TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Reina Käärik

ÜLDISTATUD TEISENDUSKAUGUSE RAKENDAMINE SÕNEDE SARNASUSE HINDAMISEKS

Bakalaureusetöö (10AP)

Juhendaja: Jaak Vilo, PhD

Autor:.. "....." mai 2006

Juhendaja:... "....." mai 2006

Õppetooli juhataja:............................"....." mai 2006

TARTU 2006

Sisukord

1. Sissejuhatus ... 3

2. Definitsioonid ... 6

2.1 Sõne ... 6

2.2 Graaf, suunatud graaf ... 7

2.3 Puu ja prefiksipuu andmestruktuurid ... 7

2.4 Dünaamiline programmeerimine ... 10

3. Levenshteini kaugus ... 12

3.1 Teisenduskauguse arvutamine graafi abil ... 14

3.2 Teisenduskauguse arvutamine kasutades dünaamilise programmeerimise tehnikat ... 16

4. Üldistatud teisenduskaugus ... 19

4.1 Üldistatud teisenduskauguse leidmine graafide abil ... 20

4.2 Üldistatud teisenduskauguse leidmine kasutades dünaamilise programmeerimise tehnikat ... 23

4.2.1 Teisenduste hulga organiseerimine prefiksipuu abil ... 28

4.2.2 Üldistatud teisenduskauguse dünaamilise programmeerimise tabelist teisenduste taastamine ... 34

5. Programm üldistatud teisenduskauguse leidmiseks ... 37

6. Üldistatud teisenduskauguse programm ja Unicode ... 43

6.1 Unicode ... 44

6.2 Kodeeringud ... 45

6.2.1 UTF-32 ... 45

6.2.2 UTF-16 ... 46

6.2.3 UTF-8 ... 46

6.3 Üldistatud teisenduskauguse programm Unicode toega ... 47

6.3.1 Programmi kasutamine Linuxi all ... 48

6.3.2 Programmi rakendamine ... 48

7. Üldistatud teisenduskauguse sõnedefaili teisendamine prefiksipuu kujule ... 51

8. Üldistatud teisenduskauguse teisendustele kaalude leidmine ... 59

8.1 Ladina ja vene keele tähtede vahelistele teisendustele kaalude automaatne õppimine........... 63
Kokkuvõte.. 65
Summary..66
Viited.. 67
URL-id..68
Lisad... 70
Lisa 1
Üldistatud teisenduskauguse programm Unicode toega.. 70
Lisa 2
Üldistatud teisenduskauguse programm, mis kasutab sõnede jaoks prefiksipuu andmestruktuuri
.... 70
Lisa 3
Programm kaalude leidmiseks... 70
Lisa 4
Antud töös kasutatud sõnede failid...70

1. Peatükk
Sissejuhatus

Väga paljudes rakendustes on tarvis mõõta sõnede vahelist sarnasust. Sõnede võrdlemist võib kohata nii bioinformaatikas, veebiotsingusüsteemides, andmete puhastamises, käekirja tuvastamises, kõnetuvastuses kui ka õigekirjakontrollijates.

Õigekirja kontrollimist arvuti abil on uuritud juba pikka aega. Aastal 1964 täheldas Damerau, et üle 80% vigaselt kirjutatud sõnadest sisaldavad ainult ühte viga – neis sisaldub üleliigne täht, mõni täht on puudu, mõni täht on asendatud teisega või kaks kõrvutiasetsevat tähte on vahetanud oma kohad [Dam64]. Seega kahe sõne sarnasuse küsimuse saab taandada sõnede võrdlusele, kasutades just neid teisendusoperatsioone – tähe lisamist, kustutamist, asendamist või vahetamist. Tihti vaadeldakse kahe kõrvutiasetseva tähe vahetamist kahe operatsioonina – tähe lisamise ja tähe kustutamisena.

Sõnede võrdlemiseks on võimalik kasutada Levenshteini kaugust, tuntud ka teisenduskauguse nime all. Esimesena tutvustas teisenduskauguse leidmise algoritmi aastal 1965 Vene teadlane Vladimir Levenshtein, kelle järgi sai see ka oma nime. Teisenduskaugust võib vaadelda kui teisendusoperatsioonide arvu, mis tuleb teha, et saada esialgsest sõnest teine.

Võimalused, mida meie pakub tavaline teisenduskaugus, ei ole sageli piisavad, sest seal kehtib eeldus, et kõik teisendused toimuvad sama tõenäosusega. Kuid päris elus see nii alati ei ole: näiteks kui vaadata DNA-s evolutsiooni käigus toimunud muutusi, siis kõige sagedasemaks muutuseks selles on ühe nukleotiidi asendamine teisega või väikese ploki kõrvutiasetsevate nukleotiidide lisamine või kustutamine. Vahel on muutuseks mõne lõigu asendamine selle sama lõigu pööratud järjekorraga elementidega, mõne lõigu asukoha muutus, kahe kromosoomi lõpus olevate lõikude vahetus ning mingi lõigu dubleerimine [Ped00]. Kui vaadelda näiteks eesti keeles ajalooliselt toimunud muutusi: uurides 18. sajandist pärinevaid eestikeelseid tekste, siis võib märgata, et võrreldes praeguse aja eesti keele kirjapildiga, on seal väga paljud kaashäälikuid kirjutatud ühe tähe asemel kahega – näiteks *vanna, pühha.* Praeguses kirjapildis oleksid need vastavalt *vana* ja *püha.*

Samas mitte kunagi ei esine näiteks *k* tähe kahekordselt kirjutamist. Kahekordse täishääliku asemel kasutati tihti ühekordset täishäälikut – *kulus*, *sowib* (praeguses kirjapildis *kuulus* ning *soovib*). Kui arvutaksime nendes tekstides olevate sõnade sarnasusi nende tänapäevaste kirjapiltidega kasutades tavalist teisenduskaugust, siis sõnale *tössine* oleks sama lähedased nii sõna *tõsine* kui ka näiteks sõnad *tassike* või *töine*. Otsides vanaaegsetele sõnadele vasteid sooviksime, et just neist esimene oleks antud sõnale lähedane.

Vaadelda võiks ka inimeste poolt trükkimisel tehtavaid vigu – sagedaseim on nende tähtede asendamine, mis asuvad arvuti klaviatuuril teineteisega lähestikku – suurema tõenäosusega on vigaselt kirjutatud sõnas asendatud täht *a* tähega *s* kui näiteks tähega *ü*. Väga sagedasteks vigadeks on ka õigekirjavead – näiteks *g* asendamine tähega *k* esineb palju suurema tõenäosusega kui *g* asendamine tähtedega *kk* või isegi tähega *r*.

Omamoodi probleem tekib ka eestikeelsetest dokumentidest otsinguga – kui tahetakse leida dokumenti, milles oleks kirjeldatud teisenduskaugust, siis võib esineda olukord, kus otsides sõna *teisenduskaugus*, ei pruugi me leida kõiki meid huvitavaid dokumente – dokumendid, mis sisaldavad ainult sõna *teisenduskauguse* või *teisenduskaugusega*, selle otsingu tulemuses ei ole.

Probleemi lahenduseks tutvustati autori semestritöös üldistatud teisenduskaugust, mis arvestab erinevate tähtede/täheühendite lisamiste, asendamiste ja kustutamiste jaoks erinevaid kaale. Antud töö on semestritööle otseseks jätkuks.

Autori semestritöö käigus valmis programm üldistatud teisenduskauguse leidmiseks. See programm eeldab, et ühe tähe kodeerimiseks kasutatakse ühte baiti. See aga seab piirangud programmile ning selle sisendile – näiteks ei saa defineerida teisendusi erinevate tähestike tähtede vahel. Samuti ei saa me leida teisenduskaugusi sõnede vahel, milles ühe tähe kujutamiseks kasutatakse rohkem kui ühte baiti. Käesolevas töös lisame sellele programmile Unicode toe ning sellega ka võimaluse defineerida teisendusi erinevate keelte tähestike vahel.

Semestritöö raames valminud üldistatud teisenduskauguse programm arvutab iga uue sõne puhul terve dünaamilise programmeerimise tabeli. Sellega tehakse programmi töö käigus väga palju lisatööd – otsisõne võrdlemisel sõnedega, millel on sama prefiks, on dünaamilise programmeerimise tabel prefiksi pikkuses identne. Siin on aga optimeerimise võimalus – lisame sõned, millega otsisõne võrdleme, prefiksipuusse. Sõnede puhul, millel on sama prefiks, kasutame prefiksi pikkuses sama dünaamilise programmeerimise tabelit.

Üheks suuremaks probleemiks üldistatud teisenduskaugusele teisenduste defineerinise juures on neile reaalsete kaalude leidmine. Antud töös tutvustame meetodit teisendusetele näidete hulga varal kaalude õppimiseks.

Käesolev töö on jaotatud kaheksaks peatükiks, neist esimesed viis on koostatud semestritöö põhjal. Teises peatükis selgitatakse töös kasutatud põhimõisteid. Kolmandas peatükis antakse ülevaade Levenshteini teisenduskaugusest ning selle leidmise võimalustest. Neljas peatükk käsitleb üldistatud teisenduskauguse leidmist. Viiendas peatükis on kirjeldatud autori semestritöö käigus valminud programmi üldistatud teisenduskauguse leidmiseks. Kuuendas peatükis antakse ülevaade Unicodest ning Unicode toega üldistatud teisenduskauguse programmist. Seitsmendas peatükis on tutvustatud üldistatud teisenduskauguse programmi optimeeritud varianti, mis loeb ka sõnede faili prefiksipuusse ning kasutab samade prefiksitega sõnede puhul ühise prefiksi jaoks sama dünaamilise programmeerimise tabeli osa. Kaheksandas peatükis on kirjeldatud meetodit teisenduste kaalude automaatseks õppimiseks näidete hulga baasil ning on antud ülevaade tulemustest.

2. Peatükk

Definitsioonid

2.1 Sõne

Tähistagu Σ lõplikku tähtede hulka, *tähestikku*. Tähestiku Σ võimsust tähistatakse $|\Sigma|$. Iga jada $S=a_1 a_2 ... a_n$, mille puhul $n \geq 0$ ja iga $a_i \in \Sigma$, nimetatakse *sõneks*. Sõne *S* pikkuseks $|S|$ on *n*. Sõnet pikkusega 0 on tähistatud λ.

Tähti eristame sõnes nende positsioonide järgi. Tähemärk a_i positsioonil *i* võib olla tähistatud ka *S*[*i*]. Tähemärkide positsioonid mittetühjas sõnes *S* on vahemikus $1 \leq i \leq |S|$ – sõne esimene täht on positsioonil 1 ning viimane täht on positsioonil $|S|$.

Sõnes *S* kõrvutiasetsevad tähed $a_i ... a_j$ moodustavad sõne *S* alamsõne, mis algab positsioonilt *i* ja lõpeb positsioonil *j*. Tähistame selle alamsõne *S*[*i*...*j*], kus $1 \leq i \leq j \leq |S|$ [Vil02].

2.2 Graaf, suunatud graaf

Graafiks G nimetatakse paari (*V, E*), kus *V* on tippude hulk ja *E* on servade hulk tippude vahel *E* = { (*u, v*) | $u, v, \in V$ } [url:BT]. Tavaliselt esitatakse graaf joonisena, kus iga tipp on kujutatud punktina tasandil ning iga serv kahte tippu ühendava joonena [BLV03]. Graafi, mille servadele on antud suund, nimetatakse *suunatud ehk orienteeritud graafiks.* Orienteeritud graafi servi nimetatakse *kaarteks*.

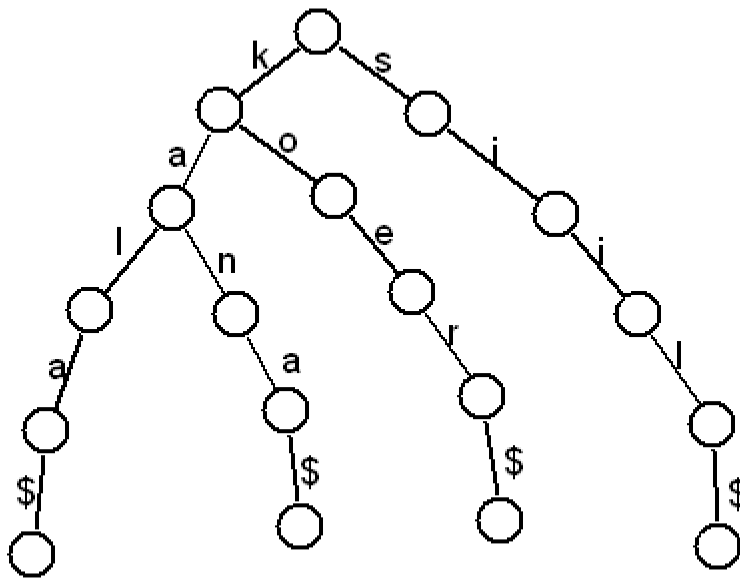
2.3 Puu ja prefiksipuu andmestruktuurid

Puuks nimetatakse lõplikku tippude hulka, mis on kas tühi või milles on üks tipp – *juur* ehk *juurtipp* – on välja eraldatud ning ülejäänud tipud on jaotatud $m \geq 0$ mittelõikuvaks alamhulgaks $T_1, T_2, \ldots T_m$, millest igaüks on omakorda puu; alamhulkasid $T_1, T_2, \ldots, T_m$ nimetatakse puu juure *alampuudeks*. *Järjestatud puu* korral nõutakse, et juure alampuude hulk oleks lineaarselt järjestatud. Alampuu juurt nimetatakse puu juure *alluvaks* (ka *vahetuks järglaseks*). Tippu nimetatakse oma alluva *ülemuseks* (ka *vahetuks eellaseks*). *Leht* ehk *lehttipp* on alluvateta tipp. Tippu, mis pole leht, nimetatakse *vahetipuks* ehk *sõlmeks* (ka *sisemiseks tipuks*). Tippu *y* nimetatakse tipu *x järeltulijaks*, kui leidub tee $x = t_{0,} t_{1,} \ldots, t_k = y$, kus iga *i* korral $(0 \leq i \leq k)$ t_{i+1} on t_i alluv. Tipp *x* on niisugusel juhul tipu *y eelkäija* ja tipp *y* asub tipust *x kaugusel k* [Kih03].

Prefiksipuu[1] (ingl.k. *trie*) on järjestatud puu sõnede talletamiseks. Prefiksipuu andmestruktuuri ülesehitus põhineb kahel printsiibil: fikseeritud hulgal indeksitel ja hierarhilisel indekseerimisel [url:trie1]. Olgu meil talletamiseks mingi hulk sõnesid $S \subseteq \Sigma^*$. Iga kaar, mis ühendab kahte sisetippu, on märgistatud ühe tähestiku Σ elemendiga. Iga kaar, mis viib leheni, on märgistatud sümboliga *$* (mingi sümboliga, mis tähestikku Σ ei kuulu). Iga sõne $s \in S$ jaoks leidub tee prefiksipuu juurtipust kuni leheni. Kui selle tee peale jäävate kaarte märgistused konkateneerida, siis need moodustavad sõne *s,* millele on lõppu lisatud lõpusümbol *$.* Iga juurtipust leheni viiva tee kaarte märgistuste konkatenatsioonil moodustub mingi sõne, mis kuulub hulka *S* [url:trie2]. Hulga *S* saame prefiksipuust taastada puu süvitsi läbimise algoritmi kasutades.

1 http://www.cc.ioc.ee/jus/gtglossary/gtglos_t.htm

Näide 2.1. Olgu salvestamiseks antud sõnede hulk $S=\{siil, kana, koer, kala\}$. Prefiksipuu, mis talletaks sõnede hulga *S,* on näha joonisel 2.1:



Joonis 2.1. Sõnede *siil, kana, koer, kala* kujutamine prefiksipuu andmestruktuuris.

Sõned, mis omavad sama prefiksit, on ühendatud ühe ja sama tipuga. Igal tipul saab olla maksimaalselt $|\Sigma|+1$ alluvat – igale erinevale tähele tähestikus tehakse oma haru, lisaks omaette haru lõpusümbolile *$*.

Sõne $A=a_1 a_2 ... a_m$ prefiksipuust otsimiseks kuluv aeg on halvimal juhul $O(m*|\Sigma|)$. Sõne otsimist tuleks alustada juurtipust ning vaadata selle vahetuid alluvaid. Kui otsitava sõne esimene täht a_1 on võrdne mõne juurtippu ja selle vahetut alluvat ühendava kaare märgendiga, siis jätkame otsingut sellest alampuust, mille juurtipuks on see alluv. Sõnes võtame vaatluse alla järgmisel positsioonil asuva tähe. Seda protseduuri jätkame, kuni ühegi vaatluse all oleva tipu ja selle alluva vahelise kaare märgend ei klapi hetkel vaatluse all oleva tähega või kui sobitasime sõne viimase tähe ning ühegi vaatluse all olevast tipust väljuva kaare märgendiks ei ole sõne lõppu märkiv sümbol – järelikult sellist sõnet antud prefiksipuus ei asu. Kui jõudsime sõne lõppu ning vaatluse all olevast tipust väljub kaar, mille märgendiks on sõne lõppu märkiv sümbol *$*, siis olemegi oma otsitava sõne leidnud. Järgnevalt toome algoritmi 2.1, mis kirjeldab mingi sõne otsimist prefiksipuust.

Algoritm 2.1. Sõne *S* otsimiseks prefiksipuust *T*.

Sisend: Otsitav sõne *S* = $s_1 s_2 \ldots s_n$ ja prefiksipuu *T*.

Väljund: True, kui antud prefiksipuus on see sõne talletatud ning **false**, kui sellist sõnet selles prefiksipuus ei leidu.

Meetod:

// võtame vaatluse alla T juurtipu

1. $T_{juur} = juur(T)$

// tähistame T_{juur} *vahetute järglaste hulka* $\{T_1, T_2, \ldots, T_{|T_{juur} vahetud\ järglased|}\}$

2. $täht = 1$ *// hetkel vaatluse all olev sõne S positsioon*

3. **for** i = 1 **to** $|\{T_{juur} vahetud\ järglased\}|$ **do**

4. **if** täht = n + 1 **and** $kaar(T_{juur}, T_i).märgend = '\$'$ **then**

5. **return true**

// saame prefiksipuus allapoole liikuda

6. **else if** $kaar(T_{juur}, T_i).märgend = sõne[täht]$ **then**

7. $T_{juur} = T_i$

8. täht++

9. **return false** *// vaatasime läbi kõik konkreetse tipu vahetud alluvad või vaadeldav tipp oli leht*

2.4 Dünaamiline programmeerimine

Dünaamiline programmeerimine on meetod algoritmide käitusaja vähendamiseks kasutades *ülekattuvaid alamprobleeme* ja *optimaalset alamstruktuuri*.

Optimaalse alamstruktuuri all mõtleme seda, et üldisele probleemile optimaalse lahenduse leidmiseks saab kasutada alamprobleemide optimaalseid lahendusi. Näiteks graafis lühima tee leidmiseks mingisse tippu võime alguses leida lühimad teed antud tipu naabertippudesse. Kasutades neid lühimaid teid, saame leida lühima tee antud tippu. Optimaalse alamstruktuuriga probleemidele lahenduse leidmiseks saab kasutada järgnevat kolmesammulist protsessi:

1. jaota probleem väiksemateks alamprobleemideks
2. leia saadud alamprobleemidele optimaalsed lahendused, kasutades seda sama kolmesammulist protseduuri
3. kasuta nendele alamprobleemidele leitud optimaalseid lahendusi esialgsele probleemile optimaalse lahenduse leidmiseks

Alamprobleemidele lahenduse leidmiseks jaotatakse need omakorda alamprobleemideks ja nii edasi, kuni jõuame mingi lihtsalt lahendatava probleemini.

Probleemil on *kattuvad alamprobleemid,* kui samu alamprobleeme kasutatakse mitme erineva suurema probleemi lahendamisel. Näiteks Fibonacci arvude jadas $F_3 = F_1 + F_2$ ja $F_4 = F_2 + F_3$ – mõlema arvutamine sisaldab endas F_2 arvutamist. Arvu F_5 arvutamiseks kasutame nii arvu F_4 kui ka F_3. Leides F_5 väärtust naiivset lähenemist kasutades, arvutaksime F_2 väärtust kaks või isegi enam korda.

Alamprobleemide optimaalsete lahenduse uuesti arvutamise vältimiseks salvestatakse saadud tulemused. Kui meil tuleb lahendada sama alamprobleemi hiljem, siis saame kasutada juba leitud lahendust [url:DP].

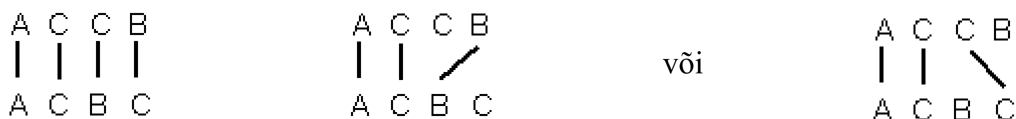
3. Peatükk
Levenshteini kaugus

Levenshteini kauguseks ehk *teisenduskauguseks* kahe sõne vahel nimetatakse vähimat teisendusoperatsioonide arvu, mis tuleb teha selleks, et muuta üks sõne teiseks. Lubatud teisendusoperatsioonideks on ühe tähe lisamine, kustutamine või asendamine teisega, mõningatel juhtudel ka kahe kõrvutioleva tähe vahetamine. Nendele redigeerimisoperatsioonidele on antud kaalud, milleks triviaalsel juhul on 1. Mida suurem on kahe sõne vaheline teisenduskaugus, seda rohkem on need kaks sõne erinevad ning vastupidi – mida väiksem see on, seda sarnasemad sõned on. Sõne teisenduskaugus iseendast on 0.

Näide 3.1. Olgu meil antud kaks sõnet *ACCB* ja *ACBC.* Vaatame võimalikke teisendusi, mida tuleks teha esimese muutmiseks teiseks. Selleks võiks näiteks kustutada sõne *ACCB* kaks viimast tähte ja lisada seejärel lõppu tähed *B* ja *C* või asendada lihtsalt sõne lõpus oleva *C* tähega *B* ning sõne lõpus oleva *B* tähega *C*. Esimesel juhul tehtavate teisendusoperatsioonide arv on 4, teisel juhul 2, mis on ka teisenduskauguseks nende sõnede vahel – vähema arvu operatsioonidega seda teisendust pole võimalik teha.

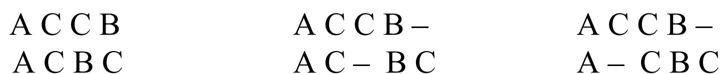
Ühe sõne teiseks muutmisel tehtavad teisendusoperatsioonid ja nende järjekord ei ole üheselt määratud. Näiteks sõne S_1 = *ACCB* muutmiseks sõneks S_2 = *ACBC* võime sõne S_1 lõpus olevad tähed *C* ja *B* asendada vastavalt tähtedega *B* ja *C.* Samuti saame sama tulemuse, kui sõnest S_1 kustutame kolmandal positsioonil oleva tähe *C* ning lisame seejärel lõppu *C* või kui lisame sõne kolmandale positsioonile tähe *B* ning seejärel kustutame sõne lõpus asuva tähe *B.* Teisendusoperatsioone on kõigil neil juhtudel 2. Ei oma tähtsust, kas esimesel juhul asendada enne *B* ja seejärel *D* või vastupidi – tähtsam on pigem teisenduseks vajalik minimaalne teisenduste arv kui tehtavate teisenduste järjekord või tehtavad teisendused ise. Neid teisendusi võime esitada ka ülevaatlikumal kujul [url:TA]:

1. Jälitus (Ingl.k. *trace*)



Joon, mis ühendab esimese sõne S_1 mingil positsioonil i olevat tähte ja teise sõne S_2 mingil positsioonil j olevat tähte, näitab, et teisenduse käigus teisendatakse täht $S_1[i]$ täheks, mis asub sõnes S_2 positsioonil kohal j, ehk täheks $S_2[j]$. Kui $S_1[i]=S_2[j]$, siis täht $S_1[i]$ jäetakse muutmata. Osad tähed sõnest S_1 ei ole ühendatud ühegi sõne S_2 tähega – need positsioonid kujutavad tähti, mis sõnest S_1 teisenduse käigus kustutatakse. Sarnaselt, osad positsioonid sõnes S_2 pole sõne S_1 ühegi positsiooniga joonega ühendatud, need positsioonid kujutavad tähti, mis teisenduse käigus sõnesse S_1 lisatakse [Wag74].

2. Joondamine (Ingl.k *alignment*) – standardne viis teisenduste esitamiseks. Sõned asetatakse kaherealisse maatriksisse. Algne sõne asetatakse esimesse ritta. Sõne, milleks teisendada vaja, asetatakse teise ritta. Maatriksit analüüsitakse veerukaupa – veerg, mis sisaldab $\begin{bmatrix} x \\ - \end{bmatrix}$ viitab tähe x kustutamisele; veerg mis sisaldab $\begin{bmatrix} - \\ y \end{bmatrix}$ viitab tähe y lisamisele; veerg, milles on $\begin{bmatrix} x \\ y \end{bmatrix}$ viitab x asendamisele y-ga. Veerg, mis sisaldaks kahte tühisümbolit „–“, pole lubatud [HL92].

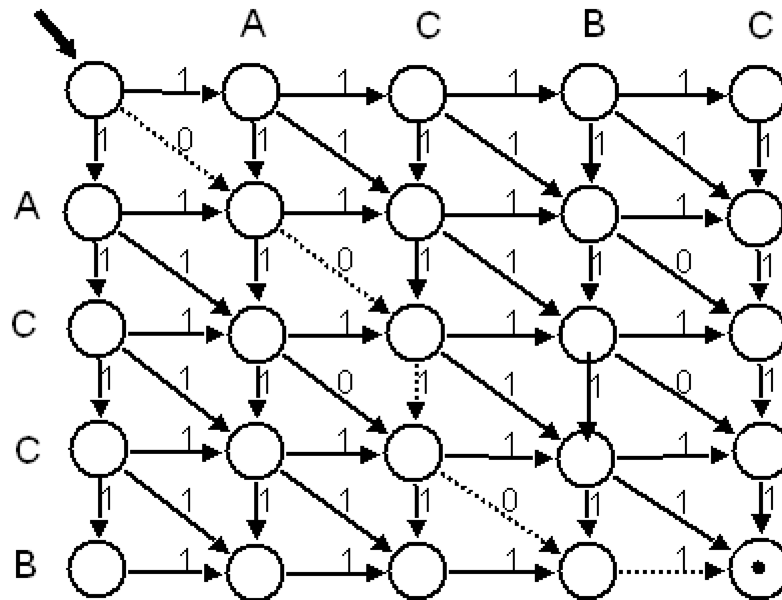


3. Operatsioonide loend

Esimesel juhul:	AC**C**B	$C \rightarrow B$	ACBB
	ACB**B**	$B \rightarrow C$	ACBC
teisel juhul:	AC**C**B	$C \rightarrow \lambda$	ACB
	ACB	$\lambda \rightarrow C$	ACBC
ning kolmandal juhul:	A**C**CB	$C \rightarrow \lambda$	ACB
	ACB	$\lambda \rightarrow C$	ACBC

3.1 Teisenduskauguse arvutamine graafi abil

Teisenduskaugust kahe sõne vahel on võimalik vaadelda kui lühima (vähima kaaluga) tee leidmist graafis. Võtame vaatluse alla eelmises näites olnud sõned S_1 = ACCB ja S_2 = ACBC. Parema ülevaate saamiseks esitame graafi järgmisel kujul (joonis 3.1):



Joonis 3.1. Sõne ACCB võimalikke teisendusi sõneks ACBC kirjeldav graaf.

Joonisel 3.1 on punktiirjoonega märgitud ka üks lühim tee, mis vastab teisendusele, kus alguses kustutatakse sõnest *ACCB* kolmandal positsioonil olev *C* täht ning seejärel lisatakse lõppu *C*.

Graafi läbimisel võib iga liikumist paremale vaadelda kui esialgsesse sõnesse tähe lisamist, liikumist alla võib vaadelda kui esialgsest sõnest tähe kustutamist ning liikumist diagonaalis võib vaadelda kui ühe tähe asendamist teisega või tähe klappimist (kui vastava kaare kaal graafis on võrdne nulliga). Graafi kaartele on antud hinnad, mis vastavad antud teisendusoperatsioonide hindadele. Teed vasakust ülemisest tipust ükskõik millise graafi tipuni võime vaadelda kui esialgse sõne alamsõne $S[1]...S[i]$ teisendamist teise sõne alamsõneks $S[1]...S[j]$. Antud graafi paigutust vaadates tähistab i vaatluse all oleva tipu rida ning j veergu (ridu ja veerge nummerdame 0, 1, ...). Iga teed selles graafis, mis algab vasakust ülemisest tipust ja lõppeb paremas alumises tipus

(joonisel 3.1 märgitud täpiga), võib vaadelda kui mingit teisendusoperatsioonide järjekorda esialgse sõne teisendamiseks teiseks. Antud tee hinda võib vaadata kui hinda selle teisenduste järjekorra kasutamiseks. Kõige väiksema kaaluga tee hinda võime vaadelda kui teisenduskaugust antud sõnede vahel ning antud teed võime vaadelda kui minimaalseks teisenduseks tehtavate teisenduste järjekorda.

Graafis lühima tee leidmiseks võib kasutada:

- jõumeetodit – iga graafi kuuluva tipu *v* korral leitakse lühim tee algtipust tippu *v*. Selle algoritmi ajaline keerukus on $O(|V|+|E|)$, kus $|V|$ tähistab graafi tippude arvu ja $|E|$ tähistab graafi kaarte arvu.
- Dijksta algoritmi – abivahendina kasutatakse eelistusjärjekorda Q selleks, et meeles pidada tippe, mille kõiki eellasi pole (võibolla) veel arvesse võetud. Algoritmi idee põhineb asjaolul, et vähima „jooksva“ kaugusega (*v.d*) tipul $v \in Q$ on kõik eellased juba arvestatud ning järelikult $v.d$ enam ei muutu; sellise tipu v võib hulgast *Q* eemaldada, lisades hulka *Q* tema need järglased, mis veel vaatlusele (ja hulka *Q*) pole võetud. Seejuures korrigeeritakse tipu v järglaste *w* „jooksvat“ kaugust algtipust *a*, kui osutub, et tippu *w* pääseb läbi *v* veelgi lühemat teed (*a...v, w*) pidi kui seda oli senini teadaolev lühim tee (*a...w*). Algoritmi ajaline keerukus sõltub oluliselt sellest, kuidas on korraldatud eelistusjärjekorra „pidamine“. Kahendkuhja kasutamise korral on hinnanguks $O((n + m) \log(n))$, kus $n=|V|$ ja $m=|E|$ [Kih03].

3.2 Teisenduskauguse arvutamine kasutades dünaamilise programmeerimise tehnikat

Teisenduskaugust kahe sõne vahel võib arvutada dünaamilise programmeerimise tehnikat kasutades. Defineerime rekurrentse valemi kahe sõne $A=a_1 a_2 \ldots a_m$ ja $B=b_1 b_2 \ldots b_n$ vahelise teisenduskauguse leidmiseks, tähistame $d_{i,j}=D(a_1 a_2 \ldots a_i, b_1 b_2 .. b_j)$, $0 \leq i \leq m$, $0 \leq j \leq n$, kus $d_{i,j}$ tähistab vähimat teisendusoperatsioonide arvu, mis kulub selleks, et teisendada alamsõne $a_1 a_2 \ldots a_i$ alamsõneks $b_1 b_2 \ldots b_j$.

Minimaalse teisendusoperatsioonide arvu arvutame järgmiselt:

$$d_{i,0}=i, 0 \leq i \leq m \tag{1}$$

$$d_{0,j}=j, 0 \leq j \leq n$$

$$d_{i,j}=min\begin{cases} d_{i-1,j-1}+(if\ a_i=b_j\ \ then\ 0\ \ else\ 1) & viimase\ tähe\ ekvivalents\ või\ asendus \\ d_{i-1,j}+1 & tähe\ lisamine \\ d_{i,j-1}+1 & tähe\ kustutamine \end{cases}$$

$$1 \leq i \leq m, 1 \leq j \leq n.$$

Väärtust $d_{|A|,|B|}=d_{m,n}$ nimetatakse teisenduskauguseks sõnede *A* ja *B* vahel. Ülaltoodud valemit (1) võiks selgitada järgnevalt: alustuseks, $d_{i,0}$ ja $d_{0,j}$ kujutavad teisenduskaugust sõnede pikkusega vastavalt *i* ja *j* ning tühja sõne vahel. Täpsemalt, selleks on vaja teha *i* kustutamist (ja vastavalt *j* lisamist) on selleks. Kahe mittetühja sõne, pikkustega *i* ja *j,* puhul eeldame, et kõik teisenduskaugused neist lühemate sõnede puhul on juba arvutatud ning proovime teisendada sõnet $a_1 a_2 \ldots a_i$ sõneks $b_1 b_2 \ldots b_j$. Võtame vaatluse alla nende sõnede viimased tähed – a_i ja b_j. Kui need tähed on võrdsed, $a_i=b_j$, siis jätkame lihtsalt parimat teed, mille leidsime $a_1 a_2 \ldots a_{i-1}$ teisendamiseks sõneks $b_1 b_2 \ldots b_{j-1}$. Paneme tähele, et sellisel juhul kulub sõne $a_1 a_2 \ldots a_i$ teisendamiseks sõneks $b_1 b_2 \ldots b_i$ sama palju teisendusoperatsioone, kui sõne $a_1 a_2 \ldots a_{i-1}$ teisendamiseks sõneks $b_1 b_2 \ldots b_{j-1}$. Kui aga need tähed ei ole võrdsed, peame vaatlema kolme lubatud teisendusoperatsiooni – me võime kustutada tähe a_i ning kasutada parimat teisendust sõne $a_1 a_2 \ldots a_{i-1}$ muutmiseks sõneks $b_1 b_2 \ldots b_j$, võime lisada tähe b_j sõne $a_1 a_2 \ldots a_i$ lõppu ning kasutada parimat teisendust sõne $a_1 a_2 \ldots a_i$ teisendamiseks sõneks $b_1 b_2 \ldots b_{j-1}$ või asendada

tähe a_i tähega b_j ning kasutada parimat teisendust $a_1 a_2 ... a_{i-1}$ muutmiseks sõneks $b_1 b_2 ... b_{j-1}$. Kõigil neil juhtudel tehtud teisenduse hind on 1 pluss ülejäänud teisenduste hind (need on meil juba arvutatud) [Nav01].

Näide 3.2. Olgu antud sõned *kalur* ja *kallan.* Leiame selle algoritmi järgi teisenduskauguse nende sõnede vahel. Joonisel 3.2 on näha vastav dünaamilise programmeerimise tabel.

		k	a	l	l	a	n
	0	1	2	3	4	5	6
k	1	**0**	1	2	3	4	5
a	2	1	**0**	1	2	3	4
l	3	2	1	**0**	1	2	3
u	4	3	2	1	**1**	**2**	3
r	5	4	3	2	2	2	**3**

Joonis 3.2. Dünaamilise programmeerimise algoritmi abil arvutatud teisenduskaugus sõnede *kalur* ja *kallan* vahel. Paksu kirjaga märgitud võimalik teisendus.

Algoritm 3.1. Teisendukauguse *D*(*A*, *B*) leidmine kasutades dünaamilise programmeerimise tehnikat.

Sisend: Sõned $A=a_1 a_2 .. a_m$ ning $B=b_1 b_2 ... b_n$.

Väljund: Väärtus $d_{m,n}$ maatriksis ($d_{i,j}$), $0 \leq i \leq m$, $0 \leq j \leq n$.

Meetod:

1. **for** i = 0 **to** m **do** $d_{i,0}=i$
2. **for** j = 0 **to** n **do** $d_{0,j}=j$
3. **for** j = 1 **to** n **do**
4. **for** i = 1 **to** m **do**
5. $d_{i,j}$ = **min** (
6. $d_{i-1,j-1}+(if\ a_i=b_j\ \ then\ \ 0\ \ else\ \ 1)$,
7. $d_{i-1,j}+1$,
8. $d_{i,j-1}+1$)
9. **return** $d_{i,j}$

Selle algoritmi ajaline keerukus on $O(|m||n|)$, mäluvajadus on $O(|m| \times |n|)$.

Kasutades dünaamilise programmeerimise tabelit, saame väga lihtsalt taastada antud sõnede teisendamiseks vajalike minimaalsete teisenduste järjekorra. Selleks tuleb lihtsalt arvutusprotsessi pöörata – peame leidma tee tabeli väljalt $d_{m,n}$ väljani $d_{0,0}$, et näha, mis teed pidi minimiseerimine $d_{i,j}$ arvutamisel kulges. Selleks genereerime hulgad *pred*[*i, j*] – tähistamaks millisest ruudust tuldi ruutu $d_{i,j}$.

Hulga *pred*[*i, j*] genereerime järgmise algoritmi järgi, hulka lisame elemendi:

- $(i-1, j-1)$, *kui* $d_{i,j}=d_{i-1,j-1}+(if\ a_i=b_j\ then\ 0\ else\ 1)$
- $(i-1, j)$, *kui* $d_{i,j}=d_{i-1,j}+1$
- $(i, j-1)$, *kui* $d_{i,j}=d_{i,j-1}+1$

$0 \leq i \leq m\ , 0 \leq j \leq n$. Teisenduse järjekorra taastamiseks tuleb liikuda mööda hulkasid *pred*[*i, j*] kuni väljani $d_{0,0}$ [url:TA]. Ilmestamiseks toome joonise sõnede *kallur* ja *kallan* jaoks arvutatud dünaamilise programmeerimise tabeli abil taastatud võimalike lühimate teisendusteedega (joonis 3.3).

		k	a	l	l	a	n
	0	1	2	3	4	5	6
k	1	**0**	1	2	3	4	5
a	2	1	**0**	1	2	3	4
l	3	2	1	**0**	1	2	3
u	4	3	2	1	**1**	2	3
r	5	4	3	2	2	2	3

Joonis 3.3. Teisenduste taastamine dünaamilise programmeerimise algoritmi põhjal arvutatud tabelist.

Teisenduste taastamist dünaamilise programmeerimise tabelist alustame paremast alumisest nurgast. Tabelis saame liikuda vasakule, kui selles olev arv on ühe võrra väiksem, kui arv, mis asub väljal, millel tabelis hetkel oleme. Sama kehtib antud välja kohal asuva välja kohta – sinna saame samuti liikuda vaid juhul, kui sellel olev arv on väiksem kui hetkel vaatluse all oleval väljal asuv arv. Diagonaalis vasakule üles saame liikuda, kui vastavat veergu ja rida tähistavad samad tähed või kui need tähed on erinevad ning vasakul üleval asuval väljal olev arv on ühe võrra väiksem kui arv väljal, millel oleme hetkel.

4.Peatükk
Üldistatud teisenduskaugus

Üldistatud teisenduskauguse leidmist kahe sõne vahel võiks vaadelda kui tavalise teisenduskauguse leidmist, millele on juurde defineeritud veel hulk lisaoperatsioone. Lubatavateks operatsioonideks oleks ühe tähe lisamine, kustutamine, tähe asendamine teisega või mitme kõrvutiasetseva tähe kustutamine, lisamine või asendamine ühe või mitme tähega. Tavalistele redigeerimisoperatsioonidele kehtivad vaikimisi lisamisele, kustutamisele ja asendamisele antud üldised kaalud, samas on võimalik nii laiendatud kui ka tavalistele teisendusoperatsioonidele defineerida vaikimisi kaaludest erinevad kaalud. Saadud teisendukaugust tuleks vaadata kui minimaalse teisenduse hinda, mitte kui vähimat teisenduste arvu, sest võib tekkida olukord, kus vähima hinnaga teisenduseks tuleb teha rohkem teisendusoperatsioone kui seda tuleks teha näiteks tavalise teisenduskauguse puhul.

Näide 4.1. Olgu antud sõned *laul* ja *laulmine.* Defineerime tavalistele teisendusoperatsioonidele lisaks ka operatsiooni:

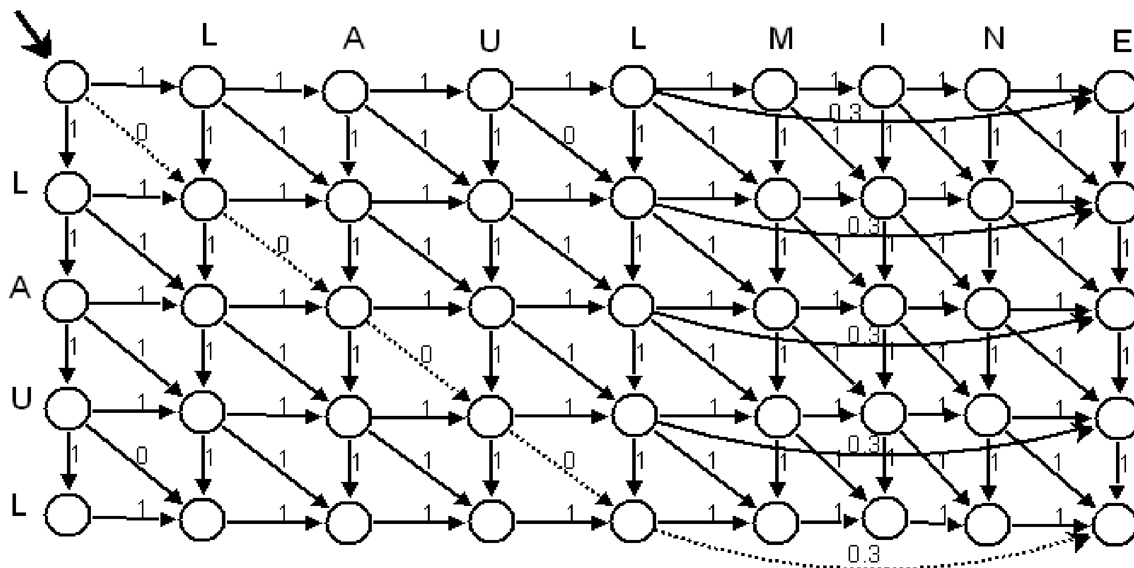
$\lambda \rightarrow mine$, millele anname hinnaks näiteks 0.3.

Sellisel juhul on üldistatud teisenduskaugus nende sõnede vahel 0.3 – sõnele *laul* lihtsalt lisame lõppu sõne *mine.* Tavaline teisenduskaugus nende sõnede vahel oleks võrdne neljaga – sõnele *laul* tuleks lisada lõppu tähed *m*, *i*, *n* ja *e*.

4.1 Üldistatud teisenduskauguse leidmine graafide abil

Üldistatud teisenduskauguse leidmise ülesannet on võimalik esitada kui graafis lühima tee leidmise probleemi.

Näide 4.2. Esitame sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse leidmise probleemi graafi kujul (joonis 4.1):

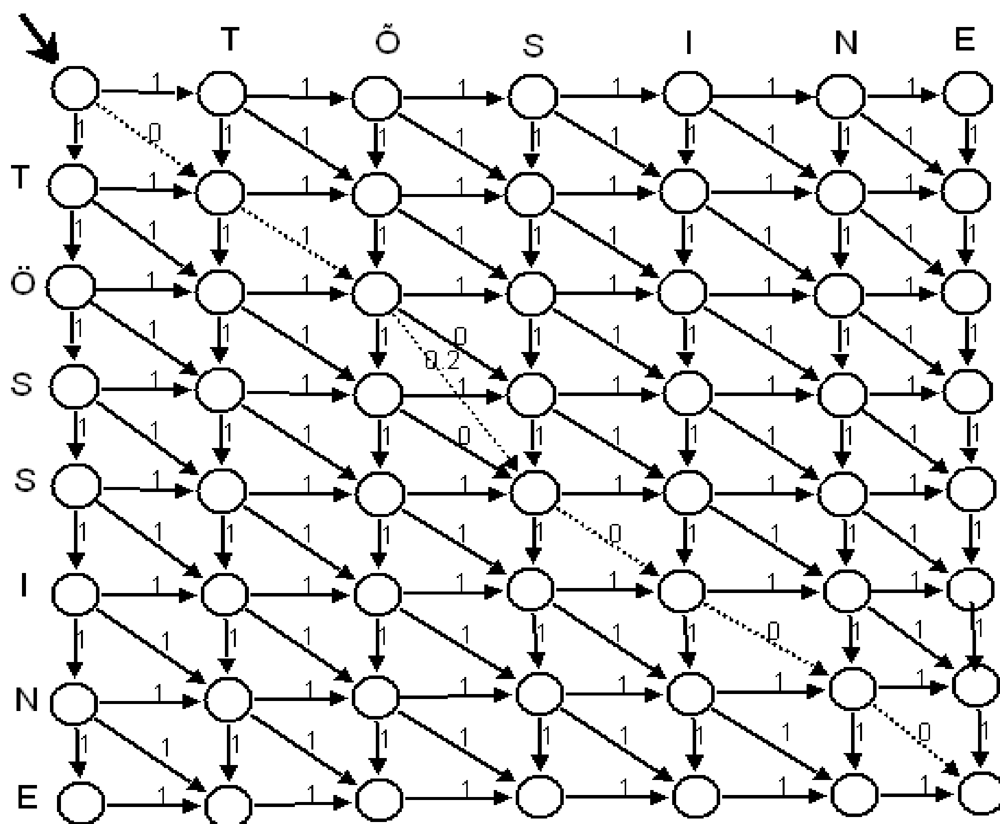


Joonis 4.1. Sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse leidmine, kui tavalistele teisendusoperatsioonidele lisaks on defineeritud lõpu *mine* lisamine, kaaluga 0.3.

Nagu ka tavalise teisenduskauguse graafi puhul, on toodud graafis paremale liikumine võrdne tähe lisamisega, liikumine alla – tähe kustutamisega ning liikumine diagonaalis – tähe asendamisega. Erinevalt tavalisest teisenduskaugusest on selles graafis n.ö otseteed, mis kujutavad lisaredigeerimisoperatsioone. Selles graafis lühimat teed leides võime kasutada nii neid servi, mis viitavad tavalistele redigeerimisoperatsioonidele kui ka neid, mis viitavad lisatud redigeerimisoperatsioonidele. Joonisel 4.1 kujutatud graafis on punktiiriga märgitud ka väikseima kaaluga tee, mille hind on samaväärne üldise teisenduskaugusega nende sõnede vahel.

Näide 4.3. Tahame leida sõnede *tössine* ja *tõsine* ning *tössine* ja *tassike* vahelisi üldistatud teisenduskaugusi. Olgu antud lisaks teisendus $ss \rightarrow s$ kaaluga 0.2.

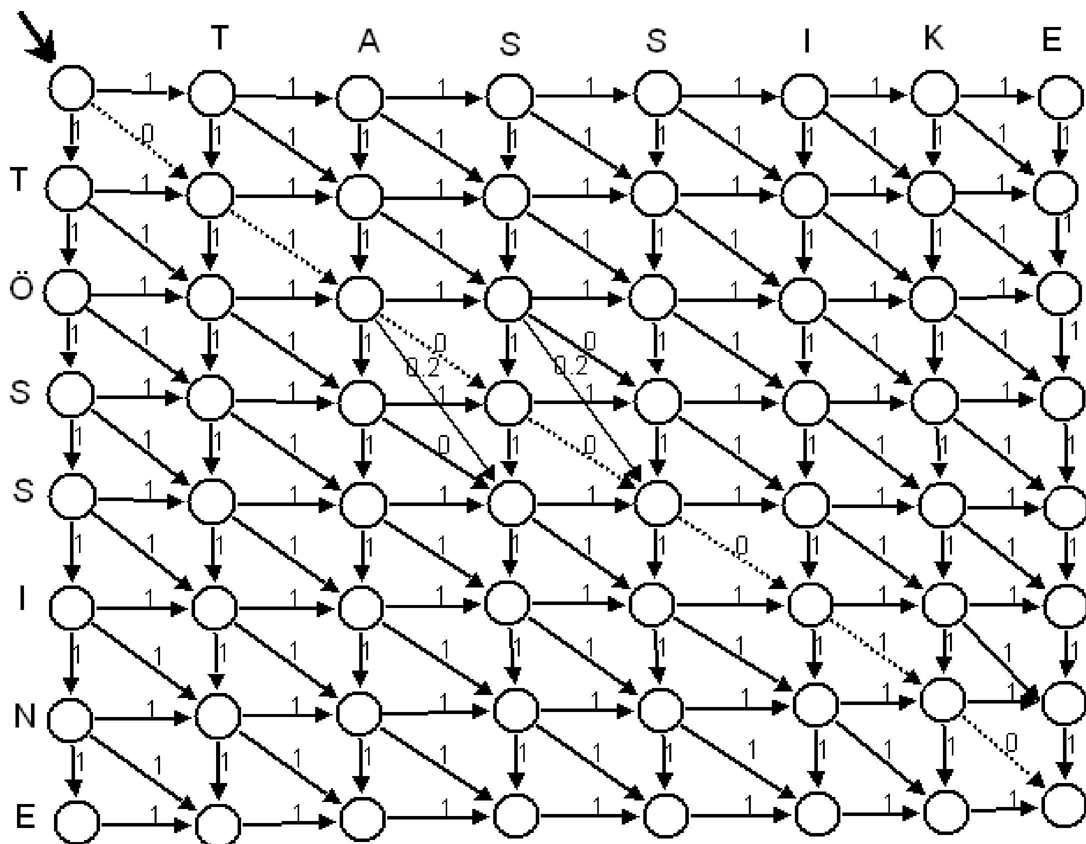
Esitame antud üldistatud teisenduskauguse leidmise probleemid graafi kujul (joonised 4.2 ja 4.3). Esimesel juhul oleks üldistatud teisenduskauguse graaf järgmine:



Joonis 4.2. Üldistatud teisenduskauguse leidmine sõnede *tössine* ja *tõsine* vahel.

Üldistatud teisenduskaugus sõnede *tössine* ja *tõsine* vahel on 1.2. Ülaloleval joonisel 4.2 on punktiirjoonega märgitud ka antud teisenduskaugusele vastav lühim tee.

Leiame üldistatud teisenduskauguse ka sõnede *tössine* ja *tassike* vahel:



Joonis 4.3. Üldistatud teisenduskauguse leidmine sõnede *tössine* ning *tassike* vahel.

Sõnede *tössine* ning *tassike* vaheliseks üldistatud teisenduskauguseks tuli 2. Kasutades teisendust $ss \rightarrow s$, saime sõned *tössine* ning *tõsine* palju lähedasemaks kui *tössine* ja *tassike.* Defineerides juurde veel näiteks teisenduse $ö \rightarrow õ$, mille kaal oleks väiksem kui 1, saaksime sõnet *tössine* sõnele *tõsine* veelgi lähendada.

4.2 Üldistatud teisenduskauguse leidmine kasutades dünaamilise programmeerimise tehnikat

Ka üldistatud teisenduskaugust saab leida dünaamilise programmeerimise tehnikat kasutades. Olgu antud sõned $A=a_1 a_2 \ldots a_m$ ja $B=b_1 b_2 \ldots b_n$. Defineerime rekurrentse valemi üldistatud teisenduskauguse $d'_{i,j}=D'(a_1 a_2 \ldots a_i, b_1 b_2 \ldots b_j), 0\le i\le m, 0\le j\le n$ leidmiseks:

$$d'_{0,0}=0$$

$$d'_{i,0}=min\begin{cases} i \\ d'_{k-1,0}+w, & kui\ leidub\ teisendus\ \ a_k \ldots a_i \rightarrow \lambda\ \ kaaluga\ w \end{cases},\quad 1\le i\le m$$

$$d'_{0,j}=min\begin{cases} j \\ d'_{0,l-1}+w, & kui\ leidub\ teisendus\ \ \lambda \rightarrow b_l \ldots b_j\ \ kaaluga\ w \end{cases},\quad 1\le j\le n$$

$$d'_{i,j}=min\begin{cases} d'_{i-1,j-1}+(if\ a_i=b_j\ then\ 0\ else\ 1) \\ d'_{i-1,j}+1 \\ d'_{i,j-1}+1 \\ d'_{k-1,l-1}+w, & kui\ leidub\ asendus:\ \ a_k \ldots a_i \rightarrow b_l \ldots b_j\ \ kaaluga\ w \\ d'_{k-1,j}\ \ +w, & kui\ leidub\ teisendus:\ \ a_k \ldots a_i \rightarrow \lambda\ \ kaaluga\ w \\ d'_{i,l-1}\ \ +w, & kui\ leidub\ teisendus:\ \ \lambda \rightarrow b_l \ldots b_j\ \ kaaluga\ w \end{cases}$$

$$1\le i\le m,\ \ 1\le j\le n,\ \ 1\le k\le i,\ \ 1\le l\le j,\ \ i\ne j\ne 0$$

Väärtust $d'_{|A||B|}$ võime vaadelda kui üldistatud teisenduskaugust sõnede *A* ja *B* vahel.

Algoritmi selgituseks: väärtusi $d'_{i,0}$ ja $d'_{j,0}$ võib vaadelda kui üldistatud teisenduskaugusi sõnede (vastavalt pikkusega *i* ja *j*) ning tühja sõne vahel. Tähti sõnesse võib lisada ükshaaval või grupiviisi – kui selline teisendus on defineeritud. Vaatleme kahte mittetühja sõnet $A=a_1 a_2 \ldots a_i$ ja $B=b_1 b_2 \ldots b_j$, pikkustega *i* ja *j*. Eeldame, et nendest lühemate sõnede puhul on kõik üldistatud teisenduskaugused juba leitud. Võtame vaatluse alla alguses tavalised teisendusoperatsioonid – tähe lisamine, kustutamine ja asendamine, ning vaatleme sõnede viimaseid tähti a_i ja b_j. Kui $a_i=b_j$, siis sõne *A* teisendamiseks sõneks *B* võib kasutada parimat teisendust, mis oli leitud alamsõne $a_1 a_2 \ldots a_{i-1}$ teisendamiseks alamsõneks $b_1 b_2 \ldots b_{j-1}$ ning sinna lõppu lihtsalt lisada täht a_i. Kui aga $a_i\ne b_j$, siis võib asendada tähe a_i tähega b_j ning kasutada leitud parimat teisendust $a_1 a_2 \ldots a_{i-1} \rightarrow b_1 b_2 \ldots b_{j-1}$, kustutada tähe a_i ning kasutada parimat teisendust

$a_1 a_2 \ldots a_{i-1} \rightarrow b_1 b_2 .. b_j$ või lisada tähe b_j sõne $a_1 a_2 \ldots a_i$ lõppu ning kasutada parimat teisendust $a_1 a_2 \ldots a_i \rightarrow b_1 b_2 \ldots b_{j-1}$ [Nav01]. Kui aga on defineeritud teisendus $a_k \ldots a_i \rightarrow b_l \ldots b_j$ $, 1 \leq k \leq i, 1 \leq l \leq j,$ siis võime asendada sõne *A* lõpust *i – k* + 1 tähte sõne *B* alamsõnega $b_l \ldots b_j$ ning edasi kasutada parimat teisendust $a_1 a_2 \ldots a_{k-1} \rightarrow b_1 b_2 \ldots b_{l-1}$. Kui on defineeritud teisendus $a_k \ldots a_i \rightarrow \lambda$ $, 1 \leq k \leq i,$ siis võib kustutada sõne *A* lõpust *i – k* + 1 tähte ning kasutada teisenduseks parimat teisendust $a_1 a_2 \ldots a_{k-1} \rightarrow b_1 b_2 \ldots b_j$. Kui on defineeritud teisendus $\lambda \rightarrow b_l \ldots b_j$ $, 1 \leq l \leq j,$ siis võib sõne *B* positsioonidel *l – j* asetsevad tähed lisada sõne *A* lõppu ning kasutada teisendust $a_1 a_2 \ldots a_i \rightarrow b_1 b_2 \ldots b_{l-1}$.

Üldistatud teisenduskauguse tabeli arvutamisel sõnede *A* ja *B* vahel peame iga välja (*i*, *j*) täitmisel arvesse võtma kõiki teisendusi $\alpha \rightarrow \beta$, mille puhul $\alpha = A[i - |\alpha| + 1, i]$ ning $\beta = B[j - |\beta| + 1]$, s.o teisendusi, mille vasak pool on võrdne sõne *A* alamsõnega, mis lõppeb kohal *i* ning parem pool on võrdne sõne B alamsõnega, mis lõppeb kohal *j*. Teisendustes võib α või β asemel olla ka tühi sõne, sellisel juhul vaatleme seda teisendust kui lisamis- või kustutamisoperatsiooni.

		b_1	b_2	...	b_{l-1}	b_l	b_{l+1}	...	b_{j-1}	b_j	...	b_n
				...				...			...	
a_1				...				...			...	
a_2				...				...			...	
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱			⋱	⋮
a_{k-1}				...				...			...	
a_k				...				...			...	
a_{k+1}				...				...			...	
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱			⋱	⋮
a_{i-1}				...				...			...	
a_i				...				...			...	
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱			⋱	⋮
a_m				...				...			...	

Joonis 4.4. Üldistatud teisenduskauguse tabeli välja (*i*, *j*) täitmine dünaamilise programmeerimise meetodi abil.

Joonisel 4.4 on kujutatud tabeli välja (*i, j*) täitmist. Hetkel kasutatavateks teisendusteks on sõnest *A* *i*-nda tähe kustutamine – $a_i \rightarrow \lambda$, sõne *B* *j*-nda tähe lisamine – $\lambda \rightarrow b_j$, sõne *A* *i*-nda tähe asendamine sõne *B* *j*-nda tähega ning lisateisendusoperatsioonid – $a_k \dots a_i \rightarrow b_l \dots b_j$, $a_{k+1} .. a_i \rightarrow \lambda$, $a_i \rightarrow b_{l+1} \dots b_j$, ning $\lambda \rightarrow b_1 \dots b_j$. Olgu üldistele teisendustele antud kaal w_1 ning lisateisendusoperatsioonidele kaalud vastavalt w_2, w_3, w_4 ja w_5. Väljale (*i, j*) kirjutame vähima arvu hulgast $\{ (i-1, j)+w_1, (i-1, j-1)+w_1, (i, j-1)+w_1, (k-1, l-1)+w_2, (k, j)+w_3, (i-1, l)+w_4, (i, 1)+w_5 \}$.

<u>Näide 4.4</u>. Leiame sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse, kui peale tavaliste teisendusoperatsioonide on veel defineeritud teisendus $\lambda \rightarrow mine$ kaaluga 0.3. Esitame selle leidmiseks arvutatud dünaamilise programmeerimise tabeli joonisel 4.5.

		l	a	u	l	m	i	n	e
	0	1	2	3	4	5	6	7	4,3
l	1	0	1	2	3	4	5	6	3,3
a	2	1	0	1	2	3	4	5	2,3
u	3	2	1	0	1	2	3	4	1,3
l	4	3	2	1	0	1	2	3	0,3

<u>Joonis 4.5</u>. Sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse leidmine.

Näide 4.5. Leiame üldistatud teisenduskauguse sõnede *tössine* ja *tõsine* ning *tössine* ning *tassike* vahel kasutades dünaamilise programmeerimise meetodit. Defineerime lisateisendusoperatsiooni $ss \rightarrow s$ kaaluga 0.2. Vastavad dünaamilise programmeerimise meetodi abil arvutatud üldistatud teisenduskauguse tabelid on kujutatud joonistel 4.6 ja 4.7:

		t	õ	s	i	n	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
ö	2	1	1	2	3	4	5
s	3	2	2	1	2	3	4
s	4	3	3	1,2	2	3	4
i	5	4	4	2,2	1,2	2,2	3,2
n	6	5	5	3,2	2,2	1,2	2,2
e	7	6	6	4,2	3,2	2,2	1,2

Joonis 4.6. Sõnede *tössine* ja *tõsine* vahelise üldistatud teisenduskauguse leidmine kasutades dünaamilise programmeerimise meetodit.

		t	a	s	s	i	k	e
	0	1	2	3	4	5	6	7
t	1	0	1	2	3	4	5	6
ö	2	1	1	2	3	4	5	6
s	3	2	2	1	2	3	4	5
s	4	3	3	1,2	1	2	3	4
i	5	4	4	2,2	2	1	2	3
n	6	5	5	3,2	3	2	2	3
e	7	6	6	4,2	4	3	3	2

Joonis 4.7. Sõnede *tössine* ja *tassike* vahelise üldistatud teisenduskauguse leidmine.

Joonistel 4.5, 4.6 ja 4.7 on nooltega näidatud kasutatud lisateisendused ja nende hinnad.

Algoritm 4.1. Üldistatud teisenduskauguse $D'(A, B)$ leidmine kasutades dünaamilise programmeerimise tehnikat.

Sisend: Sõned $A=a_1a_2..a_m$, $B=b_1b_2...b_n$ ning lisateisendusoperatsioonide hulk $T=\{t_1, t_2, ..., t_s\}$.

Väljund: Väärtus $d'_{m,n}$ maatriksis ($d'_{i,j}$), $0 \le i \le m$, $0 \le j \le n$.

Meetod:

1. $d'_{0,0}=0$

// täidame esimese veeru

2. **for** i = 1 **to** m **do**

3. $d'_{i,0}$ = **min** (

4. i,

5. $\{d'_{i-|t_r|,0}+w_{t_r} \mid t_r=a_k...a_i \to \lambda, \ 1 \le k \le i, \ 1 \le r \le s\}$) *// vaatame läbi teisenduste hulga*

// ülejäänud tabeli täitmine

6. **for** j = 1 **to** n **do**

// täidame esimese rea

7. $d'_{0,j}$ = **min** (j, $\{d'_{0,j-|t_r|}+w_{t_r} \mid t_r=\lambda \to b_l...b_j, \ 1 \le l \le i, \ 1 \le r \le s\}$)

// töötleme ülejäänud read

8. **for** i = 1 **to** m **do**

9. $d'_{i,j}$ = **min** $\{ d'_{i-|\alpha|, j-|\beta|}+w_{\alpha \to \beta} \mid \alpha \to \beta \in \{\textit{teisendused , mida saab hetkel kasutada}\} \}$

10. **return** $d'_{i,j}$

Algoritmi 4.1 puhul vaadatakse iga välja täitmisel terve teisenduste järjend iga kord algusest lõpuni läbi, listi läbivaatamine on aga ajalise keerukusega $O(n)$, millele lisandub veel iga teisenduse puhul tehtud alamsõnega võrdlemiste arv. Probleemi ei kergendaks palju isegi see, kui lisamised, kustutamised ja asendused jagada kolme eraldi hulka – see vähendaks läbivaatamiste arvu ainult esimese veeru ja esimese rea täitmisel.

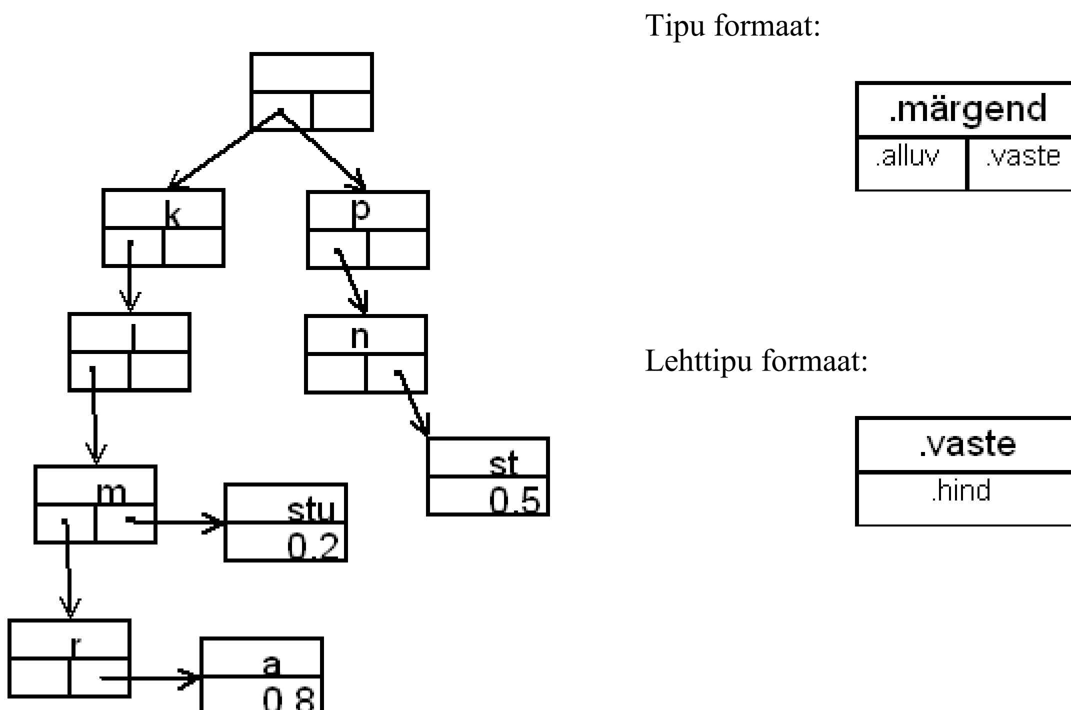
Lahenduseks oleks teisenduste hulga organiseerimine prefiksipuu abil.

4.2.1 Teisenduste hulga organiseerimine prefiksipuu abil

Autori semestritöö põhitulemuseks oli teisenduste hulga viimine kompaktsemale kujule ning sobivate teisenduste otsimise kiirendamine. Jaotame teisendused kolme hulka – asendused, kustutamised ja lisamised ning teeme igaühele neist omaette prefiksipuu. Kuna prefiksipuud oleks raske organiseerida nii teisenduste vasaku kui ka parema poole järgi, siis organiseerime teisenduse $t_1 \rightarrow t_2$ prefiksipuusse teisenduse vasaku poole – t_1 järgi, lisamisoperatsioonid t_2 järgi, kuna nende puhul on $t_1 = \lambda$.

Prefiksipuu kujutamiseks arvutis on mõttekam kaarte märgendites olev informatsioon salvestada tippudesse ning kaari kujutada kui viitasid tippude vahel. Igasse prefiksipuu vahetippu, mis asub juurtipust kaugusel k, salvestame märgendi $t_1[k]$. Lehttipu sisse salvestame sõnelõppu tähistava $\$$ asemel aga hoopis selles kohas lõppeva teisenduse parema poole ning sellele teisendusele vastava hinna. Lisamise ja kustutamise prefiksipuude korral salvestame lehe sisse ainult teisenduse hinna.

Näide 4.6. Olgu antud teisendused $klm \rightarrow stu$ kaaluga 0.2, $klmr \rightarrow a$ kaaluga 0.8 ning $pn \rightarrow st$ kaaluga 0.5. Neid teisendusi sisaldavat puud võiks kujutada järgmiselt (joonis 4.8):



Joonis 4.8. Teisenduste kujutamine prefiksipuu andmestruktuuris.

Joonisel 4.8 kujutatud prefiksipuu puhul hoiame sisetipu väljal *.märgend* vastava kaare märgendit, viidavälja *.alluv* kaudu on tipp seotud oma vahetute alluvatega ning väljal *.vaste* hoiame viita selle koha peal lõppeva sõne vastele (teisenduse paremale poolele). Lehttipu puhul hoiame väljal *.vaste* teisenduse paremat poolt ning väljal *.hind* antud teisenduse hinda.

Teisenduste hulga lisamiseks prefiksipuusse kasutame algoritmi 4.2.

Algoritm 4.2. Teisenduste hulga lisamine prefiksipuu andmestruktuuridesse

Sisend: Teisenduste hulk $T=\{t_1, t_2, \ldots, t_s\}$

Väljund: Lisamise, kustutamise ja asendamise prefiksipuud: $T_{lisamised}$, $T_{kustutamised}$ ning $T_{asendamised}$.

Meetod:

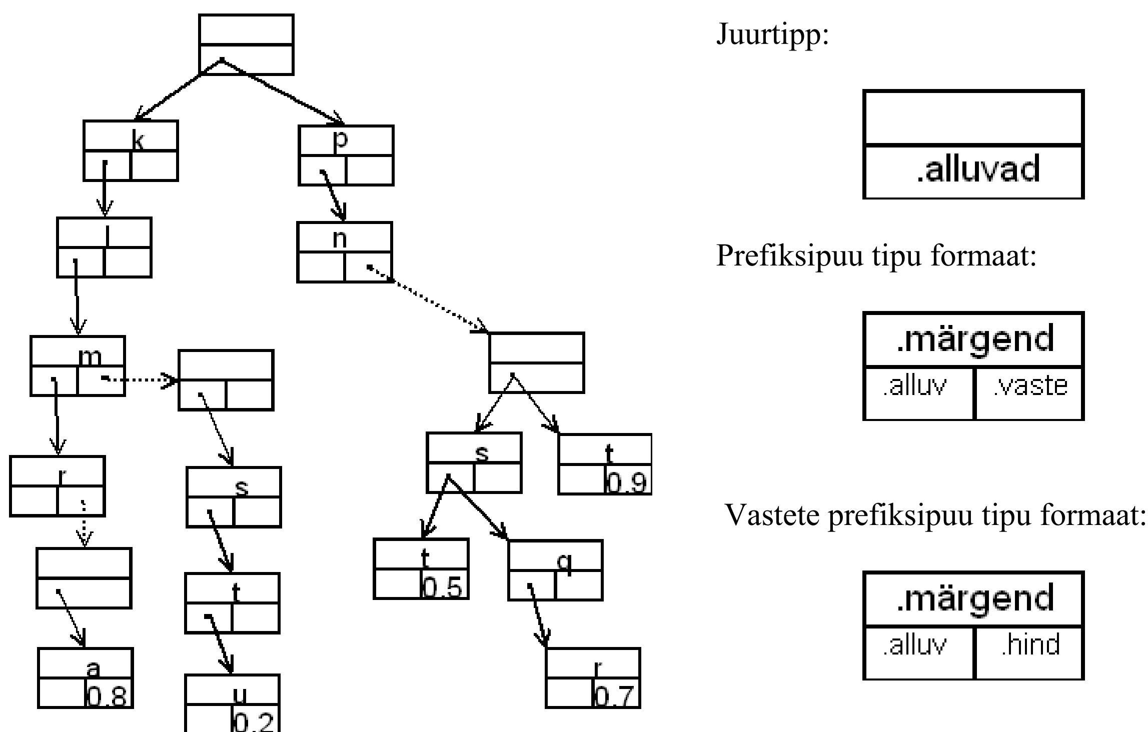
1. **for** i = 1 **to** $|T|$ **do**
2. **if** t_i = lisamisoperatsioon **then** *lisaPrefiksipuusse*(t_i, $T_{lisamised}$)
3. **else if** t_i = kustutamisoperatsioon **then** *lisaPrefiksipuusse*(t_i, $T_{kustutamised}$)
4. **else** *lisaPrefiksipuusse*(t_i, $T_{asendamised}$) *// tegemist on asendusoperatsiooniga*

//meetod teisenduse $\alpha \to \beta$ lisamiseks prefiksipuusse, kui $\alpha=\lambda$ või $\beta=\lambda$, siis organiseerime
//sõne prefiksipuusse vastavalt ainult β või α järgi ning lehttippu salvestame vaid teisenduse
//kaalu.

5. *lisaPrefiksipuusse*($\alpha \to \beta$, prefiksipuu)
6. **while**(*Prefiksipuus leidub prefiks* $\alpha[1]...\alpha[k]$)
7. *Liigu prefiksipuus allapoole*
8. *Lisa prefiksipuusse teisenduse lõpp* $\alpha[k+1]...\alpha[|\alpha|]$
9. *Lisa prefiksipuusse teisenduse parem pool* β *ning teisenduse kaal* $w_{\alpha \to \beta}$

Kui leiduvad teisendused $t_1 \to t_2$ ning $t_1 \to t_3$, st. sõnet t_1 on võimalik asendada nii sõnega t_2 kui ka sõnega t_3 ning seega on asenduste prefiksipuus sõne t_1 lõppemise koha peal kaks lehttippu. Sellisel juhul võib need lehed organiseerida näiteks ahelana. Kui selliseid asendusi on palju ning asenduste parematel pooltel on ühiseid prefikseid, siis oleks mõistlik ka asenduste paremad pooled organiseerida prefiksipuudesse.

Näide 4.7. Olgu antud teisendused $klm \to stu$ kaaluga 0.2, $klmr \to a$ kaaluga 0.8, $pn \to st$ kaaluga 0.5, $pn \to sqr$ kaaluga 0.7 ning $pn \to t$ kaaluga 0.9. Joonis 4.9 illustreerib antud teisenduste hulga kujutamist, kui ka teisenduste paremad pooled on organiseeritud prefiksipuudena.



Joonis 4.9. Teisenduste puu kujutamine, lisades teisenduse vasted omaette prefiksipuudesse.

Joonisel 4.9 on juurtipul ainult üks väli, kus hoiame viitasid vahetutele alluvatele. Teisenduse vasaku poole tipu puhul hoiame väljal *.märgend* vastava tipu märgendit, väljal *.alluv* viitasid alluvatele ning väljal *.vaste* viita vastete prefiksipuule. Vastete prefiksipuu puhul on juurtipp sama formaadiga kui teisenduste vasakute poolte prefiksipuus. Ülejäänud tipud vastete puus on järgmise formaadiga – väljal *.alluv* hoiame viitasid alluvatele ning väljal *.hind* vastaval kohal lõppeva teisenduse hinda. Joonise selguse huvides on viidad vastete prefiksipuudele märgitud punktiirjoontega.

Kui leiduvad teisendused $t_1 \rightarrow t_2$ kaaluga w_1 ning $t_1 \rightarrow t_2$ kaaluga w_2, kusjuures $w_1 \leq w_2$, siis prefiksipuusse lisame neist vaid esimese – kaalude võrdsuse puhul oleks tegemist lihtsalt täpselt sama teisendusega. Juhul kui $w_1 < w_2$, ei kasutaks me teisendust kaaluga w_2 kunagi – teisenduseks valime alati vähima kaaluga võimaliku teisenduse.

Järgnevas algoritmis 4.3 kirjeldame dünaamilise programmeerimise tabeli täitmise protseduuri, kui teisenduste hulk on organiseeritud prefiksipuudena.

Algoritm 4.3. Üldistatud teisenduskauguse *D*'(*A*, *B*) leidmine kui teisenduste hulk on organiseeritud prefiksipuu andmestruktuuri abil.

Sisend: Sõned $A=a_1 a_2 .. a_m$, $B=b_1 b_2 ... b_n$ ning lisateisendusoperatsioonide hulk *T*.

Väljund: Väärtus $d'_{m,n}$ maatriksis ($d'_{i,j}$), $0 \le i \le m$, $0 \le j \le n$.

Meetod: Teisenduste prefiksipuusse organiseerimiseks kasutame algoritmi 4.2

1. *Organiseeri teisenduste hulk T prefiksipuuse.*

2. $d'_{0,0}=0$

// *täidame esimese veeru*

3. **for** i = 1 **to** m **do**

4. **while**(*kustutamiste prefiksipuus leidub teisendusi* $a_i ... a_k \to \lambda$, $i \le k \le m$)

5. $d'_{k,0}$ = **min**($d'_{k,0}$, $d'_{i-1,0} + w_{a_i ... a_k \to \lambda}$)

6. $d'_{i,0}$ = **min**($d'_{i-1,0} + w_{kustutamine}$, $d'_{i,0}$)

7. **for** j = 1 **to** n **do**

// *täidame esimese rea*

8. **while**(*lisamise prefiksipuus leidub teisendusi* $\lambda \to b_j ... b_l$, $j \le l \le n$)

9. $d'_{0,l}$ = **min**($d'_{0,l}$, $d'_{0,j-1} + w_{\lambda \to b_j ... b_l}$)

10. $d'_{0,j}$ = **min** ($d'_{0,j-1} + w_{lisamine}$, $d'_{0,j}$)

//*täidame ülejäänud read*

11. **for** i = 1 **to** m **do**

12. **while**(*kustutamiste prefiksipuus leidub teisendusi* $a_i ... a_k \to \lambda$, $i \le k \le m$)

13. $d'_{k,j}$ = **min**($d'_{k,j}$, $d'_{i-1,j} + w_{a_i ... a_k \to \lambda}$)

14. **while**(*lisamise prefiksipuus leidub teisendusi* $\lambda \to b_j ... b_l$, $j \le l \le n$)

15. $d'_{i,l}$ = **min**($d'_{i,l}$, $d'_{i,j-1} + w_{\lambda \to b_j ... b_l}$)

16. **while**(*asenduste prefiksipuus leidub teisendusi* $a_i ... a_k \to b_j ... b_l$, $i \le k \le m$, $j \le l \le n$)

17. $d'_{k,l}$ = **min**($d'_{k,l}$, $d'_{i-1,j-1} + w_{a_i .. a_k \to b_j ... b_l}$)

18. $d'_{i,j}$ = **min**{ $d'_{i-1,j} + w_{kustutamine}$,

19. $d'_{i,j-1} + w_{lisamine}$,

20. $d'_{i-1,j-1} + w_{asendamine}$,

21. $d'_{i,j}$ }

22. **return** $d'_{i,j}$

Algoritmis 4.3 kirjeldatud meetodis enne tabeli iga välja (*i, j*) täitmist vaatame, kas leidub lisakustutamisoperatsioone $a_i \dots a_k$, $i \leq k \leq m$. Iga sellise teisenduse puhul lisame tabelisse kohale (*k, j*) väärtuse $min\{(k, j), (i-1, j)+w_{a_i \dots a_k \to \lambda}\}$. Sama teeme läbi ka lisaasendus- ning lisamisoperatsioonidega. Esimese puhul, kui saame kasutada mõnda juurdedefineeritud lisamisoperatsiooni sõne $a_i \dots a_k$ teisendamiseks sõneks $b_j \dots b_l$, $i \leq k \leq m$, $j \leq l \leq n$, siis väljale (*k, l*) kirjutame väärtuse $min\{(k, l), (i-1, j-1)+w_{a_i \dots a_k \to b_j \dots b_l}\}$. Ning teisel juhul – kui leidub lisateisendus $\lambda \to b_j \dots b_l$, $j \leq l \leq n$, siis väljale (*i, l*) kirjutame väärtuse $min\{(i, l), (i, j-1)+w_{\lambda \to b_j \dots b_l}\}$. Tabeli väljale (*i, j*) kirjutame väärtuse $min\{(i-1, j)+w_{kustutamine}, (i, j-1)+w_{lisamine}, (i-1, j-1)+w_{asendamine}, (i, j)\}$ – neist kolm esimest on tavalised teisendusoperatsioonid, väljal (*i, j*) on aga kirjas vähim lisateisendusoperatsiooni kasutamise hind, mida saab kasutada alamsõne $a_k \dots a_i$ teisendamiseks alamsõneks $b_l .. b_j$, $1 < k < i$, $1 < l < i$.

Üldistatud teisenduskauguse leidmiseks, kui teisendused on organiseeritud prefiksipuude abil, võib kasutada ka algoritmi versiooni, kus iga tabeli välja väärtuse leidmiseks leitakse seal lõppevatest teisendustest vähim. Sellisel juhul tuleks teisendusi organiseerida prefiksipuusse alustades viimasest sümbolist. Ettevaatavalt võib öelda, et seda algoritmi versiooni on kasutatud peatükis 7.

4.2.2 Üldistatud teisenduskauguse dünaamilise programmeerimise tabelist teisenduste taastamine

Nagu tavalise teisenduskauguse puhul, on ka üldistatud teisenduskauguse dünaamilise programmeerimise tabelist võimalik taastada minimaalsed teisendusteed. Võimalike teisendusteede jaoks enam ei piisa lihtsalt üles, vasakule ja diagonaalis ülevale jääva välja vaatamisest, lisaks neile tuleks ka jälgida võimalikke lisateisendusoperatsioone. Samuti kui tavalise teisenduskauguse puhul võib hakata moodustama hulki *pred*[*i, j*] – tähistamaks, millisest ruudust tuldi ruutu $d'_{i,j}$.

Hulka *pred*[*i, j*] lisame elemendi:

- $(i-1, j-1)$, *kui* $d'_{i,j} = d'_{i-1,j-1} + (\mathit{if}\ a_i = b_j\ \mathit{then}\ 0\ \mathit{else}\ 1)$
- $(i-1, j)$, *kui* $d'_{i,j} = d'_{i-1,j} + 1$
- $(i, j-1)$, *kui* $d'_{i,j} = d'_{i,j-1} + 1$
- $(i-k-1, j-l-1)$, kui leidub teisendus $a_k \ldots a_i \rightarrow b_l \ldots b_j$ kaaluga *weight* ning $d'_{i,j} = d'_{i-k-1,j-l-1} + weight$, $1 \leq k \leq i$, $1 \leq l \leq i$
- $(i-k-1, j)$, kui leidub teisendus $a_k .. a_i \rightarrow \lambda$ kaaluga *weight* ning $d'_{i,j} = d'_{i-k-1,j} + weight$, $1 \leq k \leq i$
- $(i, j-l-1)$, kui leidub teisendus $\lambda \rightarrow b_l \ldots b_j$ kaaluga *weight* ning $d'_{i,j} = d'_{i,j-l-1} + weight$, $1 \leq l \leq i$

$0 \leq i \leq m$, $0 \leq j \leq n$.

Näide 4.8. Olgu meil antud sõned *poster* ja *session* ning teisendusoperatsioonid:

$ost \rightarrow s$ *kaaluga* 0.3,

$post \rightarrow \lambda$ *kaaluga* 0.1,

$\lambda \rightarrow ssio$ *kaaluga* 0.2,

$post \rightarrow essi$ *kaaluga* 0.4,

$os \rightarrow s$ *kaaluga* 0.5,

$p \rightarrow \lambda$ *kaaluga* 0.8 ning

$er \rightarrow on$ *kaaluga* 0.9.

Leiame vastavalt üldistatud teisenduskauguse algoritmile dünaamilise programmeerimise tabeli ning taastame sellest võimalikud lühima kaaluga teisendusteed (joonis 4.10).

		s	e	s	s	i	o	n
	0	1	2	3	4	5	2,2	3,2
p	0,8	1	2	3	4	5	2,2	3,2
o	1,8	1,8	2	3	4	5	2,2	3,2
s	2,8	1,3	2,3	2	3	4	2,5	3,2
t	0,1	1,1	2,1	2,3	3	1,4	2,3	3,3
e	1,1	1,1	1,1	2,1	3,1	2,4	1,3	2,3
r	2,1	2,1	2,1	2,1	3,1	3,4	2,3	2,3

Joonis 4.10. Üldistatud teisenduskauguse dünaamilise programmeerimise tabelist teisenduste taastamine.

Iga noolekest joonisel 4.10 võiksime vaadelda kui elemendi lisamist hulka *pred*. Liikudes mööda hulki *pred*[*i, j*] kuni hulgani *pred*[0, 0] saame taastada kõik teisendused.

Nagu ka tavalise teisenduskauguse puhul, on üldise teisenduskauguse puhul võimalik esitada tehtud teisendusi ülevaatlikumal kujul. Teisenduste ilmestamiseks võiks neid kujutada näiteks joondamise abil:

P OST E — R
— S E SSIO N

ning

— POST ER
S ESSI ON

või kasutades jälitust:

POSTER
SESSION

ning

POSTER
SESSION

Ülevaatlikkuse huvides võib teisenduste puhul, milles asendatakse, kustutatakse või lisatakse tähti mitmehaaval, ühendada teisenduses osalevad tähed kaarekesega.

Samuti võib teisendusi kujutada kui nende läbiviimiseks tehtud operatsioonide loendit:

Esimesel juhul:	poster	$p \rightarrow \lambda$	oster
	oster	$ost \rightarrow s$	ser
	ser	$\lambda \rightarrow ssio$	sessior
	sessior	$r \rightarrow n$	session
ning teisel juhul:	poster	$\lambda \rightarrow s$	sposter
	sposter	$post \rightarrow essi$	sessier
	sessier	$er \rightarrow on$	session

5. Peatükk
Programm üldistatud teisenduskauguse leidmiseks

Autori semestritöö käigus valmis C programm üldistatud teisenduskauguse leidmiseks. Programmi on võimalik käivitada kahe erineva parameetrikombinatisooniga:

- EditDistance.exe **-best parimaidTulemusi teisendustefail ostsisõne sõnedefail**
- EditDistance.exe **teisendustefail otsisõne sõnedefail maxEditDistance**

Esimesel juhul tagastatakse nii mitu parimat tulemust, kui on antud täisarvuga *parimaidTulemusi*, teisel juhul väljastatakse kõik sõned, mille teisenduskaugus otsisõnest on väiksem või võrdne sisendparameetriga *maxEditDistance*.

Ülejäänud parameetreid võib kirjeldada järgnevalt:

- *teisendustefail* – fail, milles on allpoololevate reeglite järgi kirja pandud lisateisendusoperatsioonid
- *otsisõne* – sõne, mida hakatakse failis *sõnedefail* olevate sõnedega võrdlema. Leitakse nendevahelised teisenduskaugused, kasutades failis *teisendustefail* kirjeldatud lisateisendusoperatsioone
- *sõnedefail* – fail, millest sõne kaudseid teisendusi otsima hakatakse; eeldatakse, et failis iga uus sõna asub uuel real

Teisendused peavad olema kirjeldatud vastavalt järgmistele reeglitele:

- *str1:str2:weight* - sõne *str1* asendamine sõnega *str2*, millele on antud kaaluks *weight*
- *:str2:weight* - sõne *str2* lisamine kaaluga *weight*
- *str1::weight* - sõne *str1* kustutamine kaaluga *weight*

Kirjeldatud teisenduste ja neid eraldavate koolonite vahele ei tohi jääda üleliigseid tühikuid, vastasel juhul arvestatakse ka need teisenduse sisse. Näiteks teisendust *str1: :weight* tõlgendataks, kui sõne *str1* asendamist tühikuga kaaluga *weight*. Teisendustele antavad kaalud peavad ära mahtuma *double* arvutüübi sisse – s.o vahemikku $(2.2250738585072014*10^{-308}, 1.7976931348623157*10^{308})$.

Teisenduste failis on võimalik peale teisenduste defineerida ka kaalud vaikimisi lisamisele, kustutamisele ja asendustele:

- >rep:*weight* – asendusoperatsioonile antud uus kaal *weight*
- >rem:*weight* – kustutusoperatsioonile uus kaal *weight*
- >add:*weight* – lisamisoperatsioonile uus kaal *weight*

Kui mõni neist kaaludest pole failis defineeritud, siis kasutatakse vaikimisi kaalu, milleks on 1.

Vastavalt teisenduste failis olevatele teisendustele moodustatakse kolm teisenduste prefiksipuud – lisamiste, kustutamiste ja asenduste jaoks. Kasutades teisenduste prefiksipuid, arvutatakse üldistatud teisenduskauguse maatriks. Vastavalt sisendile kas väljastatakse saadud teisenduskaugus, kui see on väiksem sisendina antud teisenduskaugusest, või kogutakse parimate tulemuste listi ning väljastatakse nii palju tulemusi, kui antud parameetriga *parimaidTulemusi*. Samade teisenduskaugustega sõned grupeeritakse ühe teisenduskauguse alla ning tulemuses väljastatakse nõutud arv parimaid teisenduskaugusi, mitte parima teisenduskaugusega sõnesid. Näiteks kui sõnedefailis on antud 10 sõna, mis kõik on otsisõnest teisenduskaugusel 1 ning tahetakse tulemuses näha ainult 5 parimat tulemust, siis väljastatakse tulemuses kõik need 10 sõna.

Näide 5.1. Kasutades failis *teisendused.txt* kirjeldatud teisendusoperatsioone, tahame leida failis *kohanimed.txt* olevatest kohanimedest ainult kahte parimat tulemust.

```
>editdistance.exe -best 2 teisendused.txt tartu kohanimed.txt

 0.000000
tartu
------------------------


 1.000000
harku
hertu
taritu
------------------------
```

Tulemused esitatakse sorteerituna teisenduskauguse järgi, alustades kõige lähemast.

Näide 5.2. Kasutades failis *teisendused.txt* kirjeldatud teisendusoperatsioone, tahame leida kohanimede failist ainult selliseid, mis on sõnest *tartu* teisenduskaugusel ülimalt 1.

```
>editdistance.exe teisendused.txt tartu kohanimed.txt 1


harku
1.000000


hertu
1.000000


taritu
1.000000


tartu
0.000000
```

Näide 5.3. Leiame teisenduste failis *teisendused.txt* kirjeldatud teisendusi kasutades kohanimede failist ainult sellised kohanimed, mille teisenduskaugused sõnest *palasi* oleksid väiksemad kui 2.

```
> EditDistance.exe -best 5 teisendused.txt palasi kohanimed.txt

 0.000000
palasi
------------------------

 0.500000
palase
------------------------

 1.000000
alasi
kalesi
paasi
pagasi
pelesi
------------------------

 1.500000
jalase
pajusi
pakase
palace
palade
palatu
pallase
palmse
papusi
pedasi
pelsi
pilati
------------------------

 1.700000
kärasi
pardsi
parksi
partsi
pirusi
pärase
------------------------
```

Programmi väljundit vaadates võib märgata, et sõned, mis on teisenduskaugusel 0.5 ja 1, on sõnega *palasi* hääldusel väga sarnased. Sõnede hulgas, mis on antud sõnest teisenduskaugusel 1.5, leidub juba selliseid, mis esialgse sõnega hääldusel palju ei sarnane – näiteks *palmse* ja *pilati*. Teisenduskaugusel 1.7 on enamus sõnesid sellised, mis kõlalt sõnet *palasi* ei meenuta.

<u>Näide 5.4</u>. Leiame üldistatud teisenduskaugused sõne *tössine* ning sõnede *tõsine, tassike, töine* ja *tössike* vahel, kasutades üldistatud teisenduskauguse programmi.

Esimeseks etapiks on teisenduste faili tegemine. Olgu teada, et kahekordse kaashääliku asendamine ühekordse kaashäälikuga on meil väga sage teisendus (anname sellele kaaluks näiteks 0.3), vähem sageli aga esineb *ö* asendamist tähega *õ* (anname sellele kaalu 0.5). Kirjeldame need teisendused failis *teisendus.txt*:

```
ö:õ:0.5
ss:s:0.3
```

Sõned, millega tahame sõnet *tössine* võrdlema hakata, kirjutame omaette faili (nimetame selle faili nt. *soned.txt*):

```
tõsine
tassike
töine
tössike
```

Leiame üldistatud teisenduskaugused:

```
>EditDistance.exe -best 4 teisendus.txt tössine soned.txt

 0.800000
tõsine
------------------------

 1.000000
tössike
------------------------

 2.000000
tassike
töine
------------------------
```

6. Peatükk
Üldistatud teisenduskauguse programm ja Unicode

Arvutis kujutatakse andmeid kui bitijadasid. Ühte ja sama bitijada saab tõlgendada mitmel erineval viisil, olenevalt sellest, mida see esindab. Näiteks neli järjestikust bitioktetti võivad moodustada ühiku, mis väljendab reaalarvu. Sama hästi võivad need kujutada hoopis sõnet pikkusega neli, milles iga sümbol on ühe baidi suurune. Sõnede kodeerimisel määrab konkreetse bitijada esitluse kodeering [url:char]. Kodeering seab vastavusse digitaalsed bitijadad ning sümbolid, võimaldades digitaalsetel seadmetel teineteisega suhelda ning töödelda, salvestada ning vahetada sümbol-orienteeritud andmeid [url:ASCII].

Peaaegu kõik tavalised personaalarvutid kasutavad ASCII kodeeringut või mõnda selle laiendust. Akronüüm ASCII lahtikirjutatult oleks ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange*. ASCII standard publitseeriti juba aastal 1967 ning see baseerus inglise tähestikul. Sümbolite kujutamiseks kasutatakse 7 bitti – seega sai defineerida ainult 128 sümbolit. Esimesed 32 väärtust on mittetrükitavad kontrollsümbolid nagu näiteks *return, nullsümbol, faili lõpp* jne. Koodivahemikus 33-126 on kirjeldatud trükitavad sümbolid nagu tähemärgid, numbrid, kirjavahemärgid [url:ASCII1]. Suur osa tänapäeva teksti kodeerimismeetodeid baseeruvad ajallooliselt ASCII kodeeringul.

Näide 6.1. Olgu meil antud baidijada 77, 97, 106, 97. Dekodeerides seda ASCII kodeeringut kasutades, saame sõne *Maja* – kood 77 vastab suurtähele *M*, kood 97 tähele *a* ning 106 tähele *j*.

Erinevates keeltes tekstide salvestamiseks kasutatakse erinevaid kodeerimismeetodeid – näiteks rõhumärkidega tähtedega Lääne-Euroopa keelte jaoks Latin-1 (üks bait vastab ühele tähele), KOI-8 vene keele jaoks või EUC-JP (3 baiti tähemärgi kohta) jaapani keele jaoks. Nendes süsteemides on ühes ja samas dokumendis võimalik ainult väga väheste keelte kooskasutamine. Näiteks saab kombineerida inglise ja jaapani keelt, kuid seda ainult seetõttu, et inglise keeles ei kasutata rõhkudega tähemärke, kasutatud on ainult ASCII väärtused kuni 0xF0 [url:unic2].

Erinevate keelte kooskasutamise probleemi lahenduseks oleks üks tähemärgisüsteem, mis sisaldaks kõikides maailma keeltes kasutatavaid tähti. Selline tähemärgisüsteem on olemas ning seda kutsutakse Unicode-ks.

6.1 Unicode

Unicode-s on igale tähemärgile vastavusse seatud number vahemikus 0x1 - 0x10FFFF. Tähemärgile vastavat numbrit kutsutakse *kodeerimispunktiks* (ingl.k *code point*) ja kirjutatakse kujul: U+64A. U+ viitab, et tegemist on Unicode süsteemiga ning 64A on kuueteistkümnendsüsteemi arv, mis viitab konkreetsele tähemärgile [url:unic4]. Unicode-s on ruumi rohkem kui miljoni tähemärgi jaoks. See miljoni märgiline ruum on jagatud 17 tasandiks, milles igaühes on umbes 65000 tähemärki. Tänaseks on Unicode-s kirjeldatud alla 100 000 tähemärgi.

Tasandit 0 kutsutakse põhiliseks mitmekeelseks tasandiks – BMP (*Basic Multilingual Plane*) – see sisaldab kõiki tähemärke, mis olid programmeerijale saadaval ennem Unicode tulekut. Vahemikus 0-127 kirjeldatud tähemärkide koodid vastavad täpselt ASCII kodeerimistabeli samade tähtede koodidele. ISO-Latin-1 kodeeringu tähed on kirjeldatud vahemikus 128 - 255. Lisaks nendele on seal koodid vene, kreeka, araabia, katakana, rõhumärkidega ladina ja paljude muude tähtede jaos [url:unic2].

Tasandeid 1 kuni 16 on kutsutud ka astraaltasanditeks (ingl.k *astral planes*). Neid on kasutatud eksootiliste, haruldaste ja ajalooliselt tähtsate tähtede kirjeldamiseks – näiteks „Old Italic", „Deseret", „Byzantine Musical Symbols" jne. Täpsemalt võib neid tähemärke vaadata [url:unic1].

6.2 Kodeeringud

Koos tähemärkide nummerdamisega defineerib Unicode ka meetodid nende salvetamiseks baidijadadesse. Levinumateks kodeeringuteks on UTF-8, UTF-16 ning UTF-32. Kodeeringute nimede prefiks UTF lahti kirjutatuna on *Unicode Transformation Format* [url:unic3].

Kasutame näidetes nelja erinevat tähemärki:

- U+0026 – ampersandi märk & (kümnendarvuna 38)
- U+0416 – vene tähestiku suur Ж (kümnendarvuna 1046)
- U+4E2D – Han(hiina) ideograaf 中 (kümnendarvuna 20 013)
- U+10346 – gooti tähestiku täht 𐍆 (kümnendarvuna 66 374)

Juhime tähelepanu, et näiteks toodud tähtedest viimane on astraaltasandi täht (BMP-s on ainult kodeerimispunktid 0 - 65 535).

6.2.1 UTF-32

UTF-32 on lihtsaim viis Unicode tähemärkide kodeerimiseks. Iga tähe jaoks kasutatakse 32 bitti, seega iga täht salvestatakse kui 4 baidine number. Meie näitetähemärgid salvestataks selles kodeeringus kui neljabaidised numbrid vastavalt väärtustega 38, 1046, 20013 ning 66374.

See kodeerimisviis vastab sellele, kuidas C kompilaatorid salvestavad tähti, mis on deklareeritud kui *wchar_t* tüüpi sümbolid. Ette rutates võib öelda, et sama moodi on käsitletud tähemärke ka Unicode toega üldistatud teisenduskauguse programmis.

Selle kodeeringu probleem seisneb selles, et 32 biti sisse salvestatakse ka tähed, mis mahuksid 8 biti sisse. Näiteks tekst, mis sisaldaks ainult inglise tähestiku tähti, võtaks selles kodeeringus ruumi umbes neli korda rohkem kui ASCII kodeeringus.

6.2.2 UTF-16

UTF-16 salvestab Unicode tähemärgid 16 bitistesse plokkidesse. Kõik tähed BMP-s salvestatakse iseendana. Astraaltasandite tähemärgid aga ei mahu 16 biti sisse. Nende salvestamiseks kasutatakse *surrogaatplokke* (ingl.k *surrogate blocks*) – BMP-s asub kaks 1024 koodipunkti mahutavat plokki, mida tavaliste tähtede kujutamisel ei kasutata. Kõrgem surrogaatplokk algab kodeerimispunktist U+D800 ning madal kodeerimispunktist U+DC00. Astraaltasandi tähtede kodeering jaotatakse kahte ossa – alumise 10 biti salvestamiseks kasutatakse madalamat surrogaati ning ülemise 10 biti salvestamiseks kõrgemat. Seega meie näidete astraaltasandi tähemärk U+10346 kodeeritakse kahe 16-bitise väärtuse 0xD800 ning 0xDF46 abil. Ülejäänud tähed meie näitest kodeeritakse lihtsalt 16 biti sisse – nende väärtused oleksid vastavalt 0x26, 0x416 ning 0x4E2D.

Selline viis võimaldab meil lisaks salvestada 2^{20} tähemärki, mis mahutab täpselt 16 astraaltasandit, milles igaühes on 2^{16} tähemärki.

UTF-16 on arvatavasti kõige efektiivsem viis näiteks Aasia tähemärkide esitamiseks. ASCII tähemärkide esitamisel UTF-16 abil lõpetame jälle seal, kus ühe baidi sisse mahtuva tähemärgi esitamiseks kasutame kahte.

6.2.3 UTF-8

UTF-8 kasutab 1 - 4 baiti ühe tähemärgi kodeerimiseks. Esimesed 127 Unicode sümbolit salvestatakse ühe baidi sisse. Kuna esimesed 127 Unicode sümbolit on identsed ASCII sümbolitega, siis sõne, mille tähemärkide koodid jäävad vahemikku 0-127, on UTF-8 kodeeringus baiditasandil täpselt samasugune kui ASCII kodeeringus.

Kodeerides tähemärki, mille kodeerimispunkt on suurem kui 0x7F (127):

- esimese baidi kõrgemad bitid näitavad, mitu baiti kasutatakse antud tähemärgi salvestamiseks. Nendele bittidele järgneb 0, mis näitab informatsioonibittide lõppu ning tähemärgi kodeeringu algust.
- kõikide järgnevate baitide kaks ülemist bitti on 10 (signaalbitid)
- tähemärgi kirjeldamiseks kasutatakse bitte, mis jäävad üle peale signaalibittide

Näiteks olgu meil antud tähemärk, mis on kodeeritud kahe baidi sisse. Esimese baidi kolm kõrgemat bitti oleks 110. Kaks esimest bitti 11 näitavad, et kodeerimiseks kasutatakse kahte baiti. Kolmas bitt 0 näitab informatsioonibittide lõppu. Peale selle jääb selles baidis 5 bitti tähemärgi jaoks. Teises baidis on kaks kõrgemat bitti signaalbittideks väärtustega 1 ja 0, jättes 6 bitti tähemärgi koodi jaoks [url:unic3].

Vaatleme meie näitemärke:

- U+0026 kodeeritakse iseendana: 0x26
- U+0416 kodeeritakse kahe baidi sisse: 0xD0, 0x96
- U+4E2D kodeeritakse kolme baidi sisse: 0xE4, 0x8, 0xAD
- U+10346 nelja baidi sisse: 0xF0, 0x90, 0x8D, 0x86

6.3 Üldistatud teisenduskauguse programm Unicode toega

Autori semestritöö käigus valminud programm (kirjeldatud peatükis 5) eeldab, et kõik tähemärgid on 1 baidised – seega ei oleks võimalik arvutada teisenduskaugusi sõnede vahel, mille puhul ühe tähe kujutamiseks kasutatakse rohkem kui 8 bitti. Samas kodeeringud, mis kasutavad tähemärgi kujutamiseks 1 baiti, ei ole ka alati piisavad. Näiteks kui tahaksime defineerida teisendusi ladina ning vene keele tähtede vahel – kuigi ISO vene keele tähti sisaldavas kooditabelis (*Cyrillic*) vastavad esimesed *n* baiti ASCII sümbolitele, jääb meil siiski veel sellest väheks. Näiteks ei saaks me kuidagi defineerida teisendusi *š* ning *ш* tähe vahel, kuna ISO kooditabelite seas ei leidu sellist, mis kirjeldaks ära kõik vene tähed ning sisaldaks ka š tähte.

Lisame üldistatud teisendukauguse programmile Unicode toe – vaatame iga tähte kui mitmebaidilist sümbolit. Loeme teisendused ning sõned sisse nagu tavaliselt – baitide massiivina. Saadud baidimassiivi teisendame aga *wchar_t* tüüpi sõnedeks ning leiame teisenduskauguse nende vahel.

6.3.1 Programmi kasutamine Linuxi all

Programm töötab keskkondades, kus on olemas standardne ISO C *locale* tugi. Programmi kasutamiseks Linuxi all on kõigepealt vajalik seadistada keskkond kasutama UTF-8 kodeeringut. Selleks tuleb keskkonnamuutujas LANG seada kodeeringuks UTF-8 – näiteks käsuga *setenv LANG et_EE.utf8* või *export LANG=et_EE.utf8* sõltuvalt kasutatavast interpretaatorist. Keskkonna automaatseks seadmiseks on lisades number 1, 2 ja 3 ka käsureaskript. Skripti laadimine seadistab keskkonna ainult konkreetse sessiooni jaoks.

Samuti peab kasutatav terminal olema seadistatud kasutama UTF-8 kodeeringut. Näiteks terminaliemulaatori ja SSH klientprogrammi PuTTy [url:PuTTy] seadistamiseks tuleks valida peamenüüst *Change settings -> Translation-> UTF-8.*

Programmi sisendiks olevad teisenduste ning sõnede failid peavad olema UTF-8 kodeeringus. Samuti tuleb jälgida, et kuigi osad tähed on välimuselt täiesti sarnased, ei ole sarnased nende Unicode koodid – näiteks ladina tähestiku *a* ning kirillitsa tähestiku *а* on välimuselt täpselt sarnased, kuid esimese puhul on kodeerimispunktiks U+0061 ning teisel U+0430. Programmi jaoks on need tähed sama erinevad kui näiteks *e* ja *ю*.

6.3.2 Programmi rakendamine

Programmi sisendparameetrid on täpselt samasugused nagu peatükis 5 kirjeldatud üldistatud teisenduskauguse programmil:

- EditDistance.exe **-best parimaidTulemusi teisendustefail ostsisõne sõnedefail**
- EditDistance.exe **teisendustefail otsisõne sõnedefail maxEditDistance**

Esimesel juhul tagastatakse nii palju parimaid tulemusi, kui on määratud parameetriga *parimaidTulemusi.* Teisel juhul tagastatakse ainult tulemused, mille teisenduskaugus on väiksem või võrdne kui *maxEditDistance*.

Defineerime teisendused ladina ning vene keele tähtede vahel. Kasutame vene-ladina transliteratsiooni, mille on heaks kiitnud ÜRO V kohanimekonverents Montréalis 1987, Eesti Emakeele Seltsi keeletoimkond 1996 ning Vabariigi Valitsus. Samuti vene-eesti transkriptsiooni, mille kinnitas ametlikult aastal 1998 Vabariigi Valitsus [Erelt05]. Lisaks võtame arvesse venekeelsetes foorumites ladina tähtedega kirjutatud vene sõnades tihedamini kasutatud teisendusi. Antud teisendused on toodud failis *rusTeisend.txt* lisas 4. Eesmärgiks on ladina tähtedega kirjutatud vene keelse sõna kokkuviimine selle vene tähtedega kirjutatud kujuga. Programmi saaks kasutada näiteks vene-eesti sõnastikus – otsides mingi vene keelse sõna eestikeelset vastet, ei pruugi kasutajal olla vene klaviatuuri käepärast. Seega oleks väga mugav viis talle võimaldada sõna kirjutamist ladina tähtedega.

Järgnevalt on ära toodud kaks näidet programmi kasutamisest (näide 6.2 ja 6.3).

Näide 6.2. Leiame failis *rusTeisend.txt* defineeritud teisendusi kasutades failist *testS6nad.txt* sõnele *ulitsa* kolm parima teisenduskaugusega tulemust. Saadud tulemus on joonisel 6.1.

```
./EditDistance -best 3 rusTeisend.txt ulitsa testS6nad.txt

 0.000800
улица
------------------------


 3.000200
конца
------------------------


 3.000400
сейчас
плочу
------------------------
```

Joonis 6.1. Sõnele *ulitsa* sõnede failist *testS6nad.txt* leitud kolm parima teisenduskaugusega tulemust.

Näide 6.3. Leiame failis *rusTeisendused.txt* defineeritud teisendusi kasutades sõnele *ulitsa* failist *testSõnad.txt* kõik sõned, mille teisenduskaugus on ülimalt 4. Saadud tulemus on joonisel 6.2.

```
./EditDistance rusTeisend.txt ulitsa testS6nad.txt 4


сейчас
3.000400


летится
3.800600


плочу
3.000400


улица
0.000800


конца
3.000200


личная
3.000600


учить
3.000600


теплиться
3.801000
```

Joonis 6.2. Failist *testS6nad.txt* leitud sõned, mis on sõnest *ulitsa* üldistatud teisenduskaugusel 4.

7. Peatükk
Üldistatud teisenduskauguse sõnedefaili teisendamine prefiksipuu kujule

Seni demonstreeritud juhtudel otsisõne ning kõigi sõnedefailis olevate sõnede vaheliste teisenduskauguste leidmisel arvutasime iga sõnedefailis oleva sõne puhul terve dünaamilise programmeerimise tabeli. Sõnede puhul, mis omavad sama prefiksit, olgu selle pikkus k, on dünaamilise programmeerimise tabelis täpselt k+1 veergu identsed. Sama prefiksit omavad sõned võiksid seda identset tabeli osa jagada. Sellega vähendaksime üleliigset tööd ning kokkuvõtteks tõstaksime programmi kiirust.

Näide 7.1: Vaatleme sõne *tartu* ning sõnede *alliksoo* ja *alliku* vahelise teisenduskauguste leidmiseks arvutatud dünaamilise programmeerimise tabeleid (joonis 7.1).

		a	l	l	i	k	s	o	o
	0	1	2	3	4	5	6	7	8
t	1	1	2	3	4	5	6	7	8
a	2	1	2	3	4	5	5	7	8
r	3	2	2	3	4	5	6	7	8
t	4	3	3	3	4	5	6	7	8
u	5	4	4	4	4	5	6	7	8

		a	l	l	i	k	u
	0	1	2	3	4	5	6
t	1	1	2	3	4	5	6
a	2	1	2	3	4	5	6
r	3	2	2	3	4	5	6
t	4	3	3	3	4	5	6
u	5	4	4	4	4	5	5

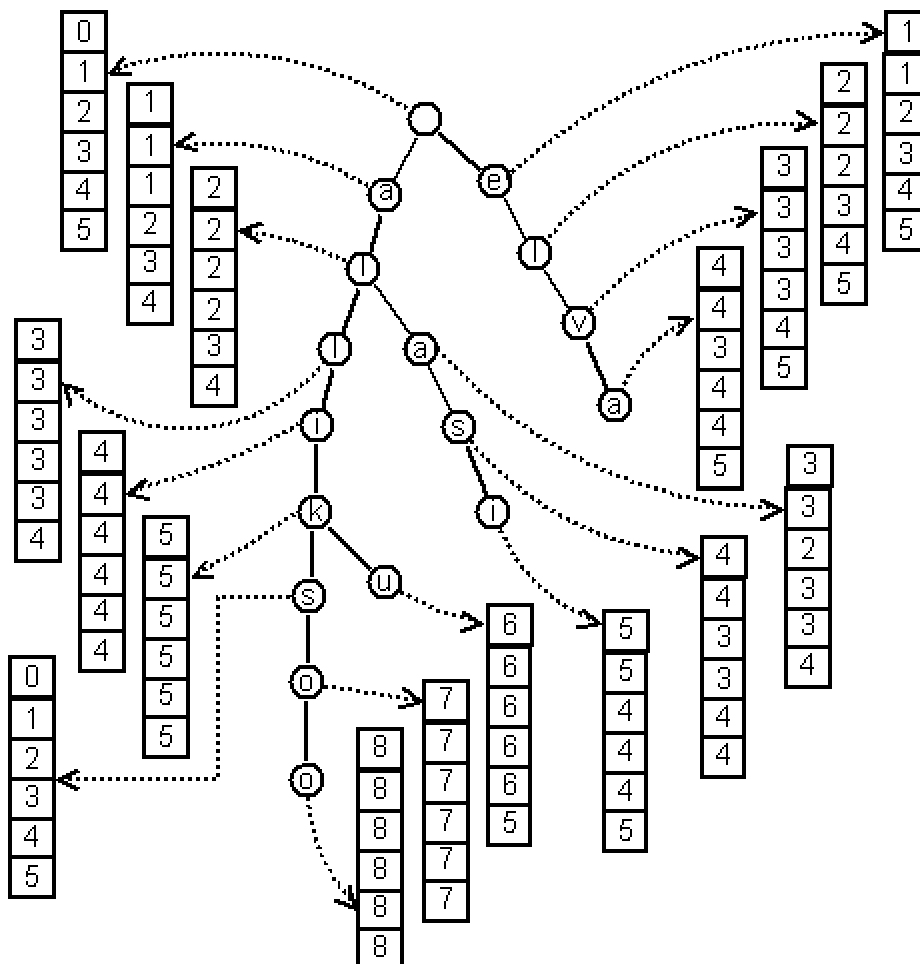
Joonis 7.1. Sõne *tartu* ning sõnede *alliksoo* ning *alliku* vahelise teisenduskauguse dünaamilise programmeerimise tabelid.

Sõnede *alliksoo* ning *alliku* ühised prefiksid on pikkusega 5. Vaadates tabeleid joonisel 7.1, võimegi näha, et tabelite esimesed 6 veergu on identsed. Sama reegel kehtib ka siis, kui juurde on defineeritud lisateisendusi – nende prefiksi ulatuses on tegemist sama sõnega.

Selle omaduse ärakasutamiseks organiseerime ka sõnedefaili prefiksipuu kujule. Iga prefiksipuu tipuga seome ühe tabeli veeru, mis vastab otsisõne ning sõnede prefiksipuus antud kohal lõppeva sõne vahelisele üldistatud teisenduskaugusele. Vastavalt prefiksipuu omadustele on sama prefiksit omavad sõned sama prefiksipuu haru küljes ning jagavad sama tabeli osa.

Üldistatud teisenduskauguse sõne *S* ning kõigi prefiksipuus olevate sõnede vahel leiame kasutades puu sügavuti läbimise algoritmi. Igal liikumisel prefiksipuus järgmise tipu juurde, lisame juurde ühe tabeli veeru. Liikumisel tipu juurtipu või naabertipu juurde, kustutame hetkel vaatluse all oleva tipuga seotud tabeli veeru – seda meil enam vaja ei lähe. Iga tipuga seotud veeru täidame nagu täidaksime tavalist dünaamilise programmeerimise tabeli veergu. Kogu vajalik informatsioon selle tegemiseks on olemas eelnevates tippudes.

Näide 7.2: Tahame leida sõne *tartu* ning sõnede *alliksoo*, *alasi*, *alliku* ja *elva* vahelisi teisenduskaugusi. Selleks lisame sõned sõnede *alliksoo*, *alasi*, *alliku* ning *elva* prefiksipuusse (joonis 7.2).



Joonis 7.2. Otsisõnede prefiksipuu, mille iga tipuga on seotud vastav tabeli veerg

Kõigi joonisel 7.2 kujutatud prefiksipuu tippude puhul kehtib reegel: olles ükskõik millise tipu juures, väljendab selle tipuga seotud tabeli veerg sõne *tartu* ning selles punktis lõppeva sõne vahelist teisenduskaugust. Veeru kõige alumine element ongi teisenduskauguseks otsisõne ja selles kohas lõppeva sõne/alamsõne vahel. Tipuga, milles lõppeb mingi sõne, seotud veeru viimane element ongi teisenduskaugus nende kahe sõne vahel.

Jagatud dünaamilise programmeerimise tabeli kasutamiseks peame üldistatud teisenduskauguse algoritmi natukene muutma. Varem vaatasime enne iga välja täitmist, kas sellest väljast algab mõni lisateisendus ning selle leidumisel kirjutasime tulemuse tabelisse. Praegusel juhul me seda enam teha ei saa, kuna veeru täitmisel pole tahapoole jäävad veerud veel loodudki. Seega iga välja täitmisel leiame kõik teisendused, mis sellel kohal lõppevad ning välja väärtuseks võtame neist vähima.

Antud väljal lõppevate teisenduste leidmiseks on mitmeid mooduseid. Naiivne lahendus oleks iga välja täitmisel vaadata kõiki välju, mis jäävad sellest üles vasakule ning vaadelda, kas mõnest neist algab teisendus, mis lõppeb antud väljal. Selline lahendus aga tekitaks algoritmile tunduvalt lisatööd. Kavalam oleks aga hakata teisendusi otsima alates viimasest tähest. Järgnevalt toome algoritmi (algoritm 7.1), milles on kirjeldatud, kuidas konkreetse välja (i, j) täitmisel leida, kas antud teisendust $\alpha \rightarrow \beta$ saab kasutada või mitte.

Algoritm 7.1. Meetod leidmiseks, kas teisendust $\alpha \to \beta$ saab kasutada antud tabeli välja (*i, j*) täitmisel.

Sisend: Sõned $A=a_1 a_2 ... a_m$ ja $B=b_1 b_2 ... b_n$ ning teisendus $\alpha \to \beta$, kusjuures $\alpha=\alpha_1 ... \alpha_k$, $\beta=\beta_1 ... \beta_l$.

Väljund: Algoritm tagastab tõeväärtuse **true**, kui seda teisendust saaks selle koha peal kasutada ning **false**, kui seda teisendust selle koha peal kasutada ei saaks.

Meetod:

1. t = 0

// vaatame, kas teisenduse vasak pool sobib

2. **while**(t < k)

3. **if**($\alpha[k-t]=a[i-t]$) **then** t = t +1

4. **else return false**

// vaatame, kas teisenduse parem pool sobib

5. t = 0

6. **while**(t < l)

7. **if**($\beta[l-t]=b[j-t]$) **then** t = t+1

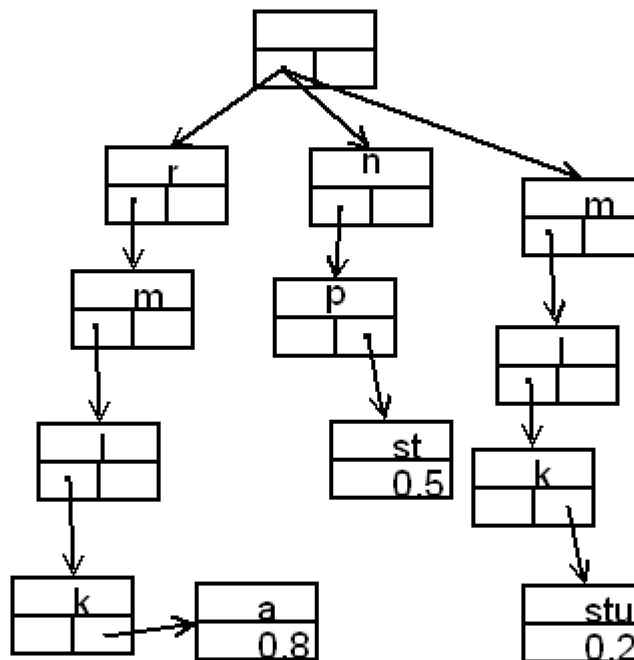
8. **else return false**

9. **return true**

Antud veeru täitmiseks sobivaid lisakustutamisoperatsioone otsides tuleb koodiread 5 – 8 vaatluse alt välja jätta. Lisamisoperatsioonide puhul pole vaja täita ridu 1 – 4.

Organiseerime lisateisendused prefiksipuudesse nii, et teisendus algab juurest alates viimasest tähest. Tipuga, mis vastab mingi teisenduse algusele, seome asenduste prefiksipuu puhul teisenduse parema poole ning lisamise ja kustutamise prefiksipuu puhul teisenduse hinna. Seega lisades prefiksipuusse teisendust $\alpha \to \beta$, $\alpha=\alpha_1 ... \alpha_t$, lisame alguses tähe α_t, seejärel α_{t-1} kuni täheni α_1. Tipuga, millesse salvestame α_1, seome teisenduse parema poole β ning selle teisenduse hinna.

Näide 7.3. Vaatleme näites 4.6 antud teisendusi $klm \to stu$ kaaluga 0.2, $klmr \to a$ kaaluga 0.8 ning $pn \to st$ kaaluga 0.5 ning organiseerime need prefiksipuusse alustades viimasest tähest (joonis 7.3):



Joonis 7.3. Teisenduste organiseerimine prefiksipuu andmestruktuuri alates viimasest tähest.

Võrreldes joonisel 4.8 kujutatud prefiksipuuga, muutus teisenduste prefiksipuu laiemaks, kuid see ei ole reegel. See, kas prefiksipuu muutub laiemaks või kitsamas, sõltub antud teisenduste prefiksitest.

Üldistatud teisenduskauguse leidmiseks sõnede jaoks prefiksipuud ning jagatud dünaamilise programmeerimise tabelit kasutades peame iga tipu puhul võtma selles kohas rakendatavate teisendusoperatsioonide hindadest vähima. Sõnede prefiksipuud läbime süvitsi – kui vaatluse all oleval tipul on alluvaid, liigume alluva töötlemise juurde. Kui tipul alluvaid pole, töötleme tipu parempoolset naabrit ning kui tipul pole parempoolseid naabreid ning alluvaid või kõik alluvad on töödeldud, siis liigume tipu vahetu eellase töötlemise juurde. Antud üldistatud teisenduskauguse leidmise meetod, kasutades prefiksipuu süvitsi läbimist, on kirjeldatud algoritmis 7.2.

Algoritm 7.2. Üldistatud teisenduskauguse leidmine, kasutades sõnede prefiksipuud ja jagatud tabelit.

Sisend: Otsisõne $A = a_1 a_2 \ldots a_m$, sõnede prefiksipuu *S*.

Meetod:

1. tipp = $S_{juurtipp}$

2. **while**(tipp != **null**)

// tipu töötlemine

3. **for** i = 0 **to** length(tipp->veerg) **do**

4. tipp->veerg[i] = min(selles kohas lõppevad teisendused)

// otsime töötlemiseks järgmise tipu

5. **while**(tipp != **null**)

6. **if**(tipp->alluvad != **null**) **then**

7. tipp = tipp->esimene_alluv

8. loo(tipp->veerg)

9. **break**

10. **else if**(tipp->parempoolne_naaber != null) **then**

11. tipp = tipp->parempoolne_naaber

12. loo(tipp->veerg)

13. **break**

14. **else**

15. kustuta(tipp->veerg)

16. tipp = tipp->ülemtipp

17. **return**

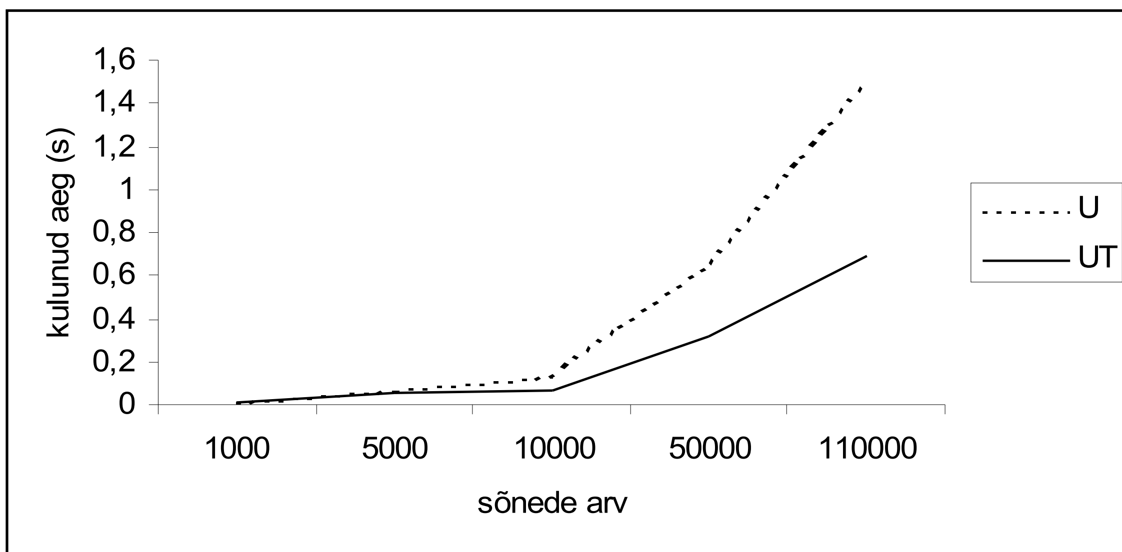
Võrdleme antud programmi kiirust peatükis 6 tutvustatud programmiga. Võrdleme ainult otsisõne ning kõigi sõnedefailis olevate sõnede vahelise üldistatud teisenduskauguse arvutamiseks kuluvat aega. Joonistel 7.4, 7.5, 7.6 ning 7.7 tähistab U tavalist üldistatud teisenduskauguse programmi ja UT tähistab üldistatud teisenduskauguse programmi, mis loeb sõnedefaili prefiksipuusse ning kasutab jagatud dünaamilise programmeerimise tabelit.

Võtame vaatluse alla failid, milles on 100, 1000, 5000, 10 000, 50 000 ning 110 000 sõne. Otsisõne pikkuseks on 4 sümbolit. Iga sõnedehulgal tegime 10 katset ning tabelisse sisestasime nende keskmise väärtuse. Joonisel 7.4 on näha selle katse tulemus.

	1000	5000	10000	50000	110000
U	0,01	0,07	0,13	0,66	1,49
UT	0,01	0,05	0,07	0,32	0,69

Joonis 7.4. Programmide võrdlemise tabel.

Joonisel 7.5 on näha antud katse tulemus graafikul. Näeme, et mida suuremaks muutub sõnede hulk, millega etteantud sõne võrdleme, seda suuremaks läks programmide tööaja erinevus.



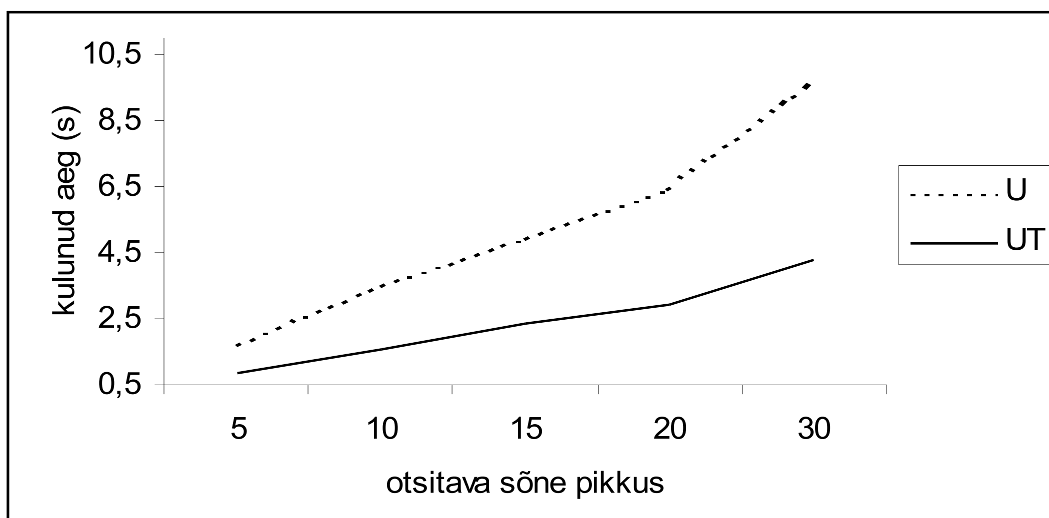
Joonis 7.5. Programmide tööaja võrdlemine.

Uurime programmide töötamise aega otsisõne pikkuse kasvades. Leiame üldistatud teisenduskauguse kui otsisõne pikkuseks on 5, 10, 15, 20 ja 30 sümbolit. Sõnedefailina kasutame 110 000 sõnega faili. Saadud tulemused on näha joonisel 7.6

	5	10	15	20	30
U	1,74	3,53	4,96	6,41	9,68
UT	0,84	1,61	2,32	2,94	4,26

Joonis 7.6. Programmide tööajad otsisõne pikkuse kasvades.

Esitame saadud tulemuse ka graafikul (joonis 7.7).



Joonis 7.7. Programmide tööaeg sõne pikkuse kasvades.

8. Peatükk
Üldistatud teisenduskauguse teisendustele kaalude leidmine

Üheks suurimaks probleemiks üldistatud teisenduskauguse jaoks teisenduste defineerimise juures on neile reaalsete kaalude leidmine. Näiteks vene keele tähtede ja ladina tähtede vahele teisenduste defineerimisel võib kindlalt öelda, et ladina tähestiku *b* täht teisendatakse alati vene tähestiku täheks *б*, kuid näiteks täheühendi *sh* teisendamisel me nii kindlad pole. *Sh* teisendatakse mõnel juhul täheks *ш*, mõnel juhul hoopis täheks *щ*. Siiamaani oleme teisendustele kaale pannud subjektiivse hinnangu põhjal. Antud peatükis kirjeldamegi automaatset meetodit teisendustele kaalude genereerimiseks näidete hulga pealt.

Olgu meil antud mingi hulk teisendusi, millele tahame kaale leida. Alguses anname kõikidele teisendustele kaaluks 1. Seega iga lisateisenduse kasutamine on samaväärne tavaliste teisendusoperatsioonide kasutamisega.

Teisenduste õppimiseks on meil vaja sõnepaaride komplekti. Nimetagem nendest sõnedest esimest tinglikult esialgseks sõneks ning teist saadavaks sõneks. Meie ülesandeks on vaadata, milliseid teisendusoperatsioone kasutades muudetakse esimene sõne teiseks – teisendusoperatsioonide seas arvestame nii tavalisi teisendusoperatsioone kui ka juurdedefineeritud teisendusoperatsioone. Selleks leiame nende sõnede vahel üldistatud teisenduskauguse ning vaatame, milliseid teisendusi kasutati parimatel teisendusteedel.

Näide 8.1. Sõne *pismo* ja *письмо* vahelise teisenduskauguse leidmisest, kui juurde on defineeritud teisendused $o \to о$, $p \to п$, $i \to и$, $s \to с$, $\lambda \to ь$, ning $m \to м$. Saadud üldistatud teisenduskauguse tabel on toodud joonisel 8.1.

		п	и	с	ь	м	о
	0	1	2	3	4	5	6
p	1	1	2	3	4	5	6
i	2	2	2	3	4	5	6
s	3	3	3	3	4	5	6
m	4	4	4	4	4	5	6
o	5	5	5	5	5	5	6

Joonis 8.1 Sõnede *pismo* ning *письмо* vahelise üldistatud teisenduskauguse tabel.

Joonisel 8.1 on näidatud ka kõik parimad teisendusteed sõne *pismo* teisendamiseks sõneks *письмо.* Punktiirjoonega noolega on märgitud teisendused, kus on kasutatud juurdedefineeritud teisendusoperatsioone.

Iga teisenduse jaoks leiame tõenäosuse konkreetse teisenduse kasutamiseks. Tõenäosused leiame valemite abil, mida tutvustati allikas [LH01].

$$P_R[\alpha,\beta]=\frac{\sum(kasutatud\ teisendusi\ \alpha\to\beta)}{\sum(\beta\ esinemiste\ arv\ saadavas\ sõnes)}$$

$$P_D[\alpha]=\frac{\sum(kasutatud\ teisendusi\ \alpha\to\lambda)}{\sum(\alpha\ esinemiste\ arv\ esialgses\ sõnes)}$$

$$P_I[\beta]=\frac{\sum(kasutatud\ teisendusi\ \lambda\to\beta)}{\sum(\beta\ esinemiste\ arv\ saadavas\ sõnes)}$$

Antud valemites $P_R[\alpha,\beta]$ on tõenäosus teisenduse $\alpha\to\beta$ kasutamiseks – s.o tõenäosus, et mingis sõnes asendatakse täht α tähega β. $P_D[\alpha]$ on tõenäosus tähe α kustutamiseks ning $P_I[\beta]$ tõenäosus tähe β lisamiseks [LH01]. Nende tõenäosuste leidmisel kasutatakse ainult ühte parimat teisendusteed.

Allikas [LH01] kirjeldatud meetodis kasutati tõenäosuste leidmist ainult kahe tähe vähel, meil aga võib teisenduses α või β rollis olla ka rohkem tähti. Sellisel juhul võime vaadata iga tähekombinatsiooni kui mingit tähestiku tähte. Näiteks teisenduse $\lambda \rightarrow klmn$ puhul vaatleme täheühendit $klmn$ kui mingit tähestiku tähte ning selle teisenduse kasutamise tõenäosuse leiame valemist:

$$P(\lambda \rightarrow klmn) = \frac{teisenduse\ \ \lambda \rightarrow klmn\ kasutamiste\ arv}{\sum (alamsõne\ klmn\ arv\ saadavas\ sõnes)}$$

Ühe sõne teisendamisel teiseks võib leiduda mitu parimat teisendusteed. Võtame arvesse neid kõiki. Parimatel teisendusteedel kasutatud teisendused leiame kasutades algoritmi 8.1.

Algoritm 8.1. Dünaamilise programmeerimise tabelist teisenduste taastamine.

Sisend: Sõned $A = a_1 a_2 \ldots a_m$ ning $B = b_1 b_2 \ldots b_n$, nendevahelise teisenduskauguse leidmiseks arvutatud dünaamilise programmeerimise tabel *D* ning lisateisenduste hulk *T*.

Väljund: Teisenduste hulk, mida kasutati mingil parimal teisendusteel.

Meetod:

// Väljad, mida läbib mingi teisendustee salvestame hulka Pred, parimatel teisendusteedel

// kasutatud teisendusi hakkame salvestama hulka Teisendused.

// välju tähistame: väli[i, j], kus i on antud välja rida ning j on antud välja veerg

// lisame tabeli parempoolseima alumise välja massiivi Pred

1. väli[i, j] = (m, n)
2. **while**(väli[i, j] != NULL)
3. **if**($\exists\ \alpha_1 \ldots \alpha_k \rightarrow \beta_1 \ldots \beta_l \in T$:
4. $\alpha_1 \ldots \alpha_k = a_{i-(k-1)} \ldots a_i \ \wedge$
5. $\beta_1 \ldots \beta_l = b_{j-(l-1)} \ldots b_j \ \wedge$
6. $D[i-k, j-l] + weight_{\alpha_1 \ldots \alpha_k \rightarrow \beta_1 \ldots \beta_l} = D[i, j]$) **then**
7. *Lisa hulka Pred väli [i-k, j-1]*
8. *Lisa hulka Teisendused teisendus* $\alpha_1 \ldots \alpha_k \rightarrow \beta_1 \ldots \beta_l$
9. väli = väli->*järgmine_väli*
10. **return** *Teisendused*

Teisendustele tõenäosuste leidmiseks leiame, mitu korda neid kasutati mingil parimal teisendusteel ning mitu korda neid teisendusi oleks üldse kasutada saanud. Seega teisendustele tõenäosuste arvutamiseks kasutame valemeid:

(2)

$$P_R[\alpha,\beta]=\frac{\sum(kasutatud\ teisendusi\ \alpha\rightarrow\beta\ parimatel\ teisendusteedel)}{\sum(\beta\ esinemiste\ arv\ saadavas\ sõnes)}$$

$$P_D[\alpha]=\frac{\sum(kasutatud\ teisendusi\ \alpha\rightarrow\lambda\ parimatel\ teisendusteedel)}{\sum(\alpha\ esinemiste\ arv\ esialgses\ sõnes)}$$

$$P_I[\beta]=\frac{\sum(kasutatud\ teisendusi\ \lambda\rightarrow\beta\ parimatel\ teisendusteedel)}{\sum(\beta\ esinemiste\ arv\ saadavas\ sõnes)}$$

Kui parimal teisendusteel kasutati ka tavalisi teisendusoperatsioone – s.o teisendusoperatsioone, mida me ei ole juurde defineerinud, siis neid me lihtsalt ignoreerime.

Näide 8.2. Leiame näites 8.1 kasutatud teisendustele kaalud. Vaadates joonist 8.1 näeme, et iga lisateisendusoperatsiooni kasutati parimatel teedel üks kord. Kasutades valemeid (2) leiame antud lisateisendustele tõenäosused. Antud juhul tuleb kõigi nende lisateisendusoperatsioonide kasutamise tõenäosuseks 1.

Teisendustele kaalude leidmiseks peame defineerima mingi funktsiooni, mis seoks teisenduse kasutamise tõenäosust ning selle hinda. Kuna kõik tavalised teisendusoperatsioonid on meil kaaluga 1, siis kaalufunktsioon peaks vastama järgmistele reeglitele:

$$f(t)=\begin{cases}1, & kui\ t=0\\ 0, & kui\ t=1\\ 0<f(t)<1, & muul\ juhul\end{cases}$$

Kui teisenduse kasutamise tõenäosus on võrdne 1-ga, siis peaks teisenduse hind olema väikseim võimalik – selle kaal peaks olema võrdne 0-ga. Kui teisendust üldse ei kasutanud, peaks selle kaal olema sama, mis tavalistel teisendusoperatsioonidel – antud juhul on selleks 1. Muul juhul peaks kehtima reegel, et mida suurem on teisenduse kasutamise tõenäosus, seda väiksema kaalu me sellele anname. Selleks funktsiooniks võtame: $f(t)=1-t$, kus t on mingi teisenduse kasutamise tõenäosus.

8.1 Ladina ja vene keele tähtede vahelistele teisendustele kaalude automaatne õppimine

Antud töö käigus valmis programm näitesõnede hulga pealt teisendustele kaalude leidmiseks. Programmi käivitamiseks on vaja seada keskkond kasutama UTF-8 kodeeringut nagu kirjeldatud peatükis 6.3.1.

Programmile on vaja ette anda sisend kujul:

> findWeights **teisendusteFail näitesõnedeFail tulemusteFail**

Failist *teisendusteFail* loeme sisse teisendused, millele hakkame kaale leidma. Teisendused peavad olema failis kirjeldatud kujul:

teisenduse_vasak_pool:*teisenduse_parem_pool*

Iga teisendus peab olema defineeritud eraldi real ning faili lõpus ei tohi olla üleliigseid reavahetussümboleid.

Teisendused kirjutame prefiksipuudesse ning anname neile kaaluks 1. Teisenduste prefiksipuude tippudel on lisaväljad *usedTimes* ja *usedTotal*, millesse hakkame salvestama vastavalt antud teisenduse kasutamise kordade arvu ning kui palju oleks antud teisendust kasutada saanud.

Teisenduste kaalude õppimiseks kasutatavad sõned loeme sisse failist *õppeSõnedeFail*. Failis peavad olema kirjeldatud sõnede paarid kujul:

esialgne_sõne:*saadav_sõne*

Iga failis *õppeSõnedeFail* oleva paari puhul arvutame nende vahelise üldistatud teisenduskauguse. Kui fail on läbi vaadatud, siis töötame läbi kõik teisenduste prefiksipuud. Leiame teisendustele kaalud, kasutades valemeid (2) ning kirjutame tulemuse faili.

Leiame kaalud peatükis 6 defineeritud vene keele teisenduste jaoks. Moodustame 100 paariga õppesõnede faili (*6ppeSõnad.txt*). Kontrollimiseks moodustame kaks 500 sõnega testfaili (*testLeftS6nad.txt* ning *testS6nad.txt*), milledest esimeses on ladina tähtedega kirjutatud venekeelsed sõnad ning teises nendele vastavad kirillitsa tähtedega kirjutatud sõnad. Vaatame kui paljudele esimeses failis olevatele sõnedele leitakse parimaks tulemuseks temale vastav sõne. Võrdleme sama tulemust ka juhuga, kus teisendustele olid antud suvalised kaalud. Mõlema teisenduste faili puhul leiti 494 korral 500-st parimaks tulemuseks õige sõne. Seega õnnestumise protsent oli 98.

Leiame, mitmel korral jäi leitud parimate tulemusete teisenduste hind alla 1. Antud juhul andis programm mõlemal juhul tulemuseks jälle sama palju sõnu. Tulemusi, mille kaal jäi alla 1, oli 426. Seega 85% kordadest oli parima tulemuse üldistatud teisenduskaugus väiksem või võrdne ühega. Antud katsete tulemused on ülevaatlikumal kujul joonisel 8.2.

	katse1	katse1 õnnestumise %	katse2	katse2 õnnestumise %
treenitud	494	98	426	85
treenimata	494	98	426	85

Joonis 8.2. Programmi väljundi võrdlemine kasutades treenitud ning treenimata teisenduste kaale.

Üldistatud teisenduskauguse puhul on teiseks suurimaks probleemiks teisenduste defineerimine. Suhteliselt lihtne oleks mingite näidete hulga pealt genereerida ühetähelisi teisendusi – tavalisi teisendusi. Keerulisemaks läheb aga teisendustega, kus teisenduse paremas või vasakus pooles on rohkem kui üks tähemärk. Antud töö edasiarenduseks olekski teisenduste automaatne genereerimine mingi näidete hulga pealt.

Kokkuvõte

Sõnede vahelise sarnasuse mõõtmist kasutatakse väga paljudes rakendustes – erinevates otsingusüsteemides, mustrite sobitamises, pakkimisalgoritmides jne. Kõige lihtsam viis kahe sõne vahelise sarnasuse määramiseks on nende vahelise teisenduskauguse leidmine. Võimalused, mida pakub tavaline teisenduskaugus, pole alati piisavad, kuna selle leidmisel eeldatakse, et kõik võimalikud teisendused toimuvad sama tõenäosusega.

Autori semestritöös uurisime teisenduskauguse leidmise üldistamist juhule, kus teisendustele saab anda erinevaid kaale. Nii tavalise kui ka üldistatud teisenduskauguse puhul vaatlesime nende leidmise võimalusi kasutades dünaamilise programmeerimise tehnikat ning graafis lühimate teede leidmise algoritmi – eesmärgiks oli näidata, et üldistatud teisenduskaugust on võimalik leida kasutades tavalise teisenduskauguse algoritmi, mida on natuke täiendatud, et see suudaks arvestada ka lisateisendusoperatsioonidega.

Bakalaureusetöös lisasime üldistatud teisenduskauguse programmile Unicode toe. Samuti optimeerisime antud programmi, lisades otsitavad sõned prefiksipuu andmestruktuuri. Töö käigus uurisime ka teisendustele kaalude automaatselt genereerimist, valmis ka programm teisendustele näidete hulga abil kaalude leidmiseks.

Valminud programmid on vormistatud bakalaureusetöö lisadena, mis asuvad tööga kaasas oleval CD-l.

Töö edasiarenduseks võiks uurida üldistatud teisenduskaugusele automaatselt teisenduste genereerimist.

Summary

Generalized edit distance

Bachelor Thesis (10 cp)

Reina Käärik

Abstract

There are lots of applications that use string comparing – for instance information retrieval, pattern matching, data compression algorithms etc. The simplest way to measure the similarity between two strings is to use edit distance, also known as Levenshtein distance. But the opportunities that edit distance offers, are not always sufficient. The edit distance algorithm expects that all transformations have the same weight.

In this paper we have studied the generalized edit distance – the edit distance that allows to define additional edit operations, each having possibly different cost. We studied the possibilities to find the edit distance and the generalized edit distance using the shortest path finding algorithm in graphs and dynamic programming method. Also we studied the possibilities to improve generalized edit distance by adding Unicode support and optimizing the algorithm. Additionally we developed a method to find automatically edit costs for transformations.

The main outcome of this paper is a tool for calculating the generalized edit distance that supports UTF-8 encoding. Edit operations are managed in a *trie* datastructure. This makes the edit distance calculation faster. We have also developed a method for finding weights for transformations using example data set.

Viited

[BLV03] A. Buldas, P. Laud, J. Villemson. Graafid. TÜ Kirjastus, 2003

[Dam64] Fred J. Damerau. A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM 7(3):171-176, 1964

[Erelt05] Tiiu Erelt. Eesti ortograafia:19-23, Eesti Keele Sihtasutus, 2005

[HL92] Dzung T. Hoang, Daniel P. Lopresti. FPGA Implementation of Systolic Sequence Alignment, 1992

[Kih03] Jüri Kiho. Algoritmid ja andmestruktuurid. TÜ Kirjastus, 2003

[LH01] Thomas A. Lasko, Susan E. Hauser. Approximate String Matching Algorithms for Limited-Vocabulary OCR Output Correction. Documnet Recognition and Retrieval VIII, San Jose, CA, January 2001, 232-40

[Nav01] Gonzalo Navarro. A Guide Tour to Approximate String Matching. ACM Computing Surveys 33(1):31-88, 2001

[Ped00] Christian N. S. Pedersen. Algorithms in Computational Biology, 2000

[Vil02] Jaak Vilo. Pattern Discovery from Biosequences. PhD Thesis, 2002

[Wag74] Robert A. Wagner. The String-to-String Correction Problem. Journal of the Association for Computing Machinery 21(1):168-173, 1974

[WM92] Sun Wu, Udi Manber. Agrep A Fast Approximate Pattern-Matching Tool. Proceedings USENIX Winter 1992 Technical Conference

URL-id

[url:ASCII] ASCII, Wikipedia
<http://en.wikipedia.org/wiki/ASCII>

[url:ASCII1] ASCII Character Set
<http://www.robelle.com/library/smugbook/ascii.html>

[url:BT] Paul E. Black and Paul J. Tanenbaum, "graph", from Dictionary of Algorithms and Data Structures, Paul E. Black, ed., NIST.
<http://www.nist.gov/dads/HTML/graph.html >

[url:DP] Dynamic Programming, Wikipedia
<http://en.wikipedia.org/wiki/Dynamic_programming>

[url:char] A tutorial on character code issues
<http://www.cs.tut.fi/~jkorpela/chars.html>

[url:PuTTy] PuTTy: a free telnet/ssh client
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

[url:TA] Tekstialgoritmide loengumaterjal. Loeng 3 – sõnede sarnasus ja teisenduskaugus, 2003
<http://www.egeen.ee/u/vilo/edu/2002-03/Tekstialgoritmid_I/Loengud/Loeng3_Edit_Distance/>

[url:trie1] Yehuda Shiran, Ph.D. The trie Data Structure.
<http://www.webreference.com/js/tips/000318.html>

[url:trie2] PlanetMath, Trie
<http://planetmath.org/encyclopedia/Trie.html>

[url:unic1] Unicode Charts
<http://www.unicode.org/charts/>

[url:unic2] Multi-lingual text on Linux
<http://www.jw-stumpel.nl/stestu.html>

[url:unic3] Characters vs.Bytes
<http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF>

[url:unic4] The Absolute Minimum Every Software Developer Absolutely, Positivelly Must Know About Unicode and Character Sets
<http://www.joelonsoftware.com/articles/Unicode.html>

Lisad

Bakalaureusetööga on kaasas CD, millel on töö käigus valminud programmid ning kasutatud näitefailid. CD peal asuvad lisad on organiseeritud allpool kirjeldatud viisil. Samuti on igas kaustas olemas fail README.txt, mis kirjeldab kaustas olevat sisu ning programmide kaustade puhul ka õpetab, kuidas antud kaustas olevat programmi käivitada. Kõik programme sisaldavad kaustad sisaldavad ka *shelli* skripti keskkonna seadistamiseks kasutama UTF-8 kodeeringut. Samuti on programmide kaustas olemas *Makefile* programmide mugavaks kompileerimiseks.

Lisa 1
Üldistatud teisenduskauguse programm Unicode toega

Lisa 2
Üldistatud teisenduskauguse programm, mis kasutab sõnede jaoks prefiksipuu andmestruktuuri

Lisa 3
Programm kaalude leidmiseks

Lisa 4
Antud töös kasutatud sõnede failid