VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
KOMPIUTERIJOS KATEDRA

Mokslo tiriamasis darbas I

OPTIMALIŲ MARŠRUTŲ PAIEŠKOS ALGORITMAI

Atliko: magistro 1 kurso, 10 grupės studentas
Karolis Šarapnickis

Darbo vadovas:
doc. Tadas Meškauskas

Vilnius
2014

Turinys

1. Įvadas

Dauguma žmonių, kiekvieną dieną susiduria su optimaliausio maršruto paieškos problema. Nepriklausomai nuo to, kur planuojama nuvykti (į darbą, parduotuvę, universitetą) ar kokią transporto priemonę pasirinkti (dviratį, mašiną, autobusą, ar išvis jokios), dažniausiai siekiama rasti patį trumpiausią maršrutą, kuris padėtų sutaupyti kiek įmanoma daugiau brangaus laiko.

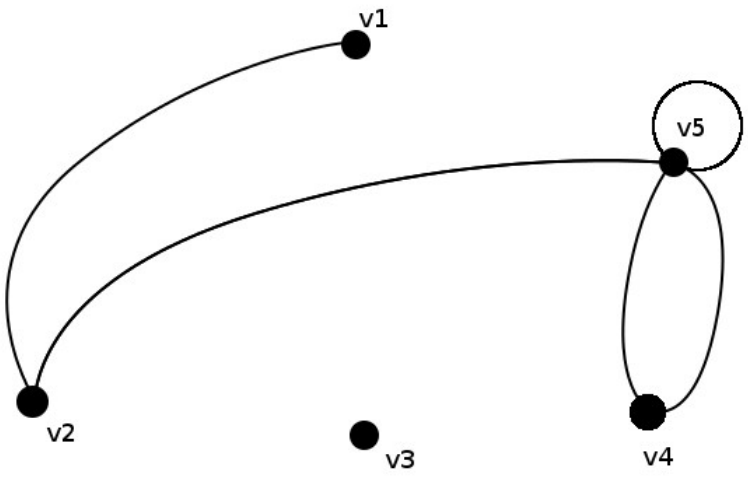
Optimalių maršrutų paieška aktuali ne tik kasdieniniame gyvenime, bet ir versle. Logistikoje apskaičiuojant trumpiausius maršrutus tolimųjų reisų vairuotojams bei kurjeriams, reikia atsižvelgti ne vien į aplankomų miestų koordinates, bet ir į įvairias kintančias aplinkybes (transporto srautą, kelių būklę, oro sąlygas). Taip pat su šia problema susiduria žmonės, kurie sudarinėja autobusų, laivų bei lėktuvų maršrutus. Spręsti tokias kompleksines ir dažnai kintančias problemas reikia efektyviai – būtina rasti ne tik trumpiausią maršrutą, bet ir tai atlikti greitai. Todėl optimaliausio maršruto paieškos atliekamos naudojant tam tikrus algoritmus.

Šio mokslinio darbo užduotis ir yra susipažinti su trumpiausio kelio paieškos algoritmais ir išanalizuoti, kurie algoritmai yra optimalūs, sprendžiant daugelio taškų optimalaus maršruto problemą.

2. Pagrindinės grafų ir algoritmų teorijos sąvokos

2.1. Grafų teorija

Konceptualiai grafą sudaro viršūnės ir kraštai (arba briaunos, jungtys...), jungiantys viršūnes [Ruo08]. Viršūnės vaizduojamos taškais, briaunos – kreivėmis. Digrafo atveju papildomai nurodomos ir briaunų kryptys [Man01]. Grafo pavyzdys:



1. pvz. Grafo pavyzdys.

$$G=(V,E) \tag{1}$$

Formaliai, grafas yra pora rinkinių (V,E), kur V yra viršūnių (angl. *Vertices*) rinkinys, o E – briaunų (angl. *Edges*), susidariusių tarp susijungusių viršūnių rinkinys. Kada vietoj briaunų aibės E imamas briaunų rinkinys (šeima) su galimais pasikartojimais, pora (V,E) yra vadinama multigrafu. Jį vaizduojant plokštumoje, dvi viršūnės jungiamos atitinkamu kiekiu briaunų [Man01]. Grafas dažnai užrašomas 1 formulės pavidalu. Elementai E rinkinyje gali kartotis, o V rinkinyje – ne [Ruo08]. Iš 1 pavyzdinio grafo galima užrašyti tokias viršūnes ir kraštus:

- $V=\{v_1,v_2,v_3,v_4,v_5\}$,

- $E=\{(v_1,v_2),(v_2,v_5),(v_5,v_5),(v_5,v_4),(v_4,v_5)\}$.

Grafų teorijoje dažnai sutinkama terminologija [Ruo08]:

1. Dvi viršūnės *u* ir *v* yra vadinamos briaunos *(u, v)* kraštinėmis viršūnėmis, arba briaunos galais., arba incidenčiomis viršūnėmis.
2. Kraštai turintys vienodas viršūnes vadinami paraleliais.
3. *(u, u)* formos kraštas yra kilpa (anlg. *loop*).
4. Grafas yra paprastas jei jis neturi paralelinių kraštų ir kilpų.
5. Grafas be kraštų (*E* rinkinys yra tuščias) yra tuščias.
6. Grafas be viršūnių (*V* ir *E* rinkiniai yra tušti) yra nulinis (angl. *null*).
7. Grafas su viena viršūne yra trivialus.
8. Kraštai yra gretimi, jei jie turi bendrą galinę viršūnę.
9. Viršūnės *u* ir *v* yra gretimos, jei jos sujungtos briauna, kitaip tariant, jei yra *(u, v)* kraštas.
10. Viršūnės *v* laipsnis, užrašomas *d(v)*, yra skaičius kraštų, turinčių galinę viršūnę *v*. Kilpa atitinka 2 laipsnius, o paraleliniai kraštai skaičiuojami atskirai.
11. Kabanti (angl. *pendant*) viršūnė yra viršūnė turinti 1-ą laipsnį.
12. Kraštas turintis kabančią, galinę viršūnę yra kabantis kraštas.
13. Izoliuota viršūnė yra viršūnė turinti 0-inį laipsnį.

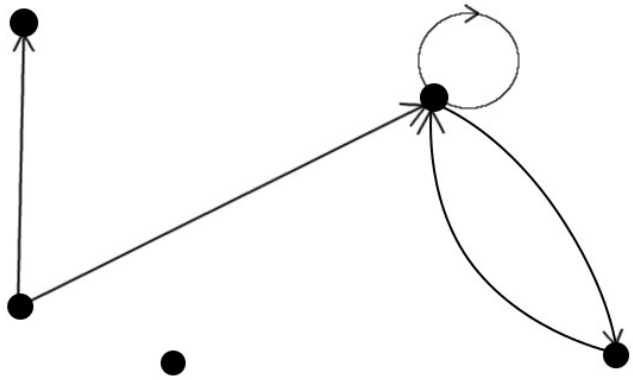
Mažiausias laipsnis grafe *G* yra užrašomas $\delta(G)$ (= 0, jei grafe yra izoliuota viršūnė), o didžiausias laipsnis užrašomas $\Delta(G)$. 1 pavyzdžio grafe $\delta(G)=0$, o $\Delta(G)=5$. Taip pat viršūnių skaičius žymimas $|V|=n$ ir atitinkamai kraštų skaičius $|E|=m$ [Ruo08].

$$v_{i_0},e_{j_1},v_{i_1},e_{j_2},\dots,e_{j_k},v_{i_k} \tag{2}$$

Kelias grafe $G=(V,E)$ yra baigtinė seka viršūnių ir kraštų (2 formulės forma), priklausančių

grafui *G*. Kelias prasideda viršūnėje. Viršūnės $v_{i_{t-1}}$ ir v_{i_t} yra krašto $e_{j_t}(t=1,\ldots,k)$ galinės viršūnės, o *k* yra kelio ilgis. v_{i_0} yra pradinė viršūnė, o v_{i_k} yra galutinė [Ruo08].

Grafo kraštai gali turėti kryptį. Tada toks grafas yra vadinamas kryptiniu grafu arba digrafu (2 pavyzdys).



2. pvz. Digrafo pavyzdys.

Digrafas nuo paprasto grafo skiriasi tuom, kad jo kraštų kryptis yra svarbi. Grafo kraštinė iš viršūnės *v* į viršūnę *u* yra užrašoma (v,u) , o priešingos krypties kraštinė iš viršūnės *u* į *v* užrašoma (u,v) [Ruo08].

Daugumoje, realių situacijų, naudojamų grafų kraštai turi tam tikras skaitines vertes vadinamas svoriais. Dažniausiai svoriai būna teigiami skaičiai, tačiau nebūtinai. Svorį gali turėti tiek orientuotas tiek neorientuotas grafas. Neretai grafo kraštinės svoris vadinamas kraštinės „kaina“. Svoriniai grafai savo kraštinėse gali saugoti atstumą tarp 2-iejų jungiamų viršūnių, reikalingą energijos kiekį norint įveikti atstumą tarp viršūnių ir panašiai. Pasitelkiant svorinius grafus yra įmanoma spręsti trumpiausio atstumo tarp grafo viršūnių uždavinį [Mcq09].

2.1.1. Grafų reprezentacija duomenų struktūromis

Programuojant grafą sudarančias viršūnes, svorius ir briaunas galima aprašyti įvairiomis duomenų struktūromis, pavyzdžiui:

1. Matricomis.
2. Sąrašais.

Grafo viršūnes, svorius ir briaunas galima išreikšti matricos pavidalu. Pavyzdžiui, *n*-osios eilės orientuoto multigrafo (multidigrafo) *G* viršūnės sunumeruotos skaičiais $1,\ldots,n$ ir a_{ij} - briaunų, išvestų iš *i*-osios į *j*-ąją viršūnes skaičius. Matrica A_G su elementais $a_{ij}, 1 \leqslant i, j \leqslant n$, vadinama gretinumo matrica. Neorentuoto multigrafo atveju, matrica yra simetrinė, o kilpų kiekis yra dvigubinamas [Man01].

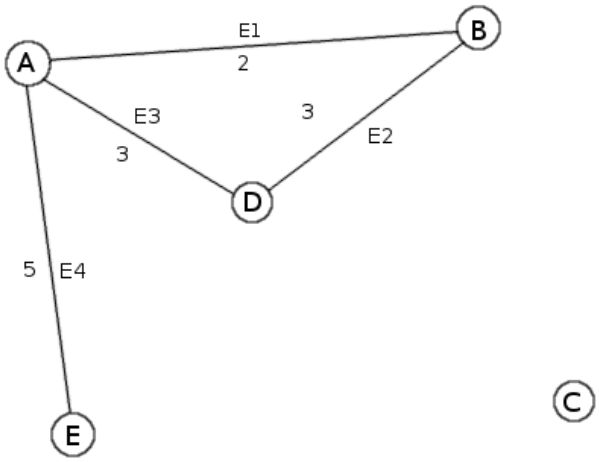
Sunumeravus ir briaunas skaičiais $1,\ldots,m$, čia *m* – grafo didumas, galime sudaryti grafo incidentumo matricą $B_G = B = (b_{ij}), 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m$, su

$$b_{ij} = \begin{cases} 1, & \text{jei } i \text{ viršūnė yra incidenti } j \text{ briaunai, kuri nėra kilpa} \\ 2, & \text{jei } i \text{ viršūnė yra incidenti } j \text{ briaunai, kuri yra kilpa} \\ 0, & \text{jei } i \text{ viršūnė yra incidenti } j \text{ briaunai} \end{cases}$$

Apibrėžiant digrafo incidentumo matricą, atsižvelgiama į briaunos kryptį. Dabar bekilpiam digrafui

$$b_{ij} = \begin{cases} 1, & \text{jei } i \text{ yra pradinė } j \text{ briaunos viršūnės} \\ -1, & \text{jei } i \text{ yra galinė } j \text{ briaunos viršūnė} \\ 0, & i \text{ viršūnė nėra incidenti } j \text{ briaunai} \end{cases}$$

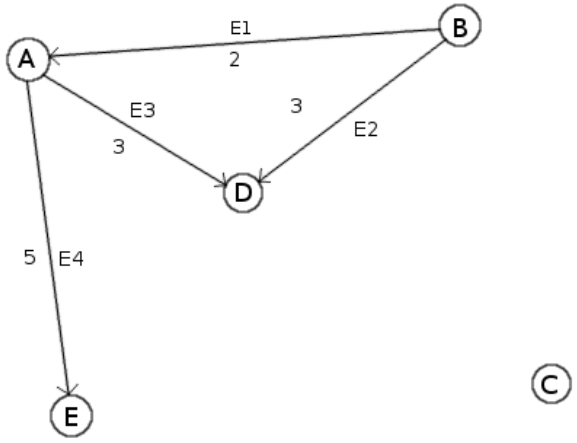
Jei *i* viršūnė yra incidenti kilpai, pažymėtai *j* numeriu, tai dažnai vartojamas žymėjimas $b_{ij} = -0$ [Man01]. Neorientuoto svorinio paprastojo grafo atveju, jo viršūnių, briaunų ir svorių reprezentacija atrodytų kaip 3 pavyzdyje.



$$Matrica = \begin{array}{c} \begin{matrix} E1 & E2 & E3 & E4 \end{matrix} \\ \begin{bmatrix} 2 & 0 & 3 & 5 \\ 2 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}$$

3. pvz. Neorientuoto svorinio paprastojo grafo reprezentacija matricos pavidalu.

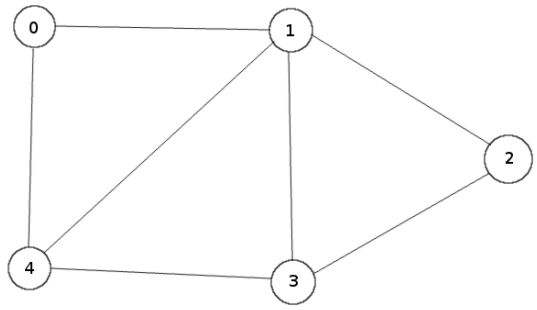
O orientuoto svorinio paprastojo grafo atveju, jo viršūnių, briaunų ir svorių reprezentacija atrodytų kaip 4 pavyzdyje.



$$Matrica = \begin{array}{c} \begin{matrix} E1 & E2 & E3 & E4 \end{matrix} \\ \begin{bmatrix} -2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -3 & -3 & 0 \\ 0 & 0 & 0 & -5 \end{bmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}$$

4. pvz. Orientuoto svorinio paprastojo grafo reprezentacija matricos pavidalu.

Grafo viršūnes, svorius ir briaunas taip dar galima išreikšti sąrašų pavidalu, kadangi matricinė grafo išraiška turi perterklinių duomenų (matricos nulinės vertės), sąrašų pavidalas neretai būna priimtinesnis variantas jei turime ribotą atmintį. Tokia išraiška sąrašais vadinama gretimumo sąrašu (angl. *adjacency list*). Neorientuoto paprastojo grafo viršūnių ir briaunų reprezentacija sąrašų aibe galėtų atrodyti kaip 5 pavyzdyje.



Viršūnių sąrašai:

- $0 \rightarrow 1 \rightarrow 4$
- $1 \rightarrow 0 \rightarrow 4 \rightarrow 2 \rightarrow 3$
- $2 \rightarrow 1 \rightarrow 3$
- $3 \rightarrow 1 \rightarrow 4 \rightarrow 2$
- $4 \rightarrow 3 \rightarrow 0 \rightarrow 1$

5. pvz. Neorientuoto paprastojo grafo reprezentacija sąrašų pavidalu.

Norint grafą su *n* viršūnių išreikšti gretimumo sąrašu, prireiks *n* sąrašų, kuriuose bus surašytos visos kraštinės, kurias turi viršūnė *i.* Sąrašas *i* turi savyje viršūnę *j* jei tarp viršūnių *i* ir *j* yra kraštas. Svorinis grafas gali būti reprezentuojamas viršūnių ir svorių porų sąrašais [BocBla08].

2.2. Algoritmo sudėtingumas

Algoritmų mokslo srityje dažniausias resursas yra laikas. Jis dažniausiai skaičiuojamas remiantis dydžiu *n* – įvesties eilutės (angl. *Input string*) ilgiu [God04].

Rinkinys problemų, kurios gali būti išspręstos per polinominį laiką yra vadinamos *P* sudėtingumo problemomis. Tai reiškia, kad sprendžiant *P* sudėtingumo problemą egzistuoja eksponentė *k* ir algoritmas, kurio sudėtingumas $O(n^k)$ (*n* - įvesties eilutės ilgis). *P* sudėtingumo problemų klasė – tai praktiškai išsprendžiamų problemų klasė [God04].

Rinkinys problemų, kurios gali būti išspręstos polinominiu laiku nesinaudojant determinismu yra vadinamos *NP* sudėtingumo problemomis. Ne deterministiniai algoritmai, kurie gauna ne tik įprastą įvesties eilutę, bet ir užuominą (dar vadinama sertifikatu), padedančią priimti teisingą atsakymą. Iškelta problema priklauso *NP* sudėtingumo problemų klasei, jei egzistuoja tokia eksponentė *k* ir ne deterministinis algoritmas problemai, kurios sprendimas su užuominomis užtrunka $O(n^k)$ laiką, kur *n* yra įvesties eilutės ilgis [God04].

Nors panašu, kad *P* ir *NP* sudėtingumo problemos yra skirtingos, tačiau matematikos ir kompiuterių moksle viena iš neišspręstų problemų yra:

$$P \neq NP \tag{3}$$

Jei 3 formulė yra teisinga, vadinasi dauguma problemų, kurioms norėtume turėti efektyvius algoritmus jų neturi, o jei 3 formulė yra neteisinga, vadinasi daugelis kriptografijos formuluočių yra neteisingos [God04]. Tai yra viena iš Clay matematikos instituto keliamų tūkstantmečio problemų, už kurios sprendimą yra paskirta 1 milijono dolerių premija. Šios problemos gyvenimišką pavyzdį būtų galima suformuluoti taip:

> *Tarkime, kad gauti būstą universiteto bendrabutyje nori 400 studentų, o vietų bendrabutyje yra 100. Taip pat universiteto dekanas pateikė sąrašą studentų porų, kurie abu vienu metu negali gyventi bendrabutyje.*

Ši problema yra vadinama NP sudėtingumo problema, nes ganėtinai lengva patikrinti ar iš 100 žmonių nėra porų iš dekano pateikto sąrašo, tačiau sudarymas teisingo 100 žmonių sąrašo tenkinančio keliamus reikalavimus nėra lengva užduotis. Iš tikrųjų skaičius, galimų variantų sudaryti 100 žmonių sąrašą iš 400 kandidatų, yra didesnis nei atomų skaičius visatoje. Tokio problemos sprendimas perrinkimo būdu (brutalia jėga, angl. *brute force*) praktiškai yra neįmanomas, todėl kompiuterių moksle nustatymas ar yra koks nors metodas ar algoritmas galintis greičiau išspręsti tokio pobūdžio problemas yra gana svarbus uždavinys [CMI14].

Nors ir negalime įrodyti ar $P=NP$, tačiau galime identifikuoti problemas, kurios tikrai yra tik *NP* sudėtingumo problemos. Tokio tipo problemos yra vadinamos pilno *NP* sudėtingumo problemomis. Jos pasižymi savybe, jei bent vienai problemai yra polinominio laiko algoritmas, tada visoms *NP* klasės problemoms yra polinominio laiko algoritmas [God04]. Apibrėžimas:

Problema *S* yra vadinama pilno *NP* sudėtingumo problema, jei:

1. Ji priklauso *NP* klasei, ir
2. Visiems *A* iš *NP* galioja savybė $A \leq p\, S$.

Tai reiškia, kad:

- Jei *S* priklauso pilno *NP* sudėtingumo problemai ir *S* priklauso *P*, tada $P=NP$.
- Jei *S* priklauso pilno *NP* sudėtingumo problemai ir T priklauso *NP* ir $S \leq p\, T$, tada *T* priklauso *NP-complete.*

6. pvz. Pilno *NP* sudėtingumo problemos apibrėžimas. $A \leq p\, S$ užrašymas reiškia, kad problema *A* gali būti polinomiškai sumažinta laike iki problemos *S*) [God04].

3. Optimalus atstumas tarp dviejų taškų

3.1. Problema

Trumpiausio kelio paieška tarp dviejų taškų yra jau senai nagrinėjama sritis. Ieškant trumpiausio kelio, siekiama sutrumpinti kelio atstumą ir tuo pačiu kelionės laiką. Pavyzdžiui greitosios pagalbos, policijos pareigūnams gavus iškvietimą neretai yra gyvybės ir mirties klausimas ar jie spės atvykti į įvykio vietą. O dar jei iškvietimas būna dideliame mieste piko metu, kai eismo sąlygos keičiasi kas keletą minučių, trumpiausią atstumą rasti reikia kuo greičiau.

Siekiant apskaičiuoti atstumą tarp grafo viršūnių, jo kraštinėms yra suteikiami atitinkami svoriai, kurie gali reikšti atstumą, laiką, pralaidumą ir t.t. Grafui su tik teigiamais kraštinių svoriais klasikinis algoritmas skaičiuoti trumpiausią atstumą yra Dijkstra algoritmas, paskelbtas 1956 m. E. W. Dijsktra. Grafams tiek su teigiamais, tiek su neigiamais kraštinių svoriais klasikinis algoritmas skaičiuoti trumpiausią atstumą yra Floyd-Warshall algoritmas, paskelbtas R. Floyd 1962 m. ir S. Warshall 1962 m. Nors abu algoritmai publikuoti daugiau nei prieš 50 metų, tačiau remiantis jais buvo ir yra kuriami šių laikų algoritmai, todėl ir yra svarbu juos žinoti nagrinėjant trumpiausio maršruto algoritmus.

3.2. Dijkstra algoritmas

Dijsktra algoritmas ieško trumpiausio kelio grafe su teigiamais svoriais (angl. *weighted*) nuo pradinės iki galutinės viršūnės. Algoritmo efektyvumas yra $O(|E|+|V|\log(|V|))$.

Iš tikrųjų Dijkstra algoritmas sprendžia vieno šaltinio trumpiausio kelio (angl. *single-source shortest-path*) problemą, tai reiškia kad algoritmas randa trumpiausia atstumą nuo pradinio iki visų kitų grafo viršūnių vienu metu [WolRao00]. Dijsktra algoritmo pseudo-kodas:

- Trumpiausias kelias tarp jungčių prilyginamas begalybei.
- Pradinis atstumas nuo pradžios taško iki pradžios taško lygus 0.
- Kol yra nežinomų viršūnių (kurių atstumas lygus begalybei) grafe:
  - Gobšaus pasirinkimo principu, pasirenkama nežinoma viršūnė *N* su trumpiausiu atstumu.
  - N pažymimas kaip žinomas.

- Kiekvienai *N* gretimai viršūnei *A*:
  - Jei kelias nuo pradinio taško iki *N*, suma su keliu tarp *N* ir *A* yra mažesnė už atstumą nuo pradinio taško iki *A*:
    - Atstumas nuo pradinio taško iki *A* nustatomas kaip kelio nuo pradinio taško iki *N* suma su keliu tarp *N* ir *A*.
    - Išsaugojamas kelias nuo pradinio taško iki *A*.

7. pvz. Dijsktra algoritmo pseudo-kodas [WolRao00].

Dijkstra algoritmas yra „gobšaus“ (angl. *greedy*) algoritmo pavyzdys. Gobšūs algoritmai visada atlieka sprendimus, kurie tuo metu atrodo geriausi neatsižvelgdami į galimus geresnius variantus. Dijsktra atveju – pasirenkamas tuo metu geriausias kelias, tačiau yra tikimybė, kad egzistuoja geresnis ir trumpesnis variantas [WolRao00]. Gobšių algoritmų savybės:

- Trumparegystė – nenumatoma, kokie pokyčiai bus išanalizavus tolimesnes grafo viršūnes.
- Pasižymi lokaliu optimalumu, kuris dažnai negarantuoja globalaus optimalumo.

Kadangi Dijkstra algoritmas ieškodamas trumpiausio atstumo tarp dviejų taškų taip pat randa trumpiausius atstumus iš pradinio taško į visas grafo viršūnes, šis algoritmas nėra pats efektyviausias sprendžiant pastarąją problemą. Yra nemažai algoritmų įkvėptų Dijsktra algoritmo, tačiau A* algoritmas yra *de facto* algoritmas ieškant trumpiausio atstumo tarp 2 taškų [LeiLouChr07].

Nors mokslo tiriamojo darbo metu atstumų tarp 2 grafo viršūnių neieškojome (nes buvo analizuojami pilni grafai), tačiau realiuose grafuose dažniausiai tiesioginio atstumo tarp 2 viršūnių nebūna. Todėl ateities darbuose bus pasitelkiama tokio pobūdžio algoritmais sprendžiant keliamus uždavinius.

3.3. Floyd-Warshall algoritmas

Floyd-Warshall algoritmas ieško visų grafo viršūnių porų trumpiausius maršrutus. Grafo kraštai gali turėti tiek teigiamus, tiek neigiamus svorius, todėl algoritmas geba surasti netik trumpiausią, bet ir mažiausiai „sveriantį“ kelią. Algoritmo sudėtingumas yra $O(|V|^3)$.

Turint grafą $G=(V,E)$, su viršūnių rinkiniu $V=\{1,2,\ldots,n\}$ ir kraštų rinkiniu $E=\{(i,k):i,k\in V,i\neq k\}$, kur $|V|=n$, Floid-Warshal algoritmas yra bene vienas garsiausių ir geriausių algoritmų grafe G ieškančių trumpiausių atstumų tarp visų viršūnių i ir k porų. Šis algoritmas remiasi 4 žingsnių procedūra, kurioje yra skaičiuojamos 2 kvadratinės matricos D_j ir R_j, kur $j=0,\ldots,n$. Matricose saugomi trumpiausiai maršruto ilgiai, ir patys maršrutai tarp i ir k viršūnių. Floyd-Warshall algoritmas pasižymi tuom, kad atlieka nemažai skaičiavimų. Grafe, kur $|V|=n$, D_j ir R_j matricos turi būti suskaičiuotos $n+1$ kartų pradedant nuo D_0 ir R_0, kur kiekviena matrica turi n^2-n verčių [AinSal12]. Detaliau algoritmas pateiktas 8 pavyzdyje:

1. Sukuriamos $n\times n$ matricos D_j ir R_j
2. Kai $j=0$ suskaičiuojamos D_0 ir R_0 matricos:

   $D_0=[d_{ik}]$, kur

   $$d_{ik}=\begin{cases} d_{ik}, & \text{jei yra tiesioginis kelias tarp viršūnių } i \text{ ir } k \\ \infty, & \text{jei nėra tiesioginio kelio jungiančio viršūnes } i \text{ ir } k \\ 0, & \text{jei } i=k \end{cases}$$

   $R_0=[r_{ik}]$, kur

   $$r_{ik}=\begin{cases} k, & \text{jei yra tiesioginis kelias tarp viršūnių } i \text{ ir } k \\ -, & \text{jei nėra tiesioginio kelio jungiančio viršūnes } i \text{ ir } k \\ -, & \text{jei } i=k \end{cases}$$

3. Likusiems $j=1,\ldots,n$ suskaičiuojamos D_j ir R_j matricos. j-osios matricos vertės gaunamos iš prieš tai buvusių D_{j-1} ir R_{j-1} matricų.

   $D_j=[d_{ik}]$, kur

   $$d_{ik}=\begin{cases} d_{ik}, & \text{jei } i=k, i=j, k=j \\ min(d_{ik},d_{ij}+d_{jk}), & \text{priešingai} \end{cases}$$

   $R_j=[r_{ik}]$, kur

   $$r_{ik}=\begin{cases} k, & \text{jei } i=k, i=j, k=j \\ -, & \text{jei } d_{ik}\leqslant d_{ij}+d_{jk} \\ -, & \text{jei } d_{ik}>d_{ij}+d_{jk} \end{cases}$$

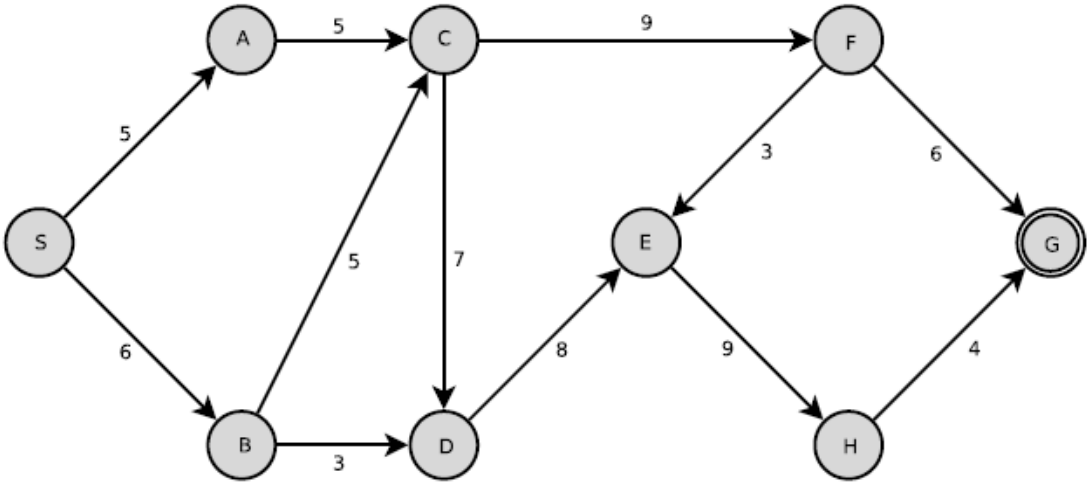
4. Kartojame 3-ą žingsnį kol randamos D_n ir R_n matricos.

8. pvz. Floyd-Warshall algoritmas [AinSal12].

Floyd-Warshall algoritmas vėl gi, sprendžia platesnį uždavinį nei atstumo tarp dviejų taškų. Todėl šis algoritmas nėra pats optimaliausias sprendžiant paprastą optimalaus maršruto problemą, tačiau galima įžvelgti platesni šio algoritmo panaudojimą sprendžiant sudėtingesnes problemas. Pavyzdžiui ieškant maršruto dideliame mieste, kuriame eismo sąlygos labai greitai keičiasi, ateities darbuose būtų galima remtis turimais šio algoritmai skaičiavimais greitai perskaičiuoti trumpiausią maršrutą.

3.4. A* algoritmas

A* yra trumpiausio kelio paieškos algoritmas, kuris sukurtas remiantis Dijsktra algoritmu. Tai yra standartinis, dažniausiai naudojamas algoritmas, kuris, tinkamomis sąlygomis, garantuoja rasti trumpiausią medžio paieškos (angl. *tree traversal*) kelią. A* ir kiti trumpiausio kelio paieškos algoritmai, paiešką atlieka grafo tinkle. 9 pavyzdyje matoma grafo sistema – viršūnių ir kraštų visuma. Kiekvienas kraštas turi teigiamą svorį, kuris nurodo patirtas išlaidas keliaujant tuo kraštu. Trumpiausio kelio paieškos algoritmai bando surasti viršūnių ir kraštų seką nuo pradinės iki galutinės viršūnės taip, kad būtų kuo mažesnės išlaidos [LeiLouChr07].



9. pvz. Viršūnių ir kraštų sistema (pradinė viršūnė – S, galutinė viršūnė - G) [LeiLouChr07].

A* algoritmo veikimo principas pagrįstas medžio paieška, sudarant visus galimus trumpiausius kelius iki tikslo, paieškos metu kiekvieno jų trumpumą įvertinant, remiantis formule:

$$f(x)=g(x)+h(x) \tag{4}$$

kur *x* - kažkuris nepilnas kelias, *f* – apytikrės galutinės trumpiausio kelio išlaidos keliu *x* iki tikslo, *g* – *x* kelio paieškos išlaidos, *h* – įvertinta likusio kelio dalis nuo kelio *x* pabaigos iki tikslo viršūnės *g*. Garantija, kad pasirinktas kelias yra trumpiausias priklauso tik nuo to, ar likusio kelio dalies įvertinimas nebus pervertintas [LeiLouChr07]. A* algoritmo pseudo-kodas pateiktas 10 pav.

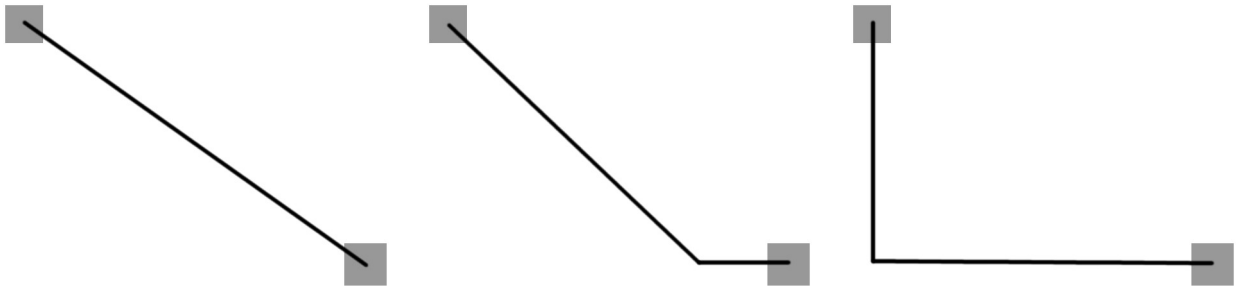
- Sudaroma vieno elemento eilė, kurią sudaro tik pradžios viršūnė.
- Kol pirmasis kelias eilėje nesibaigia galutiniame taške, arba eilė nėra tuščia:
  - Ištrinamas pirmasis kelias iš eiles; sukuriami nauji keliai, kurie pratesia prieš tai ištrintą kelią į kaimyninius mazgus.
  - Atmetami keliai, kurie jungiasi patys su savimi.
  - Jei 2 ar daugiau kelių pasiekia bendrą mazgą, ištrinami visi keliai išskyrus tą, kurio bendras kelio ilgis trumpiausias.
  - Išrūšiuojama visa eilė naudojant 4 formulę (trumpiausi keliai būna eilės pradžioje).
- Jei pasiekiamas galutinis taškas pranešama sėkmė, kitu atveju – nesėkmė.

10. pvz. A* algoritmo pseudo-kodas [LeiLouChr07].

A* algoritmo pseudo-kodą įmanoma optimizuoti pasirenkant kitokį trumpiausių kelių išsaugojimo mechanizmą – vietoj išrūšiuotos eilės (angl. *priority queue*) pasirenkant stekų sąrašą (angl. array of stacks). Pasirinkus mažus *f(x)* intervalus galima implementuoti šį optimizavimą. Tada kiekviename steke yra kaupiamos tam tikro intervalo *f(x)* viršūnės, o steko indeksas ir yra *f(x)* vertė. Tokiu būdu gaunama sistema, kurioje pastoviu laiku yra išsaugojamos naujos grafo viršūnės, tačiau trumpiausias kelias randamas greičiau [Caz06].

3.4.1. Euristikos

A* algoritmo veikimo efektyvumas labai priklauso nuo likusios kelio dalies įvertinimo, vadinamo euristika (angl. *heuristics*). Norint, kad A* algoritmas rastu trumpiausia kelią reikia, kad euristika nepervertintų likusio atstumo iki tikslo. Kai A* algoritmas randa kelią, kuris yra trumpesnis nei visi kiti tarpiniai keliai sudėti su likusiu atstumu iki galutinės viršūnės, toks kelias būna optimalus. Optimalus kelias negali būti trumpesnis nei jo įvertintas ilgis. Jei euristika pervertintų kelią, tada kiltu rizika, jog A* algoritmas atmes trumpesnį kelią, nes parinkto dalinio kelio ilgio suma su įvertinta likusia kelio dalimi iki tikslo bus ilgesnė, nei rastas trumpiausias atstumas iki tikslo [LeiLouChr07].



11. pvz. Euristikos: (a) Euklido, (b) diagonulus sutrumpinimas, (c) Manhatano blokas.

Norint rasti visiškai optimalų trumpiausią maršrutą, pasirenkama Euklido euristika (11 a pav.), kuri garantuoja, kad atstumas nuo esamos grafo viršūnės iki tikslo nebus pervertintas. Euklido euristika puikiai padeda rasti trumpiausią maršrutą tarp dviejų taškų, tačiau apskaičiuojant likusį atstumą reikia du kartus panaudoti kėlimą kvadratu ir ištraukti šaknį (5 formulė), o tai sunaudoja nemažai procesoriaus skaičiavimo resursų [LeiLouChr07].

$$Atstumas = \sqrt{(x_{kelio} - x_{tikslo})^2 + (y_{kelio} - y_{tikslo})^2} \quad (5)$$

Tačiau jeigu mums nereikėtų rasti idealaus trumpiausio kelio, jeigu mus tenkintu beveik optimalus trumpiausias kelias, tada būtų galima naudoti kitokias euristikas. Diagonalaus sutrumpinimo euristika (11 b pav.) geresnė nei Euklido skaičiavimo resursų prasme, nes skaičiuojant atstumą naudojama tik šaknis (6 formulė) [LeiLouChr07].

$$Atstumas = \sqrt{2} * dx + (dx - dy) \quad (6)$$

Dar viena iš galimų euristikų yra Manhatano bloko euristika (11 c pav.). Joje sumuojamas vertikalus ir horizontalus nukeliautas atstumas. Skaičiavimo efektyvumo prasme ji yra pati optimaliausia iš triejų paminėtų euristikų, tačiau Manhatano bloko euristika turi didžiausia tikimybę pervertinti likusį atstumą [LeiLouChr07].

Taigi, optimalios euristikos pasirinkimas priklauso nuo uždaviniui keliamų reikalavimų. Norint rasti idealų trumpiausią maršrutą turint neribotą laiką bus pasirenkama Euklido euristika, o norint rasti beveik optimalų maršrutą daug greičiau bus pasirenkama Manahtano bloko, ar panaši euristika.

4. Optimalaus maršruto, aplankančio visus paskirties taškus algoritmai

4.1. Problema

Praktikoje yra nemažai pavyzdžių, kuriuose reikia aplankyti eilę nurodytų taškų kuo trumpesniu keliu. Tai labai aktualu logistikoje kurjeriams bei tolimųjų reisų vairuotojams, taip pat sudarinėjant autobusų maršrutus, nes kuo trumpesnis maršrutas - tuo mažiau išlaidų ir didesnės pajamos. Tokia problema yra *NP* sudėtingumo problema, todėl visiškai tikslus uždavinio sprendimas yra gaunamas tik atlikus pilną perrinkimą. Tačiau pilnas perrinkimas didesniam miestų skaičiui yra labai lėtas metodas, todėl tam tikrais atvejais pasirenkami algoritmai, kurie randa beveik trumpiausią maršrutą per gerokai trumpesnį laiką.

4.1.1. Keliaujančio pirklio problema

Nagrinėjant sudėtingas trumpiausio maršruto problemas, dažnai sprendžiama susijusi keliaujančio pirklio problema, kuri formuluojama taip:

> *Turint tam tikrą kiekį miestų, taip pat kelionės iš vieno miesto į kitą kainas, reikia rasti pigiausią maršrutą, kad aplankius kiekvieną miestą maršrutas baigtųsi pradiniame mieste.*

Keliaujančio pirklio problema ir yra optimaliausio maršruto, aplankančio visus paskirties taškus, radimas. Iš esmės keliaujančio pirklio problemos sprendinys yra trumpiausias grafo Hamiltono ciklas. Hamiltono ciklas grafe yra uždaras kelias, kuris aplanko kiekvieną grafo viršūnę tik vieną kartą ir baigiasi pradiniame taške. Grafas, kuris turi Hamiltono ciklą yra vadinamas Hamiltono grafu, tačiau grafas tokio ciklo gali ir neturėti.

Keliaujančio pirklio problemą grafų teorijoje būtų galima užrašyti ir taip:

Tarkime $V=\{a,\ldots,z\}$ yra miestų rinkinys, $E=\{(r,s):r,s\in V\}$ yra kraštų rinkinys ir $\delta(r,s)=\delta(s,r)$ yra kelio ilgumo įvertinimas susijęs su kraštu $(r,s)\in V$.

Tada keliaujančio pirklio problema yra uždaro kelio rato, kuris aplanko kievieną miestą vieną kartą, suradimas.

Jei miestai $r\in V$ yra duoti su koordinatėmis (x_r, y_r) , o $\delta(r,s)$ yra Eiklidinis atstumas tarp miestų *r* ir *s*, tada turime Euklidinę keliaujančio pirklio problemą.

12. pvz. Keliaujančio pirklio problema [DorGam97].

Nepaisant to, kad keliaujančio pirklio problema yra intuityvi ir ją nesunku suformuluoti, tai yra viena labiausiai analizuojamų pilnos *NP* klasės sudėtingumo kombinatorinių optimizacijos problemų. Problemos sprendimo sudėtingumas tampa labai aiškiai suvokiamas net ir nuo nedidelio miestų skaičiaus, norint „brutalia jėga“ (angl. *brute force*) rasti trumpiausią maršrutą. Pavyzdžiui, 20 miestų $(n=20)$ problemai išspręsti „brutalia jėga“, reiktų perrinkti $(20-1)!/2$ maršrutų, o tai yra daugiau nei 10^{18} galimų variantų. Manoma, kad keliaujančio pirklio problema priklauso problemų klasei, kurios sprendimas yra eksponentinio laiko sudėtingumo ir kad nėra įmanoma sukurti algoritmą, kuris galėtų išspręsti šią problemą per polinominį laiką [Sin12].

Šiuo metu yra 2 kategorijos algoritmų, sprendžiančių keliaujančio pirklio problemą:

- *Tikslūs* algoritmai;
- *Apytiksliai* (arba *heuristiniai*) algoritmai.

Tikslūs algoritmai ieško pačio trumpiausio atstumo keliaujančio pirklio problemoje per ribotą žingsnių skaičių. Deja tokie algoritmai dažnai būna labai sudėtingi ir reikalauja daug procesoriaus skaičiavimo resursų. Vieni iš efektyviausių tikslių algoritmų yra perpjautos plokštumos (angl. *Cutting-Plane*) ir paviršių radimo (angl. *Face-Finding*) algoritmai. Apytiksliai algoritmai negarantuoja rasti optimaliausią maršrutą, bet sugeba pasiūlyti sąlygini optimalų maršrutą (keliais procentais prastesnį už optimaliausią) ir jų vykdymo laikas yra žymiai trumpesnis, o ir patys algoritmai yra paprastesni. Apytiksliai algoritmai skirstomi į 3 klases [Sin12]:

1. Konstravimo (angl. *Construction*) algoritmai, kurie sudaro optimalų maršrutą po kiekvieno iteracijos žingsnio pridėdami po vieną miestą.
2. Tobulinimo (angl. *Improvement*) algoritmai, kurie optimizuoja sugeneruotą maršrutą kiekvieną iteraciją.
3. Sudėtiniai (angl. *Composite*) algoritmai, kurie apjungia konstravimo ir tobulinimo algoritmus.

4.2. Pilnas perrinkimas

Vienintelis tikslus keliaujančio pirklio problemos sprendimo būdas yra pilnas perrinkimas (visų permutacijų kelio ilgio įvertinimas). Šis metodas dažnai vadinamas brutalios jėgos metodu (angl. *brute force*). Pavyzdžiui, norint rasti trumpiausią maršrutą tarp 20 miestų, reiktų palyginti

$(20-1)!/2$ permutacijų, kas apskaičiavus būtų 60822550204416000 maršrutų. Sprendžiant n miestų keliaujančio pirklio problemą, šio metodo sudėtingumas yra $O((n-1)!/2)$.

Kadangi daugeliu atveju turime maršrutą apskaičiuoti sąlyginai greitai, šis metodas nėra priimtiniausias. Siekiant drastiškai sumažinti skaičiavimo laiką sprendžiant keliaujančio pirklio problemą, pasirenkami kiti algoritmai (skruzdėlių kolonijos, genetiniai algoritmai ir t.t.), kurie randa beveik trumpiausią maršrutą per gerokai trumpesnį laiko tarpą.

4.3. Skruzdėlių kolonijos sistemos algoritmas

Skruzdėlių kolonijos sistemos algoritmas paremtas gamtoje gyvenančių skruzdėlių elgesiu ir vienas iš jo panaudojimų yra keliaujančio pirklio problemos sprendimui. Šis algoritmas ieško optimalaus, bet ne pačio optimaliausio maršruto, todėl jis priskiriamas apytiksliems algoritmams.

Gyvosios skruzdėlės gali rasti trumpiausią atstumą nuo savo lizdo iki maisto šaltinio ir atgal, nenaudodamos jokių vizualių užuominų, bet remdamosis feromonų informacija. Vaikščiodamos skruzdėlės ant žemės skleidžia savo feromonus ir seka kitų skruzdėlių išskleistus feromonus. 13 pavyzdyje matyti kaip skruzdėlės naudodamos feromonus randa trumpiausią atstumą. Toks gyvųjų skruzdėlių elgesys įkvėpė skruzdėlių kolonijos sistemos algoritmą. Tokiame algoritme grupė dirbtinių skruzdėlių bendradarbiauja ieškodamos trumpiausio kelio, palikdamos savo feromonus ant grafo kraštų [DorGam97].

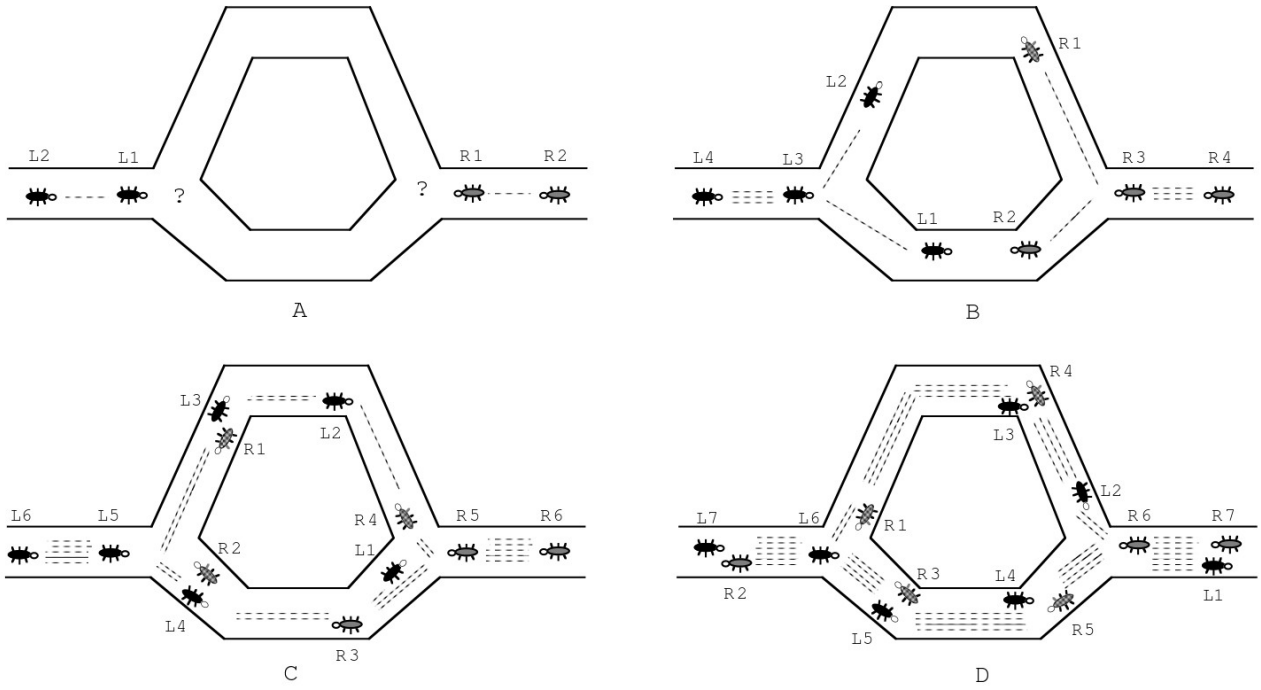
Skruzdėlių kolonijos sistemos algoritmas išplečia keliaujančio pirklio problemą. Algoritmas, vertindamas kiekvienos kraštinės (r,s) ilgį, vertina ne tik kraštinės svorį $\delta(r,s)$, bet ir pageidautinumą $\tau(r,s)=\tau(s,r)$, vadinamą feromonu, kuris atnaujinamas dirbtinių skruzdėlių kolonijos algoritmo veikimo metu [DorGam97]. Algoritmas veikia praktiškai taip:

1. Kiekviena skruzdėlė sugeneruoja pilną maršrutą pasirinkdama miestus remdamasi tikimybine būsenos kitimo taisykle (7 formulė). Skruzdėlės keliauja tarp miestų trumpesniais keliais, kurie turi didesnius feromono kiekius.

2. Kai visos skruzdėlės užbaigia savo maršrutus, aktyvuojama globali feromonų atnaujino taisyklė (arba globali atnaujinimo taisyklė) (8 formulė). Nedidelis kiekis feromonų „išgaruoja“ nuo grafo kraštinių, o kraštai iš skruzdėlės maršruto gauna feromono priklausomai nuo to, kiek trumpas visas maršrutas buvo. Kitaip tariant kraštai, kurie

priklauso trumpiems maršrutams tampa „patrauklesni“ skruzdėlėms.

3. Procesas kartojamas iš naujo pasirinktą iteracijų skaičių, arba tam tikrą laiką.



13. pvz. Kaip tikros skruzdėlės randa trumpiausią kelią. a) Skruzdėlės atkeliauja iki kryžkėlės. b) Kai kurios skruzdės pasirenka viršutinį kelią, kai kurios apatinį. Pasirinkimas yra atsitiktinis. c) Kadangi skruzdės juda apytiksliai vienodu, pastoviu greičiu, skruzdėlės, kurios pasirinko apatinį kelią, kliūtį apeina greičiau, nei per viršutinį kelią. d) Feromonas kaupiasi spartesniu tempu trumpesniame kelyje, todėl skruzdės ji pasirenka dažniau. Brūkšniuotos linijos atvaizduoja apytikslį feromono kiekį [DorGam97].

Būsenos kitimo taisyklė pateikta 7 formulėje. Dar ši taisyklė gali būti pavadinta atsitiktinio proporcingumo (angl. *random-proportional*) taisykle, kuri nusako tikimybę skruzdėlei *k* mieste *r* pasirinkti kelią į miestą *s*:

$$p_k(r,s)=\begin{cases}\dfrac{[\tau(r,s)]\cdot[\eta(r,s)]^{\beta}}{\sum_{u\in J_k(r)}[\tau(r,u)]\cdot[\eta(r,u)]^{\beta}}, & \text{jei} \quad s\in J_k(r)\\ 0, & \text{priešingu atveju}\end{cases} \tag{7}$$

kur τ yra feromonas, $\eta=1/\delta$ yra atstumo $\delta(r,s)$ inversija, $J_k(r)$ yra rinkinys miestų, kuriuos dar reikia aplankyti skruzdėlei *k* iš miesto *r*, o β yra parametras, kuris nulemia santykinę

svarbą tarp feromono ir kraštinės ilgio $(\beta>0)$ [DorGam97].

7 formulėje kraštinės (r,s) feromoną padauginame iš atitinkamos heuristinės vertės $\eta(r,s)$. Tokiu būdu pirmenybė teikiama trumpesniems kraštams, turintiems daugiau feromono. Kai visos skruzdėlės užbaigia savo maršrutus, yra atnaujinami kraštinių feromonų kiekiai remiantis formule:

$$\tau(r,s)\leftarrow(1-\alpha)\cdot\tau(r,s)+\sum_{k=1}^{m}\Delta\tau_k(r,s) \tag{8}$$

kur

$$\Delta\tau_k(r,s)=\begin{cases}\dfrac{1}{L_k}, & \text{jei } (r,s) \text{ priklauso skruzdėlės } k \text{ keliui}\\ 0, & \text{priešingu atveju}\end{cases}$$

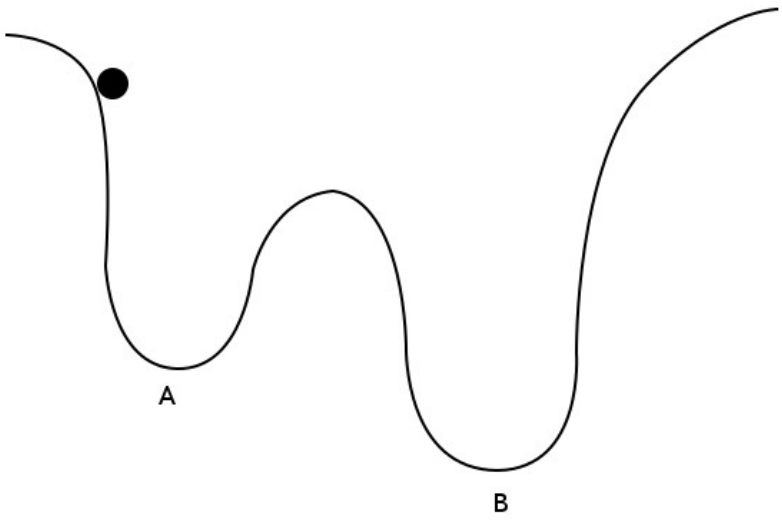
$0<\alpha<1$ yra feromono nykimo parametras, L_k yra skruzdėlės *k* kelio ilgis, o *m* yra skruzdėlių skaičius.

Feromonas saugomas ant grafo kraštų, o ne pačiose skruzdėlėse. 8 feromono atnaujinimo taisyklė simuliuoja fermono kaitą atsižvelgiant į tai kiek skruzdėlių aplanko atitinkamą grafo kraštą ir kiek jo „išgaruoja“ [DorGam97].

4.4. Simuliuoto „atkaitinimo“ algoritmas

Atkaitinimas – tai procesas, kurio metu yra randama mažos energijos būsena. Simuliuoto „atkaitinimo“ algoritmo pagalba galima ieškoti beveik optimalaus maršruto keliaujančio pirklio problemai spręsti, todėl šis algoritmas priskiriamas apytiksliems algoritmams.

Gamtoje tai atliekama įkaitintą metalą lėtai vėsinant. Temperatūra yra kontroliuojama vertė atkaitinimo proceso metu, kuri nulemia kiek atsitiktinė yra energijos būsena. Suprasti, kodėl simuliuotas atkaitinimas veikia, galima išanalizavus 14 pavyzdį. Energijos barjeras visą pasiskirstymą padalina į du lokalius energijos minimumus *A* ir *B*. Kamuoliukas, padėtas atsitiktinėje vietoje ant barjero kreivės, judėdamas žemyn turi vienodą tikimybę atsidurti tiek *A*, tiek *B* energijos minimume [MalGurDyp+89].



14. pvz. Energijos pasiskirstymas.

Norint padidinti tikimybę kamuoliukui patekti į *B* energijos minimumą, sistemą „drebiname“ sistemą simuliuodami temperatūros pokyčius. Dėl nedidelio energijos barjero, sistemos „drebinimas“ padidina tikimybę kamuoliukui įveikti energijos barjerą tarp *A* ir *B*, taip pat, kuo labiau „drebiname“ sistemą, tuo didesnė tikimybė kamuoliukui įveikti barjerą abiejomis kryptimis. Todėl drebinimas pradžioje turi būti didesnis ir einant laikui mažėti. Mažinant „drebėjimo“ intensyvumą mažinama tikimybė įveikti barjerą, tačiau tikimybė įveikti barjerą iš *A* į *B* yra didesnė, nei iš *B* į *A*. Rezultate gauname kamuoliuką, mažesnės energijos duobėje (*B*) [MalGurDyp+89].

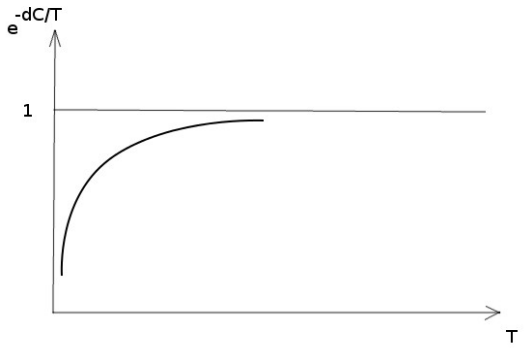
Simuliuoto „atkaitinimo“ algoritmo pseudo-kodas pateiktas 15 pavyzdyje:

- Kol nėra pasiektas ciklo sustabdymo kriterijus:
  - Kol nėra pasiekta pusiausvyra:
    - Sugeneruojamas naujas žingsnis.
    - Jei $\Delta C<0$, arba jei $e^{-\Delta C/T}>random(0.0,1.0)$:
      - Atnaujinamas maršrutas.
  - Suskaičiuojama nauja temperatūra.

15. pvz. Simuliuoto „atkaitinimo“ algoritmo pseudo-kodas, kur *T* – temperatūra, ΔC - kelio atstumo pokytis, $random(0,1)$ - atsitiktinis skaičius intervale (0.0, 1.0) [MalGurDyp+89].

Vidiniame cikle sugeneruojamas naujas žingsnis. Jei žingsnis atitinka pripažinimo funkciją ($\Delta C<0$, arba $e^{\Delta C/T}>random(0,1)$), jis yra prijungiamas prie esamo maršruto. Ciklas nutrūksta pasiekus pusiausvyrą, kai nebėra didelių energijos (atstumo) pokyčio amplitudžių. Išoriniame cikle mažinant temperatūrą yra optimizuojamas maršrutas tol kol ciklas nutrūksta pasiekus tam tikrą kriterijų (iteracijų skaičių, laiką ir t.t.) [MalGurDyp+89].

Tikimybė priimti žingsnį būna lygi 1 tada kai $\Delta C<0$, arba $e^{\Delta C/T}>random(0.0,1.0)$. Jei ΔC yra neigiamas, kelionės ilgis sumažėjo, vadinasi tikimybė lygi 1. Kitu atveju tikimybė, kad žingsnis bus tinkamas lygi $e^{\Delta C/T}$. Ar žingsnis iš tiesų bus priimtas priklauso nuo to, ar tikimybė bus didesnė už atsitiktinai sugeneruotą skaičių intervale (0.0, 1.0). Tada tikimybės funkcijos pavidalas atrodys taip:



16. pvz. Simuliuoto „atkaitinimo“ tikimybės funkcija.

Galima pastebėti, kad prie didelių temperatūrų dauguma žingsnių yra priimami, nepaisant to, koks žingsnio ilgis. Galiausiai, temperatūrai mažėjant, tik žingsniai į mažėjančias energijos sritis (kurių kelias bus trumpesnis) bus priimami.

4.5. Genetiniai algoritmai

Genetiniai algoritmai atlieka tikimybinę paiešką simuliuodami natūralią atranką. Genetiniai algoritmai priklauso vienai iš evoliucinių algoritmų atšakų, o evoliucinių algoritmų idėja buvo įkvėpta C. Darvin'o natūralios atrankos evoliucinės teorijos. Natūralios atrankos teorija teigia, kad gamtoje egzistuoja natūralios optimizacijos procesai, kurie siekia pastovios būsenos. Evoliuciniai algoritmai buvo pristatyti prieš daugiau nei 40 metų, o genetiniai algoritmai pristatyti 1975 m. J. H.

Holland'o. Tačiau tik nuo 90-ųjų šie algoritmai pradėti taikyti keliaujančio pirklio problemai spręsti [Sin12].

Spręsdamas keliaujančio pirklio problemą genetinis algoritmas sukuria atsitiktinę pradinių tėvų populiaciją ir iš eilės kuria tolimesnes populiacijas taip, kad geriausias maršrutas paskutinėje populiacijoje nėra blogesnis už pradinės populiacijos geriausią maršrutą. Genetinio algoritmo pseudo-kodas keliaujančio pirklio problemai spręsti:

- Sukuriama atsitiktinė populiacija.
- Kol nepasiekta ciklo sustabdymo sąlyga:
  - Pasirenkami tėvai iš populiacijos.
  - Iš tėvų yra gaunami vaikai.
  - Vaikai yra paveikiami genetiniu operatoriumi.
  - Išplečiama populiacija prie jos pridedant vaikus.
  - Sumažinama išplėsta populiacija.

17. pvz. Genetinio algoritmo pseudo-kodas [Sin12].

Iš pradžių yra parenkama atsitiktinė pirminė tėvų (grafo viršūnių) populiacija ir įvertinama naudojantis evoliucijos funkcija. Parenkamos grafo viršūnės iš kurių bus gaunami vaikai (viršūnės). Vaikai bus paveikti genetiniu operatoriumi ir pridėti prie populiacijos. Tada kai kurios viršūnės bus panaikintos iš populiacijos siekiant išlaikyti pastovų populiacijos dydį. Viena ciklo iteracija yra vadinama karta. Po tam tikro kartų skaičiaus bus gautas optimaliausias maršrutas. Yra du genetiniai operatoriai, kurie pagerina iš tėvo gautų vaikų maršrutus:

- *Mutacija* (angl. *Mutation*).
- *Perėjimas* (angl. *Crossover*).

Mutacijos metu yra atliekami maži pakeitimai vaiko maršrutui. *Perėjimo* metu iš tėvų *A* ir *B* maršrutų kraštų $E(A)$ ir $E(B)$ yra suformuojamas rinkinys kraštų su trumpesniais maršrutais. Šie trumpesni maršrutai yra implementuojami teviniuose maršrutuose, todėl jie generuoja geresnius vaikinius maršrutus [Sin12].

Nors šio mokslinio darbo metu genetiniai algoritmai nebuvo realizuoti, jie gali būti pritaikomi ieškant optimaliausių parametrų skruzdėlių kolonijos algoritme.

5. Praktinė dalis

Atliekant praktinę mokslo tiriamojo darbo dalį, buvo suprogramuoti 3 algoritmai:

- Pilnas perrinkimas;
- Skruzdėlių kolonijos algoritmas;
- Simuliuoto „atkaitinimo“ algoritmas.

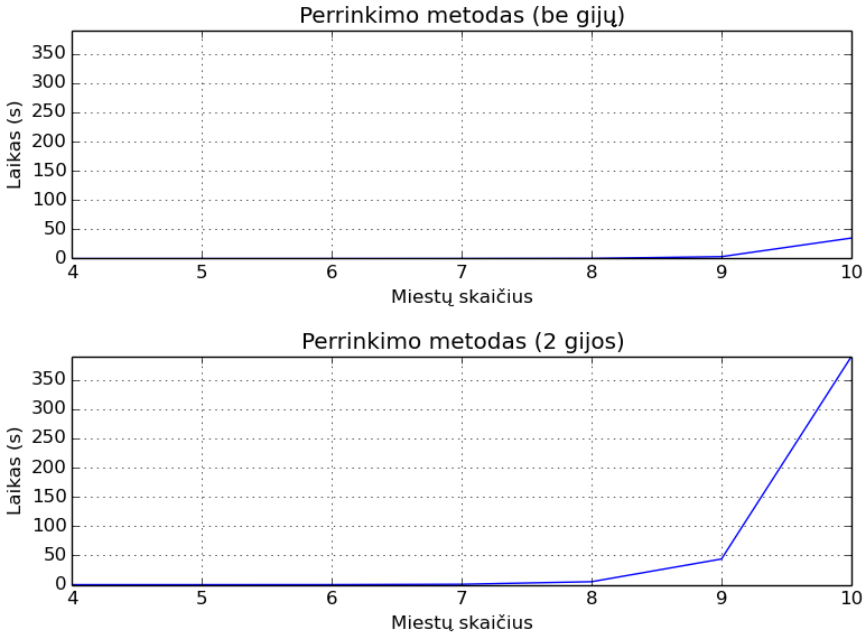
Siekiant rasti optimalius skruzdėlių kolonijos algoritmo parametrus, buvo pasitelkta simuliuoto „atkaitinimo“ algoritmu.

Praktinė mokslo tiriamojo darbo dalis buvo įgyvendinta Python 2.7 programinėje kalboje.

5.1. Pilnas perrinkimas

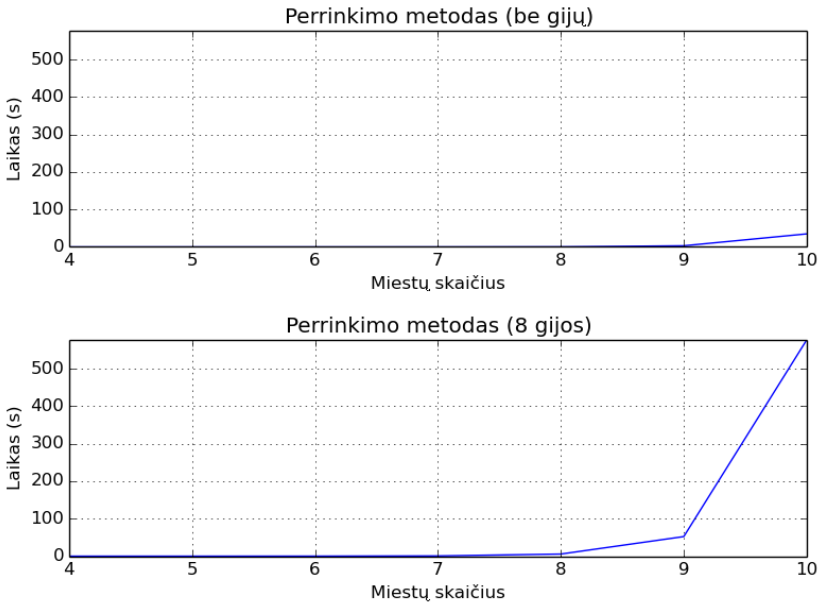
Kadangi pilno perrinkimo algoritmas spręsdamas 10 atsitiktinai sugeneruotų miestų keliaujančio pirklio problemą jau užtruko virs 40 sekundžių, buvo siekiama paspartinti šį algoritmą. Vienas iš būdų šio metodo paspartinimui yra išlygiagretinimas.

Buvo pasirinkta pilno perrinkimo metodo skaičiavimus išskirstyti į gijas, rezultatai matomi 18 ir 19 pavyzdyje.



18. pvz. Pilno perrinkimo rezultatai atsitiktinai generuotiems miestam nenaudojant gijų (viršutinis paveikslėlis) ir naudojant 2 gijas (apatinis paveikslėlis).

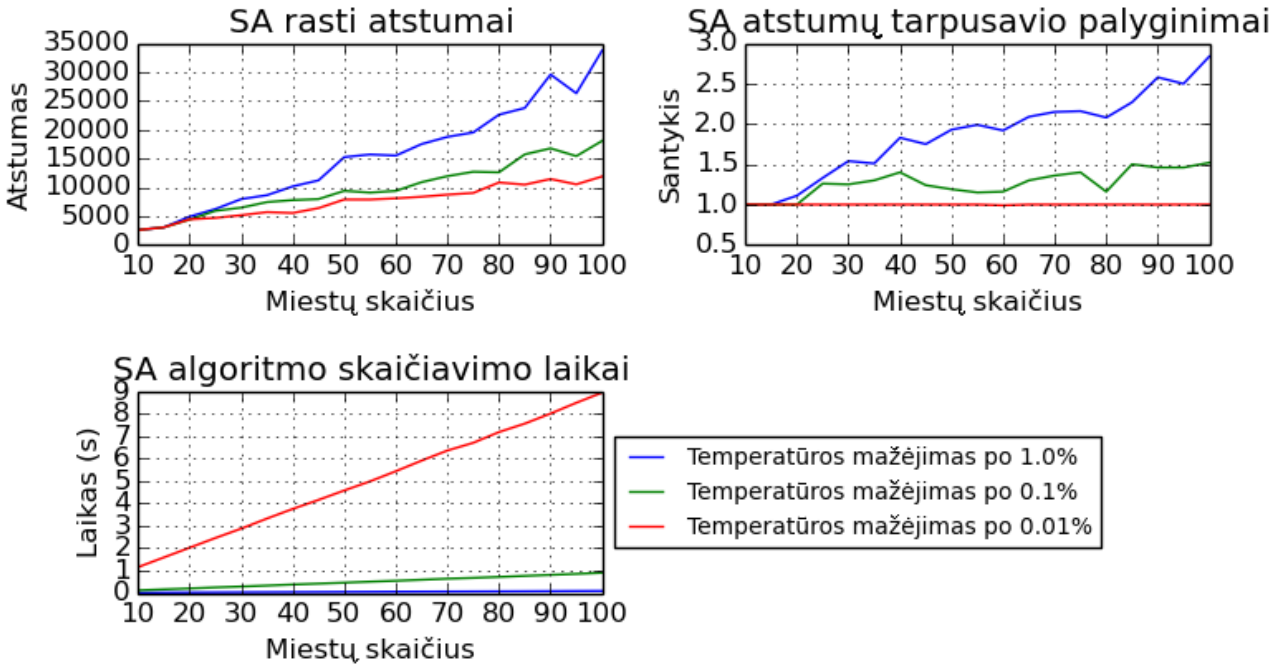
Kaip matyti iš atliktų skaičiavimų grafikų – procesoriaus darbo padalijimas gijoms nepasiteisino. Esant 8 ir daugiau miestų skaičiavimo laikas pradeda kilti eksponentiškai naudojant gijas, taip pat kuo daugiau gijų, tuo spartesnis kilimas.



19. pvz. Pilno perrinkimo rezultatai atsitiktinai generuotiems miestam nenaudojant gijų (viršutinis paveikslėlis) ir naudojant 8 gijas (apatinis paveikslėlis).

5.2. Simuliuoto „atkaitinimo“ algoritmas

Siekiant įvertinti simuliuoto „atkaitinimo“ algoritmą, sprendžiant keliaujančio pirklio problemą, buvo sugeneruota 10 atsitiktinių grafų, nuo 10 iki 100 miestų. Vykdant algoritmą buvo keičiamos jo pradinės sąlygos – algoritmo vienos iteracijos temperatūros kritimo sparta (1.0%, 0.1%, 0.01%).



20. pvz. Atsitiktinai generuotų miestų keliaujančio pirklio problemos sprendimas naudojantis simuliuoto „atkaitinimo“ (SA) algoritmu. Kiekvieno grafo problema sprendžiama keičiant temperatūros mažėjimo spartą.

Simuliuoto „atkaitinimo“ algoritmo rezultatuose (20 pvz.) pateikti 3 grafikai. Apatiniame matyti algoritmo skaičiavimo laikas, dešinėje viršuje – apskaičiuoto keliaujančio pirklio problemos trumpiausių maršrutų ilgiai, o viršuje kairėje pusėje – apskaičiuotų atstumų santykis su trumpiausiu maršrutu.

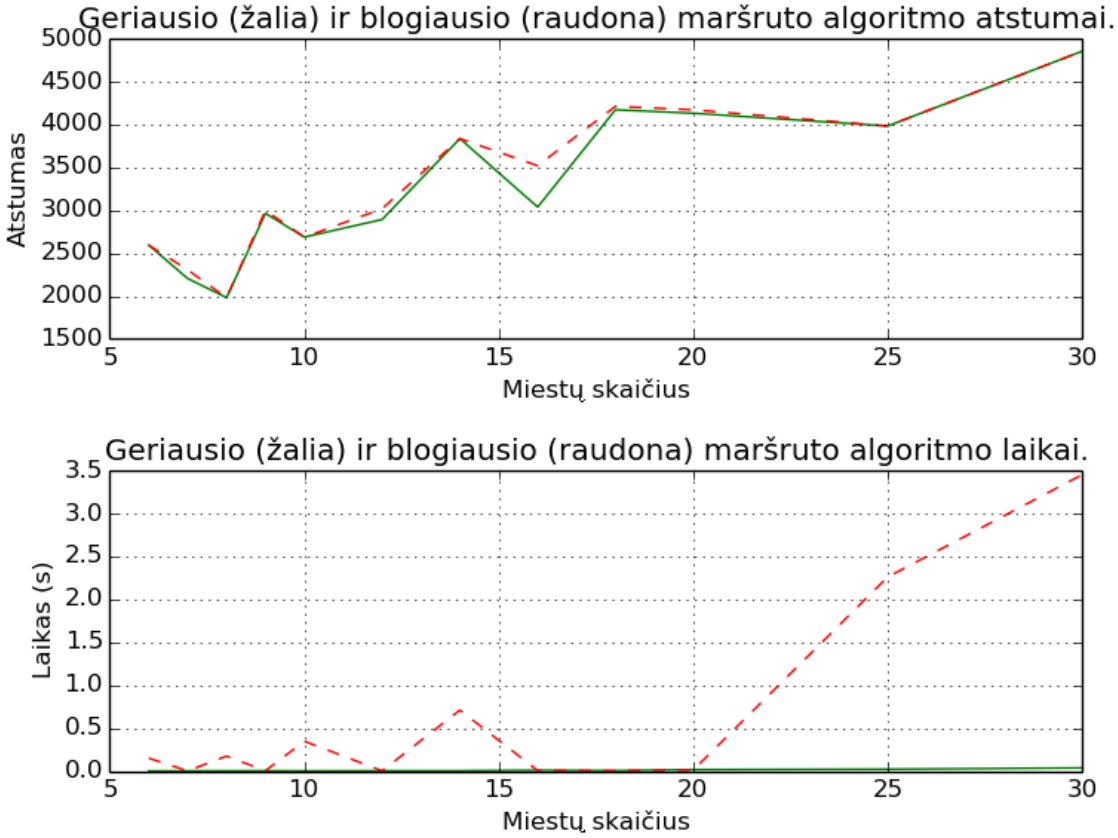
Kuo lėtesnis temperatūros mažėjimas, tuo randamas atstumas yra trumpesnis, tačiau taip pat gerokai ilgesnis ir algoritmo veikimo laikas. Tačiau matoma, kad veikimo laikas didėja kur kas sparčiau nei randamų atstumų santykiai. Taip pat galima pastebėti, kad iki 15 miestų su visomis temperatūros mažėjimo spartomis algoritmo rasti atstumai buvo praktiškai vienodi, o iki 20 miestų su 0.1% ir 0.01% mažėjimo spartomis.

5.3. Skruzdėlių kolonijos algoritmas

Keliaujančio pirklio problemai spręsti buvo naudojamas ir skruzdėlių kolonijos algoritmas. Algoritmui yra nurodomi 4 parametrai: skruzdėlių skaičius, iteracijų skaičius, α (feromono nykimo parametras) ir β (santykinė svarba tarp feromono ir kelio ilgio).

Kadangi pasirinkti optimalią 4 parametrų kombinaciją nėra lengva, buvo naudojamas simuliuoto „atkaitinimo“ algoritmas ieškant optimalių parametrų. Buvo atsitiktinai sugeneruoti grafai, nuo 6 iki 30 miestų ir naudojant 1.0% temperatūros mažėjimo spartą ieškoma optimaliausių parametrų (parametrų su kuriais skruzdėlių kolonijos algoritmas rastų trumpiausią keliaujančio pirklio problemos maršrutą per trumpiausią laiką).

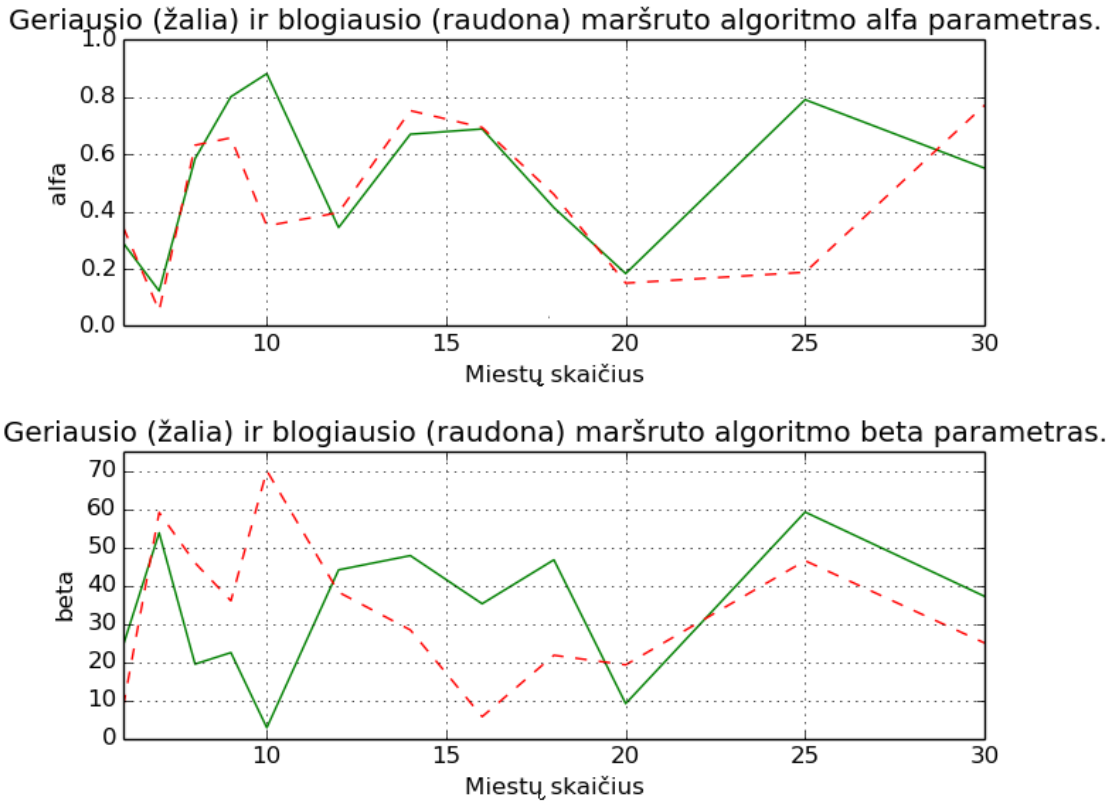
Skruzdėlių kolonijos algoritmo geriausių (žali) ir blogiausių (raudoni) rezultatų atstumai ir veikimo laikai pavaizduoti 21, α ir β parametrai - 22, o skruzdėlių ir iteracijų skaičius 23 pavyzdžio grafikuose.



21. pvz. Skruzdėlių kolonijos algoritmo optimalių parametrų paieška naudojantis simuliuoto „atkaitinimo“ algoritmu.

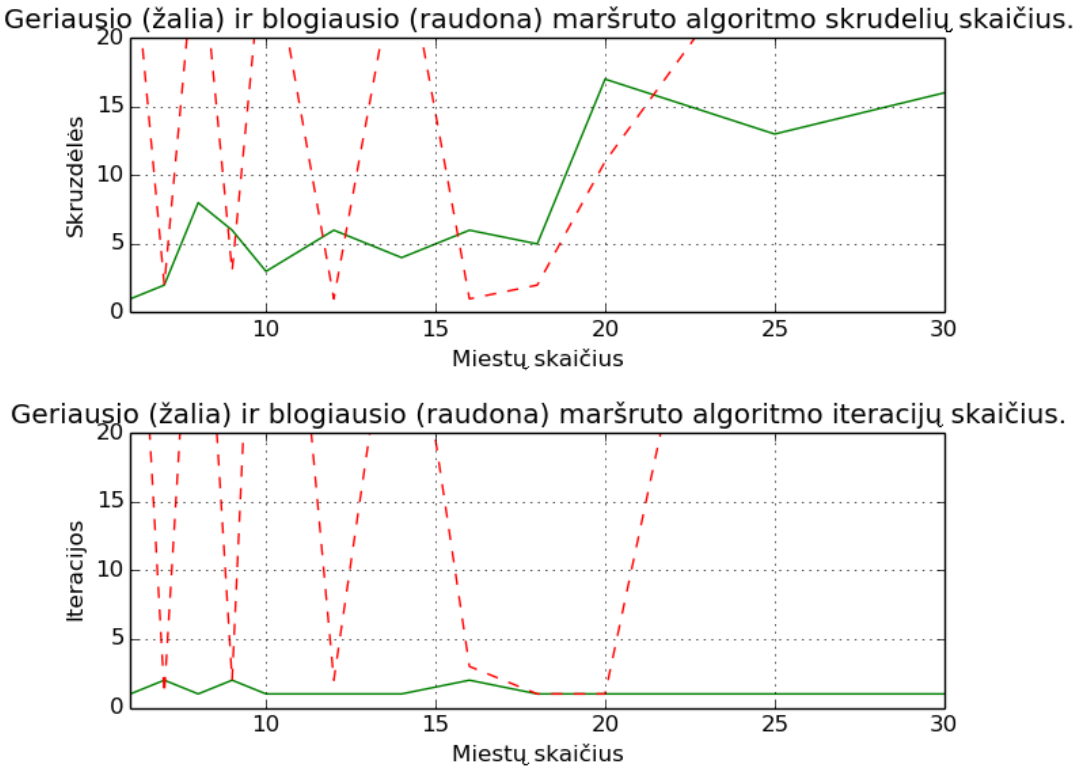
Iš 21 pavyzdžio grafikų matyti, kad algoritmas praktiškai visada randa vienodą trumpiausią atstumą, tik skaičiavimo laikas, priklausomai nuo parametrų, skiriasi gerokai tarp trumpiausio ir ilgiausio laikų.

Analizuojant 22 pavyzdžio grafikų α (viršutinis grafikas) ir β (apatinis grafikas) parametrus aiškų dėsningumą tarp geriausio ir blogiausio skruzdėlių kolonijos algoritmo sprendinio įžvelgti sunku.



22. pvz. Skruzdėlių kolonijos algoritmo optimalių parametrų paieška naudojantis simuliuoto „atkaitinimo“ algoritmu.

Tačiau analizuojant 23 pavyzdžio skruzdėlių (grafikas viršuje) ir iteracijų skaičiaus (grafikas apačioje) grafikus galima daryti tam tikras išvadas. Matyti, kad algoritmas veikia prasčiau kai skruzdėlių ir iteracijų skaičius yra maksimalūs (įtakoja ilgą veikimo laiką), arba lygūs vienetui (įtakoją ilgesnį maršrutą). Algoritmas gražina geriausią rezultatą kai iteracijų skaičius yra artimas 1, o skruzdėlių skaičius didėja, didėjant miestų skaičiui.

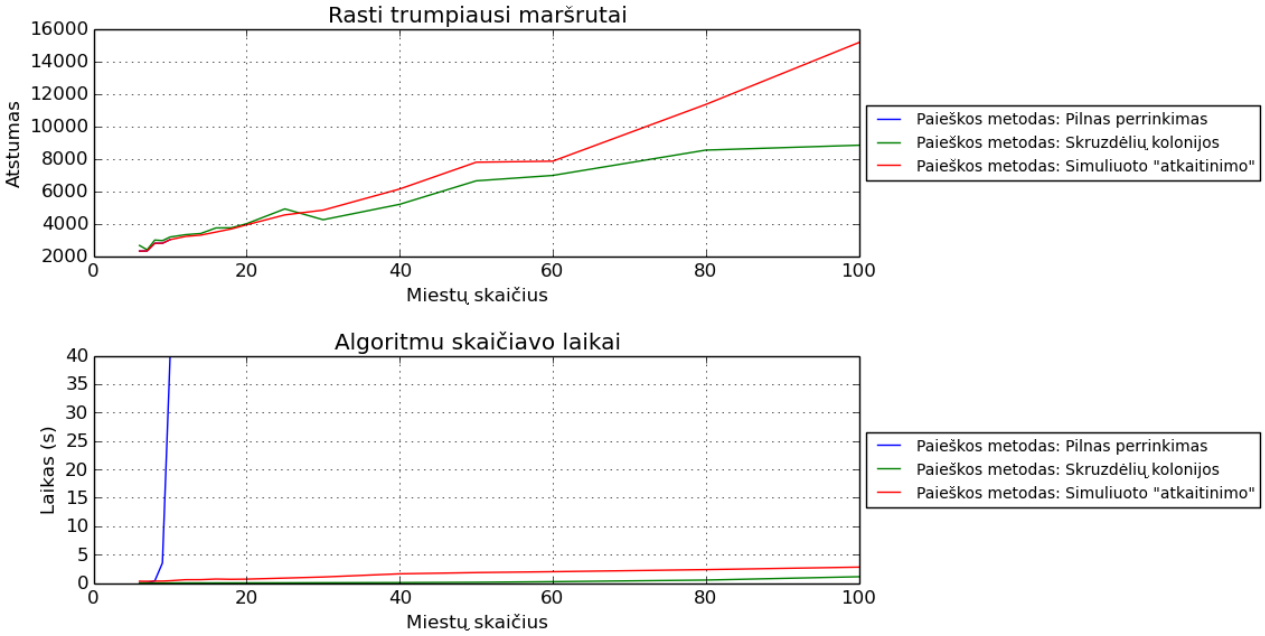


23. pvz. Skruzdėlių kolonijos algoritmo optimalių parametrų paieška naudojantis simuliuoto „atkaitinimo“ algoritmu.

5.4. Visų algoritmų palyginimas

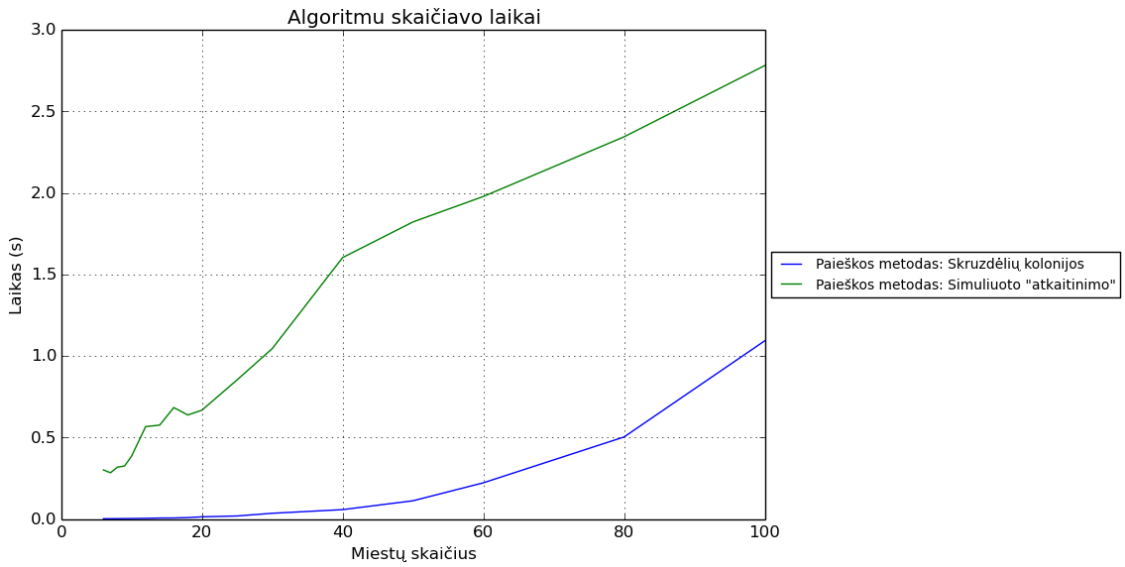
Pilnas perrinkimas (iki 10 miestų), skruzdėlių kolonijos ir simuliuoto „atkaitinimo“ algoritmai buvo palyginti tarpusavyje sprendžiant keliaujančio pirklio problemą. Buvo sprendžiama tiek žinoma (26 pvz.), tiek atsitiktinai sugeneruota problema.

Atsitiktinai generuotų grafų nuo 6 iki 100 miestų keliaujančio pirklio problemų sprendimo rezultatai pavaizduoti 24 ir 25 pavyzdžio grafuose.



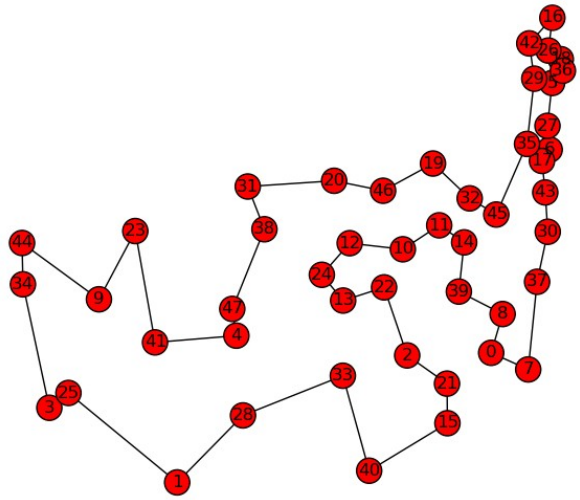
24. pvz. Atsitiktinai generuotų keliaujančio pirklio problemų sprendimų pilno perrinkimo, skruzdėlių kolonijos (10 skruzdėlių, 1 iteracija, $\beta=20$, $\alpha=0.5$) ir simuliuoto „atkaitinimo“ (pradinė temperatūra – 1000, temperatūros mažėjimo sparta – 0.05%) algoritmų rezultatai (trumpiausi maršrutai - viršutiniame grafe, skaičiavimo laikai - apatiniame).

Iš 24 pavyzdžio grafikų galima daryti išvadą, kad iki 10 miestų dydžio grafuose (kol yra naudojamas pilnas perrinkimas), visi algoritmai randa panašaus ilgio trumpiausią maršrutą. Simuliuoto „atkaitinimo“ ir skruzdėlių kolonijos algoritmai iki 25 miestų dydžio grafuose randa panašaus ilgio trumpiausius maršrutus, toliau didėjant miestų skaičiui, skruzdėlių algoritmas randa trumpesnius maršrutus. Taip pat atsižvelgiant ir į 25 pavyzdžio grafiką galima teigti, kad pats greičiausias algoritmas yra skruzdėlių kolonijos.



25. pvz. Atsitiktinai generuotų keliaujančio pirklio problemų sprendimų skruzdėlių kolonijos (10 skruzdėlių, 1 iteracija, $\beta=20$, $\alpha=0.5$) ir simuliuoto „atkaitinimo“ (pradinė temperatūra – 1000, temperatūros mažėjimo sparta – 0.05%) algoritmų rezultatų skaičiavimo laikai.

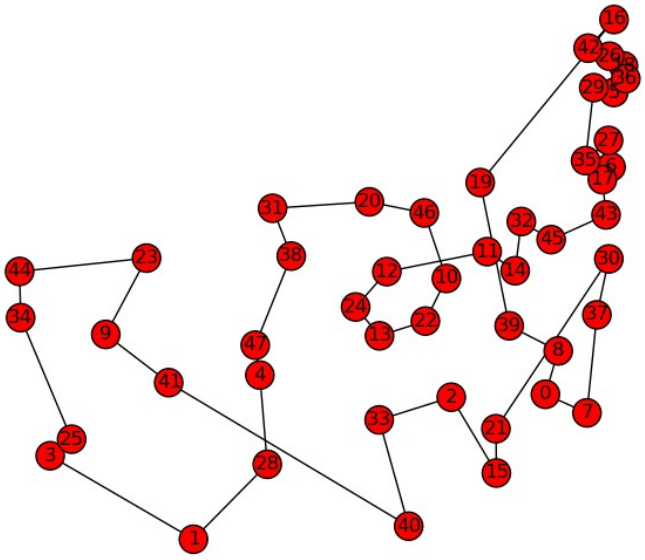
Simuliuoto „atkaitinimo“ ir skruzdėlių kolonijos algoritmai taip buvo palyginti sprendžiant žinomo sprendinio 48 JAV valstijų sostinių (trumpiausias maršrutas - 33551) keliaujančio pirklio problemą (26 pvz).



26. pvz. 48 JAV valstijų sostinių keliaujančio pirklio problemos trumpiausias maršrutas – 33551.

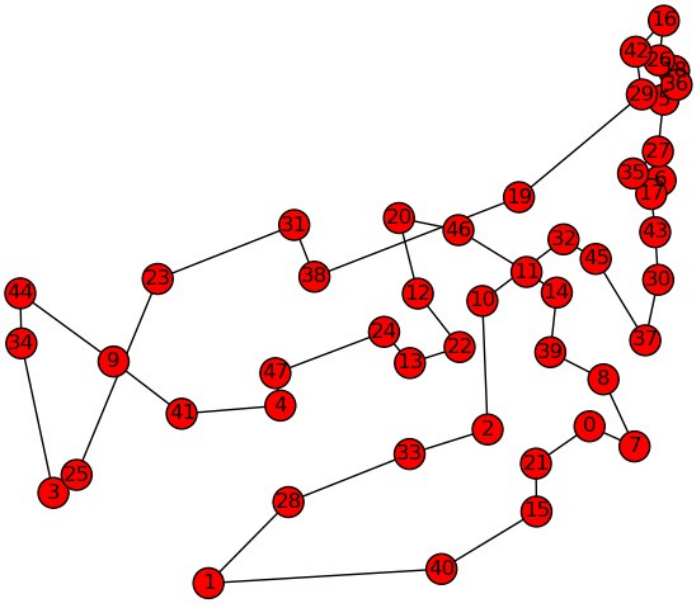
Atlikus 10 pakartotinių skaičiavimų, rezultatus galima pamatyti 29 pavyzdžio lentelėje.

Geriausias 48 JAV valstijų sprendinys skruzdėlių kolonijos algoritmu pateiktas 27 pvz:



27. pvz. 48 JAV valstijų sostinių keliaujančio pirklio skruzdėlių kolonijos algoritmo rezultatas – 39264 atstumas per 0.85 sekunės (10 skruzdėlių, 1 iteracija, $\beta=20$, $\alpha=0.5$).

Geriausias 48 JAV valstijų sprendinys simuliuoto „atkaitinimo“ algoritmu pateiktas 28 pvz:



28. pvz. 48 JAV valstijų sostinių keliaujančio pirklio simuliuoto atkaitinimo algoritmo rezultatas – 39659 atstumas apskaičiuotas per 1.31 sekundės (pradinė temperatūra - 1000, temperatūra kiekvieną iteraciją mažėja per 0.05%).

ACS		SA	
Laikas	Atstumas	Laikas	Atstumas
1.183011	39264	1.289429	39947
0.856368	39264	1.298449	47471
0.850651	39264	1.319927	39659
0.837202	39264	1.32696	41048
0.854027	39264	1.314563	47412
0.851687	39264	1.285553	43584
0.837356	39264	1.289801	47274
0.834112	39264	1.328329	45998
0.919217	39264	1.491662	42440
0.865176	39264	1.331333	43132

29. pvz. 48 JAV valstijų sostinių keliaujančio pirklio skruzdėlių kolonijos algoritmo (10 skruzdėlių, 1 iteracija, $\beta=20$, $\alpha=0.5$), ir simuliuoto „atkaitinimo“ algoritmo (pradinė temperatūra – 1000, temperatūros mažėjimo sparta – 0.05%) rezultatai.

Iš 29 pavyzdžio lentelės galima daryti išvadą, kad geriausiu atveju skruzdėlių kolonijos ir simuliuoto „atkaitinimo“ algoritmas randa beveik panašų trumpiausią atstumą, kuris 17.5% ilgesnis nei idealaus trumpiausias atstumas. Tačiau simuliuoto „atkaitinimo“ algoritmas yra vykdomas vidutiniškai 47% ilgiau, taip pat jo randami trumpiausi atstumai daugeliu atvejų yra ilgesni nei gauti naudojant skruzdėlių kolonijos algoritmą.

6. Ateities darbai

Vietoje atsitiktinai generuotų miestų, implementuoti Google žemėlapius, kad skaičiuojamas trumpiausias maršrutas būtų su tikrais duomenimis.

Naudojant genetinius algoritmus rasti skruzdėlių kolonijos algoritmo optimalių skruzdėlių ir iteracijų skaičiaus, bei α ir β parametrų priklausomybę nuo miestų skaičiaus.

Patobulinti skruzdėlių kolonijos algoritmą naudojant A* ir Floyd-Warshall algoritmus, kad ACS algoritmas galėtų spręsti keliaujančio pirklio problemą ne tik pilname grafe.

7. Išvados

Sprendžiant iki 10 miestų keliaujančio pirklio problemą, pilnas perrinkimas, skruzdėlių kolonijos (ACS) ir simuliuoto „atkaitinimo“ (SA) algoritmai randa beveik panašaus ilgio trumpiausius maršrutus. Sprendžiant iki 25 miestų keliaujančio pirklio problemą, ACS ir SA algoritmai taip pat randa panašaus ilgio trumpiausius maršrutus.

Sprendžiant keliaujančio pirklio problemas kai miestų skaičius didesnis nei 25, ACS randa trumpesnį maršrutą nei SA algoritmas. Taip pat pastebėta, kad SA algoritmo rezultatai vienodoms sąlygoms, tarpusavyje skiriasi kur kas labiau nei ACS algoritmo.

Visais atvejais pilnas perrinkimas yra lėčiausias, o ACS yra greičiausias algoritmas. Todėl galima teigti, kad sprendžiant įvairaus sudėtingumo keliaujančio pirklio problemas, iš minėtų 3 algoritmų, pats optimaliausias yra skruzdėlių kolonijos algoritmas.

8. Literatūros sąrašas

[AinSal12]	A. Aini, A. Salehipour, Speeding up the Floyd–Warshall algorithm for the cycled shortestpath problem, 2012
[BocBla08]	Bob Bockholt, Paul E. Black, "adjacency-list representation", in Dictionary of Algorithms and Data Structures, 2008, URL: http://www.nist.gov/dads/HTML/adjacencyListRep.html, Žiūrėta 2014-05-24
[Caz06]	T. Cazenave, Optimizations of data structures, heuristics and algorithms forpath-finding on maps, 2006
[CMI14]	Clay matematikos instituto interneto puslapis, URL: http://www.claymath.org/millenium-problems/p-vs-np-problem Žiūrėta 2014-06-15
[DorGam97]	M. Dorigo, L. M. Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, 1997
[God04]	Wayne Goddard, Intrduction to Algorithms, 2004
[LeiLouChr07]	R. Leigh, S. J. Louis, Chris Miles, Using a Genetic Algorithm to Explore A*-like PathfindingAlgorithms, 2007
[Man01]	Eugenijus Manstavičius, Grafų teorija, 2001, URL: http://www.mif.vu.lt/ttsk/bylos/man/files/grmag.pdf, Žiūrėta 2014-05-23
[MalGurDyp+89]	M. Malek, M. Guruswamy, M. Dypana, H. Owens, Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem, 1989
[Mcq09]	William D McQuain, Data Structures & Algorithms, 2009, URL: http://courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf, Žiūrėta 2014-05-05
[Ruo08]	K. Ruohonen, Graph theory, 2008
[Sin12]	L. Sing, Optimal tree for Genetic Algorithms in the Traveling SalesmanProblem (TSP), 2012
[WolRao00]	S. Wolfman, R. Rao, URL: http://courses.cs.washington.edu/courses/cse373/01sp/Lect24_2up.pdf, 2000, Žiūrėta 2014-05-05