

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Vizualizácia algoritmov a dátových štruktúr

Bakalárska práca

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Vizualizácia algoritmov a dátových štruktúr

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky FMFI
Vedúci práce: Mgr. Jakub Kováč

Pod'akovanie

Tu si nemôžem vyliat' dušu na celý svet, ale len stroho poďakovať za „odbornú“ pomoc.

Abstrakt

Táto práca sa zaoberá vizualizáciou dátových štruktúr. Nadväzuje na bakalársku prácu Jakuba Kováča (2007) a rozvíja kompiláciu o ďalšie dátové štruktúry. V prvej časti sú tieto dátové štruktúry, konkrétne union-find, lexikografický strom a sufixový strom, definované a konkrétne algoritmy popísané. V druhej časti sa pojednáva samotná vizualizácia a popisuje sa implementácia. Práca je prehľadom dátových štruktúr s priloženým Java appletom na CD ako pomôckou pri vizualizácii.

Kľúčové slová: vizualizácia, ADŠ, algoritmy a dátové štruktúry, union-find, stringológia, lexikografický strom, trie, suffix tree.

Abstract

Here should be an abstract in English language.

Predhovor

Toto je zaujímavá časť, ale ešte som ju u nikoho nevidel, tak asi pôjde preč. Tu si veľmi vylejem dušu na celý svet. :) Zatiaľ tu patrí poďakovanie najmä Marekovi Zemanovi, Michalovi Forišekovi, Kubovi Kováčovi a všetkým učiteľom, ktorí mi pomohli nadobunúť vedomosti, aby som mohol vypracovať túto prácu.

Obsah

Úvod	1
1 Union-find problém	3
1.1 Použitie	3
1.2 Popis	3
1.3 Heuristika na spájanie	4
1.4 Heuristiky na kompresiu cesty	4
1.5 Vizualizácia	6
2 Písmenkový strom	7
2.1 Popis	7
2.2 Použitie	8
2.3 Vizualizácia	9
3 Sufixový strom	11
3.1 Zostrojenie sufixového stromu	11
3.2 Ukkonenov algoritmus	11
3.2.1 Sufixové linky	12
3.2.2 Zníženie nárokov na pamäť	12
3.2.3 Rozširovanie stromu	12
3.2.4 Zhrnutie	13
4 Popis softvéru	17
5 Implementácia softvéru	19
Záver	21
Literatúra	23

Úvod

Dátové štruktúry a algoritmy tvoria základnú, prvotnú časť výučby informatiky. Vizualizácia algoritmov a dátových štruktúr je grafické znázornenie, ktoré abstrahuje od implementačných detailov a reprezentácie v pamäti. Je teda vhodnou pomôckou pri výučbe i samoštúdiu.

Ukážka nášho softvéru *Gnarley Trees* je na obrázku 1. Tento projekt začal ako bakalárska práca Jakuba Kováča (KOVÁČ, 2007); v tejto práci popisujeme nové dátové štruktúry, ktoré sme vizualizovali a nové funkcie a vylepšenia, ktoré sme doplnili.

Konkrétne ide o *union-find problém*, *písmenkový strom (trie)* a *sufixový strom*.

Okrem vizualizácie softvér prerábame a neustále vylepšujeme. Doplnili sme ho o históriu krokov a operácií, jednoduchšie ovládanie a veľa ďalších funkcií. Softvér je celý v slovenčine aj angličtine a je implementovaný v jazyku JAVA. Dostupný je na stránke <http://people.ksp.sk/~kuko/gnarley-trees> vo forme appletov s jednotlivými dátovými štruktúrami, a tiež vo forme samostatného programu, ktorý obsahuje všetky dátové štruktúry a je určený na používanie offline.

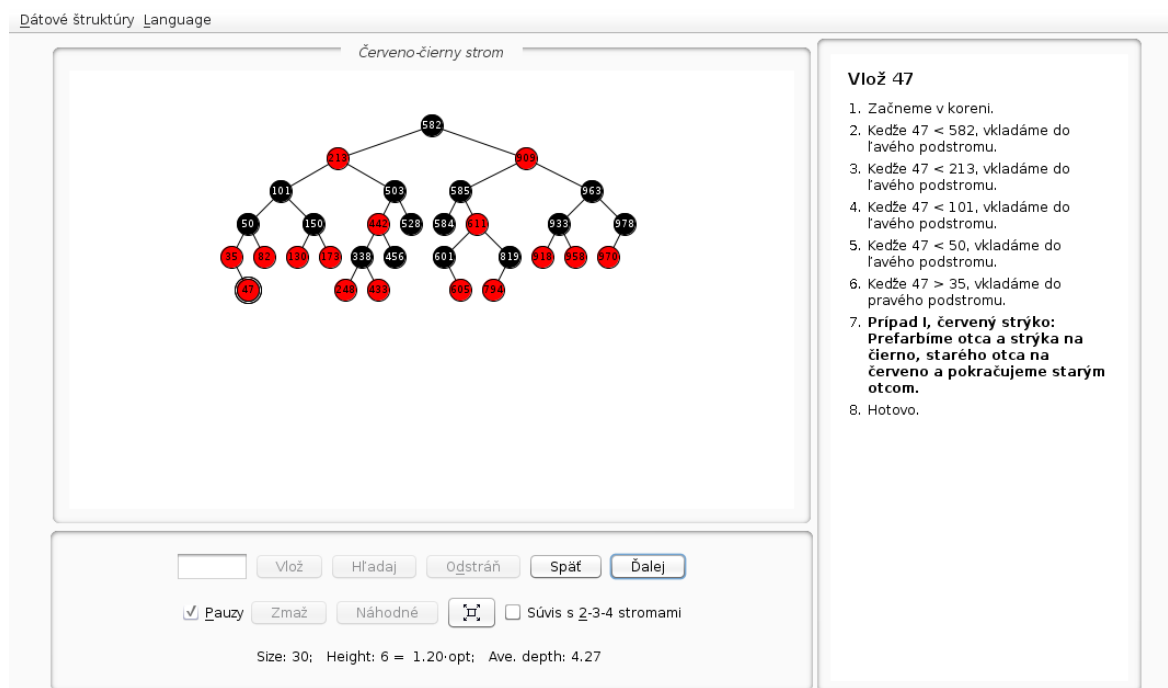
Našou snahou je vytvoriť kvalitný softvér nezávislý od operačného systému, ktorý bude vyhovovať ako pomôcka pri výučbe ako aj pri samoštúdiu a bude voľne prístupný.

Predchádzajúci výskum v oblasti pedagogiky zatiaľ nedokázal úplne preukázať pedagogickú efektívnosť vizualizácií (SHAFFER; COOPER; ALON a spol., 2010), avšak viacero štúdií potvrdilo zvýšený záujem a zapojenosť študentov (NAPS; RÖßLING; ALMSTRUM a spol., 2002; C.D. HUNDHAUSEN; DOUGLAS; STASKO, 2002).

Rozmach vizualizácie algoritmov priniesla najmä JAVA a jej fungovanie bez viazanosti na konkrétny operačný systém. Kvalita iných existujúcich vizualizácií sa líši a keďže ide o ľahko naprogramovateľné programy, je ich veľa a sú pomerne nekvalitné (SHAFFER; COOPER; ALON a spol., 2010). Zbieraním a analyzovaním kvality sa venuje skupina AlgoViz (<http://algoviz.org/>).

Zaujímavé je pozorovanie, že určovanie si vlastného tempa pri vizualizácii je veľká pomôcka. Naopak, ukazovanie pseudokódu alebo nemožnosť určenia si vlastného tempa (napríklad animácia bez možnosti pozastavenia), takmer žiadne zlepšenie neprináša (SHAFFER; COOPER; ALON a spol., 2010; SARAIYA; SHAFFER; MCCRICKARD; NORTH, 2004).

Prácu sme rozdelili na popis jednotlivých dátových štruktúr, ktoré sme vizualizovali a popis samotného softvéru. V kapitolách 1, 2, 3 je postupne popis *union-find*, *písmenkového stromu* a *sufixového stromu*. V kapitole 5 je popis softvéru.



Obrázok 1: *Softvér Gnarley Trees*. V ovládacom paneli dole môže užívateľ zvoliť operáciu a vstupnú hodnotu a sledovať priebeh algoritmu (momentálne vkladanie prvku 47). Užívateľ postupuje vlastným tempom pomocou tlačidiel „Ďalej“ alebo „Späť“. Vpravo je popis vykonávaných krokov; kliknutím na konkrétny krok v histórii sa môže užívateľ vrátiť.

Kapitola 1

Union-find problém

V niektorých aplikáciach potrebujeme udržiavať prvky rozdelené do skupín (disjunktných množín), pričom skupiny sa môžu zlučovať a my potrebujeme pre daný prvok efektívne zistiť, do ktorej skupiny patrí. Predpokladáme, že každá množina S je jednoznačne určená jedným svojim zástupcom $x \in S$ a potrebujeme implementovať nasledovné tri operácie:

- $makeSet(x)$ – vytvorí novú množinu $S = \{x\}$ s jedným prvkom;
- $union(x, y)$ – ak x, y sú zástupcovia množín S a T , $union$ vytvorí novú množinu $S \cup T$, pričom S aj T zmaže. Zástupcom novej množiny $S \cup T$ je x alebo y .
- $find(w)$ – nájde zástupcu množiny, v ktorej sa prvok x nachádza.

1.1 Použitie

Union-find sa dá použiť na reprezentáciu neorientovaného grafu, do ktorého pridávame hrany a odpovedáme na otázku: „Sú dané dva vrcholy spojené nejakou cestou?“ (t. j. sú v rovnakom komponente súvislosti?) Medzi najznámejšie aplikácie patria Kruskalov algoritmus na nájdenie najlacnejšej kostry (KRUSKAL, 1956) a unifikácia (KNIGHT, 1989).

GILBERT; NG; PEYTON, (1994) ukázali, ako sa dá union-find použiť pri Choleského dekompozícií riedkych matic. Autori navrhli efektívny algoritmus, ktorý zistí počet nenulových prvkov v každom riadku a stĺpci výslednej matice, čo slúži na efektívnu alokáciu pamäte.

Pre offline verziu úlohy, kde sú všetky operácie dopredu známe, GABOW; R. TARJAN, (1985) navrhli lineárny algoritmus. Článok obsahuje tiež viacero aplikácií v teoretickej informatike.

1.2 Popis

Dátová štruktúra *union-find* sa reprezentuje ako les, kde každý strom zodpovedá jednej množine a korene stromov sú zástupcovia množín. Pri implementácii si stačí pre každý prvok

x udržiavať smerník $p(x)$ na jeho otca (pre koreň je $p(x) = \text{NULL}$).

Operácia $\text{makeset}(x)$ teda vytvorí nový prvok x a nastaví $p(x) = \text{NULL}$.

Operáciu $\text{find}(w)$ vykonáme tak, že budeme sledovať cestu po smerníkoch, až kým ne-nájdeme zástupcu.

Operáciu $\text{union}(x, y)$ ide najjednoduchšie vykonať tak, že presmerujeme smerník $p(y)$ na prvok x , teda $p(y) \leftarrow x$. Môžeme ľahko pozorovať, že takýto *naivný* spôsob je neefektívny, lebo nám operácia $\text{find}(w)$ v najhoršom prípade, na n prvkoch, trvá $\Omega(n)$ krokov.

Existujú dva prístupy ako zlepšiť operácie a tým aj zrýchliť ich vykonanie. Sú to: heuristika *spájanie podľa ranku* a rôzne heuristiky na *kompresiu cesty*.

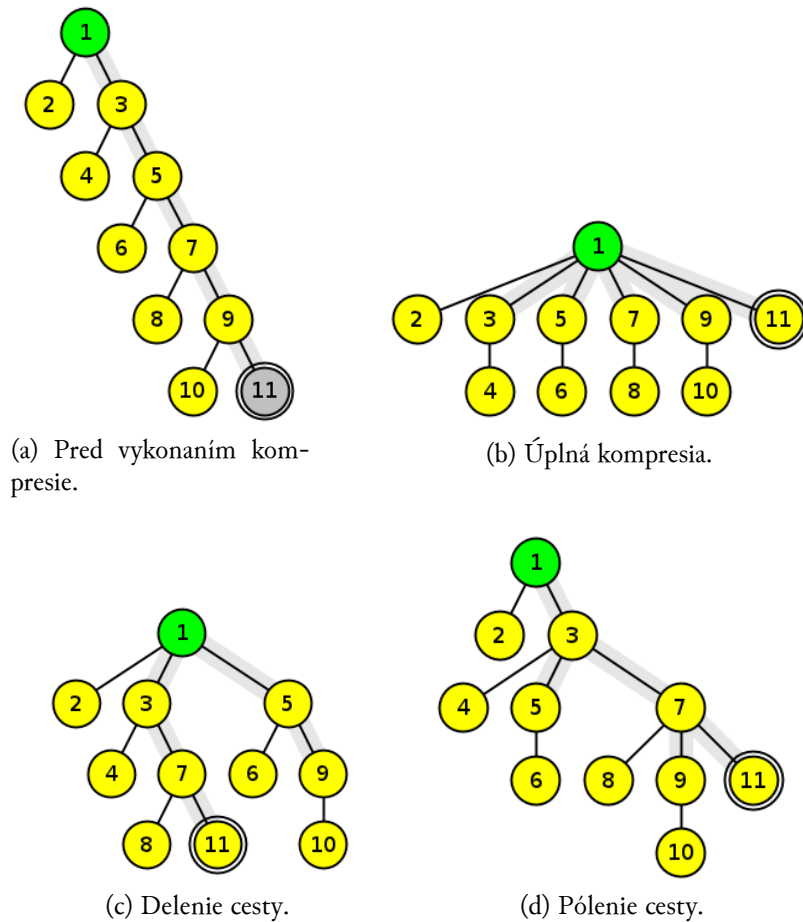
1.3 Heuristika na spájanie

Prvá heuristika pre každý vrchol x udržiava hodnotu $\text{rank}(x)$, ktorá určuje najväčšiu možnú hĺbku podstromu s koreňom x . Pri operácii $\text{makeset}(x)$ zadefinujeme $\text{rank}(x) = 1$. Pri operácii $\text{union}(x, y)$ porovnáme $\text{rank}(x)$ a $\text{rank}(y)$ a vždy napojíme strom s menším rankom pod strom s väčším rankom. Ak majú oba stromy rovnaký rank, napojíme povedzme x pod y a $\text{rank}(y)$ zvýšime o 1.

1.4 Heuristiky na kompresiu cesty

Druhou heuristikou je kompresia cesty. Algoritmov na efektívnu kompresiu cesty je veľa (R. E. TARJAN; Jan van LEEUWEN, 1984), tu popíšeme tie najefektívnejšie. Prvou z nich je *úplná kompresia* (HOPCROFT; ULLMAN, 1973). Pri vykonávaní operácie $\text{find}(w)$, po tom, ako nájdeme zástupcu, napojíme všetky vrcholy po ceste priamo pod koreň (obr. 1.1b). Toto síce trochu spomalí prvé hľadanie, ale výrazne zrýchli ďalšie hľadania pre všetky prvky na ceste ku koreňu. Druhou heuristikou je *delenie cesty* (J. LEEUWEN; WEIDE, 1977). Pri vykonávaní operácie $\text{find}(w)$ pripojíme každý vrchol v ceste od vrcholu x po koreň stromu na otca jeho otca (obr. 1.1c). Tretou heuristikou je *pólenie cesty* (J. LEEUWEN; WEIDE, 1977). Pri vykonávaní operácie $\text{find}(w)$ pripojíme každý druhý vrchol v ceste od vrcholu x po koreň stromu na otca jeho otca (obr. 1.1d).

Časová zložitosť union-findu závisí od toho, koľko prvkov je v množinách a koľko je operácií celkovo vykonaných operácií. Všetky uvedené spôsoby ako vykonať operáciu $\text{find}(w)$ sa dajú použiť s oboma realizáciami operácie $\text{union}(x, y)$. Počet prvkov označme n a počet operácií m . V praxi je zvyčajne počet operácií oveľa väčší ako počet prvkov. Pri tomto predpoklade ($m \geq n$) je pri použití spájania podľa ranku časová zložitosť pre algoritmus bez kompresie $\Theta(m \log n)$ a pre všetky tri uvedené typy kompresie $\Theta(m \alpha(m, n))$. V tabuľke 1.1 je porovnanie časových zložítostí (R. E. TARJAN; Jan van LEEUWEN, 1984).



Obrázok 1.1: *Kompresia cesty z vrcholu 11 do koreňa.* Cesta je vyznačená šedou. Pri úplnej kompresii (b) sa všetky vrcholy napoja na zástupcu. Pri delení cesty (c) a pólení cesty (d) sa cesta skráti približne na polovicu.

	Naivné spájanie	Spájanie podľa ranku
Naivné hľadanie	$\Theta(mn)$	$\Theta(m \log n)$
Úplná kompresia, delenie cesty, pólenie cesty	$\Theta\left(m \log_{1+m/n} n\right)$	$\Theta(m\alpha(m, n))$

(a) Prehľad časových zložítostí, ak $m \geq n$.

	Naivné spájanie	Spájanie podľa ranku
Naivné hľadanie	$\Theta(mn)$	$\Theta(n + m \log n)$
Úplná kompresia	$\Theta(n + m \log n)$	$\Theta(n + m\alpha(n, n))$
Delenie cesty	$\Theta(n \log m)$	$\Theta(n + m\alpha(n, n))$
Pólenie cesty	$\Omega(n + m \log n),$ $O(n \log m)$	$\Theta(n + m\alpha(n, n))$

(b) Prehľad časových zložítostí, ak $m < n$.

Tabuľka 1.1: Porovnanie časových zložítostí pre rôzne kombinácie hľadání prvkov a spájání množín pre union-find. Počet prvkov je n a počet operácií je m . Ackermannova inverzná funkcia je označená ako α . V praxi zväčša platí, že $m > n$.

1.5 Vizualizácia

Dátovú štruktúru union-find sme vizualizovali ako les. Pre názorné oddelenie množín sme si zvolili pravidlo, ktoré zakazovalo vykresliť vrchol z iného stromu (prvok inej množiny) napravo od najľavejšieho vrcholu a naľavo od napravejšieho vrcholu inej množiny. Jednotlivé množiny sme už vykreslovali tesným Walkerovým algoritmom (WALKER II, 1990). Vizualizácia poskytuje všetky vyššie spomínané heuristiky a aj tlačidlo na vykonanie viacerých náhodných spojení naraz. Toto je užitočné, keď chce užívateľ vidieť, ako dátová štruktúra vyzerá, po vykonaní veľa operácií.

Kapitola 2

Písmenkový strom

Písmenkový strom reprezentuje množinu slov. Oproti binárnym vyhľadávacím stromom je hlavný rozdiel v tom, že kľúče nie sú uložené vo vrcholoch, ale samotná poloha v strome určuje kľúč (slovo).

2.1 Popis

Písmenkový strom je *zakorenený strom*, v ktorom každá hrana obsahuje práve jeden znak z abecedy alebo *ukončovaci znak*. Teda, každá cesta z koreňa do listu so znakmi $w_1, w_2, \dots, w_n, \$$ prirodzene zodpovedá slovu $w = w_1 w_2 \dots w_n$. *Ukončovaci znak* je ľubovoľný, dopredu dohodnutý symbol, ktorý sa v abecede nenachádza (napr. \$).

Písmenkový strom je *asociatívne pole (slovník)*, čiže poskytuje tieto tri operácie:

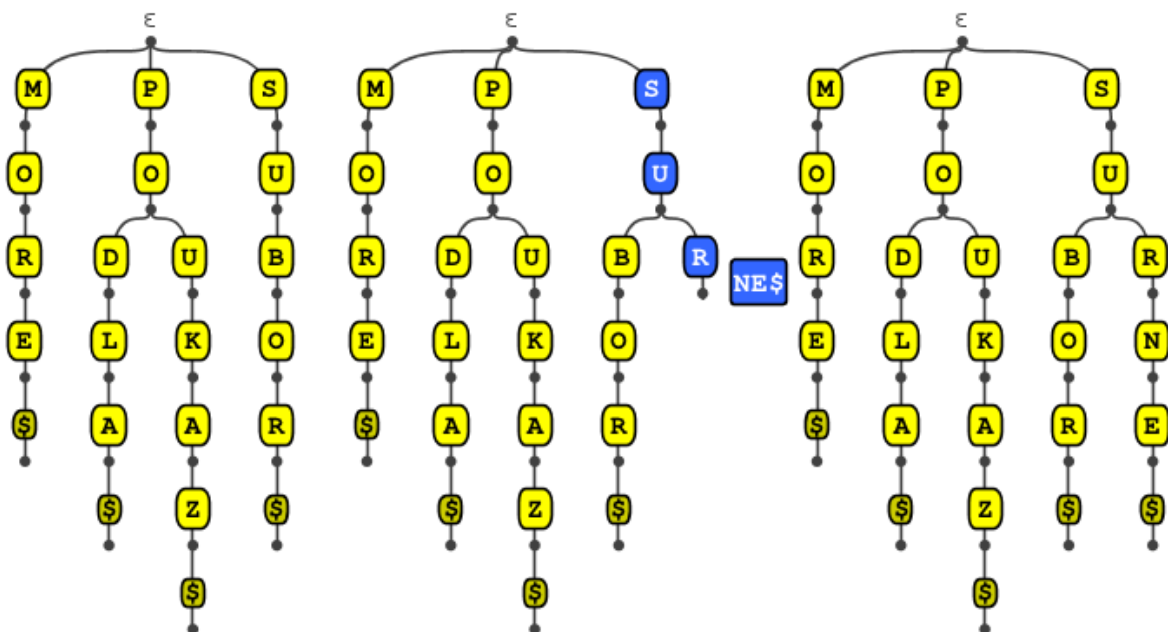
- $insert(w)$ – pridá do stromu slovo w ;
- $find(w)$ – zistí, či sa v strome slovo w nachádza;
- $delete(w)$ – odstráni zo stromu slovo w .

Všetky operácie začínajú v koreni a ku slovu pridávajú ukončovaci znak, teda pracujú s reťazcom $w\$$.

Operácia $insert(w)$ vloží do stromu vstupný reťazec tak, že z reťazca číta znaky a prechádza po príslušných hranách. Ak hrana s daným symbolom neexistuje, pridá ju (pozri obr. 2.1).

Operácia $find(w)$ sa spustí z koreňa podľa postupnosti znakov. Ak hrana, po ktorej sa má spustiť neexistuje, dané slovo sa v strome nenachádza. Ak prečíta celý vstupný reťazec, dané slovo sa v strome nachádza.

Operácia $delete(w)$ najprv pomocou operácie $find(w)$ zistí umiestnenie slova. Ak sa slovo v strome nachádza, algoritmus odstráni hranu s ukončovacím znakom a vrchol, ktorý bol na nej zavesený. V tomto štádiu sa nám môže stať, že v strome ostane *mŕtva vetva* – nie je ukončená ukončovacím znakom. Pre fungovanie stromu to nevadí, všetky operácie by



Obrázok 2.1: Vloženie slova „SURNE“. Začiatok slova „SU“ sa v strome nachádza, ešte treba pripojiť hrany so znakmi R, N, E a \$.

prebiehali správne, ale takto štruktúra zaberá zbytočne veľa miesta. Preto je dobré túto mŕtvu vetvu odstrániť (pozri obr. 2.2).

Všetky tri operácie majú časovú zložitosť $O(|w|)$, kde $|w|$ je dĺžka slova.

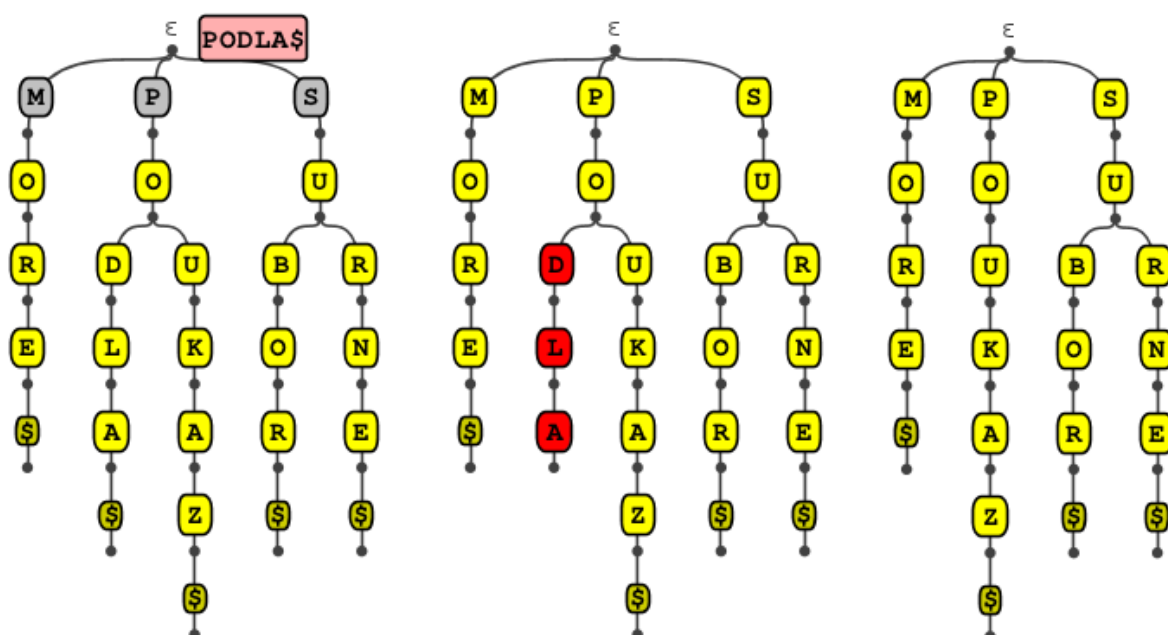
2.2 Použitie

Prvýkrát navrhol písmenkový strom FREDKIN, (1960), ktorý používal názov *trie memory*¹, keďže išlo o spôsob udržiavania dát v pamäti. MORRISON, (1968) navrhol písmenkový strom, v ktorom sa každá cesta bez vetvení skomprimuje do jedinej hrany (na hranách potom nie sú znaky, ale slová). Táto štruktúra je známa pod menom *PATRICIA* (tiež *radix tree*, resp. *radix trie*) a využíva sa napríklad v *routovacích tabuľkách* (SKLOWER, 1991).

Písmenkový strom (tzv. *packed trie* alebo *hash trie*) sa používa napríklad v programe \TeX na slabikovanie slov (LIANG, 1983). Pôvodný návrh (FREDKIN, 1960) ako uložiť trie do pamäte zaberá príliš veľa nevyužitého priestoru. LIANG, (1983) však navrhol, ako tieto nároky zmenšiť.

Písmenkové stromy sa podobajú na *konečné automaty*. Vznikli rôzne modifikácie stromov na automaty, ktorých hlavnou výhodou je, že v komprimovanej podobe spájajú nielen predpony, ale aj prípony slov a teda v slovách ľudských jazykov výrazne znižujú pamäťový priestor potrebný na uchovanie dátovej štruktúry. Vďaka tomu sa využívajú na jazykovú korekciu, automatické dopĺňanie slov a podobne (APPEL; JACOBSON, 1988; Claudio L. LUCCHESI; Claudio L. LUCCHESI; KOWALTOWSKI, 1992).

¹Z anglického *retrieval* – získanie.



Obrázok 2.2: Odstránenie slova „PODLA“. Po odstránení \$ nám v strome ostane nepotrebná prípona „DLA“ (mŕtva vetva), ktorá je vyznačená červenou.

Ďalšie použitie písmenkového stromu je pri triedení zoznamu slov. Všetky slová sa pridajú do stromu a potom sa spraví *preorderový prechod* stromu. Túto myšlienku spracovali Ranjan SINHA; Justin ZOBEL, (2004) a veľmi výrazne zrýchlil triedenie dlhých zoznamov slov. Neskôr tento algoritmus vylepšili R. SINHA; RING; J. ZOBEL, (2006). Kvôli tomu, ako algoritmus pracuje, sa nazýva *burstsor*.

2.3 Vizualizácia

Pri vizualizácii písmenkového stromu sme použili Walkerov algoritmus pre úsporné rozloženie vrcholov v strome (WALKER II, 1990). Keď má vrchol viacej synov a hrany kreslíme priamo, tak vzniká nedostatok priestoru pre umiestnenie znakov na hrany. Preto sme sa rozhodli kresliť hrany zakrivené, podľa Bézierovej krivky určenej štyrmi bodmi. Vo vizualizácii sa dajú vložiť náhodné slová podľa momentálne nastaveného jazyka. Taktiež sa automaticky odstraňuje diakritika a interpunkcia, takže sa dá naraz vložiť súvislý text.

Kapitola 3

Sufixový strom

Sufixový strom je písmenkový strom pre všetky *sufixy* (*prípomy*) daného slova. Jeho veľká výhoda spočíva v rýchlom zostrojení a veľkom množstve efektívnych algoritmov na ňom. V softvéri sme vizualizovali jeho zostrojenie, a preto ho aj tu popíšeme.

3.1 Zostrojenie sufixového stromu

Existuje veľa spôsobov ako zostrojiť sufixový strom. Najjednoduchšie riešenie je využiť písmenkový strom a pridať do neho všetky sufixy. Takéto riešenie má časovú aj pamäťovú zložitosť $O(n^2)$.

UKKONEN, (1995) navrhol, ako zostrojiť strom rýchlejšie a za behu.

3.2 Ukkonenov algoritmus

Hlavnou myšlienkou algoritmu je pre slovo $w_1w_2w_3 \cdots w_n\$$ postupne vytvoriť suffixové stromy pre slová $w_1, w_1w_2, \dots, w_1w_2w_3 \cdots w_n\$$. Označme tieto stromy $T(1), T(2), \dots, T(n), T$. Teda, vytvárame n stromov a každý zatiaľ na vytvorenie potrebuje $O(n^2)$ krokov. Toto je na prvý pohľad veľmi nešikovné riešenie pretože počet krokov potrebných na zostrojenie všetkých stromov je $O(n^3)$. Avšak zopár vylepšeniami je možné tento algoritmus zrýchliť až na hranicu $O(n)$ a to aj pre rýchlosť, aj pre pamäť.

Môžeme si všimnúť, že keď už máme vytvorený strom $T(i-1)$ (ktorý obsahuje sufixy $w_1 \cdots w_{i-1}, w_2 \cdots w_{i-1}, \dots, w_{i-1}$), tak pre vytvorenie stromu $T(i)$ netreba znovu vytvárať nový strom a pridávať do neho sufixy $w_1 \cdots w_{i-1}w_i, w_2 \cdots w_{i-1}w_i, \dots, w_{i-1}w_i$, ale stačí rozšíriť existujúce sufixy o znak w_i .

Druhým dobrým pozorovaním je fakt, že keď máme znak a , slovo v a v strome sa nachádza sufix av , tak sa v strome určite nachádza aj slovo v . Vyplýva to zo základnej vlastnosti sufixových stromov – sufixový strom obsahuje všetky sufixy.

3.2.1 Suffixové linky

Pri vytváraní stromu $T(i)$ zo stromu $T(i - 1)$ postupne rozširujeme sufíxy, no bolo by neefektívne ich vždy vyhľadávať z koreňa. Preto zavedieme *suffixový link* pre každý vrchol okrem koreňa, ten suffixový link nemá. Pre vrchol končiaci sufíx av to je vrchol končiaci sufíx v .

Algoritmus teda zmeníme tak, že namiesto opätovného vyhľadávania sufíxu z koreňa, vyhľadáme bod, z ktorého začneme pridávať len raz a ďalej sa navigujeme po linkoch. Po pridaní vrchola, nazvime ho u_i , do stromu prejdeme suffixovým linkom na miesto, kde opäť pripojíme vrchol u_{i+1} . Vytvoríme suffixový link pre vrchol u_i – bude to smerník na u_{i+1} .

Ako nám to pomohlo zlepšiť časovú zložitosť? Teraz nám stačí pre každý z n stromov nájsť sufíx, kde začneme rozširovať strom. Ten nájdeme na $O(n)$ krokov. Rozšírenie stromu o jeden znak zaberie tiež $O(n)$ krokov. Dokopy venujeme jednému stromu $O(n)$ krokov a n stromom venujeme $O(n^2)$.

Ďalším zlepšením bude pamäťová optimalizácia.

3.2.2 Zníženie nárokov na pamäť

Veľmi priamočiarym krokom k zlepšeniu pamätevej náročnosti sa zdá byť kompresia hrán. Po nej na hranách nie je len jeden znak, ale celý reťazec znakov. Takto má celý strom dovedna $O(n)$ hrán. Dôvod, prečo ju uvádzam až tu, je jednoduchý. Prichádza prirodzene s druhým vylepšením. Namiesto toho, aby sme si na hrane pamätali celý reťazec, budeme si na nej pamätať len počiatočnú a konečnú pozíciu v slove, pre ktorý beží algoritmus¹. Po týchto zlepšeniach nám stačí pamätať si $O(n)$ informácií.

Keďže kompresiou sa niektoré vrcholy stratili, musíme upraviť spôsob, ako sa pohybovať po suffixových linkoch. Úprava bude mierna. Po pridaní znaku prejdeme do najbližšieho vrcholu a popri tom si zapamätáme reťazec znakov, ktorý sme prešli. Keďže si znaky na hrane pamätáme ako dvojicu čísel, toto nám zaberie len $O(1)$ krokov. Keď následne prejdeme po suffixovom linku, vyhľadáme zapamätaný reťazec. Z vlastností stromu vyplýva, že stačí pozerat' len na prvé písmeno každej hrany preto, aby sme vedeli, či po nej máme prejsť alebo nie.

To, akým spôsobom sa písmenko po vyhľadaní pridá a kde vieme ušetriť počet krokov, popíšeme v nasledujúcej časti.

3.2.3 Rozširovanie stromu

Pri rozširovaní stromu o písmenko môžu nastať tieto tri prípady:

- *prvý prípad*: písmenko pripájame k vrcholu, z ktorého nepokračuje hrana, a teda je listom;

¹Tento krok predpokladá, že vieme adresovať reťazec priamo.

- *druhý prípad*: písmenko pripájame na miesto, z ktorého nepokračuje hrana s daným písmenkom, ale pokračuje z neho hrana s iným písmenkom;
- *tretí prípad*: písmenko pripájame na miesto, z ktorého pokračuje hrana s daným písmenkom.

Po tomto rozdelení môžeme ľahšie pozorovať ďalšie veci. Prvou je, že akonáhle vytvoríme novú vetvu (nastane prvý alebo druhý prípad), tak ju v ďalších krokoch môžeme len rozšíriť. Čiže, ak sme raz pridali list, ten aj listom navždy ostane. Túto skutočnosť môžeme využiť v implementácii tak, že pri vytváraní hrany nastavíme indexy hrany na i (index momentálneho písmenka, o ktoré strom rozširujeme) a e , čo je globálna premenná označujúca momentálny koniec v slove, ktorá sa každým rozšírením zvýši o jeden.

Druhým pozorovaním je, že po prvom výskyte tretieho prípadu sa strom už nerozšíri. Vyplýva to zo základnej vlastnosti sufixového stromu: ak rozširujeme strom o písmenko w_i a v strome je slovo $w_j \cdots w_i$, tak sú v ňom aj slová $w_{j+1} \cdots w_i, w_{j+2} \cdots w_i, \dots, w_i$.

Ďalším pozorovaním je, že pri rozširovaní najprv nastávajú prvé prípady, potom druhé a až potom tretie.

So všetkými týmito poznatkami môžeme konečne skonštruovať výslednú podobu algoritmu.

3.2.4 Zhrnutie

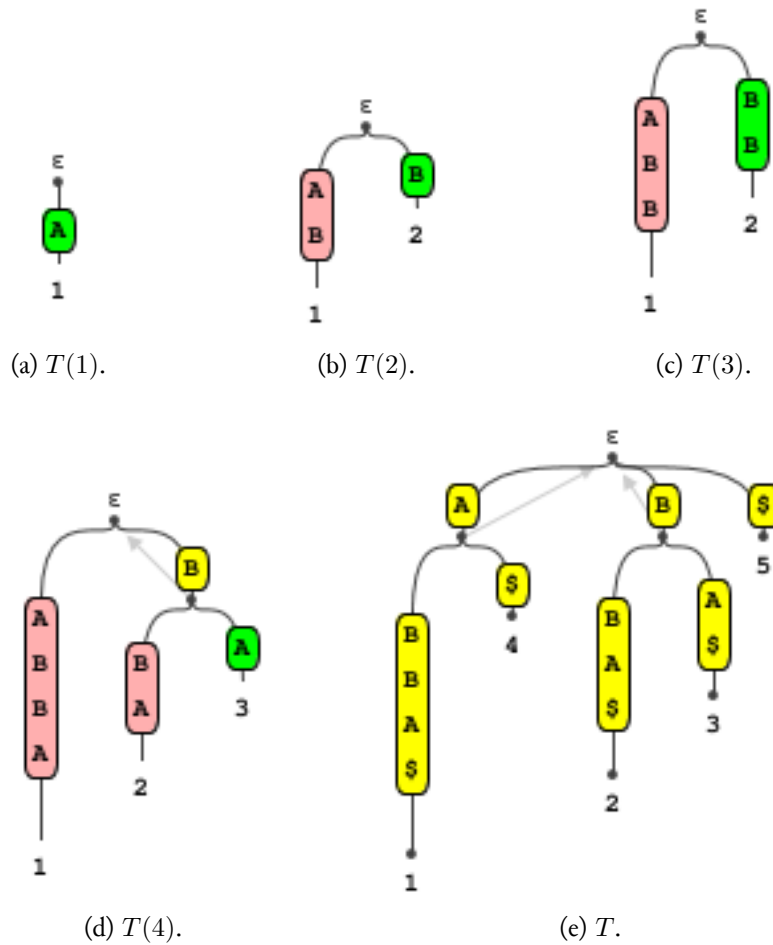
Sufixový strom T pre slovo $w = w_1w_2w_3 \cdots w_n\$$, vytvoríme tak, že z prázdneho stromu vytvoríme postupne stromy $T(1), T(2), T(3), \dots, T(n), T$.

Strom $T(1)$ vytvoríme triviálne; pridáme koreňu hranu s dvojicou indexov $(1, e)$ a nastavíme štartovací vrchol u_s na práve vytvorený vrchol.

Postupne vytvárame strom $T(i)$ zo stromu $T(i-1)$ rozšírením stromu $T(i-1)$ o písmenko w_i ² nasledovne:

1. index e nastavíme na novú hodnotu i . Tým sme vykonali všetky prvé prípady. Vieme si udržiavať ich počet j ,
2. za aktuálny vrchol u si zvolíme štartovací vrchol u_s ,
3. z aktuálneho vrcholu prejdeme po hrane vyššie a zapamätáme si reťazec α ,
4. ak nie sme v koreni, prejdeme po sufixovom linku. Nastavíme aktuálny vrchol,
5. z aktuálneho vrcholu vyhľadáme reťazec $w_{j+1} \cdots w_i$,
6. teraz môže nastať druhý alebo tretí prípad:

²Respektíve strom T zo stromu $T(n)$ o písmenko w_n .



Obrázok 3.1: *Suffixové stromy*. Na obrázku sú suffixové stromy pre slová A, AB, ABB, ABBA, ABBA\$. Šípky znázorňujú suffixové linky, ružovou a zelenou sú označené hrany, na ktoré sa bude v nasledujúcom, kroku aplikovať prvé pravidlo. Čísla označujú, na ktorom znaku začína daný suffix v slove.

- ak nastal druhý prípad, v prípade potreby rozdelíme hranu. Nastavíme aktuálny vrchol na posledný navštívený (respektíve vytvorený) a v prípade potreby nastavíme sufixový link. K aktívnemu vrcholu pripojíme hranu s indexami (i, e) , zvýšime index j o 1 a nastavíme štartovací vrchol na práve vytvorený. Vrátime sa na krok 3;
- ak nastal tretí prípad, v prípade potreby nastavíme sufixový link.

7. zvýšime hodnotu i o jeden a vrátime sa na krok 1.

Takto vybudujeme sufixový strom pre slovo w .

Kapitola 4

Popis softvéru

Kapitola 5

Implementácia softvéru

Záver

Tu by malo byť niečo v zmysle, že som to strašne nestihol a vymenované všetko, čo by som chcel dorobiť?

Literatúra

- APPEL, A. W.; JACOBSON, G. J. 1988. The world's fastest Scrabble program. *Communications of the ACM*. 1988, č. 31, č. 5, str. 572–578.
- FREDKIN, Edward. 1960. Trie memory. *Commun. ACM*. 1960, č. 3, č. 9, str. 490–499. Dostupné z WWW: <http://doi.acm.org/10.1145/367390.367400>. ISSN 0001-0782.
- GABOW, H.N.; TARJAN, R.E. 1985. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*. 1985, č. 30, č. 2, str. 209–221.
- GILBERT, J.R.; NG, E.G.; PEYTON, B.W. 1994. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*. 1994, č. 15, str. 1075.
- HOPCROFT, John E.; ULLMAN, Jeffrey D. 1973. Set Merging Algorithms. *SIAM J. Comput.* 1973, č. 2, č. 4, str. 294–303. Dostupné z WWW: <http://dx.doi.org/10.1137/0202024>.
- HUNDHAUSEN, C.D.; DOUGLAS, S.A.; STASKO, J.T. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*. 2002, č. 13, č. 3, str. 259–290.
- KNIGHT, Kevin. 1989. Unification: a multidisciplinary survey. *ACM Comput. Surv.* 1989, č. 21, č. 1, str. 93–124. Dostupné z WWW: <http://doi.acm.org/10.1145/62029.62030>. ISSN 0360-0300.
- KOVÁČ, Jakub. 2007. *Výhl'adávacie stromy a ich vizualizácia*. 2007. Bakalárska práca, Univerzita Komenského v Bratislave.
- KRUSKAL, Joseph B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*. 1956, č. 7, č. 1, str. 48–50. Dostupné z WWW: <http://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/S0002-9939-1956-0078686-7.pdf>.
- LEEUEWEN, J.; WEIDE, Th.P. van der. 1977. *Alternative Path Compression Rules*. 1977. An outline of the results were presented at the Fachtagung on Algorithms and Complexity Theory, Oberwolfach, Oct 1977.

- LIANG, F. M. 1983. *Word hy-phen-a-tion by com-put-er*. 1983.
- LUCCHESI, Claudio L.; LUCCHESI, Cl'audio L.; KOWALTOWSKI, Tomasz. 1992. *Applications of Finite Automata Representing Large Vocabularies*. 1992.
- MORRISON, Donald R. 1968. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*. 1968, č. 15, č. 4, str. 514–534. Dostupné z WWW: <http://doi.acm.org/10.1145/321479.321481>. ISSN 0004-5411.
- NAPS, T.L.; RÖßLING, G.; ALMSTRUM, V. a spol. 2002. Exploring the role of visualization and engagement in computer science education. In. *ACM SIGCSE Bulletin*. Č. 2, 2002, str. 131–152.
- SARAIYA, Purvi; SHAFFER, Clifford A.; MCCRICKARD, D. Scott; NORTH, Chris. 2004. Effective features of algorithm visualizations. In. *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. New York, NY, USA : ACM, 2004, str. 382–386. SIGCSE '04. Dostupné z WWW: <http://doi.acm.org/10.1145/971300.971432>. ISBN 1-58113-798-2.
- SHAFFER, Clifford A.; COOPER, Matthew L.; ALON, Alexander Joel D. a spol. 2010. Algorithm Visualization: The State of the Field. *Trans. Comput. Educ.* 2010, č. 10, str. 9:1–9:22. Dostupné z WWW: <http://doi.acm.org/10.1145/1821996.1821997>. ISSN 1946-6226.
- SINHA, Ranjan; ZOBEL, Justin. 2004. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*. 2004, č. 9. Dostupné z WWW: <http://doi.acm.org/10.1145/1005813.1041517>. ISSN 1084-6654.
- SINHA, R.; RING, D.; ZOBEL, J. 2006. Cache-efficient String Sorting using Copying. *J. Exp. Algorithmics*. 2006, č. 11, str. 1.2. ISSN 1084-6654.
- SKLOWER, K. 1991. A tree-based packet routing table for Berkeley Unix. In. *Proceedings of the Winter 1991 USENIX Conference*. 1991, str. 93–104.
- TARJAN, Robert E.; LEEUWEN, Jan van. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM*. 1984, č. 31, č. 2, str. 245–281. Dostupné z WWW: <http://doi.acm.org/10.1145/62.2160>. ISSN 0004-5411.
- UKKONEN, Esko. 1995. *On-Line Construction of Suffix Trees*. 1995.
- WALKER II, John Q. 1990. A node-positioning algorithm for general trees. *Software: Practice and Experience*. 1990, č. 20, č. 7, str. 685–705.