Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia union-findu, písmenkového stromu a sufixového stromu

Bakalárska práca

2012 **Viktor Tomkovič**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia union-findu, písmenkového stromu a sufixového stromu

Bakalárska práca

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

Bratislava, 2012 **Viktor Tomkovič**

Poďakovanie

Veľká vďaka za to, že mi pomohli patrí môjmu školiteľovi Kubovi Kováčovi a mojim rodičom.

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 sme tieto dátové štruktúry, konkrétne union-find, lexikografický strom a sufixový strom, definovali a konkrétne algoritmy popísali. V druhej časti sme spravili návrh aplikácie a popísali implementáciu. Práca je prehľadom dátových štruktúr s priloženým Java appletom na CD ako pomôckou pri vizualizácií.
Kľúčové slová: vizualizácia, ADŠ, algoritmy a dátové štruktúry, union-find, stringológia, lexikografický strom, trie, sufixový strom.

Abstract

This work is about visualization of algorithms and data structures. Continues the bachelor thesis of Jakub Kováč (2007) and extends the compilation by another data structures. In the first part are the structures, namely union-find, lexicographical tree (trie) and suffix tree, defined and the algorithms described. The second part is a description and a implementation of the software. The work is an overview of data structures with a software CD included.
Keywords: visualization, ADS, algorithms and data structures, union-find, stringology, lexicographical tree, trie, suffix tree.

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

2 Písmenkový strom **7**
2.1 Popis . . . 7
2.2 Použitie . . . 8

3 Sufixový strom **10**
3.1 Zostrojenie sufixového stromu . . . 10
3.2 Ukkonenov algoritmus . . . 10
3.2.1 Sufixové linky . . . 11
3.2.2 Zníženie nárokov na pamäť . . . 11
3.2.3 Rozširovanie stromu . . . 12
3.2.4 Zhrnutie . . . 12
3.3 Použitie . . . 14

4 Popis softvéru **15**
4.1 Analýza existujúcich riešení . . . 15
4.1.1 Union/Find Algorithm Visualization (union-find) . . . 15
4.1.2 AlVie (union-find) . . . 16
4.1.3 Data Structure Visualizations (union-find) . . . 17
4.1.4 TRAKLA2 (trie) . . . 17
4.1.5 Sufixové stromy . . . 19
4.1.6 Ďalšie vizualizácie . . . 19
4.2 Špecifikácia požiadaviek . . . 20
4.2.1 Požiadavky . . . 20

4.2.2 Prevádzkové požiadavky 20
4.3 Návrh 21
4.3.1 Technické požiadavky 21
4.3.2 Používateľské požiadavky 21
4.3.3 Použité technológie 21
4.3.4 Rozhranie aplikácie 21
4.3.5 Prepojenie s predošlým systémom 22

5 Implementácia softvéru **23**
5.1 Vizualizácia 23
5.1.1 Union-find 23
5.1.2 Písmenkový strom 24
5.1.3 Sufixový strom 24
5.2 Popis ovládania 25
5.2.1 Union-find 25
5.2.2 Písmenkový a sufixový strom 25
5.3 Testovanie, prevádzka a údržba 26

Záver **27**

Literatúra **28**

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.

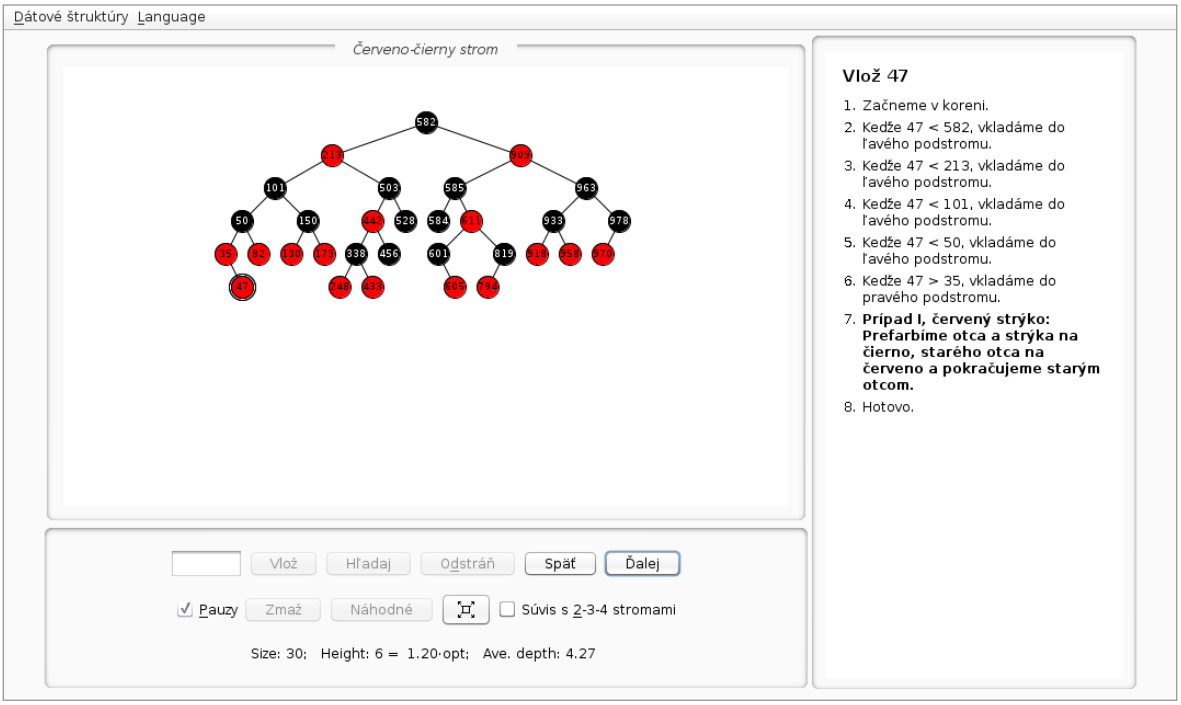
Našou snahou je vytvoriť kvalitný softvér nezávislý od operačného systému, ktorý bude vyhovovať ako pomôcka pri výučbe ako aj pri samoštúdiu a bude voľne prístupný.

Predchádzajúci výskum v oblasti pedagogiky zatiaľ nedokázal úplne preukázať pedagogickú efektívnosť vizualizácií (Shaffer a kol. 2010), avšak viacero štúdií potvrdilo zvýšený záujem a zapojenosť študentov (Naps a kol. 2002; C.D. Hundhausen, Douglas a 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 a kol. 2010). Zbieraním a analyzovaním kvality sa venuje skupina AlgoViz (`http://algoviz.org/`).

Zaujímavé je pozorovanie, že určovanie si vlastného tempa pri vizualizácií je veľká pomôcka. Naopak, ukazovanie pseudokódu alebo nemožnosť určenia si vlastného tempa (napríklad animácia bez možnosti pozastavenia), takmer žiadne zlepšenie neprináša (Shaffer a kol. 2010; Saraiya a kol. 2004).

Prácu sme rozdelili na popis jednotlivých dátových štruktúr, ktoré sme vizualizovali a popis samotného softvéru. V kapitolách 1, 2 a 3 je postupne popis *union-find*, *písmenkového stromu* a *sufixového stromu*. V kapitole 4 je popis softvéru a v kapitole 5 je implementácia.



Obrázok 1: *Softvér Gnarley Trees.* V ovládacom paneli dole môže užívateľ zvoliť operáciu a vstupnú hodnotu a sledovať priebeh algoritmu (monentálne 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*(x) – 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 a Peyton (1994) ukázali, ako sa dá union-find použiť pri Choleského dekompozícii riedkych matíc. 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 a 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 $p(x) =$ `NULL`).

Vytváranie prvkov. Operácia *makeset*(x) teda vytvorí nový prvok x a nastaví $p(x) = \texttt{NULL}$.

Hľadanie zástupcu. Operáciu *find*(x) vykonáme tak, že budeme sledovať cestu po smerníkoch, až kým nenájdeme zástupcu.

Spájanie množín. Operáciu *union*(x, y) najjednoduchšie vykonáme 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 *find*(x) 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*.

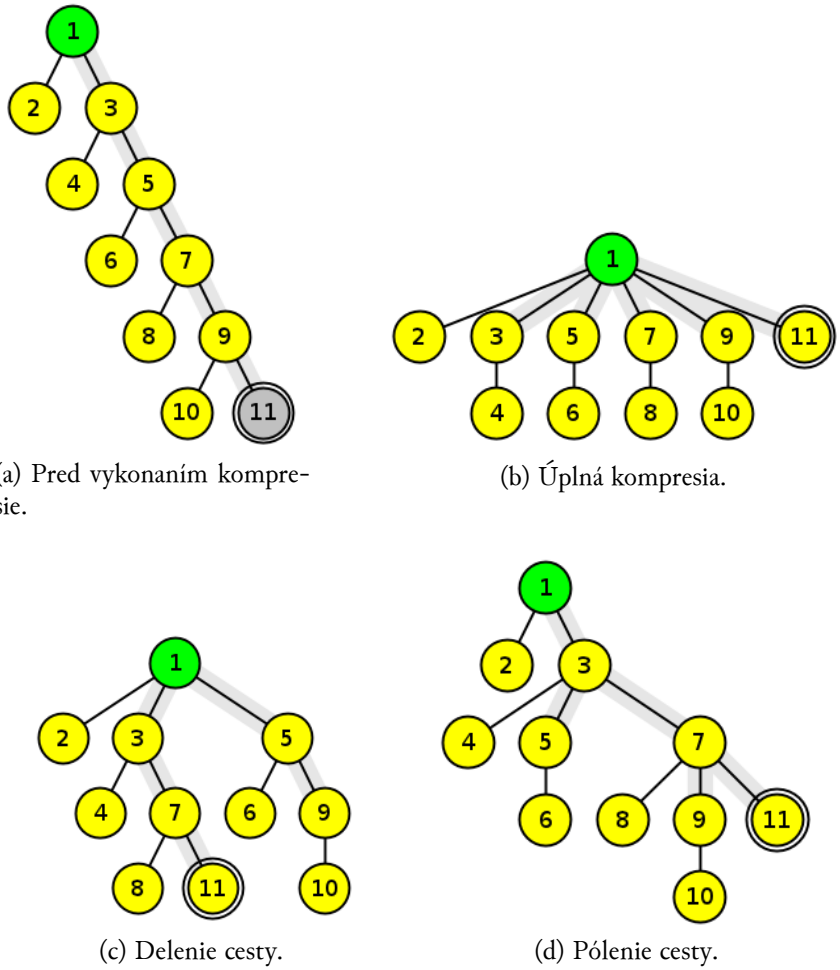
1.3 Heuristika na spájanie

Prvá heuristika pre každý vrchol x udržuje hodnotu *rank*(x), ktorá určuje najväčšiu možnú hĺbku podstromu s koreňom x. Pri operácií *makeset*(x) zadefinujeme *rank*$(x) = 1$. Pri operácií *union*(x, y) porovnáme *rank*(x) a *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 *rank*(y) zvýšime o 1.

1.4 Heuristiky na kompresiu cesty

Druhou heuristikou je kompresia cesty. Algoritmov na efektívnu kompresiu cesty je veľa (Robert E. Tarjan a Jan van Leeuwen 1984), tu popíšeme tie najefektívnejšie. Prvou z nich je *úplná kompresia* (Hopcroft a Ullman 1973). Pri vykonávaní operácie *find*(x), 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 a Weide 1977). Pri vykonávaní operácie *find*(x) pripojíme každý vrchol v ceste od vrcholu x po koreň stromu na otca jeho otca (obr. 1.1c). Treťou heuristikou je *pólenie cesty* (J. Leeuwen a Weide 1977). Pri vykonávaní operácie *find*(x) pripojíme každý druhý vrchol v ceste od vrcholu x po koreň stromu na otca jeho otca (obr. 1.1d).

Časová zložitosť union-findu záleží od toho, koľko prvkov je v množinách a koľko je operácií celkovo vykonaných operácií. Všetky uvedené spôsoby ako vykonať operáciu *find*(x) sa dajú použiť s obomi realizáciami operácie *union*. 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 kompresií $\Theta(m\,\alpha(m, n))$, kde α je inverzná Acker-



(a) Pred vykonaním kompresie.

(b) Úplná kompresia.

(c) Delenie cesty.

(d) Pólenie cesty.

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 zložitostí, 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 zložitostí, ak $m < n$.

Tabuľka 1.1: *Porovnanie časových zložitostí pre rôzne kombinácie hľadaní prvkov a spájaní množín pre union-find (Robert E. Tarjan a Jan van Leeuwen 1984).* Počet prvkov je n a počet operácií je m. Inverzná Ackermannova funkcia je označená ako α. V praxi zväčša platí, že $m > n$.

mannova funkcia tak, ako ju definoval Tarjan (Robert Endre Tarjan 1983). V tabuľke 1.1 je porovnanie časových zložitostí (Robert E. Tarjan a Jan van Leeuwen 1984).

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

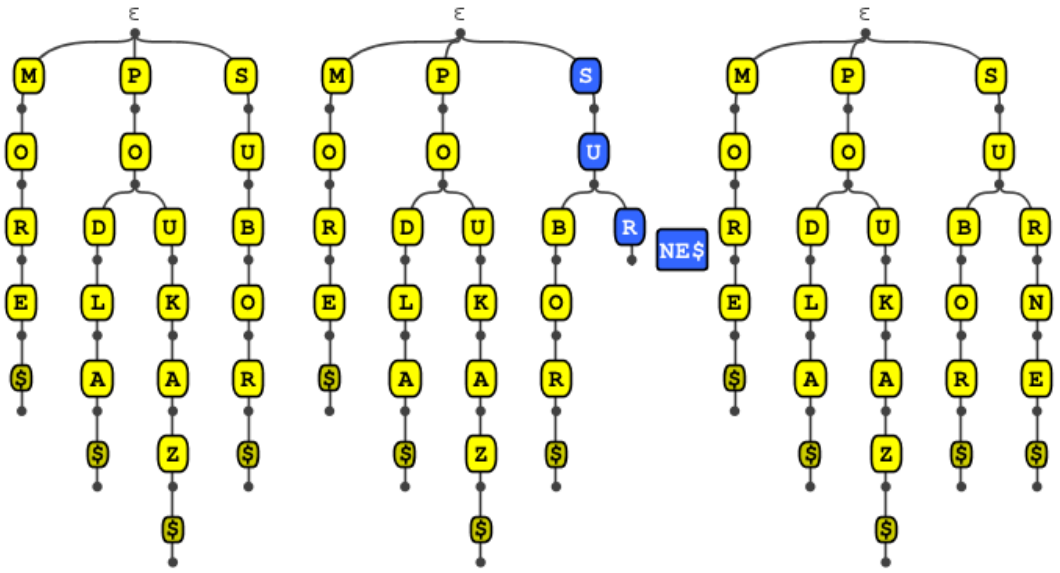
- *insert*(w) – pridá do stromu slovo w;
- *find*(w) – zistí, či sa v strome slovo w nachádza;
- *delete*(w) – odstráni zo stromu slovo w.

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

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

Vkladanie slova. 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).

Hľadanie slova. 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.



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

Mazanie slova. 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 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.

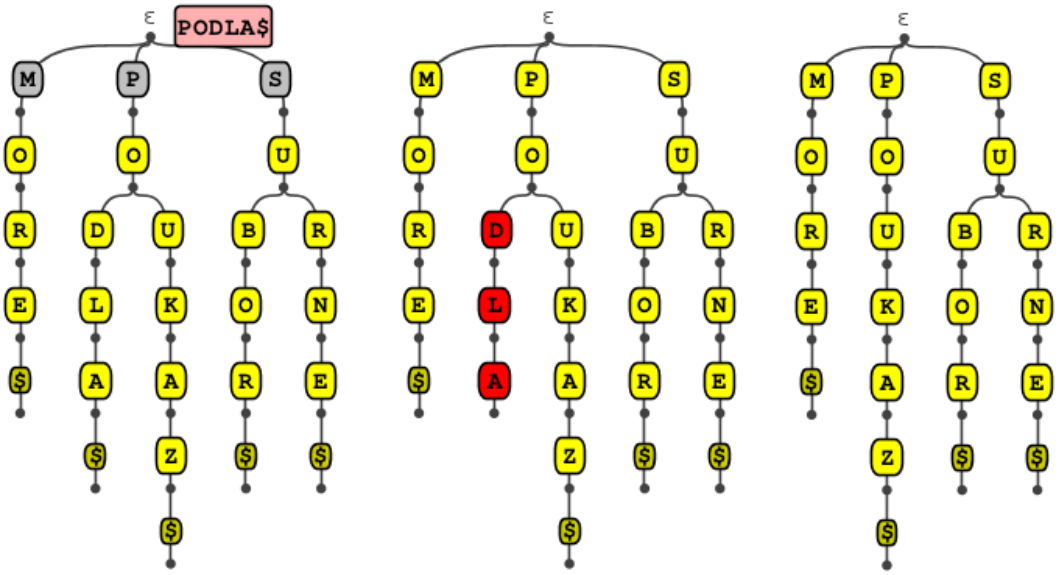
2.2 Použitie

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 zaberal 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 prirodzených jazykov výrazne znižujú pamäťový priestor potrebný na uchovanie dátovej štruktúry. Vďaka tomu sa využívajú na jazykovú

[1] Z anglického re*trie*val – získanie.



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

korekciu, automatické dopĺňanie slov a podobne (Appel a Jacobson 1988; Lucchesi a Kowaltowski 1992).

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 a Justin Zobel (2004) a veľmi výrazne zrýchlili triedenie dlhých zoznamov slov. Neskôr tento algoritmus vylepšili R. Sinha, Ring a J. Zobel (2006). Algoritmus používa trie s poľami, ktoré sa môžu zväčšiť. Pokiaľ sa pole zväčší za povolenú hranicu, „prepáli“ sa. Preto bol algoritmus nazvaný *burstsort*.

Kapitola 3

Sufixový strom

Sufixový strom je písmenkový strom pre všetky *sufixy (prípony)* daného slova. Jeho veľká výhoda spočíva v rýchlom zostrojení a veľkom množstve efektívnych algoritmov na ňom.

3.1 Zostrojenie sufixového stromu

Najjednoduchšie riešenie ako zostrojiť sufixový strom je využiť písmenkový strom a pridať do neho všetky sufixy. Takéto riešenie má časovú aj pamäťovú zložitosť $O(n^2)$. Ako prvý navrhol lineárne riešenie Weiner (1973). Weinerove riešenie zjednodušil McCreight (1976). Ďalšie zjednodušenie vymyslel Ukkonen (1995). Navyše Ukkonen (1995) navrhol, ako zostrojiť strom za behu.

3.2 Ukkonenov algoritmus

Často budeme chcieť označovať podslovo slova $w = w_1w_2w_3 \cdots w_n$. Pre podslovo od indexu i po index j sme zaviedli označenie $w[i : j]$, teda $w = w[1 : n]$. Keďže každá cesta z koreňa do nejakého vrcholu zodpovedá slovu, vrcholy stotožníme so slovami. Pokiaľ hovoríme, že sme slovu w pripojili písmenko a, myslíme tým, že sme do písmenkového stromu za korešpondujúci vrchol pripojili ďalší s hranou a. V prípade stromu s komprimovanými hranami to znamená, že sme hranu rozšírili o písmenko a (obrázok **??**).

Hlavnou myšlienkou algoritmu je pre slovo $w = w_1w_2w_3 \cdots w_n\$$ postupne vytvoriť sufixové stromy pre slová $w[1 : 1], w[1 : 2], w[1 : 3], \ldots, w[1 : n + 1]$. Označme tieto stromy $T(1), T(2), \ldots, T(n), T(n + 1) = T$. Teda, vytvárame n stromov a pomocou písmenkového stromu s nekomprimovanými hranami, 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

$w[1:i-1], w[2:i-1], \ldots, w[i-1:i-1]$), tak pre vytvorenie stromu $T(i)$ netreba znovu vytvárať nový strom a pridávať do neho sufixy $w[1:i], w[2:i], \ldots, w[i-1:i], w[i:i]$, ale stačí rozšíriť existujúce sufixy stromu $T(n-1)$ 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 Sufixové linky

Pri vytváraní stromu $T(i)$ zo stromu $T(i-1)$ postupne rozširujeme sufixy, no bolo by neefektívne ich vždy vyhľadávať z koreňa. Preto zavedieme *sufixovú linku*, čo je orientovaná hrana, ktorú má každý vrchol okrem koreňa. Zo sufixu av ide linka do sufixu v.

Algoritmus teda zmeníme tak, že namiesto opätovného vyhľadávania sufixu z koreňa, vyhľadáme bod, z ktorého začneme pridávať len raz a ďalej sa navigujeme po linkách.

Pri vytváraní stromu $T(i)$ ako prvý pridávame najdlhší sufix $w[1:i]$. Sufix $w[1:i-1]$ v strome máme a pamätáme si ho ako miesto z kade vytváranie stromu $T(i)$ začneme. Rozšírime sufix $w[1:i-1]$ o písmenko w_i. Ďalej by sme chceli rozšíriť sufix $w[2:i-1]$. Ten sa v strom nachádza a smeruje na ňho sufixová linka. Stačí po nej prejsť, rozšíriť sufix $w[2:i-1]$ o písmenko w_i a vytvoriť sufixovú linku z $w[1:i]$ do $w[2:i]$. Takto pokračujeme, až kým nepripojíme sufix $w[i:i]$.

Ako nám to pomohlo zlepšiť časovú zložitosť? Miesto, z kade štartujeme pridávanie sufixov si pamätáme. Rozšírienie stromu $T(i-1)$ o jeden znak zaberie $O(i)$ krokov. Postupne rozširujeme n ráz. Na vybudovanie stromu T teda potrebujeme $O(n^2)$ krokov.

Ďalším zlepšením bude pamäťová optimalizácia.

3.2.2 Zníženie nárokov na pamäť

Veľmi priamočiarym krokom k zlepšeniu pamäťovej 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. 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 algoritmus[1]. Po týchto zlepšeniach nám stačí pamätať si $O(n)$ informácií.

Keďže kompresiou sa niektoré vrcholy stratili, musíme upraviť spôsob, ako sa pohybovať po sufixových linkoch. Úprava bude mierna. Po rozšírení sufixu $w[i:j]$ o znak prejdeme do najbližšieho vrcholu a popri tom si zapamätáme reťazec znakov α na hrane, ktorou sme prešli. Keďže si znaky na hrane pamätáme ako dvojicu čísel, toto nám zaberie len $O(1)$ krokov. Keď následne prejdeme po sufixovom linku, vyhľadáme zapamätaný reťazec. Z vlastností stromu vyplýva, že stačí pozerať len na prvé písmeno každej hrany preto, aby sme vedeli, či po nej máme prejsť alebo nie.

[1]Tento krok predpokladá, že vieme adresovať reťazec priamo.

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;
- *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 implementácií 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 $w_{j+1} \cdots w_i, w_{j+2} \cdots w_i, \ldots, w_i$.

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.

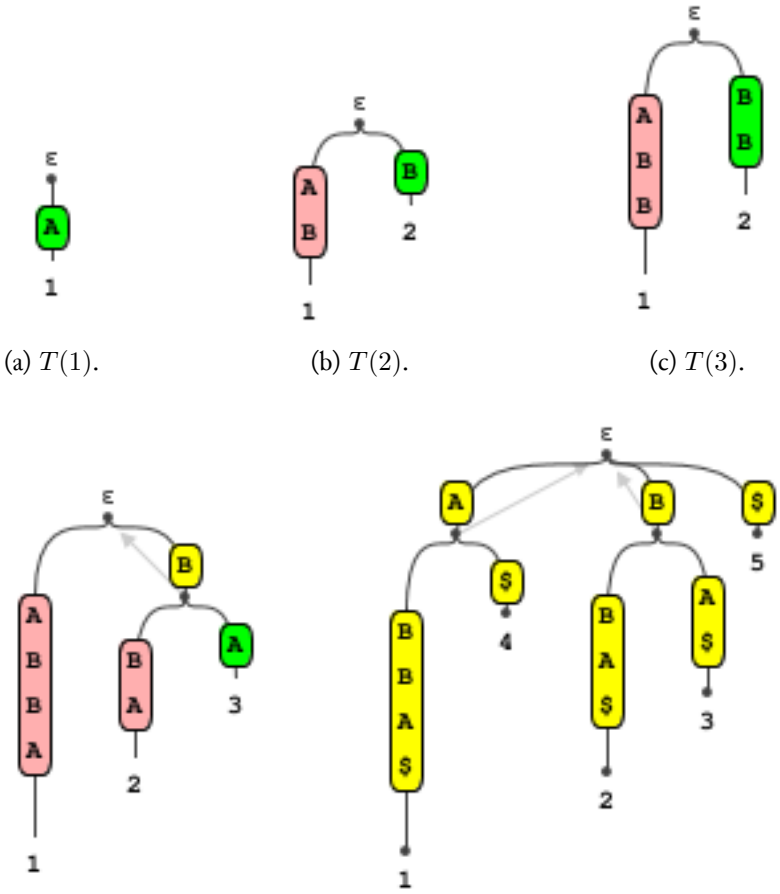
3.2.4 Zhrnutie

Sufixový strom T pre slovo $w = w_1 w_2 w_3 \cdots w_n\$$ vytvoríme tak, že z prázdneho stromu vytvoríme postupne stromy $T(1), T(2), T(3), \ldots, T(n), T(n+1) = T$.

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 následovne:

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;



(a) $T(1)$. (b) $T(2)$. (c) $T(3)$.

(d) $T(4)$. (e) T.

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

3. z aktuálneho vrcholu prejdeme po hrane vyššie a zapamätáme si reťazec na hrane po ktorej sme prešli;

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:

   - 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.

3.3 Použitie

Sufixový strom má v stringológií veľa využití (Gusfield 1997). Algoritmus na zistenie prítomnosti podreťazca v slove je rovnaký ako pri písmenkovom strome. Navyše vieme určiť, na ktorej pozícií sa podreťazec vyskytuje. Pri konštrukcii stromu stačí označovať listy číslami v poradí podľa vytvorenia. Hľadanie podreťazca p v predspracovanom texte T nám zaberie len $O(|p|)$ krokov.

Sufixový strom pre viacej reťazcov sa nazýva *všeobecný sufixový strom* a dá sa zostrojiť miernou úpravou Ukkonenovho algoritmu. Reťazce $v_1, v_2, v_3, \ldots, v_n$ spojíme a oddelíme unikátnymi oddelovačmi. Vznikne nám slovo $v_1\$_1 v_2\$_2 v_3\$_3 \cdots v_n\$_n$, z ktorého vieme vybudovať sufixový strom. Tento strom sa dá využiť na vyhľadanie *najdlhšieho spoločného podreťazca*.

Najdlhší opakujúci sa podreťazec sa v sufixovom strom určite vetví, takže ho nájdeme ako najdlhšie podslovo, ktoré sa nekončí v liste.

Pre slovo α vieme pomocou sufixových stromov nájsť efektívne *najdlhšie palindormatické podslovo*, čo je slovo $\beta \subset \alpha$ také, že $\beta^R = \beta$. Zostrojíme všeobecný sufixový strom pre slová β a β^R. Pre tento strom vieme efektívne zostrojiť dátovú štruktúru pre najnižšieho spoločného predka (Harel a R. Tarjan 1984). Potom nám stačí pre každý index q od 1 po $n - 1$ hľadať najnižšieho spoločného predka pre sufixy $w[q : n]$ a $w^R[n - q : n]$.

Kapitola 4

Popis softvéru

V predchádzajúcich kapitolách sme si popísali niektoré dátové štruktúry a vybrané algoritmy. Hlavnou náplňou je ich vizualizácia. Ako sme povedali v úvode, vizualizácia je názorné vykreslenie dátovej štruktúry. Vizualizácia algoritmu je znázornenie priebehu algoritmu na dátovej štruktúre.

4.1 Analýza existujúcich riešení

Softvér vizualizuje dátové štruktúry, algoritmy na nich a popisuje ako algoritmy fungujú. Preto sa hodí ako učebná pomôcka pre študentov na samoštúdium a pre učiteľov na názorné ukážky pri vyučovaní.

Vizualizované dátové štruktúry sú známe a rozšírenie vizualizácie je veľké, preto exituje veľa podobných aplikácií a appletov znázorňujúcich tieto dátové štruktúry a algoritmy. Väčšina z nich je na úrovni školských projektov a je súkromná. Preto porovnám len tie vizualizácie, ktoré sú vybrané skupinou ľudí venujúcej sa vizualizácií algoritmov; skupinou `algoviz.org`. Analýza prebehla v máji 2012.

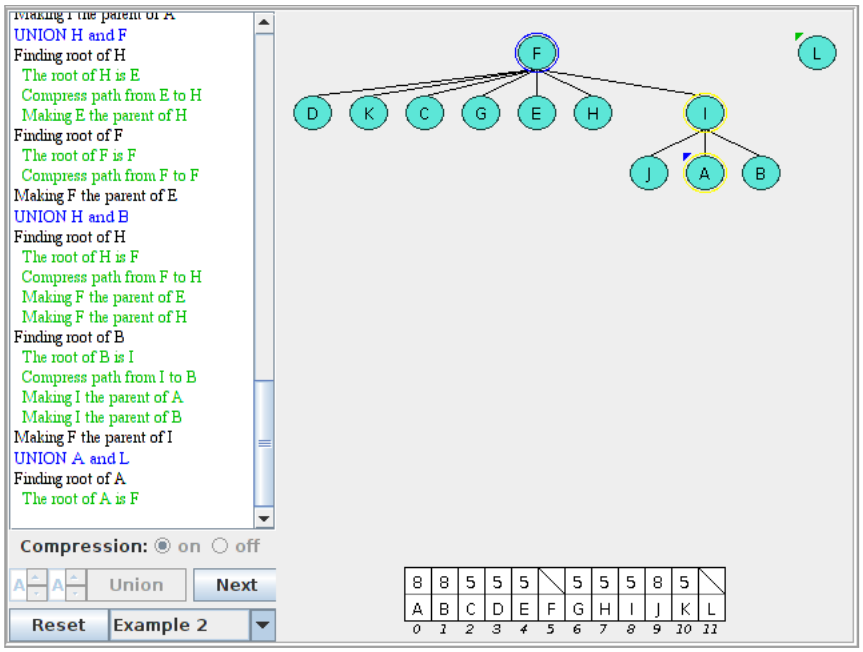
4.1.1 Union/Find Algorithm Visualization (union-find)

Tento malý applet je súčasťou väčšieho projektu *Virginia Tech Algorithm Visualizations*. Applet je implementovaný v programovacom jazyku `JAVA` a po grafickej stránke je veľmi podobný ako zvyšok projektu. Softvér je vydaný pod licenciou `GPL`. Vyvíjali ho Cris Kania a Cliff Shaffer počas jesene 2004.

Je dostupný na: `http://research.cs.vt.edu/AVresearch/UF/`.

Popis

Applet poskytuje operácie *union* a *find* na ôsmich, dvanástich, šestnástich alebo dvadsiatich vrcholoch. Na obrázku 4.1 je obrazovka appletu. Môžeme vidieť, že dátová štruktúra je vykreslená vpravo. Použitý je jednoduchý algoritmus. Vpravo dole je vykreslená reprezentácia



Obrázok 4.1: *Softvér Union/Find Algorithm Visualization.* V ľavej časti obrazovky sa nachádzajú výpisy a možnosti nastavenia, vpravo je vizualizácia dátovej štruktúry a prebiehajúceho algoritmu.

v poli a pre každý vrchol je vypísaný otec. Vľavo je riadne vypísaný zoznam vykonaných krokov, ktorý je navyše farebne odlíšený. Farby vo výpise a vo vizualizácií spolu korešpondujú. Vľavo dole sú nastavenia. Applet poskytuje operácie *union* s heuristikou na spájanie a *find* s možnosťami bez kompresie a s kompresiou cesty (popísané v sekciách 1.3 a 1.4).

Softvér poskytuje priestor len na malé príklady. Na pochopenie ako fungujú algoritmy to je postačujúce. Ovládanie je intuitívne.

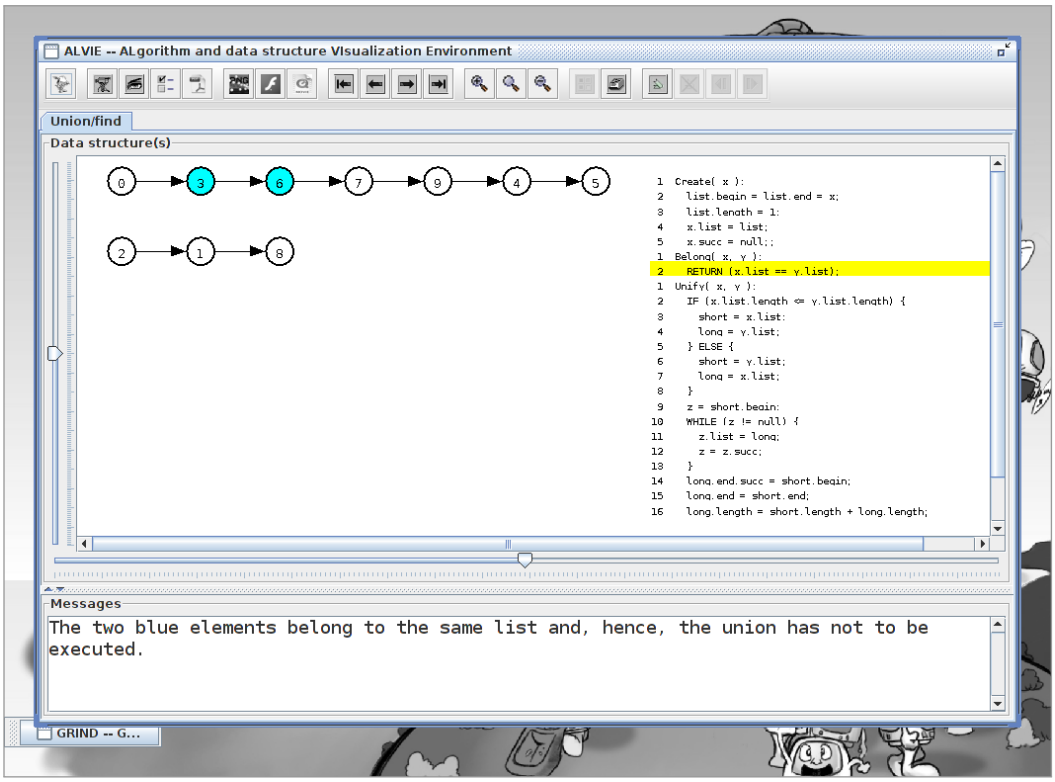
4.1.2 AlVie (union-find)

AlVie je veľký projekt. Je to prostredie na vizualizáciu algoritmov, vytváranie vlastných vizualizácií a veľmi flexibilné pridávanie vlastného materiálu. Slúžil na podpru talianskych škôl. Projekt momentálne spravuje Pilu Crescenzi. Je implementovaný v jazyku `JAVA`.

Popis

Na webovej stránke projektu (`http://alvie.algoritmica.org/alvie3`) je prehľadný popis funkcionality a používania. Aplikácia ponúka množstvo nastavení a veľa možností exportovania danej vizualizácie. Avšak nepodarilo sa mi nastaviť si vlastný vstup, preto som ostal len pri základnej možnosti. Union-find je tu bohužiaľ implementovaný nie cez les, ale ako spájaný zoznam (obr. 4.2), čo asymptoticky spomaľuje beh algoritmu.

Napriek veľkému množstvu dodávaných algoritmov, nastavení a zobrazenému pseudokódu počas vizualizácie má tento softvér nevýhody v podobe horšieho ovládania a neoptimálnej implementácie disjunktných množín.



Obrázok 4.2: *Softvér Alvie.* Prebieha XML skript. Vpravo je pomocný pseudokód.

4.1.3 Data Structure Visualizations (union-find)

Vizualizácia union-find algoritmu je v tomto prípade súčasťou veľkého projektu využívajúceho veľa technológií (Flash, JAVA, HTML5). Autorom je David Galles.

Projekt je prístupný na: http://www.cs.usfca.edu/~galles/visualization/.

Popis

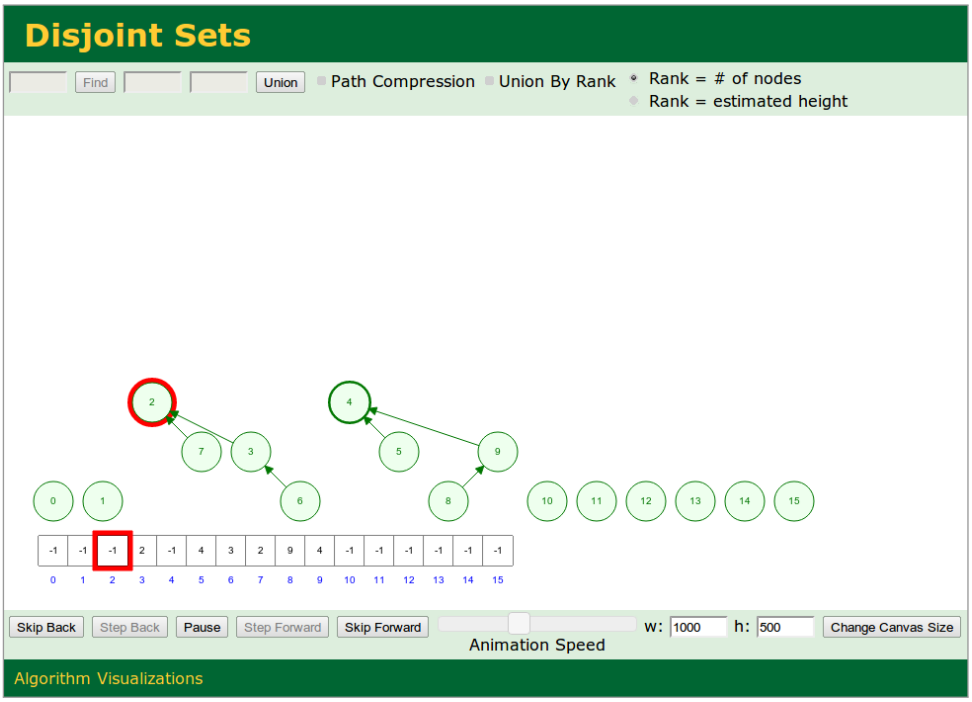
Autor si dal záležať na použití moderných technológií a celá internetová aplikácia vyzerá veľmi pekne. Aplikácia zobrazuje štruktúru aj ako les aj ako pole. Dá sa nastaviť rýchlosť algoritmu a možnosti heuristík. Vstupné parametre operácií sa zadávajú cez textové políčka.

Aplikácia je dobre popísaná, dá sa na nej nastaviť všetko potrebné vrátane rýchlosti premietania a veľkosti vykreľovacej plochy. Nevýhodou je, že nevypisuje, čo sa práve deje a teda používateľ musí algoritmus poznať.

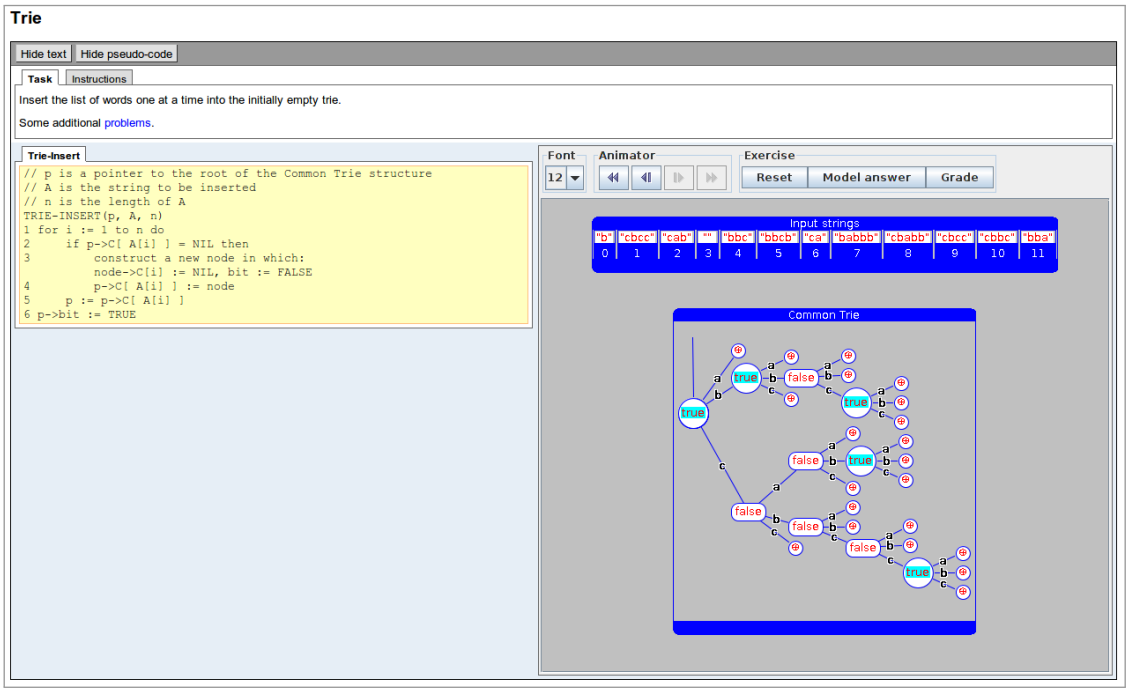
4.1.4 TRAKLA2 (trie)

TRAKLA2 je učebná pomôcka vyvíjaná skupinou ľudí „Software visualization group“. Všetci sú študenti alebo zamestnanci na Helsinskej technickej univerzite. Obsahuje veľa algoritmov a ku každému má sadu úloh, ktoré žiak môže riešiť.

Celý výučbový systém je dostupný a popísaný na: http://www.cse.hut.fi/en/research/SVG/TRAKLA2/index.shtml.



Obrázok 4.3: *Softvér Data Structure Visualizations.* Prebieha spájanie dvoch množín. Celá aplikácia beží v internetovom prehliadači.



Obrázok 4.4: *Prostredie TRAKLA2.* Riešenie zadania; budovanie písmenkového stromu obsahujúceho zadané reťazce. Celá aplikácia beží v internetovom prehliadači.

Popis

Už na prvý pohľad je vidieť, že na projekte sa podieľa veľa ľudí. Stojí za ním viac ako 40 ľudí. Systém je premyslený, každý algoritmus obsahuje pseudokód. Všade sú pomôcky, takže ak sme niečomu nerozumeli, našli sme navigáciu na odpoveď.

Samotné vizualizovanie písmenkového stromu je vykonávané niektorým tesným algoritmom, čo je veľká výnimka pri vizualizáciach, ktoré sa nezaoberajú tým, ako vykresľovať grafy! V hornej časti je umiestnený pomocný text, vľavo je pseudokód a vpravo je dátová štruktúra, na ktorej sa vykonáva zadanie (obrázok 4.4). Je možné zvoliť si veľkosť písma.

Všetko je spravené pekne a prehľadne, možno grafická stránka by sa dala vylepšiť. Jediná chyba, na ktorú som narazil bola, že pokiaľ som kroky nevykonával v správnom poradí, nebola mi uznaná správna odpoveď aj keď výsledok správny bol. Toto je ale špecifická chyba len pre písmenkový strom (pre iné dátové štruktúry môže poradie pridávania zmeniť reprezentáciu).

4.1.5 Sufixové stromy

Na sufixové stromy, konkrétne Ukkonenov algoritmus, ktorý sme vizualizovali, sa podarilo nájsť len jednu a aj to textovú vizualizáciu.

4.1.6 Ďalšie vizualizácie

Uviedli sme tri softvéry pre union-find a jeden pre písmenkový strom. Na jednoduché písmenkové stromy sme viac vizualizácií nenašli. Ďalšie programy vizualizovali textové dáta alebo iné modifikácie písmenkových stromov (niektoré sú uvedené v sekcií 2.2).

Všetky aplikácie, ktoré sme analyzovali, však mali jeden spoločný nedostatok. Nedalo sa rýchlo dostať ku stavu, kedy je na štruktúre vykonaných veľa operácií a pokračovať z tohto stavu ďalej. Na prázdnej dátovej štruktúre nemusí byť dobre vidieť priebeh operácie. Taktiež to môže byť nepríjemné v prípade, že si chce používateľ len pripomenúť na už existujúcej štruktúre ako algoritmus funguje, prípadne ako štruktúra po vykonaných operáciách vyzerá.

4.2 Špecifikácia požiadaviek

Počas analýzy sme zistili, že vizualizácie mali nedostatky najmä v tom, že sa na nich nedalo naraz vykonať viac operácií. V prípade, že sa dalo (sekcia 4.1.2), tak sa z tohto stavu nedalo pokračovať. Ďalej sme zistili, že aplikácie neposkytujú rozšírené metódy vstupu. Všeobecne bolo veľmi obtiažne vôbec vybudovať väčšiu dátovú štruktúru, na ktorej by boli algoritmy a hlavne ich efektivita viditeľnejšia. Pri niektorých vizualizáciach nebolo jasne viditeľné, čo sa ide vykonať.

4.2.1 Požiadavky

Naša aplikácia bude používaná najmä študentmi, ktorí si chcú zopakovať učivo z prednášok, alebo učiteľmi, ktorí túto aplikáciu budú chcieť využívať na hodinách.

Preto sa nám zdalo byť vhodné, aby študent dostával výpisy o tom, čo sa bude diať a aby bolo prostredie pre študenta dostatočne používateľsky príjmené. Pre učiteľa je potrebné, aby sa aplikácia prijateľne zobrazovala cez projektor, aby bola dátová štruktúra od začiatku naplnená (veľakrát sa algoritmy vysvetlujú na pripravenej dátovej štruktúre) a taktiež, aby bolo prostredie používateľsky príjemné.

Softvér ako taký mal vizualizovať tieto veci:

- dátovú štruktúru pre disjunktné množiny reprezentovanú stromom a na nej tieto algoritmy:
  - naivné spájanie, spájanie podľa ranku;
  - nekomprimované hľadanie, hľadanie s jednoduchou kompresiou, hľadanie s delením cesty, hľadanie s pólením cesty.
- písmenkový strom (trie) a na ňom tieto algoritmy:
  - vkladanie slova;
  - testovanie na prítomnosť slova;
  - mazanie slova.
- sufixový strom a jeho vytváranie pomocou Ukkonenovho algoritmu.

4.2.2 Prevádzkové požiadavky

Pretože ľudia používajú rôzne operačné systémy a softvér nepracuje so špecifickými systémovými zdrojmi bolo potrebné, aby bola aplikácia nezávislá od operačného systému. Keďže má byť dostupná na študentské účely, nemala by byť hardvérovo náročná.

4.3 Návrh

Táto práca je pokračovaním práce Jakuba Kováča a preto návrh vychádzal hlavne z jeho práce. Našou hlavnou úlohou bolo implementovať vybrané dátové štruktúry a algoritmy. V nasledujúcich častiach sme popísali požiadavky kladené na náš softvér, či už z hľadiska technického alebo používateľského, zvolené technológie a rozhranie systému.

4.3.1 Technické požiadavky

Softvér slúži na vizualizáciu a bolo potrebné ho implementovať multiplatformovo. Taktiež slúži aj na opisovanie prebiehajúceho algoritmu, teda bola potrebná plocha na vykresľovanie, plocha na vypisovanie toho, čo sa deje, ovládacie prvky algoritmu a ovládacie prvky na výber dátových štruktúr.

4.3.2 Používateľské požiadavky

Ako sme už naznačili, používateľmi sú prevažne študenti a učitelia, veľmi pravdepodobne informatiky. Preto bolo vhodné okrem obyčajného a pomalšieho zadávania vstupu po jednej operácií vytvoriť prostredie, v ktorom sa bude dať zadávať aj zložitejší vstup.

Ďalej bolo potrebné názorne vizualizovať dátové štruktúry a všetko podstatné k pochopeniu algoritmov. Konkrétne pri union-finde išlo o prípadné ranky, pri písmenkovom strome o odstraňovanie vetiev, prehľadné vykresľovanie pri prebiehajúcich algoritmoch a pri sufixovom strome sufixové linky, práve pridávaný sufix a celé slovo, ktorému sufixový strom budujeme.

4.3.3 Použité technológie

Keďže projekt bol pokračovaním predošlej práce za hlavný programovací jazyk bola zvolená `JAVA`. Na grafické znázornenie sú to jednotlivé triedy balíkov AWT a Swing. Na generovanie náhodných reťazcov sme použili špeciálne upravenú triedu s návrhovým vzorom *singleton*. Vďaka celkovej nenáročnosti na technológie a zameraniu projektu nebolo potrebné použiť viac technológií.

4.3.4 Rozhranie aplikácie

Rozhranie vizualizácií bolo podobné ako všetkých iných štruktúr implementovaných v predošlej verzií softvéru. Vstupné dáta boli zadávané do vstupného textového poľa, alebo klikaním myšou. Podľa stlačeného tlačidla sa spustil daný proces. Výstupne údaje boli prezentované vo vykresľovacom okne pre dátovú štruktúru a vo výpise pre komentáre. Vykreslená štruktúra sa dala zväčšiť, zmenšiť a posunúť pomocou myšky.

Pre každú dátovu štruktúru bolo potrebné zabezpečiť vstupné textové pole, prípadne dodatočné vstupné metódy a tlačidlá pre všetky operácie.

Union-find

Pre union-find bolo potrebné zabezpečiť vstupné textové pole, tlačidlá pre operácie *union* a *find*, vyberanie vstupných prvkov myšou, možnosť výberu medzi naivným spájaním a spájaním podľa ranku a medzi hľadaním zástupcu bez kompresie, s jednoduchou kompresiou, delením cesty a pólením cesty (popísané v kapitole 1).

Písmenkový strom

Pri písmenkovom strome bolo nutné zabezpečiť vstupné pole tak, aby sa do systému dostali slová, len s abecedou, ktorú chceme. Pre nás abecedu tvorili veľké písmená anglickej abecedy. Ďalej bolo potrebné, aby sa dal pridať do stromu ucelenejší text. Na vykonávanie operácií *insert*, *find* a *delete* bolo potrebné implementovať tlačidlá.

Sufixový strom

Sufixový strom bolo potrebné vytvoriť zo vstupného slova. Preto treba na tento účel implementovať jedno tlačidlo.

4.3.5 Prepojenie s predošlým systémom

Po implementácií mal systém tvoriť jeden celok, preto bola potreba dbať na celistvosť a podobný štýl programovania. Na jednotlivé implementovanie obrazoviek, vstupných polí a tlačidiel bolo nutné použiť už existujúce rozhranie a tak isto na implementáciu vizualizácie bolo treba použiť rovnaký systém, ktorý vyžadoval dedenie z predpripravených abstraktných tried.

V hornej lište aplikácie bolo menu na výber dátových štruktúr a menu na výber jazyka. Systém fungoval na princípe kontajnera inštanciovaných tried udržiavajúcich jednotlivé obrazovky. Objekty mali priradenú dátovú štruktúru. Panel obsahoval plochy pre vykresľovanie, výpis komentárov (toho, čo sa bude diať) a ovládacie tlačidlá. Pri vykonaní zvolenej operácie sa spustilo nové vlákno vykonávajúce danú operáciu a obsluhujúcu obrazovku.

Všetky predošlé vizualizácie pracovali s binárnymi stromami. Naše dátové štruktúry však boli všeobecné stromy, respektíve lesy. Preto bolo nutné implementovať triedu na prácu so všeobecnými stromami.

Kapitola 5

Implementácia softvéru

V tejto kapitole postupne uvedieme realizáciu návrhu popísaného v kapitole 4. Popíšeme ako sme vizualizovali dané dátové štruktúry a algoritmy (sekcia 5.1) a ukážeme si implementáciu a ovládanie aplikácie (sekcia 5.2). Na záver uvedieme ako dopadlo testovanie softvéru (sekcia 5.3).

5.1 Vizualizácia

Dátové štruktúry sme vizualizovali rôzne. Základom bol upravený algoritmus pre tesné vykresľovanie a použitie rôznych farieb pre prehľadnejšiu vizualizáciu.

5.1.1 Union-find

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 heuristiky na spájanie a nájdenie reprezentanta množiny a aj tlačidlo na vykonanie viacerých náhodných spojení naraz. Toto je užitočné, keď chce užívateľ vidieť, ako dátová štruktúra vyzerá, po vykonaní veľa operácií. Okrem bežného textového vstupu je možné vyberať prvky aj myšou. Vybrané prvky sú zvýraznené.

Operácia *find*

Hľadanie reprezentanta množiny sme vizualizovali tak, že sme vybratý prvok označili, vyznačili sme cestu do koreňa a reprezentanta množiny (algoritmus dopredu nepozná reprezentanta, ale v rámci vizualizácie sme to považovali za vhodné). Následne sme vykonali kompresiu, aktívny vrchol je modrý. Cestu sme nechali znázornenú šedou farbou, aby bolo vidieť, ako vyzerá cesta pred a po prevedení algoritmu.

Pre heuristiky delenie a pólenie cesty sme použili ružovú farbu na označenie syna a vnuka, keďže tieto algoritmy s nimi pracujú.

Operácia *union*

Spájanie prvkov sme vizualizovali vykonaním dvoch hľadaní reprezentanta a následného napojenia reprezentantov podľa typu algoritmu.

5.1.2 Písmenkový strom

Pri vizualizácií písmenkového stromu sme použili Walkerov algoritumus 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 zakrivene, podľa Bézierovej krivky určenej štyrmi bodmi. Vo vizualizácií sa dajú vložiť náhodné slová podľa momentálne nastaveného jazyka. Taktiež sa automaticky odstraňuje diakritika a interpunkcia, takže sa dá naraz vložiť súvislý text.

Operácia *insert*

Aby bolo zreteľné, aké slovo vkladáme, znázorňujeme zbytok vkladaného slova pri mieste, kde sa práve v strome nachádzame. Časť, ktorú sme vložili a aj časť, ktorú ideme vložiť vypisujeme bielou na modrom pozadí. Miesto, na ktoré sa presunieme, prípadne kam ideme pripájať, sme označili ružovou farbou. Ukončovací znak $ sme vykreľovali tmavšie a menším písmom.

Operácia *find*

Podobne ako pri vkladaní slova, aj pri vizualizácií hľadania slova, sme použili pomocnú bublinu so zbytkom nespracovaného vstupu. Na rozdiel od vkladania sme použili ružovú farbu. Na znázornenie množiny prvkov, kde sa môže vyskytovať následujúce písmenko sme použili šedú farbu.

Operácia *delete*

Mazanie slova z písmenkového stromu sme implementovali ako nájdenie slova a následné zmazanie prípadnej mŕtvej vetvy (popísané v sekcii 2.1). Na nájdenie slova sme použili ten istý farebný princíp, ako pri operácií *find*. Mŕtvu vetvu sme zvýraznili červenou farbou.

5.1.3 Sufixový strom

Sufixový strom sme vizualizovali podobne ako písmenkový strom, použili sme Walkerov algoritmus (Walker II 1990) so zakrivenými hranami. Sufixové linky sme znázornili šedými

šípkami. Na rozdiel od písmenkového stromu sa v sufixovom strome vyskytujú na hranách reťazce. Tie sme znázornili ako viac spojených hrán a text sme spojili.

Ukkonenov algoritmus

Pri vizualizácií algoritmu označujeme hrany, pre ktoré platí prvý prípad, ružovou farbou. Hranu, pre ktorú platí prvý prípad a je pridaná ako posledná, označujeme zelenou farbou. Z tejto hrany začína „zaujímavejšia" časť rozširovania stromu. Pri pridávaní sufixov do stromu sme použili modrú farbu.

5.2 Popis ovládania

Obrazovka softvéru sa skladá z menu na výber dátových štruktúr, menu na výber jazyk, obrazovky na vykresľovanie, ovládacích tlačidiel a obrazovky na výpis komentárov.

Keď si z menu vyberieme dátovú štruktúru zobrazí sa inicializovaná dátová štruktúra a príslušné tlačidlá. Celá dátová štruktúra sa dá posúvať ťahaním myšou, približovať alebo oddľalovať použitím kolieska myši, alebo automaticky vycentrovať a priblížiť pomocou tlačidla.

5.2.1 Union-find

Vstup. Vstup zadávame do vstupného poľa alebo vyberieme prvok myšou. Prioritu majú prvky vybrané myšou. Pre operáciu *find* sa vyberie prvý prvok, pre operáciu *union* sa vyberú prvé dva prvky. Pokiaľ je zadaný menší počet prvkov náhodne sa doplnia.

Náhodné spojenie. Tlačidlo pre vykonanie viacerých náhodných spojení berie parameter – počet spojení zo vstupného poľa. Pokiaľ nie je hodnota zadaná, za parameter sa berie základná hodnota.

5.2.2 Písmenkový a sufixový strom

Vstup. Do vstupného poľa zadávame slová oddelené medzerou. Textový vstup sa pred pridávaním ošetrí: odstráni sa diakritika, interpunkcia a čísla. Následne sa text prevedie na veľké písmená a za každé slovo sa pridá ukončovací znak ($). Prázdne slovo sa do stromu dá pridať zadaním vstupného reťazca „$". Pokiaľ je vstup prázdny, vykoná sa pridanie náhodného slova podľa nastaveného jazyka.

Náhodné vloženie. Tlačidlo pre vykonanie viacerých náhodných pridaní funguje ako pri union-finde. Slová, ktoré pridá, sú vybraté z databázy slov nastaveného jazyka. Sufixový strom sa vytvára pre jedno slovo a preto nemá zmysel pridávať viacero náhodných slov.

5.3 Testovanie, prevádzka a údržba

Pri testovaní aplikácie sa nevyskytli nečakané chyby. Za najväčšie nedostatky sme považovali občas nezrozumiteľné komentáre a neoptimálne znázornenie Ukkonenovho algoritmu. Pri premietaní cez projektor sme prišli na nedostatok: niektoré farby sú príliš sýte a preto sa kontrast medzi písmom a pozadím zdá byť malý a farby príliš tmavé. Tento problém sa vyskytoval len občas, ale pri bežnom osvetlení.

Softvér nepotrebuje špecifickú údržbu, keďže sa jedná o samostatnú aplikáciu. Avšak vhodné je sledovať aktualizácie, ktoré opravujú chyby. Vývoj a aj najnovšie verzie sú voľne dostupné na systéme pre správu zdrojových kódov Github, konkrétne na webovej stránke: `https://github.com/kuk0/alg-vis`. Stabilná verzia s popisom algoritmov sa nachádza na stránke: `http://people.ksp.sk/~kuko/gnarley-trees`. Na oboch sídlach sa nachádzajú kontakty, ktorými môže používateľ upozorniť na chyby v aplikácií.

Záver

V práci sme popísali dátové štruktúry union-find, písmenkový strom a sufixový strom (kapitoly 1, 2 a 3). Spísali sme návrh a implementáciu softvéru. Softvér sme implementovali podľa návrhu a splnili sme všetku funkcionalitu: vizualizovali sme všetky dátové štruktúry a algoritmy, doplnili funkčnosť, spravili komentáre v angličtine a slovenčine.

V ďalšom vývoji by sme chceli prerobiť komentáre pre sufixové stromy, keďže sa nám zdajú byť neprehľadné. Taktiež by sme chceli zvýrazniť niektoré časti Ukkonenovho algoritmu a pridať nové algoritmy vymenované v sekcií 3.3.

Softvér by sme chceli v budúcnosti rozšíriť o nové dátové štruktúry, ktoré sa týkajú stringológie, napríklad *radix tree*, *PATRICIA*, *orientovaný acyklický graf pre slová (DAWG)*, *všeobecný sufixový strom*.

Chceli by sme projekt podstúpiť širšiemu použitiu a zozbierať viac spätnej väzby, aby sme softvér odladili podľa používateľských potrieb a opravili chyby, ktoré sme neobjavili.

Literatúra

Appel, A. W. a G. J. Jacobson (1988). "The world's fastest Scrabble program". In: *Communications of the ACM* 31.5, str. 572–578.

Fredkin, Edward (sep. 1960). "Trie memory". In: *Commun. ACM* 3.9, str. 490–499. ISSN: 0001-0782. DOI: 10.1145/367390.367400. URL: http://doi.acm.org/10.1145/367390.367400.

Gabow, H.N. a R.E. Tarjan (1985). "A linear-time algorithm for a special case of disjoint set union". In: *Journal of computer and system sciences* 30.2, str. 209–221.

Gilbert, J.R., E.G. Ng a B.W. Peyton (1994). "An efficient algorithm to compute row and column counts for sparse Cholesky factorization". In: *SIAM Journal on Matrix Analysis and Applications* 15, str. 1075.

Gusfield, Dan (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York, NY, USA: Cambridge University Press. ISBN: 0-521-58519-8.

Harel, D. a R.E. Tarjan (1984). "Fast algorithms for finding nearest common ancestors". In: *SIAM Journal on Computing* 13, str. 338.

Hopcroft, John E. a Jeffrey D. Ullman (1973). "Set Merging Algorithms". In: *SIAM J. Comput.* 2.4, str. 294–303. URL: http://dx.doi.org/10.1137/0202024.

Hundhausen, C.D., S.A. Douglas a J.T. Stasko (2002). "A meta-study of algorithm visualization effectiveness". In: *Journal of Visual Languages & Computing* 13.3, str. 259–290.

Knight, Kevin (mar. 1989). "Unification: a multidisciplinary survey". In: *ACM Comput. Surv.* 21.1, str. 93–124. ISSN: 0360-0300. DOI: 10.1145/62029.62030. URL: http://doi.acm.org/10.1145/62029.62030.

Kováč, Jakub (2007). "Vyhľadávacie stromy a ich vizualizácia". Bakalárska práca, Univerzita Komenského v Bratislave.

Kruskal, Joseph B (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". In: *Proceedings of the American Mathematical Society* 7.1, str. 48–50. URL: http://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/S0002-9939-1956-0078686-7.pdf.

Leeuwen, J. a Th.P. van der Weide (sep. 1977). *Alternative Path Compression Rules*. Tech. spr. An outline of the results were presented at the Fachtagung on Algorithms and Complexity Theory, Oberwolfach, Oct 1977. University of Utrecht, The Netherlands.

Liang, F. M. (1983). "Word hy-phen-a-tion by com-put-er". Diz. práca. Stanford, CA 94305: Stanford University.

Lucchesi, Cláudio L. a Tomasz Kowaltowski (1992). *Applications of Finite Automata Representing Large Vocabularies*.

McCreight, Edward M. (apr. 1976). "A Space-Economical Suffix Tree Construction Algorithm". In: *J. ACM* 23.2, str. 262–272. ISSN: 0004-5411. DOI: 10.1145/321941.321946. URL: http://doi.acm.org/10.1145/321941.321946.

Morrison, Donald R. (okt. 1968). "PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4, str. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481. URL: http://doi.acm.org/10.1145/321479.321481.

Naps, T.L. a kol. (2002). "Exploring the role of visualization and engagement in computer science education". In: *ACM SIGCSE Bulletin*. Č. 35. 2. ACM, str. 131–152.

Saraiya, Purvi a kol. (2004). "Effective features of algorithm visualizations". In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. SIGCSE '04. New York, NY, USA: ACM, str. 382–386. ISBN: 1-58113-798-2. DOI: http://doi.acm.org/10.1145/971300.971432. URL: http://doi.acm.org/10.1145/971300.971432.

Shaffer, Clifford A. a kol. (aug. 2010). "Algorithm Visualization: The State of the Field". In: *Trans. Comput. Educ.* 10 (3), 9:1–9:22. ISSN: 1946-6226. DOI: http://doi.acm.org/10.1145/1821996.1821997. URL: http://doi.acm.org/10.1145/1821996.1821997.

Sinha, Ranjan a Justin Zobel (dec. 2004). "Cache-conscious sorting of large sets of strings with dynamic tries". In: *J. Exp. Algorithmics* 9. ISSN: 1084-6654. DOI: 10.1145/1005813.1041517. URL: http://doi.acm.org/10.1145/1005813.1041517.

Sinha, R., D. Ring a J. Zobel (2006). "Cache-efficient String Sorting using Copying". In: *J. Exp. Algorithmics* 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*, str. 93–104.

Tarjan, Robert E. a Jan van Leeuwen (mar. 1984). "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2, str. 245–281. ISSN: 0004-5411. DOI: 10.1145/62.2160. URL: http://doi.acm.org/10.1145/62.2160.

Tarjan, Robert Endre (1983). *Data structures and network algorithms*. Philadelphia, PA, USA: Society for Industrial a Applied Mathematics. ISBN: 0-89871-187-8.

Ukkonen, Esko (1995). *On-Line Construction of Suffix Trees*.

Walker II, John Q. (1990). "A node-positioning algorithm for general trees". In: *Software: Practice and Experience* 20.7, str. 685–705.

Weiner, P. (1973). "Linear pattern matching algorithms". In: *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*. IEEE, str. 1–11.