TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Erich Jagomägis

Erinevad meetodid ja nende võrdlus rändkaupmehe probleemi lahendamisel

Bakalaureusetöö (6 EAP)

Juhendaja: prof. Mare Koit

Autor:.....“.....“ august 2013
Juhendaja:.....“.....“ august 2013

Lubada kaitsmisele
Professor: .....“.....“ august 2013

TARTU 2013

Sisukord

Sissejuhatus

Bakalaureusetöö eesmärk on tutvustada erinevate probleemilahenduse algoritmide tööpõhimõtteid, uurida nende omadusi ning leida nende eelised ja puudused. Töö tulemusi saavad kasutada Tartu Ülikoolis õpetatava kursuse Tehisintellekt I kuulajad toetava ja lisamaterjalina probleemilahendusmeetodite omandamiseks.

Algoritme uuritakse ühe kindla algoritmilise probleemi baasil. Iga algoritmi juures tutvume esmalt selle ülesehituse ja algoritmiliste omadustega. Lisaks võrdleme kõikide algoritmide tööd ja tulemusi omavahel. Uurimistöö sisu eeldab lugejalt arusaama programmeerimises tuntumatest andmestruktuuridest ja terminitest. Mõningate tehnoloogiate tausta seletatakse täpsemalt, juhul kui need ei ole Tartu Ülikooli matemaatika-informaatikateaduskonna informaatika bakalaureuseõppe õppekavas.

Käsitletud algoritme on implementeeritud õpiprogrammis, mis hõlbustab algoritmide tulemuste ja töö visualiseerimist.

Bakalaureusetöö raames tutvume tehisnärvivõrkudega ja nende seast lähemalt Hopfieldi närvivõrguga. Uurime tuntumaid pimeotsingualgoritme: süvitsi- ja laiutiotsingut. Käsitletavate heuristiliste otsingualgoritmide hulka kuuluvad lähima naabri algoritm, parim-enne algoritm, geneetiline algoritm ja libalõõmutamise algoritm.

Esimeses peatükis tutvustame konkreetset probleemi, mis on valitud algoritmide käsitlemiseks. Sellele järgnevalt uurime iga algoritmi eraldi. Kui oleme tutvunud algoritmidega, analüüsime ja võrdleme neid omavahel. Töö lõpus tutvustame valminud programmi ja teeme ka mõningad tähelepanekud ning ettepanekud tulevaseks arenduseks.

Lisades on kirjeldatud matemaatilised funktsioonid, mille abil on võimalik tasandil punktide koordinaatidest luua kaugusmaatriks ja kaugusmaatriksist punktid kahemõõtmelisele tasandile (see on seotud algoritmide käsitlemisel aluseks võetud konkreetse probleemiga). Programm on tööle lisatud CD-l.

Rändkaupmehe probleem

Probleemi võib sõnastada järgmiselt: olgu antud n linna ja nendevahelised kaugused. Ülesandeks on leida selline teekond, mis läbiks iga linna täpselt üks kord ja lõpeks alglinnas, kusjuures summaarne läbitud teepikkus peab olema lühim. Probleem püstitati 1930. aastal ja see on olnud üks intensiivsemalt uuritud probleeme optimeerimises ja algoritmiteoorias. Rändkaupmehe probleem on mittedeterministliku

polünomiaalse keerukusega kombinatoorse optimeerimise probleem arvutiteaduse vallas [17]. See probleem on leidnud kasutust erinevate algoritmide optimaalsuse hindamisel.

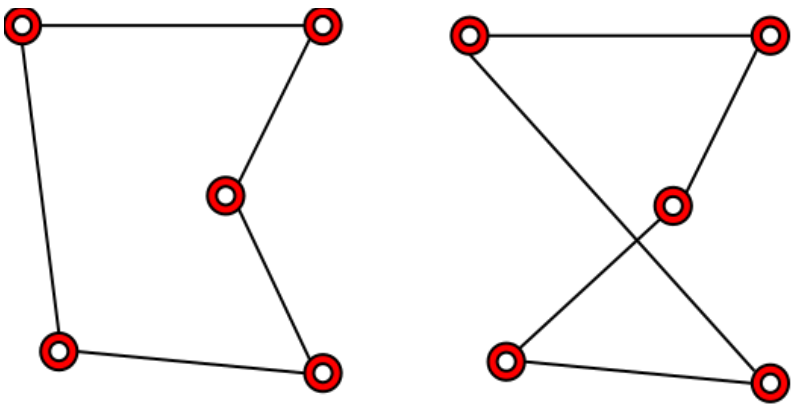
Käesolevas töös eeldame, et igast linnast saab liikuda igasse teise linna, seega kui kujutada andmed graafina, siis on tegu graafide erijuhuga ehk turniiriga. Andmed on salvestatud maatriksisse. Tabel 1 on esitatud näide 4 linna maatriksist, kus igal kohal $a_{i,j}$ on väljendatud teepikkus linnast i linna j.

Tulemus on korrektne siis, kui iga linn on läbitud täpselt 1 kord. Kahe tulemuse võrdlemisel peetakse paremaks sellist teekonda, millel on lühem tee pikkus (s.t kaarte pikkuste summa on väiksem).

Tabel 1. Näidismaatriks linnadevahelistest kaugustest.

Linn/kaugus(km)	Elva	Haapsalu	Kuressaare	Narva
Elva	0	267	308	211
Haapsalu	267	0	155	314
Kuressaare	308	155	0	429
Narva	211	314	429	0

Otsingualgoritmid ei pruugi alati anda kõige lühemat teepikkust. Lühima teekonna leidmine võib olla liiga kulukas, mistõttu vahel võib leppida ka „hea“ tulemusega. Kuidas mõõta kas tulemus on hea? Üks viis on teekonna pikkus. Teine viis on vaadelda teede (kaarte) ristumiste arvu. Mida vähem on kaarte ristumisi, seda parem on teekond. Joonisel 1 võime näha kahte viisi, kuidas läbida viis linna, kuid parempoolsem teekond on halvem ja pikem kui vasakpoolsem teekond.



Joonis 1. Näide heast ja vähem heast tulemusest.

Tulemuste hulk koosneb n! (linnade arvust faktoriaal) elemendist, millest tavaliselt heade tulemuste hulka kuulub väga väike alamhulk.

1. Tehisnärvivõrgud

Antud peatükis anname sissejuhatava ülevaate tehisnärvivõrkudest. Lisaks vaatleme ka tehisnärvivõrkude ühte kindlat liiki- Hopfieldi tehisnärvivõrku. Kasutades Hopfieldi tehisnärvivõrku, proovime lahendada rändkaupmehe probleemi.

1.1. Mis on tehisnärvivõrk?

Tehisnärvivõrk on arvutiteaduses kasutatav algoritmiline mudel, mis on inspireeritud bioloogilistest närvivõrkudest. Tehisnärvivõrgud koosnevad töötlemiselementidest (neuronitest) ja nendevahelistest ühendustest (sünapsidest). Tehisnärvivõrke modelleeritakse kindla probleemi lahendamiseks. Selliste probleemide hulka kuuluvad mustrituvastus, andmete klassifikatsioon ja simuleerimine. Erinevalt tavalistest algoritmidest, tehisnärvivõrke tuleb õpetada näidisandmete põhjal. Tehisnärvivõrgud on suutelised ka õppima reaalandmete põhjal töö käigus. Tehisnärvivõrgude loomist inspireerisid bioloogiliste närvivõrkude tööpõhimõtted. [15]

1.2. Tehisnärvivõrkude kasutamise eelised

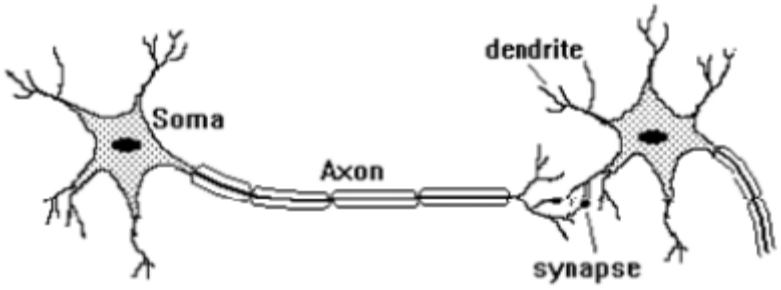
Miks on tehisnärvivõrgud vajalikud ja millal on nad paremad traditsioonilistest algoritmidest? Tehisnärvivõrgud on tihtipeale paremad selliste probleemide lahendamisel, mida on raske lahendada algoritmiliselt, näiteks mustrituvastus. Tehisnärvivõrgud on suutelised interpreteerima keerulisi ja ebatäpseid/ebatäielikke andmeid [14]. Selliste andmete hulka kuulub näiteks mürataustaga helisignaal. Tehisnärvivõrgud on suutelised ka interpreteerima täiesti uusi andmeid, leida nendel seoseid varem esinenud andmetega ja selle põhjal anda ka sobiv väljund [1].

Tehisnärvivõrgud on nagu bioloogilised närvivõrgudki tõrkekindlad. Kuna tehisnärvivõrgud on oma ehituselt hajusad ja mõned osad närvivõrgust võivad olla teineteisest sõltumatud, siis ühe sellise osaga toimunud tõrke korral ei seisku kogu süsteem. Tõrkekindlus koos reaalajas õppimisvõimega annab tehisnärvivõrgule võime ennast parandada, et taastuda tõrkest. Selline protsess võib kahandada kogu närvivõrgu töö kvaliteeti. [1]

1.3. Bioloogiline närvivõrk

Elusorganismi närvivõrk on kogum omavahel ühendatud spetsialiseeritud rakkudest. Selliseid rakke nimetatakse neuroniteks. Neuronid on omavahel ühendatud jätketega (ingl protrusions). Jätkeid on aga kahte liiki. Neuron on suuteline jätkete kaudu võtma vastu või edastama elektro- või keemilisi impulsse. Jätkeid, mille kaudu jõuavad impulsid neuronini, nimetatakse dendriitideks. Kui impulssidest tingitud stimulatsioon ületab kindlat lävendit, siis neuron edastab jätkete kaudu elektro- või keemilise impulsi järgmistele neuronitele. Jätkeid, mis viivad impulssi rakust välja, nimetatakse aksoniteks. Aksonid ühendavad ennast teise neuroni dendriidiga või kinnituvad otse neuroni raku külge. [8]

Neuronite vahele tekivad jätked kogu aeg. Kui neid jätkeid ei kasutata, siis need kaovad, vastasel juhul jätked tugevnevad ja nii kasvabki närvivõrk. Neurobioloogia on valdkond, mis uurib närvivõrkude ehitust ja sellega seotud erinevaid mehhanisme. Joonisel 2 on lihtsustatud kujul näha kahe neuroni vaheline ühendus. Tegelikkuses on ühel neuronil tuhandeid jätkeid [8]. Bioloogilised närvivõrgud on oma ehituselt väga keerulised, kuid üpriski robustsed. Üks väga olulisi iseärasusi närvivõrkude puhul on ka see, et kui kord süsteemis toimub mingi tõrge, siis katkenud infoallikat töödelnud rakud ühendavad ennast teiste infoallikatega. Näiteks sellepärast on pimedaks jäänud inimestel parem kuulmine- aju nägemiskeskus hakkab töötlema informatsiooni, mis tuleb teistest allikatest.

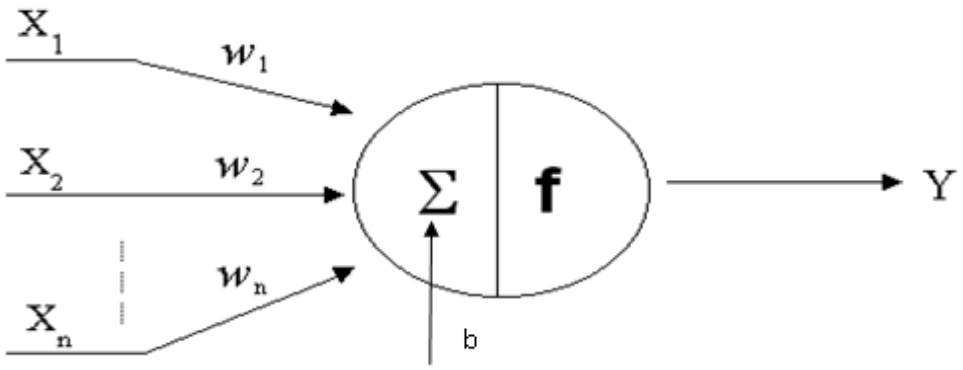


Joonis 2. Bioloogilise närvirakkude vahelise ühenduse ehitus [5].

1.4. Tehisneuroni mudel

Neuron on tehisnärvivõrgu töötluselement. Neuron võtab andmeid vastu, töötleb neid mingil viisil ja annab väljundi. Üldjuhul on tehisnärvivõrgus palju neuroneid, mis on omavahel ühenduses sünapsidega. Igal tehisneuronil võib olla mitu sisendit ja üks väljund. See, kuidas tehisneuron käitub sisendandmete põhjal ja millise väljundi väljastab, määrab funktsioon. Neuroni sisendiks võivad olla väliskeskkond kui ka väljund teistelt neuronitelt. [16:1]

Joonis 3 on kujutatud tehisneuron, kus X on sisend, w - kaal ja Y - väljund, b on väljundfunktsiooni f sisendväärtus [16:1]. Kui tehisnärvivõrk õpib, ta muudab ühendusi neuronite vahel. Õppimine seisneb ennemainitud kaalude väärtuste muutmises. Bioloogilistes närvivõrkudes võivad olla neuronite vahel tugevad või nõrgad ühendused (või üldse puududa), lähemalt on sellest räägitud paragrahvis 1.3. Kaal määrab, kui tugev on ühendus kahe neuroni vahel.



Joonis 3. Tehisneuron [16:1].

Sisendväärtused [X_1 .. X_n] korrutatakse läbi vastavate sisendite kaaludega [W_1 ... W_n] ja tulemused summeeritakse, see moodustab sisendväärtuse b. Funktsioon **f**(b) (seda nimetatakse aktiveerimisfunktsiooniks) määrab, millise väljundi (Y) antud neuron tagastab.

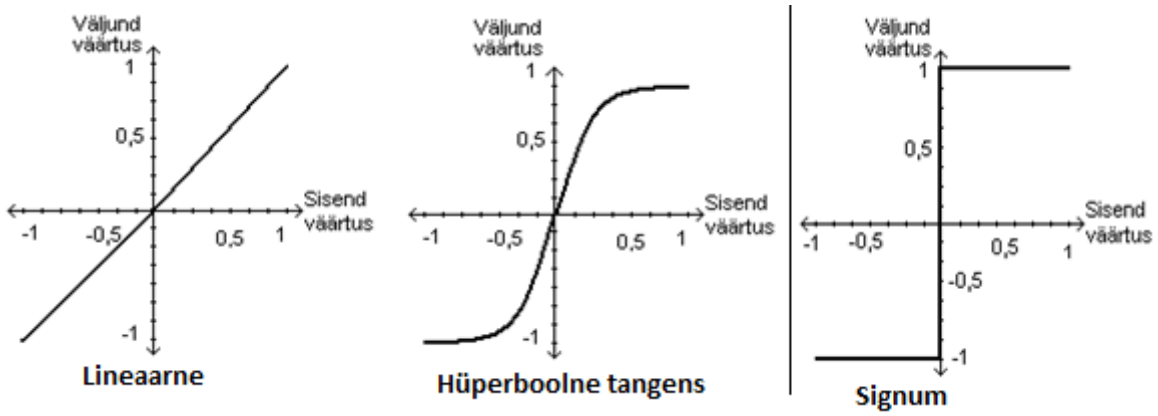
Sisendi väärtust arvutatakse järgmise valemi järgi:

$$b = \sum_{i=1}^{n} X_i * W_i.$$

Tavaliselt on aktiveerimisfunktsioon pidev mittelineaarne funktsioon, aga mõnedel rakendustel võib olla ka lineaarne. Kõige levinumad aktiveerimisfunktsioonid on sigmoidfunktsioonid [16:2].

Sigmoidfunktsioonid on ülemise ja alumise raja (0 ja 1 või -1 ja 1) vahel monotoonselt kasvavad pidevad funktsioonid. Nende funktsioonide põhiliseks eesmärgiks on hoida neuronite väljundid mõistlikes piirides [16:2].

Joonisel 4 on esitatud mõned sageli kasutatavad aktiveerimisfunktsioonid.



Joonis 4. Aktiveerimisfunktsioonid: lineaarne, hüperpoolne tangens, signum [16:2].

Sigmoidfunktsiooni valem aktiveerimise puhul on järgmine [16:3]:

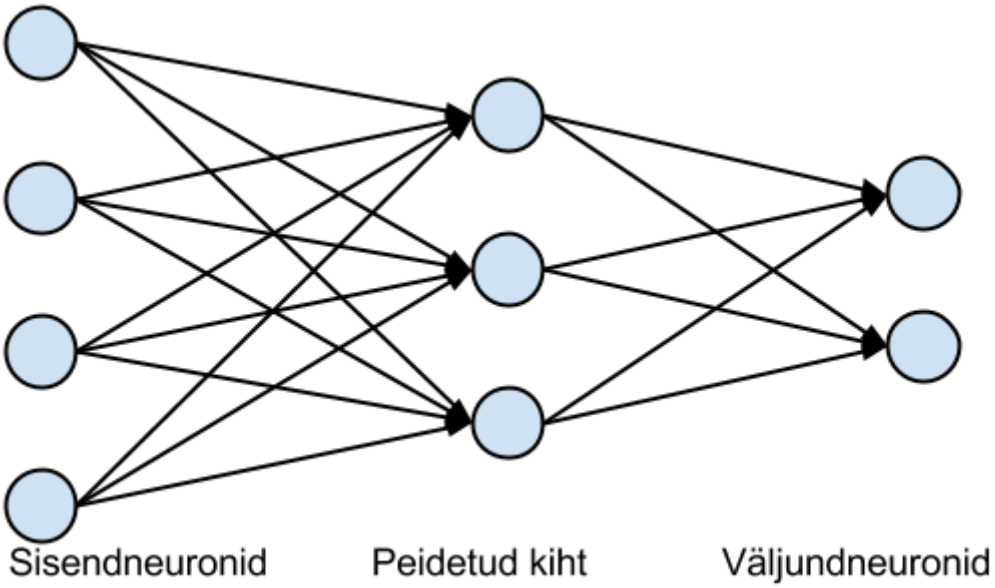
$$f = \frac{1}{1+e^{-x}} . \quad (1)$$

1.5. Tehisnärvivõrkude kihid ja tüübid

Tehisnärvivõrk on struktureeritud kihtidena. Neuronid, mis võtavad vastu sisendandmeid, moodustavad kokku sisendkihi (input layer). Neuronid, mis väljastavad tulemused, moodustavad väljundkihi (output layer). Tehisnärvivõrgud võivad omada sisendkihi ja väljundkihi vahel veel kihte - neid nimetatakse peidetud kihtideks (hidden layer). Tehisnärvivõrgul võib olla vastavalt vajadusele mitu peidetud kihti. See, kuidas neuronid on ühendatud kihtide vahel, oleneb närvivõrgu tüübist. [15]

Otsesuunatud närvivõrk (feed-forward neural net) on selline võrk, kus iga kaar (sünaps), mis väljub neuronist, siseneb järgmise kihi neuronisse. Ühendus toimub sisendkihist väljundkihi suunas [15]. Joonisel 5 võime näha sisendneuronite kihti, ühte peidetud kihti ja väljundneuronite kihti. Nagu joonisel on näha, on tegu otsesuunatud

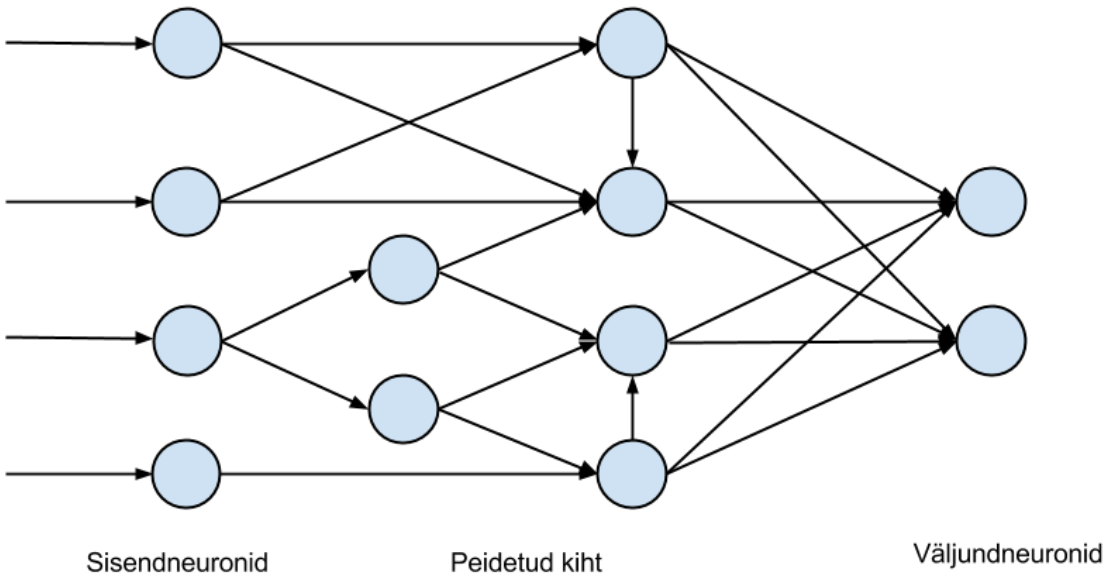
närvivõrguga, sest kõik kaared liiguvad järgneva kihi neuronitesse (nool näitab kaare suunda) [15].



Joonis 5. Otsesuunatud närvivõrk.

Otsesuunatud närvivõrgud on oma ehituselt väga lihtsad, neis puuduvad tagasisidemed ja tsüklid, st. iga taseme neuroni väljund ei mõjuta sama taseme teiste neuronite väljundit. Seda tüüpi närvivõrke kasutatakse enamjaolt mustrite tuvastamiseks. [15]

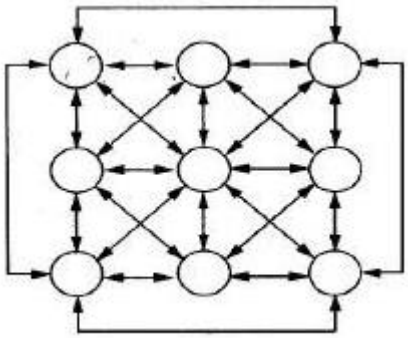
Tagasisideme närvivõrgud (recurrent neural net) on tehisnärvivõrkude eriliik, kus ühendused neuronite vahel võivad olla mõlemas suunas. Erinevalt otsesuunatud närvivõrkudest, tagasisideme tehisnärvivõrgud ei anna tulemust ühe iteratsiooniga. Tagasisideme tehisnärvivõrkude puhul tuleb itereerida seni kaua kuni, iga neuroni väljund enam ei muutu. Seda nimetatakse tasakaalustatud olekuks. Tehisnärvivõrgud jäävad tasakaalustatud olekusse, kuni antakse uus sisend ja nad peavad leidma uut tasakaalustatud olekut uue sisendi järgi. Tagasisideme närvivõrke nimetatakse ka interaktiivseteks ja rekurrentseteks. [15]
Joonisel 6 on toodud tagasisideme tehisnärvivõrk, millel esinevad kõik kolm kihti (sisend-, peidetud ja väljundkiht) ja ühendused neuronite vahel ei ole kitsendatud kihtide järgi. See tähendab, et tagasisideme närvivõrkude puhul on raske kihte eristada ja neuronid võivad kuuluda mitmesse kihti. [15]



Joonis 6 . Tagasisideme närvivõrk.

1.6. Hopfieldi võrk

John Hopfield uuris 1980-ndate aastate alguses teatud tüüpi närvivõrku, mida praegu nimetatakse Hopfieldi võrguks (ingl Hopfield neural net) või Hopfieldi mudeliks [4]. Hopfieldi võrgus ei ole spetsiaalseid sisend- ega väljundneuroneid, vaid kõik neuronid on varustatud nii sisendi kui ka väljundiga ning ühendatud mõlemas suunas (võrdsete kaaludega) teiste neuronitega. Sisendsignaal rakendatakse korraga kõigile neuronitele, väljundid edastatakse üksteisele ning protsess jätkub stabiilse oleku saavutamiseni, mis esindabki võrgu üldist väljundit. Hopfieldi närvivõrku võib vaadelda kui täisgraafi (joonis 7). [3:3]



Joonis 7.Hopfieldi närvivõrk. [3:3]

Neuronid Hopfieldi võrgus on binaarse lävendiga, s.t need võtavad oma olekuks vastu ainult kaks väärtust ja väärtus on määratud selle järgi, kas sisendväärtus ületab neuroni lävendi või mitte. Kõige sagedamini kasutatakse väärtusteks bipolaarseid arve ehk arve hulgast {1, -1} või Boole'i aritmeetika väärtusi {1,0}. Valem (2) on aktiveerimisfunktsioon bipolaarsete arvude jaoks ja valem (3) on Boole'i arvude jaoks. [4]

$$a_i = \begin{cases} 1 \; kui \; \sum_j w_{ij} s_j > \alpha_i \\ -1 \; muidu \end{cases} \quad (2)$$

$$a_i = \begin{cases} 1 \; kui \; \sum_j w_{ij} s_j > \alpha_i \\ 0 \; muidu \end{cases} \quad (3)$$

kus

- w_{ij} on sisendi kaalu väärtus neuronite i ja j vahel,
- s_j on neuroni j olek (väljundväärtus),

- α_i on neuroni i lävendväärtus.

Hopfieldi võrgu neuronitevahelistel ühendustel on järgmised kitsendused [4]:

- $W_{i,i} = 0, \forall i\ korral$(mitte ükski neuron ei tohi ühenduda iseendaga),
- $W_{i,j} = W_{j,i}\ \forall i,j\ korral$ (ühendused on sümmeetrilised).

1.6.1. Energia

Hopfieldi närvivõrku kirjeldab tema olek E (energia e. energy). Kui me tahame lahendada mingit probleemi sellel võrgul, siis me peame defineerima sellise funktsiooni, mis võtab sisendiks selle närvivõrgu ja tagastab mingi hinnangu (energia väärtuse) närvivõrgust. See, kuidas suurust E interpreteerida, sõltub funktsioonist. Hopfieldi närvivõrgul itereerimisel on eesmärgiks saavutada selline E, mida saab ainult parima lõpptulemust väljendava oleku puhul. Tihtipeale on parima E väärtusega selline närvivõrk, mis on oma stabiilses seisundis (itereerimine ei muuda enam E väärtust). Kui ehitame närvivõrku mingi kindla probleemi lahendamiseks, tuleb meil defineerida E nii, et ta kajastaks endas seda probleemi. Kui meil on probleem kodeeritud nii, et mida väiksem on E, seda lähemal oleme lahendusele, siis taandub probleem funktsiooni E= f(x) globaalse miinimumi leidmisele. [4]

1.7. Hopfieldi närvivõrk ja rändkaupmehe probleem

Rändkaupmehe probleemi lahendamiseks defineerime sellise närvivõrgu, kus neuronite arv n võrdub linnade arvu ruuduga. Paigutame need neuronid ruutmaatriksisse nii, et i-nda rea j-ndas veerus asuv neuron vastutab i-nda linna eest positsioonil j tulemuse teekonnas. [3:3]

Iga neuroni väljund asub lõigul [0,1]. Vaatleme seda ehitust järgmises näites Tabel 2).

Tabel 2. Hopfield närvivõrgu korrektse teekonda väljendava närvivõrgu neuronite väljundmaatriks [3:3].

	1	2	3	4
C1	1	0	0	0
C2	0	0	1	0
C3	0	1	0	0
C4	0	0	0	1

Antud maatriksi iga rida väljendab mingit linna C ja veerg i väljendab positsiooni teekonnas. Rea ja veeru ristumiskohal olev 0 või 1 väljendab seda, kas linn C asub või ei asu teekonnas positsioonil i. Mida me siin näeme, on neuronite väljundsignaalid. Antud näite tulemuseks on teekond - C1 – C3 – C2 – C4 - .
Üheks raskuseks on see, kuidas defineerida närvivõrku nii, et tulemus vastaks alati rändkaupmehe probleemi definitsioonile. Vaatleme järgmist tabelit (Tabel 3), kus närvivõrk on tasakaalustunud olekusse, kus me ei saa välja lugeda korrektset teekonda. [3:3]

Tabel 3. Ebakorrektselt tasakaalustunud Hopfieldi närvivõrgu neuronite väljundmaatriks [3:3].

	1	2	3	4
C1	1	0	0	0
C2	0	1	1	0
C3	0	1	0	0
C4	0	0	0	0

Esiteks, teises veerus kaks neuronit väidavad, et nendele vastavad linnad läbitakse teisena. Ilmselgelt ei mahu ühte positsiooni kaks linna. Teiseks, ükski neuron ei väljasta väärtust 1 veerus 4. Milline linn siis neljandana läbida? [3:3]

1.7.1. Energiafunktsioon

Mainitud ebakõlade vältimiseks defineerime närvivõrku analüüsiva funktsiooni E = f(V) nii, et selliste vigade puhul närvivõrk ei saavutaks kunagi lõpetamistingimusi rahuldavat E väärtust [3:4]:

$$E = A_1 \sum_i \sum_k \sum_{j \neq k} V_{ik} V_{ij} + A_2 \sum_i \sum_k \sum_{j \neq k} V_{ki} V_{ji} + A_3 \left[\left(\sum_i \sum_k V_{ik} \right) - n \right]^2 + A_4 \sum_k \sum_{j \neq k} \sum_i d_{kj} V_{ki} \left(V_{j,i+1} + V_{j,i-1} \right) \quad (4)$$

Vaadeldav funktsioon koosneb neljast liidetavast (reeglist), kus iga liidetav on vastava reegli väljundväärtus.

Esimene reegel ($A_1 \sum_i \sum_k \sum_{j \neq k} V_{ik} V_{ij}$) on rea piirang. Närvivõrgu väljundmaatriksi igal real olevate neuronite väljundsignaalide summa peab olema täpselt 1. See reegel vastab eeldusele, et iga linn läbitakse täpselt üks kord [3:4].

Teine reegel ($A_2 \sum_i \sum_k \sum_{j \neq k} V_{ki} V_{ji}$) on veeru piirang. Närvivõrgu igas veerus olevate neuronite väljundsignaalide summa peab olema täpselt 1. See reegel vastab eeldusele, et ühest linnast ei proovita korraga minna mitmesse linna [3:4].

Kolmas reegel ($\mathbf{A_3}$ $\mathbf{[(\Sigma_i\Sigma_k V_{ik})- n]^2}$) on läbitud linnade piirang. Närvivõrgus saab eksisteerida korraga ainult N (linnade arv) neuronit, millel on väljundsignaal 1. Seda reeglit on vaja selle tagamiseks, et läbitud linnade arv oleks N [3:5].

Neljas reegel ($\mathbf{A_4}$ Σ_k $\Sigma_{j\neq k}\Sigma_i \mathbf{d_{kj}V_{ki}(V_{j,i+1} + V_{j,i-1})}$) on lühima tee reegel: närvivõrgust väljaloetav tulemusteekond peab olema lühim [3:5].

Energiafunktsiooni E interpreteerime järgmiselt: mida väiksem on E väärtus, seda parem. Seega meie otsing taandub antud funktsiooni globaalse miinimumi otsimisele.

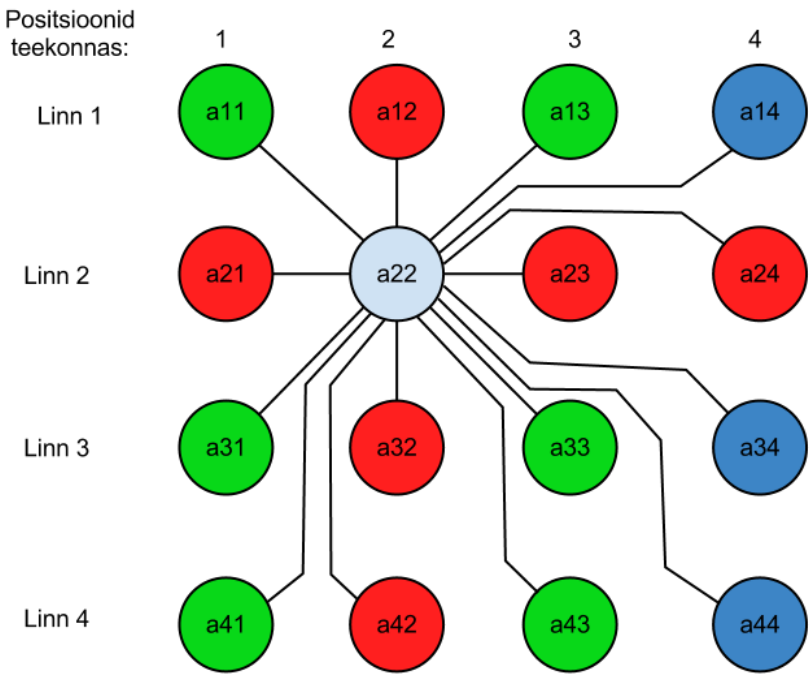
1.7.2. Kaared ja kaalud

Neuronid on omavahel ühendatud kaartega ja igal ühendusel (kaarel) on olemas oma kaal. Kuna meil on n^2 neuronit ja iga neuron on ühenduses iga teise neuroniga,siis kaale on kokku n^4. Kaalud paigutame n^2 järku ruutmaatriksisse. Kaale võime defineerida järgmise valemi järgi [3:5]:

$$\mathbf{W_{ik,\,lj}= -A_1\,\delta_{il}(1-\delta_{kj}) - A_2\,\delta_{kj}(1-\delta_{jl}) - A_3 - A_4\,d_{jl}(\delta_{j,\,k+1} + \delta_{j,\,k-1})}. \quad (5)$$

Siin tähistab sümbol δ (delta) Kroneckeri funktsiooni:
δ_{ij}= 1, kui i=j, ja
δ_{ij}= 0, kui i≠j.
Suurused A1, A2, A3, A4 tähistavad samu konstante, mida kasutame energiafunktsioonis f(x)= E, ning d_{ij} tähistab kaugust linnade i ja j vahel.

Selgitame tehisnärvivõrgu toimimist järgmise näite varal (joonis 8).



Joonis 8. Teiste neuronite rollid ühe neuroni suhtes.

Joonisel 8 vaatleme neuronit a_{22} ja uurime tema puhul, missugused neuronid teda mõjutavad ning kuidas nad seda teevad. Neuroni a_{22} kaalud nendesse neuronitesse, mis on samal real, on –A1 ja nendesse, mis on samas veerus, on -A2. Kuna neuron a_{22} „tahab“ olla oma reas ja veerus ainuke, millel on väljundväärtus 1, siis tema mõjub nendele neuronitele (punased neuronid) negatiivselt. Samuti ka kõik punased neuronid mõjuvad temale negatiivselt. Rohelised neuronid on joonisel need neuronid, mis vastutavad teiste linnade eest naaberpositsioonidel ja neuron a_{22} on „huvitatud“ sellest, et üks nendest oleks väärtusega 1. Reeglina „tahab“ neuron a_{22}, et aktiveeritud naabrid vastutaksid nende linnade eest, mis on temale lähimad. Seega neuron a_{22} mõjub neuronitele naaberveergudes proportsionaalselt linnade vahelise kaugusega. Neuron a_{22} on kaudselt huvitatud ka siniste neuronite aktiveerumisest, sest nii rohelised kui ka sinised neuronid mõjuvad negatiivselt punastele neuronitele oma reas või veerus. Sinised neuronid sümboliseerivad neid neuroneid, mis pole neuroniga a_{22} ei naaberveergudes ega ka samal real või veerus.

1.7.3. Aktiveerimisfunktsioon

Igal neuronil närvivõrgus on olemas oma lävendväärtus a.
Neuroni lävendväärtuse arvutamiseks defineerime funktsiooni [3:6]:

$$a_{ij} = -A1 \sum_{x \neq j} V_{ix} - A2 \sum_{x \neq i} V_{xj} - A3(\sum_{x}\sum_{k} V_{xk} - N) - A4 \sum_{y} d_{iy}(V_{y,i+1}, V_{y,i-1}) . \quad (6)$$

Antud valem koosneb neljast liidetavast.

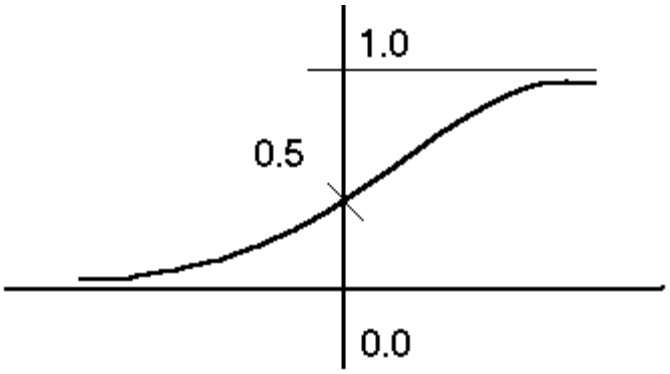
- Esimene liidetav summeerib uuritava neuroniga samas reas olevate neuronite väljundväärtusi ja korrutab summa läbi kaaluga A1 (neuronil a_{ij} on kõikide teiste samal real olevate neuronitega sama kaal).
- Teine liidetav summeerib uuritava neuroniga samas veerus olevate neuronite väljundväärtusi ja korrutab summa läbi kaaluga A2 (neuronil a_{ij} on kõikide teiste samal real olevate neuronitega samakaal).
- Kolmas liidetav summeerib kõikide neuronite väljundväärtusi ja lahutab sellest N, mis on konstant. Tulemus korrutatakse kaaluga A3.
- Neljas liidetav summeerib kaalu A4 ja kahe linna vahelisekauguse korrutised, mida representeerivad kaks neuronit.

NB! Antud valem juba arvestab kaaludega ja reeglitega, seega kaalumaatriksit looma ei pea.
Lävendväärtus tuleb igal iteratsioonil uuendada. Selleks kasutame valemit [3:6]:

$a_{ij\_Uus} = a_{ij\_Vana} + \Delta t(-a_{ij\_Vana} + a_{ij})$, kus Δt on konstant. (7)

1.7.4. Väljundfunktsioon

Neuroni väljund arvutatakse valemist [3:6]:
$V_{ij} = (1 + \tanh(a_{ij}/u0)/2$. (8)
Muutuja u0 on siin konstant, mis määrab hüpertangensi funktsiooni kallakut [3:6]. Väljundfunktsiooni on graafiliselt kujutatud Joonis 9. Ideaalis me tahaksime, et väljundfunktsiooni väärtus oleks kas 0 või 1. Tihtipeale aga võib väljundfunktsiooni väärtuseks tulla näiteks 0.005 või 0.99. Seega peame leidma sellise vahemiku, mille raames me võime väljundiga olla rahul (ümardada) [3:6].



Joonis 9. Väljundfunktsiooni graafik [3:6].

1.7.5. Konstandid ja algväärtustus

Eksisteerib terve hulk erinevat kirjandust, kus seletatakse Hopfieldi närvivõrgu realisatsiooni, aga paraku kasutatakse paljudeks konstantideks erinevaid väärtusi [3:7], [12:16]. Selles töös valminud rakendus (lähemalt 5. peatükis) töötab järgmiste väärtustega, mis on saavutatud suure hulga katsete-eksituste tulemusena:
A1 = 2.0
A2 = 2.0
A3 = 0.2
A4 = 0.9
U0 = 0.7
$\Delta t = 0.001$

Närvivõrgu töökulg sõltub väga neuronite väljundite ja lävendväärtuste algväärtustest. Siiani pole käesoleva töö autor leidnud teooriat, mis selgitaks, kuidas neid väärtusi valida, öeldakse, et tegu on pigem kunsti kui teadusega [13:17].

1.7.6. Algoritm

Kasutame rändkaupmehe probleemi lahendamiseks järgmist algoritmi.
Initsialiseerimine

1. Loome linnade koordinaatidest kaugustemaatriksi D.
2. Loome 2 n-dat järku ruutmaatriksit, ühte salvestame neuronite lävendväärtused ja teise väljundväärtused, nimetame neid vastavalt A ja V.
3. Omistame iga neuronile suvalised algsed lävendväärtused vahemikus (0, 0.5)
4. Omistame igale neuronile väljundväärtused vahemikust (0, 0.1).

Peamine töötsükkel

1. Arvutame närvivõrgu energia E, kasutades energiafunktsiooni (4).
2. Kontrollime, kas E = 0.Kui jah, siis lõpetame tsükli ja loeme välja teekonna.
3. Arvutame iga neuroni uued lävendväärtused, kasutades väljundväärtusi ning funktsioone (6) ja (7).
4. Kasutades uusi lävendväärtusi, arvutame uued väljundväärtused kasutades valemit (8).

Sisuliselt on algoritmi eesmärgiks leida energiafunktsiooni E globaalne miinimum. On oht, et närvivõrk võib tasakaalustuda hoopis lokaalsel miinimumil. Selle tulemusena ei suuda närvivõrk enam leida globaalset miinimumi ja algoritm jääb töötama lõpmatult. Selle vältimiseks on soovitatav kasutada loendurit, mis peab arvet selle üle, mitu iteratsiooni on tehtud. Kui töö on ületanud iteratsioonide arvu poolest mingit läve, siis on mõistlik kas peatada töö või alustada algusest uute algväärtustega.
Antud töö realisatsioonis (lähemalt 5. peatükis) me kasutame natuke muudetud versiooni energiafunktsioonist E, nimelt:

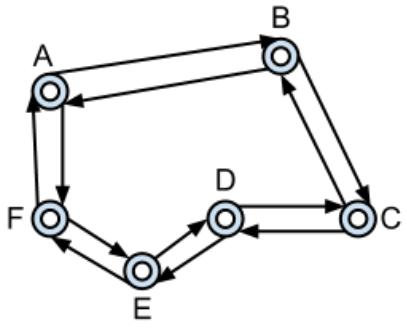
$$\mathbf{E = A_1 \sum_i \sum_k \sum_{j \neq k} V_{ik} V_{ij} + A_2 \sum_i \sum_k \sum_{j \neq k} V_{ki} V_{ji} + A_3 \left[\left(\sum_i \sum_k V_{ik} \right) - n \right]^2 .}$$

Selline funktsioon kontrollib ainult seda, et närvivõrk oleks töö lõpetamiseks kõlblikus olekus. See funktsioon lihtsustab otsustamist, millal tuleks töö lõpetada.

2. Pimeotsingu algoritmid

Selles peatükis vaatleme pimeotsingu algoritme ja nende kasutamist rändkaupmehe probleemi lahendamisel. Pimeotsingu algoritmid otsivad lahendust probleemile olekuruumist. Pimeotsingu algoritmidel ei ole informatsiooni olekuruumi kohta, ainus, mida nad suudavad tuvastada, on asjaolu, kas uuritav olek on lõppolek või ei ole. Erinevalt heuristilistest otsingutest, ei erista pimeotsing olekuid omavahel, st. ei tee olekuruumi läbimisel teadlikke otsuseid otsingusuuna valimisel. Pimeotsingu algoritmid suudavad eristada lõppolekut teistest olekutest. Pimeotsingu algoritmid määravad, millise eeskirja järgi läbida olekuruum. Olekuruumiks on üldjuhul struktureeritud andmemudel, kas graaf või enamlevinumalt puu (erijuht graafist).

Nii süvitsi- kui laiutiotsingu puhul kasutame SULETUD nimestikku. Sellesse nimestikku paigutatakse olekuid, mida on juba vaadeldud. Rändkaupmehe probleemi puhul on aga võimalik läbida üks teekond kahes suunas (joonis 10). Seega, nii süvitsi- kui laiutiotsingu puhul kontrollime, kas uuritava oleku ümberpööratud olek on juba olemas nimestikus SULETUD. Kui olek on juba olemas, siis me seda olekut edasi ei uuri. Kõige paremini saame seda kontrollida, kui me defineerime nimestikuks SULETUD paisktabeli. Paisktabelisse me paigutame võtmeteks olekust genereeritud räsi. Selline optimeering võimaldab olekuruumi vähendada ~50 % võrra.



{A, B, C, D, E, F} ~ {F, E, D, C, B, A}

Joonis 10. Teekonda saab läbida mõlemas suunas.

2.1. Süvitsiotsing

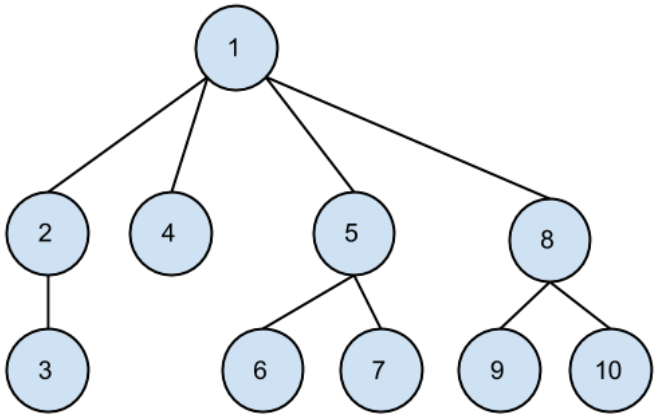
Süvitsiotsing (ingl Depth-first search) on algoritm olekute puus või graafis lõppoleku otsinguks. Süvitsiotsing alustab tööd juurtipust (algolekule vastavast tipust) ja vaatab läbi olekud nii sügavale kui võimalik, nii vara kui võimalik, sellisel viisil süstemaatiliselt läbides kõik harud otsingupuus (või -graafis).

Selle algoritmi üheks peamiseks kitsaskohaks on järgmine asjaolu: kui puu sügavusele ei ole seatud piirangut (puu võib ulatuda lõpmatult sügavale), siis tuleb sätestada kunstlik piirang, millest algoritm ei tohi sügavamale minna. Kui piirang on sätestatud valesti, siis algoritm ei pruugi leida otsitavat lõppseisundit, sest lõppolek võib asuda piirist sügavamal, kust algoritm ei saa otsingut teostada.

Oletame, et otsingupuu sügavus on d ja puu hargnemistegur on b. Sellisel juhul on süvitsiotsingu ajaline keerukus halvimal juhul $O(b^d)$ ja mäluvajadus on lineaarne O(d).

Algoritm: [6]
1) Paiguta algolekule vastav tipp nimestikku AVATUD. Kui algolek on ühtlasi lõppolek, siis on lahend leitud, lõpeta.
2) Kui AVATUD on tühi, siis lahendit ei leidu, lõpeta.
3) Tõsta esimene tipp (n) nimestikust AVATUD nimestikku SULETUD.
4) Leia tipu n kõik vahetud järglased. Kui tipul n pole järglasi, siis mine 2.
5) Paiguta tipu n kõik vahetud järglased nimestiku AVATUD **algusesse**.
Valikuliselt võib ka nii:
Paiguta tipu n vahetud järglased nimestikku AVATUD algusesse juhul, kui neid ei leidu nimestikus SULETUD.
6) Kui mõni tipu n vahetutest järglastest on lõpptipp, siis on lahend leitud, lõpeta. Muidu mine 2.

Joonisel 11 on kujutatud puu tippude läbimise järjekord süvitsiotsingul (nummerdatult).



Joonis 11. Tippude läbimise järjekord süvitsiotsingul.

Nimestikus Avatud hoitakse neid olekuid, mille seast otsitakse lõppolekut. Nimestikku Suletud paigutatakse neid olekuid, mida on juba läbi vaadatud. Nimestik Suletud on kasulik sellisel juhul, kui olekust on võimalik genereerida selline järglane, mida on juba varem vaadeldud. Korduvad olekud on aga ohtlikud sellega, et nendest võivad tekkida tsüklid. Aga kui me teame, et olekust ei saa genereerida järglast, mida on varem uuritud, siis nimestik Suletud polegi vajalik.

2.1.1. Süvitsiotsing rändkaupmehe probleemi lahendamisel

Antud töös on algoritm realiseeritud mitterekursiivselt. Igaks olekuks on korteež indeksitest, millest iga indeks esitab ühte linna linnade maatriksis. Korteež osutub lõppolekuks, kui ta sisaldab kõiki linnu täpselt ühe korra. Nimestiku AVATUD andmestruktuuriks on magasin (stack).

Antud töös on süvitsiotsingu olekuruumiks puu, kus algolekuks on tühi korteež ja puu igal tasemel x on korteežid, milles on seni läbitud x linna. Kui meil on kokku n linna, siis lõppolek asub alati olekuruumis n-dal tasemel. Oleku järglaste genereerimisel vaatame, millised linnad ei ole veel läbitud ja iga sellise linna jaoks teeme koopia uuritavast olekust ja paigutame linna koopia lõppu. Kui meil on y linna läbimata, siis olekul on y järglast.

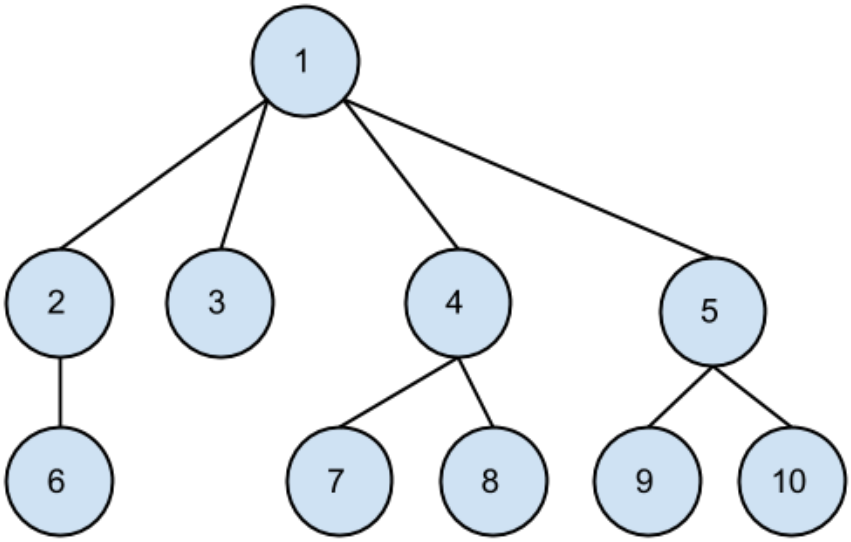
2.2. Laiutiotsing

Laiutiotsing (ingl Breadth-first search) on nagu süvitsiotsingki olekute puu või graafi läbimise algoritm. Laiutiotsing alustab tööd juurtipust (algolekule vastavast tipust) ja vaatab läbi kõik tipud tasemete kaupa, sellisel viisil läbides kõik harud otsingupuus. Teoreetiliselt leiab laiutiotsing kõige kõrgema taseme tulemuse kõige enne. Kui antud otsingupuus on lõppolek olemas, siis teoreetiliselt piisava mälu ja ajaga garanteerib laiutiotsing selle leidmise.

Oletame, et otsingupuu sügavus on d ja puu hargnemistegur on b. Sellisel juhul on süvitsiotsingu ajaline keerukus halvimal juhul eksponentsiaalne ehk $O(b^d)$ ja mäluvajadus on samuti $O(b^d)$.

Algoritm: [6]
Nimestikud: AVATUD - veel läbiuurimata tipud, SULETUD - läbiuuritud tipud.
1) Paiguta algolekule vastav tipp nimestikku AVATUD. Kui algolek on ühtlasi lõppolek, siis on lahend leitud, lõpeta.
2) Kui AVATUD on tühi, siis lahendit ei leidu, lõpeta.
3) Tõsta esimene tipp (n) nimestikust AVATUD nimestikku SULETUD.
4) Leia tipu n kõik vahetud järglased. Kui tipul n pole järglasi, siis mine 2.
5) Paiguta tipu n kõik vahetud järglased nimestiku AVATUD **lõppu**.
Valikuliselt võib ka nii:
Paiguta tipu n vahetud järglased nimestikku AVATUD algusesse juhul, kui neid ei leidu nimestikus SULETUD.
6) Kui mõni tipu n järglastest on lõpptipp, siis on lahend leitud, lõpeta. Muidu mine 2.

Joonisel 12 on näidatud puu tippude läbimise järjekord laiutiotsingul.



Joonis 12. Laiutiotsingu tippude läbimise järjekord.

2.2.1. Laiutiotsing rändkaupmehe probleemi lahendamisel

Antud töös on algoritm realiseeritud mitterekursiivselt. Olekuks on korteež indeksitest, millest iga indeks esindab linna linnade maatriksis. Korteež osutub lõppolekuks, kui ta sisaldab kõiki linnu täpselt ühe korra. Nimestiku AVATUD andmestruktuuriks on järjekord (queue).

Antud töös on laiutiotsingu olekuruumiks puu, kus algolekuks on tühi korteež ja puu igal tasemel x on korteežid, milles on seni läbitud x linna. Kui meil on kokku n linna, siis lõppolek asub alati olekuruumis n-dal tasemel.

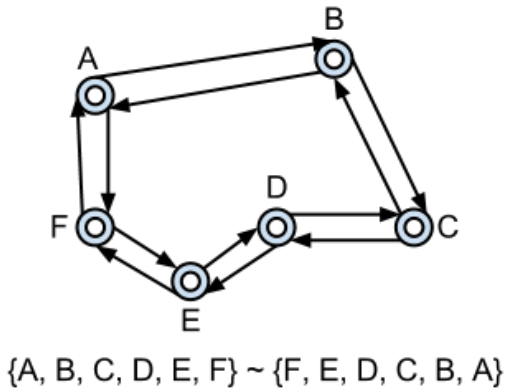
Oleku järglaste genereerimisel vaatame, millised linnad ei ole veel läbitud ja iga sellise linna jaoks teeme koopia uuritavast olekust ja paigutame linna koopia lõppu. Kui meil on y linna läbimata, siis olekul on y järglast.

3. Heuristilise otsingu algoritmid

Heuristilise otsingu algoritmid otsivad lahendust probleemile olekuruumist. Olekuruumiks on üldjuhul struktureeritud andmemudel, kas graaf või enamlevinumalt puu (erijuht graafist). Erinevalt pimeotsingu algoritmidest, kasutavad heuristilised algoritmid mingit kriteeriumi, mille järgi nad hindavad jooksvate olekute perspektiivikust. See võimaldab suunata kogu otsingut selles suunas, kus võib suurema tõenäosusega olla lõppolek. Eeskirja, mille järgi hinnatakse tippude perspektiivikust, nimetatakse heuristiliseks funktsiooniks. [7:81]

Heuristiline otsing ei taga alati parimat tulemust, kuid tagab hea tulemuse mõistliku aja piires.

Lähima naabri algoritmi ja parim enne-otsingu puhul kasutavad algoritmid nimestikku SULETUD. Sellesse nimestikku paigutatakse olekuid, mida on juba vaadeldud. Rändkaupmehe probleemi puhul on aga võimalik läbida üks teekond kahes suunas (joonis 13). Seega, nii süvitsi- kui laiutiotsingu puhul kontrollime kas uuritava oleku tagurpidine olek on juba olemas nimestikus SULETUD. Kui olek on juba olemas, siis me seda olekut edasi ei uuri. Kõige paremini saame seda kontrollida, kui me defineerime nimestikuks SULETUD paisktabeli. Paisktabelisse me paigutame võtmeteks olekust genereeritud räsi. Selline optimeering võimaldab olekuruumi vähendada ~50 % võrra.



Joonis 13. Teekonda saab läbida kahes suunas

3.1. Lähima naabri algoritm

Lähima naabri algoritm (ingl Nearest neighbour algorithm) on heuristiline otsingualgoritm. Lähima naabri algoritmi otsing teostatakse olekuruumis, mille ehituseks on kas graaf või puu. Lähima naabri algoritm on heuristiliste otsingualgoritmide seas üks lihtsamaid algoritme.

Algoritm [7:30]

Nimestikud: AVATUD - veel läbiuurimata tipud, SULETUD - läbiuuritud tipud.
1) Paiguta algolekule vastav tipp nimestikku AVATUD. Kui algolek on ühtlasi lõppolek, siis on lahend leitud, lõpeta.
2) Kui AVATUD on tühi, siis lahendit ei leidu, lõpeta.
3) Tõsta esimene tipp (n) nimestikust AVATUD nimestikku SULETUD.
4) Leia tipu n kõik vahetud järglased. Kui tipul n pole järglasi, siis mine 2.
5) Paiguta tipu n kõik vahetud järglased nimestiku AVATUD algusesse sorteeritult paremuselt kahanevalt.
Valikuliselt võib ka nii:
Paiguta tipu n vahetud järglased nimestikku AVATUD algusesse juhul, kui neid ei leidu nimestikus SULETUD.
6) Kui mõni tipu n järglastest on lõpptipp, siis on lahend leitud, lõpeta. Muidu mine 2.

Lähima naabri algoritmi nimestiku AVATUD defineerimisel kasutame magasini (stack) andmestruktuuri.

Läbides olekuruumi, valib lähima naabri algoritm alati uuritava oleku järglastest kõige soodsama heuristilise väärtusega järglase. Lähima naabri algoritm langetab oma otsuseid lokaalselt, seega ei ole ta suuteline vahetama läbitava struktuuri (puu, graafi jne) haru, kui otsingusuund muutub ebaoptimaalseks. Lähima naabri algoritmi suurimaks probleemiks ongi lokaalse miinimumi ohvriks langemine. Tihtipeale, sattudes lokaalsesse miinimumi, ei suuda algoritm enam sellest väljuda, mis takistab tal globaalse miinimumi leidmist.

3.1.1. Lähima naabri algoritm rändkaupmehe probleemi lahendamisel

Algoritm on oma ehituse poolest väga sarnane eelnevalt uuritud süvitsiotsinguga. Ainus erinevus seisneb selles, et lähima naabri algoritmi puhul järglased sorteeritakse endi seas paremuselt kahanevalt ja siis paigutatakse magasini. Heuristiliseks väärtuseks iga oleku puhul on tema teekonna pikkus. Antud töös on algoritm implementeeritud mitterekursiivselt.

Antud töös on lähima naabri algoritmi olekuruumiks puu, kus algolekuks on tühi korteež ja puu igal tasemel x on korteežid, milles on seni läbitud x linna. Kui meil on kokku n linna, siis lõppolek asub alati olekuruumis n-ndal tasemel.

Oleku järglaste genereerimisel vaatame, millised linnad ei ole veel läbitud ja iga sellise linna jaoks teeme koopia uuritavast olekust ning paigutame linna koopia lõppu. Kui meil on y linna läbimata, siis olekul on y järglast.

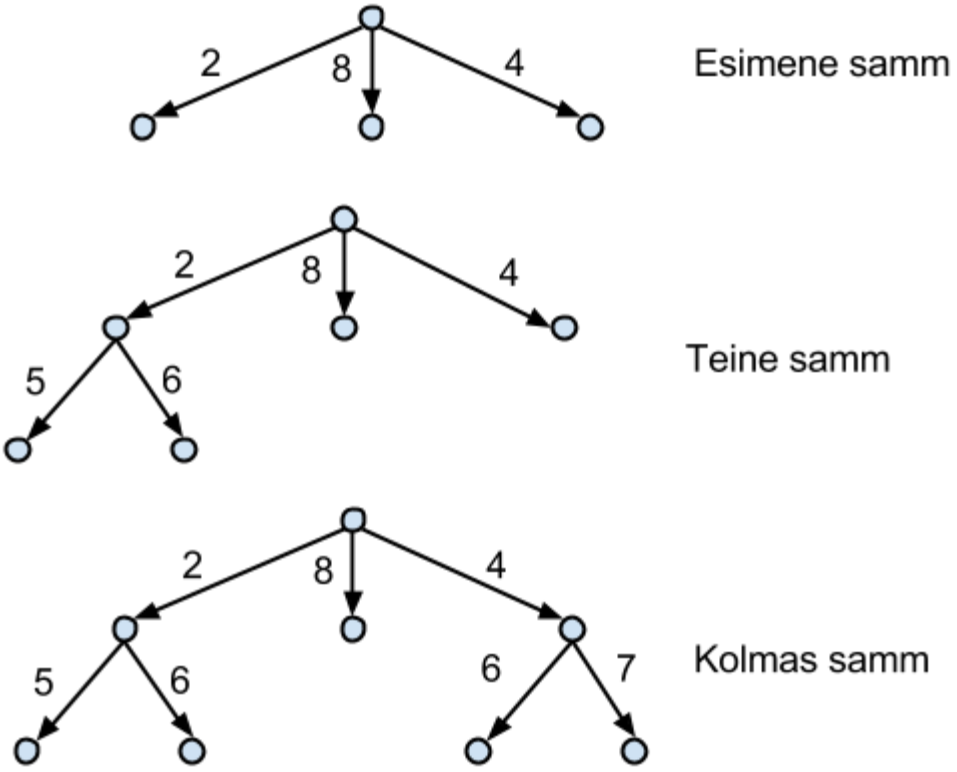
3.2. Parim enne-otsing

Parim enne-otsing (ingl Best-first search) on otsingualgoritm graafil, mis valib tippe, mis on mingi heuristilise funktsiooni väärtuse suhtes kõige parema hinnaga. Puul otsingu teostamisel ei uurita ühte kindlat taset või haru, pigem uuritakse mitut haru/taset paralleelselt, kuni selgub perspektiivikam haru ja otsingut jätkatakse seda haru mööda.

Algoritm: [2]

Nimestikud: AVATUD - veel läbiuurimata tipud, SULETUD - läbiuuritud tipud.
1) Paiguta algolekule vastav tipp nimestikku AVATUD. Kui algolek on ühtlasi lõppolek, siis on lahend leitud, lõpeta.
2) Kui AVATUD on tühi, siis lahendit ei leidu, lõpeta.
3) Tõsta esimene tipp (n) nimestikust AVATUD nimestikku SULETUD.
4) Leia tipu n kõik vahetud järglased. Kui tipul n pole järglasi, siis mine 2.
5) Paiguta tipu n kõik vahetud järglased nimestiku AVATUD **vastavatele kohtadele, nii et nimestik oleks järjestatud heuristiliste väärtuste järgi parematest halvemateni**.
6) Kui mõni tipu n järglastest on lõpptipp, siis on lahend leitud, lõpeta. Muidu mine 2.

Joonis 14 näitab samm-haaval, kuidas tippe uuritakse, tippude juures olevad arvud väljendavad vastava tipu heuristilist hinda.



Joonis 14. Näide sellest, kuidas algoritm läbib puud (arvud näitavad tippude hinda juurest alates).

3.2.1. Parim enne-otsing rändkaupmehe probleemi lahendamisel

Antud töös realiseeriti algoritm mitterekursiivselt. Avatud tippude nimestikuks on java prioriteet-järjekord (Priority-Queue). Kui element paigutatakse AVATUD nimestikku, siis ta paigutatakse sellisesse kohta, et nimestik püsiks sorteeritud. Sinna salvestatakse tippe, mis hoiavad endas nii läbitud linnade teekonda kui ka läbitud teekonna heuristilist hinda. Teekonna pikkust kasutatakse muuhulgas ka selleks, et leida vastav koht elemendile nimestikus AVATUD.

Antud töös on parim enne-otsingu olekuruumiks puu, kus algolekuks on tühi korteež ja puu igal tasemel x on korteežid, milles on seni läbitud x linna. Kui meil on kokku n linna, siis lõppolek asub alati olekuruumis n-ndal tasemel.

Oleku järglaste genereerimisel vaatame, millised linnad ei ole veel läbitud ja iga sellise linna jaoks teeme koopia uuritavast olekust ning paigutame linna koopia lõppu. Kui meil on y linna läbimata, siis olekul on y järglast.

3.3. Otsing geneetilise algoritmiga

Geneetilised algoritmid (ingl Genetic algorithm) on inspireeritud Darwini evolutsiooniteooriast. Probleemi lahendatakse selle evolutsioneerimise teel. [9]

Algoritm alustab tööd nn. Kromosoomide hulgaga, mida kutsutakse populatsiooniks. Iga kromosoom on sisuliselt olek otsinguruumis. Neid olekuid kasutatakse selleks, et genereerida uus populatsioon olekuid, mis on loodetavasti paremate omadustega kui eelmised olekud. Olekuid uue populatsiooni loomiseks valitakse nende heuristilise väärtuse järgi: mida paremate omadustega nad on, seda tõenäolisemalt neid valitakse. Seda protsessi jätkatakse seni, kuni jõutakse lõppolekuni või lõpetamiseks kõlbliku tingimuse täitmiseni.

Üldine algoritm on järgmine: [10]

1. Algus: Genereerida suvaliselt populatsioon olekutest (näiteks kõlblikud tulemused püstitatud probleemile).
2. Arvutada kõikide olekute heuristiline väärtus populatsioonis.
3. Genereerida uus populatsioon, kasutades järgnevaid samme:
    a. valida kaks vanemat (olekut) populatsioonist vastavalt nende heuristilisele väärtusele (mida parem väärtus, seda suurem tõenäosus, et neid valitakse).
    b. Ristata vanemate olekuid, vastavalt ristamise tõenäosusele (kui ristamist ei teostata, siis järglased on täpsed koopiad oma vanematest) ja kasutades ristamisreegleid, saamaks järglased.
    c. Muteerida järglased, vastavalt muteerimise tõenäosusele, kasutades muteerimisreegleid.
    d. Paigutada järglased uude populatsiooni.
4. Kasutada uut populatsiooni edasiseks tööks.
5. Kui lõputingimused on rahuldatud, siis lõpetada töö ja tagastada parima heuristilise väärtusega olek.
6. Minna sammule 2.

Nagu näha, on üldine kirjeldus algoritmist väga abstraktne ja selleks, et seda kasutada mõne spetsiifilise probleemi lahendamiseks (antud juhul rändkaupmehe probleem), tuleb mõningaid osi täpsustada.

Antud töös on meil üheks olekuks korteež linnadest, mille indeks määrab linna läbimise järjekorda. Algolekud on genereeritud suvaliselt ja sisaldavad endas kõiki linnu. Antud töös algoritm lõpetab töö siis, kui viimase 50 iteratsiooniga ei ole genereeritud paremat tulemust.

3.3.1. Ristamisreeglid

On olemas erinevaid viise, kuidas kahe vanema (kahe oleku) ristamise teel genereerida järglasi. Antud töös kasutame nn. ühe punkti ristamist. [11]
Ühe punkti ristamisel me defineerime punkti olekukirjelduses, milleni me võtame osa ühe vanema olekust ja pärast mida võtame ülejäänud osa teise vanema olekust. Kuna meil on olekuks permutatsioon linnadest, siis peame arvestama ka oleku valiidsuse säilitamisega. Selleks võtame ühest olekust alamhulga elemente ja järglasi valides võtame puuduvad elemendid teise vanema olekust. Vaatame selle kohta järgmist näidet.

Olgu vanemad (1 2 3 4 5 6 7 8 9) ja (4 5 3 6 8 9 7 2 1). Ristamisel valime esimese vanema kirjeldusest 5 esimest linna ja teise vanema kirjeldusest 4 esimest sellist linna, mida seni pole valitud:

(**1 2 3 4 5** 6 7 8 9) + (4 5 3 **6 8 9 7** 2 1) = (**1 2 3 4 5** 6 8 9 7).

Muteerimiseks valime kaks elementi järglase olekus ja vahetame nende kohad.
On mõistlik teha selline muteerimine ainult sel juhul, kui selle tulemusel oleku heuristiline väärtus paraneb.
Sellise käitumise eesmärk on luua järglased, mis saavad oma vanematelt paremad omadused ja neid muteerime selleks, et eemaldada halvad omadused. [11]

3.3.2. Elitarism

Elitarism on geneetilise algoritmi omadus, kus uude populatsiooni võetakse mõned parimad (suurima heuristilise väärtusega) olekud vanast populatsioonist. See tagab, et kunagi ei kaotata parimat saavutatud olekut. [12]

3.4. Libalõõmutamise algoritm

Libalõõmutamine (ingl Simulated annealing) on heuristiline otsingualgoritm, mis kasutab tõenäosust probleemi globaalse optimaalsuse otsingul. Algoritm on loodud, saades inspiratsioon metallurgiast, kus metalle kuumutatakse kindla temperatuurini ja siis lastakse aeglaselt jahtuda. Selle abil omandavad metallid kindlaid omadusi.

Erinevalt paljudest teistest otsingualgoritmidest, võib libalõõmutamise algoritm valida olekuruumist ka halvemaid olekuid. See on vajalik selleks, et vältida sattumist lokaalsesse miinimumi, juhul kui see takistab leidmast globaalset miinimumi.
Libalõõmutamise algoritm valib sihilikult mõnikord halvema oleku olekuruumist, vastavalt tõenäosusele (mis sõltub „temperatuurist“) - mida väiksem on temperatuur, seda väiksem on tõenäosus võtta halvem olek olekuruumist. Otsingu vältel temperatuur langeb, kuni otsingusammudel enam ei valita halvemaid olekuid. Töö lõpetatakse siis, kui n sammu vältel ei ole saavutatud uut (paremat) tulemust.

Libalõõmutamise algoritm ei ole kavandatud otsima parimat tulemust, vaid optimaalset ja vastuvõetavat tulemust. Seega ei pruugi libalõõmutamise algoritm vaadata läbi kogu olekuruumi.

Algoritm: [14]

1. Fikseerime algolekuks oleku O ja omistame ta muutujale TULEMUS.
2. Genereerime olekust O uue oleku O_2.
3. Arvutame oleku O ja oleku O_2 heuristiliste väärtuste vahe $d = h(O) – h(O_2)$.

4. Kui $d > 0$, siis omistame O_2 muutujale TULEMUS ja omistame O_2 muutujale O.
   Kui $d < 0$, siis genereerime suvalise arvu A vahemikus (0,1) ja kontrollime, kas $A < e^{d/Temp}$, kui jah, siis omistame O_2 muutujale TULEMUS ja omistame O_2 muutujale O.
5. Vähendame temperatuuri.
6. Lõpukontroll.

Lõpetamiseks võib defineerida erinevaid reegleid. Näiteks, kui x iteratsiooni järel pole paremat tulemust saavutatud, siis lõpetame töö.

3.4.1. Libalõõmutamisealgoritm rändkaupmehe probleemi lahendamisel

Olekuks O kasutame n-elemendilist korteeži, kus n on linnade arv ja korteež sisaldab linnadele vastavad indeksid. Korteežis olevate linnade järjekord väljendab ühtlasi ka teekonna läbimisjärjekorda. Antud töös on algolek genereeritud (samm 1) suvaliselt (kõik linnad on paigutatud suvaliselt järjekorda täpselt üks kord). Uue järglase O_2 genereerimisel (samm 2), me valime olekus O kaks suvalist linnapaari a → b (kus linnale a järgnevalt läbitakse linn b) ja c → d (kus linnale c järgnevalt läbitakse linn d) ja teeme teekonna vahetust nii, et a → c (kus linnale a järgnevalt läbitakse linn c) ja b → d (kus linnale b järgnevalt läbitakse linn d). Sammul 3 kasutatavaks heuristiliseks väärtuseks on teekondade pikkused. Viiendal sammul vähendame temperatuuri 0.01% võrra (hetke temperatuurist). Temperatuuri algväärtuseks valisime 10. Selline temperatuur on valitud seetõttu, et see on katsetuste tulemusel osutunud piisavaks, et lahendus tuleks optimaalne ja samas lahenduse leidmise aeg püsiks suhteliselt madal (rakendusest lähemalt 5. peatükis).

Kuna iga olek (samuti ka algolek) on kõlblik lõppolek (rändkaupmehe probleemi definitsiooni kohaselt), siis pole lõpetamise tingimuses vajalik vaadata, kas olekud reeglitele vastavad. Antud töös kasutame lõputingimusena järgmist reeglit: kui viimase 50 iteratsiooniga pole saavutatud paremat tulemust, siis lõpetame töö.

4. Analüüs

Antud peatükis analüüsime käsitletud algoritmide otsingutulemusi ja võrdleme tulemuste leidmise aega. Kõik algoritmid on võrreldud samadel kaartidel, vältimaks igasuguseid kaardi erinevustest tulenevaid ebatäpsusi. Võrdlusuuringul kasutasime 14 kaarti erineva linnade arvuga. Kui algoritm ei tulnud toime mingi kaardiga, siis suurema linnade arvuga kaartidel teda enam ei vaadeldud, sest ressursikulu on suurema linnade arvuga kaardile suurem, mis tähendab, et ka seal ei pruugi need algoritmid toime tulla. Iga algoritmi juurde on lisatud ka mõõtmete tabelid. Veerus „Aeg” märgitakse algoritmi kogu töö kulgemise aega. Veerus „Parima tulemuse aeg” märgitakse viimase parima tulemuse leidmise ajahetke (töö algusest). Viimases veerus „Teepikkus” märgitakse parima tulemuse teekonna pikkuse. Viimast on mõistlik võrrelda erinevate algoritmide tabelite vahel . Need andmed on lõpus koondatud ka ühtseks graafikuks.

4.1. Hopfieldi tehisnärvivõrk

Antud töös eksperimenteerisime kõigepealt Hopfieldi närvivõrguga rändkaupmehe probleemi lahendamisel. Hopfieldi närvivõrgu ajalist keerukust on raske määrata, sest see sõltub paljudest parameetritest. Umbes 20% kordadest ei suutnud tehisnärvivõrk leida tasakaalus olekut, seega ei produtseerinud ka valiidset olekut. Kui aga lõppolek sai genereeritud, siis ei pruukinud see olla optimaalne ja üldjuhul ei olnudki. Mälukasutuse seisukohast on tehisnärvivõrk konstantse mäluvajadusega, tööks on vaja kaks n-ndat järku ruutmaatriksit. Lisaks ei läbi tehisnärvivõrk olekuruumi, nagu seda teevad teised töös käsitletud algoritmid, pigem saavutatakse lõppolek närvivõrgu tasakaalustamise teel.

Tabelis 4 võime jälgida ajakulu, mis linnade arvu suurenedes kasvab. Iga tasakaalustamise iteratsioon on keerukusega n^2 ja iteratsioonide arv on varieeruv. Tavalisel personaalarvutil on võimalik antud töös loodud rakendusega lahendada kuni 80-linnalisi probleeme.

Tabel 4. Hopfieldi närvivõrgu lahendamiseks kulunud aeg (millisekundites).

		Hopfield	
Linnade arv	Aeg (ms)	Parima tulemuse aeg (ms)	Teepikkus
5	0,35	0,35	2605
6	0,71	0,71	1613
7	0,8	0,8	2430
8	1,66	1,66	2227
9	1,61	1,61	2670
10	3,17	3,1	3701
15	15,42	15,42	4607
25	72,11	72,11	9325
40	872	872	14174
80	39913	39913	27937

4.2. Pimeotsingu algoritmid

Pimeotsingu algoritmid ei anna teadlikke hinnanguid olekutele, vaid lihtsalt läbivad olekuruumi mingi kindla reeglistiku järgi. Mida suurem on olekuruum, seda vähem sobivad pimeotsingu algoritmid probleemi lahendamiseks.

Rändkaupmehe probleemi olekuruum on suurusega n! , mis tähendab, et olekuruum kasvab faktoriaalselt. Pimeotsingu algoritmid ei tule sellistes tingimustes enam hästi toime.

4.2.1. Süvitsiotsing

Süvitsiotsingu algoritm on lineaarse mäluvajadusega, mis võimaldab tal teoreetiliselt läbida suuremate probleemide puhul kogu olekuruum. Sellegipoolest võib terve olekuruumi läbivaatamine osutuda väga ajanõudlikuks tegevuseks. Rändkaupmehe probleemi puhul on süvitsiotsingu eelis aga see, et esimese tulemuse saab ta kätte alati n-dal sammul puu sügavusel n. Kuid siis jääb algoritm läbi vaatama olekuruumi puu kõige madalamat vasakpoolsemat osa. Kui tulemus on aga olekuruumis teisel pool (paremal), siis läbib süvitsiotsing kogu olekuruumi, enne kui ta juhuslikult leiab parima tulemuse.

Tabel 5 esitab süvitsiotsingu ajakulude tabel erineva linnade arvu suhtes. Võime tähele panna, et alates 15 linna komplektist, kasvas olekuruum nii suureks, et süvitsiotsing ei olnud enam võimeline kogu olekuruumi läbima. Tabeli 5 koostamisel ei lastud süvitsiotsingul otsida kauem kui 100 sekundit. Sellest hetkest alates, kui algoritm enam ei suutnud läbida kogu olekuruumi, muutusid tulemused vägagi kaootiliseks (kord võis tulla ajapiires hea tulemus, teine kord suvaliselt genereeritud lõppolekust raskesti eristatav).

Tabel 5. Süvitsiotsingu lahendamiseks kulunud aeg (millisekundites).

		Süvitsi otsing	
Linnade arv	Aeg (ms)	Parima tulemuse aeg (ms)	Teepikkus
5	0,22	0,04	1747
6	0,59	0,17	967
7	2,76	2,64	1527

8	20,71	4,1	2032
9	163,94	11,57	1833
10	1518,7	1102,8	1966
15	100000	91403	2981
25	100000	66178	7450
40	100000	70150	12319
80	100000	21075	28377
100	100000	68672	33071
200	100000	99471	68364
400	100000	30966	151114
800	100000	39098	274886

4.2.2. Laiutiotsing

Laiutiotsingu mäluvajadus on eksponentsiaalne, mis teeb ta ebaefektiivseks algoritmiks probleemide puhul, mille olekuruum kasvab faktoriaalselt. Rändkaupmehe probleemi puhul see ongi nii.

Tabel 6 esitab laiutiotsingul rändkaupmehe probleemi lahendamiseks kulunud aega. Võime tähele panna, et parima tulemuse leidmise aeg on suhteliselt lähedane kogu olekuruumi läbimise ajaga. Probleemiks on see, et laiutiotsing vaatab läbi kõik n-1esimest olekuruumi taset, enne kui ta jõuab n-nda tasemeni, kuhu on paigutatud kõik lõppolekud. Selline otsinguruumi läbimisviis on aga ebaefektiivne. Seetõttu ei ole laiutiotsing suuteline suurema linnade arvuga probleemi lahendamisel toime tulema tavalisel personaalarvutil- selleks pole piisavalt mälu. Sellegipoolest, kui probleem on piisavalt väike, leiab laiutiotsing alati parima tulemuse üles.

Tabel 6. Laiutiotsingu lahendamiseks kulunud aeg (millisekundites).

		Laiutiotsing	
Linnade arv	Aeg (ms)	Parima tulemuse aeg (ms)	Teepikkus
5	1,74	1,7	1747
6	6,59	3,99	967
7	7,78	3,77	1527
8	35,66	33,57	2032
9	278,42	271,36	1833
10	2955	2220,33	1966

4.3. Heuristilised otsingualgoritmid

Võrreldes pimeotsingu algoritmidega, on heuristilistel otsingualgoritmidel parem olekuruumi läbimisviis. Paljud heuristilised otsingualgoritmid ei läbigi kogu olekuruumi, mis tähendab, et nad ei saa garanteerida parima tulemuse leidmist. Sellegipoolest, kui juhtimisstrateegia on hea ja heuristiline funktsioon on defineeritud hästi, on heuristilised otsingualgoritmid suutelised leidma optimaalse tulemuse mõistliku mälu- ja aja piirangutega.

4.3.1. Lähima naabri algoritm

Rändkaupmehe probleemi lahendamisel on Lähima naabri algoritm kiiremaid algoritme, mille esimene väljastatav tulemus on üpriski hea. Mäluvajaduse ja ajakulu seisukohalt on antud algoritm sama, mis tavaline süvitsiotsing.Kuid erinevalt süvitsiotsingust on Lähima naabri algoritmi tulemused üldiselt palju paremad (eriti kui ei jõuta ajapiirangute tõttu terve oleku ruum läbida) ja tihtipeale on esimene tulemus hea. Tabel 7esitab antud algoritmi erinevate linnade arvuga probleemi lahendamise ajakulu ja parima tulemuse teepikkuse. Võime panna tähele, et parima tulemuse leidmise aeg ja parima tulemuse teepikkused on palju paremad teistest algoritmidest eriti sel juhul, kui olekuruum ei ole täielikult läbi uuritud (aeg ületab 100 sekundit).

Tabel 7. Lähima naabri algoritmi rändkaupmehe probleemi lahendamise ajad (millisekundites) ja teepikkused.

		Lähima naabri otsing	
Linnade arv	Aeg	Parima tulemuse aeg(ms)	Teepikkus
5	1,63	0,24	1747
6	1,44	0,2	967
7	3,5	1,3	1527
8	22,62	4	2032
9	183	38	1833
10	1700	252	1966
15	100000	13938	2433
25	100000	334,74	3609
40	100000	95030	4578
80	100000	21792	6047
100	100000	1628	7190
200	100000	45,2	9905
400	100000	363	13239
800	100000	25153	19262

4.3.3. Parim enne-otsing

Erinevalt eelnevalt vaadeldud algoritmidest ei ole parim enne-otsing algoritmiliselt nii disainitud, et ta töö oleks piiratud olekuruumi mingi kindla haruga, st. algoritm saab iga hetk minna ühelt harult üle teisele. Paljude probleemide puhul annab selline vabadus parim-enne-otsingule märkimisväärse eelise nende algoritmide ees, mis teostavad otsingut otsinguruumi mingis kindlas alas enne teise alasse minekut (nt. 15-mängu lahendamisel). Millist haru olekuruumis aga uurida, määrab heuristiline funktsioon. Paraku on rändkaupmehe probleemile raske määrata selline heuristiline funktsioon, et ta suudaks olekuruumi läbides aina rohkem kitseneda ühele heale harule. Vastupidiselt on parim-enne-otsingualgoritmil kalduvus käituda nagu laiutiotsing- tihti läbitakse kõik olemasolevad olekud (va. äärmuslikult ebaoptimaalsed), enne kui minnakse järgmisele tasemele. Seetõttu jääb parim-enneotsing hätta suurema linnade arvu korral (kui olekuruum kasvab väga suureks). Muidugi on võimalik optimeerida heuristilist funktsiooni, mis aitaks algoritmil paremini filtreerida need olekud, kus ei pruugi leiduda head tulemust. Sellise heuristilise funktsiooni leidmine ei kuulunud antud töö ülesannete hulka.

Tabelis 5 võime tähele panna, et parim-enne-otsingu leitud teekondade teepikkused võrreldes teiste algoritmidega on suhteliselt head, kuid linnade arvu kasvamisega kaasneb ka suurenev ajakulu. Alates 40 linnast hakkab parim enne-otsing käituma sarnaselt laiutiotsingule ja selle tulemusel aja- ja mälukulud kasvavad eksponentsiaalselt.

Tabel 8. Parim-enne otsingu rändkaupmehe probleemi lahendamise ajad (millisekundites).

		Parim enne-otsing	
Linnade arv	Aeg (ms)	Parima tulemuse aeg (ms)	Teepikkus
5	1,73	0,34	1747
6	3,9	0,47	967
7	9,84	1,05	1527
8	34,14	9,45	2032
9	282,34	18,29	1833
10	2712,18	145,22	1966
15	100000	21,56	2142
25	100000	622	3273
40	100000	15454	4051

4.3.4. Geneetiline algoritm

Võrreldes pimeotsingu algoritmidega ja parim-enne otsinguga on geneetiline algoritm tunduvalt efektiivsem. Mäluvajaduse seisukohalt vajab geneetiline algoritm niipalju mälu, kui suur on selle kromosoomide (olekute) hulk. Antud töös on olekute hulk 16-elemendiline. Ajakulu on geneetilisel otsingualgoritmil eelmistega võrreldes samuti palju väiksem. Kuid suurema linnade arvu korral hakkavad ka tulemused progressiivselt halvenema. Seda võime jälgida tabelist 10. Tõenäoliseks põhjuseks on halb ristamisalgoritm.

Tabel 9. Geneetilise algoritmi rändkaupmehe probleemi lahendamise ajad (millisekundites).

		Geneetiline otsing	
Linnade arv	Aeg (ms)	Parima tulemuse aeg(ms)	Teepikkus
5	14,85	6,23	1747
6	20,7	4,9	967
7	25,41	4,4	1527
8	28,19	4,8	2063
9	28,69	4,2	2003
10	30,62	4,14	2121
15	74,97	22,15	2841
25	259,85	223,5	3347
40	618	522	4020
80	2212	1654	6823
100	3422	2330	9147
200	18444	9878	26213
400	38003	32629	77300
800	100000	99242	184140

4.3.5. Libalõõmutamise algoritm

Antud töös on osutunud libalõõmutamise algoritm parimaks algoritmiliseks valikuks rändkaupmehe probleemi lahendamisel. Võrreldes teiste algoritmidega on libalõõmutamise algoritm ainus, mis suudab üle tuhande linna puhul genereerida optimaalse tulemuse (teede ristumised puuduvad) suhteliselt lühikese ajaperioodi jooksul.

Libalõõmutamise algoritm on samuti ka väikseima mäluvajadusega. Antud töös on libalõõmutamise algoritmil vaja mälus hoida 2 olekut (teekonda).

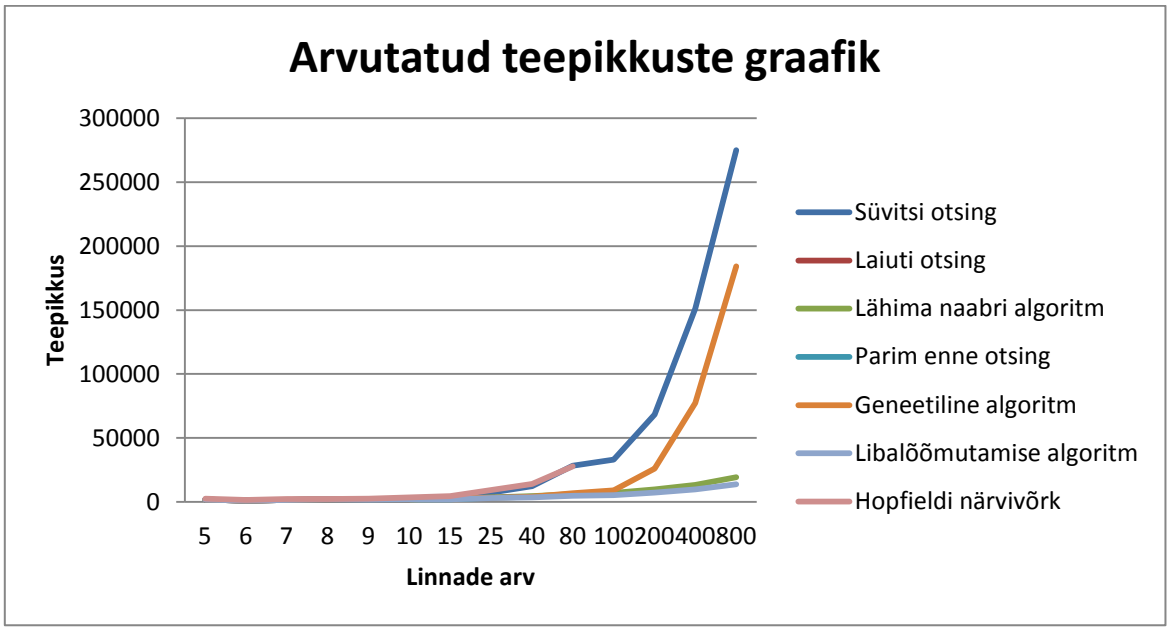
Tabelis 10 võime tähele panna, et isegi 800 linna puhul jõuab libalõõmutamise algoritm lõpetada oma tööd alla 100 sekundi. Võrreldes teiste algoritmidega on libalõõmutamise algoritm leidnud parimaid tulemusi selles töös käsitletud algoritmide seas.

Tabel 10. Libalõõmutamise algoritmi ajad (millisekundites) rändkaupmehe probleemi lahendamisel.

		Libalõõmutamise otsing	
Linnade arv	Aeg (ms)	Parima tulemuse aeg (ms)	Teepikkus
5	4,9	0,1	1747
6	2,42	0,1	967
7	3,2	0,06	1527
8	4,5	0,16	2032
9	6	0,7	1833
10	6,4	0,2	1966
15	15,74	0,36	2142
25	22	6,5	3179
40	51,44	11,69	3618
80	112	188	4766
100	126	529	5299
200	743,1	1039	7245
400	4849.16	25487	9744
800	48756	97328	13785

4.4. Üldine ülevaade

Graafikul 1 võime vaadelda kõikide algoritmide leitud parimate teekondade pikkusi vastavalt linnade arvule. Iga linnade arvu kohta on genereeritud kaart ja kõik algoritmid lahendasid rändkaupmehe probleemi samal kaardil. Libalõõmutamise algoritm ja lähima naabri algoritm on läbinud kõik testid ja seejuures tulemused tulid head. Geneetiline algoritm ja süvitsiotsing läbisid samuti kõik testid, kuid suuremate probleemide korral võime täheldada, et nende tulemused on eksponentsiaalselt halvenevad. Selle põhjuseks on aga teede ristumiste rohkus teekonnas.



Graafik 1. Algoritmide leitud parimate teekondade pikkused.

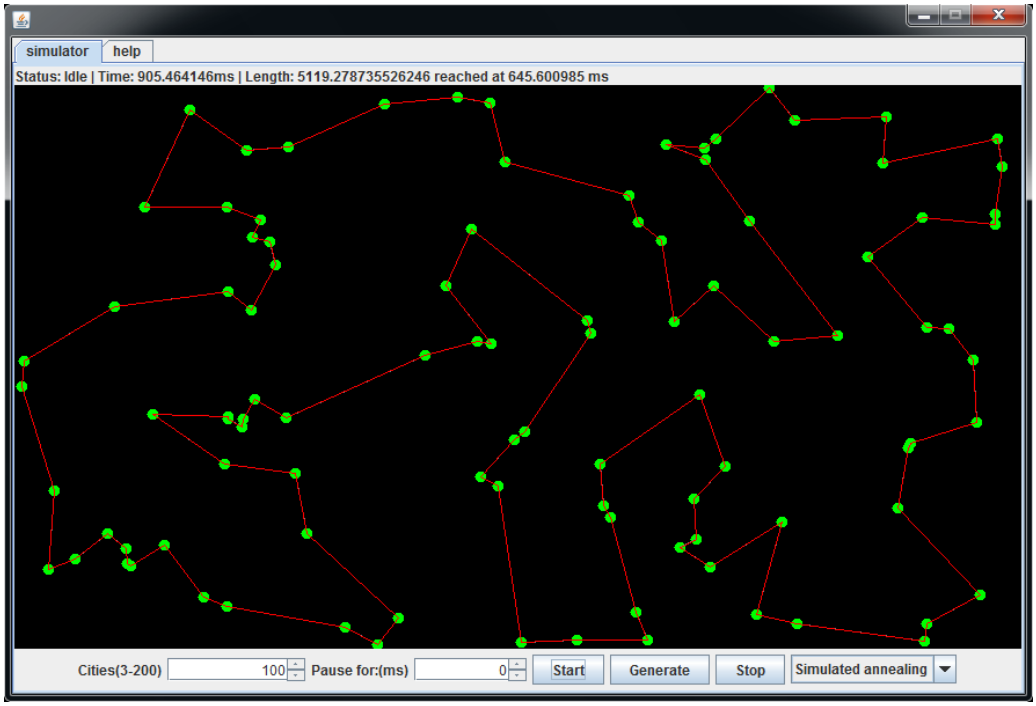
Kõige paremini sai rändkaupmehe probleemi lahendamisega hakkama libalõõmutamise algoritm. Mäluvajaduselt on algoritm konstantne ja ajakulu on võimalik reguleerida, muutes algoritmi jahtumiskiirust. Muidugi tuleb seejuures olla ettevaatlik, liiga kiire jahtumine võib anda halvema tulemuse.

5. Rakendus

Antud töö käigus on valminud ka õpiprogramm. Programm võimaldab kasutajal valida antud töös käsitletud algoritmide vahel. Lisaks on võimalik kasutajal määrata linnade arvu ja genereerida valitud linnade arvuga suvaline kaart. Programmi kasutajaliideses on võimalik näha kaarti kahemõõtmelisel tasandil. Kui kasutaja paneb algoritmi tööle, siis on kaardil näha algoritmi hetkel uuritavat olekut (teekond on markeeritud hallide joontega). Kasutajaliideses on samuti näha parimat tulemust (teekond on markeeritud punaste joontega). Kasutajaliidese päises on näha algoritmi tööaeg (millisekundites), staatus, parima teekonna teepikkus ja hetk (millisekundites), millal on parim teekond leitud.

Lisaks on kasutajal võimalik ka aeglustada algoritmi tööd, määrates millisekundites pause, mis leiavad aset pärast oleku läbivaatust (iteratsiooni). Pausi pikkus ei mõjuta kasutajaliidese päises kirjeldatud aegasid. See teeb kasutajal mugavamaks vaatamise, kuidas algoritm töötab.

Programm on kirjutatud java keeles, seega kompileeritud programm on kasutatav igal java virtuaalmasinal. Kasutajaliides ja otsingualgoritmi töö on paigutatud eraldi lõimedele, selleks, et tagada paremat kasutajamugavust.



Joonis 15. Rakenduse kasutajaliides.

5.1. Kasutusjuhend

Kasutajaliidesese päises on kaks paani: „simulator” ja „help”. Simulator on paan kus võimalik käivitada erinevaid algoritme ja vaadelda nende tööd. Paanis „help” on kirjas kasutusjuhend ja mõned soovitused.

Rakenduse silmapaistvaimaks osaks on kaart kasutajaliidese keskel. Sellel võime näha linnade paigutust ja teekonda.
Hallide joontega kuvatakse teekonda, mida vaadeldi viimati. Punaste joontega kuvatakse hetkeks leitud parim teekond.

Kaardi peal on ülevaade sel hetkel tehtud tööst. Nimelt kuvatakse algoritmi staatus, millel on kaks olekut „idle” (ootel) ja „working” (töötab). Lisaks on kuvatud ka töö algusest möödunud aeg (Time). Selle abil on võimalik mõõta kogu algoritmi tööks kulunud aega.
Samuti on kuvatud ka parima leitud teekonna pikkus ja ajahetk, millal on see leitud (programm võib jätkata tööd ka pärast parima tulemuse leidmist).

Kaardi all on programmi juhtpaneel. Väljale „Cities” sisestatakse linnade arv, mida järgmine genereeritud kaart sisaldab. Samuti on võimalik lisada ka iteratsioonidevaheline paus, mis võimaldab kasutajal aeglustada algoritmi tööd. Seda on hea kasutada, kui kasutajal on soov vaadelda, kuidas algoritm suunab oma tööd.

Nupp „Start” alustab algoritmi tööd ja nupp „Stop” seiskab selle. Kui kasutaja tahab jätkata algoritmi tööd pärast „Stop” nupu vajutamist, siis võib lihtsalt vajutada nuppu start.
Nupp „Generate” genereerib uue kaardi niipaljude linnadega, kui palju on sätestatud väljal „Cities”. Selle nupu vajutusega ajutiselt seisatud algoritm kustutab oma hetke andmestruktuurid ja alustab tööd algusest.
Juhtpaneeli paremas nurgas on eraldi menüü, kus saab valida mitme algoritmi vahel. Kui kasutaja seiskab algoritmi töö ja valib teise algoritmi, siis algoritm ei jätka eelmise kohalt, vaid alustab tööd algusest. Menüüst saab valida järgmiste algoritmide vahelt:

1. Süvitsiotsing (Depth first search)
2. Laiutiotsing (Breadth first search)
3. Lähima naabri algoritm (Nearest neighbour search)
4. Parim enne-otsing (Best-first search)
5. Geneetiline algoritm (Genetic algorithm)
6. Libalõõmutamise algoritm (Simulated annealing)
7. Hopfieldi närvivõrk (Hopfield neural net)

6. Ettepanekuid edasiseks uurimiseks

6.1. Rändkaupmehe probleemi tingimuste muutmine

Antud töös me eeldasime, et igast linnast saab liikuda igasse teisse linna. Oleks huvitav uurida, kuidas algoritmid käituksid, kui igast linnast ei saaks liikuda igasse teise. Soovitaksin uurida, kuidas käituksid algoritmid sellise lisatingimusega.

6.2. Mitme otsingualgoritmi rakendamine

Erinevatel otsingualgoritmidel on oma eelised ja puudused. Huvitav oleks teada, kas mingite otsingualgoritmide järjestikune kasutamine võiks tuua kiiremini paremat tulemust. Näiteks kasutada lähima naabri algoritmi kiirust ära selleks, et genereerida enam-vähem optimaalse tulemuse, mida saaks kasutada kas geneetiline või libalõõmutamise algoritm algolekuna sisendiks edasiseks otsinguks.

6.3. Probleemi jagamine alamprobleemideks

Rändkaupmehe probleemi on võimalik jagada ka alamprobleemideks. Iga alamprobleemi saab lahendada eraldi ja paralleelselt. Kuna iga alamprobleemi olekuruum on väiksem, saame alamprobleemid lahendada kiiremini ja väiksema mälukulutusega. Kui kõik alamprobleemid on lahendatud, saame tulemused kokku panna algse probleemi lahenduseks.

Näiteks kui meil on 100 linna ja kasutatav algoritm on ruutkeerukusega, siis algoritm peab tegema 10000 tegumit. Kui aga jagada 100 linna neljaks 25 linnaliseks portsjoniks, siis iga portsjoni lahendamiseks on vaja teha 625 tegumit ja kokku on vaja teha 2500 tegumit, millele lisandub x tegumit, mida kasutatakse tulemuste kokku kleepimiseks.

6.4. Geneetilise otsingualgoritmi ristamisreegli parandamine

Üheks probleemiks selles töös implementeeritud geneetilise algoritmiga on see, et ristamisreegel on suhteliselt primitiivne. Oletan, et selle tagajärjel ei saa algoritm ka häid tulemusi suurema linnade arvu korral. Soovitaksin uurida, kas on võimalik ristamisreeglit täiendada või teha ümber nii, et igast geenist võetaks ainult head jupid ja need kombineeritaks kokku. Seejuures ei tohi unustada, et liiga keerukad algoritmid lisavad ajakulu juurde.

7. Kokkuvõte

Rändkaupmehe probleem on faktoriaalse keerukusega ja on väga paljude rakenduslike probleemide abstraktsioon. Kasutades seda alusprobleemina, võime võrrelda erinevaid algoritme.

Antud töös uurisime pimeotsingu algoritmidest süvitsi- ja laiutiotsingut. Heuristilistest otsingualgoritmidest uurisime lähima naabri otsingualgoritme, parim enne-otsingut, geneetilist algoritmi ja libalõõmutamise algoritmi. Lisaks tegime põgusa sissejuhatuse tehisnärvivõrkudesse ja uurisime Hopfieldi tehisnärvivõrku.

Iga mainitud algoritmi kohandasime rändkaupmehe probleemi lahendamiseks ja seejärel katsetasime ka probleemi lahendamist erinevate linnade arvuga. Analüüsisime algoritmide töötulemusi, ressursinõudlikkust ja võrdlesime neid omavahel.

Analüüsist võime järeldada, et pimeotsingu algoritmid ei ole praktilised rändkaupmehe probleemi lahendamisel. Libalõõmutamise algoritm näitas kõige lootustandvamaid tulemusi. Hopfieldi närvivõrk on näidanud ennast samuti üpriski ebasoodsa valikuna rändkaupmehe probleemi lahendamisel. Tegelikult oleme sellega ühtlasi näidanud, et tehisnärvivõrgud ei ole väga head algoritmiliselt lahendatavate probleemide lahendamisel.

Antud töö käigus on valminud ka õpiprogramm, mis aitab visualiseerida algoritmide tööd, hõlbustab ajakulu mõõtmist ja aitab paremini näha erinevate algoritmide puudujääke ja eeliseid. Lisaks aitab visualiseering vaadata, kuidas algoritm läheb üle ühelt olekult teisele

Different Methods and Their Comparison in Solving Traveling Salesman Problem

Erich Jagomägis

Summary

Traveling salesman problem is a NP-hard problem which is commonly used as an abstraction of many real life problems. It is widely used as benchmark to many optimization methods.

In given thesis we analyze various methods, describe their algorithms and benchmark their performance when solving traveling salesman problem. Furthermore we compare results.

Firstly an introduction is made to artificial neural networks. Then Hopfield neural network is introduced and applied to solving traveling salesman problem.

Secondly blind search algorithms are introduced. In given thesis we focus on depth first search and breadth first search. In both cases we introduce some optimizations to algorithms which enable them to work faster.

Thirdly heuristic search algorithms are introduced and some more common algorithms are analyzed. In given thesis nearest neighbour algorithm, best first algoritm, genetic algorithm and simulated anneal are described.

Once the algorithms and theory behind them are described, they are implemented in a java application, which is used to benchmark their performance. Application is made firsthand for students to observe how algorithms work, as well as measure their time consumption and measure the results of each algoritm.

Having done the benchmarks in this thesis, it is concluded that blind search algoritms are not suitable methods for solving traveling salesman problem. Furthermore it is concluded that Hopfield neural networks are also impractical when it comes to solving traveling salesman problem. Nearest neighbour search, genetic algorithm and simulated annealing are found to be most promising methods when solving problem described above. Most suitable method is found to be simulated annealing proving to find good solutions in relatively short time.

In the end of thesis some proposals are made, which could be further investigated by students. These proposals could benefit work done so far by improving described methods.

Mathematical functions to generate distance matrix from coordinates and vice versa are described in extras.

Kirjandus

[1] *Benefits of neural networks* [WWW] *http://www.ehow.com/list_6193138_benefits-neural-networks_.html* (05.01.2012)

[2] *Best-first search* [WWW] http://www.cs.washington.edu/education/courses/cse326/03su/homework/hw3/bestfirstsearch.html (20.04.2012)

[3] Gandhi, R. (2001). *Implementation of traveling salesman's problem using neural network* [e- artikkel] http://www.ece.uic.edu/~rgandhi/resume/Hope_tsp.PDF (05.05.2012)

[4] Hopfield Network [WWW] http://en.wikipedia.org/wiki/Hopfield_network (06.01.2012)

[5] Eduard Petlenkov (2004) Tehisnärvivõrgud ja nende rakendused [WWW] http://www.dcc.ttu.ee/las/ISS0010/Tehisnarvivorgud-Eduard2004.pdf (13.08.2013)

[6] Koit, M *Probleemid ja probleemilahendus* [WWW] http://www.cs.ut.ee/~koit/Tehisintellekt/T5.html (03.02.2012)

[7] Koit, M., & Roosmaa, T. (2011) *Tehisintellekt.* TÜ Kirjastus.

[8] Orr, G. & Schraudolph, N, & Cummins, F (1999) *Computation in the brain* http://www.willamette.edu/~gorr/classes/cs449/brain.html (05.05.2012)

[9] Obitko, M (1998) *Genetic algorithms* [WWW] http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php (25.03.2012)

[10] Obitko, M (1998a) *Genetic algorithms* [WWW] http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php (25.03.2012)

[11] Obitko, M (1998b) *Genetic algorithms* [WWW] http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php (25.03.2012)

[12] Obitko, M (1998c) *Genetic algorithms* [WWW] http://www.obitko.com/tutorials/genetic-algorithms/selection.php (25.03.2012)

[13] Potvin, J.-Y. *The Traveling Salesman Problem: A Neural Network Perspective*

[14] *Simulated annealing* [WWW] http://en.wikipedia.org/wiki/Simulated_annealing (25.03.2012)

[15] Stergiou, C. & Dimitrios, S*. Neural networks* [WWW] http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html (20.04.2012)

[16] *Tehisnärvivõrk* [WWW] http://www.staff.ttu.ee/~jmajak/Tehisnarvivorgud.doc (20.04.2012)

[17] *Traveling salesman problem* (10.04.2013) [WWW] http://en.wikipedia.org/wiki/Travelling_salesman_problem(15.04.2013)

Lisad

1. Punktide koordinaatidest kaugusmaatriksi loomine

Olgu meil olemas punktide koordinaadid kahemõõtmelises ruumis. Kaugusmaatriksis on igal punktil olemas oma rida ja oma veerg. Seega N punkti jaoks on kaugusmaatriksiks N * N e. ruutmaatriks. Linna N_1 ja N_2 vaheliseks kauguseks vastab maatriksis element $M_{n1,n2} = \sqrt{(N_{1x} - N_{2x})^2 + (N_{1y} - N_{2y})^2}$

2. Kaugusmaatriksist koordinaatide loomine

Olgu meil olemas n-astmeline ruutmaatriks M, kus element $M_{i,j}$ on kaugus punkti i ja j vahel. Meie peame kasutades kaugusmaatriksist arvutama koordinaadid. Seda saame teha järgmise algoritmi abil.

1. Määrame esimeseks punktiks P_1 = (0,0).
2. Määrame teiseks punktiks P_2 (0, $M_{2,1}$).
   Ülejäänud punktid P_n, kus n ∈ {3…n} saame arvutada järgmiste valemite abil.

$P_n = \left(M_{1,n} * \cos(a), M_{1,n} * \sin(a)\right)$, kus

$$\cos(a) = \frac{M_{1,2}^2 + M_{1,n}^2 - M_{2,n}^2}{2 * M_{1,2} * M_{1,n}}$$

3. Arvutame P_3 kasutades defineeritud valemeid punktis 2.
4. Arvutades järgnevaid punkte, peame kontrollima kas kehtib võrdus
   $M_{3,n} = \sqrt{(P_{3x} - P_{nx})^2 + (P_{3x} - P_{nx})^2}$, ning kui ei kehti, siis
   $P_n = \left(M_{1,n} * \cos(a), -(M_{1,n} * \sin(a))\right)$.

Mina Erich Jagomägis
(*autori nimi*)

(sünnikuupäev: 12.12.1989)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

Erinevad meetodid ja nende võrdlus rändkaupmehe probleemi lahendamisel,
(*lõputöö pealkiri*)

mille juhendaja on Mare Koit,
(*juhendaja nimi*)

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **15.08.2013**