, implementácia prebieha tak, ako to ukazuje Pseudokód 5 v podkapitole 3.4.2;
- **`ukk85b.c`**: optimalizovaná implementácia základnej verzie algoritmu s opravenými chybami z pôvodného článku;
- **`ukk85c.c`**: algoritmus `ukk85b.c` s minimalizovaným počtom výpočtov (optimalizácia z podkapitoly 3.4.3).

### 5.4.4 Hybridný algoritmus

Nový hybridný algoritmus aktívne využíva všetky tri fázy: vytvorenie univerzálnych regiónov prebieha vo funkcii `init` (Pseudokód 13), zistenie abecedy textu vo funkcii `prepare` a charakteristické vektory sú vytvorené vo funkcii `search`. Algoritmus bol implementovaný v dvoch verziách, pričom obe v sebe už zahŕňajú opravu algoritmu WMM pri nedeliteľnosti patternu veľkosťou regiónu (podkapitola 4.2.6) aj optimalizácie pre transformáciu tranzitívnej tabuľky (podkapitola 4.2.7) a minimalizáciu výpočtov (podkapitola 4.2.8).

- **`hyb14a.c`**: základný algoritmus vytvorený kombináciou a úpravou základného suffix–tree algoritmu a algoritmu WMM;
- **`hyb14b.c`**: optimalizovaná verzia algoritmu pre lepšiu heuristiku prechádzania suffixového stromu z podkapitoly 4.3.6.

### 5.4.5 Rozšíriteľnosť a možnosť implementácie nových algoritmov

Preklad programu (viď Príloha A) automaticky podľa prípony detekuje všetky zdrojové kódy algoritmov nachádzajúcich sa v adresári `./algorithms`. V prípade potreby implementovať nové algoritmy ich preto stačí pridať do tohoto adresára a pri preklade budú automaticky detekované a zahrnuté do zoznamu možných algoritmov pre vyhľadávanie.

## 5.5 Skripty a pomocné nástroje na testovanie

Kvôli potrebe intenzívneho testovania všetkých algoritmov bolo nutné okrem samotného programu na približné vyhľadávanie a konkrétnych algoritmov implementovať aj niekoľko pomocných nástrojov.

### 5.5.1 Skript na generovanie náhodných textov

V `Pythone` napísaný skript `strgen.py` slúži na generovanie náhodných textov pre testovanie približného vyhľadávania algoritmov. Umožňuje zvoliť nielen dĺžku generovaného textu, ale aj cieľovú abecedu, a to buď priamym zadaním znakov abecedy ako parametru, alebo výberom jednej z preddefinovaných znakových sád. Podrobné informácie o spôsobe použitia je možné získať zadaním príkazu `python ./strgen.py -help` do terminálu. Potrebný je nainštalovaný interpreter `Python` vo verzii `2.6+`.

### 5.5.2 Testovacia sada skriptov

Pre uľahčenie testovania slúži sada troch testovacích skriptov, ktoré jednoduchým spôsobom umožňujú testovať všetky implementované algoritmy:

- **`generate.sh`**: automaticky vygeneruje použitím skriptu `strgen.py` sadu náhodných textov a patternov v rozlične veľkých abecedách s rozličnými dĺžkami a uloží ich do adresáru `./strings`.
- **`clean.sh`**: zmaže všetky náhodne vygenerované texty a patterny v adresári `strings`.
- **`tests.sh`**: použitím prechádzajúcich dvoch skriptov automaticky zmaže a vytvorí nové náhodné texty a patterny. Následne vykoná sadu vyhľadávaní použitím každého algoritmu s rozličnými editačnými vzdialenosťami a všetky výsledky uloží do adresáru `./tests`.

# 6 Testovanie

Cieľom tejto kapitoly je otestovať výkonnosť a použiteľnosť algoritmov pri rozličných vstupoch a zvolených parametroch, najmä porovnanie efektívnosti nového hybridného algoritmu s už existujúcimi algoritmami a detailné určenie oblastí superiority každého z nich. Kapitola 7 potom bude obsahovať zhrnutie tohoto testovania a zhodnotenie praktického prínosu nového algoritmu. Predtým sa ale v krátkosti zmienime o spôsobe, akým bola overovaná správnosť implementovaných algoritmov.

## 6.1 Verifikácia algoritmov

Hoci boli všetky algoritmy naprogramované podľa pôvodných prác, v ktorých boli uverejnené, pred ich použitím je nutné otestovať správnosť implementácie. Verifikácia každého algoritmu bola vykonaná experimentálne. Boli vybrané viaceré testovacie scenáre, v ktorých sa overila správnosť algoritmu v rozličných situáciách:

- správne detekovanie zhody na začiatku alebo konci textu;
- schopnosť algoritmu nájsť zhody, ktoré sa nachádzajú veľmi blízko pri sebe;
- správne výstupy pri nastavení povolenej vzdialenosti na `0`, tzn. prepnutie programu na *exact string matching*—algoritmus musí detekovať presné zhody (a žiadne iné);
- vyhľadávanie patternu dĺžky `2` s povolenou vzdialenosťou `1`.

Ako najjednoduchší spôsob sa ponúka porovnanie výstupov jednotlivých algoritmov navzájom. Môžeme predpokladať, že v prípade chybne implementovaného algoritmu by sa táto chyba prejavila výstupom odlišným od ostatných algoritmov. V prípade hybridného algoritmu boli samostatne testované aj situácie, kedy bol kvôli nedeliteľnosti patternu veľkosťou regiónu použitý padding (vysvetlený v podkapitole 4.2.6).

Po týchto kontrolách boli nájdené zhody skontrolované aj priamo v texte. Pre tieto účely bol do programu pridaný prepínač `-L`, ktorý zapne vypisovanie podreťazcov textu, ktoré tvoria približnú zhodu s vyhľadávaným patternom (ďalšie informácie o prepínačoch a ovládaní programu obsahuje Príloha A).

## 6.2 Metodika testovania

Pred samotnými testami musíme analyzovať parametre, ktoré najviac vplývajú na dĺžku behu algoritmov. Vo fáze predspracovania to je logicky dĺžka textu, keďže jeho indexácia vo všetkých offline algoritmoch má zložitosť $\boldsymbol{O(n)}$. Medzi základné parametre ovlyvňujúce dĺžku vyhľadávania, ktoré budeme pri testovaní skúmať, potom patria:

- dĺžka patternu $\boldsymbol{m}$;
- povolená editačná vzdialenosť $\boldsymbol{k}$;
- a veľkosť abecedy textu $\boldsymbol{\Sigma}$.

Ako bude z testov nižšie viditeľné, použiteľnosť algoritmov sa v rozličných kombináciách týchto parametrov veľmi líši. Algoritmy pritom budeme hodnotiť najmä z hľadiska:

- počtu navštívených uzlov suffixového stromu;
- času potrebného na vytvorenie regiónov a tranzitívnej tabuľky $h$ pri hybridnom algoritme;
- a rýchlosti následného vyhľadávania.

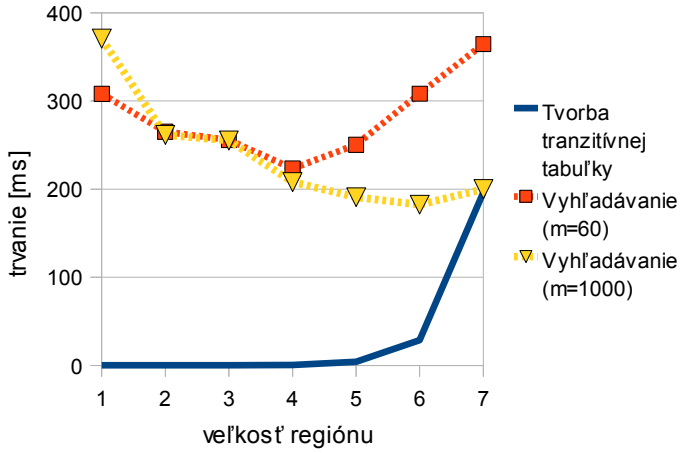
Hoci bola ako pôvodne dôležitý parameter na sledovanie predpokladaná aj celková pamäťová náročnosť algoritmov, pri testovaní sa potvrdilo, že algoritmy majú prakticky totožnú pamäťovú náročnosť určenú veľkosťou suffixového stromu a veľkosť tranzitívnej tabuľky $h$ hybridného algoritmu sa ukázala v porovnaní s ňou nevýznamná. Vyhľadávanie prebiehalo vo všetkých testovaniach nad textom dĺžky milión znakov. Použitím prepínača `-S=10` bolo každé vyhľadávanie automaticky desaťkrát zopakované a program vrátil priemerné časové hodnoty.

## 6.3 Určenie ideálnej veľkosti regiónov

Pred začiatkom testovania bolo potrebné určiť, pri akej veľkosti regiónov $r$ dosahuje hybridný algoritmus `hyb14a/b` najlepšiu rýchlosť vyhľadávania.

Pri testovaní oboch verzií algoritmu sa ukázalo, že vyššia veľkosť regiónov síce urýchľuje výpočet stĺpcov dynamickej matice, no zároveň znepresňuje približné určenie koncu výpočtu pri neplatnosti všetkých regiónov (podkapitola 4.3.6). Tento jav sa najviac prejavuje pri kratších patternoch ($|m| < 60$), kde sa z tohoto dôvodu najvýhodnejšou ukázala veľkosť regiónu `4`. Pri väčších dĺžkach patternu ($|m| > 500$) sa naopak ukázala najvhodnejšia veľkosť `6`.

Ako kompromis bola preto v ďalšom testovaní použitá hodnota `5`, tranzitívna tabuľka algoritmu obsahovala teda `23 328` ($3*6^r$) kombinácií. Pri tejto veľkosti prebiehalo vyhľadávanie hybridným algoritmom pri stredných veľostiach patternu najrýchlejšie—podrobnejšie to zobrazuje graf 6.1, v ktorom sú porovnané časy tvorby tranzitívnej tabuľky $h$ a vyhľadávania v závislosti od veľkosti regiónu.




*Graf 6.1: Časová závislosť hybridného algoritmu od veľkosti regiónu $r$.*

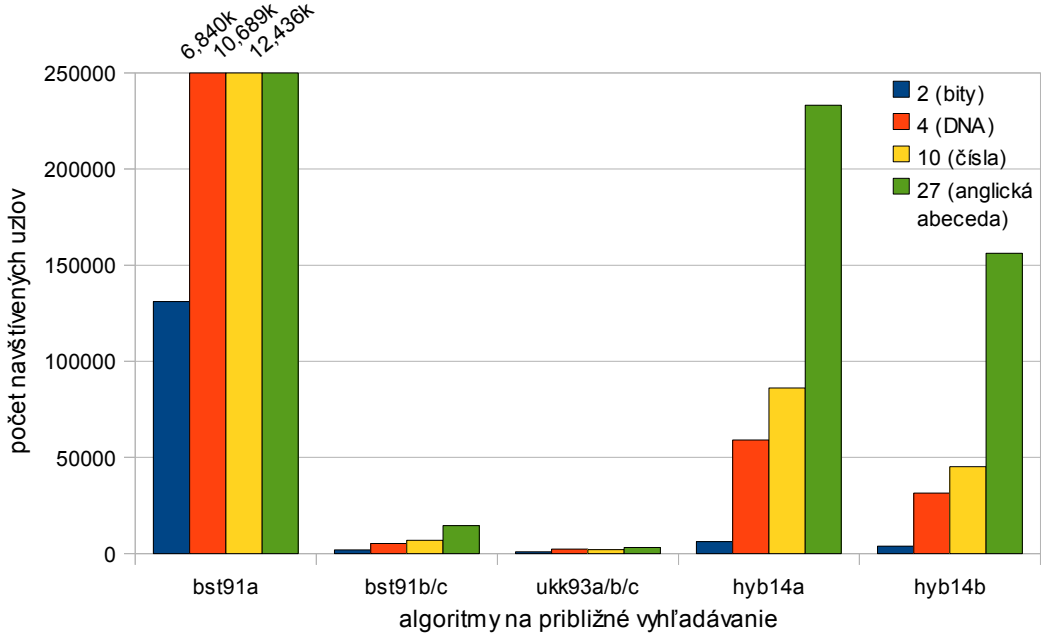
## 6.4 Veľkosť prechádzaného podstromu

Najdôležitejším teoretickým parametrom pri hodnotení výkonnosti testovaných algoritmov je počet uzlov suffixového stromu, ktoré pri vyhľadávaní algoritmus navštívi.

Pre otestovanie závislosti veľkosti prechádzaného stromu od parametrov vyhľadávania boli vykonané tri testy. V testoch nebudeme uvažovať moduly `sel80a/b` (keďže sú to online algoritmy nevytvárajúce suffixový strom). V grafoch spojíme dohromady moduly `bst91b` a `bst91c`, pretože sa odlišujú len pridaním podpory pre výpočet dĺžky zhody z podkapitoly 3.3.4, a tri moduly `ukk93a+b+c`, pretože sa navzájom odlišujú len implementačnou optimalizáciou, resp. minimalizovaným počtom výpočtov (podkapitola 3.4.3).

### 6.4.1 Závislosť od veľkosti abecedy

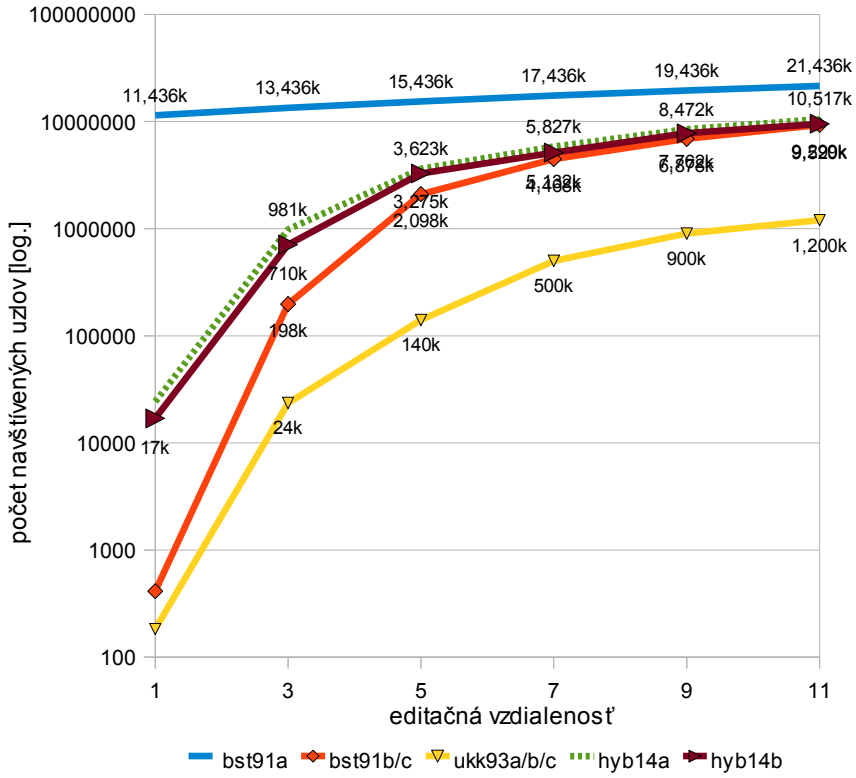
Vplyv veľkosti abecedy na počet navštívených uzlov suffixového stromu zobrazuje graf 6.1. Test bol urobený pre pattern dĺžky `15` a editačnú vzdialenosť `2`. Vidíme, že Ukkonenov algoritmus so svojou pokročilou heuristikou udržiava počet navštívených uzlov výrazne nižšie ako všetky ostatné algoritmy. Náš hybridný algoritmus je lepší ako základná verzia neoptimalizovaného suffix–tree algoritmu, no oproti optimalizovanej verzii navštevuje viac uzlov, čo je spôsobené tým, že koniec výpočtu pri všetkých nevalídnych regiónoch je možné určovať len približne, preto sa navštevujú aj niektoré uzly, ktoré je optimalizovaná verzia základného suffix–tree algoritmu schopná preskočiť.




*Graf 6.2: Závislosť veľkosti prechádzaného podstromu od zvolenej abecedy textu $\Sigma$.*

### 6.4.2 Závislosť od editačnej vzdialenosti

Graf 6.3 zobrazuje veľkosť prechádzaného podstromu (v logaritmickom merítku) v závislosti od editačnej vzdialenosti. Pri teste bol použitý náhodne vygenerovaný text z `27` znakovej anglickej abecedy (`26` písmen + medzera). Z rovnakej abecedy bol vygenerovaný aj pattern dĺžky `15`.



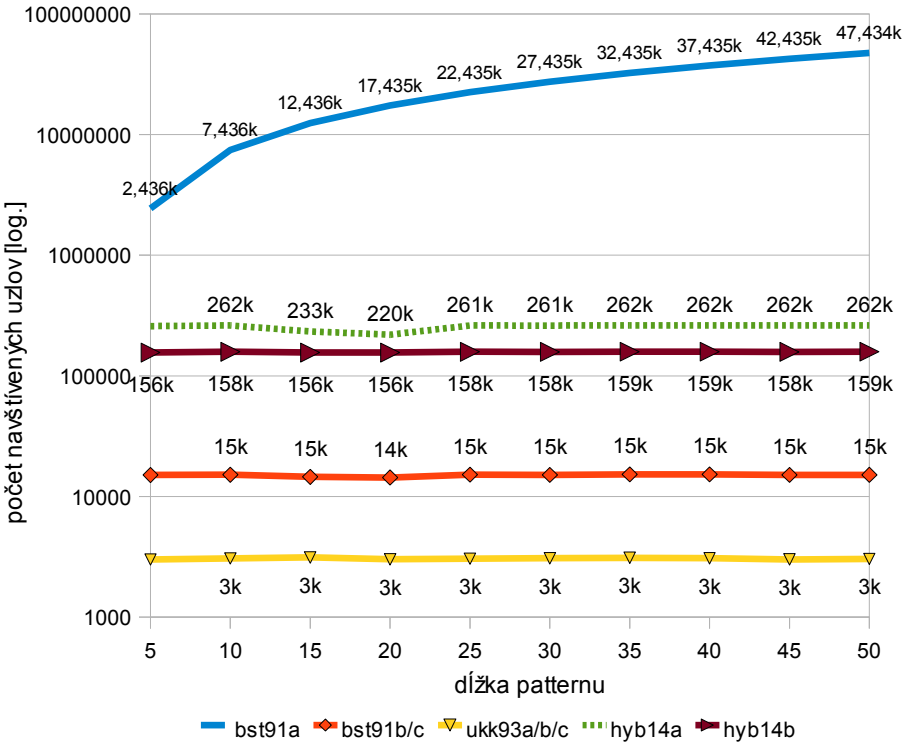

*Graf 6.3: Závislosť veľkosti prechádzaného podstromu od editačnej vzdialenosti $k$.*

Z grafu môžeme vidieť, že počet stavov modulu `bst91a` je takmer konštantný—je to preto, že algoritmus prechádza všetky uzly do danej hĺbky $(m+k)$ bez ohľadu na reálne hodnoty v dynamických stĺpcoch. Všetky ostatné moduly využívajú optimalizáciu prechodov stromom tak, aby neprechádzali uzly so všetkými nevalídnymi hodnotami, preto veľkosť nimi prechádzaného suffixového stromu rastie pozvoľna. Vidíme, že `ukk93a/b/c` dosahuje opäť najmenší prechádzaný podstrom, `hyb14a` a `hyb14b` majú výsledky horšie, no pri vyšších hodnotách $k$ prejdú prakticky rovnaký počet uzlov ako `bst91b/c`. Toto sa opäť dalo očakávať kvôli približnému odhadu nevalídnych regiónov.

### 6.4.3 Závislosť od dĺžky patternu

V grafe 6.4 vidíme závislosť tabuľky prechodov od dĺžky patternu v rozmedzí `5` až `50`, opäť v logaritmickom merítku. Abeceda aj text boli zvolené rovnako ako v predchádzajúcom grafe, editačná vzdialenosť $k$ je `2`. Takmer konštantná veľkosť prechádzaného podstromu (resp. zdanlivá nezávislosť na dĺžke patternu) je spôsobená optimalizáciami výpočtov prítomnými vo všetkých

moduloch okrem `bst91a`, ktoré spôsobujú, že sa prechod stromom zastavuje v závislosti od editačnej vzdialenosti $k$, ktorá je menšia ako dĺžka patternu. Hybridný algoritmus má opäť jednu z najhorších heuristík.




*Graf 6.4: Závislosť veľkosti prechádzaného podstromu od dĺžky patternu $P$.*

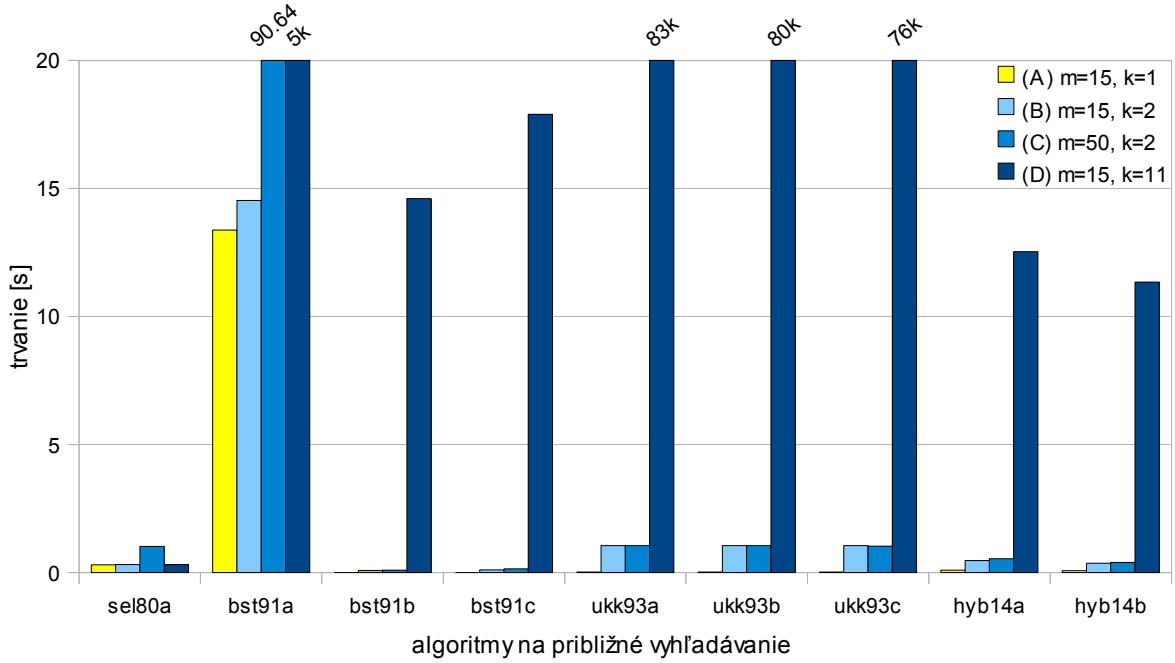
## 6.5 Časové nároky algoritmov

Výsledky testovania v predchádzajúcej kapitole ukazujú, že Ukkonenov algoritmus prechádza najmenšiu a, naopak, náš hybridný algoritmus najväčšiu časť podstromu z takmer všetkých algoritmov. Teoreticky by sme preto mohli očakávať s týmito výsledkami korelujúce časové nároky jednotlivých algoritmov pri vyhľadávaní. Ako však v testoch uvidíme, rozdielna náročnosť operácií vykonávanými jednotlivými algoritmami spôsobuje, že sú časové výsledky odlišné od týchto očakávaní. V testoch bola opäť použitá anglická abeceda, z ktorej bol vygenerovaný náhodný text.

Graf 6.5 zobrazuje časové nároky všetkých algoritmov pre štyri rozlične zvolené hodnoty dĺžky patternu $m$ a vzdialenosti $k$: `15` a `1` (A), `15` a `2` (B), `50` a `2` (C), `15` a `11` (D). Z grafu je na prvý pohľad viditeľné, že základný Sellersov algoritmus je jednoznačne rýchlejší v poslednej situácii a v situácii (B) a (C) sú lepšie len algoritmy `bst91b/c` a `hyb14a/b`. Toto je dané rýchlym nárastom veľkosti prechádzaného stromu v závislosti od editačnej vzdialenosti $k$ (v Ukkonenovom algoritme je tento nárast exponenciálny). Podrobnejšie testovanie ukázalo, že všetky testované suffixové algoritmy sú rýchlejšie ako Sellersov algoritmus pri editačnej vzdialenosti $k=1$ bez ohľadu na dĺžku patternu či abecedu textu, čo len potvrdzuje predošlé tvrdenie.

Hybridný algoritmus `hyb14a/b` dosahuje najlepšie výsledky v poslednej situácii, keď je editačná vzdialenosť vysoká, a v ostatných troch sú dĺžky vyhľadávania porovnateľné s výsledkami Ukkonenovho a základného suffix–tree algoritmu.

Veľmi zlé výsledky Ukkonenovho algoritmu v poslednej situácii sú spôsobené jeho príliš zložitou heuristikou. Keďže sa pri každom navštívenom uzle využívajú suffixové odkazy vedúce až ku koreňovému uzlu a následne musia byť overené všetky uzly, ktorých suffixový odkaz vedie do rovnakého uzlu ako suffixový odkaz uzlu aktuálneho, pri väčšej editačnej vzdialenosti rapídne stúpa počet vykonaných prechodov a algoritmus sa tak stáva prakticky nepoužiteľný už pri relatívne nízkych hodnotách $k$, a to aj napriek najnižšej veľkosti prechádzaného podstromu.




*Graf 6.5: Časové nároky algoritmov pre štyri vybrané situácie: (A) $m=15, k=1$, (B) $m=15, k=2$, (C) $m=50, k=2$ a (D) $m=15, k=11$.*

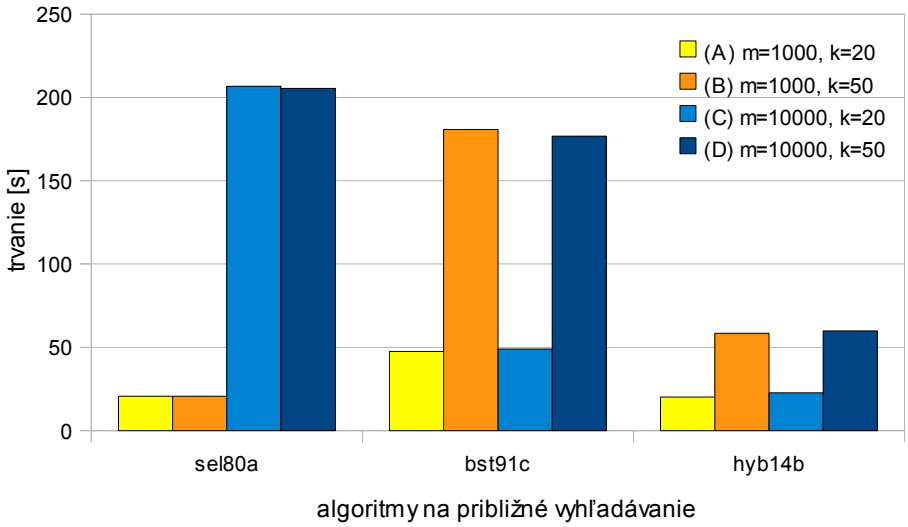
# 6.6 Oblasť superiority hybridného algoritmu

V predchádzajúcej podkapitole sa ukázalo, že hybridný algoritmus sa oproti ostatným offline algoritmom veľmi dobre osvedčil pri vyššej editačnej vzdialenosti. Aj napriek tomu ale stále nebol rýchlejší ako základný Sellersov algoritmus. V tejto podkapitole preto určíme oblasť, v ktorej je nový hybridný algoritmus rýchlosťou najlepší v porovnaní so všetkými ostatnými testovanými algoritmami.

Vzhľadom k tomu, že sa pri vyššej editačnej vzdialenosti hybridný algoritmus osvedčil, bola podrobnejšie otestovaná jeho efektívnosť pri použití veľmi dlhých patternov a vysokých hodnôt $k$. Takéto situácie bežne nastávajú v oblasti výpočtovej biológie pri vyhľadávaní zmutovaných génov, keďže ide o veľmi dlhé sekvencie aminokyselín („znakov“) `A`, `C`, `G` a `T` s vysokou mierou chybovosti

(pomer chybovosti $\alpha$, tj. editačnej vzdialenosti ku dĺžke reťazca, je oveľa väčší, ako napr. pri vyhľadávaní textu s preklepmi). Preto by dobré výsledky hybridného algoritmu v týchto podmienkach boli prakticky vysoko použiteľné.

Graf 6.6 ukazuje časy vyhľadávania pri patternoch dĺžky `1000` a `10000` a editačných vzdialenostiach `20` a `50`. Vidíme, že hybridný algoritmus má oproti základnému suffix–tree algoritmu výrazne menšie časy vo všetkých štyroch prípadoch a základný Sellersov algoritmus jasne prekonáva pri dĺžke patternu `10000`.




*Graf 6.6: Časové nároky algoritmov pri extrémne dlhých patternoch ($|m|$ = `1000` a `10000`) a vysokej chybovosti ($k$ = `20` a `50`).*

Po ďalších testoch sa ukázalo, že oblasť superiority hybridného algoritmu medzi offline algoritmami môžeme určiť na všetky také vyhľadávania, kde je editačná vzdialenosť približne rovná alebo väčšia ako `10` a medzi online algoritmami tam, kde je dĺžka patternu väčšia ako `5000` znakov.

## 6.7 Analýza aplikovateľnosti hybridného prístupu na iné algoritmy a editačné modely

Hybridný algoritmus predstavený v kapitole 4 je postavený na aplikácii princípu blokového výpočtu na algoritmus hľadajúci približné zhody pomocou dynamickej programovacej matice. V prípade nášho hybridného algoritmu ide teda konkrétne o aplikáciu výpočtu po regiónoch na výpočet dynamickej matice prechodom suffixového stromu.

Nakoľko ide o aplikáciu princípu, ponúka sa otázka, či a za akých okolností je takto možné skombinovať blokový výpočet s inými offline algoritmami založenými na suffixových stromoch. Druhou otázkou je tiež to, či je možné využiť iné modely editačných vzdialeností. Pre analýzu

týchto otázok musíme zobrať do úvahy obmedzenia, ktoré v sebe má výpočet pomocou regiónov v takej podobe, ako je použitý v pôvodnom algoritme WMM (podkapitola 4.2).

Prvé obmedzenie, uvedené hneď na začiatku pri definícii diferencií, bolo, že rozdiel dvoch nad sebou sa nachádzajúcich hodnôt musí patriť do rozsahu $\langle -1; 1 \rangle$. Z tohoto predpokladu vyplýva fakt, že nech už použijeme akýkoľvek model editačných vzdialeností, musí mať všetky editačné operácie v rovnakej cene (*unit-cost model*). Táto cena nemusí mať reálne hodnotu 1, no za predpokladu, že je rovnaká u všetkých operácií, ju tak pre potreby generovania regiónov môžeme značiť. Naviac nie je možné použiť Hammingovu ani epizódnu vzdialenosť (viď podkapitolu 2.2.1), pretože v niektorých situáciách je za ich použitia editačná vzdialenosť dvoch reťazcov nekonečná, čo opäť nie je možné v regiónoch vyjadriť a ošetriť.

Druhé obmedzenie implicitne vyvstávajúce pri použití regiónov je nemožnosť priameho výpočtu dĺžky zhody pomocou optimalizácie v podkapitole 3.3.4. Z tohoto dôvodu žiadny algoritmus, ktorý ku svojmu priebehu (resp. heuristike prechádzana suffixového stromu) potrebuje presný údaj o dĺžke realizovateľného prefixu, nie je možné použiť v kombinácii s hybridným prístupom počítania pomocou blokov. Jednotlivé bloky (resp. regióny) totiž pre svoju univerzálnosť nemôžu obsahovať reálne dĺžky realizovateľných prefixov a v prípade ich „postranného“ počítania v priebehu algoritmu by sme zase stratili urýchlenie výpočtu stĺpcov dynamickej matice z $\boldsymbol{O(m)}$ na $\boldsymbol{O(m/r)}$, ktoré sme získali vďaka zavedeniu regiónov, pretože dĺžku realizovateľného prefixu by bolo potrebné počítať pre každú bunku matice $\boldsymbol{L_{i,j}}$.

V prípade, že je ľubovoľný algoritmus a editačný model možné použiť aj za týchto dvoch obmedzení, mal by byť podľa našej analýzy na neho aplikovateľný aj hybridný prístup.

# 7 Záver

## 7.1 Zhodnotenie práce

Cieľom diplomovej práce bolo zoznámenie sa s rozličnými prístupmi k problému približného vyhľadávania v predspracovaných textoch a návrh vlastného algoritmu.

Vzhľadom k zložitosti témy bolo potrebné definovať problém samotný a vysvetliť základný algoritmus založený na dynamickej programovacej matici, keďže z využitia jej vlastností vychádzajú prakticky všetky optimalizácie nachádzajúce sa v ďalších algoritmoch.

Počas štúdia tejto problematiky som prečítal mnoho článkov a príspevkov, z ktorých ma najviac zaujalo využitie suffixových stromov na indexáciu a následné približné vyhľadávanie textu. Taktiež bolo potrebné naštudovať si algoritmy na vytváranie dátových štruktúr, ktoré jednotlivé algoritmy využívajú, keďže postup indexácie textu aj vyhľadávanie samotné sú s nimi úzko späté. Z tohoto dôvodu sa v práci nachádza aj miesto venované zoznámeniu sa s týmito štruktúrami. Väčšina prác z tejto oblasti predpokladá u čitateľa veľmi pokročilú znalosť problematiky, takže bolo ich naštudovanie relatívne náročné. Preto bolo jedným z mojich cieľov aj napísanie práce, pomocou ktorej môže získať všeobecný prehľad aj čitateľ, ktorý sa v tejto oblasti ešte orientuje len okrajovo. Dve práce ([2] a [9]) neboli z dôvodu rozsahu, ktorý by vyžadovalo ich vysvetlenie v kapitole 3, a časovej náročnosti ich následnej implementácie do konečnej práce zahrnuté.

Vybrané dva algoritmy poskytujú spolu s podrobným vysvetlením a pseudokódmi dostatočný základ pre túto oblasť. Keďže implementáciu žiadneho z algoritmov som na internete nevedel nájsť, určite je prínos práce aj v tom, že tieto algoritmy voľne sprístupňuje a doprevádza podrobnými komentármi v samotnej textovej správe. Základný online Sellersov algoritmus [20] je všeobecne známy a s jeho použitím som sa stretol v mnohých prípadoch, kedy by bolo vhodnejšie využiť iný. Táto chyba je podľa mňa spôsobená malým povedomím o tom, že na približné vyhľadávanie existujú aj iné a lepšie algoritmy. Dúfam, že moja práca pomôže prekonať túto informačnú bariéru práve tým, že nepredpokladá u čitateľa hlboké matematické znalosti a podáva dostupnú implementáciu v široko používanom jazyku `C`.

Nový hybridný algoritmus bol v samostatnej kapitole podrobne opísaný, implementovaný spolu s existujúcimi algoritmami a v širokom spektre testov bol s nimi porovnaný. V testoch sa ukázalo, že nový algoritmus je veľmi použiteľný a vo viacerých scenároch prekonal ostatné implementované algoritmy.

V neposlednom rade vidím význam práce aj v tom, že boli dva existujúce algoritmy prakticky otestované, keďže v pôvodných prácach sa nachádzalo len teoretické zhodnotenie ich rýchlosti pomocou asymptotickej časovej zložitosti. Naviac sú testy v tejto práci plne zdokumentované a je možné ich (vďaka skriptom na priloženom CD) kedykoľvek zopakovať.

## 7.2 Závery testovania

Algoritmy boli otestované s použitím širokého spektra hodnôt všetkých dôležitých parametrov. Pomocou testovanie sme získali viaceré hodnotné poznatky, ktoré môžu byť využité v budúcnosti.

Rozličné algoritmy je vhodné využiť v rozličných situáciách. Ukkonenov algoritmus je najrýchlejší pri vyhľadávaní s editačnou vzdialenosťou `1`. Pri ostatných relatívne nízkych vzdialenostiach sa javí lepšie použiteľný základný suffix–tree algoritmus. Vo všeobecnosti sa ukázalo, že urýchlenie vyhľadávania offline algoritmov oproti základnému Sellersovmu algoritmu začína byť skutočne markantné až pri vyhľadávaní veľmi dlhých reťazcov, pričom pri stredných dĺžkach, resp. stredných editačných vzdialenostiach je Sellersov algoritmus rýchlejší ako všetky ostatné algoritmy. Jedným z predpokladaných dôvodom tohoto javu je, že všetky offline algoritmy využívajú veľmi zložité dátové štruktúry, ktoré by bolo potrebné dopodrobna rozanalyzovať a optimalizovať každú ich časť.

Hybridný algoritmus jasne pekonáva ostatné algoritmy pri vyhľadávaní veľmi dlhých patternov a použití veľkých editačných vzdialeností, preto sa výborne hodí v oblasti výpočtovej biológie na vyhľadávanie zmutovaných sekvencií DNA v genómových databázach.

## 7.3 Možné vylepšenia a rozšírenia

Pri písaní práce som narazil na viaceré možné rozšírenia a úpravy, ktoré nemohli byť z časových a rozsahových dôvodov vykonané.

Za najväčšiu výzvu považujem lepšiu analýzu približnej detekcie nevalídnych regiónov v hybridnom algoritme. So zväčšujúcou sa veľkosťou regiónov totiž stúpa nepresnosť tohoto približného určovania, čo spôsobuje prehľadávanie zbytočných uzlov a znižuje tak časovú výhodu, ktorú nám poskytuje výpočet stĺpcov dynamickej matice po regiónoch. S lepšou detekciou by malo byť teoreticky možné dosiahnuť toľkonásobné urýchlenie vyhľadávania, aká ja zvolená veľkosť regiónu.

Ďalším pokračovaním práce by mohlo byť aj naštudovanie ďalších algoritmov na približné vyhľadávanie založených na použití suffixových stromov a analýza (resp. otestovanie) ich kombinovateľnosti s hybridným prístupom, ktorý bol v práci predstavený.

# Literatúra


[1] ARLAZAROV, V.; DINIC, E.; KRONROD, M. a FARADZEV, I. On economic construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*. 1970, vol. 194, no. 11. ISSN 0869-5652.

[2] COBBS, Archie L. Fast approximate matching using suffix trees. *Combinatorial Pattern Matching*. 1995, vol. 937, s. 41-54. ISBN: 978-3-540-60044-2.

[3] CROCHEMORE, Maxime. Transducers and Repetitions. *Theoretical Computer Science*. 1986, vol. 45, s. 63-86. Dostupné z: <http://www-igm.univ-mlv.fr/~mac/Articles-PDF/Cro86tcs-trans-rep.pdf>.

[4] DAMERAU, Fred J. A Technique for Computer Detection and Correction of Spelling Errors. *Communications of the ACM*. March 1964, vol. 7, no. 3, s. 171-176. ISSN 0001-0782.

[5] GHITZA, Alexandru Edgar; WARDA, Philippe-Antoine. Growing A Suffix Tree. INRIA. *Virtual building 8* [online]. 1997 [cit. 2014-01-07]. Dostupné z: <http://pauillac.inria.fr/~quercia/documents-info/Luminy-98/albert/JAVA+html/SuffixTreeGrow.html>.

[6] GUSFIELD, Dan. *Algorithms on Strings, Trees and Sequences : Computer Science and Computational Biology*. Davis (California) : Cambridge University Press, 1997. Inexact Matching, Sequence Alignment, Dynamic Programming, s. 209-391. ISBN 9780521585194.

[7] HAMMING, R. W. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*. April 1950, vol. 26, no. 2, s. 147-160. Dostupný také z WWW: <http://www.lee.eng.uerj.br/~gil/redesII/hamming.pdf>. ISSN 0005-8580.

[8] JOKINEN, Petteri; TARHIO, Jorma; UKKONEN, Esko. A Comparison of Approximate String Matching Algorithms. *Software : Practice & Experience*. December 1996, vol. 26, no. 12, s. 1439-1458. Dostupný také z WWW: <http://www.cs.hut.fi/~tarhio/papers/jtu.pdf>. ISSN 0038-0644.

[9] JOKINEN, Petteri; UKKONEN, Esko. Two algorithms for approximate string matching in static texts. *Mathematical Foundations of Computer Science*. 1991, vol. 520, s. 240-248. Dostupný také z WWW: <http://www.cs.helsinki.fi/u/ukkonen/MFCS91.pdf>. ISBN 978-3-540-54345-9.

[10] LEVENSHTEIN, Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*. 1966, vol. 10, no. 8, s. 707-710.

[11] MASEK, William Joseph; PATERSON, Michael S. A Faster Algorithm Computing String Edit Distances. *Journal of Computer and System Sciences*. February 1980, vol. 20, no. 1, s. 18-31. Dostupný také z WWW: <http://www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/MaPa80.pdf>. ISSN 0022-0000.

[12] MELICHAR, Bořivoj. Approximate String Matching by Finite Automata. In HLAVÁČ, Václav; ŠÁRA, Radim. *CAIP '95 Proceedings of the 6th International Conference on Computer Analysis of Images and Patterns*. [London] : Springer-Verlag, 1995. s. 342-349. Dostupný také z WWW: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.3721&rep=rep1&type=pdf>. ISBN 3-540-60268-2.

[13] MEŠKAUSKAS, Audrius. *Ultrastudio.org* [online]. June 2010, August 2010 [cit. 2014-01-07]. Dynamic programming table. Dostupné z WWW: <http://ultrastudio.org/en/Dynamic_programming_table>.

[14] MEŠKAUSKAS, Audrius. *Ultrastudio.org* [online]. June 2010, January 2011 [cit. 2014-01-07]. Four Russians (algorithm). Dostupné z WWW: <http://ultrastudio.org/en/Four_Russians_(algorithm)>.

[15] NAVARRO, Gonzalo. *Approximate Text Searching*. Santiago (Chile), 1998. xi, 211 s. Dizertačná práca. University of Chile, Department of Computer Science. Dostupné z WWW: <http://www.dcc.uchile.cl/~gnavarro/ps/thesis98.pdf>.

[16] NAVARRO, Gonzalo. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*. 2001, vol. 33, no. 1, s. 31-88. Dostupný také z WWW: <http://www.dcc.uchile.cl/~gnavarro/ps/acmcs01.1.pdf>.

[17] NAVARRO, Gonzalo et al. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*. 2001, vol. 24, no. 4, s. 19-27. Dostupný také z WWW: <http://www.dcc.uchile.cl/~gnavarro/ps/deb01.pdf>.

[18] ПОРОМОВ, С. Construction of suffix automaton. SAINT-PETERSBURG STATE UNIVERSITY. *Discrete Mathematics: Algorithms* [online]. St. Petersburg, Russian Federation, 2009 [cit. 2014-01-07]. Dostupné z: <http://rain.ifmo.ru/cat/view.php/vis/strings/suffix-automaton-build-2009>.

[19] RIDLEY, Nathan et al. Ukkonen's suffix tree algorithm in plain English?. STACK EXCHANGE INC. *Stack Overflow* [online]. [cit. 2014-01-07]. Dostupné z: <http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english>.

[20] SELLERS, Peter H. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*. December 1980, vol. 1, no. 4, s. 359-373. ISSN 0196-6774.

[21] TOTH, Róbert. *Približné vyhľadávanie reťazcov.* Brno, 2011. Dostupné z: <http://www.fit.vutbr.cz/study/DP/BP.php.cs?id=12745&file=t>. Bakalárska práca. FIT VUT v Brně. Vedoucí práce Ing. Jan Kaštil.

[22] UKKONEN, Esko. Algorithms for approximate string matching. *Information and Control*. January/February/March 1985, vol. 64, nos. 1-3, s. 100-118. Dostupný také z WWW: <http://www.cs.helsinki.fi/u/ukkonen/InfCont85.PDF>. ISSN 0019-9958.

[23] UKKONEN, Esko. Finding Approximate Patterns in Strings. *Journal of Algorithms*. March 1985, vol. 6, no. 1, s. 132-137. Dostupný také z WWW: <http://www.cs.helsinki.fi/u/ukkonen/JAlg.pdf>. ISSN 0196-6774.

[24] UKKONEN, Esko. Approximate string-matching over suffix trees. *Combinatorial Pattern Matching.* 1993, vol. 684, s. 228-242. Dostupný také z WWW: <http://www.cs.helsinki.fi/u/ukkonen/cpm931.pdf>. ISBN 978-3-540-56764-6.

[25] UKKONEN, Esko. On–line construction of suffix trees. *Algorithmica.* 1995, vol. 14, no. 3, s. 249-260. Dostupný také z WWW: <http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf>.

[26] UKKONEN, Esko. Suffix tree and suffix array techniques for pattern analysis in strings. UNIVERSITY OF HELSINKI. *Department of Computer Science* [online]. 2005 [cit. 2014-01-07]. Dostupné z: <http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt>.

[27] WU, Sun; MANBER, Udi; MYERS, Gene. A Sub-quadratic Algorithm for Approximate Limited Expression Matching. *Algorithmica*. January 1996, vol. 15, no. 1, s. 50-67. Dostupný také z WWW: <ftp://ftp.cs.arizona.edu/reports/1992/TR92-36.pdf>. ISSN 0178-4617.

[28] Levenshtein distance. In BLACK, Paul et al. Dictionary of Algorithms and Data Structures [online]. Gaithersburg (Maryland) : U.S. National Institute of Standards and Technology, 1999, 14 August 2008 [cit. 2014-01-07]. Dostupné z WWW: <http://xlinux.nist.gov/dads/HTML/Levenshtein.html>.

[29] Levenshtein distance. In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 18 December 2003, last modified on 7 July 2013 [cit. 2014-01-07]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Levenshtein_distance>.

# Zoznam príloh

- Príloha A. Návod na použitie programu
- Príloha B. CD nosič obsahujúci:
  - zdrojové texty programu `amatch` pre približné vyhľadávanie (spolu s externými knižnicami potrebnými k jeho správnemu behu);
  - zdrojový text generátoru náhodných textov `strgen.py`;
  - testovací skript `tests.sh`;
  - licenciu, readme súbor a ostatné textové dokumenty v adresári ./`docs`;
  - sadu testovacích patternov a textov v zložke `./strings`;
  - technickú správu vo formáte `*.pdf` a `*.odt` v adresári `sprava`;
  - súbory, obrázky a diagramy použité v technickej správe

# Príloha A Návod na použitie programu

Pred použitím programu je potrebné preložiť zdrojové súbory pomocou príkazu `make`. Na cieľovom počítači musí byť prítomný prekladač `gcc` vo verzii podporujúcej štandard jazyka `C99`. Testovanie prebehlo na 64bitových verziách operačných systémov `Mac OS X 10.9.2` a `Ubuntu 12.04.2`.

Program sa spúšťa zadaním príkazu `./amatch <parametre>` v príkazovom riadku. Parametre definované pre aplikáciu sú:

```
amatch -h
amatch -a=<algoritmus>  -p=<pattern>  -t=<text>  [-k=<vzdialenosť>]
       [-(l|L)]  [-r=<veľkosť_regiónu>]  [-(s|S=<počet>)]  [-q]
```

Nasleduje popis jednotlivých parametrov:

| | |
|---|---|
| `-h` | Vypíše nápovedu k programu |
| `-a=?` | Vyberie metódu, ktorou sa bude vyhľadávať pattern v texte |
| `-p=?` | Cesta k súboru s patternom |
| `-t=?` | Cesta k súboru s textom |
| `-k=?` | Povolená editačná vzdialenosť, defaultne 2 |
| `-(l\|L)` | Zapne výpis dĺžok zhôd. Prepínač `L` zapne aj výpis samotných zhôd v texte a zmení výpis na oddelený zalomeniami riadku namiesto medzier. *Poznámka:* nie všetky algoritmy podporujú výpis dĺžok a zhôd. V takom prípade budú mať vypísané dĺžky hodnotu `0` a nájdené zhody budú prázdne. |
| `-r=?` | Vyberie veľkosť regiónu (potrebná pri metódach `hyb14a/b`) |
| `-(s\|S=<počet>)` | Zapne výpis štatistík. Ak je použitý prepínač `S`, `<počet>` je povinný. V tom prípade bude vyhľadávanie vykonané `<počet>`krát a v štatistikách budú vypísané priemerné časové hodnoty. |
| `-q` | Zapne tichý režim (nájdené zhody sa nebudú vypisovať) |

Program po spustení so správnymi parametrami začne vyhľadávať zvolený pattern v texte a všetky približné zhody vypíše (medzerami oddelené) na obrazovku vo formáte `pozícia:vzdialenosť`.