T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Teoreetilise informaatika õppetool

informaatika eriala

Jaanus Jaeger

Ajaakendega transpordiülesande lahendamine

Magistritöö

Juhendaja: dotsent Jan Villemson

Autor: . . . . “. . .” mai 2005

Juhendaja: . . . . “. . .” mai 2005

Õppetooli juhataja: . . . . “. . .” 2005

Tartu 2005

Sisukord

1 Sissejuhatus **4**

2 Transpordiülesanne **6**

2.1 Transpordiülesannete liigid 6
2.1.1 Rändkaupmeheülesanne 6
2.1.2 Transpordiülesanne 6
2.1.3 Ajaakendega transpordiülesanne 8
2.2 Ülesandepüstitus 8
2.2.1 Algandmed 8
2.2.2 Lahend 10
2.2.3 Lahendi maksumus 11
2.3 Üldine lahendusskeem 11
2.4 Abimõisted 12

3 Klientide klasterdamine **15**

3.1 Klasterdamine kui eeltöötlus 15
3.2 Probleemid klientide klasterdamisel 16
3.3 Klastri klientide teenindatavus 17
3.4 Klastri atribuudid 18
3.5 Klastri teenindatavus klasterdamise tulemuseks olevas ülesandes . 21
3.6 Klastritevahelised kaugused 22
3.7 Hierarhilise klasterdamise efektiivne rakendamine transpordiülesandes 23
3.8 Klastrite avamine 24
3.8.1 Klastri avamine vastavalt klastriülesande lahendile 24
3.8.2 Klastri avamise optimeerimine 25

4 Lahendusmeetod sipelgakoloonia süsteem **29**

4.1 Tehissipelgad ning nende keskkond 29
4.2 Sipelgakoloonia süsteem 31
4.3 Mitme sipelgakoloonia süsteem ajaakendega transpordiülesande lahendamiseks 33
4.3.1 Juhtprotseduur 34
4.3.2 Sõitude arvu minimeeriv koloonia ACS-vehicle 34
4.3.3 Maksumust minimeeriv koloonia ACS-length 35

4.3.4 Üksik sipelgas – lahendeid konstrueeriv protseduur 35

5 Lokaalne otsing **38**

5.1 Heuristikud 38

5.2 Lokaalse otsingu kiirendamine 39

5.3 Rist-vahetus heuristiku kiirendamine 41

6 Praktiline realisatsioon **43**

6.1 Transpordiülesande lähteandmete ettevalmistamine 43

6.2 Võrdlus olemasoleva tarkvaraga 44

7 Kokkuvõte **46**

Solving Vehicle Routing Problem with Time Windows **47**

Indeks **49**

Viited **51**

A Klasterdamise algoritmid **52**

B ACS ja MACS-VRPTW algoritmid **55**

1 Sissejuhatus

Käesolevas töös käsitleme levinud optimeerimisülesande tüüpi – transpordiülesannet. Transpordiülesanne on edasiarendus klassikalisest rändkaupmeheülesandest, mille puhul tuleb leida etteantud graafi tippude jaoks optimaalne läbimisjärjekord, imiteerimaks ühe rändkaupmehe teekonda läbi erinevate linnade. Transpordiülesande korral vaadeldakse ühe rändkaupmehe asemel hulka sõidukeid, mida kasutades on tarvis teenindada hulk kliente. Teenindamine võib tähendada näiteks kauba toimetamist kliendini, prügivedu vms. Sellest tulenevalt pakub vaadeldav ülesanne huvi näiteks kaupa laiali vedavatele hulgifirmadele, kullerteenuse pakkujatele jt.

Seoses transpordiülesande kohandamisega vastavalt reaalse maailma nõudmistele, on kujunenud välja mitmesuguseid ülesande eritüüpe. Üks levinud ülesandetüüp on ajaakendega transpordiülesanne, millele antud töös keskendumegi. Ajaakendega transpordiülesande puhul on iga kliendi jaoks määratud ajavahemik, mille jooksul teda saab teenindada.

Reaalses maailmas on kliendid enamasti jaotatud geograafiliselt ebaühtlaselt. Suurem hulk kliente on koondatud linnadesse, seevastu maapiirkondades leidub neid suhteliselt hõredalt. Läbitavaid marsruute arvestades on üsna mõistlik, et ühte piirkonda kuuluvad kliendid teenindatakse ära ühekorraga ega toimu liikumist erinevate linnade vahel edasi-tagasi. Seda asjaolu tahame ära kasutada ka käesolevas töös. Nimelt, kui lühemate marsruutide korral teenindatakse lähestikku asetsevad kliendid ära järjest, võib kõiki vastava piirkonna kliente esindada üheainsa teenindamist vajava metakliendi ehk klastrina, kaotamata seejuures palju lahendi headuses. Peale klastreid sisaldava ülesande lahendamist tuleb veel vaid välja pakkuda konkreetsed läbimisjärjekorrad klastri klientide jaoks. Sellise lähenemise korral on korraga lahendatav ülesanne algandmete hulga mõttes tunduvalt väiksem ja seega lihtsam lahendada.

Transpordiülesande lahendamise pool on laiema vaatluse all olnud väga kaua, seda on põhjalikult uuritud ning välja on pakutud väga mitmeid edukaid lahendusmeetodeid. Seevastu ülesande lähteandmete analüüs ja selle lahendamiseeelne teisendamine on jäetud suurema tähelepanuta. Antud töö panus on seotud just lahendamise eeltööga – välja pakutakse meetodid ajaakendega transpordiülesande klientide klasterdamiseks.

Käesoleva töö peatükis 2 tutvustame lähemalt transpordiülesannet ning selle tüüpe. Samuti defineerime enamiku töös kasutatavatest mõistetest ja suurus-

test. Peatükis 3 pakume välja klientide klasterdamise kui meetodi ajaakendega transpordiülesande algandmete hulga vähendamiseks ning tutvustame tähtsamaid klasterdamisega seonduvaid probleeme. Järgnevalt vaatleme peatükis 4 sipelgakolooniate modelleerimisel põhinevaid tuntud lahendusmeetodeid nii rändkaupmeheülesandele kui transpordiülesandele. Peatükis 5 tutvustame üht levinud olemasolevate lahendite optimeerimismeetodit, lokaalset otsingut. Samuti pakume välja mõned võtted optimeerimise kiirendamiseks. Lõpuks, peatükis 6 räägime mõningate töös vaadeldud algoritmide realisatisoonist ühe logistikatarkvara prototüübi raames. Täpsemalt kirjeldame lisanduvaid samme, mis tuli astuda nende algoritmide rakendamiseks antud olukorras ning esitame programmi töö tulemused võrrelduna analoogse tarkvaraga.

2 Transpordiülesanne

Käesolevas peatükis anname kõigepealt punktis 2.1 lugejale üldise ettekujutuse transpordiülesandest ning selle erinevatest tüüpidest. Punktis 2.2 esitame formaalse ülesandepüstituse ja toome sisse vajalikud tähistused. Punktis 2.3 visandame üldise lahendusskeemi, mille kohaselt käesolevas töös ülesannet lahendatakse. Lõpuks, punktis 2.4, defineerime veel mõningad tähtsad töös kasutatavad mõisted ja suurused.

2.1 Transpordiülesannete liigid

Käesolevas töös vaadeldav ülesanne, transpordiülesanne, on tegelikult klassikalise rändkaupmeheülesande üldistus. Antud punktis anname mõlemast ülevaate ning tutvustame veel transpordiülesande levinud tüüpi: ajaakendega transpordiülesannet, mis on antud töö põhiline uurimisobjekt.

2.1.1 Rändkaupmeheülesanne

Rändkaupmeheülesande võib lühidalt püstitada järgnevalt. On antud hulk linnu ning linnadevahelised kaugused, eesmärk on leida lühim teekond läbi kõikide linnade. Sageli vaadeldakse ülesannet ka n-tipulise graafina, mille tipud kujutavad linnu ning linnadevahelised kaugused on servade pikkused.

Kuna n linna korral on erinevaid linnade läbimisjärjekordi kokku $n!$, siis on ülesande täpne lahendamine ehk lühima sellise teekonna leidmine vähegi suurema linnade arvu korral reaalselt võimatu. Otsing võtab variantide rohkuse tõttu liiga kaua aega ja seda sõltumata arvutuskiirusest. Seetõttu vaadeldakse rändkaupmeheülesannet enamasti optimeerimisülesandena, mille eesmärk pole leida tingimata lühim, vaid võimalikult lühike kõiki linnu läbiv teekond.

2.1.2 Transpordiülesanne

Transpordiülesanne on reaalsele elule mõnevõrra lähedasem. Üks rändkaupmees on asendatud hulga sõidukitega ning linnade asemel vaadeldakse ühte depood ja hulka kliente. Klientidel on määratud tellimus ehk tellitud kauba kogus ning sõidukitele on omistatud maksimaalne mahtuvus. Eesmärk on mingit arvu sõidukeid kasutades teenindada ära kõik kliendid selliselt, et teenindamise maksumus on minimaalne. Teiste sõnadega tuleb leida selline hulk teekondi, mis

algavad ja lõppevad depoos, läbivad vahepeal mingi hulga kliente ning mille maksumus on minimaalne. Sealjuures tuleb iga klient läbida täpselt üks kord ning ühegi teekonna poolt läbitavate klientide tellimuste summa ei tohi ületada sõiduki mahtuvust.

Kõige tüüpilisem reaalne rakendus transpordiülesandele on veoringide optimeerimine kauba laialiveol hulgifirma klientidele. Ülesande depoo on hulgifirma ladu, kliendid on näiteks kaupa tellinud poed.

Antud ülesande puhul omab mõtet rääkida erinevatest lahendi maksumustest. Lahendi maksumus ei pruugi enam olla teekonna kogupikkus vastavalt klientidevahelistele kaugustele. Arvesse võidakse võtta näiteks ka kasutatud sõidukite arvu vms. Klientide vaheline kaugus ei pruugi samuti olla klientide vahelise tee pikkus. Selleks võib olla näiteks sõiduaeg, bensiinikulu vms. Erinevaid maksumusi vaatame lähemalt punktis 2.2.3.

Rändkaupmeheülesannet saab lihtsasti esitada transpordiülesande erijuhuna, kui valida sõidukite arvuks 1 ning võtta kõigi klientide tellimused võrdseks 0-ga või kasutada piisavalt suurt sõiduki mahtuvust.

Sarnaselt rändkaupmeheülesandele kasvab ka transpordiülesande kõikvõimalike lahendite hulk klientide arvu kasvades väga kiiresti. Seetõttu on ülesande täpne lahendamine ehk kõikide variantide läbivaatamine suure algandmete mahu korral reaalselt võimatu. Käesolevas töös vaatleme transpordiülesannet kui optimeerimisülesannet, mis minimeerib kliente läbivate teekondade maksumust.

Transpordiülesanne jaguneb kahte alamklassi vastavalt sellele, kuidas on defineeritud sõidukite mahtuvus:

- *homogeense autopargiga transpordiülesanne* – kõigil sõidukitel on sama mahtuvus,
- *mittehomogeense autopargiga transpordiülesanne* – igal sõidukil on oma spetsiifiline mahtuvus.

Esimene neist on teise erijuht, võttes kõigi sõidukite mahtuvused võrdseks. Siiski on ka homogeense autopargiga ülesanne piisavalt laialt kasutatav. Sageli on hulgifirmades kasutusel suurem hulk ligikaudu sama mahtuvusega autosid, mida kasutatakse kauba laialiveoks. Käesolevas töös vaatleme homogeense autopargiga transpordiülesannet.

2.1.3 Ajaakendega transpordiülesanne

Ajaakendega transpordiülesanne on veelgi üldisem ning ühtlasi kõige lähedasem reaalse elu situatsioonile. Siin lisanduvad klientidele ajaaknad ehk ajavahemikud, mille jooksul kliente võib külastada, ja teenindusajad ehk kliendi teenindamiseks kuluvad ajad. Samuti on teada klientide vahel sõitmise ajad. Antud ülesandetüübi korral peavad lahendiks olevad teekonnad võimaldama teenindada kliente nende ajaakna jooksul.

Eespoolvaadeldud hulgifirma näite puhul on klientide ajaakendeks kauba vastuvõtuajad poodides.

Ajaakende puudumist võib väljendada näiteks väga suurte ajaakende või siis sõiduaegadega 0, saades nii hariliku transpordiülesande.

2.2 Ülesandepüstitus

Punktis 2.1 nägime, et transpordiülesande puhul on tegmist ühe spetsiifilise optimeerimisülesandega. Seega on transpordiülesande väga üldine püstitus järgmine: *leida ülesande algandmetele vastavaid lahendeid vähendades lahendi maksumust.*

Käesolev punkt on organiseeritud vastavalt toodud üldisele püstitusele. Punktis 2.2.1 fikseerime ülesande algandmed koos töös kasutatavate tähistustega, punktis 2.2.2 defineerime formaalselt ülesande lahendi ning punktis 2.2.3 defineerime lahendi maksumuse ja vaatleme erinevaid võimalikke maksumusi.

2.2.1 Algandmed

Käesolevas töös käsitleme punktis 2.1 vaadeldud ülesande tüüpidest homogeense autopargiga ajaakendega transpordiülesannet.

Olgu antud orienteeritud täisgraaf $G = (X, E)$. Transpordiülesande P algandmeteks on järgmised suurused (olgu x ja y suvalised tipud hulgast X):

- $d \in X$ – ülesande P *depoo*,
- $\{c_1, \ldots, c_n\} = X \setminus \{d\}$ – ülesande P *klientide hulk*,
- $d(x, y)$ – tippude x ja y vaheline *kaugus* ehk tippude x ja y vahelise teekonna maksumus,
- $t(x, y) \geq 0$ – tippude x ja y vaheline *sõiduaeg* ehk aeg, mis kulub liikumiseks tipust x tippu y mööda marsruuti, mis on määratud kaugusega $d(x, y)$,

- $d(x) \geq 0$ – tipu x *tellimus* ehk tellitava kauba kogus,
- $[b(x), e(x)]$ – tipu x *ajaaken*, kus $b(x)$ ehk *ajaakna algus* on varaseim aeg, mil saab alustada tipu x teenindamist ning $e(x)$ ehk *ajaakna lõpp* on hiliseim aeg, mil saab lõpetada tipu x teenindamise. Eeldame, et $b(x) \leq e(x)$,
- $s(x) \geq 0$ – tipu x *teenindusaeg* ehk tipu x teenindamiseks kuluv aeg. Eeldame, et $s(x) \leq e(x) - b(x)$,
- $v(P) \geq 0$ – ülesande P *sõidukite mahtuvus*,
- $[b(P), e(P)]$ – ülesande P *ajaaken*. Eeldame, et $b(P) \leq e(P)$.

Nõue, et G oleks täisgraaf ei tähenda, et iga klientide paari vahel peab eksisteerima otsetee. Mõeldud on, et iga kliendi või depoo juurest on võimalik mingeid teid pidi jõuda igasse teise klienti ja depoosse. Sisuliselt tähendab see eeldust, et kõik kliendid ja depoo peavad asuma aluseks oleva teedegraafi samas sidususkomponendis. See pole aga reaalselt kuigi kitsendav eeldus, arvestades sellega, et ka näiteks sõiduki ülevedu mandrilt saarele võib olla teedegraafis esindatud ühe servana. Konkreetne tippudevaheline marsruut meid transpordiülesande lahendamisel ei huvita. Tippude x ja y korral kasutame ainult nendevahelist kaugust $d(x, y)$ ehk antud ülesande mõttes optimaalse nendevahelise tee maksumust ning sõiduaega $t(x, y)$.

Tippudevahelise kauguse kohta kehtib nõue, et see peab rahuldama kolmnurgavõrratust. Sümmeetrilisust me aga ei eelda, kuna tegemist on orienteeritud graafiga, seega kaugus $d(x, y)$ ei pruugi olla võrdne kaugusega $d(y, x)$. Viimane kehtib ka sõiduaegade kohta. Paneme tähele, et tippudevaheline kaugus ei pea olema tingimata nendevahelise tee pikkus. See on üldisemalt öeldes tippudevahelise teekonna maksumus ning defineeritud konkreetse transpordiülesande poolt.

Vaatame siinkohal lähemalt, mida meie jaoks tähendab see, et tippude x ja y vaheline sõiduaeg $t(x, y)$ on aeg liikumaks tipust x tippu y mööda marsruuti, mis on määratud kaugusega $d(x, y)$. Toodud sõiduaja definitsioonist tulenevalt ei pruugi $t(x, y)$ olla lühim aeg jõudmaks tipust x tippu y, sest odavama tee läbimiseks ei pruugi alati kuluda lühem aeg (näiteks võtame kauguseks teepikkuse ning arvestame sõiduaegade leidmisel kiirusepiiranguid). Seega kui $x, y, z \in X$ ja $d(x, y) < d(x, z)$, siis sellest ei järeldu, et kehtib $t(x, y) < t(x, z)$. Samuti ei saa me tegelikult väita, et kehtiks $t(x, y) < t(x, z) + t(z, y)$, kuigi kehtib

$d(x,y) < d(x,z) + d(z,y)$. Teiste sõnadega saame, et ülesande graaf G ei pruugi rahuldada sõiduaegade suhtes kolmnurgavõrratust.

Lõpuks paneme veel tähele, et erinevalt traditsioonilisest transpordiülesande püstitusest on meil defineeritud depoo jaoks täpselt samad atribuudid, mis klientide jaoks – tellimus, ajaaken ja teenindusaeg. Sellist lähenemist on vaja peatükis 3. Depoo atribuudid võivad omada fiktiivseid väärtusi, mis ülesande lahendamist ei mõjuta (tellimus ja teeindusaeg 0, ajaaken väga suur). Teisalt on depoo ajaaknal teinekord ka sisuline tähendus, sest ka hulgifirma ladu võidakse avada ja sulgeda mingitel kindlatel aegadel.

2.2.2 Lahend

Enne lahendi juurde jõudmist defineerime *sõidu* kui ühe sõiduki poolt läbitava teekonna e järjendi, mis koosneb klientidest ja depoost.

Olgu meil transpordiülesanne P depooga d ja klientide hulgaga $X \setminus \{d\}$.

Definitsioon 1 *Transpordiülesande P* sõiduks *T nimetame järjendit $t_1, \ldots, t_m$, mille esimene ja viimane element on ülesande depoo ($t_1 = t_m = d$) ja mille ülejäänud elemendid on kliendid ($\forall i \in \{2, \ldots, m-1\} : t_i \in X \setminus \{d\}$) ning mille korral kehtivad järgmised tingimused:*

1. *sõit T ei ületa sõiduki kandevõimet:*
$$\sum_{i=1}^{m} d(t_i) \leq v(P), \tag{1}$$
2. *läbides tipud $t_1, \ldots, t_m$ esitatud järjekorras, võttes arvesse nendevahelisi sõiduaegasid $t(t_j, t_{j+1})$ ($j \in \{1, \ldots, m-1\}$), on kõik sõidu tipud võimalik teenindada nende ajaakna jooksul.*
3. *sõit T mahub ajaliselt ülesande P ajaakna sisse.*

Tähistame vaadeldud sõitu $T = (t_1, \ldots, t_m)$. Suurust m nimetame sõidu pikkuseks. *Tingimust 1 nimetame ka* mahupiiranguks *ning tingimusi 2 ja 3* ajakitsenduseks.

Tipu t teenindamise all peame silmas peatumist tipus t aja $s(t)$ vältel.

Sõidu $T = (t_1, \ldots, t_m)$ tipu t_i ajaakent, teenindusaega ja tellimust tähistame samamoodi nagu kliendil – vastavalt $[b(t_i), e(t_i)]$, $s(t_i)$ ja $d(t_i)$. Samuti tähistame tippude t_i ja t_j vahelist kaugust $d(t_i, t_j)$ ning sõiduaega $t(t_i, t_j)$.

Definitsioon 2 *Transpordiülesande* P lahendiks S *nimetame sõitude hulka* $\{T_1, \ldots, T_l\}$ *(*$T_i = (t_{i1}, \ldots, t_{im_i}), i \in \{1, \ldots, l\}$*), mille poolt teenindatakse ära kõik ülesande kliendid täpselt üks kord, st kehtib*

$$\forall x \in X \setminus d \, \exists! i \in \{1, \ldots, l\} \, \exists! j \in \{2, \ldots, m_i - 1\} : t_{ij} = x,$$

kus d *on ülesande* P *depoo. Tähistame lahendit* $S = \{T_1, \ldots, T_l\}$*.*

Nii sõit T kui lahend S on tegelikult erilised graafid. Tähistame nende tippude hulka vastavalt $V(T)$ ja $V(S)$ ning servade hulka vastavalt $E(T)$ ja $E(S)$.

2.2.3 Lahendi maksumus

Lahendi S maksumust tähistame $W(S) \in \mathbb{R}$.

Klassikaliselt on transpordiülesande lahendi maksumuseks lahendi poolt määratud tee pikkus, sageli ka sõitudele kuluv aeg. Lahendusmeetodite väljatöötamisel on aga soovitav vältida juhindumist väga spetsiifilisest maksumusest, sest muidu pole vastav meetod üldjuhul kasutatav.

Paneme tähele, et lahendi maksumus ei pea väljendama ühteainsat suurust. Maksumuse arvutamisel võivad olla arvesse võetud teepikkus, sõiduaeg, ootamise aeg enne kliendi vastuvõtu algust, kasutatavate sõidukite arv jne. Igale komponendile võib olla omistatud kaal, millega antud suurust arvestatakse. Näeme, et osad loetletud komponentidest (klientidevahelise tee pikkus ja sõidu aeg) on lisaks lahendile defineeritud ka üksiku serva kohta. Need võetakse arvesse tippudevahelise kauguse $d(x, y)$ juures.

Eeldame, et lahendi maksumus ja tippudevaheline kaugus on omavahel kooskõlas. Viimasega peame silma seda, et lahendi maksumus võtaks lahendisse kuuluvate servade juures arvesse samu serva omadusi, nagu seda teeb tippudevaheline kaugus. Vastasel juhul tekib olukord, kus lahendeid konstrueeritakse ühtede, aga võrreldakse hiljem omavahel hoopis teiste kriteeriumite alusel.

2.3 Üldine lahendusskeem

Käesolevas töös kasutame lahendusskeemi, mis koosneb järgmistest etappidest.

Lähteandmete eeltöötlus. Lähteandmete eeltöötluse all peame silmas transpordiülesande selliseid teisendusi, mis muudavad ülesande lahendamise mingis

mõttes lihtsamaks, kiiremaks või mille tulemusena annaks lahendusmeetod välja paremaid tulemusi. Eeltöötluse tulemuseks on mingi teine transpordiülesanne, mille lahendist peab olema võimalik suhteliselt lihtsalt kontsrueerida lahendit esialgsele transpordiülesandele. Antud töös käsitleme eeltöötlusena klientide lahendamise-eelset klasterdamist (vt ptk 3).

Lahendamine. Transpordiülesande lahendamiseks on olemas väga palju erinevaid algoritme. Käesolevas töös vaatleme lähemalt sipelgakolooniate modelleerimisel põhinevat algoritmi (vt ptk 4).

Lahendi järeltöötlus. Nagu mainitud, on lähteandmete eeltöötluse tulemuseks muudetud transpordiülesanne, seega on vaja metoodikaid selle ülesande lahendite teisendamiseks esialgse ülesande lahenditeks. Sellega tegeleme punktis 3.8. Lisaks eelnevale käsitleme ka mitmeid lahendite parandamise heuristikuid – lokaalse optimeerimise heuristikuid (vt ptk 5).

2.4 Abimõisted

Selles punktis defineerime mõningad sõiduga seotud mõisted.

Definitsioon 3 *Sõidu $T = (t_1, \ldots, t_m)$ tipu $t_i, i \in \{1, \ldots, m\}$* varaseimaks väljumisajaks *nimetame varaseimat lubatavat ajahetke, millal on võimalik antud tipust väljuda, läbides eelnevalt tipud $t_1, \ldots, t_{i-1}$ vastavalt ülesande kitsendustele. Varaseimat väljumisaega tähistame $ed\,(t_i)$.*

Definitsioon 4 *Sõidu $T = (t_1, \ldots, t_m)$ tipu $t_i, i \in \{1, \ldots, m\}$* hiliseimaks väljumisajaks *nimetame hiliseimat lubatavat ajahetke, millal on võimalik antud tipust väljuda, läbides järgnevalt tipud $t_{i+1}, \ldots, t_m$ vastavalt ülesande kitsendustele. Hiliseimat väljumisaega tähistame $ld\,(t_i)$.*

Paneme tähele, et mõlemad definitsioonid on rakendatavad ka depoo kohta, sh nii depoost lahkumise kui depoosse naasmise kohta.

Sõidu $T = (t_1, \ldots, t_m)$ tippude varaseimad väljumisajad avalduvad rekurrentse valemiga

$$\begin{aligned} ed\,(t_1) &= \max\{b\,(P)\,, b\,(t_1)\} + s\,(t_1), \\ ed\,(t_i) &= \max\{ed\,(t_{i-1}) + t\,(t_{i-1}, t_i)\,, b\,(t_i)\} + s\,(t_i) \end{aligned} \tag{2}$$

ning hiliseimad väljumisajad valemiga

$$\begin{aligned} ld\,(t_m) &= \min\{e\,(P)\,, e\,(t_m)\}\,, \\ ld\,(t_i) &= \min\{ld\,(t_{i+1}) - s\,(t_{i+1}) - t\,(t_i, t_{i+1})\,, e\,(t_i)\}\,. \end{aligned} \tag{3}$$

Kasutades äsjadefineeritud mõisteid, saame esitada formaalse kuju definitsioonis 1 toodud tingimustele 2 ja 3. Olgu meil transpordiülesanne P ning olgu $T = (t_1, \ldots, t_m)$ tema mingi sõit. Näeme, et

- tingimus 2 kehtib parajasti siis, kui

$$\forall i \in \{1, \ldots, m\} : ed\,(t_i) \leq e\,(t_i)\,, \tag{4}$$

- tingimus 3 kehtib parajasti siis, kui

$$ed\,(t_m) \leq e\,(P)\,. \tag{5}$$

Kummagi tingimuse täidetuseks piisab ainult ajaakna lõpu kontrollimisest seetõttu, et varaseim väljumisaeg arvestab nii ülesande kui ka klientide ajaakende algusega (vt võrrandeid (2)).

Varaseimate väljumisaegade valemite kasutust saab lihtsasti laiendada. Fikseerides mingi väärtuse $\overline{ed\,(t_j)} = t$, saame leida varaseimad väljumisajad sõidu tippude $t_{j+1}, \ldots, t_m$ jaoks, kui tipust t_j väljutakse ajahetkel t. Siinkohal on ilmselt mõistlik nõue, et fikseeritud aeg t võimaldaks sõidu korrektset läbimist, st üheski järgnevatest klientidest ei ületata lubatud ajaakent. Viimase nõude saab formaalselt kirja panna kujul $ed\,(t_j) \leq t \leq ld\,(t_j)$. Analoogiline arutelu kehtib ka hiliseima väljumisaja kohta.

Peatüki lõpuks tõestame järgmise lemma.

Lemma 1 *Olgu antud sõit $T = (t_1, \ldots, t_m)$. Kui tipust t_j ($j \in \{1, \ldots, m\}$) väljumise aega muuta varaseimast väljumisajast $ed\,(t_j)$ aja Δt ($0 \leq \Delta t \leq ld\,(t_j) - ed\,(t_j)$) võrra hilisemaks, muutub tipu t_i ($i \in \{j, \ldots, m\}$) varaseim väljumisaeg ülimalt aja Δt võrra hilisemaks.*

Tõestus

Kasutame käesolevas tõestuses tähistust $\overline{ed\,(t_i)}$ ($i \in \{j, \ldots, m\}$) tähistamaks varaseimat väljumisaega tipust t_i, kui tipust t_j väljutakse ajahetkel $ed\,(t_j) + \Delta t$ ($0 \leq \Delta t \leq ld\,(t_j) - ed\,(t_j)$).Väite tõestuseks peame nüüd näitama, et $\forall i \in \{j, \ldots, m\} : \overline{ed\,(t_i)} \leq ed\,(t_i) + \Delta t$. Kasutame selleks induktsiooni indeksi i järgi.

Induktsiooni baas: $i = j$. Baasi korral on väide ilmne, sest $\overline{ed\,(t_j)} = ed\,(t_j) + \Delta t$.

Induktsiooni samm: $i = l + 1$ ($l \in \{j, \ldots, m-1\}$). Oletame, et väide kehtib $i = l$ korral. Näiteme, et sel juhul kehtib väide ka $i = l + 1$ korral. Avaldame suuruse $\overline{ed\left(t_{l+1}\right)}$ vastavalt varaseima väljumisaja võrrandile (2).

$$\begin{aligned}
\overline{ed\left(t_{l+1}\right)} &= \max\left\{\overline{ed\left(t_l\right)} + t\left(t_l, t_{l+1}\right), b\left(t_{l+1}\right)\right\} + s\left(t_{l+1}\right) \leq \\
&\leq \max\left\{ed\left(t_l\right) + \Delta t + t\left(t_l, t_{l+1}\right), b\left(t_{l+1}\right)\right\} + s\left(t_{l+1}\right) \leq \\
&\leq \max\left\{ed\left(t_l\right) + t\left(t_l, t_{l+1}\right), b\left(t_{l+1}\right)\right\} + \Delta t + s\left(t_{l+1}\right) = \\
&= ed\left(t_{l+1}\right) + \Delta t.
\end{aligned}$$

Seega väide kehtib $i = l + 1$ korral.

MOTT

Vaatleme veel juhtu, kui $\Delta t = ld\left(t_j\right) - ed\left(t_j\right)$. Sel juhul on tipust t_j väljumisajaks ajahetk $ed\left(t_j\right) + ld\left(t_j\right) - ed\left(t_j\right) = ld\left(t_j\right)$, mis on hiliseima väljumisaja definitsiooni põhjal hiliseim aeg, millal saab tipust t_j väljuda nii, et järgnevad tipud saab korrektselt ära teenindada. Seega suuremad Δt väärtused pole lubatud.

3 Klientide klasterdamine

Käesolev peatükk on organiseeritud järgnevalt: punktis 3.1 kirjeldame üldisi klasterdamise kui eeltöötluse eesmärke, punktis 3.2 loetleme üles olulised probleemid klasterdamise juures ning punktides 3.3 – 3.8 otsime nendele lahendusi.

Lisas A on toodud transpordiülesande eeltöötluseks sobivad klasterdamise algoritmid, milles on kasutatud tulemusi käesolevast peatükist.

3.1 Klasterdamine kui eeltöötlus

Graafi tippude klasterdamine on toiming, mille käigus ühendatakse teineteisele mingis mõttes lähedased tipud ühtseteks tippudeks e. *klastriteks*. Lähedus ei pea tingimata olema tippudevaheline kaugus, vaid võib olla ka midagi muud sõltuvalt konkreetse ülesande vajadustest. Klasterdamise tulemuseks on muudetud graaf, mille tippude arv on harilikult väiksem kui esialgsel graafil, kuid võib olla ka võrdne. Seda juhul, kui leiti, et antud graafi klasterdamine pole mõistlik, näiteks kui tipud asetsevad tasandil laiali võrdlemisi ühtlase jaotusega. Saadud klastrid peavad rahuldama täpselt samu tingimusi, mis kehtivad klasterdamata tippudele.

Käesoleva peatüki idee on rakendada klasterdamist transpordiülesande klientidele. Klasterdamise kui lähteandmete eeltöötluse heuristika seisneb selles, et piisavalt lähestikku asetsevaid kliente püüame teenindada järjest. Seega võib nad liita lahendusmeetodi jaoks ühtseks tervikuks, mis teenindatakse korraga. Antud lähenemisega saame muuta korraga lahendatava ülesande algandmete hulka väiksemaks, võimaldades nii andmemahu suhtes kriitilisemate lahendusmeetodite rakendamist. Samuti, eeldades mõistlikku klastrite valikut, praagime kõikvõimalike lahendite hulgast välja sellised, mille puhul sõidetakse erinevatesse piirkondadesse kuuluvate klientide vahel mitu korda edasi-tagasi.

Peale klasterdamise tulemuseks oleva ülesande lahendamist tuleb saadud lahend teisendada esialgse ülesande lahendiks. See kujutab endast sisuliselt lahendis sisalduvate klastrite klientidele konkreetse läbimisjärjekorra määramist antud lahendi koosseisus.

Antud töös vaadeldava klasterdamise eesmärk pole sama, mis *cluster first route second* tüüpi lahendusmeetodites (vt. [1]). Viimaste põhimõte seisneb selles, et esmalt jaotatakse kliendid k klastrisse ning seejärel lahendatakse iga klastri jaoks rändkaupmehe ülesanne, saades lahendiks k veoringi.

Selleks, et lahendusmeetod saaks klastreid vaadelda harilike klientidena, peavad klastrid omama samu karakteristikuid kui kliendid. Neil peavad olema defineeritud tellimus, teenindusaeg, ajaaken, samuti maksumused $d(c_i, c_j)$ ja sõiduajad $t(c_i, c_j)$ liikumaks antud klastrist ükskõik millisesse teise graafi tippu – kas üksikusse klienti, depoosse või teise klastrisse. Seda asjaolu silmas pidades toome sisse termini *metaklient*, mis tähistab nii klientide klastreid kui ka klasterdamata kliente.

Kuna klasterdamine transpordiülesande lahendamise käigus pole eesmärk omaette, vaid abivahend ülesande lihtsamaks lahendamiseks, pole tulemuseks olev mõistlik klastrite arv ette teada. Seetõttu sobib antud juhul klasterdamismeetodiks näiteks *hierarhiline klasterdamine*, mille korral tippude arv tulemusgraafis võib olla suvaline arv hulgast $\{1, \ldots, n\}$, kus n on tippude arv esialgses graafis.

Kasutame klastrite jaoks samu tähistusi, mis hulkadel, st kui klaster C koosneb klientidest $c_1, \ldots, c_k$, siis seda asjaolu tähistame $C = \{c_1, \ldots, c_k\}$.

Hierarhiline klasterdamine

Hierarhilise klasterdamise võib kokku võtta algoritmiga 1.

Algoritm 1 Hierarhiline klasterdamine

1. Teha graafi iga tipu jaoks üheelemendiline klaster.
2. Leida kaks teineteisele lähimat klastrit.
3. Otsustada leitud klastrite paari põhjal, kas jätkata klasterdamist. Kui jah, jätkata sammust 4, vastasel korral väljuda tsüklist.
4. Ühendada sammul 2 leitud klastrid. Jätkata sammust 2.

3.2 Probleemid klientide klasterdamisel

Käesolevas peatükis loetleme olulisemad probleemid, mis ilmnevad transpordiülesande klientide klasterdamisel.

1. Loodava klastri klientide teenindatavus ühe sõiduga. Selle probleemiga tegeleme punktis 3.3.

2. Kuidas määrata klastri kui metakliendi atribuudid: ajaaken, teenindusaeg, tellimus? Vt punkt 3.4.

3. Loodava klastri kui terviku teenindatavus klasterdamise tulemuseks olevas ülesandes. Vt punkt 3.5.

4. Klastritevaheliste kauguste arvutamine. Vt. punkt 3.6.

5. Ajaakende kattuvus klastrite ühendamisel. Klastrite ühendamisel on kindlasti vaja arvestada ka klastrite klientide ajaakendega, sest ühe klastri kliendid teenindatakse alati korraga. Ajaakent arvestamata võib tekkida olukord, kus üks loodava klastri klient on teenindatav hommikul, mõni teine aga õhtul, mis tähendab, et auto peab vahepealse aja lihtsalt ootama. Antud küsimuse lihtsaks lahendamiseks võib näiteks kontrollida, et loodava klastri klientide ajaakende ühisosa moodustaks vähemalt k protsenti iga kliendi ajaaknast. Sügavama uurimise alt jätame selle küsimuse aga käesolevas töös välja.

6. Kui palju klasterdada? Sellele küsimusele vastamiseks tuleb aru saada, millal on kliendid teineteisele piisavalt lähedal, see aga sõltub peamiselt klientide jaotusest maastikul. Ka vaadeldava küsimuse jätame käesolevas töös lahtiseks.

7. Klasterdamise tulemuseks oleva transpordiülesande lahendi teisendamine esialgse ülesande lahendiks ehk klastri klientide läbimine (edaspidi *klastri avamine*). Vt. punkt 3.8.

3.3 Klastri klientide teenindatavus

Tuletame meelde, et klastri kliendid teenindatakse korraga, st ühe sõiduki poolt ühe sõidu jooksul. Ajaakendega ülesande puhul pole tegemist sugugi triviaalse nõudmisega. Olgu meil näiteks kaks klienti x ja y, mille on võrdne ajaaken (st. $b(x) = b(y)$ ja $e(x) = e(y)$) ning olgu nende ajaaken piisavalt väike, et kehtiks võrratus $t(x, y) > e(x) - b(x) = e(y) - b(y)$. Sel juhul ei saa nendest klastrit moodustada, sest ühest kliendist viimase ajaakna jooksul lahkudes ei ole võimalik jõuda teise selle ajaakna sees. Järelikult selleks, et mingist hulgast klientidest oleks võimalik klastrit moodustada, on tarvilik, et leiduks suvaline kõiki ülesande piiranguid järgiv teekond läbi antud klientide.

Kerkinud küsimusele vastamiseks piisab, kui lahendada väiksemahuline transpordiülesanne loodava klastri klientide jaoks, arvestades lisatingimusega, et kasutada võib ainult ühte sõitu.

Definitsioon 5 Klastriülesandeks *klastri C korral nimetame transpordiülesannet depooga $c \in C$ ja klientide hulgaga $C \setminus \{c\}$ ning tähistame seda P_C.*

See, milline klient valitakse klastriülesande depooks, selgub punktis 3.6. Tähtsal kohal on ka ülesande ajaaken $[b(P_C), e(P_C)]$. Selle defineerime punktis 3.5.

Olgu $C = \{c_1, \ldots, c_k\}$ klaster ning sõit $T = (t_1, \ldots, t_m)$ olgu vastava klastriülesande lahend mingi depoo valiku c korral. Paneme tähele, et kui $k = |C| = 1$, siis ka $m = 1$, st sõit T koosneb ainult klastriülesande depoost, sest klastriülesande klientide hulk $C \setminus \{c\}$ on tühi. Kui aga $k > 1$, siis $m = k + 1$, sest klastriülesande depoo on sõidus T esindatud kaks korda: $t_1 = t_m = c$.

Klastriülesande lahenduvus ühe sõiduga tähendab, et leidub koguni *kinnine* teekond läbi klastri klientide. Seda pole klastri klientide teenindatavuseks otseselt vaja, aga antud asjaolu kasutame ära järgnevates punktides.

3.4 Klastri atribuudid

Klastril kui metakliendil peavad olema defineeritud kõik kliendi atribuudid: tellimus, ajaaken ja teenindusaeg. Klastri tellimus on ilmselt klastri elementide tellimuste summa. Teenindusaeg ja ajaaken ei ole aga üheselt määratud. Siiski võime nende väärtustele esitada järgmise elementaarse tingimuse: ajaakna iga teenindusaja pikkuse lõigu jooksul peab olema võimalik ajakitsendust rikkumata teenindada kõik klastri kliendid. Teisest küljest on klasterdamise tulemuseks oleva ülesande jaoks parem, kui klastri ajaaken on võimalikult suur ning teenindusaeg võimalikult lühike. Sellisel juhul on loodavat klastrit võimalik läbida võimalikult erinevatel aegadel, seega klasterdamine piirab võimalike lahendite hulka võimalikult vähe.

Olgu antud klaster $C = \{c_1, \ldots, c_k\}$. Lahendame klastri C jaoks klastriülesande depooga $c \in C$. Ülesande lahenduvuse korral on tulemuseks üks sõit $T = (t_1, \ldots, t_m)$ (kui $|C| = 1$, siis $m = |C|$, vastasel juhul $m = |C| + 1$). Pakume välja järgmised eeskirjad atribuutide arvutamiseks.

- Tellimus. Klastri tellimus on ühendatavate klientide tellimuste summa e.

$$d(C) = \sum_{i=1}^{|C|} d(c_i) . \tag{6}$$

- Teenindusaeg.

$$s(C) = \begin{cases} s(c), & \text{kui } C = \{c\}, \\ ed(t_m) - ed(t_1), & \text{kui } |C| > 1. \end{cases} \tag{7}$$

- Ajaaken.

$$b(C) = ed(t_1) - s(c), \tag{8}$$

$$e(C) = ld(t_1) - s(c) + s(C). \tag{9}$$

Tõestame järgnevalt teoreemi selliselt arvutatud atribuutide väärtuste korrektsuse kohta. Olgu antud klaster $C = \{c_1, \ldots, c_k\}$ klastriülesande lahendiga $T = (t_1, \ldots, t_m)$, mille teenindusaeg $s(C)$ ning ajaaken $[b(C), e(C)]$ on arvutatud vastavalt võrranditele (7), (8) ja (9).

Teoreem 1 *Ajaakna $[b(C), e(C)]$ iga teenindusaja $s(C)$ pikkuse lõigu jooksul on võimalik ajakitsendust rikkumata teenindada kõik klastri C kliendid, alustades teenindamist klastriülesande depoost c.*

Tõestus

Vaatleme kõigepealt juhtu, kus $C = \{c\}$. Sel juhul koosneb sõit T ühest kliendist: $T = (t_1)$, kus $t_1 = c$ ning sõidu pikkus $m = 1$. Vastavalt võrrandile (7) on teenindusaeg $s(C) = s(c)$. Avaldame suurused $b(C)$ ja $e(C)$ võrrandites (8) ja (9) vastavalt varaseima ja hiliseima väljumisaja arvutamise valemitele (2) ja (3):

$$\begin{aligned} b(C) &= ed(t_1) - s(c) = \\ &= \max\{b(P), b(t_1)\} + s(t_1) - s(c) = \\ &= \max\{b(P), b(c)\} + s(c) - s(c) = \\ &= \max\{b(P), b(c)\}, \\ e(C) &= ld(t_1) - s(c) + s(C) = \\ &= ld(t_m) - s(c) + s(c) = \\ &= \min\{e(P), e(t_m)\} = \\ &= \min\{e(P), e(c)\}. \end{aligned}$$

Nagu näha, sisaldub klastri ajaaken kliendi c ajaaknas, järelikult on üheelemendiline klaster sellise ajaakna ja teenindusaja korral teenindatav, sest eelduse

kohaselt on klient enda ajaakna jooksul enda teenindusajaga teenindatav.

Olgu nüüd $|C| > 1$. Sel juhul on võrrandi (7) põhjal $s(C) = ed(t_m) - ed(t_1)$. Sisuliselt peame me näitama, et kõik klastri kliendid on teenindatavad ajaga $s(C)$, alustades tipu $t_1 = c$ teenindamist kell t, kus $b(C) \leq t \leq e(C) - s(C)$.

Lemmast 1 saame, et kui muuta klastriülesande depoost t_1 väljumise aega varaseimast väljumisajast aja Δt ($0 \leq \Delta t \leq ld(t_1) - ed(t_1)$) võrra hilisemaks, muutub tipu t_i ($i \in \{1, \ldots, m\}$) varaseim väljumisaeg *ülimalt* aja Δt võrra hilisemaks. Kasutades lemma tähistusi: kui $\overline{ed(t_1)} = ed(t_1) + \Delta t$, kus $0 \leq \Delta t \leq ld(t_1) - ed(t_1)$, siis

$$\forall i \in \{1, \ldots, m\} : \overline{ed(t_i)} \leq ed(t_i) + \Delta t.$$

Seega

$$\begin{aligned}
\overline{ed(t_m)} - \overline{ed(t_1)} &= \overline{ed(t_m)} - (ed(t_1) + \Delta t) \leq \\
&\leq ed(t_m) + \Delta t - ed(t_1) - \Delta t = \\
&= ed(t_m) - ed(t_1) = \\
&= s(C).
\end{aligned}$$

Vahe $\overline{ed(t_m)} - \overline{ed(t_1)}$ on aeg, mis kulub klastri kõikide klientide korrektseks läbimiseks vastavalt klastriülesande lahendile, kui tipust t_1 väljutakse ajahetkel $ed(t_1) + \Delta t$. Äsjatoodud võrratuse põhjal ei ületa see vahe klastri teenindusaega $s(C)$. Seega võime öelda, et kõik klastri kliendid on teenindatavad ajaga $s(C)$, alustades tipu t_1 teenindamist kell $ed(t_1) + \Delta t$, kus $0 \leq \Delta t \leq ld(t_1) - ed(t_1)$.

Näitame nüüd, et kui tipu t_1 teenindamist alustada ajahetkel t ($b(C) \leq t \leq e(C) - s(C)$), siis on kõik klastri kliendid teenindatavad ajaga $s(C)$. Kui me alustame tipu $t_1 = c$ teenindamist ajahetkel t, siis me väljume sealt kell $t + s(c)$ ehk $\overline{ed(t_1)} = t + s(c)$. Seega jääb üle näidata, et kui t kuulub lõiku $[b(C), e(C) - s(C)]$, siis $\Delta t = \overline{ed(t_1)} - ed(t_1) = t + s(c) - ed(t_1)$ kuulub lõiku $a = [0, ld(t_1) - ed(t_1)]$. Olgu $t = b(C)$, siis

$$\begin{aligned}
\Delta t &= t + s(c) - ed(t_1) = \\
&= b(C) + s(c) - ed(t_1) = \\
&= ed(t_1) - s(c) + s(c) - ed(t_1) = \\
&= 0,
\end{aligned}$$

mis kuulub lõiku a. Olgu $t = e(C) - s(C)$, siis

$$\begin{aligned}
\Delta t &= t + s(c) - ed(t_1) = \\
&= e(C) - s(C) + s(c) - ed(t_1) = \\
&= ld(t_1) - s(c) + s(C) - s(C) + s(c) - ed(t_1) = \\
&= ld(t_1) - ed(t_1),
\end{aligned}$$

mis kuulub samuti lõiku a. Kuna funktsioon $f(t) = t + s(c) - ed(t_1)$ on monotoonselt kasvav, kuuluvad ka t vahepealsete väärtuste korral Δt väärtused lõiku a. Seega on iga t väärtuse korral klastri kliendid teenindatavad ajaga $s(C)$.

MOTT

3.5 Klastri teenindatavus klasterdamise tulemuseks olevas ülesandes

Klastri loomisel tuleb silmas pidada ka seda, et loodud klastrit oleks üleüldse võimalik lahendisse lisada. Teiste sõnadega selle teenindusaeg ei tohi olla liiga suur võrreldes depoo või esialgse ülesande ajaaknaga. Teenindusaega lihtsalt lühemaks teha ei saa, sest klaster peab igal juhul olema selle aja jooksul teenindatav. Selle asemel saame fikseerida klastriülesandele sellise ajaakna, mis klastriülesande lahenduvuse korral tagab kindlasti klastri teenindatavuse klasterdamise tulemusena saadavas ülesandes.

Olgu meil antud transpordiülesanne P ajaaknaga $[b(P), e(P)]$ ja depooga d ning klaster C ja sellele vastav klastriülesanne P_C depooga c. Fikseerime ülesande P_C ajaakna järgmiselt:

$$\begin{aligned}
b(P_C) &= \max\{\max\{b(P), b(d)\} + s(d) + t(d, c), b(c)\}, \\
e(P_C) &= \min\{\min\{e(P), e(d)\} - s(d) - t(c, d), e(c)\}.
\end{aligned}$$

Toodud võrrandid ei tähenda midagi muud kui seda, et klastri teenindamist ei saa alustada varem kui on võimalik jõuda ülesande depoost d klastriülesande depoosse c ning ei saa lõpetada hiljem, kui on võimalik tipust c lahkudes jõuda depoosse. Mõlemal juhul arvestame nii depoo, klastriülesande depoo kui ülesande globaalset ajaakent.

3.6 Klastritevahelised kaugused

Oluline küsimus klasterdamise juures on ka see, kuidas määrata klastritevahelised kaugused e. klastritevaheliste teede maksumused. Üldisuse huvides samastame selles punktis üksikud kliendid üheelemendilise klastriga.

Hierarhilise klasterdamise puhul on enamlevinud kauguse definitsioonid järgmised (vt. [7]).

- Klastrite C_u ja C_v *ühelingikauguseks* nimetatakse suurust

  $$\min_{c_u \in C_u, c_v \in C_v} d\left(c_u, c_v\right),$$

  st minimaalset kaugust erinevatest klastritest pärit klientide vahel.

- Klastrite C_u ja C_v *täislingikauguseks* nimetatakse suurust

  $$\max_{c_u \in C_u, c_v \in C_v} d\left(c_u, c_v\right),$$

  st maksimaalset kaugust erinevatest klastritest pärit klientide vahel.

- Klastrite C_u ja C_v *keskmiseks lingikauguseks* nimetatakse suurust

  $$\frac{1}{|C_u| \cdot |C_v|} \sum_{c_u \in C_u, c_v \in C_v} d\left(c_u, c_v\right),$$

  st keskmist kaugust erinevatest klastritest pärit klientide vahel.

Näitame, et ükski toodud kauguse definitsioonidest ei sobi antud tingimustes transpordiülesande klientide klasterdamise ülesandesse.

Keskmine lingikaugus pole kasutatav sellepärast, et see ei paku välja konkreetset marsruuti klastrite vahel liikumiseks, kuid transpordiülesande lahendamisel on vaja teada klientidevaheliste marsruutide läbimisaegu $t\left(C_u, C_v\right)$.

Vaatame nüüd kaht esimest definitsiooni. Sisaldugu klaster C sõidus $T = (t_1, \ldots, t_m)$ positsioonil i ning maksumus $d\left(t_{i-1}, t_i\right)$ olgu leitud kui $d\left(c, c'\right)$ ($c \in t_{i-1}$ ja $c' \in C$) kas ühe- või täislingikauguse järgi. Näitame, et kui c' pole klastriülesande P_C depoo, siis selline maksumuse valik võib muuta sõidu T ajakitsendusele mittevastavaks. Sõidu T kohaselt saabume klastrisse C tipu c' kaudu, st me alustame C klientide läbimist tipust c'. Kui tipp c' pole klastriülesande depoo, puudub meil garantii, et klastri C kliendid on läbitavad klastri teenindusaja

$s(C)$ jooksul (vt. teoreem 1). Sellisel juhul võib klastrist C väljumine nihkuda hilisemaks, mis omakorda tähendab, et me võime mõnes hilisemas sõidu T poolt läbitavas kliendis ületada selle kliendi ajaakent.

Eelnevast arutelust on näha, et punktis 3.4 defineeritud teenindusaja ja ajaakna korral on ainsaks lubatavaks kauguse definitsiooniks selline, mis arvutab kaugusi klastriülesande depoost. Kaugused määrame järgmise arvutuseeskirjaga.

- Olgu C_1 ja C_2 klastrid ning c' ja c'' vastavalt nende klastriülesannete depood. Sel juhul defineerime nendevahelise kauguse järgmiselt:

$$d(C_1, C_2) = d(c', c'').$$

- Olgu C klaster depooga c' ja olgu c üksik klient. Sel juhul defineerime järgnevad kaugused:

$$\begin{aligned} d(C, c) &= d(c', c), \\ d(c, C) &= d(c, c'). \end{aligned}$$

Klastriülesande depooks c' valime klastri sellise kliendi, mille korral avaldise

$$\sum_{c \in C} \left(d(c', c) + d(c, c') \right)$$

väärtus on minimaalne. Siinkohal kasutame mõlemasuunalist kaugust, sest me ei tea, mis pidi läbitakse klastriülesande lahendamisel või klastri klientide läbimisel servi klastri tippude vahel. Kui siin võtta arvesse servi ainult ühes suunas, võib juhtuda, et klastriülesande lahendamisel oleks optimaalne läbida mõni serv vastupidises suunas, mis võib lahendi hüppeliselt halvemaks muuta.

3.7 Hierarhilise klasterdamise efektiivne rakendamine transpordiülesandes

Klasterdamisalgoritmi rakendamisel transpordiülesandele tuleb arvestada, et klientidevahelised kaugused pole arvutatavad konstantse ajalise keerukusega, nagu seda on näiteks eukleidiline kaugus. Antud ülesandes on kauguseks klientidevahelise odavaima tee maksumus. Odavaima tee leidmiseks on küll olemas suhteliselt efektiivseid algoritme (näit. Dijkstra algoritm keerukusega $O(n \log n)$, vt.

[5]), aga hierarhilise klasterdamise puhul tuleb pidevalt arvutada uusi kaugusi elementide vahel – peale kahe lähima klastri ühendamist algoritmi 1 sammul 4 tuleb leida kaugused loodud klastrist kõikidesse teistesse klastritesse, et leida järgmine lähim klastrite paar.

Siinkohal aga saame ära kasutada seda, et me defineerisime klastritevahelise kauguse punktis 3.6 kui klastrite depoode vahelise kauguse, kus depooks on mingi vastava klastri klient. See tähendab, et me saame taaskasutada varem väljaarvutatud kaugusi. Algoritmi 1 sammul 1 me konstrueerime üheelemendilised klastrid iga kliendi jaoks. Sammul 2 leiame lähima klastrite paari, st et me peame igal juhul välja arvutama kaugused kõigi klientide vahel. Paneme need kirja kaugustemaatriksisse. Uue klastri loomisel saame klastri depooks olevale kliendile vastavat kauguste maatriksi rida vaadelda loodava klastri kauguste maatriksi reana, st et enam pole vaja täiendavalt arvutada ühtegi kaugust. Toodud protsess on täpsemalt kirjeldatud lisas A algoritmidena 3 – 6.

3.8 Klastrite avamine

Klasterdamise tulemusena saadakse uus transpordiülesanne, mis sisaldab väiksemat arvu kliente kui esialgne ülesanne, sest osad kliendid on ühendatud klastriteks. Saadud ülesande lahendi põhjal on vaja kuidagi konstrueerida lahend esialgsele transpordiülesandele, st iga klastri kliendid on vaja läbida mingis järjekorras.

3.8.1 Klastri avamine vastavalt klastriülesande lahendile

Üks võimalik korrektne läbimisjärjekord, mis alati leidub, on määratud klastriülesande lahendiks oleva sõiduga. Samas tuletame meelde, et klastriülesande lahendiks olev sõit on kinnine, st ta algab ja lõppeb klastriülesande depoos e. mingis klastri kliendis, aga kliente tuleb lahendis läbida alati täpselt üks kord. Seega ei saa kasutada täpselt sama järjekorda, aga lihtsa parandusena võib klastriülesande depoosse naasmise ära jätta ning siirduda kohe klastrist järgmise metakliendini. Formaalsemalt, olgu antud klaster C klastriülesande lahendiga $T_C = (t'_1, \dots, t'_{m'})$ ja depooga d_C ning olgu C lisatud sõitu $T = (t_1, \dots, t_m)$ positsioonile i (st. $t_i = C$). Sel juhul konstrueerime uue sõidu T', milles on klaster C asendatud klientidega $d_C = t'_1, t'_2, \dots, t'_{m'-1}$, saame $T' = \big(t_1, \dots, t_{i-1}, t'_1, \dots, t'_{m'-1}, t_{i+1} \dots, t_m\big)$.

Paneme tähele, et tarviliku muudatusega, kus jäetakse välja klastriülesande depoosse naasmine, ei vasta me enam eeldustele sõidu T korrektse läbimise osas: minnes klastrist C sõidu T järgmisesse klienti t_{i+1} arvestasime me

sõiduaja mõttes servadega $\left(t'_{m'-1}, d_C\right)$ ja (d_C, t_{i+1}) – vastavalt klastri C teenindusaja ja klastritevahelise kauguse arvutamisel. Sõidus T' kasutame nende asemel serva $\left(t'_{m'-1}, t_{i+1}\right)$. Probleem seisneb selles, et erinevate hinnangufunktsioonide korral ei pruugi graaf rahuldada servade ajalise pikkuse mõttes kolmnurgavõrratust ning serv $\left(t'_{m'-1}, t_{i+1}\right)$ võib olla ajaliselt pikem kui servad $\left(t'_{m'-1}, d_C\right)$ ja (d_C, t_{i+1}) kokku (vt. punkti 2.2.1). Viimasel juhul tuleb muuta ülesande graafi selliselt, et klientide $t'_{m'-1}$ ja t_{i+1} vahelise marsruudi ajaline pikkus ei ületaks aega $t\left(t'_{m'-1}, d_C\right) + t\left(d_C, t_{i+1}\right)$. See muudatus on alati teostatav – marsruudiks võib valida nendesamade servade järjestikusele läbimisele vastava marsruudi $t'_{m'-1}, \ldots, d_C, \ldots, t_{i+1}$.

3.8.2 Klastri avamise optimeerimine

Eelnevas punktis näitasime, et klastrite avamine on igal juhul korrektselt teostatav, kasutades klastriülesande lahendile vastavat klientide järjestust ning muutes vajadusel tippudevahelisi teid. Selline järjestus ei pruugi aga anda ülesandele kuigi head lahendust. Vaatame näidet joonisel 1. Probleemiks on see, et klastriülesanne ei tea midagi sõidust, millesse antud klaster hiljem sisestatakse. Klastriülesande eesmärk on ju ka hoopis teine – pakkuda välja klastri atribuudid ning anda garantii klastri korrektse läbimise kohta *sõltumata* ümbritsevast sõidust.

Eelnevast lähtuvalt peaks klastri klientide läbimisjärjekorra leidmisel arvestama sõiduga, milles klaster sisaldub. Lõpliku järjekorra leidmiseks võib jällegi lahendada väiksemahulise transpordiülesande.

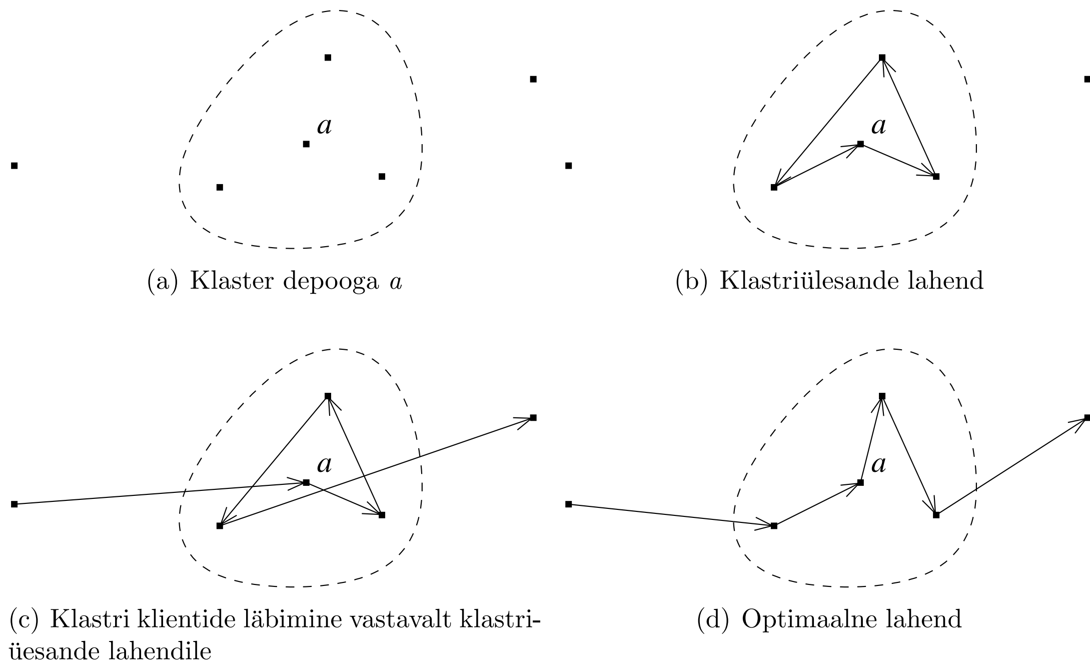
Sisaldagu sõit $T = (t_1, \ldots, t_m)$ klastrit C positsioonil i.

Definitsioon 6 Klastri C avamise ülesandeks *sõidus T nimetame transpordiülesannet P'_C, mille* algusdepoo *on t_{i-1}, klientide hulk on C ning* lõppdepoo *on t_{i+1}.*

Kuna me depood ei klasterda, siis tipud t_1 ega ka t_m ei saa olla klastrid, seega indeksite $i-1$ ja $i+1$ kasutamine on antud juhul korrekne.

Tähtsal kohal on defineeritud ülesande ajaaken. Selle fikseerime algoritmis 2.

Paneme tähele, et kui sõidu $T = (t_1, \ldots, t_m)$ klastreid hakata avama loomulikus, st tippude indeksite kasvamise järjekorras, siis eelneb järgmisena avatavale klastrile alati üksik tipp, sest t_1 on alati depoo, mida ei klasterdata. Lähtuvalt sellest hakkame konstrueerima esialgse transpordiülesande lahendit T' alustades



(a) Klaster depooga a

(b) Klastriülesande lahend

(c) Klastri klientide läbimine vastavalt klastriüesande lahendile

(d) Optimaalne lahend

Joonis 1: Klastri klientide läbimine esialgse ülesande lahendi koosseisus

üheelemendilisest sõidust (t_1). Edasi võtame järjest vaatluse alla tipud $t_2, \ldots, t_m$. Lisame sõidu T' lõppu vaadeldava tipu t_i kui see on üksik klient, vastasel juhul klastri t_i tipud vastavalt klastri avamise ülesande lahendile. Võtame kirjeldatud protsessi detailsemalt kokku algoritmiga 2. Algoritmi tööd illustreerib joonis 2.

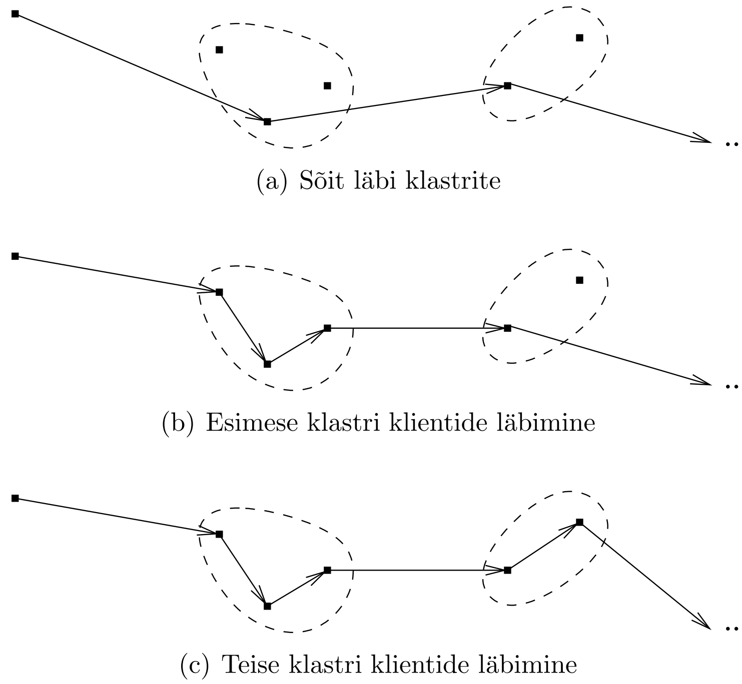
Algoritmi 2 korrektsus tuleneb sellest, et iteratsioonil i me kaasame järgmise metakliendi t_{i+1} lõppdepoona klastri avamise ülesandesse ning valime ülesande ajaakna sobivalt. Punktis 3.8.1 nägime, et läbides klastri kliendid vastavalt klastriülesande lahendile ning muutes vajadusel tippudevahelisi teid, jõuame me järgmise metakliendi ära teenindada juba ajahetkeks $ed\,(t_{i+1}) \leq ld\,(t_{i+1}) = e\,(P'_C)$. Varaseima väljumisaja kasutamine antud juhul pole aga kõige otstarbekam, sest meie lõppeesmärk on leida võimalikult *odav*, mitte tingimata kõige *kiirem* lahendus. Klastri avamise ülesande odavaim lahend aga võib olla ajaliselt pikem kui klastriülesande parim lahend. Kasutades ajaakna lõpuna hiliseimat väljumisaega $ld\,(t_{i+1})$, võimaldame leida klastri avamise ülesandele paremaid lahendeid kui seda on klastriülesande poolt defineeritud lahend. Samas sõidu T ülejäänud tipud $(t_{i+1}, \ldots, t_m)$ jäävad korrektselt teenindatavaks, sest tänu klastri avamise ülesande ajaakna lõpule $e\,(P'_C) = ld\,(t_{i+1})$ peame jõudma metakliendi t_{i+1} ära teenindada hiljemalt selle hiliseimaks väljumisajaks. Lõpuks paneme veel tähele,

Algoritm 2 Klastri avamine

Antud: sõit $T = (t_1, \dots, t_m)$.

Tulemus: sõit $T' = (t'_1, \dots, t'_{m'})$, milles on sõidu T klastrid asendatud klastri klientide järjendiga.

1. $T' := (t_1)$ – sõit algab depoost t_1 ($t'_1 = t_1$).
2. Iga $i = 2, \dots, m$ korral
   (a) Kui tipp t_i on üksik tipp, lisada see sõidu T' lõppu ning alustada järgmist iteratsiooni sammul 2, vastasel korral jätkata sammust b.
   (b) Olgu C vaadeldav klaster t_i klastriülesandega P_C ja klastriülesande lahendiga $T_C = \left(\overline{t_1}, \dots, \overline{t_{\overline{m}}}\right)$.
   (c) Konstrueerime klastri C avamise ülesande P'_C järgnevate atribuutidega:
      - algusdepoo $t'_{m'}$ – sõidu T' hetkel viimane tipp,
      - lõppdepoo t_{i+1},
      - klientide hulk C,
      - ajaaken $b\left(P'_C\right) = ed\left(t'_{m'}\right) - s\left(t'_{m'}\right)$ kuni $e\left(P'_C\right) = ld\left(t_{i+1}\right)$.
   (d) Lahendame ülesande P'_C, lahendiks olgu sõit $T'_C = (t''_1, \dots, t''_{m''})$.
   (e) Kui sõit T'_C on ebakorrektne, jätkata sammust f, vastasel juhul sammust g.
   (f) Muuta serva $\left(\overline{t_{\overline{m}-1}}, t_{i+1}\right)$ selliselt, et ta poleks ajaliselt pikem kui $t\left(\overline{t_{\overline{m}-1}}, \overline{t_{\overline{m}}}\right) + t\left(\overline{t_{\overline{m}}}, t_{i+1}\right)$. Lahendada ülesanne P'_C uuesti, lahendiks olgu T'_C (T'_C on korrektne lahend, vt punkt 3.8.1).
   (g) Lisada tipud $t''_2, \dots, t''_{m''-1}$ sõidu T' lõppu.



(a) Sõit läbi klastrite

(b) Esimese klastri klientide läbimine

(c) Teise klastri klientide läbimine

Joonis 2: Algoritmi 2 töö

et kuna me sõitu T' kliente lisades arvestame alati järgnevate tippude teenindatavusega, siis sobib klastri avamise ülesande ajaakna alguseks just sõidu T' viimase kliendi teenindamise algus $ed\left(t'_{m'}\right) - s\left(t'_{m'}\right)$.

4 Lahendusmeetod sipelgakoloonia süsteem

Käesolevas peatükis tutvustame sipelgakolooniate modelleerimisel põhinevaid lahendusmeetodeid, mis on toodud artiklites [2, 3, 4]. Esmalt tutvustame punktis 4.1 põhialuseid, millel sipelga-algoritmid baseeruvad. Punktis 4.2 vaatleme rändkaupmeheülesande lahendusmeetodit *sipelgakoloonia süsteem* (*Ant Colony System*, edaspidi ACS, vt. [2, 3]), mis on esimesi sipelga-algoritmide rakendusi antud töö valdkonnas. Punktis 4.3 käsitleme lahendusmeetodit *mitme sipelgakoloonia süsteem ajaakendega transpordiülesande lahendamiseks* (*Multiple Ant Colony System for Vehicle Routing Problem with Time Windows*, edaspidi MACS-VRPTW, vt. [4]), mis on eelmise meetodi edasiarendus transpordiülesandele. Vaadeldavad meetodid on viidatud artiklite põhjal näidanud väga häid tulemusi testülesannete lahendamisel. Lisas B on toodud vaadeldavaid meetodeid realiseerivad algoritmid, mis põhinevad originaalartiklites esitatutele.

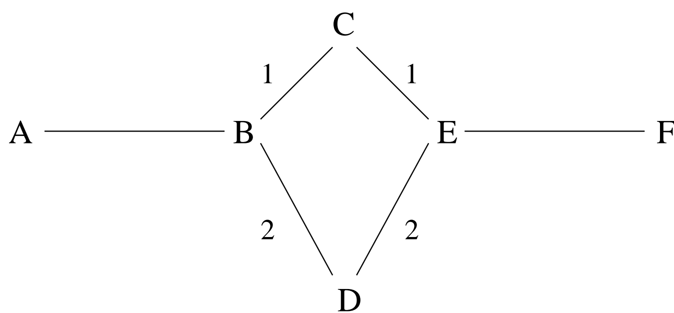
4.1 Tehissipelgad ning nende keskkond

Mõlemad meetodid ACS ja MACS-VRPTW kasutavad lahendite konstrueerimiseks lihtsate võimete ja omadustega agente, millede tegevus imiteerib mõnes mõttes reaalse maailma sipelgate käitumist. Seetõttu nimetame mainitud agente *tehissipelgateks*.

On täheldatud, et üksik sipelgas valib oma liikumissuundi üsna juhuslikult. Teisest küljest on teada, et sipelgad kasutavad näiteks pesast toidukohani liikumiseks lühimaid võimalikke teid, kuigi nad on peaaegu pimedad. Vastavate uuringute käigus avastati, et sipelgad jätavad endast liikumisel maha *feromooniraja*, mille mõju on järgmine: kui on valida mitme tee vahel, valitakse see, millel on rohkem feromooni. Sealjuures feromooni kogus läbitud teel suureneb. Seega järgmiste sipelgate saabumisel on tõenäosus veelgi suurem, et nad valivad vaadeldava tee. Sisuliselt on feromoonirada vahend liikumissuundade kohta käiva informatsiooni vahetamiseks isendite vahel.

Kirjeldame näite abil, kuidas on feromooniradadest kasu lühimate teede leidmisel. Kujutame sipelgate elukeskkonda graafina (joonis 3). Olgu tipp A sipelgapesa ning tipp F avastatud toidukoht ning sipelgad paiknegu kummaski tipus. Pesas A asuvate sipelgate eesmärk on jõuda toiduni, toidukohas F asuvate eesmärk on jõuda koos toiduga tagasi pessa. Huvi pakub sipelgate käitumine tee valimisel hargnemispunktides B ja E. Vaatleme lähemalt suunda $A \to F$ (suuna $F \to A$ korral sobib analoogiline arutelu). Tipust A tippu B jõudes on võimalik

valida kahe teekonna vahel: BCE ja BDE. Kuna alguses pole teid e. antud näites graafi servi veel läbitud, pole neil ka feromooni. Seetõttu valitakse kumbki kahest võimalikust teest võrdse tõenäosusega. Eeldame, et umbes pooled sipelgatest valivad serva BC ja pooled serva BD. Arvestades vastavaid teepikkusi, jõuavad tee BCE valinud sipelgad kohale varem, märgistades sealjuures läbitud raja feromooniga. Samaaegselt vastupidises suunas $F \rightarrow A$ liikuvad sipelgad avastavad serval EC tugevama feromooni jälje kui serval ED, kuna sellel on liigutud juba nii marsruudil $A \rightarrow F$ kui $F \rightarrow A$. Seetõttu valib suurem osa sipelgaid kahest võimalikust teest esimese. Selle tulemusena hakkab feromooni kogus lühemal teel suurenema kiiremini kui pikemal ja lõpuks liiguvad kõik sipelgad mööda lühemat teed.



Joonis 3: Näitlik tehissipelga elukeskkond

Esitatud feromooniraja kui informatsioonivahetuskanali ideed kasutame ära ka loodavate tehissipelgate juures. Juhinduvalt eesmärgist, milleks on rändkaupmehe või transpordiülesande lahendamine, on mõistlik kohandada loodavate tehissipelgate ning nende keskkonna omadusi vastavalt ülesandele.

- Tehissipelgate elukeskkond on graaf.
- Igal graafi serval on mingi kogus feromooni, mis jaotub selle serva piires ühtlaselt.
- Tehissipelgad omavad mõningast mälu. Täpsemalt, neil on meeles läbitud tippude nimistu.
- Tehissipelgad ei ole päris pimedad, olles võimelised "nägema" teede ehk graafi servade maksumusi.

- Liikumisel ühest graafi tipust teise feromooni kogus läbitud serval mitte ei suurene vaid väheneb. Sellise lähenemisega muudetakse juba läbitud servi vähem atraktiivseks, et teised sipelgad võtaksid vaatluse alla uusi ja potentsiaalselt kasulikumaid teid. Niimoodi toimida võib tänu sellele, et erinevalt päris sipelgatest, kes saavad tee optimaalsusest teada vaid sellel paikneva feromooni intensiivsuse järgi, teame meie iga serva korral tema maksumust ning saame selle alusel servi omavahel võrrelda.

- Feromooni koguse suurendamine graafi servadel toimub pärast mingi etteantud arvu teede genereerimist ja ainult senise parima lahendi alusel. Selle tulemusel liiguvad sipelgad pigem lühemate teede ümbruses ja võivad suurema tõenäosusega avastada uusi parimaid teid.

4.2 Sipelgakoloonia süsteem

Sipelgakoloonia süsteem (vt. [2, 3]) on meetod rändkaupmeheülesande lahendamiseks.

Olgu antud graaf $G = (X, E)$, kus X on linnade hulk ning E on linnadevaheliste servade hulk. Linnade $x, y \in X$ vaheline kaugus olgu $d(x, y)$ ning lahendi ehk kõiki linnu läbiva tee T maksumus olgu $W(T)$.

Sipelgakoloonia süsteemi rakendamiseks tuleb iga servaga $(x, y) \in E$ siduda järgmised suurused.

- $\eta(x, y) \in [0, 1]$ – serva *nähtavus*. Staatiline suurus, mis on pöördvõrdeline serva maksumusega. Seega $\eta(x, y) = \frac{1}{d(x,y)}$.

- $\tau(x, y) \in \mathbb{R}, \tau(x, y) \geq 0$ – feromooni kogus serval. See on muutuv suurus, mida kasutatakse arvutuste käigus. Feromooni koguse algväärtus on τ_0.

Artikli [4] põhjal võtame $\tau_0 = \frac{1}{nW\left(T^l\right)}$, kus T^l on esialgne lahend, mis on arvutatud *lähima naabri algoritmiga*. Lähima naabri algoritm on lihtne ahne algoritm: asudes linnas x, valitakse järgmiseks selline seni läbimata linn y, mille korral kaugus $d(x, y)$ on vähim.

Selleks, et vältida sipelgate tagasipöördumist juba külastatud linnadesse, peame kuidagiviisi säilitama läbitud linnade nimekirju iga sipelga kohta. Olgu k. sipelga poolt läbitud tee T^k ning $T^k(j)$ olgu j. tipp sellel teel ehk k. sipelga poolt j-ndana läbitud linn.

Olgu meil m tehissipelgat, mida kasutame ülesande lahendamiseks. Protsessi alguses paigutame nad kõik sõltumatult suvalistesse linnadesse. Sipelgad konstrueerivad teid paralleelselt: protsess koosneb sammudest, mille käigus iga sipelgas valib järgmise läbitava tipu ja liigub sinna. Kuna moodustatavad teed peavad läbima kõiki vahepealseid linnu täpselt ühe korra ja naasma alguslinna, saame n. sammu lõpus kätte m lahendit püstitatud ülesandele.

Asugu k. sipelgas linnas x. Võtame kasutusele parameetrid $q \in [0,1]$ ja $\beta \in \mathbb{N}$, mille tähendust kirjeldame allpool. Järgmise linna $y \in X \setminus V\left(T^k\right)$ valik toimub tõenäosuslikult etteantud parameetri q alusel. Valime juhusliku suuruse $r \in [0,1]$. Kui

- $r \leq q$, valitakse järgmiseks selline linn, mille korral avaldise $\tau(x,y)(\eta(x,y))^\beta$ väärtus on maksimaalne. Sellist valikustrateegiat nimetame *ekspluatatsiooniks*, kuna siin valitakse alati "parim" serv, ehk kasutatakse ära teiste sipelgate poolt jäetud feromooni jälgi.

- $r > q$, valitakse järgmine linn y tõenäosusega $p(x,y)$, kus

$$p(x,y) = \begin{cases} \dfrac{\tau(x,y)(\eta(x,y))^\beta}{\sum_{z \in X \setminus V\left(T^k\right)} \tau(x,z)(\eta(x,z))^\beta} & \text{kui } y \in X \setminus V\left(T^k\right), \\ 0 & \text{vastasel juhul.} \end{cases}$$

  Näeme, et järgmise linna valikul eelistatakse servi, millel on rohkem feromooni ja mis on lühemad. Seda valikustrateegiat nimetame *eksploratsiooniks*, kuna järgmise serva valimine toimub tõenäosuslikult ja võidakse avastava uusi teid.

Parameeter β määrab, kas tee konstrueerimisel arvestatakse rohkem serva nähtavust või feromooni kogust ning q määrab, kumba valikustrateegiat kasutatakse suurema tõenäosusega.

Iga serva läbimise järel mingi sipelga poolt toimub *lokaalne feromooni uuendamine*, mille käigus vähendatakse äsja läbitud serval asuvat feromooni kogust. Nagu ülalpool juba mainitud, on see kasulik selleks, et liiga paljud sipelgad ei kasutaks lahendi otsimise käigus sama serva ning püüaksid pigem uusi teid läbida. Liikudes linnast x linna y toimub uuendamine järgnevalt:

$$\tau(x,y) := (1-p)\,\tau(x,y) + p\tau_0, \tag{10}$$

kus τ_0 on feromooni algväärtus (toodud eespool) ning $p \in [0, 1]$ on etteantud parameeter, mis väljendab feromooni aurustumist aja jooksul.

Pärast n sammu läbimist on kõik m sipelgat konstrueerinud lahendi püstitatud rändkaupmeheülesandele. Järgmiseks toimub *globaalne feromooni uuendamine*, mille käigus muudetakse feromooni koguseid vastavalt senisele parimale lahendile T^P:

$$\forall (x, y) \in E\left(T^P\right) : \tau(x, y) := (1 - p)\, \tau(x, y) + \frac{p}{W(T^P)}. \tag{11}$$

Kirjeldatud protsessi täidame tsükliliselt, paigutades m sipelgat uuesti sõltumatult juhuslikesse alguslinnadesse. Tsüklit täidetakse seni, kuni järgnev *lõpetamistingimus* on tõene, kus

$$lõpetamistingimus = \begin{cases} tõene, & \text{kui kehtib vähemalt üks järgnevatest:} \\ & \text{- protsessi algusest on möödas etteantud aeg,} \\ & \text{- genereeritud on etteantud arv lahendusi,} \\ & \text{- lahendit pole paranduatud etteantud arvu} \\ & \quad \text{tsüklite jooksul,} \\ väär, & \text{vastasel juhul.} \end{cases} \tag{12}$$

Kirjeldatud protsessi realiseerib lisas B toodud algoritm 7.

4.3 Mitme sipelgakoloonia süsteem ajaakendega transpordiülesande lahendamiseks

Olgu meil antud graaf $G = (X, E)$ ning sellel defineeritud transpordiülesanne P. Lahendi S maksumus olgu $W(S)$. Analoogiliselt eelmise punktiga on ka siin vaja iga servaga $(x, y) \in E$ siduda feromooni kogus serval $\tau(x, y)$ ning serva nähtavus $\eta(x, y)$.

Antud lahendusmeetod minimeerib korraga nii sõitude arvu kui ka lahendi maksumust. Sealjuures kahest lahendist loetakse paremaks väiksema sõitude arvuga lahendit vaatamata sellele, kas lahendi maksumus on väiksem või mitte. Mõlema eesmärgi jaoks kasutatakse oma sipelgakoloonia. Üks neist, *ACS-vehicle* (punkt 4.3.2), tegeleb sõitude arvu vähendamisega. Teine, *ACS-length* (punkt 4.3.3), minimeerib lahendi maksumust, kasutades vähimat arvu sõite, millega ACS-vehicle on korrektse lahendi leidnud. Kumbki koloonia kasutab eraldi fero-

mooni väärtusi, aga ühist muutujat S^{gl} senise globaalselt parima lahendi hoidmiseks. Algväärtus ehk mingi korrektne lahendus suurusele S^{gl} leitakse lähima naabri algoritmiga. Koloon iate tööd koordineerib juhtprotseduur, mis on kirjeldatud punktis 4.3.1.

Teede konstrueerimiseks kasutavad mõlemad kolooniad üksikut sipelgat, mida realiseerib protseduur `new_active_ant` (vt. punkt 4.3.4). Alustades depoost läbib sipelgas kliente, kuni tal tuleb kas mahu- või ajakitsenduse tõttu tagasi depoosse pöörduda. Analoogiliselt sõithaaval läbitakse ülejäänud kliendid. Arvestada tuleb aga sellega, et igal lahendi leidmisel on kindlaks määratud sõitude maksimaalarv, olgu selleks v. Sellega arvestamiseks dubleerime enne protsessi algust esialgses graafis depoo d ja kõik temaga seotud servad v korda. Tulemuseks on $v+1$ dubleeritud depood, mille omavahelised kaugused on 0. Niimoodi saame rändkaupmeheülesandele sarnase ülesandepüstituse – tuleb konstrueerida üksainus sõit, mis algab ja lõppeb mingis dubleeritud depoos ning läbib kõik kliendid täpselt üks ja kõik dubleeritud depood ülimalt üks kord.

4.3.1 Juhtprotseduur

Olgu v globaalse parima lahendi S^{gl} poolt kasutatud sõidukite arv. Juhtprotseduur `MACS-VRPTW` (algoritm 8, lisa B) käivitab protseduurid `ACS-vehicle` ning `ACS-length`. `ACS-vehicle` püüab leida lubatavat, sealjuures mitte tingimata väikse maksumusega lahendust, mis kasutab maksimaalselt $v-1$ sõidukit. Protseduur `ACS-length` minimeerib lahendi maksumust, otsides uusi lahendusi kasutades ülimalt v sõitu. Kui üks kolooniatest on leidnud parema lahendi kui S^{gl}, uuendatakse muutuja S^{gl} väärtust. Kui uue parima lahendi sõitude arv on väiksem kui v, peatatakse mõlemad kolooniad ning käivitatakse nad uuesti vastavalt uuele sõitude arvule.

4.3.2 Sõitude arvu minimeeriv koloonia ACS-vehicle

Kolooniat ACS-vehicle realiseerib lisas B toodud algoritm 10. Algoritm saab argumendiks sõitude maksimaalarvu v, mis on ühe võrra väiksem kui sõitude arv senisel parimal lahendil. Suuruse v alusel initsialiseeritakse graaf ja feromooni algväärtused. Antud juhul pole teada, kas sellise sõitude arvuga lubatav lahend üldse eksisteerib. Seetõttu tegeleb ACS-vehicle sisuliselt külastatud klientide arvu maksimeerimisega ning hoiab senileitutest kõige rohkem kliente läbinud lahendit muutujas $S^{vehicle}$ (lokaalselt parim lahend). Muutuja $S^{vehicle}$ algväärtus

leitakse lähima naabri algoritmiga, kasutades v sõitu. Algväärtus ei pruugi olla korrektne, sest võib eksisteerida külastamata kliente. Kasutades protseduuri `new_active_ant` leitakse igal välimise tsükli sammul ülesandele uued lahendid $S^1, \ldots, S^m$. Kui nendest mõni, näit. S^k, külastab rohkem kliente, kui senine lokaalselt parim lahend $S^{vehicle}$, võetakse S^k uueks lokaalselt parimaks lahendiks. Kui nüüd $S^{vehicle}$ külastab kõiki kliente, olemegi leidnud ülesandele lubatava lahendi, mis kasutab v sõitu, ning saadame selle juhtprotseduurile `MACS-VRPTW`.

Selleks, et kord-korralt üha rohkem kliente külastada, võtame kasutusele suurused $IN(x)$ $(x \in X)$, mis on klienti x mitte läbivate lahendite arv. Suurused $IN(x)$ antakse parameetrina kaasa protseduurile `new_active_ant`, mis eelistab lahendi konstrueerimisel vähem külastatud kliente. Uue lokaalselt parima lahendi leidmisel suurused $IN(x)$ nullitakse.

Tsükli lõpus toimub globaalne feromooni uuendamine. Uuendamiseks kasutatakse nii lokaalselt kui globaalselt parimat lahendit, vastavalt $S^{vehicle}$ ja S^{gl}. Põhjus on selles, et uuendamisel ainult suuruse $S^{vehicle}$ järgi ei suurene feromooni kogus ühelgi külastamata klienti suubuval serval. Kasutades uuendamiseks ka lahendit S^{gl}, leidub iga kliendi korral mingi sisenev serv, millel feromooni kogust suurendatakse. Teisest küljest, uuendamisel ainult S^{gl} põhjal ei saavutaks me läbitavate klientide arvu suurendamise osas efekti, kuna S^{gl} kasutab lubatust rohkem sõite.

4.3.3 Maksumust minimeeriv koloonia ACS-length

Kolooniat ACS-length realiseerib lisas B toodud algoritm 9. Argumendiks saadakse lahendis kasutatavate sõitude arv v. Esmalt dubleeritakse depood v korda ning initsialiseeritakse feromooni kogused graafi servadel. Seejärel genereeritakse protseduuriga `new_active_ant` lahendid $S^1, \ldots, S^m$ iga sipelga kohta. Kui saadud lahendite hulgas leidub mõni, mis on korrektne ja mille maksumus on väiksem senise parima lahendi S^{gl} omast, saadetakse see protseduurile `MACS-VRPTW`. Peale uute lahendite genereerimist toimub lahendi S^{gl} alusel globaalne feromooni uuendamine.

4.3.4 Üksik sipelgas – lahendeid konstrueeriv protseduur

Protseduur `new_active_ant` (algoritm 11, lisa B) saab argumendiks lahendatava transpordiülesande P, tõeväärtusmuutuja *lokaalne_otsing* ja suurused $IN(x)$. Viimaseid kasutatakse ainult koloonia `ACS-vehicle` korral. Põhimõte

on sarnane rändkaupmehe ülesande lahendusmeetodile ACS (vt. punkt 7), aga siin on tee generereerijaks üksainus sipelgas. Erinevus on veel selles, et genereeritud lahendused ei pruugi olla korrektsed, st külastada kõiki kliente. Protsessi alguses paigutatakse sipelgas suvalisse dubleeritud depoosse x, mis ühtlasi pannakse konstrueeritava lahendi S esimeseks elemendiks. Jooksvat aega ja kantava koorma suurust hoitakse vastavalt muutujates *aeg* ja *last*, mis algväärtustatakse vastavalt dubleeritud depoole x.

Programmi põhiosas lisatakse lahendisse tsükliliselt tippe. Paiknegu sipelgas tsükli alguses tipus x. Võtame kasutusele suuruse M kui kõigi *lubatavate* tippude hulga. Lubatavaks peame sellist veel külastamata tippu, mida on võimalik kitsendusi rikkumata muutujate *aeg* ja *last* hetkeväärtusi arvestades järgmisena külastada. Täpsemalt, asudes tipus x kuulub tipp $y \in X$ lubatavate tippude hulka M parajasti siis kui kehtivad järgmised tingimused.

1. Vähemalt üks tippudest x ja y pole depoo – järjestikune depoode külastamine ei anna klientide teenindamise juures mingit efekti.

2. $y \notin S$ – tippu ei ole veel külastatud.

3. $last + d(y) \leq v(P)$ – tipu y tellimus mahub sõidukile.

4. $aeg + t(x, y) + s(y) \leq e(y)$ – tipp y jõutakse ära teenindada tema ajaakna jooksul. Paneme tähele, et siin pole tarvis kasutada maksimumi tippu y kohale jõudmise ajast $aeg + t(x, y)$ ja tipu y ajaakna algusest $b(y)$, sest kui maksimum saavutatakse tänu ajaakna alguse hilisusele, on tipp kindlasti teenindatav, sest punktis 2.2.1 toodud eelduse kohaselt kehtib $s(y) \leq e(y) - b(y)$.

5. kui y pole depoo, siis

$$\max\{aeg + t(x, y), b(y)\} + s(y) + t(y, d) + s(d) \leq \min\{e(P), e(d)\},$$

kus d on transpordiülesande depoo. See tingimus tagab, et kui y on klient, siis on võimalik pärast selle teenindamist jõuda tagasi depoosse enne nii ülesande kui depoo ajaakna lõppu. Antud tingimus on vajalik tänu sellele, et kasutame ülesandepüstituses ajaakNaid ka ülesande ja depoo juures (vt. punkt 2.2.1).

Asudes parajasti tipus x, arvutame iga tipu $y \in M$ jaoks serva nähtavuse $\eta(x, y)$ järgmiselt:

$$\eta(x,y) := \frac{1}{\max\left(1, \left(\max\left(aeg + t(x,y), b(y)\right) + s(y) - aeg\right) \cdot \left(e(y) - aeg\right) - IN(y)\right)}.$$

Servade nähtavuste sellise väärtustamisega eelistatakse neid kliente y, mille jooksvast hetkest alates varaseim ja hiliseim võimalik teenindamise aeg on võrdlemisi lähedal ehk vastavalt suurused $\max(aeg + t(x,y), b(y)) + s(y) - aeg$ ja $e(y) - aeg$ on suhteliselt väikesed.

Algoritmi järgmise sammuna valitakse tõenäosuslikult ekspluatatsiooni või eksploratsiooni (vt. punkt 7) abil hulgast M järgmise läbitav tipp y ning lisatakse see lahendisse. Seejärel uuendatakse vastavalt valitud tipule muutujaid aeg ja $last$. Tsükli lõpus toimub lokaalne feromooni uuendamine (10) vastavalt läbitud servale. Tsüklit täidetakse seni, kuni lahendisse pole enam võimalik ühtegi tippu sisestada, st M on tühi hulk.

Saadud lahendus S ei pruugi olla korrektne – mõned kliendid võivad olla külastamata. Seetõttu käivitame lihtsa sisestusprotseduuri, mis püüab külastamata kliente tellimuse kahanemise järjekorras lahendisse sisestada. Sisestus toimub järgmiselt. Olgu c külastamata klient ja $S = \{T_1, \ldots, T_l\}$ ($T_i = (t_{i1}, \ldots, t_{im_i})$, $i \in \{1, \ldots, l\}$) olgu ülesande ebakorrektne lahend, st ei läbi kõiki kliente, sealhulgas klienti c. Klient c sisestatakse sellise sõidu T_j sellisele positsioonile $k+1$, mille korral sisestamise maksumus $d(t_{jk}, c) + d(c, t_{jk+1})$ ($j \in \{1, \ldots, l\}$, $k \in \{1, \ldots, m_i - 1\}$) on vähim.

Lõpuks, kui parameetriks olnud tõeväärtusmuutuja *lokaalne_otsing* on tõene (koloonia `ACS-length` korral), parandame saadud lahendit lokaalse otsingu algoritmidega (vt. 5. peatükk).

5 Lokaalne otsing

Käesolevas peatükis käsitleme laialt levinud lahendite parandamise meetodit nimega *lokaalne otsing*. Lokaalne otsing tähendab tegelikult erinevate lokaalse otsingu heuristikute kasutamist sõitude optimeerimiseks. Esmalt tutvustame levinumaid heuristikuid, seejärel pakume välja mõned võtted nende töö kiirendamiseks.

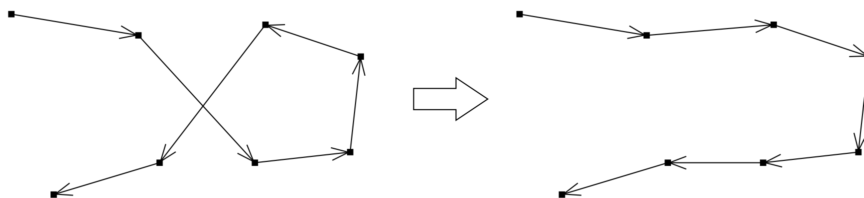
5.1 Heuristikud

Levinumad lokaalse otsingu heuristikud on järgmised (vt. ka [6]).

1. *2-opt*. See heuristik optimeerib korraga ühte sõitu ning toimib järgnevalt (vt. ka joonis 4). Olgu antud sõit $T = (t_1, \ldots, t_m)$. Heuristik 2-opt võtab järjest vaatluse alla kaks selle sõidu serva (t_i, t_{i+1}) ja (t_j, t_{j+1}) ning proovib need asendada vastavalt servadega (t_i, t_j) ja (t_{i+1}, t_{j+1}), saades tulemuseks sõidu $(t_1, \ldots, t_i, t_j, \ldots, t_{i+1}, t_{j+1}, \ldots, t_m)$. Lihtsamalt öeldes muudetakse mingi alamsõidu läbimisjärjekord vastupidiseks.

   Heuristiku nimi tuleneb mõistest *2-optimaalne sõit*, mis tähendab sellist sõitu, mille korral eespoolkirjeldatud muudatus ühegi servade paari korral ei vähenda sõidu maksumust.

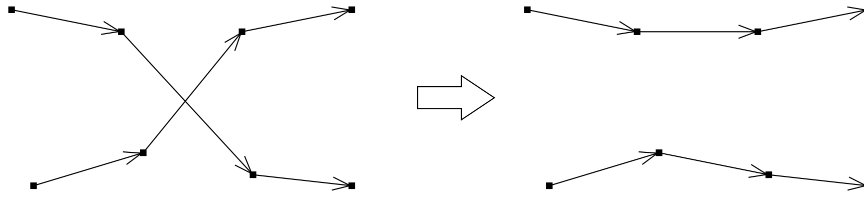
   Heuristik 2-opt on tegelikult erijuht heuristikust *k-opt*, mis vaatleb korraga kahe asemel k serva ning proovib teha kõikvõimalikke muudatusi servade otspunktide vahel. Analoogselt on defineeritud ka mõiste *k-optimaalne sõit*.



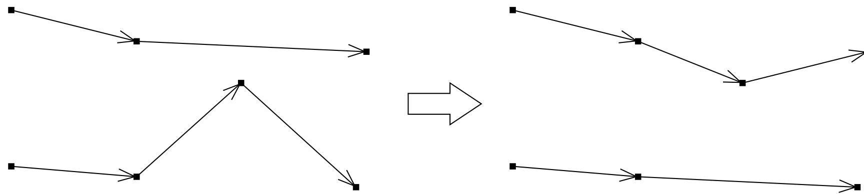
Joonis 4: Heuristik 2-opt

2. *Rist*. See heuristik vaatleb optimeerimisel korraga kahte sõitu, püüdes omavahel ära vahetada nende lõppusid mingist tipust alates (vt joonis 5).

3. *Ümberpaigutus*. See heurustik vaatleb samuti korraga kahte sõitu, püüdes eemaldada ühest mingi tipu ning sisestada selle teise sõitu (vt joonis 6).



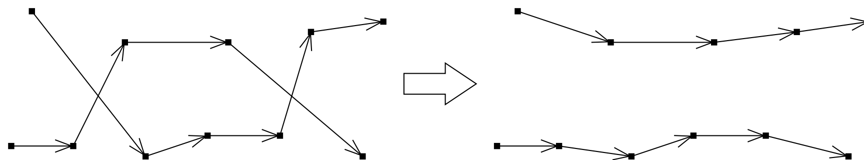
Joonis 5: Heuristik rist



Joonis 6: Heuristik ümberpaigutus

4. *Rist-vahetus*. See heuristik vaatleb samuti korraga kahte sõitu. Kummastki sõidust võetakse vaatluse alla üks alamsõit ning maksumuse vähenemist loodetakse saada mainitud alamsõitude omavahel ära vahetamisega (vt joonis 7).

   Paneme tähele, et heuristikud rist ja ümberpaigutus on tegelikult selle heuristiku erijuhud – esimesel juhul on mõlema sõidu alamteeks vastava sõidu lõpp, teisel juhul on ühe sõidu alamtee tühi ning teise oma üheelemendiline.



Joonis 7: Heuristik rist-vahetus

5.2 Lokaalse otsingu kiirendamine

Ajaakendega transpordiülesande puhul muutub lokaalse otsingu heuristikute rakendamisel omaette aeganõudvaks küsimuseks, kas vaadeldava muudatuse sisseviimisel jääb sõit ülesande kitsenduste suhtes korrektseks. Vastava kontrolli läbiviimine jõumeetodil ühe sõidu kohta on ajalise keerukusega $O(m)$, kus m on sõidu pikkus. Üldjuhul peab läbi vaatama kõik kliendid alates esimesest muu-

detud tipust kuni sõidu lõpuni, sest väljumisajad võivad muutuda hilisemaks, seega pole enam kindel ka muudetud piirkonnast hiljem läbitavate ja otseselt muutmata tippude teenindatavus.

Vaatleme lähemalt, kuidas mõjutab korrektsuse kontroll punktis 5.1 käsitletud heuristikute tööaega.

1. 2-opt. Kuna muudetakse korraga ühte sõitu, tuleb kotrollida üksnes selle korrektsust. Kontrollimiseks kulub aega keskmiselt $O(m)$, kus m on sõidu pikkus.

2. Rist. Siin muudetakse korraga kahte sõitu. Kontrollida tuleb mõlemaid, seega kulub kontrolliks aega keskmiselt $O(m^2)$.

3. Ümberpaigutus. See heuristik muudab samuti korraga kahte sõitu, aga kuna ühest sõidust ainult eemaldatakse üks tipp, siis selle puhul ei pea kitsenduste täidetust kontrollima. Seega aega kulub keskmiselt $O(m)$.

4. Rist-vahetus. Muudetakse korraga kahte sõitu ning kontrollima peab mõlemaid, seega aega kulub keskmiselt $O(m^2)$.

Peamise aja kontrollimisest võtab enda alla ajakitsenduse kontroll. Selle jaoks on vaja sõidu muudetud kohast alates välja arvutada varaseimad väljumisajad uues sõidus ning võrrelda neid vastava kliendi ajaakna lõpuga. Vastavad võrrandid (2) pole aga sagedaseks arvutamiseks kuigi lihtsad. Mahupiirangu kontrolliks seevastu piisab ühest liitmistehtest tipu kohta, et teada saada uue sõidu kogutellimus.

Ajakitsenduse kontrolli saab õnneks väga edukalt kiirendada, võttes lisaks varaseimatele kasutusele ka hiliseimad väljumisajad. Kiirendamise mõte seisneb nende väärtuste eelnevas väljaarvutamises. Ehkki iga muudatuse korral, mis tehakse sõiduga, tuleb antud väärtused uuesti korrektselt välja arvutada, on kitsendustele mittevastavate sõitude arv suhteliselt suur. Seega sõite muudetakse reaalselt harva, mis tähendab, et ümberarvutamist on samuti vähe ning kirjeldatud teguviis omab kokkuvõttes efekti.

Vaatleme näite põhjal, kuidas kasutada varaseimaid ja hiliseimaid väljumisaegu heuristiku *cross* töös. Olgu meil vaatluse all kaks sõitu $T_1 = (t_1, \ldots, t_m)$ ja $T_2 = (t'_1, \ldots, t'_{m'})$. Lahendi parandamiseks proovime ära vahetada sõidu T_1 lõpu alates indeksist $i > 1$ sõidu T_2 lõpuga alates indeksist $j > 1$. Tulemuseks on sõidud $T'_1 = \left(t_1, \ldots, t_{i-1}, t'_j, \ldots, t'_{m'}\right)$ ning $T'_2 = \left(t'_1, \ldots, t'_{j-1}, t_i, \ldots, t_m\right)$. Kirjeldatud muuatus on kasulik juhul kui ära kaduvate servade maksumus on suurem

kui uute servade maksumus:

$$d\left(t_{i-1}, t'_j\right) + d\left(t'_{j-1}, t_i\right) < d\left(t_{i-1}, t_i\right) + d\left(t'_{j-1}, t'_j\right).$$

Eeldades, et kumbki sõit T_1 ja T_2 on enne vaadeldavat muudatust korrektsed, piisab korrektsuseks ajatingimuse suhtes järgnevatest tingimustest:

$$\begin{aligned} ed\left(t_{i-1}\right) + t\left(t_{i-1}, t'_j\right) + s\left(t'_j\right) &\leq ld\left(t'_j\right), \\ ed\left(t'_{j-1}\right) + t\left(t'_{j-1}, t_i\right) + s\left(t_i\right) &\leq ld\left(t_i\right). \end{aligned}$$

Esimene tingimus kontrollib sõitu T'_1 ning teine sõitu T'_2.

Käsitleme nüüd veel varaseimate ja hiliseimate väljumisaegade ümberarvutamist antud näite puhul sõidu $T'_1 = \left(t_1, \ldots, t_{i-1}, t'_j, \ldots, t'_{m'}\right)$ jaoks. Selleks vaatame lähemalt rekurrentsete võrrandite (2) ja (3) struktuuri. Varaseimaid väljumisaegu arvutatakse välja tippude indeksite kasvamise järjekorras, kusjuures suuruse $ed\left(t_k\right)$ väärtus sõltub ainult suurustest $ed\left(t_1\right), \ldots, ed\left(t_{k-1}\right)$ ning tipust t_k endast. Seega, kuna sõitude T_1 ja T'_1 algused kuni tipuni indeksiga $i-1$ langevad kokku, on ka vastavad suurused $ed\left(t_k\right)$ ($k \in \{1, \ldots, i-1\}$) võrdsed. Seetõttu me võime varaseimate väljumisaegade arvutamist sõidu T'_1 jaoks alustada indeksist i ehk tipust t'_j, võttes

$$ed\left(t'_j\right) = \max\left\{ed\left(t_{i-1}\right) + t\left(t_{i-1}, t'_j\right), b\left(t'_j\right)\right\} + s\left(t'_j\right).$$

Analoogiliselt näeme, et hiliseimaid väljumisaegu arvutatakse indeksite kahanemise järjekorras ning tipu t_k hiliseim väljumisaeg sõltub ainult järgnevate tippude hiliseimast väljumisajast ning tipust t_k. Kuna sõitude T_2 ja T'_2 lõpud langevad kokku, langevad ka vastavad hiliseimad väljumisajad $ed\left(t'_k\right)$ ($k \in \{j, \ldots, m'\}$) kokku ning me võime arvutamist alustada alates sõidu T'_1 tipust t_{i-1}, võttes

$$ld\left(t_{i-1}\right) = \min\left\{ld\left(t'_j\right) - s\left(t'_j\right) - t\left(t_{i-1}, t'_j\right), e\left(t_{i-1}\right)\right\}.$$

5.3 Rist-vahetus heuristiku kiirendamine

Vaatleme lähemalt rist-vahetus heuristiku tööd. Olgu antud kaks sõitu $T_1 = (t_1, \ldots, t_m)$ ja $T_2 = (t'_1, \ldots, t'_{m'})$. Heuristik püüab omavahel ära vahetada nende sõitude mingeid alamsõite $(t_{i+1}, \ldots, t_j)$ ja $\left(t'_{i'+1}, \ldots, t'_{j'}\right)$, kus $i, j \in \{1, \ldots, m-1\}$ ($i < j$) ja $i', j' \in \{1, \ldots, m'-1\}$ ($i' < j'$). Muudatuse tulemuseks on sõidud T'_1

ja T'_2, kus

$$\begin{aligned} T'_1 &= \left(t_1, \ldots, t_i, t'_{i'+1}, \ldots, t'_{j'}, t_{j+1}, \ldots, t_m\right), \\ T'_2 &= \left(t'_1, \ldots, t'_{i'}, t_{i+1}, \ldots, t_j, t'_{j'+1}, \ldots, t'_{m'}\right). \end{aligned}$$

Vahetus viiakse ellu, kui see on kasulik, ehk

$$\begin{aligned} & d\left(t_i, t'_{i'+1}\right) + d\left(t'_{j'}, t_{j+1}\right) + d\left(t'_{i'}, t_{i+1}\right) + d\left(t_j, t'_{j'+1}\right) < \\ & < d\left(t_i, t_{i+1}\right) + d\left(t_j, t_{j+1}\right) + d\left(t'_{i'}, t'_{i'+1}\right) + d\left(t'_{j'}, t'_{j'+1}\right) \end{aligned}$$

ning korrektne. Korrektsuse kontrolliga tegelesime eelmises punktis. Olgu korrektsuse kontrolli ning muudatuse sooritamise ajaline keerukus $O\left(f\left(m\right)\right)$. Arvestades, et iga sõidu paari korral valitakse sõltumatult 4 indeksit ja võttes $m \approx m'$, saame rist-vahetus heuristiku ligikaudseks ajaliseks keerukuseks iga sõidu paari korral $O\left(m^4 f\left(m\right)\right)$. Sealjuures ellu viiakse väga väike osa vaadeldavatest muudatustest.

Rist-vahetus heuristikut saab muuta märgatavalt kiiremaks, kui asendada vahetuse kasulikkuse kontroll järgneva kaheosalise kontrolliga:

1. servade $\left(t_i, t_{i+1}\right)$ ja $\left(t'_{i'}, t'_{i'+1}\right)$ jaoks:

$$d\left(t_i, t'_{i'+1}\right) + d\left(t'_{i'}, t_{i+1}\right) < d\left(t_i, t_{i+1}\right) + d\left(t'_{i'}, t'_{i'+1}\right),$$

2. servade $\left(t_j, t_{j+1}\right)$ ja $\left(t'_{j'}, t'_{j'+1}\right)$ jaoks:

$$d\left(t_j, t'_{j'+1}\right) + d\left(t'_{j'}, t_{j+1}\right) < d\left(t_j, t_{j+1}\right) + d\left(t'_{j'}, t'_{j'+1}\right).$$

Näeme, et esimest kontrolli on võimalik teostada ilma indeksiteta j ja j'. Koostame algoritmi selliselt, et esmalt antakse väärtused indeksitele i ja i' ning seejärel teostame kontrolli 1. Indeksite j ja j' väärtustamiseks peab kontroll andma positiivse tulemuse, vastasel korral katkestame tsükli ja võtame vaatluse alla uued i ja i'. Arvestades, et suvaliste sõitude ja indeksite i ja i' korral tingimus 1 valdavalt ebaõnnestub (vt. näidet joonisel 7), pääseme nendel juhtudel $O\left(m^2\right)$ keerukusest, mis tuleneb indeksite j ja j' väärtustamisest.

Siinkohal tuleb märkida, et esitatud põhimõttel koostatud algoritm ei ole ilmselt samaväärne klassikalise rist-vahetus heuristiku algoritmiga. Autori katsetused on näidanud, et muudetud algoritm optimeerib traditsioonilisest mõnevõrra vähem, aga on see-eest palju kiirem.

6 Praktiline realisatsioon

Käesolev töö sai tegelikult alguse Tartu Ülikooli Tehnoloogiainstituudi ning AS Regio vahelisest koostööprojektist. Projekti käigus tuli töö autoril luua transpordiülesannet lahendava tarkvara *Logisme* põhifunktsionaalsust realiseeriv töötav prototüüp. Mainitud prototüübis on realiseeritud algoritm MACS-VRPTW ning lokaalse optimieerimise heuristik 2-opt. Teised töös esitatud osad on realiseeritud peale projekti lõppu.

Punktis 6.1 tutvustame projekti tulemuseks olnud programmi neid osi, mis pole otseselt seotud transpordiülesande lahendamisega, kuid tuli projekti käigus realiseerida. Peamiselt olid need seotud ülesande lähteandmetega. Punktis 6.2 on toodud projekti käigus realiseeritud programmi võrdlused ühe olemasoleva tarkvaraga.

6.1 Transpordiülesande lähteandmete ettevalmistamine

Nagu mainitud punktis 2.2.1, eeldab transpordiülesanne lähteandmetena täisgraafi, mille iga tipu (kliendi või depoo) paariga on seotud kaugus ehk maksumus liikumaks ühest tipust teise.

Projektis fikseeritud lähteandmed olid aga järgnevad.

- Teede graaf, mis oli esitatud mõne- kuni mõnekümnemeetriste teelõikude koordinaatidega. Iga teelõiguga oli seotud selle läbimise keskmine kiirus.
- Klientide ja depoo andmed: koordinaadid, tellimus, ajaaken ja teenindusaeg.
- Optimeeritav suurus: teepikkus, sõiduaeg või rahaline maksumus vastavalt kulule kilomeetri ja tunni kohta.

Eelnev tähendab, et klientide ja depoo vaheliste kauguste leidmine ehk kauguste maatriksi arvutamine oli üks osa ülesandest. Lähteandmete ettevalmistamine koosnes seega järgmistest sammudest.

- Teedegraafi sisselugemine. Iga teelõigu kohta loodi vajadusel uued graafi tipud tähistamaks teelõigu otspunkte ning antud tippude vahele loodi serv. Servaga oli seotud keskmine kiirus.
- Klientide ja depoo sisselugemine.

- Klientide lisamine teedegraafile. See oli omaette samm sellepärast, et täpselt kliendi koordinaatideni ei pruukinud ühtegi teed minna. Seetõttu tähendas klientide lisamine lisaservade tekitamist, mis ühendasid kliente teedevõrgustikuga. Ehkki algandmeteks olnud teelõigud olid üldiselt lühikesed, oli nõue, et lisaserv tuleb tekitada mitte kliendist lähima teedegraafi tipuni (teelõigu otspunktini) vaid kliendist lähima teedegraafi servani (suvalise teelõigu punktini, erijuhul otspunktini). See tähendab, et kliendi lisamiseks tuli leida lühim ristsirge mingist teedegraafi servast antud kliendini, see serv vajadusel poolitada ning luua servad kliendi ja lähima serva punkti vahel.

- Teedegraafi mittehargnevate järjestikuste servade ühendamine ühtseks servaks. Osutus, et järgmist sammu (kauguste maatriksi arvutamist) saab muuta oluliselt kiiremaks, kui eelnevalt asendada mittehargnevatele teedele vastavad järjestikused servad ühe servaga. Paneme tähele, et antud sammu tohtis sooritada alles pärast klientide lisamist teedegraafile, sest kliendid tuli ühendada reaalsete teedega.

- Tippudevaheliste kauguste (kaugustemaatriksi) ning sõiduaegade arvutamine. Nagu mainitud punktis 2.2.1, väljendab tippude x ja y vaheline kaugus $d(x, y)$ tipust x algava ja tippu y jõudva odavaima tee pikkust. Seega tuli antud sammus kasutada graafis lühimate teede leidmise algoritme. Täpsemalt kasutati Dijkstra algoritmi (vt. [5]). Lühima tee algoritmis kasutati serva pikkusi vastavalt ülesande algandmete hulgas määratud optimeeritavale suurusele: teepikkus, sõiduaeg või rahaline maksumus. Tippude x ja y vaheliseks sõiduajaks $t(x, y)$ tuli sõiduaja definitsiooni (vt. punkt 2.2.1) kohalselt võtta odavaima tee läbimise aeg.

6.2 Võrdlus olemasoleva tarkvaraga

Käesolevas peatükis vaadeldava koostööprojekti raames teostati ka võrdlus Logisme ja ühe olemasoleva transpordiülesannet lahendava tarkvaraga RouteView Pro[1] ver. 1.2. Viimane on mõeldud ilma ajaakendeta ülesande lahendamiseks, seetõttu kasutati võrdlemisel lähteandmeid, milles klientide ajaaknad olid fiktiivsed. Erinevus seisnes veel selles, et RouteView paigutab ülesande kliendid lähimasse teelõigu otspunkti, seevastu Logisme lisab serva kliendist lähima

[1]http://www.tetrad.com/mapinfo/usa/miroute.html

teelõiguni. Seetõttu on võrdluses toodavad numbrid võrreldavad ainult ligikaudselt.

Võrdlus on toodud tabelis 1. Testides on aluseks reaalne teede graaf ning kliendid on mingi hulgifirma reaalsed kliendid. Ülesandes klientide arvuga 99 on tegemist ühe linna piirkonda kuuluvate klientidega, teistes ülesannetes paiknevad kliendid laiali erinevates linnades ning maa piirkondades. Programmi Logisme testiti iga ülesandega 4 korda, kuna enamasti saadakse iga käivitamisega uus tulemus. Tabelis toodud lahendusajad ja teepikkused on seetõttu keskmised väärtused. Programm RouteView Pro aga annab iga kord sama lahendi ligikaudu sama ajaga. Programmi Logisme tööajaks vaadeldavate testide korral on keskmine aeg alates ülesande lähteandmete sisselugemise algusest kuni ajani, mil leiti kokkuvõttes parim lahend meetodi MACS-VRPTW poolt. Programmi RouteView Pro korral on lahendusajaks puhas transpordiülesande lahendamise aeg ilma algandmete sisselugemiseta. Testide tulemustest on näha, et üldjuhul töötab programm Logisme kiiremini, aga annab natuke halvemaid tulemusi.

	Lahendusaeg (sek)		Teepikkus (km)	
Klientide arv	Logisme	RouteView Pro	Logisme	RouteView Pro
20	20	30	445,07	442,31
61	38	93	401,88	394,48
99	87	19	61,76	59,84
140	151	354	559,73	556,21
250	221	1251	587,94	574,76

Tabel 1: Logisme ja RouteView Pro vaheline võrdlus

7 Kokkuvõte

Käesoleva töö uurimisobjektiks on ajaakendega transpordiülesanne, mille võib kokku võtta järgnevalt. Hulk kliente on tarvis teenindada keskses depoos paiknevate sõidukite poolt. Klientidel on kindlaksmääratud tellitud kauba kogus, teenindamiseks kuluv aeg ja ajavahemik, millal neid saab külastada, sõidukitel on piirmahtuvus, millest rohkem kaupa nad ei saa vedada. Eesmärk on leida võimalikult odavad teekonnad läbi kõigi klientide, arvestades mainitud piiranguid.

Ülesande lahendamine on käesolevas töös jagatud kolmeks sammuks: lähteandmete eeltöötlus, lahendamine ja lahendi järeltöötlus. Eeltöötluse eesmärk on muuta etteantud transpordiülesanne lahendusmeetodi jaoks lihtsamaks, piiramata sealjuures lahendusmeetodi tööd. Lahendi järeltöötluse all mõeldakse peamiselt lahendi parandamist erinevate heuristikute abil.

Lähteandmete eeltöötlusena vaadeldakse käesolevas töös klientide lahendamise-eelset klasterdamist: üksteisele lähedalasuvad kliendid ühendatakse ühtseks tervikuks, mis teenindatakse korraga. Selle võttega vähendatakse korraga lahendatava ülesande lähteandmete hulka. Osutub, et ajaakendega transpordiülesande puhul pole klientidest klastrite moodustamine sugugi lihtne ülesanne, kuna klastrid peavad ülesande lahendusmeetodile paistma "harilike" klientidena, st omama samu atribuute nagu üksikud kliendid, sh külastamise ajavahemikud ja teenindamiseks kuluvad ajad. Atribuutide väärtuste valikul tuleb silmas pidada, et peale klasterdamist saadud ülesande suvalise lahendi raames peab olema võimalik kõik klasterdatud kliendid korrektselt läbida. Käesoleva töö panus on klasterdamise kui lahendusmeetodi jaoks läbipaistva lähteandmete eeltöötluse rakendamine ajaakendega transpordiülesandele. Pakutakse välja metoodika klientidest klastrite moodustamiseks ning hilisemaks läbimiseks, sh tutvustatakse ühte võimalust klastrite atribuutide väärtuste leidmiseks. Viimase korrektsuse kohta tõestatakse vastavasisuline teoreem. Töös jäetakse lahtiseks küsimused, kuidas arvestada klastrite loomisel ühendatavate klientide ajaakende kattuvust ning mil määral on mingi konkreetse ülesande puhul kasulik kliente klasterdada.

Lahendusmeetodina tutvustatakse töös sipelgakolooniate modelleerimisel põhinevaid algoritme (vt. [2, 3, 4]).

Lahendite järeltöötlusena käsitletakse levinud lahendi optimeerimise meetodit *lokaalne otsing*. Lokaalse otsingu üldiseks puuduseks on selle aeglus, kuna optimeerimiseks tuleb läbi vaadata väga palju erinevaid muudatusi, millest vähesed on kasulikud ning ülesande piirangute suhtes korrektsed. Käesolevas töös pakutakse välja metoodikaid optimeerimise kiirendamiseks.

Solving Vehicle Routing Problem with Time Windows

Jaanus Jaeger, MSc thesis

Abstract

The subject of this thesis is vehicle routing problem with time windows which can be stated as follows. A fleet of vehicles located at a central depot must provide some type of service to a set of customers. The customers are assigned an amount of ordered goods, the duration of the service and the time interval during which they can be visited, the vehicles are assigned maximum capacity. The objective is to find a set of tours through all the customers so that the total cost is minimized and none of the above constraints is violated.

The general solution method used in this thesis consists of three steps: data preprocessing, solving and solution postprocessing. The purpuse of the preprocessing is to simplify the problem for the solution method without limiting the work of the solution method. Solution postprocessing means optimizing existing solutions by several heuristics.

As for data preprocessing, the thesis considers customer clustering: customers close to each-other are combined to clusters that are served all at once. By this we reduce the size of the problem to be solved at once. It appears that in the case of vehicle routing problem with time windows the forming of clusters of customers is not at all a simple matter, since the formed clusters must look like ordinary customers to the solution method, i.e. a cluster must have the same attributes (time intervals for visiting, duration of the service) as a single customer. The values of the attributes have to be such that it must be possible to visit all the clustered customers in the bounds of any solution of the problem that is the result of clustering. The thesis proposes clustering as a preprocessing step for vehicle routing problem with time windows transparent to a solution method. Methods are provided to cluster customers and to visit them afterwards. For the correctness of the cluster atributes a respectiv theorem is proved.

As for solution method, the thesis describes well-known algorithms based on ant colonies (see [2, 3, 4]).

As for solution postprocessing, local search as solution optimization is considered. The general flaw of local search heuristics is their slowness, since large

sets of possible changes are to be considered, only few of which are profitable and constraint-feasible. The thesis proposes methods to make optimization much faster.

Indeks

$E(S)$, 11
$E(T)$, 11
$T^k(j)$, 31
$V(S)$, 11
$V(T)$, 11
$W(S)$, 11
$d(x)$, 9
$d(x, y)$, 8
$\eta(x_i, x_j)$, 31
$\tau(x_i, x_j)$, 31
τ_0, 31
$[b(P), e(P)]$, 9
$[b(x), e(x)]$, 9
$ed(t_i)$, 12
$ld(t_i)$, 12
$s(x)$, 9
$t(x, y)$, 8
$v(P)$, 9
ümberpaigutus, 38
2-opt, 38

ACS, 29
ACS-length, 35
ACS-vehicle, 34
ajaaken, 9
ajaakendega transpordiülesanne, 8
ajakitsendus, 10
algusdepoo, 25

cluster first route second, 15

depoo, 8

eksploratsioon, 32
ekspluatatsioon, 32

feromoon, 29, 31
feromoonirada, 29

globaalne feromooni uuendamine, 33

hierarhiline klasterdamine, 16
hiliseim väljumisaeg, 12
homogeense autopargiga transpordiülesanne, 7

kaugus, 8
klaster, 15
klastri avamine, 17, 24
klastri avamise ülesanne, 25
klastriülesande depoo, 23
klastriülesanne, 18
klastritevahelised kaugused, 23
klient, 8
klientide hulk, 8

lähima naabri algoritm, 31
lõpetamistingumus, 33
lõppdepoo, 25
lahend, 11
lokaalne feromooni uuendamine, 32

MACS-VRPTW, 29
mahupiirang, 10
metaklient, 16
mitme sipelgakoloonia süsteem ajaakendega transpordiülesande lahendamiseks, 29, 33
mittehomogeense autopargiga transpordiülesanne, 7

nähtavus, 31

rändkaupmeheülesanne, 6
rist, 38
rist-vahetus, 39

sõidu pikkus, 10
sõiduaeg, 8
sõiduki mahtuvus, 9
sõit, 10
sipelgakoloonia süsteem, 29, 31

teenindusaeg, 9
tehissipelgas, 29
tellimus, 9
transpordiülesanne, 6

varaseim väljumisaeg, 12

Viited

[1] J. Bramel and D. Simchi-Levi. A location based heuristic for general routing problems. *Operations Research*, 43(4):649–660, 1995.

[2] Marco Dorigo and Luca Maria Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[3] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[4] Luca Maria Gambardella, Éric Taillard, and Giovanni Agazzi. MACS-VRPTW: A Multiple Ant Colony System For Vehicle Routing Problems With Time Windows. http://citeseer.ist.psu.edu/450222.html.

[5] Jüri Kiho. *Algoritmid ja andmestruktuurid.* Tartu Ülikool, 1997.

[6] P. Kilby, P. Prosser, and P. Shaw. Guided local search for the vehicle routing problem. In Proceedings of the 2nd International Conference on Metaheuristics, 1997. http://citeseer.ist.psu.edu/kilby97guided.html.

[7] Meelis Kull. Fast Clustering in Metric Spaces. Master's thesis, Tartu Ülikool, 2004.

A Klasterdamise algoritmid

Algoritm 3 Transpordiülesande klientide klasterdamine

```
function klasterda(X, d(x,y)) begin
  /* X = {c_1, ..., c_n} on klientide hulk, d(x,y) on hinnangufunktsioon. */

  /* Defineerime järgmised muutujad:
     – sümmeetriline kauguste maatriks D = (d_ij), kus kliendi kaugus iseendast
       on ∞,
     – klastrite järjend C = (C_1, ..., C_n), mis koosneb hetkel ühendamata klast-
       ritest ning mille i-ndale elemendile vastab kaugustemaatriksi i-s rida. */
  for i = 1, ..., n begin
    for j = 1, ..., n begin
      d_ij := d(c_i, c_j) + d(c_j, c_i)
    end
    d_ii := ∞
    C_i := {c_i}
  end

  repeat
    /* Leiame lähima klastrite paari C_i' ja C_j'. */
    (i', j') := lähim(D, C)                /* Vt. algoritm 5. */
    uusC := ühenda(C_i', C_j', d(x,y))     /* Vt. algoritm 4. */
    if uusC ≠ ∅ begin
      C_i' := ∅ /* Klastrid C_i' ja C_j' pole enam ühendamata klastrid. */
      C_j' := ∅
      ī := klastri uusC depooks oleva kliendi indeks järjendis X.
      C_ī := uusC /* Uus klaster on hetkel ühendamata. */
    else
      d_i'j' := ∞ /* Selleks, et C_i' ja C_j' ei leitaks enam lähimate klastritena. */
    end
  until lõpetamistingimus /* Siin otsustatakse, kui palju klasterdada. */

  /* Koostame ning tagastame allesjäänud ühendamata klastrite hulga. */
  C̄ := {}
  for i = 1, ..., n begin
    if C_i ≠ ∅ begin
      C̄ := C̄ ⋃ {C_i}
    end
  end
  return C̄
end function klasterda
```

Algoritm 4 Klastrite ühendamine

```
function ühenda(C_1, C_2, d(x,y)) begin
  /* Kasutame globaalset muutujat P, mis on esialgne transpordiülesanne
     depooga d, sõidukite mahtuvusega v(P) ja ajaaknaga b(P) – e(P). */

  /* Kui klastrite C_1 ja C_2 ajaaknad ei kattu piisavalt, siis neid ei ühendata. */
  if klastrite C_1 ja C_2 ajaaknad ei kattu piisavalt begin
    return ∅
  end

  /* Konstrueerime klastriülesande P_C.*/
  d_C := depoo(C_1 ⋃ C_2, d(x,y))        /* Vt. algoritm 6. */
  b(P_C) := max{max{b(P), b(d)} + s(d) + t(d, d_C), b(d_C)}
  e(P_C) := min{min{e(P), e(d)} − s(d) − t(d_C, d), e(d_C)}
  P_C := transpordiülesanne depooga d_C ja klientidega C_1 ⋃ C_2 \ d_C

  /* Lahendame klastriülesande P_C kasutades suvalist lahendusmeetodit. Lahen-
     diks on T = (t_1, ..., t_m). */
  T := lahenda(P_C)

  if T pole korrektne lahend begin
    return ∅
  end

  /* Koostame uue klastri C. Esmalt arvutame C atribuudid. */
  d(C) := 0
  foreach c ∈ C_1 ⋃ C_2 begin
    d(C) := d(C) + d(c)
  end
  if |C| = 1 begin
    s(C) := s(d_C)
  else
    s(C) := ed(t_m) − ed(t_1)
  end
  b(C) := ed(t_1) − s(d_C)
  e(C) := ld(t_1) − s(d_C) + s(C)
  C := klaster depooga d_C ja tippudega C_1 ⋃ C_2

  return C
end function ühenda
```

Algoritm 5 Lähima klastrite paari leidmine

```
function lähim(D = (d_ij), C = (C_i)) begin
  min := ∞, i' := 1, j' := 1
  for i = 1, ..., n begin
    for j = 1, ..., n begin
      /* Otsime lähimaid ainult seni ühendamata klastrite hulgast. */
      if C_i ≠ ∅ ∧ C_j ≠ ∅ ∧ d_ij < min begin
        min := d_ij, i' := i, j' := j
      end
    end
  end
  return (i', j')
end function lähim
```

Algoritm 6 Klastruülesande depoo leidmine

```
function depoo(X, d(x, y)) begin
  /* Muutuja X on suvaline klientide hulk ja d(x, y) on hinnangufunktsioon. */

  d := NULL
  min_reasumma := ∞

  foreach c ∈ X begin
    reasumma := 0
    foreach c' ∈ X begin
      reasumma := reasumma + d(c, c') + d(c', c)
    end
    if reasumma < min_reasumma begin
      d := c
      min_reasumma := reasumma
    end
  end

  return d
end function depoo
```

B ACS ja MACS-VRPTW algoritmid

Algoritm 7 ACS-TSP – rändkaupmehe ülesande lahendamine.

```
function ACS-TSP(G = (X, E)) begin
    /* Algväärtustamine. */
    T^P := ∅ /* Parim lahend, alguses pole määratud. */
    foreach (x, y) ∈ E begin
        τ(x, y) := τ_0 /* Algväärtustame feromooni. */
    end

    /* Konstrueerime lahendeid kuni lõpetamistingimuseni. */
    repeat
        /* Paigutame sipelgad suvalistesse linnadesse. */
        for k := 1, ..., m begin
            T^k := () /* Algväärtustame konstrueeritavad teed. */
            T^k(1) := juhuslik linn
        end

        /* Läbime kõik linnad, jõudes tagasi alguslinna. */
        for j = 2, ..., n + 1 begin
            /* Liigume iga sipelgaga järgmisesse linna. */
            for k := 1, ..., m begin
                x := T^k(j − 1) /* k-nda sipelga hetkeasukoht. */
                if j ≤ n begin
                    y := järgmine läbitav linn tõenäosuslikult, kas
                        ekspluatatsiooni või eksploratsiooni alusel
                else
                    y := T^k(1) /* Liigume tagasi alguslinna. */
                end
                T^k(j) := y /* Lisame y läbitud linnade järjendisse. */
                /* Lokaalne feromooni uuendamine (10). */
                τ(x, y) := (1 − p) τ(x, y) + pτ_0
            end
        end

        T^P := siiani parim lahend
        /* Globaalne feromooni uuendamine (11). */
        foreach (x, y) ∈ E(T^P) begin
            τ(x, y) := (1 − p) τ(x, y) + p/W(T^P)
        end
    until lõpetamistingimus /* valem (12). */

    return T^P
end
```

Algoritm 8 MACS-VRPTW – juhtprotseduur

```
procedure MACS-VRPTW(P) begin
    /* Leidku funktsioon sõitude_arv argumendikst antud
        lahendi sõitude arvu. */
    /* Algväärtustame parima lahendi. */
    S^gl :=lubatav lahend lähima naabri algoritmiga

    repeat
        v :=sõitude_arv(S^gl)
        /* Käivitame protseduurid. */
        ACS-vehicle(P,v − 1) /* Vt. algoritm 10. */
        ACS-length(P,v)      /* Vt. algoritm 9. */
        while ACS-vehilce ja ACS-length on aktiivsed begin
            S^gl :=parim lahend kas ACS-vehicle või ACS-length poolt
            if sõitude_arv(S^gl)< v begin
                peatame protseduurid ACS-vehicle ja ACS-length
            end
        end
    until lõpetamistingimus /* valem (12). */
end
```

Algoritm 9 ACS-length – maksumust minimeeriv koloonia

```
procedure ACS-length(P,v) begin
    Initsialiseerime graafi ning feromooni suuruse v alusel

    repeat
        /* Genereerime lahendused. */
        for k := 1, . . . , m begin
            S^k :=new_active_ant(P,TRUE,0) /* Vt. algoritm 11. */
            if S^k on korrektne lahend ja W(S^k) < W(S^gl) begin
                saada S^k protseduurile MACS-VRPTW
            end
        end
        /* Globaalne feromooni uuendamine (11). */
        foreach (x, y) ∈ E(S^gl) begin
            τ(x, y) := (1 − p) τ(x, y) + p/W(S^gl)
        end
    until lõpetamistingimus /* valem (12). */
end
```

Algoritm 10 `ACS-vehicle` – sõitude arvu minimeeriv koloonia

```
procedure ACS-vehicle(P,v) begin
    /* Leidku funktsioon kliente külastatud klientide arvu
        argumendiks olevas lahendis. */
    Initsialiseerime graafi ning feromooni suuruse v alusel
    S^vehicle :=lahend lähima naabri algoritmiga, mis kasutab v sõitu
    IN := (0,...,0)

    repeat
        /* Genereerime lahendused. */
        for k := 1,...,m begin
            S^k :=new_active_ant(P,FALSE,IN) /* Vt. algoritm 11. */
            foreach x ∉ V(S^k) begin
                IN(x) := IN(x) + 1
            end
            if kliente(S^k)>kliente(S^vehicle) begin
                S^vehicle := S^k
                IN := (0,...,0)
                if S^vehicle külastab kõiki kliente begin
                    saada S^vehicle protseduurile MACS-VRPTW
                end
            end
        end
        /* Globaalne feromooni uuendamine (11). */
        foreach (x,y) ∈ E(S^gl) begin
            τ(x,y) := (1 − p) τ(x,y) + p/W(S^gl)
        end
        foreach (x,y) ∈ E(S^vehicle) begin
            τ(x,y) := (1 − p) τ(x,y) + p/W(S^vehicle)
        end
    until lõpetamistingimus /* valem (12). */
end
```

Algoritm 11 `new_active_ant` – lahendeid konstrueeriv protseduur

```
function new_active_ant(P, lokaalne_otsing, IN) begin
    /* Teed alustame juhuslikust depoo duplikaadist. */
    x := juhuslik depoo duplikaat
    S := (x)
    aeg := max{b(P), b(x)} + s(x)
    last := d(x)

    repeat
        /* Leiame lubatavad tipud lähtudes tipust x. */
        M := lubatavad tipud
        foreach y ∈ M begin
            /* Arvutame nähtavused. */
            uus_ed_y := max{aeg + t(x, y), b(y)} + s(y)
            vahe := uus_ed_y − aeg
            kaugus := vahe · (e(y) − aeg)
            kaugus := max{1, kaugus − IN(y)}
            η(x, y) := 1/kaugus
        end
        y := järgmine läbitav linn tõenäosuslikult, kas
            eksplutatsiooni või eksploratsiooni alusel
        lisame tipu y lahendi S lõppu
        aeg := uus_ed_y
        last := last + d(y)
        if y on depoo begin
            aeg := max{b(P), b(y)} + s(y)
            last := d(y)
        end
        /* Lokaalne feromooni uuendamine (10). */
        τ(x, y) := (1 − p) τ(x, y) + pτ_0
        x := y
    until M = ∅

    /* Sisestusprotseduur. */
    sisestusprotseduur(S)

    /* Lokaalne otsing lahendi parandamiseks. */
    if lokaalne_otsing = TRUE ⋀ S on korrektne lahend then
        lokaalse_otsingu_protseduur(S)
    fi

    return S
end
```