T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Informaatika eriala

Taivo Teder

Iteratiivse pikendamise algoritmi rakendamine ja hindamine otsinguprobleemi lahendamisel

Bakalaureusetöö (6 EAP)

Juhendaja: prof. M. Koit

Autor: ..... "...." mai 2012

Juhendaja: ..... "...." mai 2012

Lubada kaitsmisele

Professor: ..... "...." mai 2012

Tartu 2012

Sisukord

Sissejuhatus

Käesoleva töö eesmärgiks on tutvustada otsingustrateegiat – iteratiivse pikendamise algoritmi – mida on seni kirjanduses vähem käsitletud. Iteratiivse pikendamise algoritmi kui otsingustrateegia tööpõhimõtetest arusaamiseks on oluline tuua sisse mõisted otsingu (ehk ühe võimaliku probleemilahendusmeetodi) kohta ja pakkuda taustainformatsiooni tehisintellektist tuntud mõistete kohta, mis on seotud probleemilahendusega üldiselt. Antud lõputöös antakse ka põgus ülevaade kahest teisest otsingualgoritmist (laiutiotsing ja muutumatu hinna otsing), mis on tihedalt seotud iteratiivse pikendamise algoritmiga ja mida kasutatakse ka iteratiivse pikendamise algoritmi kirjeldamise juures.

Peamine rõhk on suunatud iteratiivse pikendamise algoritmi hindamisele ja kirjeldamisele. Antud lõputöö ei väida, et tegemist on väga hea algoritmiga, vaid keskendub iteratiivse pikendamise algoritmi kui ühe võimaliku otsingualgoritmi tutvustamisele. Iteratiivse pikendamise algoritmil on teatavaid puudusi, mis tuuakse ka selles töös välja.

Iteratiivse pikendamise algoritmi kui muutumatu hinna algoritmi iteratiivse analoogi üldsõnaline tutvustus on välja toodud õpikus [2], kuid algoritmi täpsem uurimine on jäetud lugejate hooleks. Iteratiivset pikendamist on rakendatud uurimistöös [12], milles on välja pakutud tehnika, kuidas otsida efektiivselt võtmesõnu väga suurest kõneandmebaasist ilma suuri mäluhulkasid kasutamata. Lisaks on kasutatud iteratiivse pikendamise (täpsemalt lineaarse iteratiivse pikendamise) algoritmi reageerivate reeglitega (*reactive rules*) planeerimisel [13].

Antud töö on jaotatud sisult kolme suuremasse ossa. Esimeses osas tuuakse sisse mõisted, mis on probleem ja probleemilahendus tehisintellekti seisukohast ning tutvustatakse teoreetilist tausta, kuidas toimub otsinguprobleemi (lühima tee leidmine otsingugraafis) lahendamine tehisintellektis. Lisaks antakse ülevaade laiutiotsingu ja muutumatu hinna otsingu kohta, mida kasutatakse ka iteratiivse pikendamise algoritmiga seotud töö osas. Teine osa on pühendatud iteratiivse pikendamise algoritmi kirjeldamisele ja hindamisele, lisaks tuuakse välja seosed teiste otsingualgoritmidega. Omaette alampeatükk on iteratiivse pikendamisega seotud raskuste väljatoomiseks. Kolmas osa kujutab endast algoritmi implementatsiooni ja rakendamist otsinguprobleemi lahendamiseks. Võrreldakse tema efektiivsust muutumatu hinna algoritmiga. Programmi töötulemuste alusel on välja toodud ka järeldused mõlema algoritmi soorituste kohta.

Käesoleva töö raames valminud iteratiivse pikendamise algoritmi rakendav näiteprogramm on avalikult kättesaadav ja allalaetav Google Code repositooriumist aadressil: http://code.google.com/p/itlen-algorithm-app/.

Bakalaureusetööd ja selle tulemusi saab edaspidi kasutada kursuses Tehisintellekt I täiendava materjalina.

1 Probleem ja probleemilahendus

Selles peatükis defineeritakse probleem ja probleemilahendus tehisintellekti (TI) seisukohast. Tuuakse sisse mõiste olekute ruum ehk probleemi-ruum (*state space*) ehk otsinguruum (*search space*) ja tutvustatakse ühte universaalset probleemilahendusmeetodit: otsingut olekute ruumis. Eesmärk on lugejale tutvustada teoreetilist tausta, mida kasutatakse järgnevate peatükkide sisu esitamisel. Põhiliselt on siin kasutatud materjali õpikust [2].

1.1 Probleemilahendus

Probleemi lahendamine (*problem solving*) algab eesmärgi seadmisest, mis on vajalik selleks, et aru saada, millised olukorrad probleemi lahendamise käigus oleksid eelistatumad.

Probleemi formuleerimine on protsess, mille tulemusena otsustatakse, milliseid tegevusi ja olukordi arvestada ja milliseid tegevusi rakendada erinevates olukordades probleemi lahendamisel, sõltuvalt seatud eesmärgist.

Probleemi lahendamise viimane osa on **probleemilahendus** – seesuguste tegevuste järjend, mis viib meid eelnevalt püstitatud eesmärgini.

Probleemi all mõeldakse ülesannet, mida saab lahendada teatavat meetodit kasutades. Üldine meetod probleemide lahendamiseks TI-s on kasutada otsingut olekute ruumis. Antud töös tähendab otsing teatavate arvutuslike tegevuste läbiviimist (kasvõi arvutiprogrammi abil). Kui otsinguprobleemide klasse on erinevaid, näiteks kahe mängijaga mängud, kitsenduste rahuldamise probleemid ja teeleidmise probleemid (*single-agent pathfinding problems*), siis antud lõputöö keskendub nendest viimasele.

1.2 Otsing

Tihtipeale on vaja reaalses elus vaja otsustada, milliseid tegevusi ja tegevustejärgnevusi läbida ning mida kõrvale jätta, et saavutada mingisugune teadlik olukord, milleni jõudmine oleks võimalikult optimaalne. Olgu selliseks probleemiks leida lühim tee linnade A ja B vahel, mille lahendamise saab taandada otsingule.

Rakendades otsingut kui probleemilahendusmeetodit, tuleb kõigepealt probleem abstraheerida, see tähendab jätta välja kõik ebavajalikud detailid(, mis antud probleemi lahendamisel ei oma tähtsust) ja püstitada probleem võimalikult lihtsalt ja täpselt. Kui on vaja leida lühim teepikkus kahe linna A ja B vahel, siis ei arvestata ilmastikuolusid, sõidukiirust ja muid tegureid, vaid huvitutakse ainult linnadest ja nendevahelistest teepikkustest. Kui

probleemi lahendamisel töödeldakse informatsiooni, mis lahenduse leidmise seisukohalt ei ole oluline, siis väga suure otsinguruumi korral oleks selline lähenemine probleemilahendusel kulukas.

TI-s on sagedaseks nähtuseks olukord, kus ühest olekust võib minna edasi mitmesse teise olekusse ja neid alternatiivseid olekuid ei pruugi olla ainult üks, millest omakorda võib edasi minna järgmistesse olekutesse. Sellisel juhul otsing suureneb nii olekute arvu kui ka alternatiivsete olekute tasemete võrra (kui kujutada olekuid ja üleminekuid puustruktuurina, siis puustruktuur suureneb nii laiuti kui süvitsi), mille tõttu otsingu suurus võib kasvada eksponentsiaalselt. Sellise nähtuse puhul on tegemist kombinatoorse plahvatusega (*combinatorial explosion*), mille ärahoidmine on olnud tehisintellektis üheks mureküsimuseks kogu tema ajaloo jooksul. [1]

1.3 Otsing olekute ruumis

Olekute ruum ehk **probleemi-ruum** (*state space*) on keskkond, kus viiakse otsingut läbi.

Olekute ruum koosneb:

- probleemi **olekutest**;
- **tegevustest** ehk **reeglitest**, mille abil liigutakse ühest olekust teise.

Kahe linna *A* ja *B* vahelise lühima teepikkuse leidmise näite põhjal:

- olekute ruumiks on linnad ja linnadevahelised teed;
- olekuteks on linnad;
- tegevusteks on siirdumised ühest linnast teise.

Eristatakse metatasemel ja objektitasemel olekute ruume. Metatasemel olekute ruumi üheks olekuks on näiteks arvutiprogrammi olek mälus (otsingupuu esitus), ja tegevuseks on arvutus, mis muudab programmi olekut (puusse ühe tipu lisamine). Objektitasemel olekute ruum, millest lähtutakse ka antud töös olekute ruumi kirjeldamisel, on näiteks Rumeenia kaart koos linnadega. Iga metatasemele vastav olek (tipp otsingupuus) jäädvustab objektitaseme otsingupuu olekut (linna Rumeenia kaardil). [2]

Probleem defineeritakse järgmise 5 komponendi abil [2]:

1. algolek (näiteks linn A, millest alustatakse lühima tee leidmist);
2. võimalikud lubatud tegevused, mille abil liigutakse ühest olekust teise (näiteks siirdumised naaberlinnadesse);

3. kirjeldused, kuidas probleemi olek muutub, kui talle rakendada mingisugust tegevust (kõik linnad ja nendest lähtuvad naaberlinnad). Tegevuse kirjelduse näide: kui olekuks on linn A ja tegevuseks „liigu linnast A naaberlinna B“, siis tulemuseks on olek linn B;
    - Algolek, lubatud tegevused ja tegevuste rakendamiste kirjeldused määravad ära kaudselt olekute ruumi – see on olekute hulk, milleni jõutakse rakendades järjestikuselt lubatud tegevusi alustades algolekust.
4. test, kas tegemist on lõppolekuga
    - Mõnikord ei ole aga lõppolek eksplitsiitselt välja toodud, vaid lõppolekuks osutuv olek peab rahuldama mingisugust tingimust. Näiteks kui malemängus on kuningas tule all, siis on tegu lõppeolekuga (matt) või 8 lipu malelauale paigutamise probleemi puhul on lõppolekuks iga selline olek, kus ükski malelaual paiknevatest 8 lipust ei tulista ühtegi teist;
5. läbitud teepikkuse arvutamine.

Probleemi juhtumiks (*problem instance*) nimetatakse olekute ruumi koos kindlaksmääratud alg- ja lõppolekuga [3].

1.4 Probleemi olekute ruumi esitus graafina

Sageli esitatakse probleemi olekute ruum graafina[1]:

- tipud – olekute ruumi olekud;
- kaared tippude vahel – tegevused või reeglid, mis viivad olekute ruumis ühest olekust teise;

Teekond ühest tipust teise olekute ruumis on olekute järjend, kus toimub ühest olekust teise liikumine, rakendades järjestikuselt tegevusi.

Probleemi püstitusest sõltub, kas tegemist on orienteeritud või orienteerimata graafiga, näiteks kas linnadevahelised teed on ühesuunalised või kahesuunalised. Kui tegemist on ühesuunalise teega, siis me saame tegevust rakendada olekule A ainult ühes suunas, st $A \rightarrow B$, aga mitte $B \rightarrow A$.

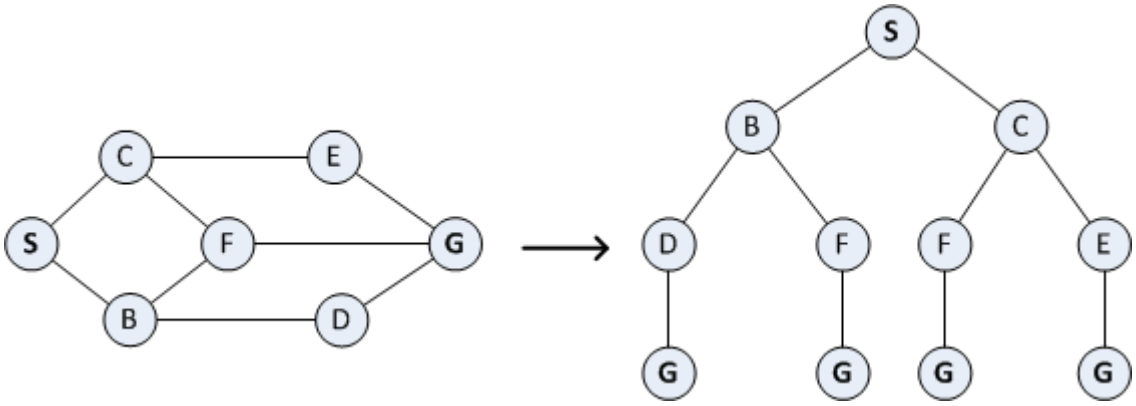
Erinevalt teistest üldistest graafiotsingu algoritmidest, esitatakse TI-s otsingugraafid kaudselt, seda eelkõige mälusuuruse piiratuse tõttu (näiteks male olekute ruumi graaf koosneb juba

[1] Graaf on paar $G = (V, E)$, kus V on mittetühi hulk ja E on hulk, mis koosneb hulga V kaheelemendilistest alamhulkadest. Hulga V elemente nimetatakse graafi tippudeks, hulga E elemente servadeks (või kaarteks) tippude vahel.[8] Sageli on servadele omistatud mingisugune reaalarv, nn kaal.

10^{25} tipust[3]). Otsingugraafis määratakse algolekule vastav tipp ja tegevuste hulk, mille abil genereeritakse järk-järgult uusi tippe olemasolevatest.

1.5 Graafi viimine puu kujule

Lihtsuse huvides esitatakse otsingugraaf mõnikord puuna, kus algolekule vastav tipp on puu juureks. Kui graafis ühte tippu suunduvaid kaari on enam kui üks, siis otsingugraafi viimine puu kujule tekitab korduvaid tippe puus ja seega suurendab puu suurust. Seevastu on aga puustruktuur tsüklitevaba ja see asjaolu lihtsustab otsingualgoritmide tööd. Tihtipeale on ka parem otsingualgoritmi hinnata ja tema tööpõhimõtet kirjeldada, kui tegemist on puuga ja mitte graafiga.



Joonis 1. Graafi viimine puu kujule

Puu konstrueerimisel me ei lisa tipust väljuvate tippude hulka tipu eellast. Joonisel 1 on algolekule vastavaks tipuks S ja lõppolekule vastavaks tipuks G.

1.6 Andmestruktuurid tippude hoidmiseks arvutimälus

Elementide ehk graafi tippude (olekute) hoidmiseks arvutimälus kasutatakse metatasemel spetsiaalseid andmestruktuure nagu järjendid, millele saab rakendada erinevaid otsingualgoritme. Kolm olulisemat järjendit on järgnevad:

- magasin (ingl k *stack* ehk *LIFO*-tüüpi järjend): viimasena lisatud element võetakse välja kõige esimesena;
- järjekord (ingl k *queue* ehk *FIFO*-tüüpi järjend): välja võetakse element, mis on järjekorras kõige esimene, ja uus element lisatakse järjekorra lõppu;
- eelistusjärjekord (ingl k *priority queue*): välja võetakse minimaalse väärtusega element.

Tipp on omaette andmestruktuur, mis sisaldab viitasid, et lihtsamini puustruktuuris ringi liikuda. Iga tipu n korral on tegemist andmestruktuuriga, mis võib sisaldada järgmisi komponente [2, lk 78]:

- olek – tipu n olek olekute ruumis (näiteks külastatud või veel külastamata);
- eellane (*predecessor* või *parent*) – tipp puus, mis genereeris tipu n, ehk tipp, millest jõuti vahetult tipuni n;
- tegevus – rakendatakse eellasele, et genereerida tippu n (ehk uut olekut olekute ruumis);
- teehind – koosneb kaarte kaaludest juurtipust s kuni tipuni n. Traditsiooniliselt tähistatakse seda $g(n)$ (s.o funktsioon, mis arvutab teehinda vastavalt tippude vaheliste kaarte kaaludele).[2]

Lisaks saab igale eellasele lisada viida tema alluvale ehk järglasele (*child* või *successor*). Kuna me teame järglase andmestruktuuri kirjelduses viita eellase objektile (nagu kirjeldatud ülevalpool andmestruktuuri kirjelduses), siis me saame alati eellase objekti selle viida kaudu kätte ja temaga teatavaid operatsioone teostada. Samuti me teame viita praegusele ehk järglase objektile (*this* võtmesõna programmeerimiskeeltest [15]). Kui meil on olemas mõlemad objektid, siis me saame eellase objektile lisada viida järglase objektile, sõltumata sellest, et esialgne andmestruktuuri kirjeldus otseselt seda ei ütle.

[2] Tipu n teehinna (*node path-cost*) all mõeldakse sageli temani jõutud teehinda: kas siis läbitud tippude vaheliste kaarte kaalude summat või läbitud tippude arvu alates juurtipust s.

2 Otsingualgoritmi keerukuse hindamine

Antud peatükis tuuakse välja võimalikud parameetrid, kuidas hinnata erinevate otsingualgoritmide sooritusi probleemi lahendamisel. Käsitlus põhineb enamasti õpikul [2].

2.1 Otsingualgoritmi hindamise parameetrid

Täielikkus – algoritmi võimekus leida lahend, kui see üldse leidub.

Optimaalsus – algoritmi võimekus leida optimaalne lahend (selline tegevuste järjend, mis viib algolekust lõppolekusse kõige lühemat teed pidi)

Ajaline keerukus – kui kaua kulub algoritmil aega lahendi leidmiseks. Tinglikult võime ajalise keerukuse lugeda võrdseks kõikide läbitud tippude arvuga.

Mäluvajadus – maksimaalne arv tippe, mida hoitakse korraga mälus, kui lahendatakse probleemi.

Teoreetilises arvutiteaduses esitatakse graafi, mille olekute ruumi suurus on $|V| + |E|$, kus V on tippude hulk ja E on tippudevaheliste kaarte hulk, eksplitsiitse andmestruktuurina, mis antakse otsingu algoritmi sisendiks. TI-s see-eest väljendatakse enamikel juhtudel probleemi suurus ja läbimisele kuuluvad tipud otsingugraafis kaudselt, arvestades otsingugraafi (või otsingupuu) omadusi [2, 3].

Otsingualgoritmi efektiivsuse hindamiseks kasutatakse 3 arvsuurust, mis otsingupuud iseloomustavad:

- b – hargnemistegur puus (maksimaalne ühest tipust väljuvate kaarte arv);
- d – probleemi lahendi (lõppoleku) sügavus puus;
- m – maksimaalne võimalik teepikkus olekute ruumis.

Algoritmi efektiivsuse hindamise juures me arvestame otsingu läbiviimise maksumust (*search cost*): aja- ja mäluvajadust. [2] Viimased väljendatakse tihtipeale vastavalt läbitud ja ühekorraga mälus hoitavate tippude arvuna.

2.2 Kompromiss aja- ja mälukulu olulisuse vahel

Arvutiteaduses tuleb sageli leida kompromiss aja- ja mälu olulisuse vahel, sõltuvalt sellest, kui kiiresti soovitakse lahendust probleemile leida ja millisel viisil kasutada andmestruktuuri probleemi lahendamiseks.

Kui aeg on olulisem ja arvutimälu on piisavalt, siis üks võimalus on kasutada mällu eelsalvestatud andmestruktuuri nagu otsingutabel (*lookup table*), milleks on sageli järjend, assotsiatiivne järjend või ahel (*linked list*) [5]. Tihti on selline otsing arvutuslikult kiirem, kuna igal sammul ei ole vaja mälust ruumi juurde küsida ja iga element on kättesaadav mälust kindla viida kaudu. Näiteks kogu otsingugraafi võib esitada arvutimälus assotsiatiivse järjendina.

Kui arvutimälu on piiratud ja aeg ei ole nii oluline, siis võib otsingutabelit ehitada käigupealt vastavalt vajadusele, mis on küll arvutuslikult aeglasem, kuid hoiab kokku arvuti mäluruumi. [6] Viimane variant on eelistatum TI seisukohast, eelkõige eksponentsiaalse keerukusega otsinguprobleemide ja mälupiiratuse tõttu.

2.3 Suhtarvude kasutamine otsingugraafis teepikkuste esitamisel

Järgnevates töö osades kasutatakse kaarte kaalude ehk naabertippude vaheliste teepikkuste kirjeldamisel osaliselt ka suhtarve, nimelt lühima tee leidmise probleemi lahendamise efektiivsuse hindamiseks halvimal juhul. Antud alampeatükis tuuakse ka välja, mida mõeldakse minimaalse teehinna ε all.

Otsingugraafi läbimisel kasutatakse kaarte kaaludena suhtarve, st. konkreetse kaare kaalu ja senileitud optimaalse teepikkuse (mingi tipu n ja juurtipus s vahel) suhet. Põhjus võib olla selles, et mõnikord võivad teehinnad minna väga suureks, kuid lihtsam on neid vaadelda kui nad on mingis vahemikus. Seega on teehindadeks reaalarvud poollõigust $(0,1]$, kus minimaalset teehinda tähistame ε ja $0 < \varepsilon \leq 1$. Optimaalne lahend on tavaliselt teepikkusega 1. Iteratiivse pikendamise algoritmi igal iteratsioonil (millest räägitakse töö järgnevas osas) juures arvutatakse suhtelised teehinnad uuesti ja iga iteratsiooniga peab hinnanguliselt minimaalne teehind optimaalsele lahendile vastava teepikkuse suhtes täpsemaks muutuma (vähenema). Samuti saab kasutada väikest reaalarvu ε halvima juhu leidmiseks. Kui meil on otsinguprobleem, kus on ette teada, et optimaalse lahendi korral on teepikkus 5 ja minimaalne kahe naabertipu vaheline teepikkus on 1, siis suhtarvudena esitades on optimaalsele lahendile vastav teepikkus $1\ (= 5/5)$ ja minimaalne teepikkus $1/5 = 0{,}2$. Kui me juhtumisi ei tea, millised on ülejäänud teepikkused, siis saame oletada, et see tee (mille teepikkus oli 5) jaotubki väga väikesteks lõikudeks teepikkustega 1 ja saame anda algoritmile ka hinnangu halvima juhu jaoks.

3 Mõned üldised otsingualgoritmid

Siinses peatükis kirjeldatakse üldisi pimeotsingu (ehk ka jõumeetodil põhinevaid) algoritme: laiutiotsingut ja muutumatu hinna algoritmi, mida kasutatakse töö järgnevates osades iteratiivse pikendamise algoritmiga võrdlemisel, ja esitatakse nende hinnanguparameetrid.

3.1 Laiutiotsing

Laiutiotsing (*breadth-first search*) on üks üldisemaid pimeotsingu algoritme, mille strateegia on läbida puujuurele kõige lähemal asuvad tipud (kas siis suunaga vasak-parem või parem-vasak) ja nii iga tipu korral. Tippude hoidmiseks mälus kasutatakse *FIFO*-tüüpi järjendit ehk järjekorda, see tähendab uusi tippe lisatakse alati järjendi lõppu ja läbivaatamiseks võetakse varem lisatud tippe järjendi algusest.

Laiutiotsingu algoritm leiab lahendi kui see leidub, kuna tippe salvestatakse ja eemaldatakse järjestikuselt tasemete kaupa liikudes puus suunaga vasak-parem või parem-vasak. Kui leidub veel teisi lahendile vastavaid tippe, siis kõige esimene leitud lahend peab olema juurtipule kõige lähemal asuv tipp. Põhjuseks on see, et ükski eelnev tipp ei rahuldanud lõppoleku tingimust ja seetõttu ei saa tekkida sellist situatsiooni, kus leitakse selline lõpptipule vastav lahend, mis kõikide võimalike lahendite hulgast ei asu juurtipule kõige lähemal.

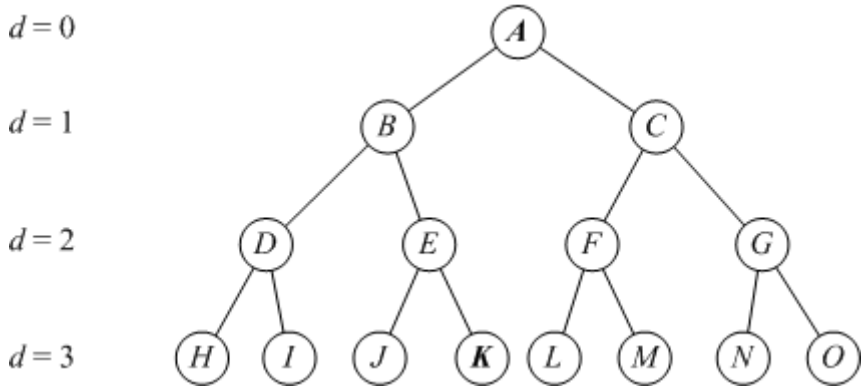
Laiutiotsing ei leia alati optimaalset lahendit, ta leiab selle ainult tingimusel, kui tippudevaheliste kaarte hinnad on ühesugused või puu sügavuse kasvades mittekahanevad ja positiivsed.[2]

Laiutiotsingu algoritmi hindamise juures arvestame puu hargnemistegurit b ja lahendisügavust d ning eeldame, et tegemist on lõpliku ja ühtlase puuga (*uniform tree*)[3]. Laiutiotsingu ajaline keerukus halvimal juhul on seega kõikidel tasemetel $0, 1 \dots d$ läbitud tippude arv: $b^0 + b^1 + \cdots + b^d \sim O(b^d)$. Kui lahend asub tasemel d ja tasemel $d-1$ ei kontrollita tippude genereerimise käigus lõppolekule vastava tipu olemasolu, siis halvimal juhul leitakse lõpptipp alles siis, kui läbitakse tasemel d asuvaid tippe ja $d+1$ tasemel asuvad tipud on selle käigus juba genereeritud, mistõttu halvim ajaline keerukus võib olla isegi $O(b^{d+1})$ [2, 4].

Laiutiotsingu mäluvajadus (üheaegselt mälus säilitatavate tippude arv) on halvimal juhul võrdeline lahendisügavusel asuvate tippude arvuga, see tähendab, et puu sügavusel $d-1$

[3] Tehisintellektis keerukuse hindamisel kasutatav mudel, millel on konstantne hargnemistegur b, ainult üks lahendile vastav lõpptipp puutasandil d.[7]

peab eelnevalt genereerima ja mällu salvestama kõik sügavusel d asuvad tipud, seega $O(b^d)$. Negatiivne omadus antud algoritmi juures on see, et suurte probleemide korral on algoritm väga kulukas nii ajaliselt kui mälu osas, kuna probleemi suurus võib eksponentsiaalselt kasvada, kui kasutada laiutiotsingut. Üldiselt kehtib siiski järeldus, et eksponentsiaalse keerukusega probleeme ei saa lahendada ühegi pimeotsinguga [10].



Joonis 2. Ühtlane puu hargnemisteguriga $b = 2$ ja lahendisügavusega $d = 3$.

Näide. Olgu meil lõplik ühtlane puu hargnemisteguriga $b = 2$ ja lahendisügavusega $d = 3$. Antud näite korral halvimal juhul laiutiotsingu algoritmi ajaline keerukus on tippude arv $2^0 + 2^1 + 2^2 + 2^3 = 15$ ja mäluvajadus on 8. Kui lahendile vastab tipp *K* joonisel 2, siis ajaline keerukus on 11 ja mäluvajadus on 8.

Tabel 1. Laiutiotsingu algoritmi hinnanguparameetrid [3]

Täielikkus:	Jah
Optimaalsus:	Jah[4]
Ajavajadus:	$O(b^d)$
Mäluvajadus:	$O(b^d)$

3.2 Muutumatu hinna otsing

Muutumatu hinna otsingu (*uniform-cost search*) puhul on tegemist laiutiotsingu täiustusega, kus puus võib leiduda tippude vahel selliseid kaari, mille kaalud ei ole ühesugused. Kui laiutiotsing läbib puud süstemaatiliselt (juurtipule kõige lähemal asuvad tipud ja suunaga vasak-parem või parem-vasak), siis muutumatu hinna otsingu korral valitakse tippude läbimise järjekord vastavalt kaarte kaaludele (nendeks on positiivsed reaalarvud). Teisisõnu, muutumatu hinna otsingu puhul on oluline tipu n ja juurtipu s vaheline lühim teepikkus, aga

[4] Laiutiotsing on optimaalne, kui teehinnad on ühesugused või mittekahanevad ja positiivsed.

mitte enam läbitud tippude arv sellel teel. Tippude hoidmiseks mälus kasutatakse eelistusjärjekorda (*priority queue*): igal sammul valitakse puus edasiliikumiseks tipp, milleni on jõutud lühimat teed pidi.

Muutumatu hinna otsingu korral võib paralleele tuua ka arvutiteaduses rohkem tuntud Dijkstra algoritmiga. Võrdlus on toodud antud artiklis [9], mille sisu me käesolevas töös ei käsitle.

Laiutiotsingu ja muutumatu hinna otsingu tööprotsessidel on järgmised erinevused:

- laiutiotsing lõpetab töö, kui lõppolekule vastav tipp leitakse. Muutumatu hinna otsing lõpetab töö alles siis, kui lõppolekule vastav tipp on vähima väärtusega tipp eelistusjärjekorras, mida eemaldatakse. Põhjuseks on see, et võib leiduda mõni lühem tee tipuni, mis vastab lõppolekule, mida seni veel avastatud ei ole. Peale selle, kui parajasti lisatakse veel teine juba varem lisatud tipp, siis eelistusjärjekorda jäetakse neist ainult üks, milleni on jõutud lühimat teed pidi;
- laiutiotsing kontrollib alles tippude läbivaatamise ajal ja mitte tippude genereerimise ajal, kas mõni neist vastab lõppolekule; muutumatu hinna algoritm kontrollib lõppolekule vastava tipu olemasolu siis, kui leitakse vähima väärtusega tipp, mida hakatakse edasi uurima (antud tipust väljuvaid tippe genereerima), ja kui tegemist on lõppolekule vastava tipuga, siis lõpetatakse töö.

Muutumatu hinna otsing on täielik, see tähendab, et uusi tippe lisatakse ja varem lisatud tippe eemaldatakse seni, kuni järjend on tühi või kui selle protsessi käigus on leitud lahendile vastav lõpptipp.

Muutumatu hinna otsing on optimaalne: kuna eelistusjärjekorrast valitakse alati vähima teehinnaga tipp n, siis lühim tee juurtipust s tipuni n on leitud. Kui see ei ole nii, siis peab leiduma eelistusjärjekorras veel läbivaatamata tipp n', mille koguteehind $g(n')$ on väiksem kui tipul n ja muutumatu hinna algoritmi tööprotsessi kirjelduse kohaselt peab varasemate sammude jooksul olema hoopis valitud tipp n'. Teine tingimus on, et teehinnad saavad olla ainult positiivsed reaalarvud. Seega, igal sammul me valime tipu n, milleni jõuti lühimat teed pidi ja tipuni jõudmiseks kulunud teehind $g(n)$ saab ainult suureneda, siis me liigume puus edasi alati kõige lühemat teed pidi.

Muutumatu hinna otsingu keerukust on raske hinnata hargnemisteguri b ja lahendisügavuse d järgi, kuna muutumatu hinna algoritmi korral tegeletakse ainult teehindadega, st. suure otsingupuu korral saab keerukust hinnata ainult kaudselt. Kui tegemist oleks ühtlase puuga,

mille hargnemistegur on b ja lahendisügavus on d, siis halvim aja- ja mäluvajadus on $O(b^{c/\varepsilon})$, kus c on optimaalse senileitud teepikkus ja ε on minimaalne teehind. [2, 3]

Antud väite korral on tegemist suure otsingupuuga (käsitsi ülesjoonistamine nõuaks inimesel väga palju tööd), mis koosneb paljudest kaartest kaaludega, mis on vähemalt ε (väike positiivne reaalarv). Antud otsingupuu kaarte kaaludeks on suhtarvud:

$$\frac{kaare\ kaal}{optimaalne\ senileitud\ teepikkus},$$

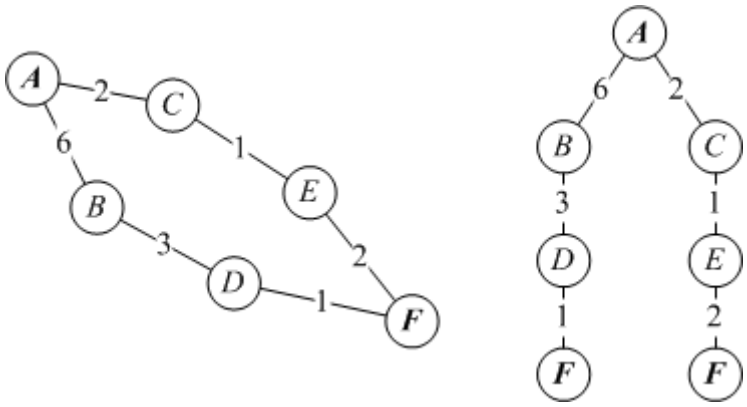
millest läbitud kaared jäävad poollõiku $(0,1]$.

Näide. Kui senine optimaalne teepikkus on 5 ja minimaalne kaare kaal on 2, siis suhtarvuna esitatuna on optimaalne teepikkus 1 $(= 5/5)$ ja minimaalse kaare kaal $\frac{2}{5} = 0{,}4$. Kui tegemist on ühtlase puuga, mille hargnemistegur on $b = 2$ ja lahendile vastav tipp asub konkreetse probleemi korral sügavusel d, siis halvim aja- ja mäluvajadus oleks $O\left(2^{1/0,4} \sim 6\right)$.

Algoritmi töös valitakse alati vähima teehinnaga tipp, seetõttu võidakse teha väga palju tööd selles alampuus, mis koosneb väga väikestest sammudest (kaarte kaaludele on omistatud väike reaalarv), enne kui jõutakse sellesse alampuusse, mis koosneb küll tippudest, mille vaheliste kaarte kaalud on suuremad, kuid mis võivad lahendi leidmise osas olla kokkuvõttes perspektiivikamad (vt Joonis 5). Halvimal juhul arvestataksegi, et terve otsingupuu võib olla selline, mille kaarte kaalud on hästi väikesed reaalarvud.

Kui muutumatu hinna otsingu korral on kõikide tippude teehinnad ühesugused, siis selline muutumatu hinna otsing on laiutiotsing. Erinevus on selles, et kui leitakse lõpptipp, siis laiutiotsing lõpetab töö, kuid muutumatu hinna otsing jätkab seni, kuni kõik tasemel d olevad tipud on läbi vaadatud, sest võib leiduda mõni teine, lühem tee lahendini, või lõpetab töö sellisel juhul, kui minimaalse väärtusega tipp, mida hakatakse eelistusjärjekorrast eemaldama, on lõppolekule vastav tipp, sõltumata kitsendusest, kas kõik tasemel d tipud on läbi vaadatud või mitte. Seetõttu muutumatu hinna otsing teeb sellisel puul, millel on ühesuguste kaaludega kaared, rohkem tööd kui vaja [2].

Muutumatu hinna otsingul esineb samuti mälupiiratuse probleeme, samal põhjusel nagu laiutiotsingulgi (tasemel d genereeritakse kõigepealt järgmise taseme $d + 1$ tipud).



Joonis 3. Leida lühim tee graafis tippude A ja F vahel.

Näide. Olgu vaja leida lühim tee graafis kahe punkti A ja F vahel (vt Joonis 3). Selles näites ei ole kasutatud suhtarve, vaid esialgseid kaarte kaalusid ja igal sammul ei väljendata kaarte kaalusid suhtarvudena. Muutumatu hinna otsingut rakendades läbime otsingupuu tipud järgmisel viisil:

1. ~~A (0)~~
2. B (6), ~~C (2)~~
3. B (6), ~~E (3)~~
4. B (6), ~~**F (5)**~~

Maha kriipsutatud element tähistab elemendi läbiuurimist; selle asemele lisatakse järgmisel sammul eelistusjärjekorda tema alluvad puus.

Oleme leidnud optimaalse lahendi 4 sammuga ja tema teepikkuse 5. Vastukaaluks muutumatu hinna otsingu tööle leiab laiutiotsing lahendi läbides otsingupuus 6 tippu (suunaga vasak-parem), mis pealegi ei ole optimaalne (vt Tabel 1. Laiutiotsingu algoritmi hinnanguparameetrid [3]).

Tabel 2. Muutumatu hinna otsingu hinnanguparameetrid [3]

Täielikkus:	Jah
Optimaalsus:	Jah
Ajavajadus:	$O(b^{c/\varepsilon})$
Mäluvajadus:	$O(b^{c/\varepsilon})$

4 Iteratiivse pikendamise algoritm

Kahe eelnevalt vaadeldud algoritmi (laiutiotsingu ja muutumatu hinna algoritmi) korral võis täheldada probleeme mälu ammendumisega suurte otsinguülesannete lahendamisel. Praktikas on tavaks saanud seada piir algoritmi mälukasutusele (*space-bound*), mida tehakse näiteks laiutiotsingu puhul [3, lk 4]. Samamoodi võib toimida muutumatu hinna otsinguga, et reguleerida algoritmi mälukasutust. Antud peatükis tutvustatakse muutumatu hinna otsingu algoritmi iteratiivset analoogi, mis peaks mäluvajadusega seotud probleeme leevendama. Lisaks esitatakse iteratiivse pikendamise algoritmi hinnanguparameetrid ja võrreldakse seda algoritmi mõlema eespool käsitletud algoritmiga (laiutiotsingu ja muutumatu hinna otsinguga).

4.1 Algoritmi kirjeldus

Iteratiivse pikendamise algoritm on muutumatu hinna algoritmi iteratiivne analoog [2, lk 90]. Idee seisneb selles, et kasutatakse jooksvat limiiti läbitud teepikkusele, mida igal iteratsioonil suurendatakse (erinevalt iteratiivsest laskumisest[5], kus hoitakse limiiti puusügavusele d). Eelmisel iteratsioonil kõrvalejäetud minimaalse teehinnaga tipu n teehind (teepikkus juurtipust s tipuni n) seatakse uueks limiidiks uuel iteratsioonil ja korratakse kogu tööprotsessi uuesti. Iga iteratsioon kujutab endast laiutiotsingut.

Algoritmi tööprotsessi kirjeldus

Olgu meil ajutine *FIFO*-tüüpi järjend L, mida kasutame igal iteratsioonil tippude hoidmiseks. Tähistused:

- l – jooksev limiit teehinnale
- m – kõrvalejäetud tippudest sellise tipu teepikkus, milleni jõuti kõige lühemat teed pidi juurtipust s ehk kõrvalejäetud tippudest minimaalne teehind;
- g – teehinna arvutamise funktsioon juurtipust s tipuni n
- $g(n') = g(n) + w(n, n')$ – w on teehind tipust n naabertipuni n'

Algoritmi etapid:

1. Määra $l = 1$; s.o esialgne limiit teehinnale.
2. Lisa järjendi L elemendiks juurtipp; minimaalne teehind $m = \infty$.

[5] Iteratiivne laskumine (*iterative deepening*) on otsingualgoritm, mille puhul igal iteratsioonil seatakse jooksev limiit puu sügavusele d. Iga iteratsioon kujutab endast süvitsiotsingut. Lõpmatu puu korral võibki süvitsiotsing jääda lahendit otsima ühest harust. Selle puuduse likvideerib iteratiivne laskumine, mis seab teatava piirsügavuse igal iteratsioonil. [10, lk 76-77]

3. Vali esimene element n järjendist L.
4. Kui n on otsitav tipp, siis lõpeta ja väljasta lahend.
    1. Kui L on tühi ja $m = \infty$, siis lahendit ei leitud.
    2. Kui L on tühi ja $m \neq \infty$, siis $l = m$ ja mine punkti 2.
5. Muidu eemalda tipp n järjendist L ja iga tipu n järglase n' korral:
    1. Kui $g(n') \leq m$, siis lisa tipp n' järjendi L lõppu.
    2. Muidu $m = \min(m, g(n'))$, mine tagasi punkti 3.

4.2 Iteratiivse pikendamise algoritmi hindamine

4.2.1 Täielikkus

Iteratiivse pikendamise algoritm leiab lahendi, kui lõppolekule vastav tipp sisaldub lõplikus otsingupuus. Iga iteratsiooni korral läbitakse (alam)puu, kasutades laiutiotsingut. Kuna laiutiotsing on täielik, siis see tähendab, et algoritm leiab lahendi, kui otsitav tipp jääb uuel iteratsioonil antud alampuu nende tippude hulka, mille teepikkus on $g(n) \leq l$ (Siin $g(n)$ on vaadeldava tipu n kaugus juurtipust s ja l on jooksev limiit teepikkusele.)

4.2.2 Optimaalsus

Iteratiivse pikendamise algoritm leiab lühima teepikkusega lahendi. Kuna algoritm on täielik, siis peab optimaalse teepikkusega tipp asuma selle alampuu tippude hulgas, mille teepikkus on $g(n) \leq l$, see tähendab, et lahendile vastava tipu korral kehtib $g(n) \leq l$ ehk kui tegemist on optimaalse lahendiga, siis selle lahendi väärtus ei tohi ületada jooksvat limiiti l.

Väidame, et tegemist ei ole optimaalse lahendiga, kuid tipp ei ületa ka jooksvat limiiti l. Olgu meil puu, mis koosneb ainult ühest tipust ehk juurtipust, siis lõppolekule vastav tipp on juurtipp s ise ja mille teepikkus on $g(s)$ (ehk 0). Ühe tipuga puu korral on tegemist optimaalse lahendiga, kuna kehtib $g(s) = g(n) = 0 \leq l$ (limiit l on alguses määratud kui 1). Kui puus on rohkem kui üks tipp, siis igal iteratsioonil me omistame uueks jooksvaks teehinna limiidiks l eelmisel iteratsioonil sellise teepikkuse m, mis saadi liikudes kõige lühemat teed pidi juurtipust s konkreetse kõrvalejäetud tipuni n (mille teehind ületas jooksvat teehinna limiiti l), s.t. $l = m$. Seega uuel iteratsioonil iga alampuu tipu n korral kehtib järgmine võrratus: $g(s) \leq g(n)$ ja $g(n) \leq l$. Kui me oleme leidnud optimaalse lahendi, siis see peab leiduma alati selles alampuus, mille kõikide tippude teehinnad ei ületa eelmisel iteratsioonil valitud minimaalse teehinnaga tipu teepikkust m. Kui see aga ei ole optimaalne lahend, siis see tähendab seda, et peab leiduma tipp n', mille teehind on väiksem, kui eelmisel

iteratsioonil leitud minimaalne teehind m, mis aga ei ole võimalik, sest siis me oleksime pidanud selle leidma juba varasemate iteratsioonide jooksul alampuud läbides.

4.2.3 Ajaline keerukus ja mäluvajadus

Iteratiivse pikendamise algoritmi juures, sarnaselt muutumatu hinna otsingule, on raske hinnata ajalist keerukust ja mäluvajadust, kuna ei arvestata hargnemistegurit b ja puu sügavust d, vaid läbitud teepikkusi.

Kui otsing toimub ühtlasel puul, mille hargnemistegur on b ja üksainus lõppolekule vastav tipp asub sügavusel d ning kõikide tippude vahelised kaared on sama kaaluga, siis iteratiivse pikendamise algoritmil kulub lahendi leidmiseks d iteratsiooni.

Lihtsuse huvides olgu meil kõik teepikkused võrdsed ühega. Kuna kõikide tippude naabertipud asuvad teineteisest võrdsetel kaugustel, siis igal iteratsioonil kõrvalejäetud tippude (järglaste) hulgast on kõigil vähim teehind, mis seatakse järgmisel iteratsioonil uueks teehinna limiidiks l. Kuna iga iteratsioon on laiutiotsing, siis läbides puus ühe taseme d, mille kõikide tippude korral $g(n) \leq l$, suurendatakse teehinna limiiti l nii, et järgmisel iteratsioonil genereeritakse ka tipud tasemel $d + 1$, mis kuuluvad kõik külastatavate tippude hulka. Seega teehinna limiidi suurendamisel suurendatakse ka puu sügavust d, mille tippude läbimiseks kasutatakse laiutiotsingut järgmisel iteratsioonil. Siit saab järeldada, et lahendisügavuse d ja iteratsioonide arvude vahel kehtib võrdeline seos.

Näide. Olgu meil ühtlane puu hargnemisteguriga 2 ja lahend asugu sügavusel 3 (Joonis 2). Kuna kõik teepikkused on võrdsed ühega, siis lihtsuse mõttes me neid joonisel ei esita. Antud näites kulub lahendi leidmiseks 3 iteratsiooni. Joonisel 2 algavad puu tasemed nullist, mistõttu antud näite korral lahendisügavus d on ühe võrra väiksem kui iteratsioonide arv (ehk tegelikult kulub 4 iteratsiooni), kuid alati saame alustada puutasemete loendamist ühest.

Iteratiivse pikendamise algoritmi korral on teada, et ajaline keerukus on võrdne igal iteratsioonil läbitud kõikide selliste tippude n (mille teepikkus juurtipust s on $g(n) \leq l$), ja tippude n vahetute järglaste summaga (kontrollitakse teepikkust juurtipust s, kuid ei lisata järjendisse). Iteratiivse pikendamise algoritm kontrollib ka neid kõrvalejäetud vahetuid tippe, milleni jõuti kõige lühemat teed pidi juurtipust s ja mis ei jäänud jooksva teehinna limiidi l piiridesse. Mäluvajadus, vastavalt laiutiotsingu definitsioonile, on maksimaalne genereeritud tippude arv tasemel d. Kui tegemist on ebaühtlase otsingupuuga, siis tasand d, mille jaoks salvestatakse mällu maksimaalne arv tippe, ei pruugi olla viimane puutasand.

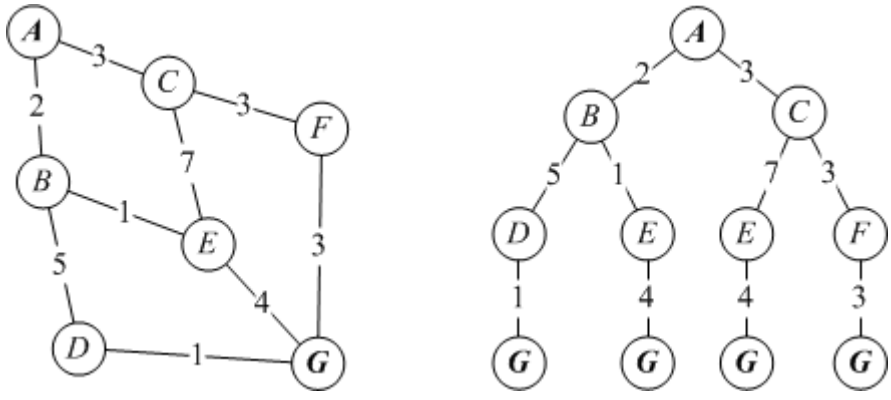
Oletame jälle, et tegemist on ühtlase puuga, mille hargnemistegur on b ja lahendisügavus on d ning kõik kaarte kaalud on ühesugused, mille põhjal saame anda umbkaudse hinnangu. Iteratiivse pikendamise algoritm läbib puud sellisel juhul tasemete haaval $(0, 1, \dots, d)$, ainult et teeb seejuures rohkem tööd kui laiutiotsing, kuna iga iteratsioon kujutab endast juba omaette laiutiotsingut. Igal iteratsioonil valitakse läbimiseks (alam)puu sügavusega $0, 1, \dots, d$ ehk kokkuvõttes läbitud tippude arv on:

- 1. iteratsioonil: b^0
- 2. iteratsioonil: $b^0 + b^1$
- 3. iteratsioonil: $b^0 + b^1 + b^2$
- …
- d. iteratsioonil: $b^0 + b^1 + \cdots + b^d$

Siit saame järeldada, et suurem osa algoritmi tööst tehakse ära viimasel iteratsioonil ja asümptootiline ajaline keerukus on sama, mis laiutiotsingul: $(db^0 + (d-1)b^1 + \cdots + b^d) \sim O(b^d)$. Halvimal juhul on algoritmi mäluvajadus viimasel iteratsioonil ja lahendisügavusel d genereeritud tippude arv: b^d.

Suurte otsinguprobleemide korral me ei saa anda iteratiivse pikendamise algoritmile täpset hinnangut ajalise keerukuse ja mäluvajaduse kohta, v.a halvima juhu kohta, või sellisel juhul, kui antud algoritm implementeeritakse programmis ja lastakse programmil vajalikud mõõtmistulemused leida (täpne läbitud tippude arv).

4.3 Näide iteratiivse pikendamise algoritmi rakendamisest otsinguprobleemi lahendamisel



Joonis 4. Otsinguprobleem: leida lühim tee punktide A ja G vahel.

Olgu meil vaja leida lühim tee kahe punkti A ja G vahel (vt Joonis 4). Probleemilahendamiseks rakendame iteratiivse pikendamise algoritmi.

Olgu antud ajutine *FIFO*-tüüpi järjend L, kuhu hakatakse lisama järjest külastatavaid tippe, mis ei ületa jooksvat limiiti l. 1. iteratsioonil on jooksev teehinnalimiit l seatud vaikimisi võrdseks ühega ja algseadistame minimaalse teehinna kui määramata suuruse: $m = \infty$. Esmalt lisame järjendisse L juurtipu A. Seejärel eemaldame tipu A järjendist ja hakkame edasi uurima tipust A väljuvaid tippe (B ja C). Mõlemad tipud ületavad jooksvat teehinna limiit l ja jäetakse kõrvale ega lisata järjendisse L. Kõrvalejäetud tippudest valitakse minimaalse teehinnaga tipp, 1. iteratsioonil on selleks tipp B teehinnaga $g(B) = 2$ $(= m)$, ja seatakse uueks teehinna limiidiks l. Kuna vahepeal enam ühtegi tippu järjendisse L ei lisatud ja $m \neq \infty$ (algoritmi etapp 4.2 algoritmi kirjelduse all alampeatükis 4.1), siis liigume edasi järgmisesse iteratsiooni ja alustame uuesti juurtipust A.

Jätkame samasugust protseduuri nagu me tegime 1. iteratsioonil. Eelmisest iteratsioonist on jooksev teehinnalimiit $l = 2$. Eemaldame tipu A järjendist L ja hakkame läbi vaatama tipust A väljuvaid tippe (B ja C). Seekord tipu B teehind ei ületa jooksvat teehinna limiiti l ja tipp lisatakse järjendisse L. Kõrvalejäetud tipu C teehind $g(C) = 3$ seatakse esialgu minimaalseks teehinnaks m. Eemaldame esimese elemendi järjendist L, milleks on äsja lisatud tipp B ja hakkame edasi uurima tipust B väljuvaid tippe (D ja E). Kumbki tipp ei mahu jooksva teehinnalimiidi l piiridesse, nad jäetakse kõrvale ega lisata järjendisse L. Kõrvalejäetud tippudest jäi minimaalseks teehinnaks $m = 3$, mille me seame uueks jooksvaks teehinnalimiidiks l. Kuna ühtegi tippu enam järjendisse L ei lisatud ja jooksev teehinnalimiit $m \neq \infty$, siis jätkame järgmise iteratsiooniga.

Kui me oleme järginud samasugust protseduuri nagu eelmiste iteratsioonide jooksul, siis jõuame 5. iteratsioonil optimaalse lahendini, milleks on tee $A - B - E - G$ pikkusega 7 (vt Lisa 1).

Kui oleks vaja lahendada selline probleem, kus lahendile vastavat lõpptippu ei leidu otsingugraafis, siis viimasel iteratsioonil läbitaks kõik puu tipud, kasutades laiutiotsingut, ja jõutakse kuni puu lehtedeni, millest enam ükski tee edasi ei vii. Seetõttu ei ole ka ühtegi kõrvalejäetud tippu, millest valida minimaalset teehinda m. Ollakse jõutud olukorda, kus järjend L on tühi ja minimaalne teehind m on määramata ($m = \infty$) ja seega võib probleemilahenduse lõpetatuks lugeda ilma lahendit leidmata.

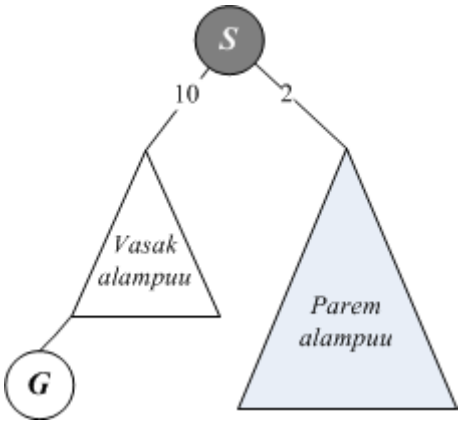
4.4 Iteratiivse pikendamise algoritmiga seotud raskused

Iteratiivse pikendamise algoritmi üheks negatiivseks omaduseks on see, et otsingu alguses võidakse valida mitteperspektiivikas haru läbimiseks (vt Joonis 5), st. võidakse suunduda

sellesse (alam)puusse, mis koosneb paljudest kaartest, mille kaalud on väikesed reaalarvud ε (lahendile vastava teepikkusega võrreldes). Selles alampuus tehakse palju tööd (läbitakse rohkem tippe kui optimaalseks lahendiks vajalik) enne, kui jõutakse teise (alam)puusse, mis koosneb tippudest, mis esimestel sammudel kõrvale jäeti, kuna teepikkus nendeni ületas tol hetkel jooksvat teehinna limiiti l. Kuid nüüd hilisematel sammudel on need optimaalse lahendi leidmise seisukohalt olulised. Siit tulenevalt ka iteratiivse pikendamise algoritmi nõrkus ja pimeotsingule vastav omadus, et ta ei suuda kaugemale näha kui määratud jooksev limiit l. Teisest küljest on iteratiivse pikendamise algoritmi käitumine omane „ahnele" algoritmile (*greedy algorihm*) [14], mis valib alati lokaalselt näiliselt parima (tipu, milleni on jõutud lühimat teed pidi), lootes jõuda globaalselt optimaalsema lahendini. Samas võib ka juhtuda, et esimestel sammudel valitud tipud, milleni jõuti lühimat teed pidi, ka pikemas perspektiivis on osa optimaalsest lahendist.

Iteratiivse pikendamise algoritmi kasutuselevõtu peamine eesmärk oli leevendada mäluammendumisega seotud probleeme, mis esinesid laiutiotsingul ja muutumatu hinna otsingul. Suure otsinguprobleemi korral võib sama puudus ka raskendada iteratiivse pikendamise algoritmi tööd, kuna iga iteratsioon on oma loomult laiutiotsing. Eriti kui määratakse selline jooksev teehinna limiit l, mille piiridesse võib jääda uuel iteratsioonil mingil puutasemel d väga palju uusi tippe. Seega igal iteratsioonil võib esineda mälupiiratuse probleeme nagu on välja toodud ka laiutotsingu kirjelduse all eespool töö osas (peatükk 3.1).

Kuna iga iteratsioon on eraldi laiutiotsing, siis siit tuleneb ka suurte otsinguprobleemide lahendamiseks (mille lahendamine käsitsi inimese poolt on võimatu) kuluv liigne tööaeg (läbitud tippude arv; vt Lisa 4), eriti kui tippe läbitakse ühekaupa, ja mälu (ühe korraga mälus hoitavate tippude arv) .



Joonis 5. Valitakse parempoolne aga mitte nii perspektiivikas alampuu läbimiseks, kuigi lõppolekule vastav tipp asub vasakus alampuus.

Iteratiivse pikendamise algoritmi hindamine on keerulisem suurte otsinguprobleemide korral, kuna iteratiivse pikendamise algoritmi puhul me ei saa kasutada hargnemistegurit *b* ja puu sügavust *d* algoritmi ühese keerukushinnangu andmiseks, võrreldes süstemaatiliste üldisemate otsingualgoritmidega. Kaudne hinnang on antud eespool iteratiivse pikendamise algoritmi hindamise osas (peatükk 4.2), kuid täpsemaks algoritmi töö võimekuse hindamiseks ja võrdlemiseks teiste algoritmidega tuleks iteratiivse pikendamise algoritm implementeerida programmina ja võrrelda tema tööd otsinguprobleemi lahendamisel teiste algoritmidega (laiutiotsing ja muutumatu hinna otsing).

4.5 Võrdlus teiste otsingualgoritmidega

Iteratiivse pikendamise algoritmi tööpõhimõte sarnaneb paljuski juba meile varem vaadeldud otsingualgoritmide tööle ja millest saame tuua välja järgmised sarnasused:

- iga iteratiivse pikendamise algoritmi iteratsioon on laiutiotsing;
- käitumismustri poolest sarnaneb muutumatu hinna otsinguga, kui kaarte kaalud on positiivsed. Mõlemad algoritmid kasutavad liiga palju mälu läbitavate tippude hoidmiseks (laiutiotsingu omadus), kuid see-eest mõlemad algoritmid leiavad optimaalse lahenduse;
- heuristiline algoritm IDA* (*iterative deepening A**) [4, lk 82-83] on iteratiivne pikendamine, kui:
  - $f(n) = g(n)$ – ei kasutata heuristilist funktsiooni (või heuristiline funktsioon h on konstantfunktsioon), vaid ainult teehinna funktsiooni g;
  - ja selle asemel, et iteratiivsel pikendamise algoritmil igal iteratsioonil puu läbimiseks kasutada laiutiotsingut, kasutatakse süvitsiotsingut.

Muutumatu hinna otsingu ja iteratiivse pikendamise algoritmi erinevused:

- muutumatu hinna algoritmi korral läbitakse puud laiuti, eemaldades iga kord eelistusjärjekorrast minimaalse teehinnaga tipu ning lisades tema järglased eelistusjärjekorda; iteratiivne pikendamine see-eest lisab järjendisse ainult need tipud, mille teepikkus juurtipust s jääb jooksva teehinna limiidi l piiridesse, ja kõrvalejäetud tippude hulgast jäetakse meelde selle tipu teepikkus, milleni jõuti kõige lühemat teed pidi juurtipust s.
- muutumatu hinna algoritmil ei pruugi olla eelistusjärjekorras alati ainult üks tipp, milleni on jõutud lühimat teed pidi, vaid mitu tippu, milleni teepikkus juurtipust s võib olla kõigil sama. Sellest tulenevalt eemaldatakse kõik sellised tipud ühekaupa ja nende

järglased lisatakse eelistusjärjekorda (kui vahepeal ei ole leitud lõppolekule vastavat tippu); iteratiivse pikendamise algoritm korral vaadatakse läbi ühe iteratsiooni jooksul korraga kõik sellised tipud, milleni teepikkus juurtipust s jääb jooksva limiidi l piiridesse, kaasa arvatud need tipud, milleni teepikkus on kõigil sama juurtipust s ja ei lisata järjendisse rohkem tippe kui vaja (lisatakse ainult need, mille teepikkus juurtipust s jääb jooksva teehinna limiidi l sisse).

Tabel 3. Mõnede otsingualgoritmide hinnangute võrdlustabel

	Täielikkus	***Optimaalsus***	***Ajaline keerukus***	***Mäluvajadus***
Laiutiotsing	Jah	Jah[6]	$O(b^d)$	$O(b^d)$
Süvitsiotsing	Jah	Ei	$O(b^m)$	$O(bm)$
Muutumatu hinna otsing	Jah	Jah	$O(b^{1+c/m})$	$O(b^{1+c/m})$
Iteratiivne laskumine	Jah	Jah	$O(b^d)$	$O(bd)$
Iteratiivne pikendamine	Jah	Jah	$O(b^d)$	$O(b^d)$

Tabelis 3 on iteratiivse pikendamise algoritmi ajaline keerukus ja mäluvajadus antud halvimal juhul, arvestades, et tegemist on ühtlase puuga, millel on hargnemistegur b ja lahend asub sügavusel d, ning puu kaartele on omistatud ühesugused positiivsed kaalud.

[6] Laiutiotsing on optimaalne, kui teehinnad on ühesugused ja positiivsed.

5 Iteratiivse pikendamise algoritmi rakendamine programmis

Koos iteratiivse pikendamise algoritmi kirjeldamisega valmis antud bakalaureusetöös ka programm, mis realiseerib seda algoritmi ühe probleemi lahendamisel. Iteratiivse pikendamise sooritusvõimet hinnatakse lühima tee leidmisel ja võrreldakse saadud tulemusi muutumatu hinna algoritmiga. Programm on kirjutatud Java programmeerimiskeeles, mis ei pruugi olla tehnoloogiliselt kõige sobivam algoritmide rakendamiseks ja nende töö hindamiseks, kuid on piisav algoritmi töövõimekust hindavate üldiste parameetrite mõõtmiseks ja algoritmi tööpõhimõtete demonstreerimiseks.

Programmi on lisatud ka laiutiotsingu implementatsioon, kuid lühima tee leidmisel on ta ebaotstarbekas, kuna läbib puud süstemaatiliselt (suunaga vasak-parem või parem-vasak ning tasemete kaupa) ja ei arvesta teepikkusi erinevate tippude vahel. Sellest hoolimata leiab ta lõppolekule vastava tipu, mis aga ei pruugi olla optimaalse teepikkusega juurtipust s.

Iteratiivse pikendamise algoritmi oli algselt plaanis rakendada traditsioonilisel Rumeenia linnade kaardil 2 linna vahelise lühima tee leidmiseks, kuid osutus, et antud probleemist algoritmi töö hindamiseks ei piisa, vaid oleks vaja ka algoritmi töötulemusi suuremate probleemide lahendamisel. Seetõttu programm genereerib ise etteantud arvu tippe (linnasid) koos teepikkustega (positiivsed reaalarvud mingis etteantud vahemikus), mis on mällu salvestatud naabrusmaatriksina (vt Lisa 2). Antud naabrusmaatriksist genereeritakse graaf, mis antakse otsingualgoritmile sisendiks.

Iteratiivse algoritmi töötamise käigus leitakse:

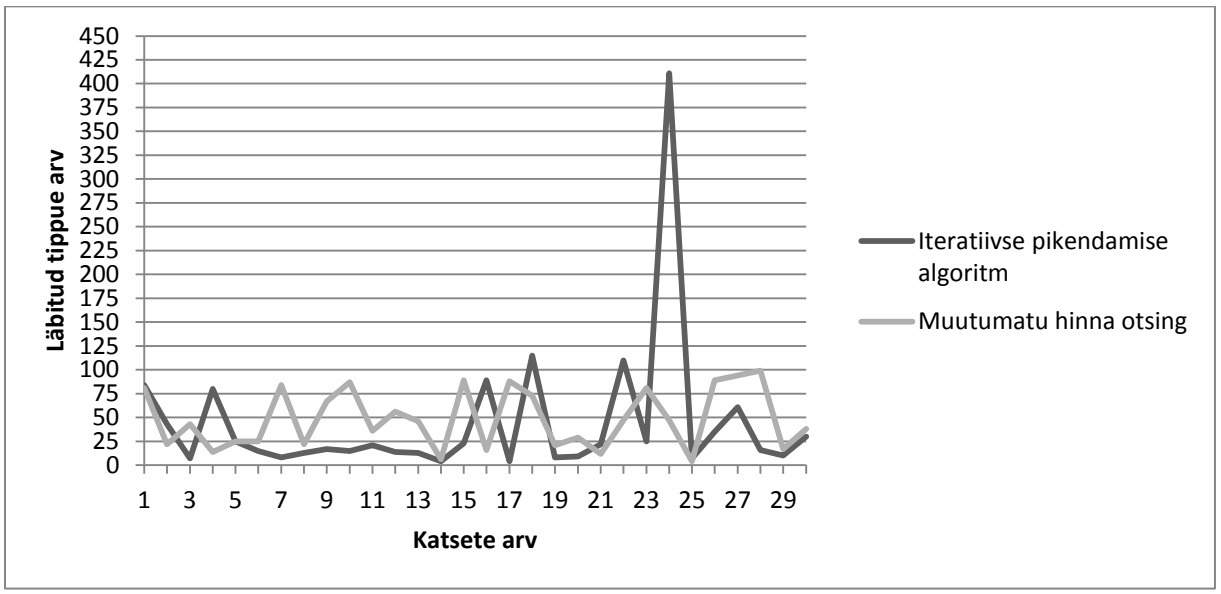
- läbitud tippude arv algoritmi kogu töö jooksul;
- viimasel iteratsioonil läbitud tippude arv;
- iteratsioonide arv;
- algoritmi tööaja kestvus antud probleemi lahendamiseks (mõõdetakse tegelikku antud algoritmi töötamiseks kulunud aega millisekundites, mitte protsessori aja järgi, mistõttu ajalised kestvused ei pruugi olla täpsed).

Muutumatu hinna algoritmi juures vaatleme läbitud tippude arvu ja tööaega.

Kui graafis ei vii algtipust ühtegi teed lahendile vastava tipuni või lahendit ei leidu otsingugraafis, siis programm väljastab tulemuse, et otsing ebaõnnestus.

5.1 Tulemused

Mõlema algoritmi töötulemuste saamiseks tehti 30 katset, kus linnade arv on 1000, ja 10 katset, kus linnade arv on 2000 (vt Lisa 4). Iga kord anti mõlemale algoritmile lahendamiseks ette sama probleemi juhtum.



Joonis 6. Võrdlus kahe algoritmi töötulemuste vahel 30 katses, kus linnade arv oli 2000. Võrreldakse iteratiivse pikendamise algoritmi viimasel iteratsioonil läbitud tippude arvu muutumatu hinna otsingu jooksul läbitud kõikide tippude arvuga.

Jooniselt 6 võib täheldada, et viimasel iteratsioonil oli enamasti (19 katset 30-st) iteratiivse pikendamise algoritm edukam (vähem läbitud tippe) kui kogu muutumatu hinna otsingu töö peale kokku. See-eest esineb ka erandeid, kus olukord oli vastupidine. Põhjus on selles, et erineva probleemi juhtumi korral moodustatakse erinevad otsingugraafid, mille läbimine viimases iteratsioonis iteratiivse pikendamise algoritmiga (nimelt laiutiotsinguga) võib olla edukam kui muutumatu hinna otsinguga (alampeatükis 4.5, kus iteratiivse pikendamise algoritmi võrreldakse muutumatu hinna otsinguga, muutumatu hinna otsing lisab eelistusjärjekorda kõik külastatavate tippude vahetud alluvad, see-eest iteratiivse pikendamise algoritm lisab järjendisse ainult need tipud, milleni teehind juurtipust s jääb jooksva teehinna limiidi l piiridesse). Samas võib iteratiivne pikendamise algoritm sama probleemi lahendada mitu korda halvemini kui muutumatu hinna otsing, mille kohta on esitatud ka selgitus iteratiivse pikendamise algoritmiga seotud peatükis 4.4.

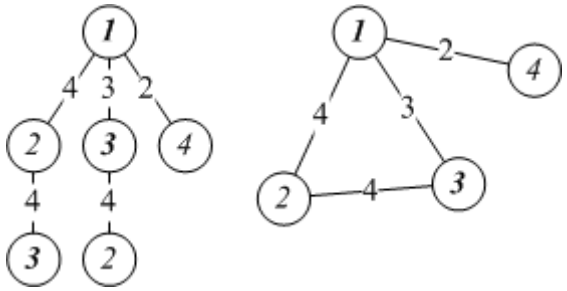
Teisest küljest võib põhjendada iteratiivse pikendamise algoritmi paremust viimases iteratsioonis sellega, et suur osa tööst on selleks ajaks juba tehtud. Viimasel iteratsioonil toimub puu läbimine laiuti, mistõttu lahendile vastav tipp asub nüüd selles puu osas, mille

tipud eelmisel iteratsioonil ei jäänud teehinna limiidi l piiridesse, kuid nüüd kuuluvad ka need tipud külastatavate tippude hulka. Kui laiutiotsing toimub suunaga vasak-parem, siis lõpplahendile vastav tipp jääb nende tippude hulka, mis on juurtipule s lähemal. Teisest küljest võib põhjendada ka iteratiivse algoritmi halvemat esitust võrreldes muutumatu hinna otsinguga, kui lõppolekule vastav tipp asub otsingupuus väga kaugel juurtipust s.

Programmi töö tulemustest (vt Lisa 4) võib välja lugeda ka seda, et kui on tegemist suure otsingupuuga, mille teehinnad on erinevad, siis teehinna limiidi l suurendamine toimub selliselt, et uuel iteratsioonil suurendatakse läbivaadatavate tippude hulka ühe uue tipu võrra. Seega igal iteratsioonil läbitakse otsingupuu uuesti, ainult et iga kord 1 tipu võrra suuremana.

5.2 Illustratiivne näide väiksema probleemi lahendamisest programmis

Vaatleme otsinguprobleemi joonisel 7. Algtipp on tähistatud kui „1“ ja lõppolekule vastav tipp kui „3“. Tippude tähistamisel on kasutatud tähtede asemel numbreid, kuna suure probleemi korral on tippe rohkem kui tähestikus tähti.



Joonis 7. Otsinguprobleem: leida lühim tee tippude "1" ja "3" vahel. Vasakul on esitatud otsingupuu ja paremal otsingugraaf sama probleemi kohta.

Tabelis 4 on toodud iteratiivse pikendamise algoritmi ja muutumatu hinna otsingu töötulemused otsinguprobleemi lahendamisel, mis on esitatud joonisel 7. Kuna tegemist on väiksemat laadi otsinguprobleemiga, siis ajaline keerukus on võrdeline iteratiivse pikendamise algoritmi korral 5 tipu arvuga ja muutumatu hinna otsingu korral 3 tipu arvuga; iteratiivse pikendamise algoritmi töö ajal hoitakse maksimaalselt mälus 2 tippu ja muutumatu hinna otsingu korral 3 tippu. Viimasel iteratsioonil läbib iteratiivse pikendamise algoritm 2 tippu, seevastu muutumatu hinna otsing kogu oma töö ajal antud probleemi lahendamisel 3 tippu.

Iteratiivse pikendamise otsingu lahenduskäik antud probleemi jaoks, kus l on vastavalt teehinna limiit ja m on minimaalne teehind kõrvalejäetud tippudest igal iteratsioonil:

- 1. iteratsioonil:

    - $l = 1$ (vaikimisi alguses); $m = 2$; läbitud tippude arv iteratsioonis: 1;
- 2. iteratsioonil:
    - $l = 2$; $m = 3$; läbitud tippude arv iteratsioonis: 2;
- 3. iteratsioonil:
    - $l = 3$; $m = 4$; läbitud tippude arv iteratsioonis: 2 (lahend leitakse laiuti otsides 2. külastatava tipu juures; muidu iteratsioonis teehinna limiidi l piiridesse jääb 3 tippu).

Oleme saanud samasuguse tulemuse nagu programmgi (vt Tabel 4).

Muutumatu hinna otsingu lahenduskäik antud probleemi jaoks:

- lisatakse järjendisse algtipp „1“, eemaldatakse järjendist ja lisatakse tipu „1“ alluvad: „2“, „3“ ja „4“, millest tipul „4“ on kõige lühem teepikkus juurtipuni „1“;
- eemaldame tipu „4“ järjendist, kuna juurtipust „1“ teepikkus temani on kõige lühem; tipul „4“ alluvaid ei olnud ja tipp „4“ ei ole lahendile vastav tipp, siis eemaldame järjendist järgmise tipu, mille teepikkus juurtipust „1“ on kõige lühem. Selleks on tipp „3“, mis on ka lõppolekule vastav tipp ja oleme leidnud lahendi.

Kogu protsessi jooksul läbisime 3 tippu: „1“, „4“ ja „3“. Seega tulemus ühtib programmi töö tulemusega (Tabel 4).

Tabel 4. Iteratiivse pikendamise algoritmi ja muutumatu hinna otsingu töötulemused (vt Lisa 3) esitatud probleemi lahendamisel (Joonis 7).

Iteratiivse pikendamise algoritm	Läbitud tippude arv viimases iteratsioonis:	2
	Läbitud tippude arv kokku:	5
	Iteratsioonide arv:	3
	Tööaja kestvus:	1,0 (ms)
Muutumatu hinna otsing	Läbitud tippude arv:	3
	Tööaja kestvus:	1,0 (ms)

5.3 Täiendamise ja edasiarendamise võimalused

Käesolev töö on rohkem keskendunud iteratiivse pikendamise algoritmi kirjeldamisele ja põhimõtete tutvustamisele, mistõttu algoritmi uurivam rakenduslik tööosa (peatükk 5) võiks sisaldada endas põhjalikumat analüüsi, kui tuua sisse erinevaid võimalikke uurimismeetodeid, mida antud töös ei ole kasutatud.

Praegune programm on tutvustava eesmärgiga, et implementeerida lühima tee leidmise probleemi lahendamist iteratiivse pikendamise algoritmiga. Programmi realiseerimise käigus

kogutud andmete põhjal on antud esmased järeldused iteratiivse pikendamise algoritmi kohta võrdluses muutumatu hinna otsinguga. Algoritmi töö efektiivsust ja ebaefektiivsust saaks veelgi edasi uurida mõne muu probleemi lahendamisel (näiteks kahe mängijaga mängud; võtmesõnade otsing andmebaasist) ja kuidas algoritm töötab nendel juhtudel ning hinnata ka antud töös esitatud tulemuste paikapidavust.

Programmis on kasutatud iteratiivse pikendamise algoritmiga võrdlemisel muutumatu hinna otsingut, mis on asjakohane, kuna iteratiivse pikendamise algoritm on iteratiivne analoog muutumatu hinna otsingule. Iteratiivse pikendamise algoritmi tööd võiks võrrelda veel mõne muu pimeotsingu algoritmiga ja lisada iteratiivse pikendamise algoritmi implementeeriva programmi koosseisu.

Programmil puudub kasutajaliidese pool, kuna programm tegeleb ainult arvutuslike tegevustega, kuid visualiseerimise ja kasutusmugavuse seisukohalt võiks programmi täiendada veel kasutajaliidese ja modulaarsuse lisamisega.

Kokkuvõte

Käesoleva töö eesmärgiks oli tutvustada ühte võimalikku otsingustrateegiat, iteratiivse pikendamise algoritmi, otsinguprobleemi lahendamiseks, mida seniajani kirjanduses on vähem käsitletud. Konkreetseks otsinguprobleemiks valiti lühima tee leidmine kahe tipu vahel otsingugraafis. Iteratiivse pikendamise algoritmiga on tihedalt seotud ka kaks teist otsingualgoritmi, mida töös tutvustatakse: laiutiotsing ja muutumatu hinna otsing.

Iteratiivse pikendamise algoritm on muutumatu hinna otsingu iteratiivne analoog, mille eesmärgiks on leevendada mäluammendumisega seotud probleeme, mis esineb nii muutumatu hinna kui ka laiutiotsingul. Iteratiivse pikendamise algoritmi idee seisneb selles, et igal iteratsioonil seatakse jooksvaks teehinna limiidiks eelmisel iteratsioonil kõrvalejäetud tippude hulgast selle tipu teepikkus juurtipust, mis on kõige lühem. Iteratiivse pikendamise algoritmi tutvustavas töö osas näidati, et algoritm rahuldab täielikkuse (leiab lahendi kui see leidub otsingupuus) ja lahendi optimaalsuse tingimust. Aja- ja mäluvajaduse hindamine on keerulisem, kuna iteratiivse pikendamise algoritmi juures arvestatakse teepikkusi, ja mitte otsingupuu hargnemistegurit b ja lahendisügavust d. Kuigi enamus probleeme, mida tuleb lahendada, ei ole esitatavad ühtlase otsingupuuna, siis ei saa ka anda täpset hinnangut algoritmi tööle. Iteratiivse pikendamise algoritmi aja- ja mäluvajadust hinnati halvimal juhul ühtlase puu näitel, mille hargnemistegur on b ja üksainus lahend asub sügavusel d ning kaarte kaalude on samasugused. Iteratiivse pikendamise algoritm teeb enamuse tööd ära viimasel iteratsioonil, arvestades, et ta teeb uuesti kõik need arvutused mis eelmistel iteratsioonidelgi. Algoritmi aja- ja mäluvajadus on halvimal juhul asümptootiliselt $O(b^d)$, mis on sama nagu laiutiotsingu aja- ja mäluvajadus. Tegelikult teeb iteratiivse pikendamise algoritm rohkem tööd, kuna iga iteratsioon on eraldi laiutiotsing.

Iteratiivse pikendamise algoritmi üheks suuremaks puuduseks on see, et suure otsingupuu korral võib ta suunduda esimestel iteratsioonidel mitteperspektiivikasse puuharru, külastades tippe, mida ühendatavate kaarte kaalud on mingi väike reaalarv ε (optimaalse lahendi suhtes ja kasutades kaalude esitamiseks suhtarve), ja tehes seejuures väga palju väikseid samme, enne kui ta jõuab sellesse puuharru, kus on küll suuremate kaaludega, kuid hilisematel iteratsioonidel see-eest perspektiivikamad kaared. Iteratiivse pikendamise algoritmi implementeeriva progammi töö tulemustest võis samuti välja lugeda seda, et suure otsingupuu korral, kus kõikide kaarte kaalud on erineva suurusega, võidakse teha väga palju iteratsioone, st. igal iteratsioonil võetakse läbivaatamiseks uusi tippe ühekaupa.

Programmi töötulemustest selgub ka tõsiasi, et erinevate probleemi juhtumite korral toimib iteratiivse pikendamise algoritm ettearvamatult. Näiteks võrreldes muutumatu hinna otsinguga võib iteratiivne enamikel juhtudel olla viimasel iteratsioonil efektiivsem (läbitud tippude arv väiksem) kui muutumatu hinna otsing, kuid mõne juhtumi korral võib ta olla isegi kordades halvem.

Kokkuvõttes võibki öelda, et iteratiivse pikendamise algoritmi puhul on tegemist arvestatava otsingualgoritmiga, kuid tuleb leida õige otstarve, mille korral iteratiivse pikendamise algoritm on kõige efektiivsem. Antud töös kasutati iteratiivse pikendamise algoritmi üldise otsinguprobleemi lahendamiseks (kahe tipu vahelise teepikkuse leidmine otsingugraafis). Algoritm leidis küll optimaalse lahendi, kuid seda sageli liigse aja- ja mäluhulga arvelt.

Evaluation of iterative lengthening algorithm and implementation on solving search problem

Bachelor thesis (6 EAP)

Taivo Teder

Abstract

The purpose of this Bachelor thesis was to introduce a search algorithm, iterative lengthening algorithm, which has not been published widely, to solve the shortest-path problem within the space-graph.

Iterative lengthening algorithm is introduced as an iterative analogue of uniform-cost search. The main purpose of this algorithm was to reduce the memory limitation problems which occur with uniform-cost search. The main idea of the iterative lengthening search algorithm is to use limits on path-cost. In this paper optimality and completeness of the iterative algorithm has been shown. Although space and time complexity are not easily characterised in terms of branching factor b and solution depth d, because iterative lengthening algorithm is dependent on path costs rather than depths. An asymptotic worst-case time and space complexity has been evaluated as $O(b^d)$, when considering an uniform-tree with branching factor b, solution depth d, and unit step costs.

Unfortunately, the iterative lengthening algorithm suffers the same memory limitation problem as the breadth-first search. The main reason is that the iterative lengthening search may choose such strategy at the beginning of the search to explore subtree which consists of many steps with small costs before exploring paths which are greater (than the first path) but more rewarding. Also every iteration of iterative lengthening search is breadth-first search, hence both of these algorithms may go through the same memory limitation problems.

To evaluate iterative lengthening and uniform-cost search algorithm performance on solving greater search problem, an example computer program was implemented. The program results show that iterative lengthening algorithm was considered to be very controversial, because in most cases the visited nodes count of iterative lengthening search at the last iteration was smaller than overall visited nodes count of uniform-cost search. However there was some cases, where the iterative lengthening algorithm performance was much worse. (Because of the problem of memory limitations with breadth-first search.)

In general, the iterative lengthening algorithm is worthwhile to get acknowledged, but it may not be so perfect when solving shortest-path problem.

Kasutatud allikate loetelu

[1] J. Haugeland. *Artificial Intelligence: The Very Idea*, The Massachusetts Institute of Technology, lk 177-178, 1987.

[2] P. Norvig, Stuart J. Russell. *Artificial Intelligence: A Modern Approach (3rd Edition)*, Prentice Hall, 3. peatükk, 2009.

[3] R. E. Korf. *Artificial Intelligence Search Algorithms*, CRC Press LLC, 1996.

[4] Matt Ginsberg. *Essentials of Artifical Intelligence*, Morgan Kaufmann Publishers, Inc., lk 49-52, 1993.

[5] Wikipedia: Lookup table, http://en.wikipedia.org/wiki/Lookup_table (Viimati vaadatud 07.05.0212)

[6] Wikipedia: Space–time tradeoff, http://en.wikipedia.org/wiki/Space-time tradeoff (Viimati vaadatud 07.05.2012)

[7] W. Zhang. *State-Space Search: Algorithms, Complexity, Extensions, and Applications*, Springer, lk 35, 1999.

[8] R. Palm. *Diskreetse matemaatika elemendid*, Tartu Ülikooli Kirjastus, lk 47, 2009.

[9] A. Felner. *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm.* Proceedings, The Fourth International Symposium on Combinatorial Search (SoCS-2011), 2011. http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4017/4357 (Viimati vaadatud 09.05.2012)

[10] M. Koit, T. Roosmaa. *Tehisintellekt*, Tartu Ülikooli Kirjastus, 2011.

[11] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publisehs, Inc., 1998.

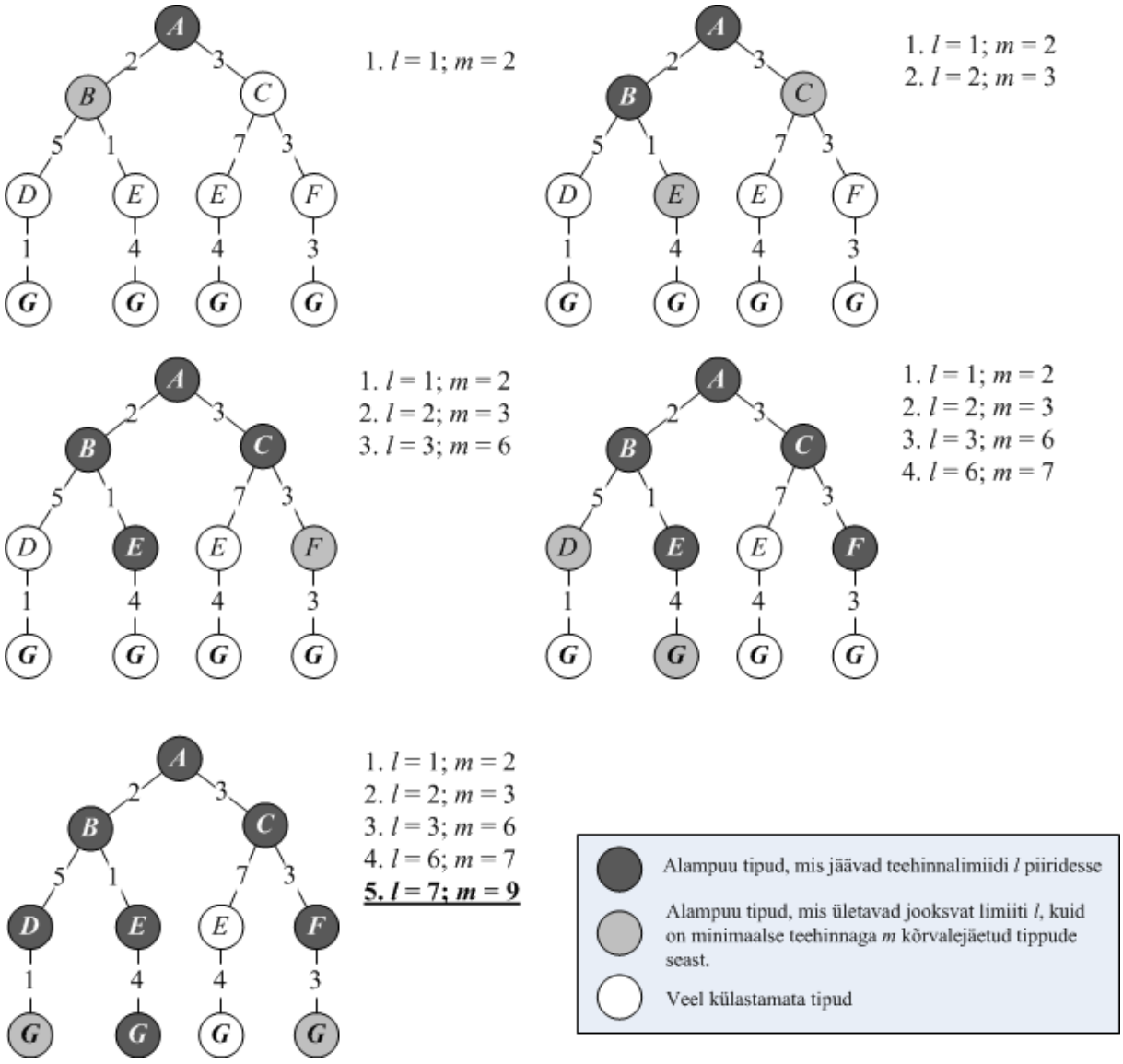
[12] Y. Iribe, K. Katsurada, K, Katsuura, T. Nitta. *Utilization of Suffix Array for Quick STD and Its Evaluation on the NTCIR-9 SpokenDoc Task.* Proceedings of NTCIR-9 Workshop Meeting. Tokyo, Japan, 2011.

[13] M. Shanahan. *Using Reactive Rules to Guide a Forward-Chaining Planner*. Pre-proceedings of the Sixth European Conference on Planning, Toledo, Spain, 2011.

[14] Wikipedia: Greedy algorithm, http://en.wikipedia.org/wiki/Greedy_search (Viimati vaadatud 09.05.2012)

[15] Wikipedia: this (computer programming), http://en.wikipedia.org/wiki/This_(computer_programming) (Viimati vaadatud 12.05.2012)

Lisa 1 Näide üldise otsinguprobleemi lahendamisest kasutades iteratiivse pikendamise algoritmi



Joonis 8. Näide üldise otsinguprobleemi lahendamisest kasutades iteratiivse pikendamise algoritmi

Lisa 2 Linnad ja linnadevahelised teepikkused esitatuna programmis

	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"	"10"	"11"	"12"	"13"	"14"	"15"	"16"	"17"	"1
"1"	0	139	134	155	178	79	124	73	95	131	24	129	175	156	179	175	25	4
"2"	139	0	155	192	198	89	75	80	171	170	177	14	159	139	32	96	188	11
"3"	134	155	0	195	100	9	163	14	136	6	15	111	174	106	144	19	72	35
"4"	155	192	195	0	142	148	81	183	40	17	96	177	68	99	15	16	70	10
"5"	178	198	100	142	0	26	136	14	64	109	134	160	0	139	86	83	46	22
"6"	79	89	9	148	26	0	62	101	43	110	166	147	176	32	128	117	183	14
"7"	124	75	163	81	136	62	0	38	76	156	91	150	5	157	30	93	134	46
"8"	73	80	14	183	14	101	38	0	184	133	52	1	116	131	23	112	135	88
"9"	95	171	136	40	64	43	76	184	0	147	32	179	59	140	123	154	37	15
"10"	131	170	6	17	109	110	156	133	147	0	149	6	147	160	139	8	97	19
"11"	24	177	15	96	134	166	91	52	32	149	0	189	122	31	49	59	56	15
"12"	129	14	111	177	160	147	150	1	179	6	189	0	105	88	115	46	53	58
"13"	175	159	174	68	0	176	5	116	59	147	122	105	0	9	29	82	31	22
"14"	156	139	106	99	139	32	157	131	140	160	31	88	9	0	141	126	8	75
"15"	179	32	144	15	86	128	30	23	123	139	49	115	29	141	0	27	131	54
"16"	175	96	19	16	83	117	93	112	154	8	59	46	82	126	27	0	141	18
"17"	25	188	72	70	46	183	134	135	37	97	56	53	31	8	131	141	0	76
"18"	4	113	35	10	22	143	46	88	156	198	156	58	22	75	54	183	76	0
"19"	39	59	187	58	181	43	22	58	94	13	113	49	1	187	159	26	117	85
"20"	164	162	110	186	67	155	163	48	70	70	38	47	171	184	183	42	141	67
"21"	141	156	136	164	197	190	110	53	170	4	121	64	198	183	128	36	136	32
"22"	115	78	106	21	156	156	177	109	87	61	131	1	145	51	66	95	25	75
"23"	16	63	185	168	75	74	115	51	53	125	151	96	190	146	152	102	133	10
"24"	128	55	59	191	16	92	51	30	180	192	84	87	101	113	122	54	28	18
"25"	112	55	91	130	86	8	94	141	165	2	110	52	197	98	41	108	144	19
"26"	118	183	174	147	49	13	13	124	122	20	191	119	55	90	120	69	126	97
"27"	100	186	35	82	124	3	9	33	97	131	161	92	44	120	148	39	57	46
"28"	67	16	106	111	80	146	10	158	32	107	22	13	84	104	135	141	85	12
"29"	137	119	134	14	188	37	19	23	107	78	79	58	15	106	150	62	78	12
"30"	157	76	157	175	87	97	155	115	57	86	167	150	36	69	7	19	89	22
"31"	188	148	50	62	125	149	106	175	186	128	181	31	89	28	179	153	136	10

Joonis 9. Linnade ja linnadevaheliste kauguste esitamiseks kasutatav naabrusmaatriks programmis

Lisa 3 Programmi töötulemuste väljavõte probleemi lahendamisel

```
        |"1"    "2"     "3"     "4"
---------------------------------------
"1"     | 0     4       3       2
"2"     | 4     0       4       0
"3"     | 3     4       0       0
"4"     | 2     0       0       0

Iterative lengthening
=====================================;
Successful
Last iteration nodes visited: 2
Overall nodes visited: 5
Iterations: 3
Duration: 1.0

Uniform-cost search
=====================================;
Successful, visited: 3
Duration: 1.0
```

Joonis 10. Programmi töötulemuste väljavõte probleemi lahendamisel

Lisa 4 Iteratiivse pikendamise algoritmi ja muutumatu hinna otsingu töötulemuste väljavõte

Tabel 5. Iteratiivse pikendamise algoritmi töötulemused 30 erineva probleemi juhtumi lahendamisel (tippude arv 2000).

	Iteratiivse pikendamise algoritm				*Muutumatu hinna otsing*	
	Läbitud tippude arv	*Läbitud tippude arv viimasel iteratsioonil*	*Iteratsioonide arv*	*Aeg (ms)*	*Läbitud tippude arv*	*Aeg (ms)*
1	10004	84	137	187,0	81	94,0
2	7935	43	158	188,0	22	62,0
3	286	7	30	93,0	43	94,0
4	5156	80	112	156,0	14	47,0
5	4780	25	118	156,0	25	47,0
6	748	15	46	109,0	25	93,0
7	513	8	33	94,0	84	140,0
8	86	13	13	47,0	22	94,0
9	1469	17	55	156,0	67	93,0
10	1384	15	54	141,0	87	109,0
11	3363	21	86	187,0	36	124,0
12	238	14	25	109,0	56	125,0
13	131	13	17	47,0	46	141,0
14	19	4	6	16,0	6	62,0
15	469	23	33	124,0	89	125,0
16	6927	89	142	281,0	16	46,0
17	24	4	6	0,0	88	172,0
18	10787	115	154	234,0	73	140,0
19	408	8	35	78,0	21	63,0
20	301	9	31	78,0	29	93,0
21	483	22	34	78,0	12	32,0
22	5261	110	116	203,0	47	94,0
23	4244	25	90	187,0	81	125,0
24	613	411	40	93,0	47	109,0
25	141	7	18	31,0	4	32,0
26	857	35	46	109,0	89	125,0
27	3865	61	93	203,0	94	125,0
28	2017	16	76	187,0	99	109,0
29	502	10	38	93,0	17	63,0
30	3782	30	90	187,0	38	109,0

Tabel 6. Iteratiivse pikendamise algoritmi töötulemused 30 erineva probleemi juhtumi lahendamisel (tippude arv 3000).

	Iteratiivse pikendamise algoritm				*Muutumatu hinna otsing*	
	Läbitud tippude arv	*Läbitud tippude arv viimasel iteratsioonil*	*Iteratsioonide arv*	*Aeg (ms)*	*Läbitud tippude arv*	*Aeg (ms)*
31	34712	38	270	718,0	141	390,0
32	482	22	33	172,0	147	421,0
33	19526	107	187	422,0	73	265,0
34	8455	18	132	234,0	72	265,0
35	5115	72	119	328,0	165	484,0
36	4814	61	116	234,0	56	281,0
37	6579	82	139	281,0	68	328,0
38	1332	5	47	172,0	23	234,0
39	254	3	26	187,0	193	515,0
40	201	18	18	93,0	6	125,0

Lisa 5 Näiteprogramm

Käesoleva töö raames valminud iteratiivse pikendamise algoritmi rakendav näiteprogramm on avalikult kättesaadav ja allalaetav Google Code repositooriumist aadressil:

- http://code.google.com/p/itlen-algorithm-app/