TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Tarkvarasüsteemide õppetool
Informaatika eriala

Reina Käärik

ÜLDISTATUD TEISENDUSKAUGUSE ARVUTAMINE

Semestritöö (5AP)

Juhendaja: Jaak Vilo, PhD

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

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

Õppetooli juhataja:....."...." 2006

TARTU 2006

Sisukord

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 juba uuritud pikka aega. Juba 1964. aastal 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 seda algoritmi aastal 1965 Vene teadlane Vladimir Levenshtein, kelle järgi sai see algoritm ka oma nime. Teisenduskaugust võib vaadelda kui teisendusoperatsioonide arvu, mis tuleb teha, et saada esialgsest sõnest teine.

Sageli võimalused, mida meie pakub tavaline teisenduskaugus, ei ole piisavad – selles arvestatakse, 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 asendamisega teisega või väikse ploki kõrvutiasetsevate nukleotiidide lisamine või kustutamine. Vähem sageli 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]. Või kui vaadelda näiteks eesti keeles ajalooliselt toimunud muutusi: uurides näiteks 18. sajandist pärinevaid eesti keelseid 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* või kahekordse täishääliku asemel kirjutatud ühekordne täishäälik – *kulus*, *sowib* (praeguses kirjapildis *kuulus* ning *soovib*) samas mitte kunagi ei esine näiteks k tähe kahekordselt kirjutamist. 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*. Kui otsiksime vanaaegsetele sõnadele vasteid, siis sooviksime, et just neist esimene oleks antud sõnale lähedane. Samuti võib vaadelda ka inimeste poolt trükkimisel tehtavaid vigu – olenevalt klaviatuuri asetusest on sagedasemad nende tähtede asendamine, mis asuvad arvuti klaviatuuril teineteisega lähestikku – suurema tõenäosusega on vigaselt kirjutatud sõnas asendatud a täht s-ga kui näiteks ü-ga. Ühtedeks sagedasemateks vigadeks on ka grammatikaga seotud vead – näiteks g asendamine k-ga esineb palju suurema tõenäosusega kui g asendamine kk-ga või isegi r-ga. Omamoodi probleem tekib ka eesti keelsetest 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.

Käesolev töö on jaotatud viieks peatükiks. Teises peatükis on kirjeldatud ja seletatud käesolevas töös tihedamalt kasutatud mõisteid. Kolmandas peatükis on kirjedatud Levenshteini teisenduskaugust ja selle leidmise võimalusi. Neljandas peatükis kirjeldame üldise teisenduskauguse leidmist, mis arvestab ka erinevate tähtede/täheühendite lisamiste, asendamiste ja kustutamiste jaoks erinevaid kaale. Viiendas peatükis on kirjeldatud antud semestritöö käigus valminud programmi üldistatud teisenduskauguse leidmiseks.

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 lõppeb positsioonil *j*. Tähistame selle alamsõne *S*[*i*...*j*], kus $1 \leq i \leq j \leq |S|$ [Vil02].

2.2 Dünaamiline programmeerimine

Dünaamiline programmeerimine on algoritmiline tehnika, mis tavaliselt baseerub tagasiulatuval valemil ja ühel või mitmel algseisundil. Algne probleem jagatakse väiksemateks alamprobleemideks ning proovitakse leida parim lahendus neile alamprobleemidele. Parimad lahendused suurematele alamprobleemidele leitakse kasutades väiksemate alamprobleemide parimaid lahendusi [url:DP].

2.3 Graaf, suunatud graaf

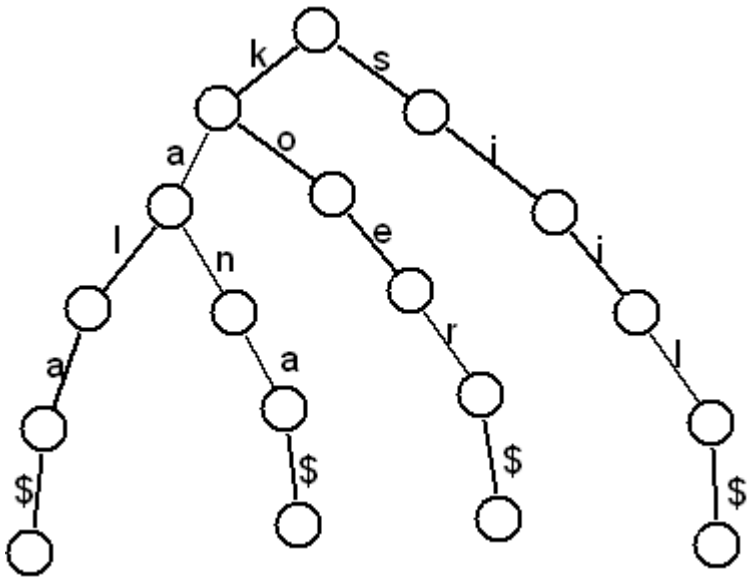
Graafiks G nimetatakse paari (*V, E*), kus V on tippude hulk ja *E* on servade hulk tippude vahel $E = \{ (u, v) \mid u, v, \in V \}$ [url:BT]. Tavaliselt esitatakse graaf joonisena,kus iga tipp on kujutatud punktina tasandil ning iga serv kahte tippu ühendava joonena [BLV03]. Graafi, mille servadele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

2.4 Puu ja trie 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].

Trie on järjestatud puu sõnede talletamiseks. Sõna *trie* tuleb inglise keelsest sõnast *retrieval* (e.k. otsing), see päritolu vihjab tema kasutamisele infootsingus. *Trie* 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 *trie* 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 *trie*-st taastada puu sügavuti läbimise algoritmi kasutades.

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



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

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

Sõne $A=a_1a_2...a_m$ *trie*-st otsimiseks kuluv aeg on halvimal juhul *O*(*m*). 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 kuniks ühegi vaatluse all olevat tippu ja selle alluvat ühendava kaare märgend ei klapi hetkel vaatluse all oleva tähega või kui sobitasime sõne viimase tähe ning vaatluse all oleva tipust ühegi väljuva kaare märgendiks ei ole sõne lõppu märkiv sümbol – järelikult sellist sõnet antud *trie*-s 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.

Algoritm 2.1. Sõne S otsimiseks *trie*-st T

Sisend: Otsitav sõne S = $s_1 s_2 \ldots s_n$ ja trie T

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

Meetod:

1. $T_{juur} = juur(T)$ // võtame vaatluse alla T juurtipu

// 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;**
6. **else if** $kaar(T_{juur}, T_i).märgend = sõne[täht]$ **then** //saame *trie*-s allapoole liikuda
7. $T_{juur} = T_i$
8. täht++
9. **return false** // vaatasime läbi kõik konkreetse tipu vahetud alluvad või vaadeldav tipp oli leht

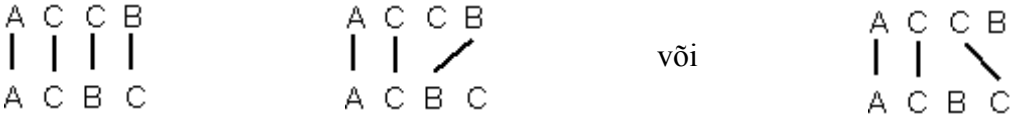
3. Peatükk
Levenshteini kaugus

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 tavaliselt 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, mis tuleks teha esimese muutmiseks teiseks. Selleks võiks näiteks kustutada sõne *ACCB* kaks viimast tähte ja lisada seejärel lõppu tähed *B* ja *C* või näiteks asendada lihtsalt sõne lõpus oleva *C* tähe *B*-ga ning sõne lõpus oleva *B* tähe *C*-ga. 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 – 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].

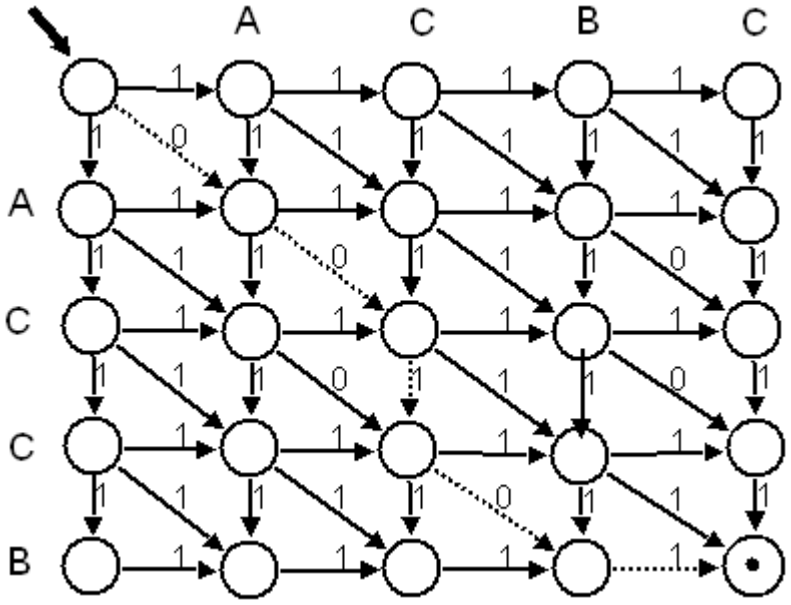
A C C B	A C C B –	A C C B –
A C B C	A C – B C	A – C B C

3. Operatsioonide loendina

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 graafis lühima (väikseima kaaluga) tee leidmist. Võtame vaatluse alla eelmises näites olnud sõned S_1 = ACCB ja S_2 = ACBC. Parema ülevaate saamiseks esitame graafi järgmisel kujul:



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 iga liikumist paremale, võib 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 graafis selle kaare kaal 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 $S[1]...S[j]$. Vaadates antud graafi paigutust, i tähistab 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 alumises tipus (joonisel 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 hulka ja $|E|$ tähistab graafi kaarte hulka.
- 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 $O((\mathrm{n} + \mathrm{m})\ \log(n)\ )$, kus $n=|V|$ ja $m=|E|$ [Kih03].

3.2 Teisenduskauguse arvutamine kasutades dünaamilise programmeerimise tehnikat

Teisenduskaugust kahe sõne vahel võib ka arvutada kasutades dünaamilise programmeerimise tehnikat. Defineerime rekurrentse valemi kahe sõne $A=a_1a_2...a_m$ ja $B=b_1b_2...b_n$ vahelise teisenduskauguse leidmiseks, tähistame $d_{i,j}=D(a_1a_2...a_i, b_1b_2..b_j), 0\le i\le m, 0\le j\le n$, $d_{i,j}$ tähistab vähimat teisendusoperatsioonide arvu, mis kulub selleks, et teisendada alamsõne $a_1a_2...a_i$ alamsõneks $b_1b_2...b_j$.

Minimaalselt teisendusoperatsioonide arvu arvutame järgmiselt:

$$d_{i,0}=i, 0\le i\le m$$

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

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

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

Väärtust $d_{|A|,|B|}=d_{m,n}$ nimetatakse teisenduskauguseks sõnede A ja B vahel. Ülaltoodut valemit võiks selgitada järgnevalt: alustuseks, $d_{i,0}$ ja $d_{0,j}$ kujutavad teisenduskaugust sõnede pikkusega vastavalt *i* ja *j* ning tühja sõne vahel. Täpsemalt, *i* kustutamist (ja vastavalt *j* lisamist) on selleks vaja teha. Kahe mittetühja sõne puhul, pikkusega *i* ja *j* eeldame, et kõik teisenduskaugused neist lühemate sõnede puhul on juba arvutatud ning proovime teisendada sõnet $a_1a_2...a_i$ sõneks $b_1b_2...b_j$. Võtame vaatluse alla nende sõnede viimased tähed – a_i ja b_j. Kui need tähed on võrdsed, $a_i=b_j$, siis pole neid vaja enam vaatluse alla võtta, jätkame lihtsalt parimat teed, mille leidsime $a_1a_2...a_{i-1}$ teisendamiseks sõneks $b_1b_2...b_{j-1}$. Paneme tähele, et sellisel juhul kulub sõne $a_1a_2...a_i$ teisendamiseks sõneks $b_1b_2...b_i$ sama palju teisendusoperatsioone, kui sõne $a_1a_2...a_{i-1}$ teisendamiseks sõneks $b_1b_2...b_{j-1}$. Kui aga need tähed ei ole võrdsed, peame vaatlema kolme lubatud teisendusoperatsiooni – me võime kustutada tähe a_i ning kasutada parimat teisendust sõne $a_1a_2...a_{i-1}$ muutmiseks sõneks $b_1b_2...b_j$, võime lisada tähe b_j sõne $a_1a_2...a_i$ lõppu ning kasutada parimat teisendust sõne $a_1a_2...a_i$ teisendamiseks sõneks

$b_1 b_2 ... b_{j-1}$ või asendada täht 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 sõnade vahel.

		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: $A = a_1 a_2 .. a_m$, $B = b_1 b_2 ... b_n$

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

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** $d_{i,j}$

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 $d_{m,n}$ -st $d_{0,0}$ -ni, et näha mis teed pidi minimiseerimine $d_{i,j}$ arvutamisel kulges. Selleks genereerime hulgad *pred*[*i, j*] – tähistamaks millisest ruudust tuldi ruutu $d_{i,j}$.

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

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

$0 \le i \le m$, $0 \le j \le n$. Teisenduse järjekorra taastamiseks tuleb liikuda mööda hulkasid *pred*[*i, j*] kuni $d_{0,0}$ -ni [url:TA]. Ilmestamiseks toome joonise sõnede *kallur* ja *kallan* jaoks arvutatud dünaamilise programmeerimise tabeli abil taastatud võimalikud lühimad teisendusteed.

		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 taasatamist 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 tabelis antud välja kohal asuva välja kohta – sinna saame samuti liikuda vaid juhul, kui sellel olev arv on väiksem kui hetkel vaatluse all oleval väljal olev 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. Seega 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 üldised lisamisele, kustutamisele ja asendamisele antud 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 teisendustakauguse puhul.

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

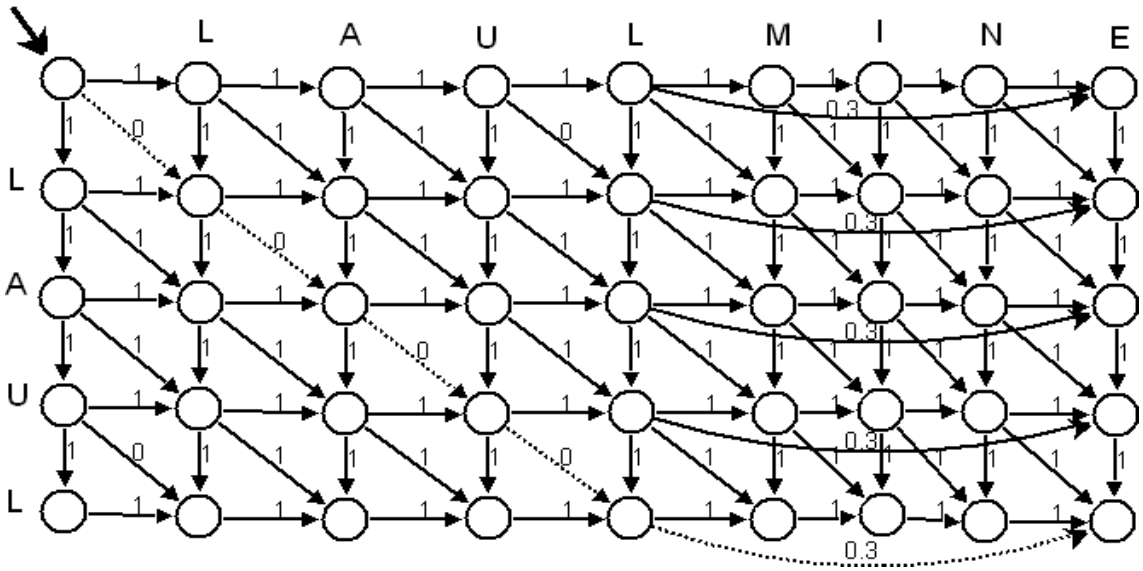
$\lambda \rightarrow mine$, millele anname näiteks hinnaks 0.3.

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:

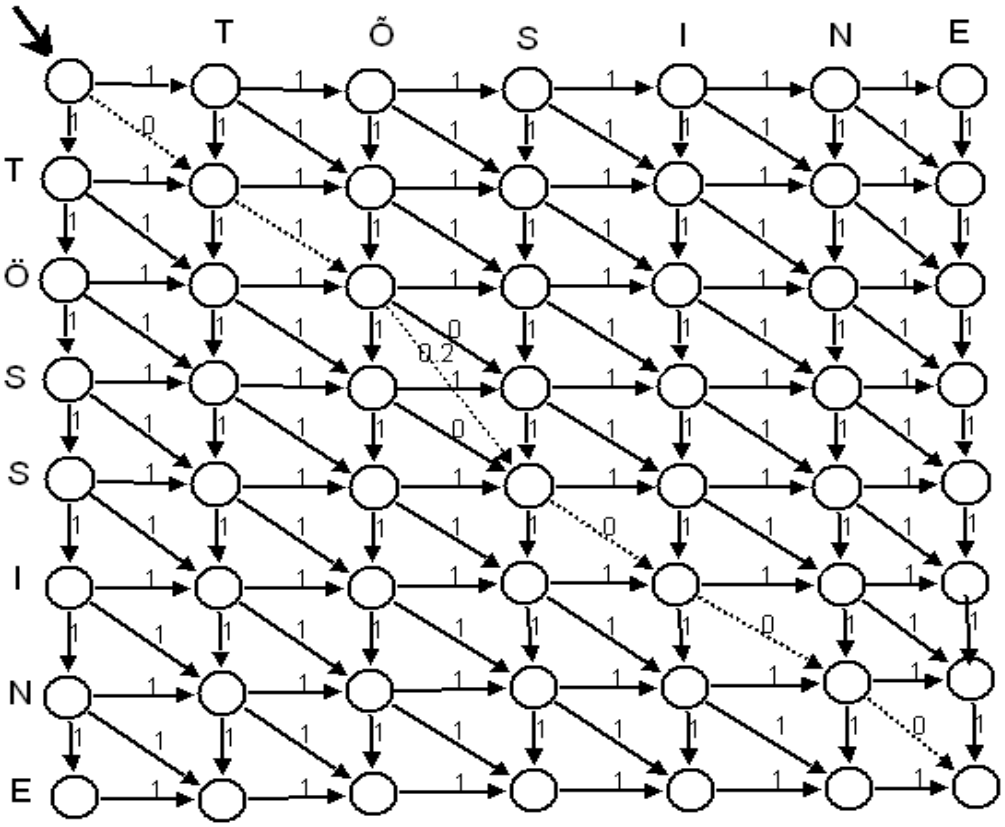


Joonis 4.1. Sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse leidmine, kui lisaks tavalistele teisendusoperatsioonidele lisaks on defineeritud lõpu *mine* lisamine, kaaluga 0.3.

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

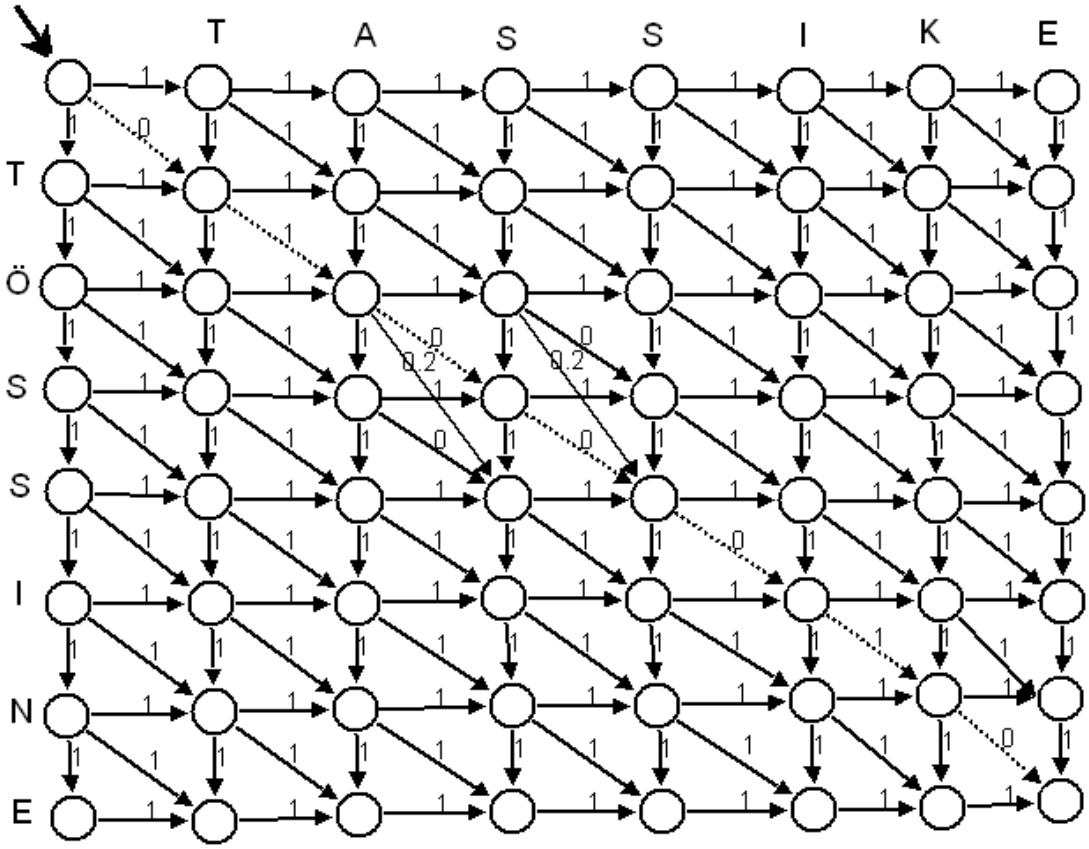
Näide 4.3. Tahame leida sõnede *tössine* ja *tõsine* ning *tössine* ja *tassike* vahelisi üldistatud teisenduskaugusi. Olgu antud lisaks teisendus $ss \to s$ kaaluga 0.2.

Esitame antud üldistatud teisenduskauguse leidmise probleemid graafi kujul. Esimesel juhul üldistatud teisenduskauguse graaf oleks järgmine:



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

Üldistatud teisenduskaugus sõnede *tössine* ja *tõsine* vahel on 1.2. Ülaloleval joonisel 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 $\ddot{o} \rightarrow \tilde{o}$, saame sõnet *tössine* veelgi lähendada sõnele *tõsine*.

4.2 Üldistatud teisenduskauguse leidmine kasutades dünaamilise programmeerimise tehnikat

Üldistatud teisenduskaugust saab samuti leida kasutades dünaamilise programmeerimise tehnikat. Olgu antud sõned $A=a_1a_2...a_m$ ja $B=b_1b_2...b_n$. Defineerime rekurrentse valemi üldistatud teisenduskauguse $d'_{i,j}=D'(a_1a_2...a_i, b_1b_2...b_j), 0\le i\le m, 0\le j\le n$ leidmiseks:

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

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

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

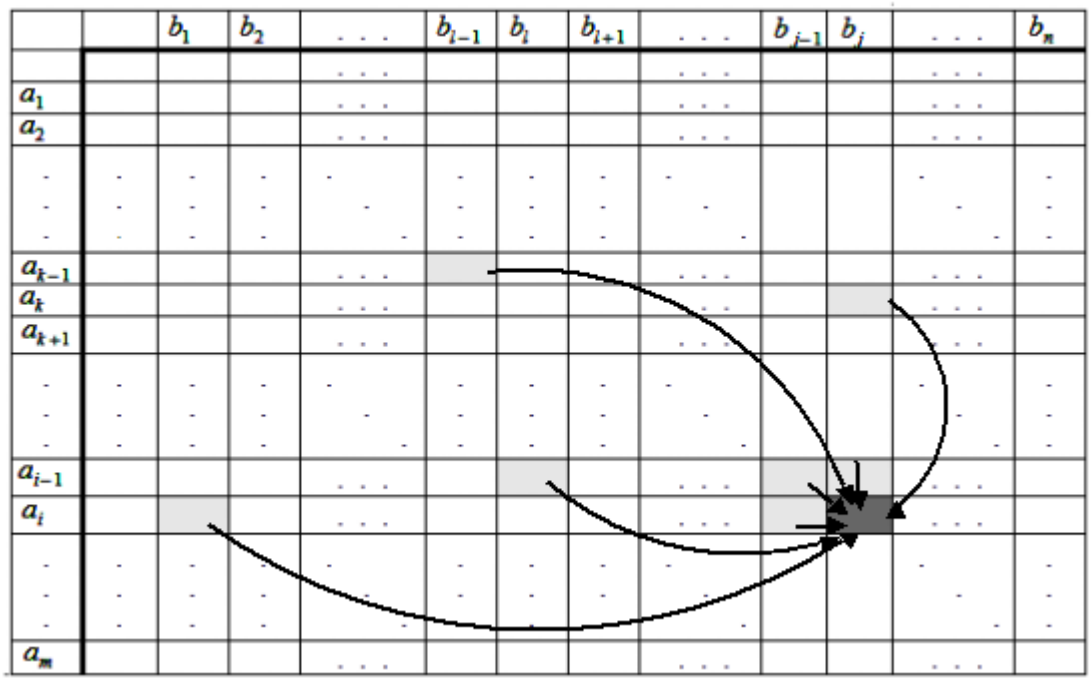
$$d'_{i,j}=min\begin{cases} d'_{i-1,j-1}+(if\ a_i=b_j\ then\ 0\ else\ 1) \\ d'_{i-1,j}+1 \\ d'_{i,j-1}+1 \\ d'_{k-1,l-1}+w, \quad kui\ leidub\ asendus:\ \ a_k...a_i\to b_l...b_j\ \ kaaluga\ w \\ d'_{k-1,j}+w, \quad kui\ leidub\ teisendus:\ \ a_k...a_i\to\lambda\ \ kaaluga\ w \\ d'_{i,l-1}+w, \quad kui\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

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

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 teisenduskaugust sõne vastavalt pikkusega i ja j ning tühja sõna vahel. Tähti sõnesse võib lisada ükshaaval või grupiviisi – kui selline teisendus on defineeritud. Vaatleme kahte mittetühja sõnet $A=a_1a_2...a_i$ ja $B=b_1b_2...b_j$, 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 ning 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 $a_1a_2...a_{i-1}$ teisendamiseks alamsõneks $b_1b_2...b_{j-1}$ ning sinna lõppu lihtsalt lisada

täht a_i. Kui aga $a_i \neq b_j$, 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 iga välja (*i, j*) täitmisel peame 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 -ks või β -ks olla ka tühisõne, sellisel juhul vaatleme seda teisendust kui lisamis- või kustutamisoperatsiooni.



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

Joonisel 4.4 on kujutatud tabeli välja (*i, j*) täitmist. Hetkel kasutatavateks teisendusteks on sõnest *A* *i*-nda tähe kustutamine – $a_i \to \lambda$, sõne *B* *j*-nda tähe lisamine – $\lambda \to b_j$, sõne *A* *i*-nda tähe asendamine sõne *B* *j*-nda tähega ning lisateisendusoperatsioonid – $a_k \ldots a_i \to b_l \ldots b_j$, $a_{k+1} .. a_i \to \lambda$, $a_i \to b_{l+1} \ldots b_j$, ning $\lambda \to b_1 \ldots b_j$. 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 $\{ (i-1, j)+w_1,\ (i-1, j-1)+w_1,\ (i, j-1)+w_1,\ (k-1, l-1)+w_2,\ (k, j)+w_3,\ (i-1, l)+w_4,\ (i, 1)+w_5 \}$.

Näide 4.4. Leiame sõnede *laul* ja *laulmine* vahelise üldistatud teisenduskauguse, kui peale tavaliste teisendusoperatsioonide on veel defineeritud teisendus $\lambda \to mine$ hinnaga 0.3.

		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**

Joonis 4.5. 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 $ss \to s$ kaaluga 0.2. Vastavad dünaamilise programmeerimise meetodi abil arvutatud üldistatud teisenduskauguse tabelid oleksid:

		t	õ	s	i	n	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
õ	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*

// ülejäänu tabeli täitmine

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

// täidame esimese rea

7. $d'_{0,j}$ = **min** (

8. j,

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

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

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

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

Selle algoritmi puhul iga välja täitmisel vaadatakse terve teisenduste list 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.

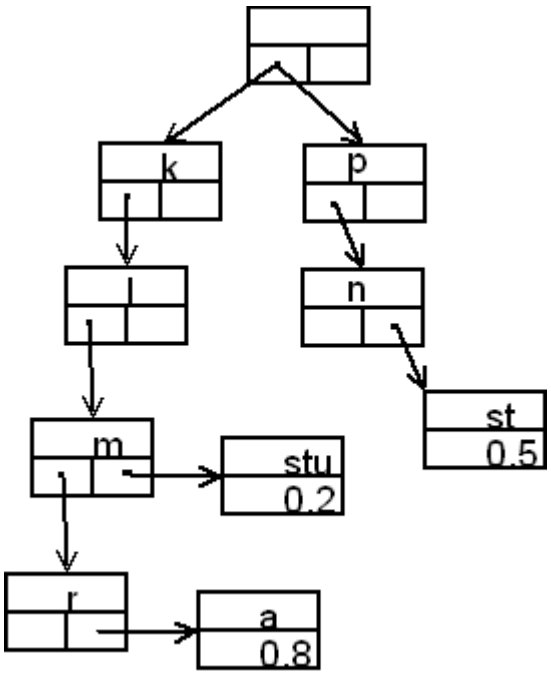
Lahenduseks oleks teisenduste hulga organiseerimine *trie* andmestruktuuri abil.

4.2.1 Teisenduste hulga organiseerimine *trie* abil

Antud semestritöö üheks põhilisemaks tulemuseks ongi teisenduste hulga viimine kompaktsemale kujule ning sobivate teisenduste otsimise kiirendamine. Jaotame teisendused kolme hulka – asendused, kustutamised ja lisamised ning teeme igaühele neist omaette *trie*. Kuna *trie*-d oleks raske organiseerida nii teisenduste vasaku kui ka parema poole järgi, siis organiseerime teisenduse $t_1 \to t_2$ *trie*-sse näiteks t_1 järgi, lisamisoperatsioonid t_2 järgi kuna nende puhul on $t_1 = \lambda$.

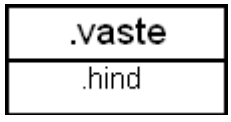
Arvutis *trie* kujutamiseks on mõttekam kaarte märgendites olev informatsioon salvestada tippudesse ning kaari kujutada kui viitasid tippude vahel. Igasse *trie* vahetippu, mis asub juurtipust kaugusel k salvestame märgendi $t_1[k]$. Lehttipu sisse salvestame sõnelõppu tähistava *$* asemel aga hoopis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

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



Tipu formaat:

Lehttipu formaat:



Joonis 4.8. Teisenduste kujutamine *trie* andmestruktuuris.

Sisetipu puhul hoiame 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.

Algoritm 4.2. Teisenduste hulga lisamine *trie* andmestruktuuridesse

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

Väljund: Lisamise, kustutamise ja asendamise *trie*-d: $T_{lisamised}$, $T_{kustutamised}$ ning $T_{asendamised}$.

Meetod:

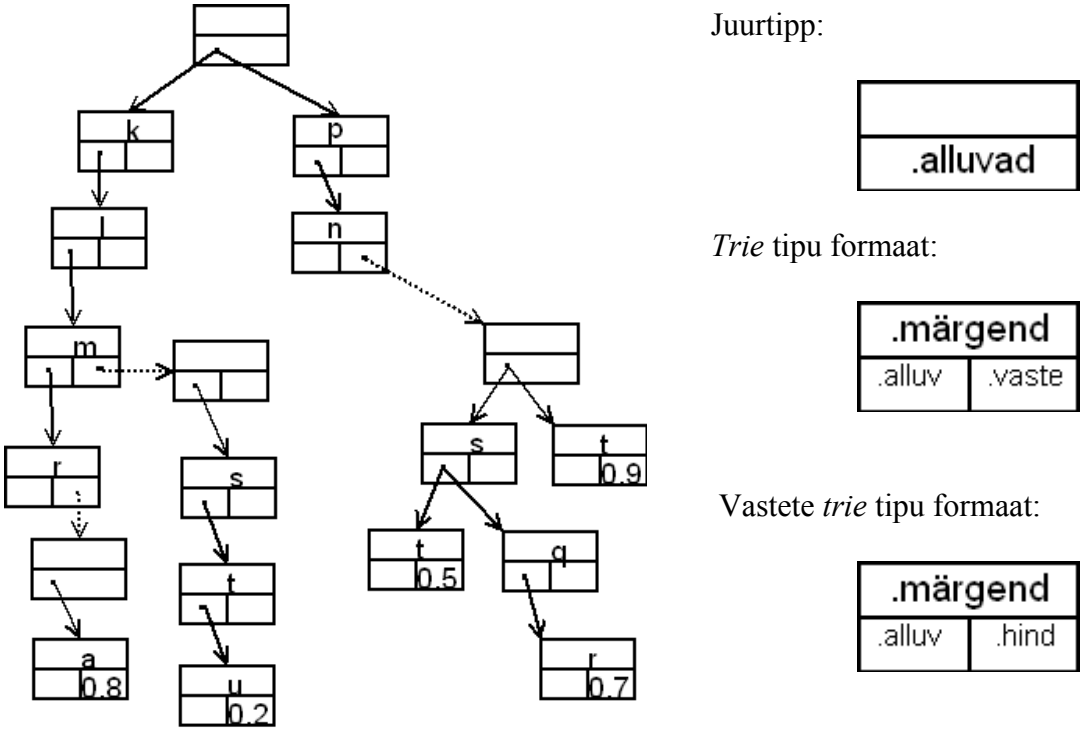
1. **for** i = 1 **to** $|T|$ **do**
2. **if** t_i = lisamisoperatsioon **then** lisaTriesse(t_i, $T_{lisamised}$)
3. **else if** t_i = kustutamisoperatsioon **then** lisaTriesse(t_i, $T_{kustutamised}$)
4. **else** lisaTriesse(t_i, $T_{asendamised}$)

// meetod teisenduse $\alpha \rightarrow \beta$ *lisamiseks trie-sse, kui* $\alpha = \lambda$ *või* $\beta = \lambda$, *siis organiseerime sõne //Trie-sse vastavalt ainult* β *või* α *järgi ning lehttippu salvestame vaid teisenduse kaalu.*

5. lisaTriesse($\alpha \rightarrow \beta$, Trie)
6. **while**(Trie-s leidub prefiks $\alpha[1]...\alpha[k]$)
7. liigu Trie-s allapoole
8. lisa Trie-sse teisenduse lõpp $\alpha[k+1]...\alpha[|\alpha|]$
9. lisa Trie-sse teisenduse parem pool β ning teisenduse kaal $w_{\alpha \rightarrow \beta}$

Kui leiduvad teisendused $t_1 \rightarrow t_2$ ning $t_1 \rightarrow t_3$, st. sõnet t_1 on võimalik asendada nii sõnega t_2 kui ka sõnega t_3 , seega oleks asenduste *trie*-s 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 *trie* andmestruktuuridesse.

Näide 4.7. Olgu antud teisendused $klm \rightarrow stu$ kaaluga 0.2, $klmr \rightarrow a$ kaaluga 0.8, $pn \rightarrow st$ kaaluga 0.5, $pn \rightarrow sqr$ kaaluga 0.7 ning $pn \rightarrow t$ kaaluga 0.9. Järgnev joonis illustreerib antud teisenduste hulga kujutamist kui ka teisenduste paremad pooled on organiseeritud *trie*-na.



Joonis 4.9. Teisenduste puu kujutamine, lisades teisenduse vaste omaette *trie*-sse.

Juurtipul on 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 *trie*-le. Vastete *trie* puhul on juurtipp sama formaadiga kui teisenduste vasakute poolte *trie*-s, 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 *trie*-dele 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 *trie*-sse lisame neist vaid esimene – kaalude võrdsuse puhul oleks tegemist lihtsalt täpselt sama teisendusega. Juhul kui $w_1 < w_2$, kaaluga w_2 teisendust me kunagi ei kasutaks – teisenduseks valime alati vähima kaaluga võimaliku teisenduse.

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

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: Teisenduste *trie*-sse organiseerimiseks kasutame algoritmi 4.2

1. organiseeri teisenduste hulk T *trie*-ss
2. $d'_{0,0}=0$

// täidame esimese veeru

3. **for** i = 1 **to** m **do**
4. otsi kustutamiste *trie*-st teisendusi $a_i...a_k\to\lambda$, $i\le k\le m$
5. $d'_{k,0}=d'_{i-1,0}+w_{a_i...a_k\to\lambda}$
6. $d'_{i,0}$ = **min**($d'_{i-1,0}+w_{kustutamine}$, $d'_{i,0}$)

//*täidame ülejäänud tabeli*

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

// täidame esimese rea

8. otsi lisamise *trie*-st teisendusi $\lambda\to b_j...b_l$, $j\le l\le n$
9. $d'_{0,l}=d'_{0,j-1}+w_{\lambda\to b_j...b_l}$
10. $d'_{0,j}$ = **min** ($d'_{0,j-1}+w_{lisamine}$, $d'_{0,j}$)
11. **for** i = 1 **to** m **do**
12. otsi kustutamiste *trie*-st teisendusi $a_i...a_k\to\lambda$, $i\le k\le m$
13. $d'_{k,j}=d'_{i-1,j}+w_{a_i...a_k\to\lambda}$
14. otsi lisamise *trie*-st teisendusi $\lambda\to b_j...b_l$, $j\le l\le n$
15. $d'_{i,l}=d'_{i,j-1}+w_{\lambda\to b_j...b_l}$
16. otsi asenduste *trie*-st teisendusi $a_i...a_k\to b_j...b_l$, $i\le k\le m$, $j\le l\le n$
17. $d'_{k,l}=d'_{i-1,j-1}+w_{a_i..a_k\to b_j...b_l}$
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** $d'_{i,j}$

Enne tabeli iga välja (i, j) täitmist vaatame, kas leidub lisakustutamisoperatsioone $a_i \ldots a_k$, $i \le k \le m$. Iga sellise teisenduse puhul lisame antud teisenduse hinna tabelisse kohale (k, j) lisame väärtuse $min\{(k, j), (i-1, j) + w_{a_i \ldots a_k \to \lambda}\}$. Sama teeme läbi ka lisaasendus- ning lisamisoperatsioonidega. Esimese puhul, kui saame kasutada mõnda juurdedefineeritud lisamisoperatsiooni sõne $a_i \ldots a_k$ teisendamiseks sõneks $b_j \ldots b_l$, $i \le k \le m$, $j \le l \le n$, siis väljale (k, l) kirjutame väärtuse $min\{(k, l), (i-1, j-1) + w_{a_i \ldots a_k \to b_j \ldots b_l}\}$. Ning teisel juhul – kui leidub lisateisendus $\lambda \to b_j \ldots b_l$, $j \le l \le n$, siis väljale (i, l) kirjutame väärtuse $min\{(i, l), (i, j-1) + w_{\lambda \to b_j \ldots b_l}\}$. 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 esimesed kolm on tavalised teisendusoperatsioonid, väljal (i, j) on aga kirjas vähim lisateisendusoperatsiooni kasutamise hind, mida saab kasutada alamsõne $a_k \ldots a_i$ teisendamiseks alamsõneks $b_l .. b_j$, $1 \le k \le i$, $1 \le l \le j$.

Üldistatud teisenduskauguse leidmiseks kasutades *trie*-sid võib kasutada ka algoritmi versiooni, kus iga tabeli välja väärtuse leidmiseks leitakse seal lõppevatest teisendustest vähim. Sellisel juhul organiseerides teisendusi organiseerida *trie*-sse alustades viimasest sümbolist.

4.2.2 Üldistatud teisendustakauguse dünaamilise programmeerimise tabelist teisenduste taastamine

Nagu ka tavalise teisenduskauguse puhul, on võimalik üldistatud teisenduskauguse dünaamilise programmeerimise tabelist taastada võimalikud minimaalsed teisendusteed. Võimalike teisendusteede jaoks enam ei piisa lihtsalt üles, vasakule ja diagonaalis ülevale jääva välja vaatamisest, neile lisaks 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* $d'_{i,j} = d'_{i-1,j-1} + (\text{if } a_i = b_j \text{ then } 0 \text{ else } 1)$
- $(i-1, j)$, *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 $a_k \ldots a_i \to b_l \ldots b_j$ kaaluga *weight* ning $d'_{i,j} = d'_{i-k-1,j-l-1} + weight$, $1 \le k \le i$, $1 \le l \le j$
- $(i-k-1, j)$, kui leidub teisendus $a_k .. a_i \to \lambda$ kaaluga *weight* ning $d'_{i,j} = d'_{i-k-1,j} + weight$, $1 \le k \le i$
- $(i, j-l-1)$, kui leidub teisendus $\lambda \to b_l \ldots b_j$ kaaluga *weight* ning $d'_{i,j} = d'_{i,j-l-1} + weight$, $1 \le l \le j$

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

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

$ost \to s$ *kaaluga* 0.3 ,

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

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

$post \to essi$ *kaaluga* 0.4 ,

$os \to s$ *kaaluga* 0.5 ,

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

$er \to on$ *kaaluga* 0.9 .

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

		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.

Joonisel iga noolekest võiksime vaadelda kui hulka *pred* elemendi lisamist. Liikudes mööda hulki *pred*[*i, j*] kuni hulgani *pred*[0, 0]-ni saame taastada kõik teisendused.

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

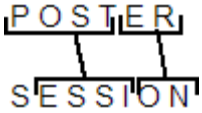
P OST E — R
— S E SSIO N

ning

— POST ER
S ESSI ON

või kasutades jälitust:

ning



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

Samuti võib teisendusi kujutada kui nende tegemiseks 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

Antud 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 võiks kirjeldada järgnevalt:

- teisendustefail – fail, milles on allpoololevate reeglite järgi kirja pandud lisateisendusoperatsioonid
- otsisõne – sõne, mida hakatakse failis 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 neid kaale pole failis defineeritud, siis kasutatakse vaikimisi kaale, milleks on 1.

Vastavalt teisenduste failis olevatele teisendustele moodustatakse kolm teisenduste *trie*-d – lisamiste, kustutamiste ja asenduste jaoks. Kasutades neid teisenduste *trie*-sid, arvutatakse üldistatud teisenduskauguse maatriks ning vastavalt sisendile kas väljastatakse saadud teisenduskaugus kui see on väiksem sisendina antud teisenduskaugusest või kogutakse parimate tulemuste listi ning pärast väljastatakse nii palju tulemusi, kui antud parameetriga *parimaidTulemusi*. Sama 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 otsisõnest on teisenduskaugusel 1 ning tahetakse tulemuses näha ainult 5 parimat tulemust, siis väljastatakse antud tulemuses kõik need 10 sõna kui parima teisenduskauguse andnud.

Näide 5.1. Tahame leida failis *kohanimed.txt* olevatest kohanimedest ainult kahe lähima teisenduskaugusega tulemusi.

```
>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. Tahame leida kohanimede failist ainult selliseid, mis on sõnest *tartu* teisenduskaugusel 1.

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


harku
1.000000


hertu
1.000000


taritu
1.000000


tartu
0.000000
```

Antud semestritöö lisades on toodud ära ka üks teisenduste fail *teisendused.txt* kohanimede sarnasuse jaoks. Teisendused on valitud järgmise loogika järgi - nime kõla jääb sarnaseks, kui nimes asendada täishäälikuid teisega või helilisi kaashäälikuid helilistega või helituid helitutega, seepärast võibki nendele teisendustele anda väiksemad kaalud. Samuti jääb kõla sarnaseks, kui mõni sulghääliku väldet muuta. Katsetused selle failiga näitasid, et kui teisenduskaugus kohanimede vahel jäi alla 2, siis võis öelda, et nende kohanimede hääldus oli sarnane.

Võrdleme antud semestritöö käigus valminud programmi agrepiga [Agrep92]. Otsides failist sõnu, mis on etteantud sõnest väikesel teisenduskauguse, programmide väljundid väga sarnased. Samas üldistatud teisenduskauguse leidmise programmi eelis agrepi ees on see, et üldistatud teisenduskauguse programm leiab sõnede sarnasust granulaarsemalt. Näiteks kui otsime kohanimede failist sõnele *kääpa* sarnaseid kohanimesid, siis üldistatud teisenduskauguse programm annab tulemused, mis on teisenduskaugusel 0, 0.5 ning 1. Agrep saab anda tulemuseks ainult kohanimed, mis on teisenduskaugusel 0 või 1.

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.

Käesolevas semestritöös uurisingi teisenduskauguse leidmise üldistamist juhule, kus teisendustele saab anda erinevaid kaale. Nii tavalise kui ka üldistatud teisenduskauguse puhul vaatlesin 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 natukene täiendatud, et see suudaks arvestada ka lisateisendusoperatsioonidega. Samuti on mõlema puhul toodud ka algoritm dünaamilise programmeerimise tabelist teisenduste järjekorra taastamiseks.

Antud töö põhilisemaks tulemuseks oli üldistatud teisenduskauguse jaoks kasutatavate teisenduste viimine kompaktsemale kujule ning sellega ka teisenduste hulgast sobivate teisenduste leidmise kiirendamine.

Semestritöö käigus valmis ka programm üldistatud teisenduskauguse leidmiseks. Teisendused antakse ette failis. Semestritöö lisas on toodud ka teisendusoperatsioonide jaoks näitefail kohanimede sarnasuse kirjeldamiseks. Samuti on lisades olemas README.TXT fail, mis õpetab kuidas seda programmi kompileerida ja käivitada.

Töö edasiarenduseks võiks uurida teisendustele kaalude leidmist ning nende kaalude rakendamist.

Summary

Generalized edit distance

Term paper

Reina Käärik

Abstract

There are lots of applications that use string comparing, for instance search engines, 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 offer are not always sufficient. The edit distance expects that all transformations have the same weight.

In this term paper we have studied the generalized edit distance - the edit distance that allows us to define additional edit operations with different costs than the default ones. 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. The main goal was to show that the generalized edit distance is very easily derivable from the usual edit distance algorithm. Also we have showed the way how to restore the edit sequences from the dynamic programming table.

The main outcome of this term paper was representing the transformations' set in a *trie* data structure. This makes the searching for the appropriate transformations faster and the transformations take less space in computer memory.

As a practical result of the paper we have made a program for finding generalized edit distance using the transformations that are given as an input file for the program. Also we have composed an example transformations' file where are described the similarity transformations for geographical names.

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

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

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

[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

[Agrep92] Sun Wu, Udi Manber. Agrep A Fast Approximate Pattern-Matching Tool. Proceedings USENIX Winter 1992 Technical Conference

URL-id

[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
<http://students.ceid.upatras.gr/~papagel/project/kef5_6.htm>

[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:TypE] Typographical error , Wikipedia.
<http://en.wikipedia.org/wiki/Typographical_error>

[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>