

Fakulta matematiky, fyziky a informatiky
Univerzita Komenského
Bratislava



Vzdialenosť medzi syntaktickými stromami

Diplomová práca

Autor: Adrián Kočiš
Diplomový vedúci: RNDr. Ján Štunc, CSc.

Bratislava 2005

Čestné prehlásenie

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne len s použitím uvedenej literatúry.

V Bratislave 26. apríla 2005

.....
Adrián Kočiš

Pod'akovanie

Ďakujem svojmu vedúcemu diplomovej práce, RNDr. Jánovi Šturcovi, CSc., za cenné rady a pripomienky pri písaní tejto práce.
Taktiež ďakujem svojim rodičom a blízkym za všestrannú podporu počas celého štúdia.

Obsah

Obsah	4
1. Úvod.....	5
Motivácia	5
Základné pojmy	7
Syntaktická analýza	7
Porovnávanie reťazcov a stromov	11
2. Cieľ	13
3. Súčasný stav	14
Editovacia vzdialenosť a reťazce	14
Wagner-Fisher.....	14
Zrýchlenie pomocou techniky „štyroch Rusov“	15
Myers	16
Úprava Myersovho algoritmu na lineárny priestor	18
Editovacia vzdialenosť a stromy	20
Selkow.....	20
Tai	21
Zhang-Shasha.....	23
EVS	25
FMES	27
mmdiff.....	32
XYDiff	35
MH-diff.....	36
X-diff.....	37
4. Implementácia.....	38
5. Zhodnotenie a ďalší vývoj	49
6. Záver	51
7. Slovník pojmov	52
8. Použitá literatúra	54
Prílohy.....	56
Inštalácia	56
Obsah CD.....	56
Príklady použitia	57

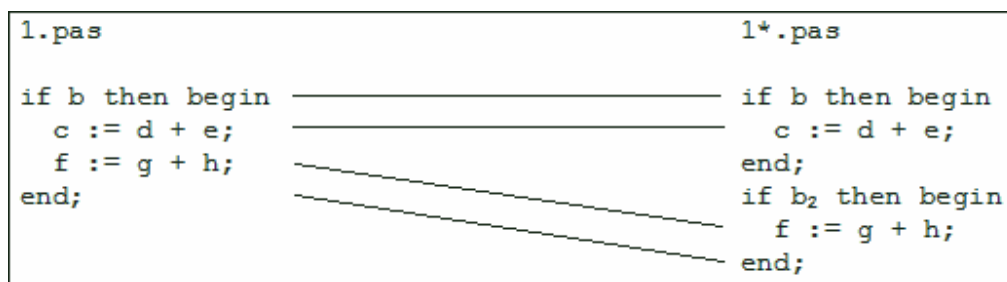
1. Úvod

Motivácia

V dnešnej dobe, keď sme zavalení obrovským množstvom informácií, je nevyhnutné, aby sme ich nejakým spôsobom triedili a filtrovali. Keď už niečo zaujímavé nájdeme, napr. webovú stránku, obvykle ju nejakto spracujeme a získame z nej informácie (prečítame, vytlačíme, zaarchivujeme, alebo iba poznačíme), teda z nej máme v danom čase spravený „snapshot“. Avšak väčšina informácií sa časom zvykne meniť, doplňovať a teda po čase, keď sa znova vrátime k tejto našej informácii (ktorou môže byť napríklad nejaký „online“ manuál), tak získame novú verziu tejto informácie. Obvykle z nej nechceme znovu vyťahovať to podstatné, ale zaujíma nás, čo sa od minula zmenilo. Inak povedané, či prišlo k nejakej zmene medzi dvomi informáciami, a ak k nejakej prišlo, tak k akej. Môže ísť napr. o zdrojové kódy, dokumenty, webové stránky, alebo nejaké údaje v databáze. Zisťovanie zmien v dátach je využívané v mnohých oblastiach nielen informatiky, ale napríklad aj v biológii (genetické reťazce).

Jedným z mnohých využití je v RCS a CVS systémoch, v ktorých sa manažujú a zhromažďujú zdrojové kódy, dokumenty a ďalšie artefakty súvisiace s vývojom softvérového projektu. Takýto systém ukladá všetky verzie týchto súborov tak, aby bolo možné sa v prípade nájdenej chyby vrátiť k staršej ešte funkčnej verzii, keďže pri opravovaní chýb sa vytvoria obvyčajne chyby nové, ktoré v starších verziách ešte neboli. Je úplne bežné, že daný projekt má súčasne vyvíjaných viacero vetiev, napr. jedna je stabilná a už sa do nej nepridávajú žiadne nové funkcie, ale už je pomerne bez chýb a druhá do ktorej sa pridávajú nové veci. A teda sa v týchto systémoch uchováva veľké množstvo verzií tých istých (mierne pozmenených) zdrojových súborov a preto je rozumné, kvôli výraznému šetreniu priestorom, miesto nich uchovávať iba zmeny medzi jednotlivými verziami. Navyše obvykle na danom projekte pracuje viacero ľudí naraz, ktorí tieto súbory modifikujú a neraz viacerí modifikujú ten istý súbor v rovnakom čase. Problém súčasného zápisu do tých istých súborov je obvykle riešený tým, že si každý stiahne lokálnu kópiu a na nej vykoná zmeny a tie pošle naspäť do systému. Ak sa pridajú dve konfliktné verzie (t.j. úprava toho istého súboru viacerými vývojármi súčasne), tak sa tieto musia spojiť (merging) do výsledku. Táto operácia je automatizovaná, pokiaľ nenastanú vážne konflikty a tu sa opäť využíva zisťovanie rozdielov (differencing) a následná oprava do konzistentného tvaru, ktorý obsahuje časti oboch vstupov (patching).

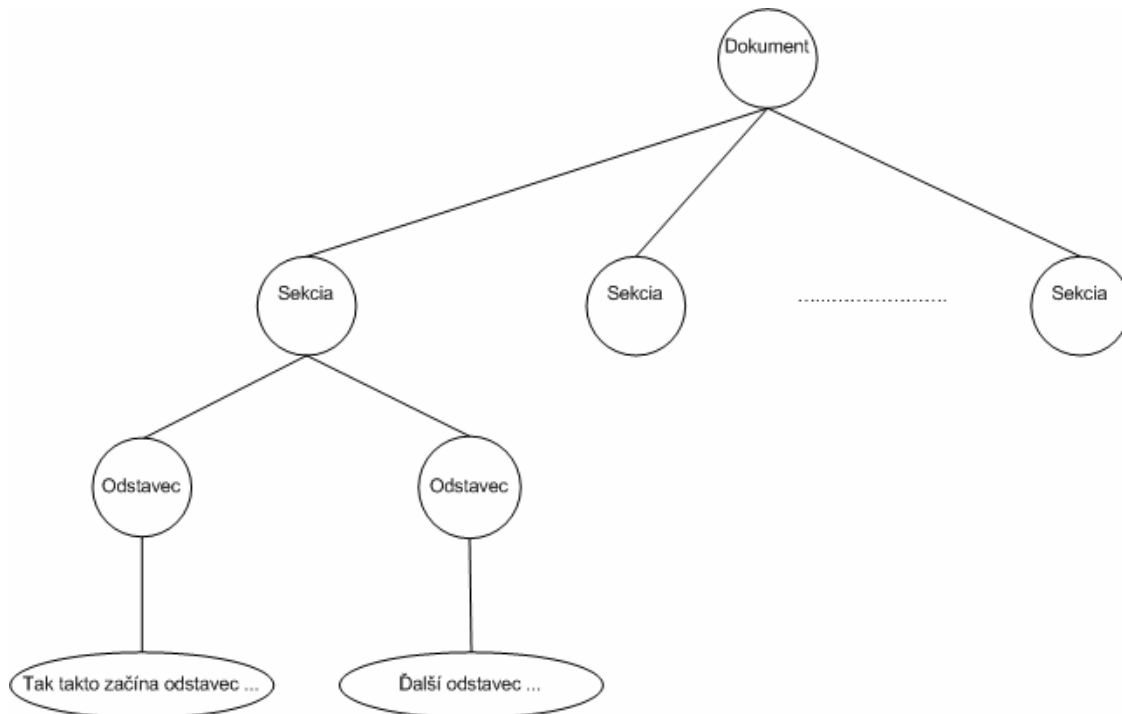
Avšak väčšinou sa zdrojové kódy, dokumenty pri zisťovaní zmien považujú iba za textové reťazce a rozdeľia sa obvyčajne na riadky, ktoré sa následne analyzujú a zisťujú sa rozdiely. Tento postup nie je úplne optimálny, keďže dokumenty, alebo zdrojové kódy majú obvykle svoju štruktúru, ktorá nám môže pomôcť pri zisťovaní rozdielov medzi nimi.



Obr. 1: Mapovanie medzi dvoma zdrojovými súbormi

Pri pohľade na obrázok č. 1, plné čiary označujú zhodné riadky, po vyhodnotení porovnávania na základe porovnávania vzdialenosti reťazcov (edit distance) medzi týmito dvoma súbormi (1*.pas je nová verzia súboru 1.pas). Výsledkom je vloženie konca prvého bloku (prvý end), a začiatku druhého bloku (ten druhý if), čo však nie je úplne najlepšie z hľadiska jeho štruktúry. Najvhodnejšie by bolo vymazať druhé priradenie z 1.pas a potom bude 1.pas zhodný s prvým blokom v 1*.pas a zostáva iba vloženie druhého bloku.

A preto je zaujímavé sa zaoberať aj štruktúrou (hierarchickým usporiadaním) textu. Túto môžeme reprezentovať pomocou stromu, kde všetky listy obsahujú celý vstupný text. Vo vnútorných vrcholoch sa už žiaden text nenachádza, tieto vrcholy popisujú štruktúru vstupného textu. Ako je možné vidieť napr. na Obr. 2.



Obr. 2: Jednoduchá štruktúra textového dokumentu

Ďalšími možnými využitiami štruktúrneho porovnávania je napr. redukcia množstva dát, či už pri ukladaní na disk, alebo prenášaných dát pri prenosoch cez internet, pri cacheovaní a podobne.

Táto práca vychádza z konceptov na budovanie kompilátorov, porovnávanie reťazcov a stromov. Snaha bola o prakticky použiteľný program na štruktúrne porovnávanie súborov na základe vstupnej gramatiky.

Vo zvyšku prvej časti sa nachádza stručný úvod do problematiky a definície, ktoré sa budú neskôr v ďalšom texte využívať. Mala by uviesť čitateľa do problematiky

Druhá časť je venovaná cieľom práce a špecifikácii návrhu programu.

V tretej časti sú popísané súčasné výsledky a algoritmy používané na porovnávanie reťazcov a stromov.

Štvrtá časť obsahuje popis riešenia a implementácie.

Piata časť je venovaná zhrnutiu výsledkov a dosiahnutým cieľom.

Základné pojmy

V tejto kapitole sa nachádza stručný úvod do problematiky syntaktickej analýzy a porovnávania reťazcov, resp. stromov.

Syntaktická analýza

Na úvod zopár definícií:

Abeceda je konečná neprázdna množina symbolov, označuje sa písmenom Σ .

Slovo v abecede Σ je konečná postupnosť symbolov zo Σ . Prázdna postupnosť sa označuje písmenom ε (prázdne slovo).

Jazyk v abecede je množina slov v abecede Σ .

Frázová gramatika je štvorica $G = (N, T, P, \sigma)$, kde N a T sú abecedy nerterminálnych a terminálnych symbolov ($N \cap T = \emptyset$), $\sigma \in N$ je počiatočný neterminál a $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ je konečná množina pravidiel.

Vetná forma je v podstate slovo z $(N \cup T)^*$, ktoré môžeme získať odvodzovaním z počiatočného neterminálu.

Krok odvodenia v gramatike G je binárna relácia \Rightarrow na množine vetných foriem, kde $x \Rightarrow y$ práve vtedy, keď $\exists w_1, w_2$ a pravidlo $u \rightarrow v \in P$: také, že $x = w_1 u w_2$ a $y = w_1 v w_2$.

Jazyk generovaný gramatikou G je množina $L(G) = \{w \in T^* \mid \sigma \Rightarrow^* w\}$.

Ľavé krajné odvodenie, resp. pravé krajné odvodenie je odvodenie (postupnosť krokov odvodenia), v ktorom sa v každom kroku odvodenia nahrádza najľavejší, resp. najpravejší neterminál.

Chomského hierarchia gramatík:

- Typ 0 – frázové gramatiky, boli definované vyššie, sú najvšeobecnejšie gramatiky (generujúce rekurzívne vyčísliteľné jazyky).
- Typ 1 – kontextové gramatiky sú frázové gramatiky v ktorých pre všetky pravidla $u \rightarrow v$ platí, že $|u| \leq |v|$ (okrem prípadného pravidla $\sigma \rightarrow \varepsilon$, v rozšírených kontextových gramatikách).
- Typ 2 – bezkontextové gramatiky sú frázové gramatiky v ktorých je množina pravidiel definovaná nasledovne $P \subseteq N \times (N \cup T)^*$
- Typ 3 – regulárne gramatiky sú frázové gramatiky v ktorých je množina pravidiel definovaná nasledovne $P \subseteq N \times (T^* \cup T^* N)^*$

Iným, ale rovnako silným formalizmom na popis jazykov sú automaty (resp. Turingov stroj). Kvôli jednoduchosti definujem iba deterministický konečný automat.

Deterministický konečný automat je päťica $A = (K, \Sigma, \delta, q_0, F)$, kde K je konečná množina stavov, Σ je vstupná abeceda, $q_0 \in K$ je počiatočný stav, $F \subseteq K$ je množina akceptujúcich stavov a $\delta: K \times \Sigma \rightarrow K$ je prechodová funkcia.

Konfigurácia deterministického konečného automatu je dvojica $(q, w) \in K \times \Sigma^*$, kde q je stav v automate, w je zvyšok vstupu.

Krok výpočtu deterministického konečného automatu je relácia \vdash_A na konfiguráciách $(q, av) \vdash_A (p, v) \Leftrightarrow p = \delta(q, a)$.

Jazyk akceptovaný deterministickým konečným automatom A je množina $L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash_A^* (q, \varepsilon), q \in F\}$:

Ku každej gramatike existuje automat, ktorý akceptuje všetky slová, ktoré gramatika generuje a naopak – $L(G) = L(A)$. Pri regulárnej gramatike je to deterministický konečný automat, pri bezkontextovej je to nedeterministický zásobníkový automat, atď., pri frázovej gramatike je to Turingov stroj.

Pri tvorbe prekladačov a kompilátorov potrebujeme vedieť nielen či dané slovo patrí do jazyka, ale potrebujeme poznať aj jeho strom odvodenia. A najmä toto je jadrom

syntaktickej analýzy – hľadanie stromu odvodenia pre dané slovo z jazyka. V syntaktickej analýze sa zaoberáme najmä bezkontextovými a regulárnymi jazykmi hlavne preto, lebo ich vieme deterministicky rozpoznávať v polynomiálnom čase (podtriedy bezkontextových jazykov dokonca v lineárnom čase).

Všeobecné metódy na rozpoznávanie bezkontextových jazykov: CYK a Earleyho algoritmus pracujú v čase $O(n^3)$, Earleyho algoritmus dokonca v čase $O(n^2)$ ak je gramatika jednoznačná (t.j. neexistuje v $L(G)$ slovo, ktoré by malo dva rôzne stromy odvodenia).

V lineárnom čase dokážeme rozpoznávať iba podtriedy bezkontextových jazykov (napr. $LL(k)$, alebo $LR(k)$ jazyky), kde k je počet symbolov, ktoré vidíme dopredu (lookahead), ak je $k = 1$, tak sa zvykne nepísať.

Pre $LL(k)$ gramatiku vieme zostrojiť $LL(k)$ parser, ktorý rozpoznáva vstupné slová z jazyka v lineárnom čase metódou zhora-nadol. Podobne pre $LR(k)$ gramatiku vieme zostrojiť $LR(k)$ parser, ktorý rozpoznáva vstupné slová z jazyka v lineárnom čase metódou zdola-nahor. Algoritmy na konštrukciu LL a LR parserov sú uvedené v [1].

V skratke je rozdiel medzi LL a LR parserom nasledovný: LL parser sa musí rozhodnúť, podľa ktorého pravidla bude redukovať už po prečítaní prvého symbolu pravej strany daného pravidla. LR parser sa rozhoduje pre pravidlo až po prečítaní prvého symbolu, ktorý patrí do pravej strany nasledovného pravidla.

Keďže sa v programe využíva LR parser (resp. jeho variant – $LALR$ parser), tak nasledujúcich pár riadkov bude venovaných jeho popisu.

Vstupom do LR parseru bývajú tokeny z lexikálneho analyzátora (obyčajne DFA, ktorý bol vybudovaný na základe regulárnych výrazov popisujúcich možné tokeny vo vstupe). Tento lexikálny analyzátor číta vstup a vracia tokeny, ktoré sa v ňom nachádzajú (obyčajne najväčšie možné, ak máme na vstupe napr. `procedure`, tak nevráti 9 tokenov typu identifikátor - `'p','r','o','c','e','d','u','r','e'`, ale jeden token typu `procedúra` - `'procedure'`). Túto činnosť by mohol robiť aj samotný parser, keďže je to vlastne zásobníkový automat (a simulácia konečného automatu je na ňom bezproblémová), ale kvôli prehľadnosti sa takmer vždy využíva separátny lexikálny analyzátor (inak tomu nie je ani v tomto programe).

Predpokladajme, že máme zostrojené parsovacie tabuľky pre operáciu posun (shift) a redukuj (reduce), potom LR parser (aj SLR , $LALR$ parser, rozdiel je iba v parsovacích tabuľkách) funguje nasledovne (je popísané fungovanie LR parsera aj s budovaním syntaktického stromu):

```
stack.push(0)    //do zásobníka sa vloží počiatočný stav
bNext := true
bAccept := true
repeat
    state := stack.pop
```

```

if bNext
    token := getnexttoken
    bNext := false
if gotoTable(state,token.type,newstate) then
    //shift, ak by pre daný stav bola možná aj redukcia,
    //tak nastal shift/reduce konflikt a vyberie sa shift
    bNext := true
    stack.push(token)
    stack.push(newstate)
else if reduceTable(state,token.type,rulenum) then
    //reduce, ak je pre daný stav možných viac redukcií,
    //tak nastal reduce/reduce konflikt a vyberie sa
    //pravidlo s najdlhšou pravou stranou
    if rulenum = 0 then
        bAccept := true
    new(children)
    for i := 1 to (dĺžka pravej strany rulenum pravidla) do
        stack.pop(newstate)
        stack.pop(token)
        children.add(token)
    newstate := stack.top
    new(token)
    token.child := children.first
    token.type := (typ ľavej strany rulenum pravidla)
    state := gotoTable(newstate, token.type, newstate)
    stack.push(token)
    stack.push(newstate)
else
    //chyba, pre tento token nie je definovaný ani posun
    //ani redukcia, čiže treba to nejako vyriešiť, napr.
    //skúsi nasledujúci token a tento odignoruje, alebo
    //skúsi vložiť na vstup token, podľa ktorého vieme
    //v tomto stave redukovať
    bNext := true
until (token.type = EOF) and ((bNext) or (bAccept))
//čiže sme skončili so vstupom, aj s možnými redukciami pre
//posledný token
//po skončení tohto algoritmu: buď bol vstup akceptovaný
//(bAccept = true) a token je koreňom syntaktického stromu,
//alebo bol vstup neakceptovaný (kvôli nejakej chybe, potom
//bAccept = 0), a na zásobníku nám zostali nejaké podstromy
//výsledného syntaktického stromu

kde stack je zásobník LR parsera
getnexttoken je funkcia lexikálneho analyzátoru, ktorá
    vracia nasledujúci token zo vstupu
gotoTable je funkcia pracujúca nad tabuľkou posunov,

```

ktorá pre stav a typ tokenu vráti nový stav po aplikácii operácie posunu (shift)
reduceTable je funkcia pracujúca nad tabuľkou redukcií, ktorá pre stav a typ tokenu vráti číslo pravidla, podľa ktorého sa bude redukovať (reduce)
children je zreťazený zoznam vrcholov (tokenov)
token je potenciálny vrchol budovaného syntaktického stromu (token.type je typ tokenu, tak ako nám ho vráti lexikálny analyzátor, token.child je prvý syn tohto vrchola v syntaktickom strome)

Konštrukcia tabuľky posunov a tabuľky redukcií je popísaná v štvrtej kapitole.

LR parser má obvykle veľké množstvo stavov (každý stav v LR parseri reprezentuje nejakú množinu čiastočne použitých pravidiel v pôvodnej gramatike, kde bodka označuje pokiaľ sme sa v danom pravidle dostali, ak je na konci, tak sa dá podľa daného pravidla redukovať vetnú formu v zásobníku), ktoré sa líšia iba v lookahead symboloch (teda majú rovnaké položky). Položka (resp. LR0 položka) je pravidlo gramatiky s bodkou na pravej strane (táto určuje ako ďaleko sme sa v danom pravidle dostali pri syntaktickej analýze vstupu). LR0 položka spolu s lookaheadom sa niekedy zvykne nazývať LR položka. LALR parser je LR parser, v ktorom sú stavy s rovnakými položkami a iba rôznym lookaheadom zlúčené (môže mať aj 100 krát menej stavov ako pôvodný LR parser). Jediným problémom je, že môžu vzniknúť nové konflikty, keď pre daný lookahead token budeme mať viac pravidiel, ktoré môžeme použiť (reduce/reduce konflikt), v tom prípade sa nejedná o LALR gramatiku. Trieda LALR jazykov je vlastnou podmnožinou triedy LR jazykov, čiže existujú jazyky ktoré sú LR, ale nie sú LALR (t.j. neexistuje žiadna LALR gramatika, ktorá ich generuje). Napr.:

```
S -> Aa | bAc | Bc | bBa
A -> d
B -> d
```

Na druhej strane, keďže prakticky všetky zmysluplné konštrukcie v programovacích jazykoch sa dajú popísať LALR gramatikou, tak nám tento fakt nejako výrazne nevaďí.

Porovnávanie reťazcov a stromov

Základným prístupom k porovnávaniu reťazcov, resp. stromov býva zadefinovanie množiny elementárnych operácií ktoré transformujú znaky, resp. vrcholy. Postupnosť takýchto elementárnych operácií, ktorá transformuje jeden vstupný reťazec, resp. strom na druhý reťazec, resp. strom sa nazýva edit script. Niekedy stačí definovať editovaciu vzdialenosť priamo z dĺžky tejto postupnosti operácií. Obyčajne sa však priradí váhová (resp. cenová) funkcia ku každému typu elementárnej operácie a pri výpočte vzdialenosti sa tieto váhy sčítavajú a hľadá sa maximálna (resp. minimálna) postupnosť

operácií vzhľadom na súčet váh jednotlivých operácií v nej. Takáto postupnosť sa nazýva minimálny edit script.

Medzi elementárne operácie pri porovnávaní reťazcov obyčajne patria:

- vloženie znaku
- vymazanie znaku
- upravenie znaku

Napr.: Uvažujme len o dvoch elementárnych operáciách (vloženie a vymazanie znaku):

nech $A = ababbbb$

a $B = bababb$

Potom je najmenším edit scriptom postupnosť $(\text{vymaž}, 0, a), (\text{vlož}, 3, a), (\text{vymaž}, 6, b)$ a teda vzdialenosť týchto dvoch reťazcov je 3.

Ak pridáme aj tretiu operáciu upravenie znaku, tak je možné pretransformovať postupnosť A na postupnosť B dvomi operáciami: $(\text{vymaž}, 0, a), (\text{zmeň}, 4, a)$, čiže vzdialenosť je 2.

Na predošlom príklade bolo vidieť, že bohatšia množina elementárnych operácií nám umožňuje nájsť kratšie edit skripty. Na druhú stranu sa takéto edit skripty ťažšie získavajú, čo bude vidieť neskôr pri algoritmoch na porovnávanie reťazcov a stromov.

Pri porovnávaní stromov zvyknú byť elementárnymi operáciami, nasledovné operácie:

- vloženie uzla
- vymazanie uzla
- upravenie uzla
- vloženie podstromu
- vymazanie podstromu
- presun podstromu
- výmena súrodencov v rámci spoločného podstromu

Viac informácií o algoritmoch sa nachádza v kapitole č. 3.

2. Cieľ

Cieľom diplomovej práce bolo spraviť prakticky použiteľný program, ktorý na základe danej syntaxe porovná dva vstupné súbory a vyznačí rozdiely medzi nimi.

Základné alternatívy riešenia sú v podstate dve:

- Simultánnu syntaxou riadený preklad a porovnávanie

V tejto alternatíve by sa najprv vygeneroval syntaktický analyzátor (parser) pre vstupnú gramatiku. Následne by sa analyzovali oba súbory paralelne a hneď by sa medzi nimi robilo aj porovnávanie a vytváral by sa edit script. Hlavným problémom tohto postupu je synchronizácia medzi analýzou v jednom a druhom súbore. Použitím LL parserov (namiesto viacej všeobecných LR parserov), by sa mohlo čiastočne zlepšiť, ale aj tak by si to vyžadovalo predspracovanie oboch vstupov, čo už je vlastne podobné ako druhá alternatíva.

- Vygenerovanie syntaktických stromov a ich následné porovnanie

V tejto alternatíve sa najprv vygeneruje syntaktický analyzátor (parser) pre vstupnú gramatiku (v tomto prípade LR, resp. LALR ako bude ukázané neskôr). Potom postupne k jednému aj druhému vstupnému súboru tento parser vygeneruje syntaktický (parsovací) strom. Následne sa stromy porovnajú a vygeneruje sa postupnosť elementárnych operácií transformujúca prvý strom na druhý strom (edit script).

Po zvážení pre a proti som si vybral druhú alternatívu, čiže vytvoriť pre oba vstupné súbory ich syntaktické stromy, tieto následne porovnať a zobraziť rozdiely. Postup sa dá rozložiť do niekoľkých po sebe idúcich krokov:

- načítanie vstupnej gramatiky
- vytvorenie lexikálneho analyzátora na základe definícií v gramatike
- vytvorenie syntaktického analyzátora
- načítanie vstupných súborov
- vytvorenie syntaktických stromov pre jednotlivé súbory
- porovnanie syntaktických stromov a vytvorenie edit scriptu
- vyznačenie / vypísanie zmien
- prípadné uloženie zmien do výstupného súboru

3. Súčasný stav

Editovacia vzdialenosť a reťazce

Hľadanie editovacej vzdialenosti reťazcov, alebo inak povedané hľadanie najkratšieho (minimálneho) edit scriptu (t.j. minimálnej postupnosti operácií vloženia a vymazania prvku z postupnosti, tak aby po aplikovaní tejto postupnosti operácií nám vznikla z postupnosti A postupnosť B) medzi dvomi reťazcami (kde jeho dĺžka je tá hľadaná vzdialenosť) je duálny problém k hľadaniu najdlhšej spoločnej podpostupnosti. Ak nájdeme najdlhšiu spoločnú podpostupnosť, tak najkratšia postupnosť operácií bude obsahovať vymazanie všetkých prvkov z postupnosti A, ktoré sa nenachádzajú v najdlhšej spoločnej podpostupnosti a vloženie všetkých prvkov z postupnosti B, ktoré sa nenachádzajú v najdlhšej spoločnej podpostupnosti.

Wagner-Fisher

Porovnávanie reťazcov, zisťovanie najdlhšej spoločnej postupnosti sú intenzívne skúmané už od 70.-tych rokov. Vďaka tomu, že sa problém najdlhšej spoločnej podpostupnosti (LCS - longest common subsequence) vyznačuje vlastnosťou optimálnej podštruktúry, sa dá pomerne efektívne riešiť dynamickým programovaním. Nech $A = a_1a_2...a_n$ a $B = b_1b_2...b_m$ sú postupnosti, ktorých najdlhšiu spoločnú podpostupnosť hľadáme. Algoritmus vytvorí $(n+1) * (m+1)$ maticu (označme ju D) do ktorej sa postupne ukladajú vzdialenosti medzi prefixami postupností A a B, kde v bunke matice so súradnicami i, j je vzdialenosť medzi A_i a B_j (A_i znamená prefix A obsahujúci prvých i prvkov). Číže výsledná vzdialenosť sa nachádza v bunke so súradnicami n, m . Táto matica sa postupne zaplňuje zľava doprava, zhora nadol. Pre každé políčko vyberáme minimálnu z troch možností, buď bude najmenšia vzdialenosť A_i, B_j rovná najmenšej vzdialenosti A_{i-1}, B_j plus jedna (vymazanie a_i), alebo bude rovná najmenšej vzdialenosti A_i, B_{j-1} plus jedna (vloženie b_j), alebo bude rovná najmenšej vzdialenosti A_{i-1}, B_{j-1} (ak $a_i = b_j$, inak plus nekonečno). Tento algoritmus a jeho mierne obmeny je dnes už v podstate ľudovým algoritmom. Podľa prvej publikácie [12] je tiež známy ako Wagner-Fisherov algoritmus (na hľadanie minimálnej editovacej vzdialenosti):

Wagner-Fisherov algoritmus

```
D[0,0] = 0
for i := 1 to n do
    D[i,0] := D[i-1,0] + cdelete(ai)
for j := 1 to m do
    D[0,j] := D[0,j-1] + cinsert(bj)
for i := 1 to n do
    for j := 1 to m do
```

```

        m1 := D[i-1,j-1] + cupdate(ai,bj)
        m2 := D[i-1,j] + cdelete(ai)
        m3 := D[i,j-1] + cinsert(bj)
        D[i,j] := min(m1,m2,m3)
result := D[n,m]

kde A,B sú vstupné postupnosti dĺžky n,m(reťazce)
cdelete,cinsert,cupdate sú funkcie vracajúce cenu
jednotlivých operácií (ak položíme
cdelete,cinsert = 1 a cupdate = nekonečno, ak
nie sú ai a bj zhodné, inak 0, tak získame
editovaciú vzdialenosť medzi A a B, čiže
riešenie LCS)
D je matica obsahujúca editovacie vzdialenosti
medzi prefixami A a B

```

Konkrétnu postupnosť operácií získame, ak prejdeme cez maticu D od indexu n,m a hľadáme cestu najmenšieho odporu. Pri každom kroku sa vyberá z troch možností tá, ktorá sa pri výpočte musela zvoliť. Následne pridáme operáciu pred začiatok doterajšieho edit scriptu (ide sa odzadu, lebo by sme inak získali edit script odzadu). Keď dorazíme do 0,0 tak máme získaný celý minimálny edit script. Časová zložitosť tohto prechodu je zjavne $O(n + m)$.

Čo sa týka celkovej časovej zložitosti, tak je tento algoritmus $O(nm)$, resp. $O(n^2)$ (ak uvažujeme o rovnako dlhých postupnostiach). Nároky na priestor sú tiež $O(nm)$, ale miernou modifikáciou sa dá docieľiť priestorová zložitosť $O(n + m)$ [13].

Zrýchlenie pomocou techniky „štyroch Rusov“

Časová zložitosť v najhoršom prípade sa dá mierne zlepšiť elegantným rozdelením na bloky o veľkosti $t = \log n/4$ a vypočítaním editovacej vzdialenosti nad týmito blokmi, čo si vyžaduje čas $O((n/t)*(n/t))$. Ku každému bloku je nutné vypočítať aj editovaciú vzdialenosť v rámci daného bloku, to vieme urobiť v čase $O(t*t)$. Takto dostaneme rovnakú zložitosť v najhoršom prípade ako v predošlom algoritme:

$$O((n/t)*(n/t)*(t*t)) = O(n^2)$$

Ale ak si predspracujeme editovacie vzdialenosti pre všetky možné bloky $t*t$, čo vieme urobiť v čase $n*\lg^2 n$ (aj vďaka vhodne zvolenému t), tak ku každému budeme schopní nájsť editovaciú vzdialenosť v čase $O(t)$. Samozrejme aby to platilo, musí platiť pár podmienok: abeceda vstupu musí byť konečná a operácie musia byť logaritmické vzhľadom na ich dĺžku v bitovom zápise (čo však je obvykle pravda v RAM strojoch, t.j. aj v počítačoch). Čiže celková zložitosť bude:

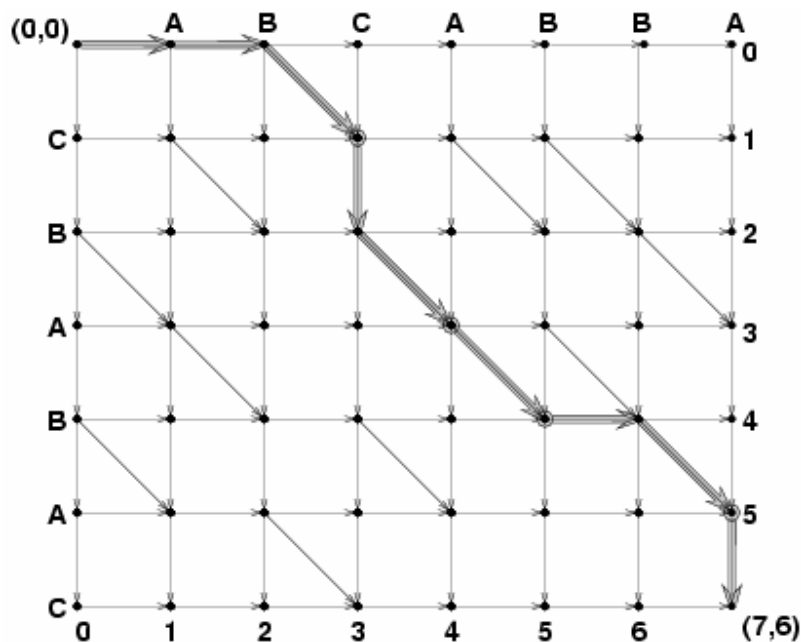
$$O((n/t)*(n/t)*t + n*\lg^2 n) = O(n^2/t) = O(n^2/\lg n)$$

Viac informácií sa dá nájsť v [15].

Myers

Avšak $O(n^2/\lg n)$ je stále ešte príliš mnoho času na porovnávanie väčších postupností. Našťastie sa to dá v priemernom prípade zlepšiť na lineárny čas [2], dokonca s použitím iba lineárneho priestoru [2]. Na druhú stranu tieto algoritmy majú v najhoršom prípade zložitosť väčšiu ako $O(n^2)$, ale na praktické využitie sú vhodné, lebo v prípadoch, v ktorých je rozdiel medzi porovnávanými reťazcami malý, je ich očakávaný čas $O(n)$.

Nech $A = a_1a_2\dots a_n$ a $B = b_1b_2\dots b_m$ sú vstupné postupnosti, ktorých editovaciu vzdialenosť hľadáme. Vytvoríme graf pre postupnosti A a B, čo bude vlastne mriežka pozostávajúca z vrcholov (x,y) , kde $x \in \{0,\dots,n\}$ a $y \in \{0,\dots,m\}$. Tieto vrcholy budú pospájané horizontálnymi a vertikálnymi hranami tak, aby skutočne vytvorili tú mriežku. Navyše ak $a_x = b_y$, tak pridáme diagonálnu hranu z vrcholu $(x-1, y-1)$ do vrcholu (x,y) . Na obrázku je zobrazený tento graf aj s vyznačenou cestou, ktorá má najmenšiu váhu (diagonálne hrany majú váhu 0, ostatné váhu 1).



Obr. 3: Graf pre postupnosti $A=abcbabba$ a $B=cbabac$ [2]

Problém minimálneho edit scriptu sa zredukoval na nájdenie cesty s najmenším počtom nediagonálnych hrán. Na obrázku č.3 je ich 5. Keďže dĺžky vstupných postupností A a B sú 7 a 6, tak maximálna cesta má 13 nediagonálnych hrán. Číže najdlhšia spoločná podpostupnosť v zobrazenom príklade má $(13 - 5) / 2$ prvkov a obsahuje prvky CABA.

D-cesta je cesta začínajúca vo vrchole (0,0) majúca práve D nediagonálnych hrán. 0-cesta je cesta zložená iba z diagonálnych hrán. D-cesta je potom zložená z (D-1)-cesty nasledovaná nediagonálnou hranou, po ktorej môžu nasledovať už iba diagonálne hrany (séria na seba nadväzujúcich diagonálnych hrán sa v literatúre obyčajne nazýva snake).

Nech k -diagonála sú všetky vrcholy (x,y) v grafe, pre ktoré platí: $x - y = k$. Potom platí, že každá D -cesta musí končiť na k -diagonále, kde $k \in \{-D, -D+2, \dots, D-2, D\}$. Dôkaz triviálny, indukciou cez definíciu D -cesty.

Nech najďalej siahajúca D -cesta na k -diagonále je taká D -cesta, že zo všetkých D -ciest končiacich na k -diagonále je jej koniec najviac vzdialený od počiatočného vrcholu $(0,0)$. Potom platí, že najďalej siahajúca 0 -cesta na k -diagonále končí v bode (x,x) , kde $a_x \neq b_x$, alebo $x = N$, alebo $x = M$. Ďalej platí, že najďalej siahajúca D -cesta na k -diagonále sa dá rozložiť na:

najďalej siahajúcu $(D-1)$ -cestu na $(k-1)$ -diagonále, horizontálnu hranu a najdlhšiu možnú sériu po sebe idúcich diagonálnych hrán (maximálny snake),
alebo
najďalej siahajúcu $(D-1)$ -cestu na $(k+1)$ -diagonále, vertikálnu hranu a najdlhšiu možnú sériu po sebe idúcich diagonálnych hrán (maximálny snake).

Dôkaz opäť jednoduchý vyplývajúci z definície D -cesty.

Na základe hore uvedeného tvrdenia môžeme konštruovať optimálnu D -cestu. Začínáme s $D=0$ a postupne zvyšujeme až do $n+m$, alebo kým koncový vrchol nejakej cesty nebude (n,m) . V pamäti si držíme koncové body (stačí nám iba hodnota x , lebo $y = x - k$) najdlhšie siahajúcich $(D-1)$ -ciest vo všetkých diagonálach, načo nám stačí pole o dĺžke $D+1$ (čiže celkovo $2*(n+m)+1$). V každom kroku pre D -cestu na k -diagonále vyberáme tú dlhšiu $(D-1)$ -cestu z dvoch možností: na $k-1$ a $k+1$ diagonále. Po vybratí sa už iba posunieme po diagonále až pokiaľ to viac nebude možné a poznačíme do poľa novú hodnotu pre diagonálu k .

Myersov algoritmus

```
V[1] := 0
for D := 0 to MAX do
  for k := -D to D step 2 do
    if k = -D or k = D and V[k-1] < V[k+1] then
      x := V[k+1]
    else
      x := V[k-1] + 1
    y := x - k
    while x < N and y < M and ax+1 = by+1 do
      x := x + 1
      y := y + 1
    V[k] := x
    if x >= n and y >= m then
      //dĺžka najkratšieho edit scriptu je D
      result := D
      exit
```

kde A, B sú vstupné postupnosti dĺžky n, m (reťazce)
 V je pole obsahujúce koncové body (stačí nám iba

hodnota x , lebo $y = x - k$) najdlhšie siahajúcich
 $(D-1)$ -ciest vo všetkých diagonálach
 MAX je hodnota obvykle omnoho menšia ako $n+m$
 a umožňuje ukončiť algoritmus a vyhlásiť že počet
 rozdielov je príliš veľký, ak je rovná $n+m$ tak dĺžku
 najkratšieho edit scriptu vždy nájdeme

Časová zložitosť tohto algoritmu je v najhoršom prípade $O((n+m)*D)$,
 resp. $O((n+m)*MAX)$, ak je $MAX < n+m$. V priemernom prípade je to $O(n+m+D^2)$,
 resp. $O(n+m+MAX^2)$, dôkaz v [2]. Algoritmus vyžaduje $O(n+m)$ priestor na alokáciu
 poľa V . Avšak tento algoritmus vracia iba dĺžku najkratšieho edit scriptu a teda ak
 chceme aj konkrétnu postupnosť operácií, tak si musíme pamätať aj priebežné V polia
 počas výpočtu, čo však spôsobí zhoršenie priestorovej zložitosti na kvadratickú. Neskôr
 bolo v [16] publikované mierne zlepšenie na zložitosť $O((n+m)*P)$, kde $P = D/2 - (n-m)/2$
 (nech bez straty na všeobecnosti je $n \geq m$), ktoré bolo dosiahnuté tesnejším ohraničením
 prehládavaných diagonál.

Existujú aj varianty tohto algoritmu využívajúce sufixové stromy s časom behu
 $O(n \lg n + D^2)$ v najhoršom prípade, avšak tieto sú v priemernom prípade horšie ako
 Myersov algoritmus.

Úprava Myersovho algoritmu na lineárny priestor

Zlepšenie vytvárania minimálneho edit scriptu na lineárny priestor je založené na
 podobnej idei ako pri Hirschbergovej úprave Wagner-Fisherovho algoritmu [13] na
 lineárny priestor. Použije sa metóda rozdeľuj a panuj. Nájdeme strednú diagonálnu časť
 optimálnej D -cesty (nazývaná aj ako middle snake), ktorá musí byť časťou celej
 optimálnej D -cesty od $(0,0)$ do (n,m) . Nech je to $(x,y) - (u,v)$, a získame dva rekurzívne
 podproblémy: nájsť optimálnu D -cestu od $(0,0)$ do (x,y) a nájsť optimálnu D -cestu od
 (n,m) do (u,v) a tieto cesty spojiť.

Algoritmus na nájdenie middle snake je založený na súčasnom spustení Myersovho
 algoritmu spredu (t.j. od vrcholu $(0,0)$) a zozadu (t.j. od vrcholu (n,m)). Keďže platí, že
 D -cesta z $(0,0)$ do (n,m) existuje iba vtedy a len vtedy, ak existuje
 (horná celá časť $D/2$)-cesta z $(0,0)$ do vrcholu (x,y) a (dolná celá časť $D/2$)-cesta z (u,v)
 do vrcholu (n,m) také, že platí: $u+v \geq \lceil D/2 \rceil \wedge x+y \leq n+m - \lfloor D/2 \rfloor$, a súčasne platí
 $x-y = u-v \wedge x \geq u$. Navyše obe $D/2$ -cesty sú obsiahnuté v rámci ciest z $(0,0)$ do (n,m) .
 Dôkaz v [2].

Algoritmus na nájdenie strednej diagonálnej časti
optimálnej D -cesty (find middle snake)

```

delta := n-m
for D := 0 to  $\lceil (m+n)/2 \rceil$  do
  for k := -D to D step 2 do

```

```

    najdi koniec najďalej siahajúcej D-cesty na diagonále k
    ak je delta nepárna a  $k \in \langle \text{delta} - (D-1), \text{delta} + (D+1) \rangle$  potom
        ak cesta sa prekrýva s najďalej siahajúcou reverznou
        (D-1)-cestou v diagonále k potom
            dĺžka najkratšieho edit scriptu je  $2 \cdot D - 1$ 
            middle snake je rovný poslednému snake-u na D-ceste
    for k := -D to D step 2 do
        najdi koniec najďalej siahajúcej reverznej D-cesty na
        diagonále k+delta
        ak je delta párna a súčasne  $(k+\text{delta}) \in \langle -D, D \rangle$  potom
            ak cesta sa prekrýva s najďalej siahajúcou (D-1)
            cestou v diagonále k potom
                dĺžka najkratšieho edit scriptu je  $2 \cdot D$ 
                middle snake je rovný poslednému snake-u na D-ceste

```

Algoritmus na výpočet najkratšieho edit scriptu

Nech je ES na začiatku prázdny zreťazený zoznam operácií.

```

CalculateSES(offA,n,offB,m)
    if n = 0 then
        if m ≠ 0 then
            ES.add(ES_INSERT,offB,m)
            D := m
        else
            D := 0
    else if m = 0 then
        ES.add(ES_DELETE,offA,n)
        D := n
    else
        najdi middle snake a dĺžku optimálnej D-cesty
        medzi A[offA..n] a B[offB..m]
        nech je to (x,y) - (u,v) a dĺžka je D
        if D > 1 then
            CalculateSES(offA,x,offB,y)
            CalculateSES(offA + u, n - u, offB + v, m - v)
        else
            if m > n then
                if x = u then
                    ES.add(ES_INSERT,off2 + (m - 1),1)
                else
                    ES.add(ES_INSERT,off2,1)
            else
                if x = u then
                    ES.add(ES_DELETE,off1 + (n - 1),1)
                else
                    ES.add(ES_DELETE,off1,1)

```

kde ES je hľadaný najkratší edit script
ES.add je funkcia na pridanie operácie do zoznamu ES,
prvky tohto zoznamu sú trojice (operácia,index,počet)

Napriek tomu, že algoritmus môže dosiahnuť až hĺbku rekurzie $\lg n$, tak celková zložitosť je stále $O((n+m)*D)$, podobne ako v predošlom algoritme, ale stačí iba $O(n+m)$ priestoru. Jedným z najznámejších programov využívajúcich tento algoritmus (označovaný aj ako Myersov $O(ND)$ algoritmus) je diff program v UNIXe.

Editovacia vzdialenosť a stromy

V tejto časti sú popísané algoritmy na zisťovanie vzdialenosti medzi stromami. Za stromy budú v ďalšom texte považované zakorenené, usporiadané stromy, kde každý vrchol má svoj identifikátor (label). Zakorenené znamená, že jeden z vrcholov je označený ako koreň. Usporiadané znamená, že synovia každého vrcholu majú určené poradie.

Pre neusporiadané stromy je tento problém ešte oveľa zložitejší. V [3] bolo ukázané, že problém editovacej vzdialenosti je NP-úplný (a teda polynomiálny algoritmus môže existovať jedine v prípade, že $P = NP$) už pre binárne stromy a abecedu labelov o veľkosti 2. Je dokonca MAX-SNP ťažký [20], čiže k-aproximatívny algoritmus bežiaci v polynomiálnom čase pre ľubovoľné k sa dá nájsť iba ak by sa $P = NP$.

Selkow

Jedným z najstarších algoritmov je modifikácia klasického Wagner-Fisherovho algoritmu na zisťovanie editovacej vzdialenosti pre reťazce, na stromy. Túto úpravu spravil už Selkow v [11]. Tento algoritmus uvažuje s operáciami zmeniť vrchol (relabel/update), vymazanie vrcholu (delete), vloženie vrcholu (insert). Avšak vymazávať a vkladať sa dajú iba listy.

Selkowov algoritmus

```
edit(A,B)
  D[0,0] := cupdate(A,B)
  for I := 1 to degree(A) do
    D[I,0] := D[i-1,0] + cdelete(Ai)
  for j := 1 to degree(B) do
    D[0,j] := D[0,j-1] + cinsert(Bj)
  for i := 1 to degree(A) do
    for j := 1 to degree(B) do
      m1 := D[i-1,j-1] + edit(Ai,Bj)
      m2 := D[i-1,j] + cdelete(Ai)
      m3 := D[i,j-1] + cinsert(Bj)
      D[i,j] := min(m1,m2,m3)
```

`result := D[degree(A), degree(B)]`

kde A, B sú vstupné stromy

A_i je i -ty podstrom vrcholu A

`degree` je počet synov daného vrcholu

D je matica obsahujúca editovaciu vzdialenosť pre
všetkých synov vrcholov A a B v danom vnorení rekurzie
`update`, `insert`, `delete` sú cenové funkcie pre dané
elementárne operácie (kde `delete(A_i)` znamená cena za
vymazanie podstromu A_i , podobne `insert(A_i)` znamená
cena za vloženie podstromu A_i)

Selkowsky algoritmus je rekurzívny, začína s koreňmi vstupných stromov A a B . Pre každú dvojicu podstromov X, Y vytvorí maticu ich synov o veľkosti $(\text{degree}(X)+1) \times (\text{degree}(Y)+1)$, ktorá sa postupne zľava-doprava, zhora-nadol naplní, kde políčko i, j obsahuje minimum z cien:

cena edit scriptu X_1, X_2, \dots, X_i a Y_1, Y_2, \dots, Y_{j-1} + cena vloženia Y_j

cena edit scriptu X_1, X_2, \dots, X_{i-1} a Y_1, Y_2, \dots, Y_j + cena vloženia X_i

cena edit scriptu X_1, X_2, \dots, X_{i-1} a Y_1, Y_2, \dots, Y_{j-1} + cena edit scriptu X_i, Y_j (rekurzívne volanie)

V políčku $D[\text{degree}(X), \text{degree}(Y)]$ je na konci cena minimálneho edit scriptu medzi podstromami X, Y . Po naplnení takejto matice pre korene stromov A a B máme hľadanú cenu minimálneho edit scriptu v $D[\text{počet synov koreňa } A, \text{počet synov koreňa } B]$.

Keďže matica D je lokálna štruktúra, po skončení algoritmu už nedokážeme zistiť poradie operácií v minimálnom edit scripture. A teda si musíme spolu s minimálnymi cenami uchovávať v poli D aj minimálny edit script.

Časová zložitosť tohto algoritmu je $O(nm)$, kde n je počet vrcholov v strome A , m je počet vrcholov v strome B . Priestorová zložitosť je rovnaká ako časová, teda $O(nm)$. Avšak priestorová zložitosť sa dá pomerne jednoducho zlepšiť na $O(m+n)$.

Nevýhodou je prílišné obmedzenie operácií vkladania a vymazávania iba na vrcholy bez potomkov (t.j. listy). A teda cena vymazania jedného vrcholu bude potom cena vymazania celého podstromu, ktorému bol daný vrchol koreňom plus cena vloženia všetkých vrcholov okrem daného vrcholu.

Tai

Algoritmus, ktorý opísal Tai v [14] tiež vychádza z algoritmu pre editovaciu vzdialenosť reťazcov, avšak na rozdiel od Selkowskyho rekurzívneho algoritmu je aj algoritmom dynamického programovania. Ako editovacie operácie povoľuje zmenu vrcholu (`relabel/update`), vymazanie vrcholu a vloženie vrcholu. Ak má vymazávaný vrchol nejakých potomkov, tak títo sú zavesení pod jeho otca. Podobne pri vkladaní vrcholu sú relevantní synovia otca vkladaneho vrcholu zavesení pod nový vložený vrchol.

Na lepšie zachytenie toho ako postupnosť editovacích operácií transformuje jeden strom na druhý, bolo použité mapovanie. Toto mapovanie je relácia medzi vrcholmi oboch stromov, ktorá zachováva poradie potomkov a aj predchodcov. Mapovanie M zo stromu A do stromu B (nech počet vrcholov $A = n$ a počet vrcholov $B = m$), je množina usporiadaných dvojíc (i,j) , kde $i \in \{1, \dots, n\}$ a $j \in \{1, \dots, m\}$, takých, že nech pre ľubovoľné dvojice $(i,j), (u,v) \in M$ platí:

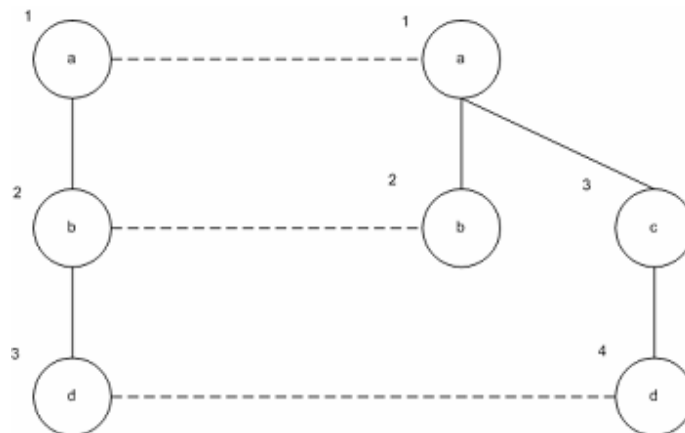
- $i=u \Leftrightarrow j=v$
- a_i je naľavo od $a_u \Leftrightarrow b_j$ je naľavo od b_v
- a_i je predchodca $a_u \Leftrightarrow b_j$ je predchodca b_v

kde a_i sú vrcholy zo stromu A ($a_i \in V(A)$) a b_i sú vrcholy zo stromu B ($b_i \in V(B)$).

Mapovanie zo stromu A do B popisuje editovacie operácie potrebné na transformovanie stromu A na strom B . Vrchol a_i , ktorý sa nevyskytuje v žiadnej dvojici $(i,j) \in M$ bude vymazaný zo stromu A . Vrchol b_j , ktorý sa nevyskytuje v žiadnej dvojici $(i,j) \in M$ bude vložený do stromu A . Dvojica $(i,j) \in M$ znamená zmenu vrcholu a_i na vrchol b_j (relabel/update).

Cena mapovania je suma cien operácií, ktoré popisuje dané mapovanie. Táto cena vytvára mieru a nie je ťažké ukázať, že minimálne mapovanie popisuje operácie v minimálnom edit scripte.

Podobný formalizmus je použitý aj v ďalších algoritmoch. Oproti porovnávaniu reťazcov si pri porovnávaní stromov treba uchovávať aj informácie o štruktúre daných stromov. Keď sme pri porovnávaní reťazcov vedeli, že ak sa $a_{N+1}..a_{N+k} = b_{M+1}..b_{M+k}$, tak potom sa vzdialenosť medzi $a_1..a_N$ a $b_1..b_M$ rovná vzdialenosti medzi $a_1..a_{N+k}$ a $b_1..b_{M+k}$. Pre stromy toto neplatí, ako je vidieť na nasledujúcom obrázku (strom A je vľavo, strom B vpravo). Ak stromy A a B mali mapovanie $M = (1,1), (2,2)$, tak po pridaní identického vrcholu do oboch stromov nebude mapovanie $M' = (1,1), (2,2), (3,4)$, lebo toto je neplatné podľa definície mapovania (nesplňa bod 3, pre dvojicu $(b,b), (d,d)$, lebo b je predchodcom d v A , ale nie v B), ale bude to $M'' = (1,1), (2,3), (3,4)$.



Obr. 4: Obrázok zobrazujúci neplatné mapovanie M'

Taiov algoritmus najprv očísľuje všetky vrcholy pomocou preorderu, a potom pomocou dynamického programovania si udržiava prehľad o neplatných podstromoch a mapovaní v 6-rozmernej matici. Nech A_1 a A_2 sú vstupné stromy, n_1, n_2 sú ich veľkosti a d_1, d_2 sú hĺbky, potom Taiov algoritmus pracuje v časovej zložitosti $O(n_1 n_2 d_1^2 d_2^2)$. Priestorová zložitosť je takisto $O(n_1 n_2 d_1^2 d_2^2)$.

Podobný algoritmus ako Taiov, avšak založený na postorderovom očísľovaní vrcholov a pracujúci v lepšej časovej a podstatne lepšej priestorovej zložitosti bol prezentovaný Zhangom a Shashom v [3].

Zhang-Shasha

Tento algoritmus využíva rovnaké elementárne editovacie operácie ako Taiov algoritmus, teda vloženie vrcholu, vymazanie vrcholu a zmenu vrcholu. Aj vďaka tomu, že je tento algoritmus dynamického programovania založený na postorderovom očísľovaní vrcholov, je minimálne mapovanie všetkých potomkov daného vrcholu vypočítané skôr, ako sa začne počítať tento vrchol. Takže minimálne mapovanie daného vrcholu sa môže rovno vypočítať bez backtracku alebo rekurzie.

Zhang a Shasha v [3] zaviedli nový pojem, takzvaný keyroot (kľúčový koreň). Keyroot je definovaný ako koreň stromu alebo vrchol, ktorý má ľavého súrodencu. Nech $K(A)$ je množina všetkých keyrootov v strome A .

$$K(A) = \{v \mid v \in V(A) \wedge \exists w \in V(A) \text{ také, že } w \text{ je ľavý súrodenec } v\} \cup \text{koreň}(A).$$

Nech A je vstupný strom s n vrcholmi, B je vstupný strom s m vrcholmi. Pole všetkých postorderových čísiel keyrootov pre stromy A, B (označme ich KA, KB) sa vypočíta ako prvý krok súčasne s postorderovým číslovaním vrcholov v jednom prechode cez stromy A a B (poradie v akom boli keyrooty získané je dôležité, čiže keyrooty s menším postorderovým číslom sú skôr v poli). Ďalšie pomocné polia LA, LB pre stromy A, B udávajú najľavejších potomkov pre všetky vrcholy (t.j. bude to list s najmenším postorderovým číslom v podstrome daného vrcholu).

$$\begin{aligned} LA(i) = \{ & \text{postorderové číslo vrcholu } v \mid v \in V(A_i) \wedge \\ & \wedge \neg \exists w \in V(A) \text{ také, že } w \text{ je ľavý súrodenec } v \wedge \\ & \wedge \neg \exists w \in V(A) \text{ také, že } v \text{ je otcom } w \} \end{aligned}$$

LA, LB sa vypočítajú pri tom istom postorderovom prechode stromom ako KA, KB . Čiže čas behu tohto prvého kroku je zhodný z časom behu jedného postorderového traverzovania stromu A a B , čiže rovný $O(n+m)$.

Nech D je matica najmenších editovacích vzdialeností medzi A_i a B_j (v $D[m,n]$ bude celková minimálna vzdialenosť medzi A a B). Nech FD je lokálna matica najmenších vzdialeností medzi lesmi. Pod lesom sa tu rozumie usporiadaný les, ktorý obsahuje

podstromy, ktorých koreňmi sú synovia nejakého vrcholu. Podles je chápaný ako prefix daného lesa. Táto definícia slúži na ľahšie označenie posunu v rekurentnom vzťahu, teda že máme vyriešené vzdialenosti pre všetky vrcholy naľavo od terajšieho.

Základom tohto algoritmu dynamického programovania sú nasledujúce dva rekurentné vzťahy (cinsert,cdelete,cupdate sú cenové funkcie ohodnocujúce operáciu vloženia, vymazania, zmeny vrcholu v strome):

$$\begin{aligned}
 FD[i,j] &:= \min(\\
 &\quad FD[i-1,j] + cdelete(a_i), \\
 &\quad FD[i,j-1] + cinsert(b_j), \\
 &\quad FD[LA(i)-1, LB(j)-1] + D[i,j] \\
 &\quad) \\
 D[i,j] &:= \min(\\
 &\quad FD[i-1,j] + cdelete(a_i), \\
 &\quad FD[i,j-1] + cinsert(b_j), \\
 &\quad FD[i-1,j-1] + cupdate(a_i,b_j) \\
 &\quad)
 \end{aligned}$$

Algoritmus postupne prechádza cez všetky keyrooty a pomocou týchto dvoch rekurentných vzťahov vyplní maticu FD (táto matica sa pre každú dvojicu keyrootov naplní postupne vzdialenosťami medzi ich podlesmi) a maticu D (táto sa naplní vzdialenosťami medzi podstromami A a B, pričom na pozícii m,n sa po skončení algoritmu bude nachádzať vzdialenosť medzi samotnými stromami A a B). Samozrejme vzdialenosť to bude iba ak nastavíme vhodne cenové funkcie, inak je to cena edit scriptu. Nech $u \in KA$ a $v \in KB$, potom matica FD pre tieto dva keyrooty bude mať veľkosť $|A_u| * |B_v|$. Matica D má veľkosť $|A| * |B|$. Celý algoritmus je uvedený nižšie:

Zhang-Shashov algoritmus

vypočítajú sa polia LA, LB, KA, KB pre vstupné stromy A a B

```

for k1 := 1 to |KA| do
  x := KA(k1)
  for k2 := 1 to |KB| do
    y := KB(k2)
    FD[LA(x)-1, LB(y)-1] := 0
    for i := LA(x) to x do
      FD[i, LA(x)-1] := FD[i-1, LA(x)-1] + cdelete(a_i)
    for j := LB(y) to y do
      FD[LA(x)-1, j] := FD[LA(x)-1, j-1] + cinsert(b_j)
    for i := LA(x) to x do
      for j := LB(y) to y do
        m1 := FD[i-1, j] + cdelete(a_i)

```



```

m2 := FD[i, j-1] + cinsert(bj)
if LA(i) = LA(x) and LB(j) = LB(y) then
    m3 := FD[i-1, j-1] + cupdate(ai, bj)
    D[i, j] := min(m1, m2, m3)
    FD[i, j] := D[i, j]
else
    m3 := FD[LA(i)-1, LB(j)-1] + D[i, j]
    FD[i, j] := min(m1, m2, m3)

```

kde KA, KB sú polia indexov keyrootov
 LA, LB sú polia indexov najľavejších potomkov
 FD je matica vzdialeností medzi podlesmi vrcholov a_x
 a b_y (resp. minimálnej ceny edit scriptu)
 D je matica vzdialeností medzi podstromami stromov A
 a B (resp. minimálnej ceny edit scriptu)
 cinsert, cdelete, cupdate sú cenové funkcie

Nech A, B sú vstupné stromy s počtom vrcholov rovným m, n a hĺbkou stromu rovnou d₁, d₂. Časová zložitosť Zhang-Shashovho algoritmu je O(nmd₁d₂). Priestorová zložitosť je len O(nm). Najhorší prípad nastáva, ak strom vyzerá ako zoznam. Vtedy je hĺbka rovná veľkosti stromu (počtu vrcholov) a teda zložitosť je rovná O(n²m²). Avšak v priemernom prípade je časová zložitosť rovná O(n^{1.5}m^{1.5}) [3].

Vyššie je uvedený iba algoritmus na vypočítanie minimálnej vzdialenosti (resp. ceny minimálnej editovacej postupnosti). Samotnú postupnosť elementárnych operácií sa však dá získať, ak štruktúrne obohatíme bunky matice D o pomocné informácie ohľadom operácií v dočasne najlepšej postupnosti. Dá sa to spraviť bez zmeny celkovej časovej a priestorovej zložitosti tohto algoritmu.

Klein v [10] dokázal zlepšiť zložitosť v najhoršom prípade pomocou rozdelenia vrcholov na ľahké a ťažké (ten vrchol spomedzi súrodencov, ktorý má najväčšiu veľkosť svojho podstromu) a dekompozície stromu na disjunktné ťažké cesty. Kleinov algoritmus má čas behu v najhoršom prípade O(n²mlg n) pri nezmenenej priestorovej zložitosti O(nm). Avšak je to zlepšenie iba pre určité typy stromov oproti Zhang-Shashovmu algoritmu, pre iné (dokonca aj v priemernom prípade) dáva lepšie výsledky práve ten.

EZS

Pridaním ďalších operácií do Zhang-Shashovho algoritmu sa zaoberali v [9]. Do množiny elementárnych operácií boli postupne pridávané operácie vloženie stromu (insertTree), vymazanie stromu (deleteTree), vymenenie súrodencov (swap) a nakoniec vymenenie súrodencov so zmenou v ich podstromoch (swap+edit). Prezentovali algoritmus, ktorý po pridaní prvých troch menovaných operácií si stále zachoval pôvodnú zložitosť, a to ako časovú, tak priestorovú. Tento algoritmus je známy aj ako EZS (Extended Zhang-Shasha algorithm):

Rozšírený Zhang-Shashov algoritmus (EVS)

vypočítajú sa polia LA, LB, KA, KB pre vstupné stromy A a B

```
for k1 := 1 to |KA| do
  x := KA(k1)
  for k2 := 1 to |KB| do
    y := KB(k2)
    FD[LA(x)-1, LA(y)-1] := 0
    for i := LA(x) to x do
      FD[i, LA(x)-1] := FD[i-1, LA(x)-1] + cdelete(ai)
    for j := LB(y) to y do
      FD[LA(y)-1, j] := FD[LA(y)-1, j-1] + cinsert(bj)
    for i := LA(x) to x do
      for j := LB(y) to y do
        m1 := FD[i-1, j] + cdelete(ai)
        m2 := FD[i, j-1] + cinsert(bj)
        m4 := FD[LA(i)-1, j] + cdeleteTree(Ai)
        m5 := FD[i, LB(j)-1] + cinsertTree(Bj)
        if LA(i) = LA(x) and LB(j) = LB(y) then
          m3 := FD[i-1, j-1] + cupdate(ai, bj)
          D[i, j] := min(m1, m2, m3, m4, m5)
          FD[i, j] := D[i, j]
        else if (i in KA) and (j in KB)
          m6 := FD[LA(LA(i)-1), LB(LB(j)-1)] + cswap(Ai, Bj)
          FD[i, j] := min(m1, m2, m3, m4, m5, m6)
        else
          m3 := FD[LA(i)-1, LB(j)-1] + D[i, j]
          FD[i, j] := min(m1, m2, m3, m4, m5)
```

kde KA, KB sú polia indexov keyrootov
LA, LB sú polia indexov najľavejších potomkov
A_i je podstrom stromu A, ktorého koreňom je vrchol
s postorderovým číslom i
FD je matica vzdialeností medzi podlesmi vrcholov a_x
a b_y (resp. minimálnej ceny edit scriptu)
D je matica vzdialeností medzi podstromami stromov A
a B (resp. minimálnej ceny edit scriptu)
cinsert, cdelete, cupdate, cinsertTree, cdeleteTree, cswap
sú cenové funkcie

Po umožnení editovania vymieňaných podstromov už bolo nevyhnutné použiť rekúziu na podstromoch, čím sa samozrejme aj výrazne zhoršila časová zložitosť na $O(n^2 m^2 d_1 d_2)$.

Iný algoritmus dynamického programovania s výrazne väčšou sadou elementárnych editovacích operácií je navrhnutý a analyzovaný v [17], kde sa zaoberali operáciami zmena vrcholu, vloženie vrcholu, vymazanie vrcholu a horizontálne zlúčenie vrcholov,

horizontálne rozdelenie vrcholov, vertikálne zlúčenie vrcholov, vertikálne rozdelenie vrcholov. Avšak časová zložitosť sa vyšplhala až na $O(n^7)$ pre vertikálnu editovaciu vzdialenosť, resp. $O(n^{2k+2})$ pre k-cestnú horizontálnu editovaciu vzdialenosť (kde $k \geq 2$).

FMES

Fast Match Edit Script (FMES) algoritmus bol navrhnutý v [4] ako komplementárny algoritmus k EZS, hlavne na porovnávanie veľkých stromových štruktúr (na malé stromy odporúčajú autori použiť EZS). Používa štyri elementárne editovacie operácie: vloženie nového listu, vymazanie listu, zmena vrcholu a presun podstromu (myslené nielen z jedného rodiča na iného rodiča, ale aj ako zmena poradia súrodencov). Ako strom sa tu takisto chápe zakorenený, usporiadaný strom, kde každý vrchol má svoj label a listy majú aj hodnotu (nejaký text prislúchajúci danému vrcholu). Nech $p(x)$ je rodič vrcholu x , $l(x)$ je label vrcholu x , $v(x)$ je hodnota vrcholu x .

Algoritmus rozdeľuje problém na dva podproblémy:

- nájdenie „dobrého“ mapovania (good matching problem)
- vytvorenie minimálneho edit scriptu (minimum conforming edit script problem)

Good Matching Problem

Na začiatok sa vstupné stromy pretraverzujú postorderom a vytvoria sa zreťazené zoznamy (reťazce) pre každý label vo vstupných stromoch, v ktorých platí, že ak je x pred y v zozname, tak je x skôr navštívený vrchol ako y pri traverzovaní stromu. V rámci daného labelu sa na prislúchajúci zoznam (reťazec) vrcholov použije algoritmus na hľadanie najdlhšej spoločnej podpostupnosti (s jedinou zmenou, že sa hodnoty neporovnávajú priamo, ale funkcia Equal určuje ich zhodu, resp. nezhodu). Takto sa prejde cez všetky vrcholy zdola-nahor a vytvorí sa čiastočné mapovanie M . Funkcia Equal je definovaná nasledovne:

pre listy:

Equal(x, y)

= true, ak $l(x) = l(y) \wedge 2 * \text{Editdist}(v(x), v(y)) / (\text{Length}(v(x)) + \text{Length}(v(y))) \leq F$

= false, inak

kde F je parameter FMES algoritmu $0 \leq F \leq 1$

pre vnútorné vrcholy:

Equal(x, y)

= true, ak $l(x) = l(y) \wedge \text{commonleaves}(x, y) / \max(\text{leaves}(x), \text{leaves}(y)) > T$

= false, inak

kde T je parameter FMES algoritmu $0,5 \leq T \leq 1$

Editdist je funkcia vracajúca editovaciú vzdialenosť medzi dvomi reťazcami (používa napr. Myersov $O(ND)$ algoritmus).

Length je funkcia vracajúca dĺžku reťazca

Leafs je funkcia, ktorá vráti počet listov medzi potomkami daného vrchola

CommonLeafs je funkcia ktorá vráti počet vzájomne namapovaných listov medzi potomkami vrcholu x a y , teda:

$$\text{CommonLeafs}(x,y) = |\{(u,v) \mid x \text{ je predchodca } u \wedge y \text{ je predchodca } v \wedge (u,v) \in M\}|$$

Parameter F určuje ako rozdielne môžu byť hodnoty v listoch aby sme ich ešte určili za vhodnú dvojicu do mapovania. Extrémny prípad je ak $F = 2$, potom idú do mapovania dvojice listov, ktorým stačí ak majú rovnaký label, naopak ak je $F = 0$, tak sa v mapovaní nachádzajú iba listy, ktoré majú totožný label aj hodnotu (text).

Parameter T určuje koľko percent vzájomne namapovaných listov musia mať podstromy vnútorných vrcholov, aby sme ich určili ako vhodnú dvojicu do mapovania. Ak je $T = 1$, tak podstromy musia byť kompletne vzájomne namapované (teda v podstate zhodné), aby mohli dané vnútorné vrcholy byť v mapovaní ak majú zhodný label.

Fast Match algoritmus

```
M.clear
for each label l do
  C1 := chainA(l)
  C2 := chainB(l)
  L := LCS(C1,C2,Equal)
  for each (x,y) in L do
    M.add(x,y)
  for each (nenamapované x) in C1 do
    if  $\exists y \in C2: \text{Equal}(x,y)$  then
      M.add(x,y)
      označí x,y ako namapované
```

kde M je mapovanie, $M.\text{clear}$ je funkcia na vyprázdnenie mapovania M , $M.\text{add}$ je funkcia na pridanie dvojice do mapovania

for each label l začína najprv s listovými labelmi a potom pokračuje s labelmi vnútorných vrcholov $\text{chain}_A(l)$ je zretazený zoznam vrcholov v A majúcich label rovný l

Equal je funkcia, ktorá vráti true alebo false na základe toho či sú dané dva vrcholy dostatočne podobné (pre listy a pre vnútorné vrcholy je definovaná rôzne)

LCS je funkcia vracajúca najväčšiu spoločnú

podpostupnosť vstupných reťazcov, kde o zhodnosti dvoch prvkov rozhoduje funkcia Equal

Čas behu tejto časti algoritmu je $(ne + e^2) * c + 2nle$, kde n je celkový počet listov, c je priemerný čas potrebný na LCS, l je počet rôznych labelov vo vstupných stromoch, e je váhovaná editovacia vzdialenosť definovaná v [4] nasledovne: nech $E = e_1 \dots e_n$ je minimálny edit script transformujúci strom A na strom B (e_i sú elementárne operácie), potom váhovaná editovacia vzdialenosť $e = \sum_{1 \leq i \leq n} w_i$, kde w_i je váha elementárnej operácie e_i . Ak je e_i vloženie, alebo vymazanie vrcholu, tak $w_i = 1$, ak je e_i presun podstromu, tak $w_i = |x|$ (označuje počet listov v presúvanom strome s koreňom x), inak $w_i = 0$. Pomer e/d sa dá ohraničiť $\lg n$ (kde d je editovacia vzdialenosť vstupných stromov), ale väčšinou to býva pomerne nízka konštanta, čiže očakávaný čas behu je skutočne lineárny, ak sa stromy príliš nelišia.

Minimal Conforming Edit Script Problem

Vstupom pre druhú časť algoritmu je nejaké čiastočné mapovanie M (teda mapovanie, ktoré spĺňa aspoň prvú podmienku z definície mapovania – pre ľubovoľné dvojice $(i,j), (u,v) \in M$ platí: $i=u \Leftrightarrow j=v$), v ktorom sú namapované aspoň korene. Ak nastane prípad, že $M = \emptyset$ po skončení algoritmu FastMatch na nájdenie dobrého mapovania, tak jednoducho pridáme dvojicu koreňov do mapovania ešte pred začatím druhej časti algoritmu. Tento algoritmus je koncipovaný tak, že pre ľubovoľné čiastočné mapovanie nám vráti platný edit script, ktorý však nemusí byť minimálny. Algoritmus postupne rozširuje mapovanie M a transformuje strom A na strom B , až pokiaľ neobsahuje všetky vrcholy a strom A nie je izomorfný so stromom B . Algoritmus sa skladá z nasledovných piatich hlavných fáz:

- **Update**

V tejto fáze algoritmus hľadá dvojice nachádzajúce sa v mapovaní M , ktorých hodnoty sú rôzne, čiže také $(x,y) \in M$, že $v(x) \neq v(y)$. Pre každú takúto nájdenú dvojicu sa zmení hodnota $v(x)$ vo vrchole x na $v(y)$ a do edit scriptu sa pridá operácia zmeny vrcholu (update).

- **Align**

Nech $(x,y) \in M$, ak medzi ich synmi nie je zachované usporiadanie, t.j. že existujú vrcholy $u,v \in A$ (kde u,v sú synovia vrcholu x) také, že pre im prislúchajúce vrcholy $w,z \in B$ (kde w,z sú synovia vrcholu y) v mapovaní M (u je partnerom w a v je partnerom z v mapovaní M) neplatí, že ak u je naľavo od v , tak potom aj w je naľavo od z , potom treba potomkov v strome A preusporiadať tak, aby táto vlastnosť mapovania začala platiť. Usporiadať tieto vrcholy sa dá obvykle viacerými spôsobmi. Keďže hľadáme minimálny edit script, tak aj množstvo operácií potrebných na preusporiadanie by malo byť minimálne. Toto sa dosahne využitím LCS algoritmu na potomkoch oboch vrcholov x,y (použije sa

napríklad Myersov $O(ND)$ algoritmus, ktorý je popísaný vyššie v tejto kapitole). Potomkovia vrcholu x v strome A , ktorí nie sú v najdlhšej spoločnej podpostupnosti sa presunú na požadované miesto (podľa ich partnera v strome B) a do edit scriptu sa pridá operácia presunu vrcholu (move).

- **Insert**

V tejto fáze sa hľadajú vrcholy, ktoré nie sú ešte v mapovaní, ale ich rodičia sú, čiže nech $v \in B$, $y = p(v)$ a $\neg \exists u \in M: (u,v) \in M$. Nech $x \in A$ je partnerom y (t.j. $(x,y) \in M$). Algoritmus vytvorí nový vrchol (bude to list) v strome A , nech je to w a zavesí ho pod vrchol x tak, aby sa neporušilo usporiadanie z fázy Align. Do edit scriptu sa pridá operácia vloženie vrcholu w (insert).

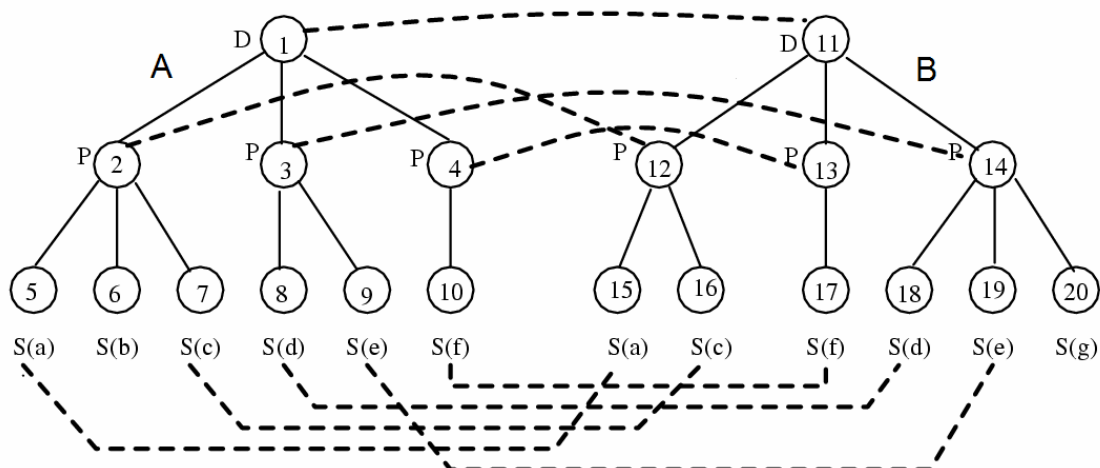
- **Move**

Táto fáza hľadá také dvojice, ktorých rodičia nie sú v mapovaní, t.j. také $(u,v) \in M$, že $(x,y) \notin M$, kde $x = p(u)$, $y = p(v)$. Nech z je partnerom y v mapovaní, t.j. $(z,y) \in M$ (táto dvojica po predošlej fáze musí existovať), potom sa vrchol x presunie pod vrchol z a do edit scriptu sa pridá operácia presunu vrcholu (move).

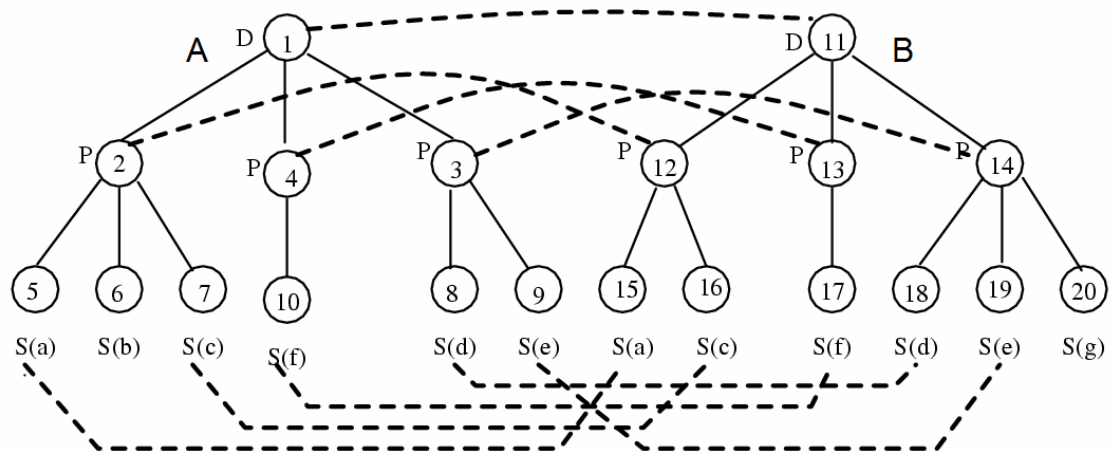
- **Delete**

Všetky vrcholy x zo stromu A , ktoré nie sú v mapovaní, t.j. $\neg \exists y \in M: (x,y) \in M$, vymaže zo stromu A a do edit scriptu pridá operácie vymazania vrcholu x pre všetky takéto vrcholy.

Ukážka transformácie stromu A vo fáze Align je zobrazená na nasledujúcich dvoch obrázkoch.



Obr. 5: Mapovanie po prvej časti algoritmu [4]



Obr. 6: Mapovanie po fáze Align [4]

Tieto fázy nejdú v takomto striktnom poradí a neaplikujú sa iba raz a na všetky vrcholy, hoci by to bol korektný algoritmus. Kvôli väčšej rýchlosti sú prvé štyri fázy zlúčené do jedného prehľadávania do šírky stromu B. Na úplný záver sa aplikuje fáza Delete pri postorderovom traverzovaní stromu A. Po ukončení behu je strom A transformovaný na strom izomorfný so stromom B, v M' je úplné mapovanie a v ES je kompletný edit script.

EditScript algoritmus

nech ES je zatiaľ prázdny edit script
 nech M je „dobré“ mapovanie z prvej časti algoritmu

```

M' := M
for (x je nasled. vrchol stromu B pri prehľad. do šírky) do
    y := p(x)
    z := partner(y)
    if partner(x) = nil then //čiže x nie je root
        k := FindPos(x);
        w := newnode(a, v(x))
        insertnode(w, z, k)
        M'.add(w, x)
        ES.add(ES_INSERT, w, z, k)
    else if p(x) <> nil do //čiže x nie je root
        w := partner(x)
        v := p(w)
        if v(w) <> v(x) then
            updatenode(w, v(x))
            ES.add(ES_UPDATE, w, v(x))
        if not ((y, v) in M') then
            z := partner(y)
    
```

```

        k := FindPos(x)
        movenode(w, z, k)
        ES.add(ES_MOVE, w, z, k)
    AlignChildren(w, x)

for (y je nasled. vrchol stromu A pri postorder trav.) do
    if partner(y) = nil then
        removenode(y)
        ES.add(ES_DELETE, y)

kde A, B sú vstupné stromy
ES je hľadaný edit script
ES.add je funkcia na pridanie operácie do zoznamu ES,
    prvky tohto zoznamu sú n-tice (operácia, ...)
insertnode, removenode, movenode, updatenode sú
    operácie na strome A, ktoré vykonávajú danú
    elementárnu transformáciu na strome A

```

Časová zložitosť tejto druhej časti algoritmu je $O(nk)$, kde k je počet zle usporiadaných vrcholov z fázy Align. Podstatnú časť tvorí LCS algoritmus vo fáze Align. Keďže k je určite menej ako vzdialenosť medzi stromami, resp. váhovaná vzdialenosť medzi stromami, tak je časová zložitosť tejto druhej časti aj v $O(ne)$.

Zložitosť prvej časti algoritmu je vyššia ako zložitosť druhej časti a teda celková časová zložitosť sa po sčítaní oboch rovná $O(ne + e^2)$, kde n je počet listov a e je váhovaná editovacia vzdialenosť. Priestorová zložitosť je $O(n)$, čo je výhodné pre porovnávanie stromov s veľkým množstvom vrcholov.

Nevýhodou tohto algoritmu je fakt, že podmienkou na to aby generoval optimálne mapovanie sú nasledovné dva predpoklady o vstupných stromoch:

- medzi všetkými labelmi v strome existuje také usporiadanie, že ak $l_1 < l_2$ potom v strome neexistuje vrchol s labelom l_1 , ktorý by bol potomkom vrcholu s labelom l_2
- ku každému listu v strome existuje maximálne jeden list v druhom strome, ktorý mu je podobný (t.j. iba jeden potenciálny kandidát na mapovanie)

mmdiff

Algoritmus je založený na idei transformácie problému hľadania minimálneho edit scriptu medzi stromami na problém nájdenia najkratšej cesty v špeciálnom editovacom grafe [5]. Elementárnymi editovacími operáciami sú v tomto algoritme operácie vloženie vrcholu, vymazanie vrcholu a zmenenie vrcholu. Najprv sa pretransformujú stromy do postupností vrcholov v preorderovom usporiadaní, presnejšie do postupností dvojíc (label daného vrchola, hĺbka daného vrchola). V [5] označované ako ld-pár (label, depth-pair).

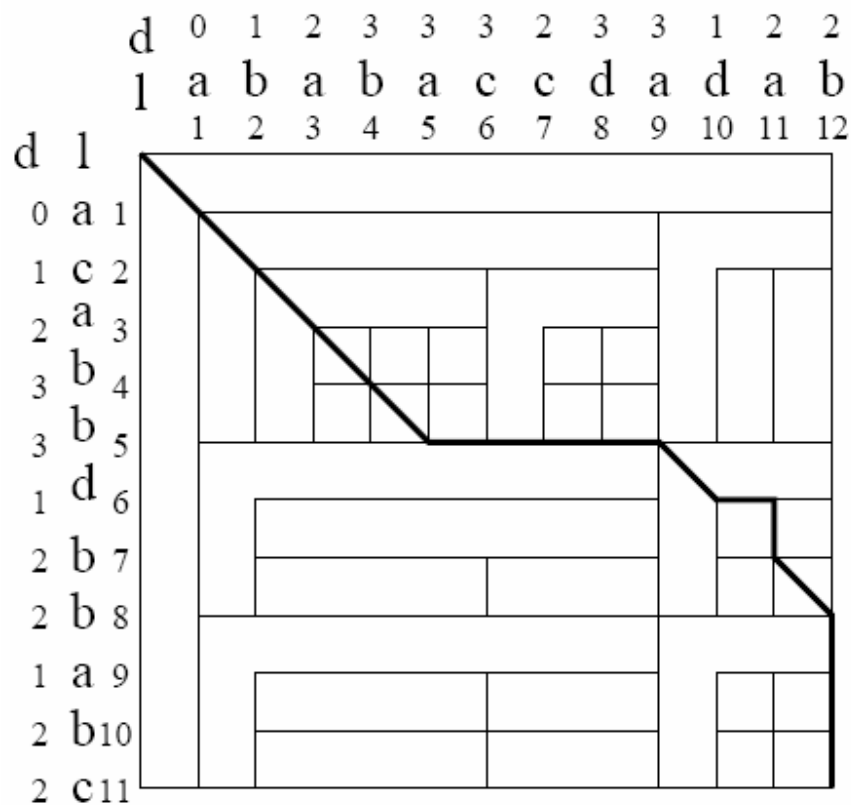
Podobne ako pri algoritmoch na porovnávanie reťazcov sa vytvorí graf v tvare mriežky, avšak nie všetky hrany sú povolené. Nech A a B sú postupnosti dvojíc vytvorené z pôvodných stromov, m a n sú ich dĺžky.

Vytvoríme graf, ktorý bude vlastne mriežka pozostávajúca z vrcholov (x,y) , kde $x \in \{0, \dots, m\}$ a $y \in \{0, \dots, n\}$. Tieto vrcholy budú pospájané horizontálnymi, vertikálnymi a diagonálnymi hranami nasledovne:

pre každé $x \in \{0, \dots, m-1\}$ a $y \in \{0, \dots, n-1\}$, také že, $A[x+1].d = B[y+1].d$ je v editovacom grafe diagonálna hrana $((x,y), (x+1,y+1))$.

pre každé $x \in \{0, \dots, m-1\}$ a $y \in \{0, \dots, n\}$, také že, $y = n \vee A[x+1].d \geq B[y+1].d$ je v editovacom grafe horizontálna hrana $((x,y), (x+1,y))$.

pre každé $x \in \{0, \dots, m\}$ a $y \in \{0, \dots, n-1\}$, také že, $x = m \vee A[x+1].d \leq B[y+1].d$ je v editovacom grafe vertikálna hrana $((x,y), (x,y+1))$.



Obr. 7: Editovací graf pre vstupné stromy $A=ababaccdadab$ a $B=acabbdbbbabc$ [5], pričom hĺbka jednotlivých vrcholov je 012333233122 a 01233122122 (diagonálne hrany sú kvôli väčšej prehľadnosti vynechané). Hrubou čiarou je označená najkratšia cesta.

Na predošlom obrázku je zobrazený tento graf aj s vyznačenou cestou, ktorá má najmenšiu váhu (diagonálne hrany, kde sú korešpondujúce vrcholy zhodné majú váhu 0, ostatné váhu 1).

Takýmto obmedzením hrán v editovacom grafe je zaručené, že ak edit script obsahuje napr. vymazanie vrcholu x vnútri stromu, tak sa v ďalších krokoch vymažú aj všetci jeho potomkovia, pretože majú väčšiu hĺbku ako vrchol x a teda k nim vedie iba horizontálna hrana, ktorá znamená vymazanie vrcholu v konečnom dôsledku.

Po tejto redukcii je samotný algoritmus už iba variantom algoritmu hľadania najkratšej cesty v grafe.

mmdiff algoritmus

```

D[0,0] := 0
for i := 1 to m do
  D[I,0] := D[i-1,0] + cdelete(A[i])
for j := 1 to n do
  D[0,j] := D[0,j-1] + cinsert(B[j])
for i := 1 to n do
  for j := 1 to n do
    if A[i].d = B[j].d then
      m1 := D[i-1,j-1] + cupdate(A[i],B[j])
    else
      m1 := INF
    if j = N or B[j+1].d <= A[i].d then
      m2 := D[i-1,j] + cdelete(A[i])
    else
      m2 := INF
    if I = m or (A[i+1].d <= B[j].d) then
      m3 := D[I,j-1] + cinsert(B[j])
    else
      m3 := INF
    D[i,j] := min(m1,m2,m3)

```

kde A,B sú vstupné polia reprezentujúce stromy (obsahujúce ld-páry)
 D je matica obsahujúca vzdialenosti medzi prefixami
 poľa A a B
 INF je konštanta reprezentujúca nekonečno
 cdelete,cinsert,cupdate sú funkcie vracajúce cenu jednotlivých operácií

Keďže po skončení algoritmu máme naplnenú maticu D, samotný edit script vieme získať cestou najmenšieho odporu v čase $O(m+n)$ jednoduchým prejdením matice D z (m,n) do (0,0), podobne ako tomu bolo v prípade Wagner-Fisherovho algoritmu pre reťazce.

Časová aj priestorová zložitosť tohto algoritmu je $O(mn)$. Jeho zaujímavosť je najmä v elegantnej a jednoduchej transformácii na problém minimálnej cesty v grafe. Navyše tento algoritmus tvorí základ algoritmu pre porovnávanie veľkých súborov, ktoré sa nezmestia do pamäte [5].

XYDiff

Algoritmus XYDiff popísaný v [18] bol vytvorený na štruktúrne porovnávanie XML dokumentov. Tento algoritmus používa podobne ako FMES štyri elementárne editovacie operácie: vloženie vrcholu, vymazanie vrcholu, zmena vrcholu a presun podstromu. Takisto ako v FMES algoritme sa najprv vytvára čiastočné mapovanie, ktoré sa ďalej rozširuje na úplné popri generovaní edit scriptu. Na začiatku sa jedným postorderovým prechodom stromu vypočíta hašovací hodnota pre každý vrchol x v strome A . Rovnako pre každý vrchol v strome B .

Táto hodnota reprezentuje celý podstrom a ak je zhodná s hodnotou iného vrcholu v druhom strome, tak sa predpokladá, že tieto vrcholy môžu byť v mapovaní. Súčasne s hašovacou hodnotou sa vypočíta aj veľkosť podstromu, ktorého koreňom je x . Toto si označme ako váhu vrcholu x . Z vrcholov sa vytvorí prioritná fronta na základe veľkosti podstromu. Na začiatku je v prioritnej fronte iba koreň stromu B .

V cykle sa vyberá vrchol s najväčšou veľkosťou v ňom zakoreneného podstromu (teda s najväčšou váhou) z prioritnej fronty a tento sa porovnáva s hašovacími hodnotami podstromov v strome A . Nech je to vrchol x . K danej hašovacej hodnote získame množinu vrcholov v strome A s rovnakou hašovacou hodnotou. Táto množina môže byť aj prázdna. Ak je naopak v množine viac vrcholov, tak vyberáme ten, ktorého hašovacia hodnota v rodičovi je zhodná s hašovacou hodnotou v rodičovi vrcholu x (pretože zrejme to bude lokálne najlepšia voľba). Ak toto nespĺňa ani jeden vrchol, tak sa posúvame na rodičov ich rodičov atď. (množstvo predchodcov, ktoré takto zvažujeme je závislé od váhy vrcholu x a v menšej miere aj od pozície zvažovaných vrcholov v rámci jeho súrodencov). V každom prípade sa algoritmus vyhýba pri väčšom množstve zhodných vrcholov overovaniu každého z nich.

Ak nájdeme zhodu, tak sa tieto podstromy pridávajú do mapovania a navyše sa začnú pridávať v cykle aj ich predkovia, pokiaľ majú zhodný label (toto správanie nie je vždy optimálne, hlavne ak by zhoda medzi malými podstromami ovplyvnila namapovanie veľkých stromov a aj preto je parametrizované cez konštantu, ktorá určuje pri akých pomeroch veľkostí sa ešte pridávajú aj predkovia daného vrcholu do mapovania).

Ak nenájdeme zhodu, tak sa do fronty pridajú potomkovia vrcholu x .

Po skončení tohto cyklu je mapovanie takmer hotové. Ešte sa ale mohlo stať, že niektoré možnosti boli vynechané a preto nasleduje optimalizačný krok, kde sa pokúšame v okolí namapovaných vrcholov získať ďalšie zhody spomedzi nenamapovaných vrcholov do

mapovania. V XYDiffé sa v tomto kroku vykoná jeden zhora-nadol a jeden zdola-nahor prechod stromu B.

Teraz na základe hotového mapovania sa už iba vygeneruje edit script. Ak nájdeme nejaké nenamapované vrcholy v strome A, tak sa označia ako vymazané a poznačí sa aký to má vplyv na pozície súrodencov. Podobne pre nenamapované vrcholy v strome B, tieto sa označia ako vložené. Vrcholy, ktoré sú síce v mapovaní, ale hodnota v partnerských vrcholoch je rôzna, budú označené ako zmenené. Vrcholy, ktoré sú namapované, ale ich rodič nie je namapovaný, označíme ako presunuté. Tie vrcholy, ktorých rodič je takisto namapovaný môžu spôsobiť zmenu usporiadania medzi súrodencami. To znamená, že medzi operácie sa musí navyše pridať niekoľko presunov. Ak chceme minimálny počet presunov, tak musíme, podobne ako to je v algoritme FMES v procedúre AlignChildren, použiť LCS algoritmus. Tento však môže byť pri veľkých hodnotách príliš časovo náročný, a preto sa vstupný reťazec vrcholov rozdelí na niekoľko podreťazcov a na každom z nich sa použije LCS algoritmus (čím si pokazíme optimálny počet presunov, ale na druhej strane nestratíme toľko času).

Zložitosť tohto algoritmu je $O(n \cdot \lg n)$, kde n je počet vrcholov v A a B spolu. Priestorová zložitosť je lineárna – $O(n)$.

Nevýhodou je to, že tento algoritmus nedokáže žiadnym spôsobom garantovať optimálne výsledky (ani dostatočne blízke k optimálnym). Avšak pri takejto nízkej časovej náročnosti sa dá používať na úvodné prefiltrovanie vstupných stromov a okrem iného aj na zistenie či nie sú veľmi podobné, alebo veľmi rozdielne.

MH-diff

V [21] Chawathe prezentoval heuristický algoritmus na získavanie edit scriptu dokonca pre neusporiadané, zakorenené, označované stromy. Navyše zavádza operácie kópia podstromu a zlúčenie podstromov ako elementárne operácie. Tento algoritmus je založený na reprezentácii edit scriptu medzi dvomi stromami ako hranové pokrytie bipartitného grafu. Na začiatku sa vytvorí úplný bipartitný graf medzi vrcholmi z jedného stromu v jednej partícii a vrcholmi druhého stromu v druhej partícii (plus špeciálne vrcholy: zdroj (source) v jednej a ústie (sink) v druhej). Pomocou množstva všeobecných i heuristických pravidiel sa tento graf redukuje na čo najmenší a zostávajúce hrany sa približne ohodnotia a potom sa použijú štandardné techniky na vyriešenie problému cenovo minimálneho hranového pokrytia. Toto ide v čase $O(ne)$, kde n je počet vrcholov a e počet hrán. Číže časová zložitosť algoritmu MH-diff je v najhoršom prípade, keď nám zostanú všetky hrany z kompletného bipartitného grafu, $O(n \cdot (n \cdot n)) = O(n^3)$.

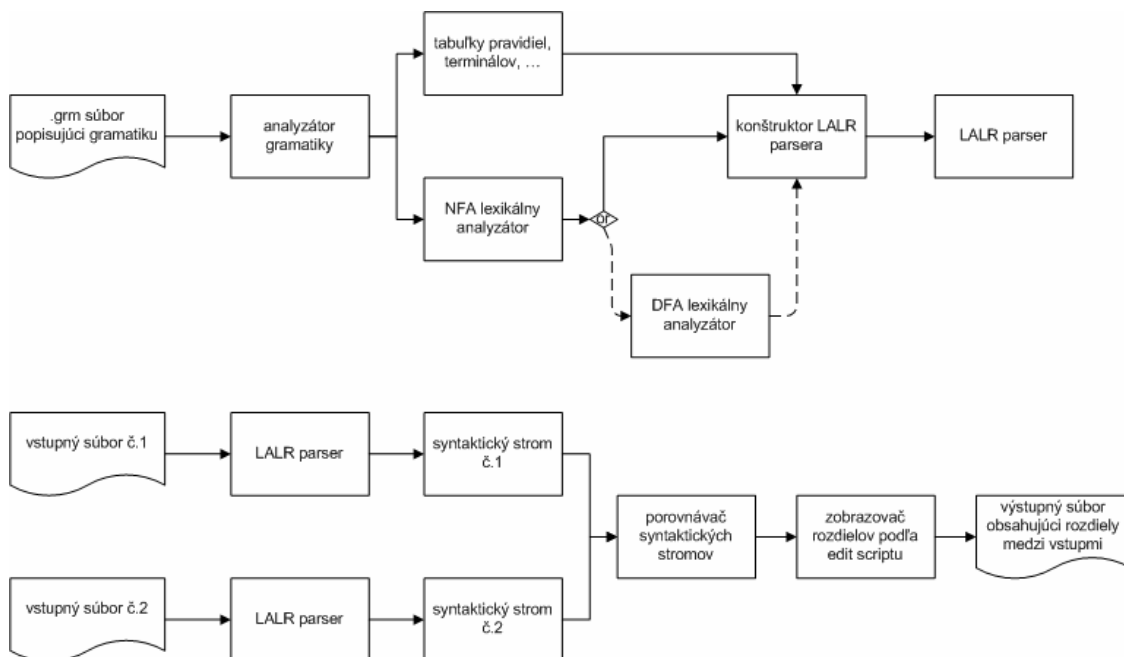
X-diff

Tento algoritmus, popísaný v [6] bol vytvorený tiež pre zakorenené, označované (každý vrchol má label) a neusporiadané stromy. Používa elementárne operácie vloženie listu, vymazanie listu a zmenenie listu. Na začiatku sa jedným prechodom cez vstupné stromy vypočíta hašovacia hodnota v každom vrchole, ktorá reprezentuje daný podstrom a súčasne sa počíta aj druhá hašovacia hodnota reprezentujúca cestu od koreňa vstupného stromu k danému vrcholu (označovaná ako signature). Tiež sa dá rozdeliť na dve časti, hľadanie mapovania a vytváranie edit scriptu. Pri hľadaní mapovania sa v hlavnom cykle, pre dané dva vstupné vrcholy najprv podľa hašovacej hodnoty ich potomkov odfiltrujú zhodné podstromy (tieto majú zhodnú hašovaciu hodnotu a teda sa s veľmi veľkou pravdepodobnosťou majú nachádzať aj v mapovaní), potom sa postupuje cez ich zvyšné podstromy do šírky, ale smerom od listov ku koreňu. Ak sa signature dvoch vrcholov nezhoduje, tak ich netreba zvažovať, ani ich podstromy na pridanie do mapovania. Ak sa ich signature zhoduje, tak sa počíta minimálna vzdialenosť medzi nimi.

Podľa [6] je časová zložitosť tohto algoritmu v najhoršom prípade rovná $O(n*m*\max(\maxdeg(A),\maxdeg(B))*\lg(\max(\maxdeg(A),\maxdeg(B))))$, kde n,m sú veľkosti vstupných stromov, $\maxdeg(A)$ je maximálny stupeň vrcholu v strome A. Tento prípad je však veľmi nepravdepodobný, v priemernom prípade sa čas behu pohybuje na úrovni XYDiff algoritmu.

4. Implementácia

Táto kapitola je venovaná popisu riešenia a samotnej implementácii, algoritmom použitým vo výslednom programe. Na úvod prezentujem diagram popisujúci v kocke funkcionality tohto programu. Jednotlivé časti budú rozobraté neskôr v tejto kapitole.



Obr. 8: Diagram stručne znázorňujúci fungovanie programu

Ako metóda syntaktickej analýzy je použité LALR parsovanie. Aj keď sme pri tvorbe gramatiky trochu obmedzovaní a nemôžeme použiť ľubovoľnú CFG (bezkontextovú gramatiku), väčšinou sa dá gramatika trochu prispôbiť, aby sme odstránili konflikty a mohli ju tak bez problémov používať pri LALR analýze. Na druhej strane získavame veľmi veľa na časovej zložitosti, ak pri všeobecnom CFG parsovaní sme nútení použiť CYK algoritmus so zložitosťou $O(n^3)$, alebo Earleyho algoritmus podobne $O(n^3)$, ale pri jednoznačných gramatikách $O(n^2)$. Pri LR syntaktickej analýze (parsovaní) je časová zložitosť $O(n)$ vzhľadom na dĺžku vstupného slova. To je obrovský rozdiel, keďže pomer lexikálnych jednotiek ku počtu riadkov je pri zdrojových kódach približne 10:1, samozrejme veľmi záleží na gramatike. Čo sa týka konštrukcie tohto LALR parseru je použitá mierna modifikácia postupu z [1] na priame vytváranie LALR parseru zo vstupných pravidiel. Zmenená bola mierne konštrukcia jadra množiny položiek, kde jadro množiny položiek obsahuje aj ϵ – položky (t.j. do jadra sa pridá $A \rightarrow \epsilon$, ak sa $A \rightarrow \epsilon$ nachádza v uzávere množiny položiek), kvôli ušetreniu konštrukcie ďalších množín, ktoré označujú neterminály, ktoré sú nulovateľné a ďalších dvoch množín, ktoré by sa museli inak predpočítavať pri lookahead propagácii. Nasleduje stručný opis konštrukcie parsovacích tabuliek LALR parsera.

Na začiatku si vytvoríme množinu First (množina obsahujúca pre každý neterminál tie terminály, ktoré sa z neho odvodlia ako prvé, v zmysle najľavejšie). Potom vytvoríme prvé jadro množiny položiek (itemset) pre položku $S' \rightarrow .S$, EOF (kde S' je nový štartovací (počiatočný) neterminál, S je pôvodný štartovací neterminál a EOF je token označujúci koniec súboru). Nový počiatočný neterminál sa vždy pridáva do pôvodnej gramatiky, hlavne kvôli ukončeniu a akceptovaniu až na konci súboru. V hlavnom cykle prechádzame cez všetky množiny položiek (množiny položiek sú uložené iba ako jadrá v množine množín položiek) a robíme nasledovné:

1. vytvoríme substitučnú tabuľku pre položky v danej množine položiek (itemset)
2. pridáme virtuálne lookaheady do jadier podľa substitučnej tabuľky
3. vytvoríme uzáver jadrových položiek (virtuálne lookaheady nám určia ako sa propagujú spontánne lookaheady)
4. vygenerujeme možné „skoky“ z uzáveru (LR0 jump table), v rámci tohto kroku aj pridávame ϵ – položky do jadra
5. v LR0 jump table sa teraz bude nachádzať pre každý symbol množina položiek, z ktorých je na tento symbol možný posun (shift) z uzáveru množiny položiek (tieto budú tvoriť jadro novej množiny), ktorú momentálne prechádzame v hlavnom cykle; pre každý symbol v tejto tabuľke sa vykonajú v cykle nasledujúce štyri kroky
 - 5.1. pre symbol v LR0 jump table sa z jemu prislúchajúcej množiny vytvára nová jadrová množina položiek, sú dve možnosti: množina s rovnakým jadrom (ϵ – pravidlá vynímajúc) sa už nachádza v množine množín položiek (v tomto prípade sa ďalej používa táto množina), alebo sa tam nenachádza (a teda bude do nej pridaná)
 - 5.2. pridanie do tabuľky posunov (gotoTable)
 - 5.3. spracovanie virtuálnych lookaheadov a vyriešenie propagácie lookaheadu k nim
 - 5.4. odstránenie virtuálnych lookaheadov, ktoré boli používané iba pri vytváraní „spojov“ pre propagáciu lookaheadu
6. pridanie položiek (z množiny položiek, ktorú momentálne prechádzame v hlavnom cykle), ktoré majú bodku na konci do množiny redukovateľných položiek

Po skončení hlavného cyklu sa ešte spustí propagácia lookaheadu podľa vybudovaných „spojov“. Na záver sa z množiny redukovateľných položiek vytvorí tabuľka redukcií (reduceTable).

Vstupom do syntaktickej analýzy sú tokeny z lexikálnej analýzy. Táto je obvykle založená na deterministickom konečnom automate, ale v programe je ponechaná aj možnosť používať nedeterministický konečný automat, hlavne kvôli regulárnym výrazom, ktoré vyžadujú exponenciálny počet stavov v deterministickom konečnom automate. Regulárne výrazy tvaru $(a \mid b)^k$, kde k je nejaká konštanta, napríklad nech $k = 10$, definícia tokenu test:

$$\text{test} = (a|b)^k a (a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)$$

Rozpoznávanie tohto tokenu deterministickým konečným automatom si vyžaduje stav pre každú 10 znakovú kombináciu symbolov a,b (týchto kombinácií je 2^k), čiže v tomto prípade bude mať deterministický konečný automat rozpoznávajúci tento token aspoň 1024 stavov. Naproti tomu nedeterministický konečný automat si v tomto prípade vystačí s približne 20 stavmi. Takže pri takýchto regulárnych výrazoch sa dá využiť lexikálny analyzátor postavený na nedeterministickom konečnom automate, ktorý je síce pomalší pri spracovávaní vstupu, ale stačí mu exponenciálne menej stavov. Konštrukcia NFA z regulárnych výrazov využíva klasický Thompsonov postup konštrukcie [1], kde pre každú elementárnu operáciu, resp. symbol z regulárneho výrazu je definovaná operácia na automatoch, resp. automat akceptujúci daný symbol. Z takýchto automatov sa postupne poskladá výsledný automat pre celý výraz. Konverzia na DFA využíva takisto klasický postup [1], kde si pre každú množinu stavov v ktorej sa môže nachádzať NFA automat vytvoríme stav v DFA automate, v ktorom si pamätáme túto množinu. Keďže počet rôznych množín stavov v NFA môže byť exponenciálny k počtu stavov (presnejšie 2^n , kde n je počet stavov), tak logicky deterministický konečný automat potenciálne môže mať 2^n stavov.

Čo sa týka porovnávania stromov, najlepším dosiaľ známym výsledkom pre všeobecné porovnávanie stromov je Zhang-Shashov algoritmus (ak nerátame mierne zlepšenia najhoršieho prípadu za cenu zhoršenia priemerného času [10][19]), bolo by ideálne použiť rozšírený Zhang-Shashov algoritmus [9]. Avšak problém nastáva s časovou i priestorovou zložitou, lebo pomer vrcholov v strome a počtu riadkov kódu je pri bežnom programovacom jazyku, akým je napríklad Java, až 30:1, čiže pri bežnom zdrojovom kóde s 1000 riadkami bude mať strom veľkosť približne 30000 vrcholov, čo si vyžiada niekoľko gigabytov priestoru. Najmä preto je pri porovnávaní stromov takmer nevyhnutné použiť algoritmus s lineárnou zložitou. Avšak žiadny taký neexistuje a preto sa treba zmieriť s hľadaním približnej vzdialenosti a nie úplne optimálnych riešení, alebo s kvadratickou časovou zložitou.

Ak nemáme žiadne ďalšie informácie okrem gramatiky a vstupných súborov, tak nám nezostáva veľa možností ako postupovať. Môžeme buď nejakým spôsobom zredukovať vstupné syntaktické stromy, napr. odstránením izomorfných podstromov, alebo nejako určiť zaujímavé uzly v stromoch a s ostatnými sa síce bude počítat, ale budú tvoriť iba akúsi výplň. Ďalšou z možností je spraviť iba približné mapovanie a toto postupne zlepšovať, podobne ako je to vo FMES algoritme [4].

V každom prípade je treba už pri vytváraní parsovacích stromov myslieť na to, že nejdeme daný súbor kompilovať, ale iba porovnávať výsledný syntaktický strom s ďalším podobným syntaktickým stromom. Preto je výhodné aplikovať operácie na zníženie počtu vrcholov v tomto strome. Napríklad ak máme v gramatike definované ľavo rekurzívne pravidlo $\langle \text{list} \rangle ::= \langle \text{list} \rangle, \langle \text{list_item} \rangle$, tak sa nám oplatí mať namiesto 30 úrovní vrcholu $\langle \text{list} \rangle$ do hĺbky mať 30 vrcholov $\langle \text{list_item} \rangle$ pod jediným vrcholom: $\langle \text{list} \rangle$. Rovnako pre pravo rekurzívne pravidlá. Podobne ak máme pravidlo, ktoré je iba substitúciou neterminálu za iný neterminál, tak tento medzičlánok do výsledného syntaktického stromu nemusíme dávať.

Samozrejme môžeme vstupné syntaktické stromy transformovať aj agresívnejšie, napríklad vymazaním uzlov, ktoré generujú iba ϵ , transformovaním stromu pri použití ľavo lineárnych ($A \Rightarrow Bw$, kde $A, B \in N$ a $w \in T^*$) a pravo lineárnych pravidiel podobne ako ľavo a pravo rekurzívnych pravidiel, t.j. zavesením pod spoločného predka. Prípadne ak máme definovaný zoznam neterminálov, ktoré sú irelevantné pre porovnávanie, potom môžeme im prislúchajúce vrcholy zo stromu vymazať. Celkovo je ale treba dávať pozor na prípadné vymazávanie vrcholov, ktoré nám držia štrukturálnu informáciu o strome, aby sme neskončili po úpravách stromu s tým, že výsledný strom bude mať hĺbku 2, čiže koreň a pod ním všetky listy.

V programe bola implementovaná modifikácia algoritmu FMES, ktorý je popísaný v predchádzajúcej kapitole.

Je síce lineárny vzhľadom na dĺžku vstupu, ale kvadratický vzhľadom na počet odlišností. Tento výsledok korešponduje s výsledkami pre zisťovanie editovacej vzdialenosti reťazcov, ako sú popísané napr. v [2]. Čo sa týka časovej zložitosti, tak asymptoticky lepší výsledok sa nedá očakávať. Priestorová zložitosť je taktiež optimálna - $O(n)$.

Jeho nevýhodou je fakt, že podmienkou pre to aby generoval optimálne mapovanie, sú nasledovné dva predpoklady:

- medzi všetkými labelmi v strome existuje usporiadanie také že, ak $l_1 < l_2$ potom v strome neexistuje vrchol s labelom l_1 , ktorý by bol potomkom vrcholu s labelom l_2
- ku každému listu v strome existuje maximálne jeden list v druhom strome, ktorý mu je veľmi podobný (t.j. iba jeden potenciálny kandidát na mapovanie)

Problémom je hlavne to, že tieto predpoklady sú takmer vždy porušené, najmä pri porovnávaní zdrojových súborov (obidva body), ale aj dokumentov (najmä druhý bod). Samozrejme záleží od gramatiky. V prípade, že ľubovoľný z týchto predpokladov neplatí, algoritmus síce generuje platný edit script (t.j. taký, že po jeho aplikácii sa transformuje strom A na strom B), avšak jeho dĺžka/cena nemusí byť, a obvykle ani nie je optimálna, práve naopak. A len niekedy sa dá takýto neoptimálny edit script zoptimálniť pomocou nejakého jednoduchého postprocessingu.

Porušenie prvého predpokladu môže spôsobiť zacyklenie pri vytváraní minimálneho edit scriptu pri presúvaní podstromu, lebo pri určitom mapovaní hrozí situácia, že operácia presunu podstromu zakoreneného vo vrchole w sa presunie pod vrchol v , kde vrchol v je potomkom vrcholu w . Riešením je pri operácii posunutia vždy overiť, či presunom podstromu nespôsobíme cyklus (čo nám spôsobí iba veľmi mierny nárast časovej zložitosti, asymptotická zložitosť zostane rovnaká).

Porušenie druhého predpokladu spôsobuje neoptimalitu generovaného edit scriptu. Mierne sa to dá zlepšiť tak, že po nájdení dobrého mapovania (FastMatch) sa začne prehľadávať strom B zhora-nadol. Nech x je vrchol v B, ak $(v, x) \in M$, tak potom, ak syn y z B vrcholu x nie je namapovaný na syna w z B, tak sa skúsi nájsť nejaký vhodný syn

vrcholu x , označme ho z , taký, že je vhodný do mapovania. Ak taký existuje, tak sa adekvátne zmení mapovanie M (t.j. vymaže sa pôvodná dvojica a pridá sa nová, resp. ak z už bol namapovaný s nejakým vrcholom, tak táto dvojica sa tiež zruší a pridá sa ďalšia obsahujúca w a pôvodného partnera z).

Ak chceme získať navyše operácie vloženie podstromu, resp. vymazanie podstromu, tak stačí jeden krát postorderom prejsť strom B , resp. A a hľadať najväčšie podstromy nedotknuté mapovaním (zdola-nahor). Toto traverzovanie sa uskutoční tesne po ukončení procedúry na nájdenie „dobrého“ mapovania.

Algoritmus FMES patrí do skupiny algoritmov, ktoré sa snažia greedy technikami zdola nahor nájsť vzdialenosť medzi stromami. Po implementácii tohto algoritmu ma výsledky pri porovnávaní zdrojových kódov presvedčili, že je potrebné hľadať nejaký iný algoritmus.

Nakoniec spôsob aký používa XYDiff v prvej časti, ktorý vlastne vychádza z pôvodného Selkowovho algoritmu na porovnávanie stromov, t.j. zhora-nadol skúšať vytvárať mapovanie, vyzerá byť celkom vhodný pre porovnávanie štruktúrovaných súborov. Väčšinou ak budujeme mapovanie zdola-nahor, tak máme často krát obrovské množstvo možných namapovaní, z ktorých je len jedno správne a preto vynaložíme mnoho úsilia na jeho nájdenie, obvykle nemôžeme vybrať to ideálne, lebo by to trvalo príliš dlho.

Kvôli hore uvedeným príčinám som sa rozhodol implementovať porovnávanie stromov zhora-nadol pomocou algoritmu, ktorý vychádza zo Selkowovho algoritmu na porovnávanie stromov, ale pracuje trochu inak. Ohodnocuje každú zhodu medzi vrcholmi a hľadá maximálne skóre cez všetkých potomkov vstupných vrcholov.

Na začiatku algoritmu sa postorderom pretraverzujú obidva stromy a vo vrchoch sa spočíta hašovacia funkcia, ktorá reprezentuje podstrom zakorenený v danom vrchole. Popri tom si počítame aj skóre celého podstromu zakoreneného v danom vrchole.

Skóre v liste je rovné hodnote ekvivalencie medzi listami tak, ako je to definované pre daný label v súbore, ktorý popisuje gramatiku. V prípade, že pre daný label nie je definovaná žiadna hodnota pre ekvivalenciu, tak sa použije globálna hodnota definovaná v súbore, ktorý popisuje gramatiku. Ak ani tá nie je definovaná, potom sa použije štandardná hodnota, definovaná v konfiguračnom súbore.

Skóre vo vnútornom vrchole stromu je rovné súčtu skóre svojich synov plus hodnota, ktorá je definovaná pre daný label v súbore, ktorý popisuje gramatiku. Ak pre tento label nie je definovaná žiadna hodnota, tak sa použije globálna hodnota definovaná v tomto súbore, ak ani táto nie je definovaná, tak sa použije štandardná hodnota, definovaná v konfiguračnom súbore.

Po vypočítaní týchto hodnôt pre všetky vrcholy nasleduje hlavná časť algoritmu, vypísaná nižšie, ktorá nám vráti mapovanie medzi vrcholmi oboch stromov. A síce mapovanie spĺňajúce klasické tri podmienky (jednoznačnosť, zachovávanie vzťahu

predchodca-nasledovník a zachovávanie vzťahu vľavo-vpravo), ktoré boli definované pri prehľade algoritmov v kapitole 3.

TreeScore algoritmus

```
TreeScore(p,r: treenode; var L: list)
  if p.hash = r.hash then
    result := p.subtreescore
    if subtreesareequivalent(p,r,L) then
      if L = nil then
        L := newlist
      exit
    else
      freelist(L)
  if getequivalencescore(p,r,score) then
    result := score
  else if getsimilarityscore(p,r,score) then
    result := score
  else
    //vrcholy nie sú zhodné, ani podobné, ukončíme
    // prehľadávanie v tejto vetve a vyhlásime, že
    // podstromy sú úplne odlišné - skóre 0
    result := 0
    L := newlist
    exit
cc1 := p.childcount
cc2 := r.childcount
if cc1 * cc2 > ARRAY_SIZE_LIMIT then
  //namiesto alokácie príliš veľkých polí vyhlásime, že
  // sa podstromy nezhodujú a pokračujeme ďalej
  result := 0;
  L := newlist;
  exit;

nech A je pole synov vrcholu p o veľkosti cc1
nech B je pole synov vrcholu r o veľkosti cc2
nech D je pole o veľkosti cc1 * cc2 obsahujúce skóre pre
  všetky dvojice vrcholov z polí A a B
nech MP je pole o veľkosti cc1 * cc2 obsahujúce mapovanie
  pre danú dvojicu vrcholov

for i := 0 to cc1 do
  D[i,0] := 0
for j := 0 to cc2 do
  D[0,j] := 0
for i := 1 to cc1 do
  for j := 1 to cc2 do
```

```

m2 := D[i-1,j]
m3 := D[i,j-1]
m1 := D[i-1,j-1] + TreeScore(A[i],B[j],L)
if m1 > m2 then
    if m1 > m3 then
        MP[i,j] := L
        D[i,j] := m1
    else
        freelist(L)
        MP[i,j] := nil
        D[i,j] := m3
    else
        if m2 > m3 then
            freelist(L)
            MP[i,j] := nil
            D[i,j] := m2
        else
            freelist(L)
            MP[i,j] := nil
            D[i,j] := m3
result := result + D[cc1,cc2]
L := newlist
i := cc1
j := cc2
while (i >= 1) and (j >= 1) do
    if D[i-1,j] = D[i,j] then
        i := i - 1
    else if D[i,j-1] = D[i,j] then
        j := j - 1;
    else
        prependtolist(L,MP[i,j])
        i := i - 1;
        j := j - 1;
prependtolist(L,r);
prependtolist(L,p);
uvoľnenie polí A,B,D,MP z pamäte

```

kde L je zreťazený zoznam vrcholov, newlist je funkcia, ktorá vytvorí nový zoznam v pamäti, freelist je procedúra odstraňujúca daný zoznam z pamäte, prependtolist je procedúra, ktorá pridá pred začiatok zoznamu daný prvok, alebo zoznam

A,B sú pomocné polia obsahujúce synov vrcholov p a r

D je matica o veľkosti $cc1 * cc2$ obsahujúca skóre pre všetky dvojice vrcholov z polí A a B

MP je matica o veľkosti $cc1 * cc2$ obsahujúca mapovanie pre danú dvojicu vrcholov

p,r sú vstupné vrcholy, kde p.hash je hašovacia hodnota pre podstrom vo vrchole p, a p.score je skóre tohto podstromu, p.childcount je počet synov vrchola p
 getequivalencescore a getsimilarityscore sú funkcie, vracajúce, či sú vrcholy zhodné, resp. podobné a ak sú, tak aj ich skóre, podľa vstupnej gramatiky, resp. nastavení v programe
 subtreesareequivalent je funkcia vracajúca, či sú skutočne podstromy vstupných vrcholov zhodné a navyše vytvára pri ich prechode zretazený zoznam vrcholov

Po skončení algoritmu máme spočítané skóre pre obidva stromy, a v zozname L celé mapovanie vrcholov jedného stromu na druhý strom. Navyše tieto dvojice sú usporiadané podľa pozície v pôvodnom strome v preorderovom usporiadaní, čo sa dá využiť pri vypisovaní a zobrazovaní výsledkov.

V programe sú implementované dve verzie hore uvedeného algoritmu. V jednej sa pri hašovaní uvažuje iba o labeloch v daných vrcholoch, čiže napr. priradenie $a := b + c$ má rovnakú hašovaciu hodnotu ako $d := e + f$. V druhej verzii majú rozdielnú hašovaciu hodnotu, lebo sa berie do úvahy aj konkrétna hodnota v listoch. Pri počítaní skóre pri tejto druhej verzii sa počíta s druhou hodnotou, ktorú je možné definovať ako hodnotu skóre pre identitu pre daný token (tiež v súbore, kde je definovaná vstupná gramatika, ak to tam nie je definované, tak sa použijú štandardné hodnoty). Využitie sa dá predstaviť v prípade, ak sa vrátíme k príkladu, že máme rekurzívne pravidlo $\langle \text{list} \rangle ::= \langle \text{list} \rangle, \langle \text{list_item} \rangle$, čiže nejaký zoznam oddelený čiarkami, napr. ako argumenty funkcie. V prípade, že budú položky tohto zoznamu vymenené, t.j. v jednom súbore bude a,b,c a v druhom zoznam b,c,a. Potom môže zavážiť práve hodnota za identitu a namiesto namapovania čiarok a zmeny všetkých troch identifikátorov sa namapuje b,c a vymaže sa: 'a,' z prvého zoznamu a pridá sa: ',a' z druhého zoznamu.

Ak nie sú dané dva vrcholy zhodné, tak funkcia getsimilarityscore nám vráti (podľa množín labelov, ktoré sú si podobné – môžu byť definované v súbore, kde je definovaná vstupná gramatika), či sú dané dva vrcholy podobné, a teda hoci sa ich label nezhoduje, tak nekončíme prehľadávanie tejto vetvy a porovnávame ďalej. Táto funkcia je myslená na prípady, v ktorých sa blokové štruktúry síce nezhodujú, ale vo vnútri nich je rovnaká (podobná) štruktúra. Napríklad, keď v jednom súbore je použitý for cyklus a v druhom while cyklus a vnútrojšok oboch blokov (cyklov) je rovnaký.

Časová a priestorová zložitosť tohto algoritmu je v najhoršom prípade kvadratická, ale priestorová zložitosť sa dá zlepšiť na lineárnu pri miernom zhoršení časovej zložitosti. V programe som sa po porovnaní výsledkov rozhodol toto zlepšenie nepoužiť. Navyše priestorová zložitosť nie je v bežnom prípade až taká zlá (kvadratická). Množstvo priestoru, ktoré pri výpočte potrebujeme (ak nerátame priestor potrebný na uloženie zoznamov namapovaných vrcholov) je možné ohraničiť zhora výrazom $c_1 * \maxdeg(A) * \maxdeg(B) * \min(\text{depth}(A), \text{depth}(B))$, pre nejaké kladné c_1 , kde \maxdeg je maximálny počet synov vrcholu v strome, depth je hĺbka stromu, pretože v zásobníku nikdy nebude viac rekurzívnych volaní ako je hĺbka menšieho zo vstupných stromov.

V danom rekurzívnom volaní nepotrebujeme viac ako $c_1 \cdot \maxdeg(A) \cdot \maxdeg(B)$, pre nejaké kladné c_1 priestoru na dočasné polia. Priestor na pamätanie zoznamu namapovaných vrcholov je lineárny k počtu vrcholov stromu A. Pri výpočte v danom rekurzívnom volaní potrebujeme najviac priestor $c_2 \cdot n \cdot \maxdeg(B)$, týchto volaní je v zásobníku najviac $\min(\text{depth}(A), \text{depth}(B))$. Spolu sa dá potrebný priestor na uloženie dočasných zoznamov ohraničiť výrazom $c_2 \cdot n \cdot \maxdeg(B) \cdot \min(\text{depth}(A), \text{depth}(B))$, pre nejaké kladné c_2 , kde \maxdeg je maximálny počet synov vrcholu v strome, depth je hĺbka stromu a n je počet vrcholov v strome A (obvykle je to oveľa menej priestoru). Sčítaním vyššie uvedených dvoch výrazov priestorových zložitostí získame výslednú priestorovú zložitosť: $O((n + \maxdeg(A)) \cdot \maxdeg(B) \cdot \max(\text{depth}(A), \text{depth}(B)))$, ani toto ohraničenie nie je príliš tesné.

Popritom sa maximálny počet synov pre ľubovoľný vrchol v strome (\maxdeg) rovná maximálnej dĺžke pravej strany pravidla, a táto nebýva príliš veľká. Vyššie popísané operácie na znižovanie hĺbky stromu nám síce môžu toto číslo zvýšiť, okrem nich aj parsovanie komentárov (ktoré sa do parsovacieho stromu pridávajú na ľubovoľné miesto a ľubovoľne veľa za sebou), ale aj tak je v bežnom prípade priestorová zložitosť stále pomerne nízka.

Najhorším prípadom je situácia, keď sa strom bude podobat' zoznamu (napr. všetky vrcholy zavesené pod koreňom). V tomto prípade algoritmus skutočne vyžaduje $O(nm)$ priestoru, čo pri napr. 100000 vrcholoch vychádza na 10^{10} , čiže približne 10 gigabytov priestoru (ak by sa pamäťové nároky jedného vrcholu rovnali iba jednému bytu).

Problémy nám môžu spôsobovať komentáre a rôzne príkazy kompilátoru. Keďže sa nerobí kompilácia, tak týmto príkazom typu `{ $IFDEF DEBUG } ... { $ENDIF }` v Delphi, resp. `#ifdef #else #endif` v C++ nerozumieme, teda nevieme určiť, či je daná podmienka splnená a teda, či máme parsovať kód medzi nimi, alebo nie. V tomto programe sa takéto príkazy berú ako komentáre (teda sa buď ignorujú, alebo vkladajú do výsledného stromu, podľa nastavení parsovania komentárov), a porovnáva sa celý vstup, čo môže spôsobiť nesprávne dvojice v mapovaní a trochu rozsynchronizovať porovnávanie. V niektorých prípadoch môžu tieto direktívy pre kompilátor spôsobiť, že súbor nebude možno parsovať, lebo bude syntakticky správny iba pre určité sekcie, ak ostatné sú ignorované, ale toto nemôžeme vedieť, pretože sémantiku vstupu ignorujeme.

V prípade, že vstupné súbory nie sú syntakticky korektné podľa vstupnej gramatiky, tak porovnávanie ich stromov bude pomerne neúspešné (v určitých prípadoch sa ešte dajú celkom dobre porovnať, ale väčšinou je ich porovnanie zbytočné).

Čo sa týka komentárov, v popise gramatiky sa dá nastaviť, či sa budú ignorovať už pri lexikálnej analýze, alebo sa vložia do syntaktického stromu a budú sa normálne porovnávať ako ostatné vrcholy v syntaktickom strome.

Po skončení nejakého algoritmu porovnávania stromov získame obvykle edit script. Problémom však môže byť ho rozumne zobrazit', obzvlášť ak obsahuje operácie presunu, alebo kopírovania.

V programe sú dve možnosti ako zobrazovať výsledok z porovnávania stromov (teda v podstate tri ak sa berie do úvahy aj ukladanie zmien do súboru v podobe edit scriptu). V obidvoch sa zobrazia pôvodné textové súbory s vyznačením zmien daných tokenov, ktoré sa nenachádzajú v mapovaní, ale líšia sa v tom či používame aj formátovanie textu podľa pravidiel definovaných v súbore so vstupnou gramatikou (pretty printing), alebo podľa pôvodnej pozície v súbore. Ak formátovanie textu nepoužívame, potom sa zobrazovanie textu snaží čo najviac napodobovať originálne súbory v počte a type prázdnych miest a riadkov. Ak sa nezhoduje text, ktorý nie je v žiadnom tokene obsiahnutom v syntaktickom strome, tak sa zlúči s textom z druhého súboru (ak ide o prázdne miesta, alebo nové riadky), inak sa do obidvoch pridajú prázdne miesta a riadky tak, aby sa zhodovala dĺžka prázdnych miest i počet riadkov v zobrazených súboroch, aby sa dali prehľadne prezeráť. Ak sa formátovanie textu používa, tak sa o pôvodné súbory už ďalej nemusíme zaujímať a zobrazíme dané stromy podľa formátovacích pravidiel definovaných v príslušnej sekcii súboru popisujúceho gramatiku.

Postup je v oboch prípadoch podobný, paralelne prechádzame preorderom cez oba stromy. Nech u je momentálny vrchol v strome A , v je momentálny vrchol v strome B , nech máme už vypočítaný edit script (a popri tom aj mapovanie):

Ak (u,v) patrí do mapovania, tak sa vypíšu hodnoty obidvoch vo farbách prislúchajúcich zhode, alebo zmene, závisiac od toho, či sú skutočne vrcholy u a v zhodné, alebo len podobné, v obidvoch listingoch. Ak ich dĺžka nie je zhodná, tak sa pridá do tej kratšej na koniec potrebný počet medzier, aby výpis zostal synchronizovaný.

Ak u patrí do mapovania a v nepatrí, tak sa postupne vypisujú prvky zo stromu B vo farbách prislúchajúcich vloženiu prvku v obidvoch listingoch (pričom v strome A bude namiesto textu vypísaný iba vhodný počet medzier, aby výpis zostal synchronizovaný), počnúc vrcholom v , pokračujúc preorderom stromu B , až pokiaľ pre nejaký vrchol w patriaci B opäť nebude platiť, že (u,w) sú v mapovaní.

Ak u nepatrí do mapovania a v patrí, tak sa postupne vypisujú prvky zo stromu A vo farbách prislúchajúcich vymazaniu prvku v obidvoch listingoch (pričom v strome B bude namiesto textu vypísaný iba vhodný počet medzier, aby výpis zostal synchronizovaný), počnúc vrcholom u , pokračujúc preorderom stromu A , až pokiaľ pre nejaký vrchol z patriaci A opäť nebude platiť, že (z,v) sú v mapovaní.

Pri vypisovaní textu je treba zároveň sledovať formátovanie textu. Ak sa používa formátovanie na základe pravidiel definovaných v súbore, kde je popísaná vstupná gramatika, tak vďaka preorderu je to celkom elegantné. Trošku komplikovanejší je postup ak formátovanie textu nepoužívame, a teda sa musia pri výpise napodobovať pôvodné súbory, zlučovať prázdne miesta a riadky, t.j. snažiť sa zachovávať ich autentický vzhľad a pritom dodržiavať synchronizáciu výpisov.

Samozrejme nie vždy sa nám musia zmestiť kompletne súbory a ich syntaktické stromy do pamäte. Potom veľa možností nemáme a viac menej musíme program rozsekať na

bloky a porovnávať jednotlivé bloky ako vstupy. Minimalizáciou použitých diskových operácií pri porovnávaní takýchto veľkých súborov sa zaoberá napr. [5]. V tejto práci sa však predpokladá, že vstupné súbory sa zmestia do operačnej pamäte.

V prípade, že máme okrem gramatiky k dispozícii aj sémantické informácie, tak môžeme výrazne zlepšiť výsledky porovnávania. Napr. ak vieme, že daný súbor je zdrojový kód, ďalej že v danom programovacom jazyku sú procedúry a funkcie nazvané určitým spôsobom, že bloky začínajú určitým znakom, že premenné sú v danej sekcii a sú unikátne vzhľadom na bloky, atď., tak sa dajú samozrejme zostrojiť lepšie a určite aj prakticky využiteľnejšie porovnávacie programy. Avšak cieľom tejto práce bolo aj zistiť, ako veľmi nám pomôže (resp. nepomôže) zadefinovaná gramatika pre vstupné súbory pri ich porovnávaní (t.j. v konečnom dôsledku, že ak máme vybudované syntaktické stromy zo vstupných súborov, tak ako dobre vieme zisťovať rozdiely medzi nimi).

5. Zhodnotenie a ďalší vývoj

Výsledný program je prakticky použiteľný pre malé a stredne veľké súbory, tak do 10000 riadkov, resp. 100000 vrcholov v syntaktických stromoch. Je v ňom implementovaný aj algoritmus na výpočet editovacej vzdialenosti a edit scriptu medzi dvomi postupnosťami (Myersov $O(ND)$ algoritmus, popísaný v tretej kapitole), aj algoritmus na výpočet štruktúrálnej vzdialenosti a edit scriptu medzi dvomi syntaktickými stromami (TreeScore algoritmus popísaný v predošlej kapitole) a aj jednoduché porovnávanie priečinkov, kde sú súbory porovnávané $O(NP)$ algoritmom, pre zisťovanie editovacej vzdialenosti [16].

V porovnaní s FMES dáva tento algoritmus (TreeScore) na porovnanie syntaktických stromov viac zmysluplné výsledky, hlavne pri porovnávaní zdrojových kódov, avšak je od neho výrazne pomalší.

Vďaka tomu, že nerobí presuny vrcholov (operácia presunu podstromu je vo výsledku porovnávania nahradená operáciou vymazania a vloženia podstromu), pomáha aj pri zobrazovaní výsledkov. Pre zobrazovanie je predsa len jednoduchšie (a aj pre čítanie, prezeranie), ak nemusíme zobrazovať aj presuny. Pri jednoduchých presunoch je to ešte v poriadku, ale pri kombinácii presunov, vložení a vymazaní pri FMES algoritme to už bol niekedy poriadny zmätok. Operácia presunu sa síce používa pri vytváraní dokumentov a zdrojových kódov, ale nie nejako extrémne často (ako bolo ukázané v štúdiu zaoberajúcej sa evolúciou programov, a ako som nakoniec zistil aj ja, keď som porovnával zdrojové kódy tohto programu, avšak určite záleží aj na štýle programovania). Skôr sa používa operácia kopírovania a následnej modifikácie, hlavne keď sa pre nedostatok času (čiže takmer vždy) nestíha refaktORIZOVAŤ vyvíjaný program tak, aby spoločné časti boli napísané iba raz v nejakej funkcii a tá použitá na relevantných miestach. Z krátkodobého hľadiska je jednoduchšie zobrať kód z jedného miesta programu a skopírovať ho na druhé, trochu zmeniť a nová funkcionálna je hotová. Avšak problém nastáva, ak sa v pôvodnej časti kódu napríklad nájde chyba, tá sa časom na danom mieste odstráni, ale na skopírovaných miestach pretrvá, lebo medzitým sa už dávno zabudlo, že odkiaľ je daný kus kódu. Nanešťastie je zisťovanie a podpora takýchto operácií – presun (resp. kopírovanie) podstromu a jeho modifikácia ešte oveľa náročnejšia na zložitosť (v trochu oklieštenej podobe sa tomuto venovali v [9], pri zhoršení výsledného času o kvadratický faktor sa im podarilo dorobiť do Zhang-Shashovho algoritmu operáciu výmena súrodencov s modifikáciou ich podstromov – swap+edit).

Zaujímavým rozvinutím tejto práce by mohlo byť spojenie s algoritmi na analýzu podobnosti dvoch súborov (similarity analyzers) a či by nemohli byť využité v predspracovávaní, alebo pri vytváraní približného mapovania. Tieto programy sa zaoberajú presne tým problémom skopírovania textu z jedného miesta na druhé a prípadnej miernej modifikácie, ktorý bol spomenutý vyššie. Obvykle pracujú na báze hašovania kusov kódu, rádovo pár riadkov, nájdenia nejakej zhody a postupnom zväčšovaní tejto zhody.

Ďalšie rozšírenie by mohlo byť dorobenie aplikovania edit scriptov na súbory a zjednocovanie verzií jednotlivých súborov do tejto aplikácie (patching and merging).

Podpora behu pod rôznymi operačnými systémami, čo sa týka Linuxu, tak v tomto ohľade by to malo ísť pomerne jednoduchou rekompiláciou zdrojových kódov s minimálnymi zmenami v Kylix (keďže je program napísaný v Delphi).

Umožnenie definovania nejednoznačných gramatík, napr. cez priority operátorov a pravidiel, resp. rozšíriť funkčnosť LALR parsera o spracovávanie sémantických atribútov.

Podpora Unicode.

Podpora viacerých jazykov v užívateľskom rozhraní.

6. Záver

Práca bola venovaná problematike štruktúrneho porovnávania zdrojových kódov a dokumentov.

Na začiatku uviedla čitateľa do problematiky a poskytla prehľad o najznámejších algoritmoch na porovnanie reťazcov, resp. stromov. Tento prehľad môže byť využitý aj pri tvorbe iných aplikácií podobného charakteru.

Úspešne boli implementované algoritmy porovnávania reťazcov a stromov a algoritmy na konštrukciu DFA automatov a LALR parserov.

Výsledok nie je príliš pozitívny, lebo žiadne optimálne riešenie, ktoré by bolo použiteľné pre všetky typy vstupných súborov sa nepodarilo nájsť. Postupy boli buď príliš pomalé na reálne použitie, alebo vytvárajú iba približné riešenia a sú pomerne ľahko „poraziteľné“, alebo neprodukujú výstup ktorý by bol príliš žiaduci (t.j. kvalitatívne oveľa lepší ako pri porovnávaní reťazcov).

Práca ukázala, že ak chceme dosiahnuť lepšie výsledky, je v podstate nevyhnutné implementovať štruktúrne porovnanie aj na základe sémantických vlastností, lebo čisto syntax nám priveľmi nepomôže pri zlepšovaní výsledkov porovnania a hľadani minimálneho počtu transformácií.

Výsledkom tejto práce je program, ktorý porovnáva dané súbory na základe vstupnej gramatiky a zobrazuje rozdiely medzi nimi (na základe vzdialenosti ich syntaktických stromov), umožňuje aj klasické porovnanie vstupných súborov bez vstupnej gramatiky, na základe editovacej vzdialenosti medzi nimi a umožňuje tieto rozdiely uložiť do súboru. Program môže fungovať aj ako „pretty printer“, teda môže formátovať dokumenty a zdrojové kódy podľa pravidiel, ktoré sa dajú definovať v gramatike. A v princípe sa dá použiť aj ako pomôcka pri vytváraní LALR gramatík v BNF notácii.

Dúfam, že tento program bude niekomu niekedy na niečo osožný, ja som ho už niekoľko krát využil pri porovnávaní rôznych verzií tohto programu (pri odstraňovaní chýb).

7. Slovník pojmov

- BNF** Jednoduchá notácia na popis syntaktických pravidiel v gramatike. (Backus Naur Form)
- CFG** Označenie bezkontextovej gramatiky, definícia v kapitole 1. (Context Free Grammar)
- CVS** Systém na manažment zmien v množine súborov, ktorý sa obvykle používa pri softvérových projektoch, hlavne na umožnenie spolupráce množstva vývojárov (ktorí sú potenciálne v rôznych kútoch sveta) na danom projekte. (Concurrent Versioning System)
- CYK** Metóda na rozpoznávanie všeobecných bezkontextových jazykov, pracujúca v časovej zložitosti $O(n^3)$. Tento algoritmus využíva redukciu všeobecnej CFG na Chomského normálny tvar a následne algoritmus dynamického programovania pracujúci nad maticou vytvorenou zo vstupných neterminálov. (Cocke – Younger – Kassami)
- DFA** Deterministický konečný automat (Deterministic Finite Automaton)

edit script

- Postupnosť elementárnych zmien po aplikácii ktorej dostaneme z pôvodnej postupnosti A, postupnosť B. Keďže väčšinou je takýchto postupností nekonečne veľa (ak máme k dispozícii napr. operácie vloženie a vymazanie, tak môžeme vždy spraviť novú v ktorej bude navyše prídanie prvku x a jeho následné vymazanie), tak nás obvyčajne zaujíma minimálny edit script, kde za kritérium minimality volíme dĺžku tejto postupnosti, alebo jej váhu (vytvoríme funkciu ktorá nám ku každej operácii vráti jej váhu. Potom váha edit scriptu je suma váh jednotlivých prvkov). Minimálny edit script (minimálna postupnosť zmien) M je taký, že neexistuje edit script M' s menšou váhou. Niekedy sa váha označuje aj ako cena.
- EZS** Algoritmus na zisťovanie edit scriptu medzi dvomi vstupnými stromami, je popísaný v kapitole 3. (Extended Zhang–Shasha algorithm)
- FMES** Algoritmus na zisťovanie edit scriptu medzi dvomi vstupnými stromami, je popísaný v kapitole 3. (Fast Match Edit Script algorithm)
- label** Nejaké označenie (v tejto práci označenie vrcholu v strome)
- LALR** Používané v spojeniach LALR parser, LALR gramatika. Parser, ktorý je LALR číta vstup zľava doprava a vytvára pravé krajné odvodenie rovnako ako LR parser, avšak nemá tak veľa stavov ako LR parser (pretože stavy s rovnakým

jadrom sú reprezentované jedným stavom), na druhej strane neakceptuje ľubovoľnú LR gramatiku (t.j. existujú LR gramatiky, pre ktoré neexistuje LALR parser). Gramatiku označíme LALR, ak existuje LALR parser, ktorý akceptuje všetky vetné formy odvoditeľné z tejto gramatiky a žiadne iné. Bližšie popísané v kapitole 1. (Look Ahead Left-to-right Right-most derivation)

LCS Najdlhšia spoločná podpostupnosť (Longest Common Subsequence)

lexer Lexikálny analyzátor

LL Používané v spojeniach LL parser, LL gramatika. Parser, ktorý je LL číta vstup zľava doprava a vytvára ľavé krajné odvodenie. Gramatiku označíme LL, ak existuje LL parser, ktorý akceptuje všetky vetné formy odvoditeľné z tejto gramatiky a žiadne iné. Bližšie popísané v kapitole 1. (Left-to-right Left-most derivation)

lookahead

Symboly (resp. tokeny, ak máme vstup z lexikálneho analyzátoru), ktoré parser „vidí“ od momentálnej pozície dopredu na vstupe

LR Používané v spojeniach LR parser, LR gramatika. Parser, ktorý je LR číta vstup zľava doprava a vytvára pravé krajné odvodenie. Gramatiku označíme LR, ak existuje LR parser, ktorý akceptuje všetky vetné formy odvoditeľné z tejto gramatiky a žiadne iné. Bližšie popísané v kapitole 1. (Left-to-right Right-most derivation)

NFA Nedeterministický konečný automat (Nondeterministic Finite Automaton)

parser Syntaktický analyzátor

RCS Systém, ktorý automatizuje ukladanie, spájanie, porovnávanie rôznych verzií súborov, predchodca CVS. Na rozdiel od CVS pracuje iba na úrovni jednotlivých súborov a nie v rámci celého projektu a neumožňuje viacerým vývojárom pracovať na tom istom súbore. (Revision Control System)

SES Najkratší edit script (Shortest Edit Script)

Unicode

Univerzálny štandard pre kódovanie znakov používaných pre reprezentáciu textu v počítačoch

XML Jednoduchý a veľmi flexibilný jazyk na tvorbu štruktúrovaných dokumentov, ktorý vznikol v roku 1998 zjednotením dovtedajších jazykov na tvorbu štruktúrovaného textu a hypertextu. (eXtensible Markup Language)

8. Použitá literatura

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques and Tools*. Addison-Wesley (1977)
- [2] E. W. Myers: *An $O(ND)$ Difference Algorithm and Its Variations*. *Algorithmica*, vol. 1 (1986)
- [3] K.Zhang and D. Shasha: *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems*. *SIAM Journal of Computing*, 18,(6) (1989)
- [4] S. Chawathe, A. Rajaraman, H.Garcia-Molina and J.Widom: *Change Detection in Hierarchically Structured Information*. *Proceedings of the ACM SIGMOD International Conference on Data Engineering*, New Orleans, Louisiana, USA (1996)
- [5] S. Chawathe: *Comparing Hierarchical Data in External Memory*. *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, Edinburgh, Scotland, UK (1999)
- [6] Y. Wang, D. DeWitt, Jin-Yi Cai: *X-Diff: An Effective Change Detection Algorithm for XML Documents*. *Proc. of the ICDE Conference*, Bangalore, India (2003)
- [7] Gabriel Valiente: *An Efficient Bottom-Up Distance between Trees*. *Proceedings of 8th International Symposium on String Processing and Information Retrieval*, Laguna de San Rafael, Chile (2001)
- [8] Philip Bille: *Tree Edit Distance, Alignment Distance and Inclusion*. *Technical Report TR-2003-23*, IT University of Copenhagen (2003)
- [9] David T. Barnard, Gwen Clarke, Nicholas Duncan: *Tree-to-tree correction for document trees*. *Technical Report 95-372*, Department of Computing and Information Science, Queen's University, Kingston, UK (1995)
- [10] P.N. Klein: *Computing the edit-distance between unrooted ordered trees*. *Proceedings of the 6th annual European Symposium on Algorithms*, Venice, Italy (1998)
- [11] Stanley M. Selkow: *The tree-to-tree editing problem*. *Information Processing Letters*, 6(6) (1977)
- [12] Wagner, R.A. and Fisher, M.J.: *The String-to-String Correction Problem*. *Journal of the Association of Computing Machinery*, 21(1) (1974)
- [13] Hirschberg, D.S.: *A Linear Space Algorithm for Computing Maximal Common Subsequences*. *Communications of the ACM* 18,(6) (1976)

- [14] K.-C. Tai: *The tree-to-tree correction problem. Journal of the Association for Computing Machinery* (1979)
- [15] Masek, W.J. and M.S. Paterson: *A faster algorithm for computing string-edit distances. Journal of Computer and System Sciences*, 20 (1980)
- [16] Sun Wu, Udi Manber, Gene Myers: *An $O(NP)$ Sequence Comparison Algorithm. Information Processing Letters*, 35 (1989)
- [17] Philip Bille: *Ordered Tree Edit Distance with Merge and Split Operations. Technical report TR-2003-35 in IT University of Copenhagen Technical Report Series* (2003)
- [18] Cobena: G.Cobena, S. Abiteboul, A. Marian: *Detecting Changes in XML Documents. Proceedings of the 18th IEEE International Conference on Data Engineering, San Jose, California* (2002)
- [19] Weimin Chen: *New algorithm for ordered tree-to-tree correction problem. Journal of Algorithms*, 40 (2001)
- [20] Kaizhong Zhang and Tao Jiang: *Some MAX SNP-hard results concerning unordered labeled trees. Information Processing Letters* 49,(5) (1994)
- [21] S. Chawathe, H. Garcia-Molina: *Meaningful Change Detection in Structured Data. Proceedings of the ACM SIGMOD International Conference on Data Engineering , Tuscon, Arizona, USA* (1997)

Prílohy

Inštalácia

Inštalácia nie je potrebná, program sa dá spustiť priamo z priloženého CD-ROMu. Výhodnejšie je ho skopírovať na disk, už kvôli inicializačnému súboru, v ktorom sa uchovávali cesty k posledne otvoreným súborom a všetky nastavenia programu, a teda sa po každom spustení zvykne meniť.

Obsah CD

Na dodanom CD-ROMe sa v hlavnom priečinku nachádzajú nasledovné súbory:

- **ParseTreeDist.exe**
skompilovaný program (po spustení si vytvorí konfiguračný súbor)
- **ParseTreeDist.cnt**
súbor s obsahom pre zobrazovanie pomoci v programe
- **ParseTreeDist.hlp**
súbor obsahujúci manuál vo Winhelp formáte

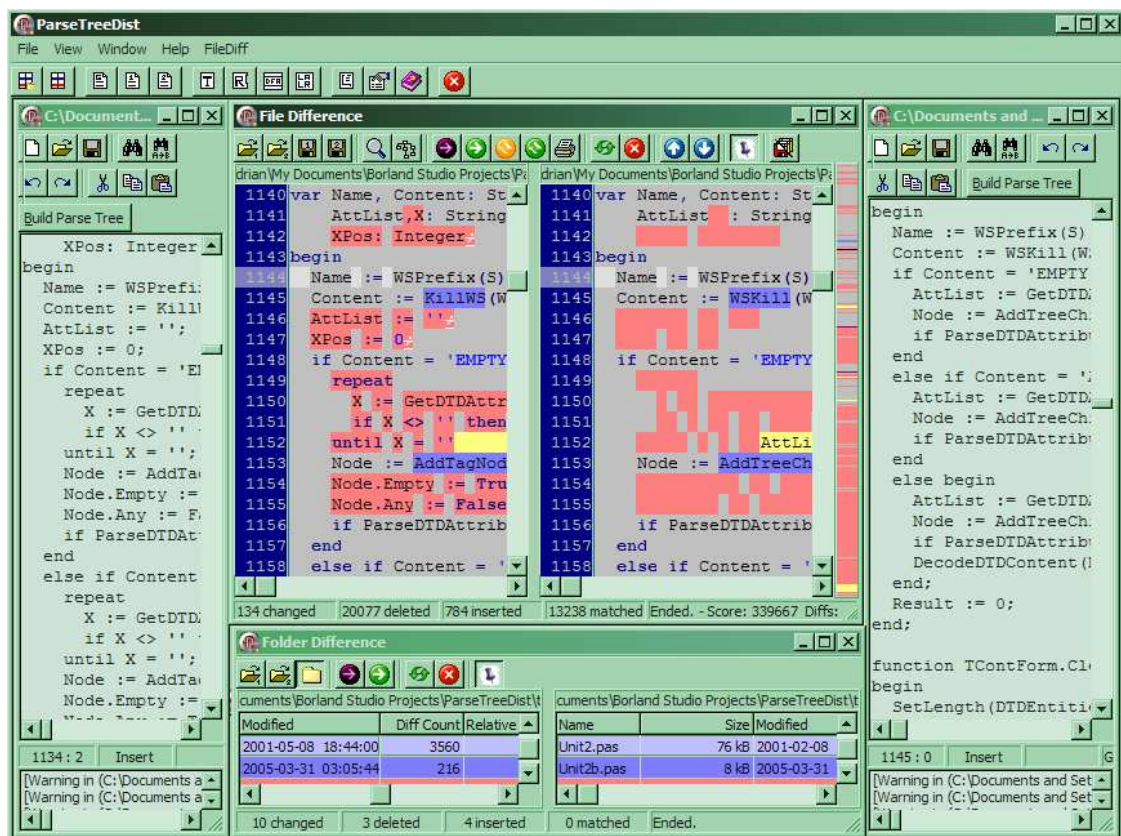
Ďalej sa v ňom nachádzajú nasledovné priečinky:

- **src**
obsahuje kompletne zdrojové kódy (900kB)
- **doc**
obsahuje manuál v PDF a HTML formáte, takisto obsahuje dokumentáciu k zdrojovým kódom v HTML formáte, ďalej sa tu nachádza aj elektronická verzia tejto diplomovej práce v PDF i DOC formáte
- **test**
obsahuje nejaké zdrojové kódy a aj gramatiky pre Javu, Delphi, HTML. Všetky sú neúplné, čiže určite sa nájdu štruktúry, ktoré spôsobia, že LALR parser sa zastaví po dosiahnutí určitého počtu chýb a teda zdrojový súbor bude nespracovateľný. Ale s Delphi gramatikou som spravil syntaktickú analýzu zdrojových kódov tohto programu a aj niekoľkých mojich ďalších programov a s Java gramatikou som spravil syntaktickú analýzu zdrojových kódov niekoľkých projektov zo SourceForge (<http://www.sourceforge.net>)

Príklady použitia

Využitie na porovnávanie súborov

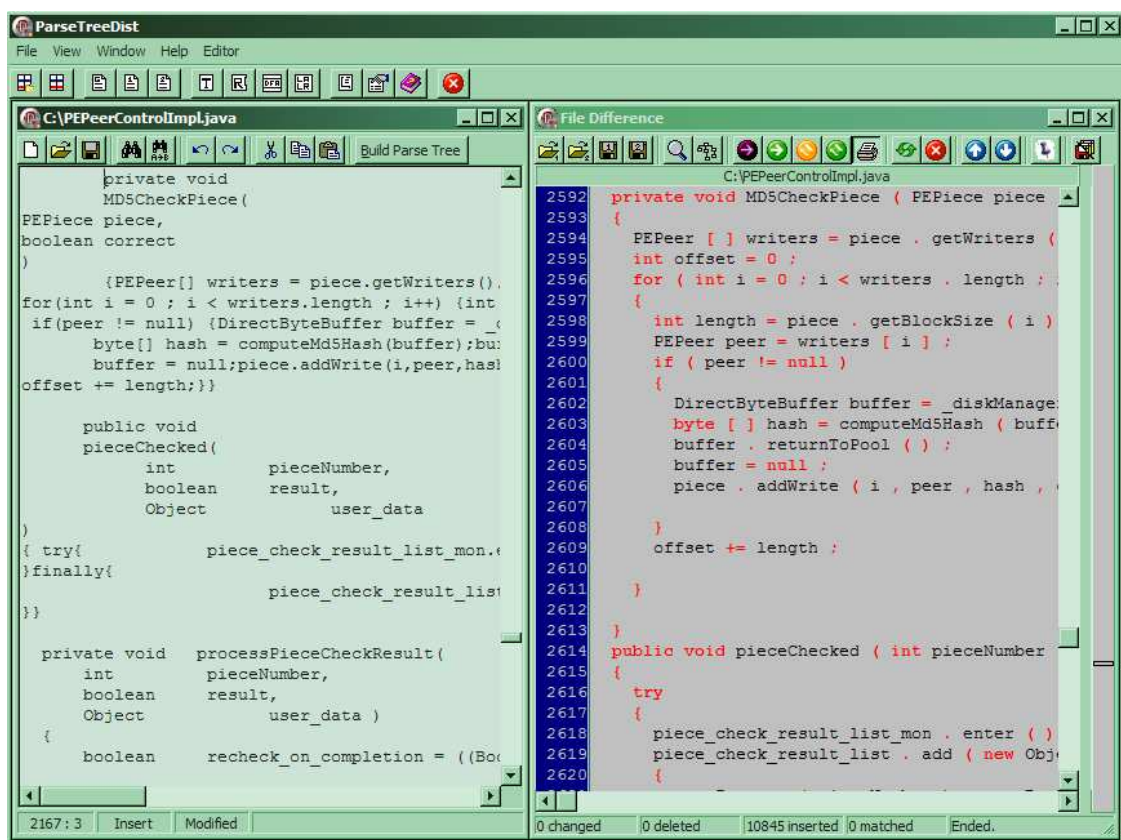
Spustí sa program, vyberieme priečinok pre ľavý panel, obvykle hlavný priečinok obsahujúci staršiu verziu zdrojových kódov. V pravom paneli si vyberieme priečinok obsahujúci novšiu verziu zdrojových kódov. Odtiaľto si buď priamo vyberieme súbory, ktorých rozdiely nás zaujímajú, alebo priečinky porovnáme a utriedime, napr. podľa počtu rozdielov. Otvorí sa nové okno obsahujúce dva konkrétne súbory. Tieto môžeme porovnať klasickým spôsobom ako dva reťazce (po riadkoch), ako pri diff utilitách a zobrazia sa nám farebne rozlíšené riadky (v prípade zmenených riadkov, sa po kliknutí na ne môžu zobrazit' rozdiely v rámci daného riadku, ak máme túto možnosť nastavenú v nastaveniach). Alebo môžeme tieto súbory štruktúrálné porovnať. Ak zatiaľ ešte nebola žiadna gramatika vybratá, tak sa nám objaví dialógové okno v ktorom vyberieme vstupnú gramatiku. Táto sa spracuje, vytvorí sa lexikálny analyzátor (pracujúci ako NFA, alebo DFA, podľa nastavenia v nastaveniach), LALR parser a tento sa použije na vytvorenie syntaktických stromov pre vstupné súbory. Tieto stromy sa následne porovnajú a rozdiely sa zobrazia. Rozdiely medzi nimi môžeme následne uložiť do výstupného súboru.



Obr. 9: Porovnávanie súborov

Využitie na formátovanie / skrášľovanie súborov (pretty printing)

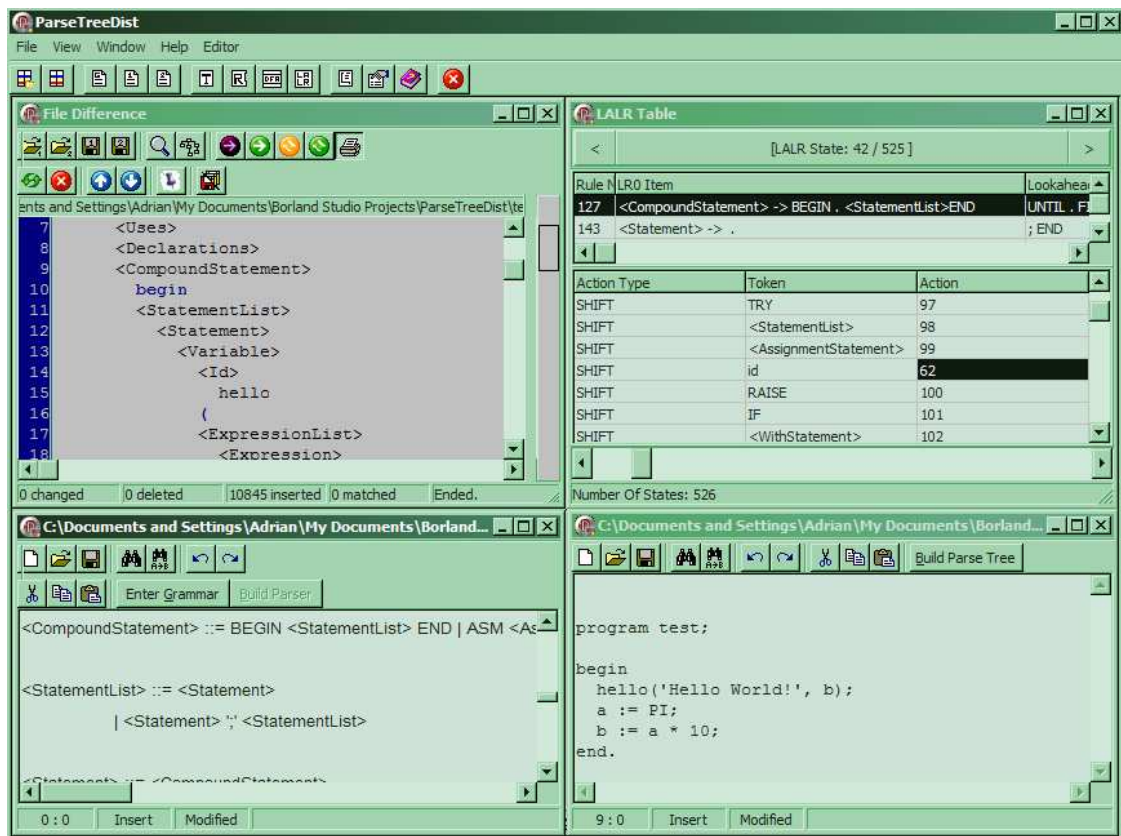
Spustí sa program, otvoríme okno s porovnávaním súborov, v jednom z panelov si vyberieme súbor, ktorý chceme inak naformátovať, prípadne vymazať komentár a podobne (code beautifying). Ďalej určíme gramatiku, v ktorej je sú definované pravidlá na formátovanie. Ak zatiaľ ešte nebola žiadna gramatika vybratá, tak sa nám objaví dialógové okno v ktorom vyberieme vstupnú gramatiku. Táto sa spracuje, vytvorí sa lexikálny analyzátor (pracujúci ako NFA, alebo DFA, podľa nastavenia v nastaveniach), LALR parser a tento sa použije na vytvorenie syntaktického stromu. Podľa tohto stromu a formátovacích pravidiel definovaných v gramatike sa nám naformátuje zobrazený text. Toto nové naformátovanie môžeme uložiť do nového súboru.



Obr. 10: Formátovanie vstupu

Využitie na vytváranie a testovanie LALR gramatík

Vzhľadom k tomu, že v programe sa dajú definovať a analyzovať LALR gramatiky, tak je možné ho využiť aj pri ich tvorbe. Na obrazovke si necháme súčasne zobrazovať gramatiku, ktorú práve vytvárame, výsledný DFA lexikálny analyzátor a LALR parser. Do ďalšieho okna si môžeme písať testovací vstup a tento skúšať preparsovať a program nám zobrazí prípadné chyby. Ďalej si môžeme nechať zobrazovať parsovací strom, na ktorom môžeme sledovať, či už je náš testovací vstup parsovaný korektne.



Obr. 11: Písanie LALR gramatiky

Pre viac detailov o ovládaní programu treba pozrieť manuál.