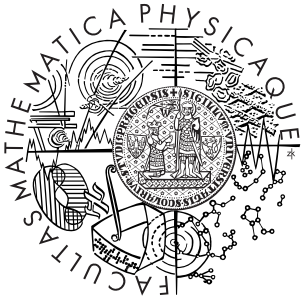
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Ján Host

Algoritmy konstrukce sufixových stromů

Kabinet software a výuky informatiky

Vedúci bakalárskej práce: Mgr. Martin Senft
Študijný program: Informatika

2007

Poďakovania: V prvom rade by som chcel poďakovať vedúcemu bakalárskej práce Mgr. Martinovi Senftovi za jeho skúsené rady a trpezlivosť, bez ktorých by táto práca nevznikla, ale aj za zapožičanie diplomovej práce vrátane softwaru, ktoré mi pomohli pri voľbe vhodnej formy písania práce a spôsobu programovania priloženého programu. Ďalej ďakujem RNDr. Tomášovi Dvořákovi, CSc za jeho prednášky v zimnom semestri, ktoré rozšírili v danej oblasti môj obzor a napomohli určiť si, čo všetko je pre moju prácu potrebné, aby sa dosiahla jej patričná objektivita pri porovnávaní algoritmov.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dne 31.5.2007 Ján Host

Obsah

Názov práce: Algoritmy konštrukcie sufixových stromov
Autor: Ján Host
Katedra (ústav): Kabinet software a výuky informatiky
Vedúci bakalárskej práce: Mgr. Martin Senft
e-mail vedúceho: Martin.Senft@mff.cuni.cz.

Abstrakt: V predloženej práci študujeme algoritmy pre výstavbu dátovej štruktúry sufixový strom. Táto štruktúra pomáha optimálne riešiť mnoho problémov z oblasti spracovávania textu. Existuje hneď niekoľko prístupov a algoritmov, ako ju postaviť. Cieľom je vytvoriť taxonómiu algoritmov konštruujúcich sufixové stromy a poskytnúť tak prehľad v ich vlastnostiach a použiteľnosti. Algoritmy dôležité z rôznych hľadísk sú prezentované, je vysvetlený princíp a myšlienka ich práce. Nasledovne sa porovnajú v testoch a odhalia sa ich prínosy a úskalia. Z výsledkov testov, ale aj teoretických znalostí a znalostí z naštudovanej literatúry, sú zistené spoločné znaky algoritmov a po predložení vhodných kritérií je vytvorená ich taxonómia.

Kľúčové slová: dátová štruktúra, sufixový strom, algoritmus, konštrukcia, porovnanie

Title: Suffix Trees Construction Algorithms
Author: Ján Host
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Martin Senft
Supervisor's e-mail address: Martin.Senft@mff.cuni.cz.

Abstract: In the present work we study suffix tree construction algorithms. This structure helps solving a variety of text problems in optimal time. There are several approaches and algorithms for building a suffix tree. The goal is to create a taxonomy of these algorithms and provide an overview of their properties and suitable usage. The algorithms important from various points of view are presented. Fundamentals and main ideas of their work are explained. Consequently, they are compared in tests to reveal their main flaws and strength. According to the test results as well as to theoretical knowledge and to information gained from previous studies common features are determined and after presenting the appropriate criteria a taxonomy is created.

Keywords: data structure, suffix tree, algorithm, construction, comparison

Kapitola 1

Úvod

1.1 Motivácia a ciele

V dnešnej dobe sa spracovávanie textu stáva čoraz dôležitejším problémom. S vyvíjajúcim sa hardwarovým výkonom, sa otvárajú možnosti, ako počítačovo pohodlne spracovávať nielen text ako taký, ale aj obrovské kvantá informácií uložených napr. v ľudskom genóme. Suffixový strom je mocná dátová štruktúra, ktorej využitie je pri spracovaní textových reťazcov veľmi široké a poskytuje efektívne riešenia mnohých problémov týkajúcich sa spracovávania textu. Jej podstatou je, že indexuje všetky prípony v reťazci. Sufixový strom umožňuje v ním indexovanom reťazci určiť výskyt akéhokoľvek podreťazca v čase úmernom dĺžke hľadaného podreťazca, nezávisle na dĺžke indexovaného reťazca. Napriek tomuto veľkému prínosu, ktorý vybudovanie takejto štruktúry poskytuje, sa až spočiatku venovala len malá pozornosť tomu, aby sa sufixové stromy skúmali podrobnejšie a hľadali sa efektívnejšie algoritmy pre ich výstavbu. Bolo to zapríčinené najmä tým, že veľkosť indexovacej štruktúry bola pre väčší vstup neúnosná. S rozvojom hardwarových možností sa ale tento problém eliminoval, čoho dôsledkom bol postupne rastúci záujem o skúmanie sufixového stromu. Teraz existuje čoraz viac algoritmov, ktoré sú reálne použiteľné v širokom spektre problémov textového spracovania a o štruktúre sa už vie pomerne veľa. Každý prístup ale so svojimi prínosmi nesie aj nejaké negatívne dôsledky, napr. pamäťová vs. časová efektivita, alebo strata on-line vlastnosti pri prechode z Ukkonenovho na McCreightov algoritmus [2].

V našej práci sa zameriame práve na skúmanie rozdielov týchto algoritmov a pokúsime sa vytvoriť taxonómiu, ktorá rozdelí algoritmy nielen podľa

ich teoretických vlastností, ale taktiež aj podľa ich praktickej využiteľnosti v konkrétnych množinách problémov.

Aj napriek zvýšenému záujmu o štúdium sufixového stromu existujú problémy, pri ktorých by sa síce vybudovanie sufixového stromu zišlo, ale žiaden algoritmus nedokáže štruktúru postaviť podľa požiadaviek problému. Často ale netreba indexovať všetky prípony, ale iba niektoré, napr. s konkrétnou rovnakou predponou. Ak si napríklad za povolenú predponu zvolíme oddeľovač slov v ľudsky čitateľnom texte, dostaneme prípony, ktoré začínajú na hranici slov. Pri indexácii čitateľného textu je to celkom logická požiadavka a nepredstavuje veľké obmedzenie. Takto vznikajú ochudobnené štruktúry, ktoré však postačujú na riešenie nejakej, spravidla malej, množiny úloh. Sú to napríklad riedke sufixové stromy (indexujúce iba niektoré prípony), sufixové polia, DAWG, CDAWG. V [4] autori ukazujú ďalšie možnosti aj so spojenými nevýhodami.

1.2 Štruktúra práce

Po úvode do problematiky poskytneme presné definície používaných pojmov, ako zásadných (sufixové stromy, trie, ...), tak aj pomocných (grafy, reťazce, prípony...). Následne ukážeme a vysvetlíme spôsob, akým štruktúru budujú niektoré algoritmy a naznačíme, v čom by mali spočívať ich výhody a nevýhody.

Poskytneme stručný popis aplikácie, ktorú sme vytvorili. Tá na priloženej sade testovacích dát poskytne pamäťové a časové nároky implementovaných algoritmov. Tieto výsledky uvedieme v tabuľkách.

V ďalšej kapitole tieto výsledky zhrnieme a vytvoríme taxonómiu algoritmov zohľadňujúcu teoretické vlastnosti, výsledky testov, ale aj očakávania iných autorov. V závere tieto poznatky zhrnieme.

Kapitola 2

Terminológia a značenie

2.1 Základné definície

Nech Σ je konečná neprázdna množina veľkosti k s definovaným usporiadaním. Nazveme ju *abeceda*. Jej prvkom budeme hovoriť *znaky*. *Reťazcom* nazveme postupnosť *písmen* $t_1, t_2, \cdots, t_n$ (píšeme skrátene $t_1 t_2 \dots t_n$). Prázdna postupnosť písmen je *prázdny reťazec*, ktorý označujeme λ. *Zreťazením reťazcov* $u = u_1 u_2 \dots u_k$ a $v = v_1 v_2 \dots v_j$ je reťazec $w = uv = u_1 \dots u_k v_1 \dots v_j$. V tomto prípade hovoríme, že k je *dĺžka reťazca* u a značíme $|u| = k$. Pre *reťazec* $t = uvw$ (u, v, w môžu byť prázdne) hovoríme, že *reťazec* u je predpona, v je *podreťazec* a w je *prípona* t. Σ^* je množina všetkých *reťazcov* nad Σ. t_i budeme označovať $i$tý znak *reťazca* t. Na označenie množiny všetkých neprázdnych *reťazcov* $\Sigma^* \setminus \lambda$ použijeme Σ^+. Predpokladáme, že t je *reťazec* nad Σ dĺžky $n \geq 1$. Potom pre každé $i \in [1, n]$ bude $s_i = t_i \dots t_n$ označovať $i$tú neprázdnu príponu t.

Graf G je usporiadaná trojica (V, Σ, E), kde V je konečná množina, Σ je *abeceda*, a E je množina usporiadaných trojíc (u, t, v), kde $u, v \in E$ a $u \neq v$ a $t \in \Sigma^+$. Prvky množín V, E, Σ nazveme postupne *vrcholy*, *hrany* a *vstupná abeceda*. U hrán nazveme reťazec t *označením hrany*. Hrany (u, t, v) budeme ďalej značiť $u \xrightarrow{t} v$. Ako *a-hranu* z vrcholu u do vrcholu v označíme hranu v tvare $u \xrightarrow{aw} v$, kde $a \in \Sigma$ a $w \in \Sigma^*$. Hrany vedúce do (z) vrcholu u sú hrany $v \xrightarrow{t} u \in E$ (u $\xrightarrow{t} v \in E$). *Podgraf* H grafu G je graf (U, Π, F), kde $U \subseteq V, \Pi \subseteq \Sigma$ a $F \subseteq E$. *Cesta z* u_1 *do* v_n v grafe $G = (V, \Sigma, V)$ je postupnosť hrán $u_1 \xrightarrow{t_1} v_1, \dots, u_n \xrightarrow{t_n} v_n$, kde $(\forall i \in \{1, \dots, n-1\})(v_i = u_{i+1})$ a $(\forall i, j \in \{1, \dots, n\})((u_i = v_j) \Rightarrow (i = j + 1) \vee ((i = 1) \wedge (j = n)))$. *Reťazec na ceste* je reťazec, ktorý vznikne postupným zreťazením označení

hrán tvoriacich cestu od prvej po poslednú. Cestu z u_i do u_i pre nejaké i nazveme *kružnica*. Povieme, že v grafe existuje *cyklus*, ak existuje množina vrcholov $\{u_1, \ldots, u_n\}$ taká, že $(\forall i \in \{1, \ldots, n-1\})(\exists s \in \Sigma^+)((u_i \xrightarrow{s} u_{i+1} \in E) \vee (u_{i+1} \xrightarrow{s} u_i \in E))$ a $(\exists t \in \Sigma^+)((u_n \xrightarrow{t} u_1 \in E) \vee (u_1 \xrightarrow{t} u_n \in E))$.

Strom je hranovo maximálny graf bez cyklov. Maximálny znamená, že akákoľvek ďalšia hrana pridaná do množiny hrán by zapríčinila vznik cyklu. *Vetva pod vrcholom* v je podgraf, ktorý obsahuje všetky vrcholy a hrany na všetkých cestách z v. Vrchol, ktorého vetvou je celý graf, nazveme *koreň stromu*. *Synovia* vrcholu u sú také vrcholy v, do ktorých vedie hrana z u. Vrchol u je potom *otcom* pre každý takýto vrchol v. *List* je taký vrchol stromu, ktorý nemá žiadnych synov. *Uzol* naopak má aspoň jedného syna. Strom je *zakorenený*, ak je jeden jeho uzol *koreňom*.

2.2 Definície sufixových štruktúr

Σ^+*-strom* T označíme konečný zakorenený strom s hranami označenými reťazcami z Σ^+ taký, že pre každé $a \in \Sigma^+$ každý vrchol u stromu T má najviac jednu a-hranu $u \xrightarrow{aw} v$ pre nejaký reťazec w a vrchol v. Hranu vedúcu do listu stromu nazveme *listová hrana*. Nech u je vrchol v T. Tento vrchol označíme $\overline{w}$ práve vtedy, ak w je reťazec na ceste z *koreňa* do u. *Koreň* teda môžeme označiť $\overline{\lambda}$. *Reťazec* s sa *vyskytuje* v T práve vtedy, ak T obsahuje vrchol $\overline{sv}$, pre nejaký reťazec v. O *výskyte* reťazca povieme, že je *explicitný*, ak $|v| = 0$, inak je *implicitný*. *Kanonická pozícia výskytu* $t = sv$ v T je vrchol $\overline{s}$, taký, že $|s|$ je maximálna. Vrchol Σ^+*-stromu* nazveme *vetviaci* práve vtedy, ak z neho vychádzajú aspoň dve hrany. *Suffixový strom* pre t, označený $ST(t)$, je Σ^+*-strom* T s nasledujúcimi vlastnosťami: (i) každý vrchol je buď list alebo *vetviaci vrchol* a (ii) reťazec w sa *(implicitne) vyskytuje* v T práve vtedy, ak w je *podreťazcom* t. Pre každý list $\overline{s_j}$ definujeme $l(\overline{s_j}) = \{j\}$. Pre každý *vetviaci* vrchol $\overline{u}$ definujeme $l(\overline{u}) = \bigcup l(\overline{uv})$, kde $\overline{uv}$ sú synovia $\overline{u}$. *Sufixový trie* pre *reťazec* t značíme $STrie(t)$ a definujeme ako Σ^+ *strom* taký, že reťazec w sa *explicitne vyskytuje* v t práve vtedy, ak w je *podreťazcom* t. Oproti *sufixovému stromu* pre ten istý *reťazec* sme vynechali podmienku (i), ale posilnili podmienku (ii) na *explicitný výskyt*.

Kapitola 3

Porovnanie algoritmov

3.1 Vysvetlenie porovnávaných algoritmov

V tejto sekcii teoreticky prezentujeme a vysvetlíme algoritmy, ktoré sme sa rozhodli implementovať a porovnať.

3.1.1 Ukkonenov algoritmus

Ukkonenov algoritmus reprezentantom algoritmov stavby sufixového stromu v lineárnom čase. Jeho podstatnou vlastnosťou je jeho on-line vlastnosť. Z teoretického a edukatívneho hľadiska je dôležitá aj jeho názornosť. Princíp výstavby stromu $ST(t)$ spočíva v zmene $ST(p_i)$ na $ST(p_{i+1})$, kde p_i $i$tá predpona reťazca t. Túto aktualizáciu predstavuje procedúra *update*.

Pri prechode z $ST(p_i)$ na $ST(p_{i+1})$ sa každá prípona obsiahnutá v pôvodnom strome zväčší o jeden znak. Takto by sa pri každej aktualizácii musel prechádzať každý list a upraviť označenie hrany do neho vedúcej, naviac pribudnú nové vrcholy. Tomu sa Ukkonen snaží vyhnúť, pretože by sa takto okamžite stratila možnosť dosiahnutia lineárneho času vzhľadom k veľkosti vstupu. Ukkonenov trik spočíva v zavedení niekoľkých nových pojmov, ktorých využitím odpadá mnoho práce s listovými hranami. Pojem *otvorená hrana* označuje hranu, ktorej označenie má konkrétny začiatok, avšak koniec je v nekonečne, resp. v poslednom znaku p_i. Pri prechode na $ST(p_{i+1})$ teda označenie listovej hrany môže ostať nezmenené, avšak implicitne sa hrana predĺži. Takto môžeme predísť zbytočnému prechádzaniu celej štruktúry. Ďalšie pojmy, ktoré Ukkonen zavádza sú *aktívny* bod a *konečný bod*. Tieto predstavujú prvý a posledný vrchol, ktorý musí pri aktu-

alizácii byť skúmaný. Z tvrdení v [11] vyplýva, že tieto vrcholy sú prítomné a jedinečné pre každý strom $ST(p_i)$ a nasledujúci spracovávaný znak t_{i+1}. Naviac z jedného jednoduchého pozorovania vyplýva, že počet vrcholov medzi aktívnym a konečným bodom je pri výstavbe celého stromu $ST(t)$ v každom medzikroku $ST(p_i)$ aspoň amortizovane konštantný, teda linearitu algoritmu môžeme stále dosiahnuť. Tzv. sufixové hrany spájajú cestu po týchto vrcholoch. Na nájdenie aktívneho bodu v konkrétnom kroku je síce potrebný až čas, ktorý môžeme zhora odhadnúť najlepšie tak, že je menší než n, avšak počas celého behu algoritmu, ako Ukkonen dokazuje, zaberá čas potrebný na nájdenie aktívneho bodu vo všetkých krokoch spolu tiež čas úmerný $\mathcal{O}(n)$, teda na jedno nájdenie pripadá aspoň amortizovane konštantný čas.

Princíp práce procedúry *update* spočíva v tom, že sa v existujúcom strome pridajú ďalšie hrany a vrcholy tam, kde oproti predošlej situácii došlo k rozdeleniu nejakej hrany. Znamená to, že ak v predošlej situácii bola v strome hrana $u \overset{t_p \ldots t_q}{\rightarrow} w$, potom v nasledujúcom kroku už hranou stromu nie je a namiesto nej sa objavia hrany $u \overset{t_p \ldots t_i}{\rightarrow} v$, $v \overset{t_{i+1} \ldots t_q}{\rightarrow} w$ a $v \overset{t_x \ldots t_y}{\rightarrow} r$, kde v a r sú nové vrcholy, $p < i < q$ a označenie tretej hrany vysvetlíme neskôr. To sa vďaka jednoduchému triku, dá dosiahnuť v aspoň amortizovane konštantnom čase[11].

Je zrejmé, že pri aplikovaní procedúry *update* postupne na všetky znaky $t_1, t_2, \cdots t_k$, kde $k = |t|$, dostávame po poslednom kroku $ST(t)$. Procedúru update bližšie ukážeme v nasledujúcom algoritme, pre detailný popis stavby stromu Ukkonenovým algoritmom viď [11].

V uvedenom algoritme 1 sa používajú procedúry *test-and-split* a *canonize*. *canonize* dostáva ako argument vrchol stromu $\overline{u}$ a dvojicu ukazovateľov (k, l) do vstupného reťazca, ktoré predstavujú podreťazec t taký, že $ut_k \cdots t_l$ (tiež podreťazec t) sa vyskytuje v $ST(p_i)$. Vracia kanonickú pozíciu pre výskyt reťazca $ut_k \cdots t_l$. *test-and-split* ako vstupné argumenty berie kanonickú pozíciu pre výskyt reťazca a znak c. Vráti informáciu o tom, či bol dosiahnutý konečný bod. Ak dosiahnutý nebol, rozdelí c-hranu a vytvorí nový vrchol. *root* je koreň stromu $ST(t)$. $x.suf$ značí vrchol na ktorý ukazuje *sufixová* hrana z x. Vstupom pre procedúru update je dvojica $(s, (k, i))$, taká, že $(\overline{s}, t_k \ldots t_{i-1})$ je kanonická pozícia aktívneho bodu.

Algorithm 1 procedure $update(s, (k, i))$

1: $oldr = root$;
2: $(endpoint, r) = test\text{-}and\text{-}split(s, (k, i-1), t_i)$;
3: **while** not $endpoint$ **do**
4: vytvor novú hranu $r \overset{t_i, t_\infty}{\rightarrow} r'$, kde r' je nový vrchol;
5: **if** $oldr \neq root$ **then**
6: vytvor novú sufixovú hranu $oldr \rightarrow r'$;
7: **end if**
8: oldr = r;
9: $(s, k) = canonize(s.suf, (k, i-1))$;
10: $(endpoint, r) = test\text{-}and\text{-}split(s, (k, i-1), t_i)$;
11: **end while**
12: **if** $oldr \neq root$ **then**
13: vytvor novú sufixovú hranu $oldr \rightarrow s$;
14: **end if**
15: **return** (s, k)

3.1.2 McCreightov algoritmus

McCreightov algoritmus má základnú myšlienku úplne odlišnú od Ukkonenovho algoritmu. Výsledný algoritmus ale vyzerá veľmi podobne a v práci [2] je dokonca ukážka, ako algoritmy medzi sebou prevádzať. McCreight vychádza z toho, že postupne pridáva do štruktúry prípony od najdlhšej po najkratšiu. Už tu môže čitateľ tušiť, že sa tým stratí on-line vlastnosť, ktorou disponuje Ukkonenov algoritmus.

Keďže pre beh algoritmu je podstatných vždy iba prvých niekoľko znakov skúmanej prípony, prebieha skutočne tak, že číta vstup postupne po niekoľkých znakoch od začiatku a upravuje výslednú štruktúru. Po ceste však v každom medzikroku nevzniká nutne sufixový strom, až v poslednom kroku. McCreightov algoritmus sa dá chápať ako úprava Ukkonenovho. Pridaním novej procedúry sa dosiahne to, že algoritmus potrebuje menej iterácií v hlavnom cykle, avšak celkovo je tetno algoritmus len preskupením krokov Ukkonenovho algoritmu. V predchádzajúcom algoritme mohla nastať situácia, kedy procedúra update nevytvorila žiadne nové hrany ani vrcholy v štruktúre, iba sa implicitne predĺžili všetky listové hrany. Tieto kroky McCreightov algoritmus eliminuje procedúrou *scan*.

Pri implementácii tohto algoritmu sme použili práve článok [2]. Uvedený neprocedurálne zapísaný algoritmus sme prepísali do procedurálnej verzie.

Procedurálnu verziu procedúry *scan* ako aj samotného algoritmu uvádzame (algoritmy 2 a 3). Procedúra *scan* má tieto vstupné argumenty: vrchol b, v ktorom sa prehľadávanie má začať a číslo i, ktoré predstavuje pozíciu písmena vo vstupnom reťazci, od ktorej sa začne prehľadávanie. Jej výstupom je trojica (u, t, j), kde u je vrchol v ktorom sa prehľadávanie zastavilo, t je časť označenia hrany, ktorá sa ešte zhoduje a j je pozícia vo vstupnom reťazci, kde sa prehľadávanie zastavilo. V podstate *scan* funguje tak, že prehľadáva strom od vrcholu zo vstupu a zisťuje, či sa z neho dá ďalej pohnúť tak, aby sa označenie hrán zhodovalo so vstupným reťazcom. Výstupy predstavujú situáciu pri prvej nezhode: posledný vrchol pred prvou nezhodou, zhodná časť označenia hrany, pozícia prvej nezhody.

Procedúra *mccbuild* má jediný parameter a tým je vstupný reťazec. Jej výstupom je strom $ST(t)$. Začína sa s prázdnou štruktúrou a každý príkaz *pridaj novú hranu* pridáva hranu prípadne vrcholy do tejto štruktúry. Procedúra používa pomocné procedúry *canonize*, *link*, *split* a *scan*. Čo robí *scan* sme si už vysvetlili. *canonize* funguje presne ako u Ukkonenovho algoritmu, s tým rozdielom, že ako vstup berie pre jednoduchosť v tomto prípade namiesto dvoch ukazovateľov reprezentujúcich podreťazec priamo podreťazec. Procedúra *link* vráti kanonizovanú pozíciu vrcholu, ktorý má byť spracovaný ako ďalší v poradí. *split* berie ako vstup vrchol a časť označenia hrany, ktorú rozdelí podobne ako *test-and-split*. Výstupom je nový vrchol, ktorý vznikol pri delení. Ich presnejší a podrobný popis sa nachádza v [2].

3.1.3 Algoritmus WOTD

Tento algoritmus (*Write-Only Top-Down*) je reprezentantom kvadratických algoritmov pre výstavbu sufixového stromu. Skonštruovaný bol však tak, aby procesorová cache bola využitá čo najoptimálnejšie, teda aby čo najmenej dochádzalo ku tzv. cache-missom, kedy požadovaná informácia sa v cache nenachádza a musí sa pre ňu pristupovať do pamäte RAM. Teoreticky síce algoritmus pracuje v asymptoticky horšom čase oproti lineárnym algoritmom, ale v praxi sa častorkát ukázal vďaka jeho optimalizovanosti na súčasnú výpočtovú techniku ako výkonnejší.

Existuje jeho veľa variácií, ktoré podobné optimalizácie na jeho základe využívajú pre efektívnu prácu s virtuálnou pamäťou na disku. V práci [9] sa podrobne rozoberá presná veľkosť jednotlivých buffrov pre konkrétne veľkosti vstupu. Okrem rozdelenia buffrov pridáva inicializačnú fázu, v ktorej sa

Algorithm 2 procedure $scan(b, i)$

1: $prev$ = b;
2: v = vrchol, do ktorého vstupuje t_i-*hrana* z $prev$, kde t je vstupný reťazec;
3: **while** v existuje **do**
4: w = označenie hrany $prev \xrightarrow{w} v$;
5: **if** $w = t_i \dots t_{i+|w|-1}$ **then**
6: $prev = v$;
7: $i = i + |w|$;
8: v =vrchol, do ktorého vstupuje t_i-*hrana* z v;
9: **else**
10: p =najdlhšia spoločná predpona w a s_i ($i$tá prípona t);
11: **return** $(prev, p, i + |p|)$
12: **end if**
13: **end while**
14: **return** $(prev, \lambda, i)$

prípony vstupu rozdelia na partície podľa zhody ich predpony zvolenej dĺžky. Výsledkom je jeden z najmodernejších algoritmov TDD (*Top-Down Disk-based*).

Stručne vysvetlíme, ako algoritmus WOTD pracuje. Pri implementácii algoritmu sme sa držali práce [3], kde čitateľ môže nájst presný popis. Keďže pre všoebecné reťazce vzniká pri implementácií tohto algoritmu niekoľko komplikácií, budeme pri jeho vysvetľovaní predpokladať, že všetky vstupné reťazce sú v tvare $w = t\$$, kde $\$ \in E$ je znak, ktorý sa v t nevyskytuje. Sufixový strom pre reťazec t ($ST(t)$) potom definujeme rovnako ako v pôvodnej definícii, až na bod (ii) reťazec w sa (implicitne) vyskytuje v T práve vtedy, ak w je podreťazcom $t\$$.

Najprv si zadefinujeme reláciu usporiadania $\prec$ na potomkoch vetviaceho vrcholu. Algoritmus $WOTD$ vytvára sufixový strom so synmi vetviaceho vrcholu zoradenými práve podľa tejto relácie. Nech $\overline{u}$ a $\overline{v}$ sú rôzni potomci rovnakého vetciaceho vrcholu. Potom $\overline{u} \prec \overline{v}$ práve vtedy, ak min $l(\overline{u}) <$ min $l(\overline{v})$. Ďalej zavedieme označenie $lp(\overline{uv}) = min(l(\overline{uv})) + |u|$. Algoritmus využíva 4 polia: $sufixes$, $tree$, $input$ a $temp$ a zásobník $stack$. Algoritmus najprv uvedieme, potom popíšeme.

Strom je v poli $tree$ reprezentovaný nasledovne. Ak sa ukladá list $\overline{u}$, všetky informácie, ktoré potrebujeme o listovej hrane do neho vedúcej sú uložené v jedinom ukazovateli do vstupu, kde označenie hrany začína ($lp(\overline{w})$).

Algorithm 3 procedure $mccbuild(t)$

1: vytvor koreň $root$;
2: pridaj hranu $root \xrightarrow{t} v$, kde t je vstupný reťazec a v je nový vrchol;
3: $b = root$; $tolink = NULL$; $i = 1$; $w = \lambda$;
4: v = vrchol, do ktorého vstupuje t_i-$hrana$ z b;
5: **while** $i \leq |t|$ or $b \neq root$ or $p \geq k$ **do**
6: **if** $w = \lambda$ **then**
7: **if** $b = root$ **then**
8: $(b, w, i) = scan(b, i + 1)$;
9: **else**
10: $(b, w, i) = scan(b.suf, i)$;
11: **end if**
12: **if** $w = \lambda$ **then**
13: $tolink = b$;
14: pridaj hranu $b \overset{t_i \ldots t_{|t|}}{\rightarrow} u$, kde u je nový vrchol;
15: **else**
16: $tolink = split(b, |w|)$;
17: pridaj hranu $tolink \overset{t_i \ldots t_{|t|}}{\rightarrow} u$, kde u je nový vrchol;
18: **end if**
19: **else**
20: $(b_2, w) = link(b, w)$; $nocanon = w$;
21: **if** $w = \lambda$ **then**
22: $tolink_2 = tolink$; $(b, w, i) = scan(b_2, w, i)$;
23: **if** $w = \lambda$ **then**
24: $tolink = b$; pridaj hranu $b \overset{t_i \ldots t_{|t|}}{\rightarrow} u$, kde u je nový vrchol;
25: **else**
26: $tolink = split(b, t_i, |w|)$;
27: pridaj hranu $tolink \overset{t_i \ldots t_{|t|}}{\rightarrow} u$, kde u je nový vrchol;
28: **end if**
29: $tolink_2.suf = canonize(b2, nocanon)$;
30: **end if**
31: $pom = split(b_2, |w|)$;
32: pridaj hranu $pom \overset{t_i \ldots t_{|t|}}{\rightarrow} u$, kde u je nový vrchol;
33: $tolink.suf = canonize(b_2, nocanon)$; $tolink = pom$; $b = b_2$;
34: **end if**
35: **end while**

Algorithm 4 WOTD

1: Naplň pole $sufixes$;
2: Zoraď $sufixes$ podľa $\prec$ s využitím $temp$;
3: Do poľa tree ulož vetviace a listové vrcholy;
4: Zaraď vrcholy ukazujúce na nevyhodnotený rozsah do $stack$;
5: **while** $stack$ nie je prázdny **do**
6: Vyber zo zásobníka $stack$ vrchol;
7: Zisti najldhšiu spoločnú predponu všetkých prípon v rozsahu vybraného vrcholu;
8: Zoraď rozsah v poli $sufixes$ s využitím $temp$;
9: Do poľa $tree$ ulož vetviace a listové vrcholy;
10: Zaraď vrcholy ukazujúce na nevyhodnotený rozsah do $stack$;
11: **end while**

Ak ukladám vetviaci vrchol $\overline{u}$, uložím ho na dve políčka v poli. Na prvé uložím $lp(\overline{u})$, na druhé ukazovateľ do poľa $tree$, ktorý ukazuje na miesto jeho prvého syna ($firstchild(\overline{u})$). Keďže sú vo výslednom strome potomci zoradení podľa $\prec$, z informácie uloženej na prvom ukazovateli prvého syna vieme, kde označenie popisovanej hrany (vedúcej do $\overline{u}$) končí (jeden znak pred tým, kde začína označenie hrany do jej prvého syna) a ušetríme tak jedno políčko v reprezentácii. Okrem toho sú v každom políčku poľa $tree$ uložené príznaky $rightmost$, $leaf$, $evaluated$, ktoré určujú, či vrchol predstavuje po rade: najpravejšieho syna, list, nevyhodnotený vrchol.

Pole $sufixes$ napĺňame pozíciami prvého znaku prípony. Na zoradenie použijeme algoritmus *countsort* [13], ktorý je výhodný, ak je kľúčov málo (v našom prípade Σ predstavuje množinu kľúčov použitých vo fáze počítania) a triedených prvkov naopak veľa (prípon je toľko, aký dlhý je vstup). Pole bude v tomto momente obsahovať najprv ukazovatele na všetky prípony začínajúce na znak t_1, potom na znak t_i, kde i je prvá pozícia v t taká, že $t_i \neq t_1$, atď.

Podľa takto zoradeného poľa $sufixes$ môžeme zistiť, akú hodnotu máme uložiť v prípade, že ide o list, resp. v prípade, že ide o vetviaci vrchol, vieme určiť hodnotu prvého políčka. Najprv si ale do $tree$ pre vetviace vrcholy dočasne uložíme informácie o rozsahu poľa suffixes, ktorý predstavuje prípony, ktoré sa z tohto vrcholu ešte budú ďalej spracovávať (usporiadanie $sufixes$ podľa relácie $\prec$ nám zabezpečí, že pre konkrétnu hranu budú všetky tieto prípony pekne vedľa seba a skutočne nám stačí uložiť iba dva ukazovatele) a nastavím príznaky.

Polohy nelistových vrcholov, ktoré sme takto vygenerovali, uložíme na zásobník. Tým dosiahneme tzv. *eager* poradie vyhodnocovania vrcholov.

Vo vnútri cyklu while sa robí skoro to isté, čo bolo popísané pred ním, až na to, že sa najprv vyberie rozsah v $sufixes$, ktorý sa má spracovávať, namiesto explicitne zadaného celého rozsahu. Ďalší rozdiel spočíva v tom, že sa hľadá najdlhšia spoločná predpona (lcp). Tento krok zároveň upraví pozície v zadanom rozsahu poľa $sufixes$ tak, aby ukazovali na prvý nezhodujúci sa znak, teda posunie ich o $|lcp|$ doprava. Teda keď sa beh algoritmu v niektorej z ďalších iterácií dostane k triedeniu tohto rozsahu, bude skutočne triediť prípony, ktoré sa pre spracovávaný vrchol majú ešte vyhodnotiť. Algoritmus si vždycky na koncy cyklu uloží na zásobník nevyhodnotené vrcholy (uzly). Inicializácia zásobníka prebehne pred prvým vstupom do cyklu. Koniec nastáva vo chvíli, keď je zásobník prázdny. Presný popis nájde čitateľ v [3].

3.1.4 Spôsoby implementácie algoritmov

Každý z uvedených algoritmov sa dá implementovať rôznymi spôsobmi. Okrem uvedeného algoritmu $WOTD$ sme u ďalších neuvádzali presný popis dátovej štruktúry reprezentujúcej vrchol, či hranu. Jednoducho sme v danom kroku povedali, že pridáme nový vrchol, alebo novú hranu s danými vlastnosťami. S odlišnou reprezentáciou vrcholu sa niekedy musí čiastočne upraviť aj program, ktorý túto štruktúru využíva, avšak podstata algoritmov ostáva pritom nezmenená.

U každej reprezentácie vrcholu je potrebné zabezpečiť, aby každý vrchol mal k dispozícii všetky hrany z neho vychádzajúce. To môžeme dosiahnuť napríklad tak, že vrchol bude obsahovať štruktúru, v ktorej si bude pamätať ukazovatele na všetkých svojich synov. To by však bolo nepraktické, pretože ku každému vrcholu by sme si museli pamätať ešte jeden ukazovateľ. Naviac, ak by táto štruktúra bola dynamická, dosiahli by sme veľmi premenlivú veľkosť vrcholu, naopak, ak by to bolo napr. pole, museli by sme ho alokovať dostatočne veľké, čiže na každý vrchol by sme potrebovali minimálne $|\Sigma|$-položkové pole. Z definície sufixového stromu ale plynie, že do každého vrcholu vstupuje práve jedna hrana. Preto sa často reprezentuje vrchol tak, že si pamätá iba označenie hrany, ktorá do neho vedie. Naviac si zapamätá ukazovateľ na jedného syna, ak ho má. Synovia sú potom štrukturovaní tak, aby boli dosiahnuteľní všetci. Najčastejšie spôsoby, ako synov štrukturovať sú spojový zoznam a hashovacia tabuľka. Existujú

pochopiteľne aj iné možnosti. My sme sa napr. pokúsili štrukturovať synov aj to binárneho stromu, algoritmus $WOTD$ uvádza ďalší spôsob, vyžívajúci usporiadanie hrán.

Pri reprezentácii spojovým zoznamom si treba pamätať ukazovateľ na ďalšieho syna, pri strome sú to dva ukazovatele. Pri hashovaní sa ako kľúč používa spravidla dvojica (pozícia otca v poli, označenie hrany). Vybraný spôsob prináša pochopiteľne odlišné pamäťové a výkonové vlastnosti. Pri zvolení akéhokoľvek postupu potrebujeme ale minimálne jeden ukazovateľ na prvého syna. Preto je ešte potrebné si určiť, či ukazovateľ ostane ukazovateľom do pamäte a vrcholy budú po jednom alokované, alebo sa vytvorí vlastná štruktúra, napríklad pole, a ukazovatele budú indexy do nej. Toto rozhodnutie zase prináša možné výhody a nevýhody. Voľba alokácie vrcholov po jednom znamená množstvo alokácií. Je ich vlastne toľko, aký veľký je strom. To však nemusí znamenať až taký problém. Ako ukazujú nasledujúce sekcie jeden vrchol zaberá v pamäti maximálne niekoľko desiatok bytov. Alokácia takého malého množstva pamäte je pre systém veľmi pohodlná, preto táto voľba nemusí byť príliš časovo náročná, ale v súčte je určite náročnejšia, než naalokovať len jeden veľký kus naraz aj napriek tomu, že pri fragmentácii voľnej pamäte môže byť alokácia veľkého kusu zložitá. Výhodou postupnej alokácie je, že sa spotrebuje presne toľko pamäte, aký veľký je výsledný strom. Pri alokácii poľa môžeme buď voliť maximálnu možnú veľkosť, alebo menšiu veľkosť s možnosťou prealokovania. Obidve voľby ale prinášajú možnosť prebytočnej spotreby pamäte, naviac druhá možnosť so sebou nesie časové, ale aj pamäťové náklady na kopírovanie pri prealokovaní. Z časového hľadiska je však tento prístup efektívnejší.

V našej aplikácii sme sa rozhodli využiť rôzne z uvedených prístupov. Pre Ukkonenov a McCreightov algoritmus sme volili prístup cez spojový zoznam a pokusne aj binárny strom a vylepšený spojový zoznam. Vo všetkých týchto spôsoboch sme volili postupnú alokáciu jednotlivých vrcholov a teda ukazovatele priamo do pamäte. Vylepšenie spojového zoznamu spočíva v tom, že si každý vrchol naviac pamätá posledného navštíveného syna. Konštručné algoritmy totiž často hľadajú dvakrát tú istú hranu, preto sme uvažovali, že by to mohlo priniesť časový zisk. Ďalšia úprava spočíva v pridaní prvého písmena z označenia hrany do reprezentácie vrcholu. To odstráni skákanie po vstupnom reťazci pri použití indexu. U stromu sme uvažovali len variantu s pridanými pomocnými premennými ako vo vylepšenom spojovom zozname. Inicializácia a akutalizácia týchto premenných však môže tieto výhody znížiť, predpokladáme však, že pre dostatočne veľký vstup, bude náš

prístup rýchlejší. Ukazovatele ako index do poľa využíva reprezentácia vrcholu u algoritmu *WOTD*. Spôsob štruktúrovania synov najviac pripomína spojový zoznam, v ktorom nie je potrebné si explicitne pamätať ukazovateľ na ďalší prvok, pretože sa nachádza hneď vedľa.

3.2 Základný popis porovnávacej aplikácie

3.2.1 Motivácia

Priložená aplikácia má za úlohu ukázať, ako sa niektoré pozorované algoritmy správajú v praxi. Vyrobili sme preto rozhranie, v ktorom sa dajú naše implementácie otestovať na konkrétnych vstupoch. Takto môžeme okrem teoreticky dokázaných vlastností algoritmov ďalej skúmať ich použiteľnosť v konkrétnych prípadoch a objektívne zohľadniť ich praktické vlastnosti. Je síce zrejmé, že použitie sufixových stromov je obrovské a existuje mnoho ďalších algoritmov a variácií pre ich implementáciu v presnom prípade využitia a určite existuje mnoho ďalších možností, ako našu aplikáciu rozšíriť, pre rozsah našej práce však postačuje a pomôže nám pri zostavovaní taxonómie.

3.2.2 Prostredie, vstupy a výstupy

Aplikácia je tvorená programom sufcomp spustiteľnom z príkazového riadku systému MS Windows XP a zo shelu operačného systému Linux. Zdrojový kód sa dá preložiť ako na 32-bitových, tak aj na 64-bitových procesoroch. Môžeme teda rovnaké testy prevádzať na rôznych kombináciách stavby systému. Aj keď to nebolo primárnym cieľom aplikácie, bolo by zaujímavé porovnať správanie jednotlivých algoritmov na odlišných systémoch a pokúsiť sa výsledky odôvodniť, ak sa budú podstatne odlišovať.

Program sufcomp si berie z príkazového riadku jeden alebo žiaden argument. Ostatné sú ignorované. Jediným argumentom je *vstupný súbor*, v ktorom sú v určitom formáte uložené ďalšie parametre behu programu. V súbore sa nachádzajú informácie o tom, ktoré algoritmy sa majú testovať, cesty k testovacím súborom a ďalšie nastavenia ako napr. typ výstupu, alebo testovanie správnosti postavenej štruktúry. Pre presný popis formátu tohto súboru viď *užívateľskú dokumentáciu*, ktorá je súčasťou priloženej aplikácie.

Výstupom programu sú súbory, v ktorých sú uložené výsledky testov. Ak sme vo vstupnom súbore uviedli, že si želáme iba pamäťovú alebo iba

časovú náročnosť, súbory pre tú druhú sa nevytvoria a ani sa pri behu tieto informácie nezisťujú. Výstup je buď vo formáte .csv alebo v textovom formáte usporiadaný do textovej tabuľky.

3.3 Výsledky testov

3.3.1 Tabuľky s výsledkami

V tejto sekcii sa nachádzajú tabuľky s výsledkami testov. Ku každej tabuľke sú v jej popise uvedené základné parametre testovacej zostavy a použitý operačný systém. Tabuľky zobrazujú výsledky tak rýchlostných, ako aj pamäťových testov. Rýchlosť znamená, koľko kB vstupu sa v priemere spracuje za jednu sekundu. Je teda udávaná vždy v kB/s. Výsledky pamäťových testov ukazujú, koľko B pamäte pripadá na jeden B vstupu pre vytvorenie štruktúry daným algoritmom. Tučne sú označené najlepšie výsledky pre daný test.

V riadkoch sú uvedené dosiahnuté výsledky pre konkrétny vstup. Názvy stĺpcov vysvetlíme. Stĺpec názov súboru obsahuje názov vstupného súboru, na ktorom bol test uskutočnený. UkkList udáva výsledky pre Ukkonenov algoritmus s hranami implementovanými do (vylepšeného) spojového zoznamu s postupnou alokáciou pre každý vrchol. UkkTree: Ukkonenov algoritmus, hrany implementované do binárneho vyhľadávacieho stromu, postupná alokácia. UkkStd: Ukkonenov algoritmus, hrany implementované do štandardného (nevylepšeného) spojového zoznamu, MccList, MccTree a MccStd: McCreightov algoritmus, spôsoby implementácie analogicky podľa popisu stĺpcov začínajúcich Ukk-. Stĺpec WOTD obsahuje výsledky pre algoritmus WOTD s hranami implementovanými do poľa, ako sme popisovali v 3.1.3. Posledný riadok Celkový priemer nespočítava aritmetický priemer hodnôt v stĺpci, ale obsahuje podiel súčtu veľkostí testovacích súborov a súčtu časov pre jednotlivé behy.

Do testov sme zahrnuli súbory tak, aby boli zastúpené rôzne veľkosti vstupných abecied, rôzne veľkosti vstupov a rôzne druhy textov: prirodzený text(väčšinou s príponou .txt), prirodzený text s formátovaním (.html), text programov, binárny súbor. Využili sme na to vybranú časť Calgarského a Canterburského korpusu, ale aj vlastný súbor hp2.pdf (viď kapitolu 5). Čitateľ može Calgasrký, Canterbusrký korpus a moje súbory nájsť postupne na internetových odkazoch:
http://links.uwaterloo.ca/calgary.corpus.html

http://corpus.canterbury.ac.nz/
http://jhost.php5.cz/my.zip.

Na testovanie rýchlosti sme použili systémové hodiny tak, že sme si zapamätali presný čas začiatku čo najtesnejšie pred prvým príkazom algoritmu, teda tak, aby sa do meraného času nezarátala napr. doba, počas ktorej sa výsledná štruktúra kontroluje na správnosť alebo sa zapisujú získané výsledky. Potom ihneď po skončení behu algoritmu sa spočíta rozdiel medzi aktuálnym a zapamätaným systémovým časom a spočíta sa priemerná rýchlosť. Pre pamäťové testy sme spočítali veľkosť jedného uzlu stromu pomocou operátoru sizeof v C++, spočítali sme vrcholy vyskytujúce sa v strome a vynásobili tieto dve hodnoty. Výnimkou je algoritmus WOTD, kde sme len zobrali veľkosť využitej časti výsledného poľa a vynásobili počtom B pripadajúcim na 1 políčko. Dostali sme veľkosť výsledného stromu v bytoch, potom sme to predelili veľkosťou vstupu a zapísali do tabuľky.

Tabuľka 3.1 zobrazuje hodnoty veľkosti súborov a veľkosti ich vstupných abecied. Stĺpec veľkosť obsahuje veľkosť pre súbor zo stĺpca názov súboru. Stĺpec k obsahuje veľkosti ich abecied.

Tabuľky 3.2 a 3.3 obsahujú výsledky testov pre pamäťovú náročnosť na OS Windows XP. Výsledky pamäťových testov pre ďalší testovaný OS Kubuntu Linux uvedené nie sú, pretože boli úplne totožné s výsledkami pre OS Windows XP. Tabuľky 3.4 až 3.7 obsahujú výsledky rýchlostných testov pre Windows a Kubuntu Linux. Do jednej tabuľky sa nezmestili výsledky pre všetky algoritmy, preto boli výsledky rozdelené pre obidva operačné systémy do dvoch tabuliek. Pri zvýrazňovaní najlepšieho času sa pochopiteľne zvýrazní najlepšia hodnota z dvoch tabuliek. V tabuľkách 3.8 až 3.11 sú uvedené percentá, ktoré vyjadrujú rýchlosti oproti najúspešnejšiemu, teda najrýchlejšiemu algoritmu. Hodnoty **INF** v týchto tabuľkách znamenajú nekonečno. Znamená to, že daný test trval príliš krátko na to, aby bol zmerateľný čas jeho behu. Symboly N/A u percentuálneho vyjadrenia znamenajú, že v pôvodných rýchlostiach sa vyskytlo na danom riadku nekonečno a totiž nie je možné určiť správny podiel. Pri počítaní priemerných hodnôt pri výsledku **INF** sme ako čas započítali nulu a ako počet spracovaných B veľkosť súboru. Percentuálne vyjadrenie pre pamäťové testy sme neuvádzali do osobitných tabuliek, pretože je pre každý vstup totožné (až na malé odchýlky u algoritmu $WOTD$), čo je vzhľadom k implementácii dosť pochopiteľné, kedže výsledný strom má rovnaký počet vrcholov a vrcholy v našich implementáciách zaberajú vždy rovnako pamäte. Výnimkou je síce $WOTD$, ktorý rozlišuje uzly a listy, ale nameraný per-

centuálny rozdiel je zanedbateľný. Namiesto toho sme preto uviedli naviac jeden riadok Percentuálne, ktorý vyjadruje percento spotrebovanej pamäte oproti najhoršiemu (teda pamäťovo najnáročnejšiemu) výsledku v riadku Celkový priemer, čo vlastne aj vyjadruje percento z najhoršieho algoritmu pre každý vstup.

Za týmito tabuľkami sa ešte nachádzajú výsledky pre veľké súbory v tabuľkách 3.12 až 3.15. Tieto testy sme prevádzali osobitne, pretože na pôvodnej zostave sa využívala virtuálna pamäť, čo spôsobilo extrémne pomalý beh a získané výsledky by pre nás nemali žiadnu cenu. Ďalšia zostava obsahuje podobný procesor, avšak väčšiu operačnú pamäť (viď tabuľky). Tieto súbory sme uvažovali osobitne aj preto, lebo by v priemere značne ovplyvnili výsledky predošlých testov.

3.4 Vyhodnotenie testov a skúmanie vlastností algoritmov

Zo zistených hodôt výsledkov testov sa pokúsime odhaliť prednosti a úskalia jednotlivých implementovaných algoritmov. Budeme brať do úvahy ich výsledky pamäťovej náročnosti ako aj efektivitu práce a pokúsime sa odôvodniť, prečo sú výsledky také, aké sú.

Pri pohľade na výsledky rýchlostných testov sa ako prvá zaujímavosť ukazuje to, že kvadratický algoritmus $WOTD$ sa často vyrovnáva a neraz predstihuje obidve implementácie jeho lineárnych kolegov. Hneď v prvom teste *hp.pdf* dokonca predstihuje druhý najlepší algoritmus (pre tento test) $Ukk\text{-}Tree$ viac než dvojnásobne na obidvoch OS. Je to zapríčinené tým, že algoritmus $WOTD$ počas konštrukcie skoro vôbec neprechádza vytváranú štruktúru, ktorá je značne väčšia než vstupný reťazec. Počet prístupov do vstupného reťazca je síce kvadratický, ten sa však prechádza sekvenčne, čo je veľmi priaznivé pre vyrovnávaciu pamäť cache na procesore. Táto vlastnosť $WOTD$ dokáže kompenzovať jeho kvadratickosť a dokonca v niektorých prípadoch pomáha prekonať toto obmedzenie.

Z nameraných výsledkov sa nedá presne povedať, aký vplyv má veľkosť abecedy na rýchlosť výstavby u $WOTD$. Pri malej abecede ako napr. u *pi.txt* ($|\Sigma| = 10$) dosahuje najlepší výsledok, avšak pri ešte menšej abecede u *E.coli* ($|\Sigma| = 4$) dosahuje najhorší výsledok. Pri spracovaní vstupov s väčšími abecedami sa taktiež výsledky značne líšia. Rýchlosť behu algoritmu $WOTD$ samozrejme nezávisí len od veľkosti vstupnej abecedy, ale aj od iných faktorov. Štruktúra vstupu môže tento algoritmus veľmi spomaliť a jeho kvadratickosť začne vážne prekážať. Vidno to napríklad v teste *alphabet.txt*, v ktorom sa opakuje jedna sekvencia písmen mnoho krát. Zlý výsledok sme zaznamenali aj pre vstup *ppt*5. Príčiny sú v tomto prípade veľmi podobné. Keďže tieto dva vstupy sú veľmi veľké a ich výsledky sú také slabé, výrazne to ovplyvnilo celkový priemer pre tento algoritmus.

U implementácií algoritmov Ukk a Mcc sme skúsili použiť ako dátovú štruktúru pre reprezentáciu množiny hrán vychádzajúcich z vrcholu spojový zoznam a binárny strom. Pri obidvoch štruktúrach sme pokusne použili trik, kedy si pamätáme poslednú navštívenú hranu, a tak sa v mnohých prípadoch urýchli hľadanie. Často totiž u algoritmov Mcc a Ukk nastáva situácia, kedy niekoľkokrát po sebe treba navštíviť tú istú hranu. Preto si každý vrchol pamätá hranu, ktorú navštívil ako poslednú. Naviac si každý vrchol pamätá prvý znak označenia hrany vedúcej do neho. To odbúrava skákanie vo vstup-

nom reťazci pri vyhľadávaní hrany, čo šetrí čas najmä pri veľkých vstupoch. Tento trik je síce pamäťovo náročný, ako ukážu testy pre pamäťovú náročnosť, avšak takto implementované algoritmy často porážajú a v priemere sú rýchlejšie než štandardné implementácie.

Ak si všimneme rozdiely výsledkov štandardnej a vylepšenej implementácie spojového zoznamu, zistíme, že hlavnú rolu pre ich rýchlosť zohráva veľkosť vstupnej abecedy. Pri menších abecedách naše vylepšenia nepomáhajú, skôr naopak, znamenajú vačšiu záťaž kvôli ich pamäťovej náročnosti a réžii spojenej s aktualizáciu pomocných premenných. Najlepšie je to vidieť na teste *E.coli*, kedy štandardná implementácia prekonáva všetky ostatné algoritmy. Podobne u *pi.txt* je štadnardná implementácia rýchlejšia než vylepšená, aj keď ostatné algoritmy neprekonáva. Vo väčšine ostatných testov však táto implementácia prehráva v porovnaní s vylepšenou.

Zaujímavý je pohľad na veľkosť vstupu a dosiahnuté výsledky. U *Ukk* a *Mcc* sa dosahujú v podstate veľmi podobné časy pri všetkých veľkostiach vstupu. Veľké výkyvy sme ale zaznamenali u algoritmu $WOTD$. Ten pre menšie vstupy neposkytuje až také dobré výsledky ako konkurencia. Najlepšie to vidno na testoch *progc*(39kB), *obj*1(21kB) a *xargs*.1,(4kB) kde skončil buď ako posledný alebo medzi najhoršími. Ak sa ale veľkosť vstupu zvýši na stovky kB ako napr. v prípade *obj*2, jeho výkon stúpa. V prípade testu *random.txt*(100kB) dokonca dosiahol jednoznačne najlepší výsledok. Pri ďalšom zvyšovaní veľkosti vstupu až na niekoľko MB (*bible.txt*, *E.coli*, *world-192.txt*) však jeho výkon opäť klesá. Aj napriek tomu pri stredne veľkých vstupoch ako (*hp2.pdf* (861 kB) a *pi.txt* (1 MB)) zase vidíme jeho výborný výsledok na prvom mieste. Tieto výkyvy sú zapríčinené potrebami jemného ladenia algoritmu $WOTD$. Autori [8] ukazujú spôsob, akým empiricky najlepšie implementovať alokáciu pamäte pre jednotlivé štruktúry potrebné na výstavbu stromu tak, aby výsledný algoritmus dosahoval najlepšie výsledky pre konkrétne veľkosti vstupov pri zohľadnení veľkosti dostupnej pamäte. Rozobrali niekoľko prípadov a zistili, že napr. pre malé vstupy je vhodné nechávať len málo miesta pre vstupný reťazec a čo najviac miesta pre výsledné pole reprezentujúce strom a taktiež pre pomocné štruktúry. Naopak pre veľké vstupy, kedy sa do pamäte nevojdú všetky štruktúry potrebné na výstavbu, vychádza ako najefektívnejšie nechať najviac miesta na buffer pre vstupný reťazec a na ostatné štruktúry iba minimum. Vtedy je totiž počet prístupov na disk počas behu algoritmu najnižší. My sme ale takéto úpravy neimplementovali a neprepisovali sme systémový spôsob riadenia činnosti virtuálnej pamäte.

Hneď na prvý pohľad do tabuľky s výsledkami testov pamäťovej náročnosti môžeme všimnúť, že hodnoty v stĺpcoch *Ukk-List* a *Ukk-Tree* sú úplne zhodné s hodnotami v *Mcc-List* a *Mcc-Tree*. Totiž jediný rozdiel medzi algoritmami *Ukk-List* a *Ukk-Tree* je v implementácii vrcholov a tým aj v implementácii operácií s vrcholmi (podobne *Mcc-*). Je zrejmé, že veľkosť výsledného stromu (teraz myslíme počet vrcholov stromu) je pre každý algoritmus pri konkrétnom vstupe rovnaká. Reálna veľkosť spotrebovanej pamäte záleží len na tom, ako presne si implementujeme jeden vrchol.

Ďalšia vec, ktorú si všimneme je fakt, že podľa výsledkov testov má naša implementácia algoritmu *WOTD* výrazne priaznivejšiu pamäťovú náročnosť než implementácie konkurenčných algoritmov. Táto skutočnosť platí pritom bez ohľadu na veľkosť a charakter vstupu, či veľkosť vstupnej abecedy. Je to zapríčinené najmä tým, aký spôsob implementácie sme volili pre konštrukciu jednotlivých algoritmov. Nami zvolený prístup vo vylepšenej implementácii vrcholov do spojového zoznamu a do stromu u algoritmov *Ukk* a *Mcc* sa podľa týchto výsledkov zdá byť v praxi nepoužiteľný. Zaujímavé je ale zistenie, že pri použití binárneho vyhľadávacieho stromu sme dosiahli bezkonkurenčne najlepšie vvýsledky skoro u všetkých testovacích súboroch. Tento prístup sme volili síce pokusne, ale zdá sa, že by možno malo zmysel uvažovať, ako znížiť jeho pamäťové nároky tak, aby bol použiteľný.

Z práce [4] vyplýva, že aj u algoritmov *Mcc* a *Ukk* je možné oproti pôvodným 28B potrebným pri ich prvých implementáciách znížiť ich pamäťové nároky pod 20B na znak vstupu (za predpokladov, že celočíselný typ zaberá 4B a počet potrebných celých čísel na implementáciu uzlov je 5 a listov len 1). Náš prístup pre štandardnú implementáciu spojového zoznamu síce kopíruje tieto hodnoty, ale vylepšená implementácia a implementácia pomocou stromu sa ukázali byť dokonca pamäťovo horšie než úplne prvé implementácie spomínaných algoritmov. Na druhej strane algoritmus *WOTD* má priemerný počet B na jeden znak vstupu 14.69 a ani v jednom teste nedosahuje 20B, a preto čo sa týka pamäťovej náročnosti je lepší než jeho konkurenti aj pri ich vylepšení podľa [4].

V ďalšom štúdiu by bolo zaujímavé implementovať vrcholy *Ukk* a *Mcc* do poľa, zmenšiť veľkosť jedného vrcholu, tým odbúrať niektoré pamäťové nároky a následne z toho určiť zmenu ich pamäťovej náročnosti vzhľadom k rýchlosti výstavby. Avšak k významným pamäťovým zlepšeniam by sme pravdepodobne neprišli. V [2] sa totižto o algoritme *WOTD* spomína, že sa teoreticky dosiahlo isté zlepšenie pamäťovej náročnosti v najhoršom prípade aj oproti postupu uvedenému v [4], kde sa pojednáva zlepšenie pamäťovej

náročnosti založené na použití algoritmu Mcc. Strom postavený algoritmom $WOTD$ pritom vôbec neprináša zhoršenie zložitosti vyhľadávania. Keď sa chceme pozrieť na označenie hrany, nemáme síce oba indexy do vstupu k dispozícii ihneď, ale musíme sa najprv pozrieť na prvého syna, čo by predstavovalo o jeden prístup do stromu naviac. Avšak, pri hľadaní správnej hrany, po ktorej sa chceme pri vyhľadávaní posunúť hlbšie do stromu, nám stále stačí vedieť jej prvý znak. Keďže máme k dispozícii ukazovateľ do vstupu na miesto začiatku označenia hrany, dá sa tento znak získať priamo, bez nutnosti navštívenia prvého syna (stále však musíme skákať po vstupe). K prvému synovi teda potrebujeme skákať iba vtedy, ak sa v strome skutočne pohybujeme ďalej a našli sme hranu s hľadaným označením.

názov súboru	**veľkosť**	**k**
hp2.pdf	861210	256
bib	111261	81
book1	768771	82
book2	610856	96
geo	102400	256
news	377109	84
obj1	21504	256
obj2	246814	256
paper1	53161	95
paper2	82199	91
progc	39611	92
progl	71646	87
progp	49379	89
trans	93695	99
random.txt	100000	64
alphabet.txt	100000	26
alice29.txt	152088	74
asyoulik.txt	129301	69
cp.html	25248	87
fields.c	11581	91
grammar.lsp	3721	76
kennedy.xls	1029744	256
lcet10.txt	426754	84
plrabn12.txt	481861	81
ptt5	513216	159
sum	38240	255
xargs.1	4227	74
bible.txt	4077775	64
E.coli	4638960	4
world192.txt	2473400	94
pi.txt	1000000	10

Tabuľka 3.1: Veľkosti súborov a ich abecied

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
hp2.pdf	43.06	47.85	28.70
bib	55.35	61.50	36.90
book1	54.04	60.04	36.02
book2	55.12	61.25	36.75
geo	45.73	50.82	30.49
news	54.69	60.77	36.46
obj1	46.10	51.22	30.73
obj2	55.45	61.61	36.96
paper1	55.65	61.83	37.10
paper2	54.92	61.06	36.61
progc	55.23	61.37	36.82
progl	59.35	65.95	39.57
progp	60.10	66.78	40.07
trans	61.50	68.34	41.00
random.txt	42.90	47.67	28.60
alphabet.txt	0.01	0.01	0.01
alice29.txt	55.13	61.26	36.75
asyoulik.txt	54.06	60.07	36.04
cp.html	54.67	60.74	36.44
fields.c	57.11	63.46	38.07
grammar.lsp	57.79	64.21	38.53
kennedy.xls	37.97	42.19	25.31
lcet10.txt	55.10	61.22	36.73
plrabn12.txt	53.71	59.67	35.80
sum	53.34	59.26	35.56
xargs.1	54.25	60.28	36.17
Celkový priemer	51.05	56.72	34.03
Percentuálne	90.00	60.00	100.00

Tabuľka 3.2: P4 1600 MHz, 512MB RAM, WinXP, priemerný počet B potrebných na spracovanie 1B vstupu.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
hp2.pdf	43.06	47.80	28.70	**11.13**
bib	55.35	61.50	36.90	**16.60**
book1	54.04	60.04	36.02	**16.01**
book2	55.12	61.25	36.75	**16.50**
geo	45.73	50.82	30.49	**12.32**
news	54.69	60.77	36.46	**16.32**
obj1	46.10	51.22	30.73	**12.85**
obj2	55.45	61.61	36.96	**16.64**
paper1	55.65	61.83	37.10	**16.73**
paper2	54.92	61.02	36.61	**16.41**
progc	55.23	61.37	36.82	**16.55**
progl	59.35	65.95	39.57	**18.38**
progp	60.10	66.78	40.07	**18.71**
trans	61.50	68.34	41.00	**19.35**
random.txt	42.90	47.67	28.60	**11.06**
alphabet.txt	0.01	0.01	0.01	**0.00**
alice29.txt	55.13	61.26	36.75	**16.50**
asyoulik.txt	54.06	60.07	36.04	**16.03**
cp.html	54.67	60.74	36.44	**16.30**
fields.c	57.11	63.46	38.07	**17.39**
grammar.lsp	57.79	64.21	38.53	**17.74**
kennedy.xls	37.97	42.19	25.31	**8.80**
lcet10.txt	55.10	61.22	36.73	**16.49**
plrabn12.txt	53.71	59.67	35.80	**15.87**
sum	53.34	59.26	35.56	**15.71**
xargs.1	54.25	60.28	36.17	**16.11**
celkovy priemer	51.05	56.72	34.03	**14.69**
Percentuálne	90.00	100.00	60.00	**25.90**

Tabuľka 3.3: P4 1600 MHz, 512MB RAM, WinXP, priemerný počet B potrebných na spracovanie 1B vstupu.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
hp2.pdf	45.38	351.75	39.03
bib	462.36	**577.94**	496.13
book1	329.13	444.76	324.72
book2	381.91	495.88	378.04
geo	130.55	492.61	118.48
news	318.57	491.03	314.22
obj1	333.33	**1312.50**	338.71
obj2	321.37	571.16	302.42
paper1	476.29	552.29	552.29
paper2	395.43	**569.31**	514.57
progc	495.93	**623.91**	**623.91**
progl	559.73	**641.90**	**641.90**
progp	513.00	618.23	618.23
trans	489.30	582.80	586.53
random.txt	173.46	447.96	195.31
alphabet.txt	**INF**	6510.42	6103.52
alice29.txt	396.06	500.08	431.75
asyoulik.txt	403.42	539.62	404.71
cp.html	770.51	770.51	770.51
fields.c	**753.97**	706.85	706.85
grammar.lsp	**INF**	**INF**	227.11
kennedy.xls	178.78	307.90	130.55
lcet10.txt	386.60	493.78	398.04
plrabn12.txt	346.26	449.44	354.34
ptt5	562.50	745.82	563.13
sum	397.27	**794.55**	339.49
xargs.1	275.20	**INF**	**INF**
celkovy priemer	174.06	447.72	153.09

Tabuľka 3.4: P4 1600 MHz, 512MB RAM, WinXP, rýchlosť v kB/s.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
hp2.pdf	44.05	336.41	44.37	**827.78**
bib	434.61	577.94	535.24	496.13
book1	328.99	**449.02**	336.06	**449.02**
book2	385.61	**509.43**	401.71	459.94
geo	128.04	492.61	139.08	**800.00**
news	309.99	**512.20**	332.07	471.54
obj1	446.81	677.42	456.52	333.33
obj2	314.66	**592.21**	335.23	496.97
paper1	471.96	552.29	**665.58**	552.29
paper2	429.26	514.57	514.57	**569.31**
progc	**623.91**	**623.91**	**623.91**	495.93
progl	636.06	**641.90**	636.06	406.78
progp	518.51	618.23	**765.42**	344.44
trans	531.97	653.56	648.93	324.47
random.txt	168.66	415.56	208.67	**1252.00**
alphabet.txt	**INF**	**INF**	**INF**	5.18
alice29.txt	396.06	**526.68**	431.75	476.04
asyoulik.txt	404.71	537.32	447.77	**576.58**
cp.html	524.60	**795.36**	**795.36**	524.60
fields.c	706.85	706.85	706.85	364.83
grammar.lsp	**INF**	227.11	**INF**	242.25
kennedy.xls	180.28	313.96	147.28	**1368.18**
lcet10.txt	392.05	**512.61**	416.75	476.29
plrabn12.txt	334.69	463.16	367.34	**486.12**
ptt5	594.53	**783.11**	617.23	10.38
sum	397.27	**794.55**	478.77	478.77
xargs.1	258.00	**INF**	**INF**	258.00
celkovy priemer	171.62	**451.73**	170.70	82.42

Tabuľka 3.5: P4 1600 MHz. 512MB RAM. WinXP. rýchlosť v kB/s.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
hp2.pdf	49.79	414.30	38.97
bib	679.08	**835.80**	639.14
book1	441.62	581.98	403.63
book2	532.62	693.65	493.01
geo	156.25	625.00	128.21
news	433.26	708.21	383.62
obj1	525.00	1050.00	420.00
obj2	446.35	892.70	365.20
paper1	741.64	**1038.30**	865.25
paper2	668.94	802.73	668.94
progc	773.65	**1289.42**	967.07
progl	999.53	999.53	999.53
progp	964.43	**1205.54**	964.43
trans	914.99	**1143.74**	914.99
random.txt	212.30	542.54	207.78
alphabet.txt	9765.62	**INF**	4882.81
alice29.txt	594.09	**742.62**	594.09
asyoulik.txt	573.96	701.50	601.29
cp.html	821.88	**1232.81**	821.88
fields.c	1130.96	1130.96	1130.96
grammar.lsp	363.38	363.38	**INF**
kennedy.xls	210.82	345.57	120.58
lcet10.txt	548.36	683.20	484.60
plrabn12.txt	470.57	611.13	443.93
ptt5	945.64	1222.41	771.06
sum	533.48	**1244.79**	466.80
xargs.1	412.79	412.79	**INF**
celkovy priemer	208.09	578.08	160.07

Tabuľka 3.6: P4 1600 MHz, 512MB RAM, Kubuntu Linux, rýchlosť v kB/s.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
hp2.pdf	48.73	398.59	44.57	**1201.46**
bib	679.08	**835.80**	679.08	679.08
book1	441.62	591.14	417.09	**605.45**
book2	537.42	**710.17**	509.86	602.57
geo	156.25	625.00	149.25	**1428.57**
news	433.26	**722.10**	404.69	613.78
obj1	525.00	**2100.00**	700.00	300.00
obj2	446.35	**927.04**	422.86	651.43
paper1	865.25	865.25	865.25	741.64
paper2	668.94	**891.92**	729.75	729.75
progc	967.07	967.07	967.07	773.65
progl	999.53	**1166.11**	999.53	499.76
progp	**1205.54**	**1205.54**	**1205.54**	370.94
trans	1016.66	1016.66	1016.66	351.92
random.txt	207.78	542.54	217.01	**1627.60**
alphabet.txt	9765.62	**INF**	**INF**	3.06
alice29.txt	618.85	**742.62**	618.85	707.25
asyoulik.txt	573.96	**742.77**	631.35	**742.77**
cp.html	821.88	**1232.81**	821.88	821.88
fields.c	1130.96	**INF**	**INF**	565.48
grammar.lsp	**INF**	363.38	**INF**	363.38
kennedy.xls	214.42	356.60	138.13	**1478.84**
lcet10.txt	534.30	**718.54**	490.30	612.87
plrabn12.txt	475.32	619.17	452.47	**635.90**
ptt5	963.82	**1392.19**	835.31	8.52
sum	622.40	**1244.79**	533.48	622.40
xargs.1	**INF**	412.79	**INF**	412.79
celkovy priemer	206.67	**587.71**	179.72	64.61

Tabuľka 3.7: P4 1600 MHz, 512MB RAM, Kubuntu Linux, rýchlosť v kB/s.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
hp2.pdf	5.48	42.49	4.72
bib	80.00	**100.00**	85.84
book1	73.30	99.05	72.32
book2	74.97	97.34	74.21
geo	16.32	61.58	14.81
news	62.20	95.87	61.35
obj1	25.40	**100.00**	25.81
obj2	54.27	96.45	51.07
paper1	71.56	82.98	82.98
paper2	69.46	**100.00**	90.38
progc	79.49	**100.00**	**100.00**
progl	87.20	**100.00**	**100.00**
progp	67.02	80.77	80.77
trans	74.87	89.17	89.74
random.txt	13.85	35.78	15.60
alphabet.txt	**100.00**	N/A	N/A
alice29.txt	75.20	94.95	81.98
asyoulik.txt	69.97	93.59	70.19
cp.html	96.88	96.88	96.88
fields.c	**100.00**	93.75	93.75
grammar.lsp	**100.00**	**100.00**	N/A
kennedy.xls	13.07	22.50	9.54
lcet10.txt	75.42	96.33	77.65
plrabn12.txt	71.23	92.45	72.89
ptt5	71.83	95.24	71.91
sum	50.00	**100.00**	42.73
xargs.1	N/A	**100.00**	**100.00**
celkovy priemer	38.53	99.11	33.89

Tabuľka 3.8: P4 1600 MHz, 512MB RAM, WinXP, rýchlosť v percentách najlepšieho.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
hp2.pdf	5.32	40.64	5.36	**100.00**
bib	75.20	**100.00**	92.61	85.84
book1	73.27	**100.00**	74.84	**100.00**
book2	75.69	**100.00**	78.86	90.29
geo	16.01	61.58	17.39	**100.00**
news	60.52	**100.00**	64.83	92.06
obj1	34.04	51.61	34.78	25.40
obj2	53.13	**100.00**	56.61	83.92
paper1	70.91	82.98	**100.00**	82.98
paper2	75.40	90.38	90.38	**100.00**
progc	**100.00**	**100.00**	**100.00**	79.49
progl	99.09	**100.00**	99.09	63.37
progp	67.74	80.77	**100.00**	45.00
trans	81.40	**100.00**	99.29	49.65
random.txt	13.47	33.19	16.67	**100.00**
alphabet.txt	**100.00**	**100.00**	**100.00**	N/A
alice29.txt	75.20	**100.00**	81.98	90.38
asyoulik.txt	70.19	93.19	77.66	**100.00**
cp.html	65.96	**100.00**	**100.00**	65.96
fields.c	93.75	93.75	93.75	48.39
grammar.lsp	**100.00**	N/A	**100.00**	N/A
kennedy.xls	13.18	22.95	10.76	**100.00**
lcet10.txt	76.48	**100.00**	81.30	92.91
plrabn12.txt	68.85	95.28	75.57	**100.00**
ptt5	75.92	**100.00**	78.82	1.33
sum	50.00	**100.00**	60.26	60.26
xargs.1	N/A	**100.00**	**100.00**	N/A
celkovy priemer	37.99	**100.00**	37.79	18.25

Tabuľka 3.9: P4 1600 MHz, 512MB RAM, WinXP, rýchlosť v percentách najlepšieho.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
hp2.pdf	4.14	34.48	3.24
bib	81.25	**100.00**	76.47
book1	72.94	96.12	66.67
book2	75.00	97.67	69.42
geo	10.94	43.75	8.97
news	60.00	98.08	53.12
obj1	25.00	50.00	20.00
obj2	48.15	96.30	39.39
paper1	71.43	**100.00**	83.33
paper2	75.00	90.00	75.00
progc	60.00	**100.00**	75.00
progl	85.71	85.71	85.71
progp	80.00	**100.00**	80.00
trans	80.00	**100.00**	80.00
random.txt	13.04	33.33	12.77
alphabet.txt	N/A	**100.00**	N/A
alice29.txt	80.00	**100.00**	80.00
asyoulik.txt	77.27	94.44	80.95
cp.html	66.67	**100.00**	66.67
fields.c	**100.00**	**100.00**	**100.00**
grammar.lsp	N/A	N/A	**100.00**
kennedy.xls	14.26	23.37	8.15
lcet10.txt	76.32	95.08	67.44
plrabn12.txt	74.00	96.10	69.81
ptt5	67.92	87.80	55.38
sum	42.86	**100.00**	37.50
xargs.1	N/A	N/A	**100.00**
celkovy priemer	35.41	98.36	27.24

Tabuľka 3.10: P4 1600 MHz, 512MB RAM, Kubuntu Linux, rýchlosť v percentách najlepšieho.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
hp2.pdf	4.06	33.18	3.71	**100.00**
bib	81.25	**100.00**	81.25	81.25
book1	72.94	97.64	68.89	**100.00**
book2	75.68	**100.00**	71.79	84.85
geo	10.94	43.75	10.45	**100.00**
news	60.00	**100.00**	56.04	85.00
obj1	25.00	**100.00**	33.33	14.29
obj2	48.15	**100.00**	45.61	70.27
paper1	83.33	83.33	83.33	71.43
paper2	75.00	**100.00**	81.82	81.82
progc	75.00	75.00	75.00	60.00
progl	85.71	**100.00**	85.71	42.86
progp	**100.00**	**100.00**	**100.00**	30.77
trans	88.89	88.89	88.89	30.77
random.txt	12.77	33.33	13.33	**100.00**
alphabet.txt	N/A	**100.00**	**100.00**	N/A
alice29.txt	83.33	**100.00**	83.33	95.24
asyoulik.txt	77.27	**100.00**	85.00	**100.00**
cp.html	66.67	**100.00**	66.67	66.67
fields.c	N/A	**100.00**	**100.00**	N/A
grammar.lsp	**100.00**	N/A	**100.00**	N/A
kennedy.xls	14.50	24.11	9.34	**100.00**
lcet10.txt	74.36	**100.00**	68.24	85.29
plrabn12.txt	74.75	97.37	71.15	**100.00**
ptt5	69.23	**100.00**	60.00	0.61
sum	50.00	**100.00**	42.86	50.00
xargs.1	**100.00**	N/A	**100.00**	N/A
celkovy priemer	35.17	**100.00**	30.58	10.99

Tabuľka 3.11: P4 1600 MHz, 512MB RAM, Kubuntu Linux, rýchlosť v percentách najlepšieho.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
bible.txt	716.48	846.20	728.27
e.coli	648.06	658.43	713.49
world192.txt	760.765	**1001.01**	820.45
pi.txt	527.02	625.20	594.38
celkovy priemer	677.30	**765.00**	725.69

Tabuľka 3.12: Centrino 1700 MHz, 1280MB RAM, WinXP, rýchlosť v kB/s.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
bible.txt	663.92	**858.79**	743.36	673.92
e.coli	623.11	645.30	**714.62**	579.21
world192.txt	687.18	984.28	846.33	772.94
pi.txt	473.37	605.43	590.78	**1121.20**
celkovy priemer	631.66	757.07	734.69	671.57

Tabuľka 3.13: Centrino 1700 MHz, 1280MB RAM, WinXp, rýchlosť v percentách najlepšieho.

názov súboru	**UkkList**	**UkkTree**	**UkkStd**
bible.txt	83.43	98.53	84.80
e.coli	90.69	92.14	99.84
world192.txt	76.00	**100.00**	81.96
pi.txt	47.00	55.76	53.01
celkovy priemer	88.54	**100.00**	94.86

Tabuľka 3.14: Centrino 1700 MHz, 1280MB RAM, WinXP, rýchlosť v percentách najlepšieho.

názov súboru	**MccList**	**MccTree**	**MccStd**	**WOTD**
bible.txt	77.31	**100.00**	86.56	78.47
e.coli	87.19	90.30	**100.00**	81.05
world192.txt	68.65	98.33	84.55	77.22
pi.txt	42.22	54.00	52.69	**100.00**
celkovy priemer	82.57	98.96	96.04	87.79

Tabuľka 3.15: Centrino 1700 MHz, 1280MB RAM, WinXp, rýchlosť v kB/s.

Kapitola 4

Taxonómia algoritmov

4.1 Teoretické vlastnosti

4.1.1 Porovnávané algoritmy, teoretické kritéria

Skôr, než začneme s rozdeľovaním algoritmov do kategórií, si určíme kritéria, podľa ktorých budeme taxonómiu vytvárať, ale ukážeme a stručne popíšeme aj algoritmus, ktorý sme pre jeho nepraktické vlastnosti v praktických testoch neuvažovali, avšak ktorý má z teoretického hľadiska a z hľadiska vývoja algoritmov dôležité miesto, *Weinerov* algoritmus.

Weinerov algoritmus pochádza z roku 1973, kedy sa o sufixových stromoch vedelo len málo, vlastne Weiner bol prvým, kto sufixový strom ako štruktúru zadefinoval. Veľmi prirodzená myšlienka, ktorá vedie k stavbe sufixového stromu, je pridávať do štruktúry postupne dlhšie a dlhšie prípony vstupného reťazca. To však bez použitia Ukkonenovho implementačného triku (viď 3.1.1) nutne skazí linearitu algoritmu, pretože sa zakaždým zmení označenie všetkých listov v strome, ktorých je $n - 1$ pri dĺžke vstupu n. Tomuto problému sa autor rozhodol vyhnúť tak, že sa bude spracovávať vstup sprava doľava a pridávať do štruktúry prípony od najkratšej po najdlhšiu. Takto bude každý list v každom kroku predstavovať príponu a meniť sa bude musieť menej často. Toto je základná myšlienka algoritmu, ktorá pri implementácii ale aj teoretickom dokončení vedie k mnohým problémom, a preto nemal veľké praktické využitie. Je to však prvý lineárny algoritmus.

Z teoretického hľadiska nás môžu zaujímať rôzne vlastnosti algoritmov konštruujúcich sufixové stromy. Medzi najdôležitejšie z nich patria pamäťová

náročnosť v najhoršom prípade, amortizovaná časová zložitosť v najhoršom prípade, ďalej nás môžu zaujímať tieto hodnoty v priemernom prípade, online vlastnosť algoritmov, či nutnosť použitia pomocných štruktúr pri výstavbe.

Okrem *Weinerovho* (wrf) algoritmu budeme taxonómiu vytvárať aj na *Ukkonenovom* (Ukk), *McCreightovom* (Mcc) algoritme a na algoritmoch $WOTD$ a TDD.

4.1.2 Rozdelenie algoritmov

Podľa časovej zložitosti môžeme algoritmy rozdeliť nasledovne (n je veľkosť vstupu, $|\Sigma|$ je rádovo menšia než dĺžka vstupného reťazca):

- $\mathcal{O}(n)$: wrf, Ukk, Mcc
- $\mathcal{O}(n^2)$: $WOTD$, TDD

Podľa pamäťovej náročnosti nemôžeme rozdeliť algoritmy do podobných kategórií ako u časovej zložitosti, pretože všetky uvedené algoritmy majú lineárne pamäťové nároky a bola by iba jedna skupina. Pri lineárnej veľkosti sufixového stromu to ale nie je nič prekvapujúce. Môžeme algoritmy zoradiť podľa najhoršieho prípadu ich najlepšej známej implementácie. Pri zoradzovaní používame hodnoty výsledkov uvedených publikácií prevedené na jednotný prípad, kedy predpokladáme 4B pre integer, 4B pre ukazovateľ a 1B pre znak. Vstupný reťazec má pritom dĺžku n bytov.

- Wrf (32n B. [2] ukazuje, že tento algoritmus potrebuje toľko pamäte, ako mcc, avšak ešte o jeden ukazovateľ naviac)
- Mcc, Ukk (32n B. Pôvodné implementácie algoritmov [10], [5] ukazujú, že je potrebných maximálne 8n integerov na celý strom pre vstupný reťazec dĺžky n)
- Mcc (20n B. Mierna úprava pôvodného algoritmu Mcc, ale okrem implementácie hrán ostáva beh algoritmu totožný. [4] ukazuje, že na beh tohto upraveného algoritmu je potrebných maximálne iba 5n integerov pre celý výsledný strom)
- $WOTD$ (12n + 1 B. [3] ukazuje, že výsledná štruktúra potrebuje na reprezentáciu jedného listu 1 a uzlu 2 integere. Keďže spolu je listov vždy n + 1 (zabezpečí to unikátny znak $ na konci reťazca) a uzlov maximálne n, potrebujeme celkom maximálne 3n + 1 integerov na celú štruktúru. Avšak oproti

ostatným algoritmom toto číslo nie je konečná veľkosť spotrebovanej pamäte, pretože počas výstavby stromu spotrebujú pomocné štruktúry ďalších maximálne 10n bytov)

4.2 Praktické vlastnosti

V tomto odstavci ukážeme niektoré praktické vlastnosti algoritmov. Praktické vlastnosti získame tak, že popíšeme výsledky našich testov a porovnáme s predpokladanými teoretickými vlastnosťami. V niektorých prípadoch sa odkážeme aj na výsledky praktických testov iných autorov. Algoritmy zotriedime podľa týchto pozorovaní do skupín.

4.2.1 Pamäťová náročnosť

Čo sa týka pamäťovej náročnosti, môže nás hneď na začiatku zaraziť výsledok nášho testu pre Ukkonenov a McCreightov algoritmus implementovaný pomocou stromu a vylepšeného spojového zoznamu a ich vysoké hodnoty pre pamäťovú náročnosť. Tieto presahujú hodnoty v teoreticky najhoršom prípade. Je to zapríčinené najmä tým, že sme pokusne implementovali tieto algoritmy tak, že vrcholy obsahujú niektoré položky naviac, aby sa dosiahlo zvýšenie rýchlosti. Zrýchlenie sa nám síce dosiahnuť podarilo, avšak cena, ktorú sme za to museli zaplatiť, je vysoká.

Ďalej nás môžu zaujať výsledky algoritmu $WOTD$ a ich nesúlad s teoreticky najhorším prípadom. Podľa sekcie 4.1.2 je najhorší prípad 12n bytov na vstup dĺžky n. Avšak v tabuľke vidíme hodnoty pohybujúce sa okolo 14 bytov na jeden znak vstupu. V tomto prípade však nejde o žiadny trik, ako zlepšiť časovú náročnosť v našej aplikácii, ale tento rozdiel je zapríčinený tým, že autori v [2] nepripočítavajú k veľkosti jedného prvku poľa, ktoré tvorí výsledný strom, pomocné bity tvoriace príznaky pre najpravejšieho syna, list a vyhodnotený uzol. Okrem príznaku pre nevyhodnotený uzol sú všetky tieto informácie potrebné na to, aby sa výsledným stromom dalo prechádzať a teda aby výsledná štruktúra bola použiteľná. Preto sme pri zisťovaní veľkosti výslednej štruktúry počítali s celou touto štruktúrou aj s príznakmi. Veľkosť informácii typu boolean v jazyku C++ je implementačne a platformovo závislá, preto sa nedá presne určiť, akú časť výsledných hodnôt zaberajú práve tieto parametre. Ďalším dôvodom pre tieto nezrovnalosti je aj to, že my sme spomínané príznaky považovali za osobitnú položku prvku poľa, kdežto autori [2] využili niektoré bity z uloženého integeru na príznaky. Aj

napriek tomu sa dá konštatovať, že namerané výsledky približne kopírujú teoretické znalosti.

Podľa zistených výsledkov aplikácie ako aj podľa výsledkov prác [4], [2] je pre ušetrenie miesta potrebného pre výslednú štruktúru najvhodnejší algoritmus $WOTD$. Mcc a Ukk za ním zaostávajú v tomto ohľade nie len v našej implementácii, ale aj v implementácii štandardnej a dokonca aj v pamäťovo vylepšenej implementácii uvedenej v [4]. Vylepšené implementácie teoreticky nezhoršujú rád časovej zložitosti, ani jej konštantu, ale ako ukázali testy, sú v priemere o niečo málo pomalšie než pôvodné implementácie. Algoritmus $WOTD$ má však podľa výsledkov uvedených v [2] rýchlostne mierne navrch a podľa našich výsledkov stále drží krok s konkurenciou, aj keď je v globále o niečo pomalší. Treba mať ale napamäti, že $WOTD$ prináša len mierne úspory čo sa týka pamäte potrebnej na beh algoritmu, keďže používa pomocné štruktúry. Po ich započítaní prináša podobné výsledky ako vylepšené algoritmy uvedené v [4]. Ak teda prax vyžaduje menšiu štruktúru, ktorá je výsledkom behu algoritmu, $WOTD$ je vhodná voľba. Ak však výsledná veľkosť nezaváži, môžeme zvažovať aj iné algoritmy tak, aby vyhovovali iným kritériám. Algoritmus *tdd* založený na algoritme $WOTD$ dokonca optimalizuje využitie pamäte pre príliš veľké súbory. Neposkytne síce žiadne zníženie pamäťových nárokov, avšak ak prax vyžaduje, aby sa znížila pamäť potrebná na beh algoritmu (pretože už jednoducho nevystačí), môže tento algoritmus značne pomôcť v dosiahnutej rýchlosti oproti využitiu systémovej virtuálnej pamäte.

Podľa predchádzajúcich pozorovaní môžeme skúmané algoritmy zotriediť do nasledujúcich kategórií podľa ich vhodnosti v jednotlivých prípadoch (zoradené sú podľa vhodnosti v konkrétnom prípade):

- **potreba malej výslednej štruktúry**: $WOTD$, *tdd*
- **potreba menších pamäťových nárokov pri behu**: *tdd*, (pamäťovo) vylepšený Ukk a Mcc, $WOTD$
- **pamäťovo náročné**: naša vylepšená implementácia Ukk a Mcc, wrf, pôvodná implementácia Ukk a Mcc

4.2.2 Časová náročnosť

Z hľadiska časovej náročnosti sa ako prvá konfrontácia teórie s praxou črtá to, že lineárne algoritmy sú často len rovnako rýchle ako algoritmy s kvadratickou amortizovanou časovou zložitosťou. Neraz však náš praktický test

použitím kvadratického algoritmu skončí oveľa skôr než jeho lineárni konkurenti. Kvadratický algoritmus *tdd* dokonca jednoznačne poráža lineárne algoritmy [8], [9]. Je to ako sme spomínali v sekcii 3.1.3 zapríčinené optimálnym využívaním pamäte cache na dnešných procesoroch a u *tdd* aj optimálnym využívaním operačnej pamäte a diskového prístupu.

Výsledky pre časovú náročnosť implementovaných algoritmov sme už zhrnuli v sekcii 3.4. Podobné konštatovania plynú aj z iných prác. Napr. práca [3] ukazuje, že v priemere je algoritmus $WOTD$ o niečo málo rýchlejší než pôvodná implementácia Mcc, avšak pri miernej úprave implementácie hrán do hashovacej tabuľky sa Mcc stáva podstatne rýchlejším. My sme ukázali, že aj inou úpravou algoritmov ako Mcc a Ukk sa dá dosiahnuť to, že kvadratický algoritmus za nimi zaostáva. Cenou za to je ale nepomerné zhoršenie pamäťovej náročnosti. Podobne naše výsledky ukazujú to, čo sa v práci [2] teoreticky očakáva a teda, že Ukkonenov algoritmus je v priemere pomalší než McCreightov. Teoreticky je to celkom prirodzené, kedže sa dá ukázať, že McCreightov algoritmus je vlastne vylepšením Ukkonenovho tým spôsobom, že odpadajú kroky, ktoré do vytváranej štruktúry nič nepridávajú. Mcc má tiež svoju nevýhodu, keď uskutočňuje zbytočné scany, avšak výsledky našej práce ukazujú, že táto strata je aspoň v priemere menej náročná než strata Ukk.

V práci [4] prichádzajú autori s nápadom, ako využitím niekoľkých trikov pri implementácii Mcc odstrániť pamäťové nároky a neznížiť tým teoretickú rýchlosť behu algoritmu. Z výsledkov testov priložených k tejto práci ale vyplýva, že reálne sa beh algoritmov o niečo málo spomalí, rádovo je to v priemere 2 - 5%. Opačný prípad sme mohli zaznamenať našou implementáciou Mcc a Ukk, kde síce rýchlosť mierne porástla, avšak pamäťové nároky stúpli rádovo o desiatky percent. Rýchlosť Mcc porástla voči štandardnej implementácii s využitím spojového zoznamu, pri využití binárneho stromu sme dokonca dosiahli oveľa lepšie výsledky čo sa týka rýchlosti.

Z oboch pohľadov ale vyplýva, že pri implementácii každého algoritmu sa stále treba dobre rozhodnúť, kde sú priority vyvíjanej aplikácie a na čo klásť dôraz, pretože ak neprichádza do úvahy úplne odlišná základná myšlienka algoritmu (ako napr. pri prechode z Mcc na $WOTD$, kedy znížime pamäťovú náročnosť, avšak rýchlosť prakticky ostáva, pričom princíp ich práce je od základu odlišný), úpravou konkrétneho algoritmu dostaneme spravidla zrýchlenie za cenu zvýšenia pamäťových nárokov, alebo zníženie pamäťovej náročnosti za cenu spomalenia behu algoritmu. Toto konštatovanie platí skôr v priemere, pretože u konkrétnych vstupov sa stále môže nájsť

taký, ktorý nahráva tej ktorej implementácii vo všetkých ohľadoch (viď napr. výsledok *pi.txt*, kedy v priemere najhorší algoritmus vyhráva a jeho pamäťové nároky sú stále malé).

Ďalším aspektom, ktorý treba pri navrhovaní implementácie zvážiť, je veľkosť vstupných dát. Pri presiahnutí veľkosti operačnej pamäte nás síce čiastočne zachráni virtuálna pamäť, avšak beh algoritmu sa značne spomalí. Tento problém sa dá riešiť, ako je nám doposiaľ známe, len algoritmom *tdd*, ktorý je síce založený na algoritme $WOTD$, avšak dá sa povedať, že jeho základnou myšlienkou je rozdelenie pamäte podľa veľkosti vstupu tak, aby výsledný beh bol čo najrýchlejší. Tým sa vlastne prepisuje systémové správanie virtuálnej pamäte. Ako vidno z výsledkov prác [8] a [9], rýchlosť ostáva približne rovnaká s rastúcou veľkosťou vstupu aj napriek niekoľkonásobnému prekročeniu veľkosti operačnej pamäte, ktorá by bola normálne potrebná na beh algoritmu.

Z uvedených hodnotení môžeme rozdeliť algoritmy z hľadiska praktickej časovej náročnosti na tieto prípady ich vhodného použitia:

• **bez pamäťových obmedzení**: resp. sa tým myslí, že predpokladaná veľkosť vstupu je v porovnaní s dostupnou pamäťou tak malá, že sa pamäťovými nárokmi netreba zaoberať. Je to napríklad pri textovej kompresii, kedy si stanovíme veľkosť okienka textu, pre ktoré strom budeme stavať, čo spravidla býva niekoľko MB [7]. Vtedy je vhodné použiť čo najrýchlejšiu implementáciu algoritmu Mcc bez vylepšení na pamäťovú náročnosť, dokonca by možno stálo za zváženie obetovať nejakú pamäť na úkor rýchlosti ako v našom algorimte MccTree. Keďže $WOTD$, *tdd* a Ukk dosahujú prinajlepšom rovnaké časové výsledky, je možné ich taktiež použiť, avšak bolo by to zbytočné.

• **operačná pamäť postačuje, avšak nie v každom prípade**: je to prípad, kedy aplikácia má za úlohu indexovať bežný text, napr. internetové stránky, alebo elektronické knihy, kedy sa môže vyskytnúť väčší súbor, avšak len zriedka až taký veľký ako napr. ľudský genóm. Vtedy je vhodné zvoliť pamäťovo menej náročný algoritmus, prípadne jeho úpravu, ktorá pamäťové nároky znižuje, pretože tak obmedzíme možnosť, že pamäť dôjde a využitie systémovej virtuálnej pamäte spomalí beh algoritmu (napríklad pri ľudsky čitateľnom texte môžeme stále natrafiť na obrovské slovníky). Vhodné sú algoritmy $WOTD$ a pamäťovo šetrné úpravy Mcc, prípadne Ukk. *tdd* by sa použiť dal, avšak jeho implementácia je v porovnaní s predchádzajúcimi značne obtiažnejšia a jeho výhody by sa prejavili len málo, alebo vôbec.

• **operačná pamäť značne nepostačuje**: je to prípad veľkých vstupov, najmä z genetického výskumu, kedy sa pri kódovaní DNA často sufixové stromy využívajú pre hľadanie zhodných genetických znakov a čŕt [6], alebo aj pri veľkých komplexných jazykových slovníkoch. Vtedy je najvhodnejšie použiť algoritmus *tdd*, ostatné neprichádzajú do úvahy, pretože by bežali veľmi dlho. V práci [8] dokonca autori uvádzajú, že to bol jediný algoritmus, ktorý vôbec dokázal indexovať celý ľudský genóm.

Kapitola 5

Záver

V našej práci sme zhodnotili teoretické a praktické vlastnosti niektorých základných algoritmov na konštrukciu sufixového stromu. V dnešnej dobe je čoraz dôležitejšie túto štruktúru, ako aj jej konštrukčné algoritmy podrobne skúmať. Preto sme aj niektoré algoritmy implementovali, aby sme takto prakticky mohli zistiť ich silné a slabé stránky.

Za pomoci vyvinutej aplikácie a naštudovaných materiálov sa nám podarilo algoritmy rozdeliť do niekoľkých teoretických skupín a praktických kategórií ich použitia. Implementovali a skúmali sme síce iba základné algoritmy, ale venovali sme sa aj ich úpravami popísanými v iných publikáciách a taktiež sme sa pokúsili o vlastnú úpravu niektorých algoritmov. Toto všetko napomohlo vytvoreniu základného prehľadu v algoritmoch konštruujúcich sufixové stromy a taxonómie poskytnutej v kapitole 4.

Pre budúcnosť by bolo zaujímavé pozorovať a prípadne implementovať podrobnejšie a detailnejšie zmeny v algoritmoch a zistiť ďalšie súvislosti s ich efektivitou. Ďalším krokom by mohlo byť aj spoznanie dátových štruktúr príbuzných sufixovému stromu a pozorovať výhody, ktoré prinášajú ich konštrukčné algoritmy v konfrontácii s obmedzeniami využitia vytvorených štruktúr.

Sufixové stromy a im príbuzné štruktúry ako aj algoritmy ich konštrukcie predstavujú obrovskú možnosť ďalšieho výskumu a ich využitie si zaslúži v dnešnej dobe veľkú pozornosť informatikov. Táto práca slúži ako oboznámenie s ich základmi a ako náčrt ďalších možností. Túto úlohu spĺňa, ale je zároveň výzvou pre ďalšie štúdium.

Kapitola 6

Obsah CD

Tu bude obsah prilozeneho CD.

Literatúra

[1] R. Cole, R. Hariharan (2000): *Faster Suffix Tree Construction with Missing Suffix Links*, 32nd Annual ACM Symposium on Theory of Computing, pp. 407–415.

[2] R. Giegerich, Stefan Kurtz (1997): *From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction*

[3] R. Giegerich, S. Kurtz, J. Stoye(1999): *Efficient Implementation of Lazy Suffix Trees.* In Proceedings of the Third Workshop on Algorithm Engineering.

[4] S. Kurtz (1999): *Reducing the space requirement of suffix trees*, Softw., Pract. Exper. 29(13) pp1149-1171.

[5] E. M. McCreight (1976): *A Space-Economical Suffix Tree Construction Algorithm*, Jrnl. of Algorithms, 23(2) pp262-272.

[6] (2007) http://mummer.sourceforge.net/

[7] M. Senft (2003): *Bezztrátová komprese dat pomocí sufixových stromů*, diplomová práca na MFF UK.

[8] Sandeep Tata, Richard A. Hankins, and Jignesh M.Patel (2004): *Practical Suffix Tree Construction*, VLDB, Toronto, Canada. (http://www.eecs.umich.edu/tdd/)

[9] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, Jignesh M. Patel (2005): *Practical Methods for Constructing Suffix Trees*, VLDB Journal 14(3), pp 281-299.

[10] E. Ukkonen (1992): *Constructing Suffix Trees On-Line in Linear Time*, In Algorithms, Software, Architecture, J.v.Leeuwen (ed), vol#1 of Information Processing 92, Proc. IFIP 12th World Computer Congress, Madrid, Spain, Elsevier Sci. Publ., pp484-492.

[11] E. Ukkonen (1995): *On-line Construction of Suffix Trees*, Algorithmica, 14(3) pp249-260.

[12] P. Weiner (1973): *Linear Pattern Matching Algorithms*, Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp1-11.

[13] http://www.codeproject.com/cs/algorithms/countsort.asp (2007)