VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Algoritmų analizė

Projektinis darbas

Užduotis nr. 2: Keliaujančio pirklio uždavinio sprendimas šakų ir rėžių metodu

Tomas Giedraitis
VU MIF Informatika
3 kursas 3 grupė

Vilnius
2021

Turinys

1. Darbo tikslas

Realizuoti paieškos su grįžimu algoritmą optimaliam keliaujančio pirklio maršrutui rasti ir ištirti šio algoritmo sudėtingumą.

Taip pat, realizavus algoritmą, pritaikyti jį Lietuvos miestų ir rajonų centrų atstumų lentelei[1].

2. Uždavinio formuluotė

Keliaujančio pirklio uždavinys (KPU) skamba taip:

Turint miestų sąrašą ir atstumus tarp jų, koks yra greičiausias įmanomas maršrutą, kuriuo aplankomas kiekvienas miestas lygiai vieną kartą ir pabaigoje grįžtama į pradinį miestą, iš kurio buvo startuota?

Šis uždavinys pirmą kartą buvo matematiškai suformuluotas W. R. Hamiltono[2] kaip grafų teorijos uždavinys su šia sąlyga:

Duota: Orientuotas svorinis grafas G, turintis n viršūnių ir m lankų.
Rasti: Minimalaus svorio Hamiltono ciklą grafe G (optimalų keliaujančio pirklio maršrutą)

Ši uždavinio formuluotė ir yra naudojama šiame darbe, realizuojant algoritmą KPU spręsti. Egzistuoja įvairūs šio uždavinio sąlygų variantai bei generalizacijos[3].

KPU problema gali būti simetrinė, kai atstumai tarp miestų yra vienodi abejomis kryptimis (neorientuotas grafas), arba asimetrinė, kai jie skiriasi (orientuotas grafas). Šiame darbe buvo nagrinėjamas bendriausias uždavinio atvejis – grafas yra orientuotas (asimetrinis atvejis), o atstumai netenkina trikampio taisyklės $d_{ij} \leq d_{ik} + d_{kj}$. Tokia situacija gali pasitaikyti ir praktiškai, kai atstumas į priekį nuo vieno miesto iki kito skiriasi nei keliaujant atgal, dėl vienos krypties kelių ar kitų priežasčių. Dėl to ir grafe viršūnes jungiančias briaunas vadiname lankais, nes jos turi kryptį, ir žymime atskiromis sekomis (i, j)ir (j, i) .

2.1. Keliaujančio pirklio uždavinio apžvalga

Keliaujančio pirklio uždavinys yra svarbus kompiuterių mokslo teorijoje ir operacijų tyrimo srityje. Šis uždavinys yra NP–pilnas[4]. Tai bene labiausiai išgarsėjęs kombinatorinis uždavinys, kuriam daug metų buvo bandoma surasti polinominio sudėtingumo algoritmą arba įrodyti, kad toks algoritmas neegzistuoja. Deja, niekam nepavyko padaryti nei viena, nei antra[5].

KPU yra naudojamas kaip testavimo etalonas daugumoje optimizavimo metodų[6]. Nors tai yra sunkus uždavinys skaičiavimo prasme, daug euristikų ir tikslių algoritmų jam yra žinoma. Tam tikrus iš jų panaudojus, galima gauti tikslų 10 000 miestų sprendimą ir net 1 mln. apytikslį maža paklaida, siekiančia tik 1%.

Šis uždavinys, su standartine maršruto tarp miestų formuluote, turi pritaikymų planavime, logistikoje, ir mikroschemų gamyboje. Šiek tiek modifikuotas jis pasitaiko kaip tarpinis uždavinys daugumoje sričių, pvz. DNR sekos nustatyme. Šiuose pritaikymuose miesto konceptas gali reprezentuoti įvairius dalykus – litavimo taškus, verslo klientus, DNR fragmentus, o atstumai tarp jų – kelionės trukmę, kainą, panašumo matą tarp DNR fragmentų. KPU turi pritaikymų ir astronomijoje, sprendžiant optimalios kontrolės uždavinį. Daugumoje pritaikymų dažnai galioja papildomi uždavinio limitai, pvz. baigtinis resursų skaičius ar limituotas laiko intervalas.

3. Realizuotas algoritmas

KPU uždaviniui spręsti buvo realizuotas šakų ir rėžių (angl. *branch and bound*) metodo algoritmas[7]. Tai yra paieškos su grįžimu algoritmas, kai optimizuojama tikslo funkcija *Cost* (šiuo atveju – nurodanti gauto maršruto bendrą ilgį), ir yra žinoma, kaip sprendinių paieškos medžio pomedžiuose gaunamų sprendinių kainą iš anksto aprėžti iš apačios aprėžiančios funkcijos *Bound* pagalba[5]. Sprendinių paieškos medis buvo pasirinktas dvejetainis, jo išsišakojimo į naujas šakas būdas buvo naudojamas apskaičiuojant rėžio pokyčius (aprašyta šaltinyje[5]), o medžio viršūnių perrinkimui buvo naudojama BeFS (geriausios viršūnės prioritetinio pasirinkimo strategija, angl. *best-first search*), kuomet kiekvieną kartą pasirenkama ta medžio viršūnė, kurios rėžis yra mažiausias iš dar nenagrinėtų medžio viršūnių.

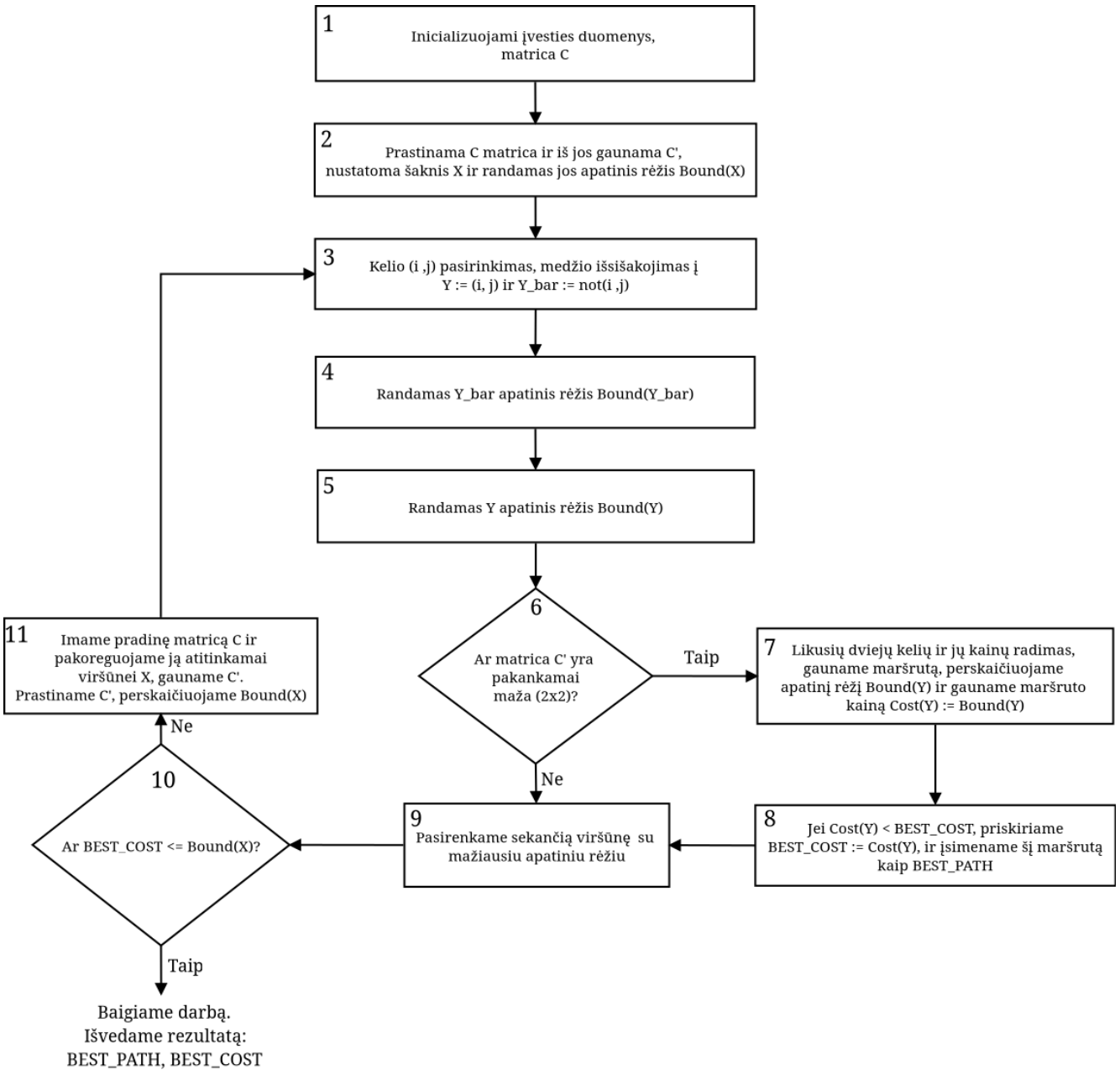
3.1. Algoritmo kintamieji

Algoritmo įvestis yra orientuotas svorinis grafas *G,* kur *n* – viršūnių skaičius, *m* – lankų skaičius, ir kiekvienas lankas turi savo svorį w_i, $i \in \{1, m\}$. Kai egzistuoja keliai iš visų viršūnių į visas kitas, ir iš jų kiti keliai vėl atgal į jas (gali sutapti kelias pirmyn ir atgal, bet bendruoju atveju leidžiama, kad jie galėtų skirtis), tai tuomet lankų skaičius yra maksimalus įmanomas, ir yra lygus $m = 2\frac{n(n-1)}{2} = n(n-1)$, priklausomai nuo *n* viršūnių, ir orientuotas grafas tuomet yra pilnasis. Jei kažkurio kelio (lanko) nėra, o algoritmas vis tiek tikrina visus įmanomus kelius, tai laikysime, kad tokio kelio svoris yra lygus 0.

3.2. Algoritmo aprašymas

Šakų ir rėžių algoritmas KPU sprendimui taip pat išsamiai aprašytas S. Goodman ir S. Hedetniemi[8] knygoje, pasinaudojant šia algoritmo schema:

Schema 1: Šakų ir rėžių algoritmas (schema išversta į lietuvių kalbą)



Algoritmas buvo realizuotas atsižvelgiant į šią schemą, ir atitinkamai programoje padalintas į atskiras kodo dalis – blokus, pažymėtus schemoje (ir programoje) numeriais nuo 1 iki 11. Ši schema pilnai atitinka parašytą programą, tik kiekvieno bloko įgyvendinimas jau yra specifinis programai.

Išsamiau apie kiekvieną bloką, realizuotą programoje:

[1] Nuskaitomas įvedimas – grafą G aprašanti kvadratinė asimetriška (bendruoju atveju) gretimumo matrica C. Kadangi dėl uždavinio specifikos grafas aiškiai neturi kilpų, nes nėra keliaujama iš miesto į jį patį, pagrindinė matricos įstrižainė bus sudaryta iš nulių, tačiau jie yra pažymimi kitaip, pvz. begalybe (arba `None` reikšme programoje), ir ignoruojami kaskart renkantis lanką, o tuo tarpu nuliai algoritme įgauna kitą prasmę. Taip pat C' pažymime šios matricos kopiją, su kuria bus atliekami veiksmai. Tai reikalinga todėl, nes algoritme gali tekti grįžti prie pradinės matricos C (**[9]** bloke, jei

prieš jį buvo **[8]** blokas). Inicializuojame sprendinių paieškos medį, tuščią viršūnių-kandidačių prioritetinę eilę, geriausios šiuo metu rastos maršruto kainos *BEST_COST* reikšmę nustatome į begalybę, inicializuojame tuščią geriausio šiuo metu rasto maršruto *BEST_PATH* sąrašą.

[2] Prastiname matricą C', iš kiekvienos eilutės atimdami po mažiausią toje eilutėje esantį narį, ir po to tą patį kartojame su visais stulpeliais. Sudėjus visus atėminius (jų bus kiekvienai eilutei po vieną ir kiekvienam stulpeliui po vieną, tačiau kai kurie jų bus lygūs 0), gauname apatinį rėžį *Bound(∅),* čia šiuo atveju ∅ reiškia sprendinių paieškos medžio viršūnę. Priskiriame šią viršūnę kintamajam *X*, *Bound*(*X*) = *Bound(∅).*

[3] Realizuojamas specifinis sprendinių paieškos medžio išsišakojimo į naujas šakas būdas: visiems keliams (*i, j*) su atsiradusiais 0 matricoje apskaičiuojame jų rėžio pokyčius *D*[*i, j*] (pagal šaltinio[5] 58 psl., 3 punktą), ir pasirenkame kelią su didžiausiu rėžio pokyčiu. Tuomet šakojame sprendinių paieškos medį iš viršūnės *X* į viršūnes$Y$ ir $\overline{Y}$(žymima *Y_bar* schemoje). Ypomedyje šis kelias *(i, j*) bus įtrauktas, o $\overline{Y}$– priešingai, šio kelio jame nebus. Pridedame kelią *(i, j*) į mūsų kaupiamą mašrutą, o abi viršūnes Y ir $\overline{Y}$– į galimų viršūnių-kandidačių prioritetinę eilę.

[4] Apskaičiuojame *Bound*($\overline{Y}$) rėžį, prie *Bound(X)* pridėję jau apskaičiuotą *D*[*i,j*].

[5] Vėl prastiname matricą C', ir gautus atėminius pridedame prie *Bound(X),* taip gaudami Yviršūnės apatinį rėžį *Bound(Y).*

[6] Patikriname, ar matrica C' šiuo metu jau yra pakankamai triviali, kad lengvai galėtumėme surasti likusius kelius. Tam žiūrime, ar matrica yra 2×2 išmatavimų.

> **[7]** Jei matrica yra triviali, pagal tai, kokį maršrutą šiuo metu jau esame išsisaugoję, nesunkiai pasirenkame du likusius kelius iš dviejų galimų kombinacijų, pridedame jų kainas, nurodytas matricoje C', prie jau turime rėžio *Bound(Y),* ir gauname maršruto kainą *Cost*(*Y*) := *Bound(Y).*
>
> **[8]** Tikriname, ar maršruto kaina yra mažesnė už jau turimą *BEST_COST*. Jei taip, įsimename šią naujai gautą kainą bei išsaugome naujai rastą maršrutą kaip *BEST_PATH*.

[9] Nepriklausomai nuo to, ar prieš tai buvo **[7]** ir **[8]** blokai, ieškome kitos geriausios viršūnės iš galimų kandidačių, ir pasirenkame ją, pažymėdami ją *X*. Šioje realizacijoje medžio viršūnių perrinkimui buvo naudojama BeFS strategija. Svarbu paminėti, kad galima pasirinkti ir tą pačią viršūnę *Y,* jeigu jos apatinis rėžis mažiausias, ir taip iš tiesų dažniau ir nutinka. Tokiu atveju *X* kintamajam priskiriam *Y*, ir toliau jau vyks išsišakojimas iš jos.

[10] Patikriname, ar pasirinktos viršūnės apatinis rėžis *Bound(X)* yra ne mažesnis už geriausią turimą maršruto kainą. Jei tai tiesa, tuomet baigiame darbą, ir išvedame atsakymą – *BEST_PATH* maršrutą su *BEST_COST* kaina.

[11] Priešingu atveju, algoritmas tęsia darbą. Jei pasirinkta viršūnė buvo *Y,* tai be jokių papildomų pokyčių vėl einama į **[3]** bloką. Tuo tarpu jei radome kitą geresnę kandidatę, prie jos reikia priderinti matricą. Naujai viršūnei *X* nustatome matricą C', jos atitinkančią einamąjį maršrutą. C' gaunama pakoregavus pradinę matricą *C* pagal šiuos žingsnius:

(1) Surandame visus kelius *(i, j),* įeinančius į *X* viršūnę turintį pomedį, ir išbraukiame *i*-ąsias eilutes ir *j*-uosius stulpelius matricoje C,

(2) Surandame visus kelius *not(i, j)*, kurie griežtai neįeina į dabartinį maršrutą, ir matricoje *C* *j*-osios eilutės *i*-ojo stulpelio reikšmę pakeičiame į begalybę.

Šiems žingsniams atlikti programoje be einamojo maršruto dar saugomas ir vis atnaujinamas sprendinių paieškos medis.

Taigi, tęsiant **[11]** bloką toliau, susumuojame visų įėjusių kelių *(i, j)* kainą, nurodytą pradinėje matricoje *C*. Tada prastiname naujai gautą matricą C', ir jos atėminius sudedame su ką tik prieš tai gauta suma, tokiu būdu gaudami perskaičiuotą dabartinės viršūnės *X* rėžį *Bound(X)*. Dabar jau galime grįžti į **[3]** bloką ir toliau vykdyti algoritmą.

Taip pat, dar yra keli aspektai, kurie neatsiskleidžia šioje schemoje ir aprašyme, bet buvo įgyvendinti programoje, užtikrinant korektišką algoritmo veikimą. Einamojo maršruto saugojimas programoje vyko tokiu būdu, kad, kaskart pridėjus naują kelią, buvo tikrinama, ar galima jį pridėti prie vieno iš jau turimų dalinių maršrutų. Tokiu būdu, prieš pridedant kelią, atsiranda galimybė nesunkiai patikrinti, ar jis, nors lyg ir pasirinktas pagal išsišakojimo procedūrą, nesudaro tarpinių ciklų. Jeigu kelias netinka, tuomet reikėtų rinktis kitą kelią, kiek galima labiau tinkamesnį. Todėl programoje **[3]** bloke pradžioje buvo įsimenami visi keliai, kurių apatinių rėžių padidėjimas buvo vienodas, ir tikrinami iš eilės dėl tarpinių ciklų sudarymo. O jeigu nė vienas iš jų netiko, kartelė sumažinama per vienetą, ir tikrinama, ar ją patenkina kuris nors iš likusių kelių. Blogiausiu atveju, kaip buvo praktiškai pastebėta, gali nė vienas kelias netikti. Tokiu atveju, esant tarpinių ciklų neišvengiamybei, buvo įvestas papildomas veiksmas, neatvaizduotas schemoje – jei **[3]** bloke taip ir nebuvo pasirinktas nė vienas kelias, tuomet iš viršūnių-kandidačių sąrašo pašalinama einamoji dar kol kas neišsišakojusi viršūnė *X*, ir ji priskiriama kintamojo *Y* reikšmei. Tuo tarpu naujai *X* reikšmei parenkame sekančią viršūnę kandidatę, ir pereiname tiesiai į **[11]** bloką. Tokiu būdu *X* ir *Y* viršūnės bus skirtingos, o viršūnių-kandidačių sąrašas tikrai bus netuščias, nes antraip tai būtų programos pradžia, o jos metu nesusidarytų tokios situacijos su daliniais ciklais. Taigi tuomet bus pilnai įvykdytas **[11]** blokas, ir algoritmas tęs darbą toliau, o prie viršūnės, privedusios prie aklavietės, jis jau niekaip negalės sugrįžti. Vienintelis nepaminėtas dalykas liko tai, kad atmetus kelią dėl jo sukuriamo dalinio ciklo, sekančioje iteracijoje jis galimai ir vėl bus laikomas geriausiu, ir vėl bus atmestas. Siekiant išvengti tokio pasikartojimo, nustatome jo reikšmę matricoje į begalybę, nes tvirtai žinome, kad einamajame pomedyje jo būti negalės.

3.3. Programa ir jos kintamieji

Algoritmas realizuotas *Python* programavimo kalba.

Atitinkamai pagal 1-ą schemą atskiri kodo blokai realizuoti funkcijomis

```
block_1()
block_2()
...
block_11()
```

Pagrindinis vykdymo ciklas kviečia šiuos blokus iš eilės, pagal schemą. Rombais schemoje pažymėti **[6]** ir **[10]** blokai reiškia išsišakojimą. Tokiu atveju kodo bloką realizuojanti funkcija grąžina reikšmę `True` arba `False`, pagal tai vykdomas atitinkamas sekantis blokas. Programa sustoja ir pateikia atsakymą tuomet, kai kreipiantis į **[10]** bloką gaunama `True`. Priešingu atveju ciklas kartojamas. Iteracijų skaičius sekamas kintamuoju `iterations`.

Šios blokus realizuojančios funkcijos neturi parametrų, jos operuoja globaliais programos kintamaisiais, kurie aprašyti žemiau.

Kintamieji, susiję su matrica:

`C` – grafą G apibrėžianti gretimumo matrica $n \times n$ dydžio, realizuota dvimačiu masyvu. Šio kintamojo reikšmė nustatoma vieną kartą pradžioje ir programoje nekinta. Nuliai pakeičiami į reikšmę `None`.

`C_prime` – darbinė matrica `C'`, gauta suprastinus matricą `C`, ir po to pati yra prastinama toliau, kol jos dydis pasiekia 2×2.

`row_map`, `col_map` – masyvai su suprastintos matricos `C_prime` eilučių ir stulpelių numeriais, atitinkančiais pradinės matricos `C` numerius, kadangi programoje jie pradeda skirtis nuo nuoseklių vis kintančios matricos `C'` indeksų.

`i_from, j_to` – šiuo metu nagrinėjamo lanko pradžios ir pabaigos viršūnės.

`max_Dij` – dydis, kuriuo dar padidinamas apatinis rėžis.

Kintamieji, susiję su sprendinių paieškos medžiu:

`X` – nagrinėjama viršūnė, `Y, Y_bar` – išsišakojusios viršūnės `X` vaikai `Y` ir $\overline{Y}$.

`candidate_nodes` – sąrašas, į kurį patenka visos dar neaplankytos šiuo metu esančio paeiškos medžio viršūnės (bevaikės, t.y. lapai), išrikiuotas apatinio rėžio didėjimo tvarka. Naujos viršūnės įterpiamos atitinkamai į savo vietą, išlaikant rikiavimo tvarką.

Kintamieji, susiję su maršrutu:

`current_tour` – dabartinis surastas maršrutas (viršūnių sąrašas), pretenduojantis tapti geriausiu einamuoju maršrutu `best_tour`. Pradžioje šis sąrašas yra tuščias. Geriausia einamoji maršruto kaina žymima best_cost, pradžioje esanti `None`.

4. Programos naudojimo instrukcija

Programa veikia autonomiškai iš vykdymo failo `tsp_branch_bound.py` (A.1 priedas).
`Python versija: 3.9.5`

Programa gali būti paleidžiama trejais režimais:

```
[1] python3 tsp_branch_bound.py input_file [-o OUTPUT_FILE] [-d/-s]
```

kuomet specifikuojamas įvesties failas `input_file` (būtinas argumentas)

```
[2] python3 tsp_branch_bound.py -c CITIES [-a ARCS] [-w MIN MAX] [-r RANDOM_SEED]
    [-o OUTPUT_FILE] [-d/-s]
```

kai įvestis yra atsitiktinai sugeneruota matrica, turinti CITIES viršūnių (būtinas argumentas), kuri gali turėti ARCS lankų, ir kurios svorių reikšmės gali būti atsitiktinai parenkamos iš sveikųjų skaičių intervalo [`MIN; MAX`]. Taip pat yra pasirenkamas argumentas `RANDOM_SEED`, kuriuo nustatomas pseudo atsitiktinio skaičių generavimo pradžios taškas. Tai reikalinga, kai norima, kad tarp kelių programos vykdymų galima būtų gauti tas pačias atsitiktinai sugeneruotas reikšmes. Tai sudaro sąlygas kiekvienam eksperimentui būti pakartojamam. `RANDOM_SEED` reikšmė yra sveikasis skaičius.

Taip pat yra ir trečiasis režimas:

```
[3] python3 tsp_branch_bound.py -c CITIES [-a ARCS] [-w MIN MAX] [-r RANDOM_SEED]
    [-o OUTPUT_FILE] [-t TEST_RUNS] [-s]
```

kuriame `TEST_RUNS` argumentu galima nustatyti, kiek kartų tą patį vykdymą norime pakartoti, ir rezultate programa išveda vidutinį vykdymo laiką. Svarbu paminėti, kad šiuo atveju `RANDOM_SEED` reikš pirmąją reikšmę, nuo kurios pradėti testavimą, ir ji bus vis keičiama, didinant ją vienetu. Pavyzdžiui, jei testavimo režime `RANDOM_SEED` pasirinksime 1, ir `TEST_RUNS` nustatysime į 5, tai bus atlikti 5 vykdymai su `RANDOM_SEED` reikšmėmis 1, 2, 3, 4, 5, tokiu būdu padidinant įvesties duomenų skirtingumą testavimo metu.

Taigi, [1] režime privaloma yra įrašyti įvesties failo pavadinimą, o [2] ir [3] – viršūnių (miestų) skaičių. Papildomų argumentų reikšmės [2] ir [3] režimuose pagal nutylėjimą yra šios:

`ARCS` – maksimalus lankų skaičius, lygus $n(n-1)$ nuo viršūnių.
`MIN, MAX` – minimali svorių reikšmė yra 1, o maksimali – 100.
`RANDOM_SEED` – 0.

Kiekviename variante taip pat dar galimi tokie pagalbiniai argumentai:

`-d, --debug` – programa veikia vizualiu režimu, kuomet kiekvieną iteraciją yra spausdinami visi skirtinguose blokuose atliekami veiksmai. Nesvarbu, ar matrica nuskaitoma iš failo, ar sugeneruota atsitiktinai, `debug` režime ji bus atspausdinta programos išvesties pradžioje. Šis argumentas negali būti naudojamas testavimo režime [3].

`-s, --silent` – sutrumpinti programos išvestį. [1] ir [2] režime vietoj aptikto geriausio maršruto, jo kainos ir iteracijų skaičiaus bei vykdymo laiko programa išspausdins tik vykdymo laiką vienoje eilutėje. [3] režime bus praleista dalis testavimo informacijos (viršūnių, lankų, svorių reikšmės, ir kiti išsamūs pranešimai), ir bus vienoje eilutėje atspausdinta vidutinė programos trukmė, vidutinė vienos iteracijos trukmė ir vidutinis iteracijų skaičius. Šios reikšmės atskiriamos kableliu, kad galima būtų iš karto kopijuoti išvestį pvz. į Excel programą ar .csv formato failą.

`-o OUTPUT_FILE, --output_file OUTPUT_FILE` – galima nukreipti išvestį į nurodytą failą. Ypač patogu naudoti su vizualiu (`debug`) režimu, kuomet išvestis gali užimti daug eilučių. Jei failas jau egzistuoja ir yra netuščias, išvestis įrašoma failo pabaigoje, t.y. failo duomenys nėra prarandami. Jei failas pateiktu pavadinimu neegzistuoja, programa jį sukuria.
`-h, --help` – spausdinamas trumpas programos ir argumentų aprašas.

Programos įvestis ir argumentai yra validuojami, ir, esant neteisingai įvesčiai, spausdinami klaidų pranešimai.

Kartu su vykdomuoju failu pateikiamas ir įvesties failo pavyzdys `input.example.txt`. Jame yra atkomentuota Lietuvos miestų atstumų matrica. Virš jos pateiktos dar kelios 5×5 matricos, ir dėl nedidelio jų dydžio jos puikiai tinka vizualiam režimui, vaizduojančiam visą programos vykdymą nuo pradžios iki pabaigos.

Po `@names` raktažodžiu pažymėtos eilutės surašomi viršūnių (miestų) pavadinimai, kiekvienas naujoje eilutėje. Pavadinimai privalo būti aprašyti faile pirmiau nei matrica. Pagal jų skaičių nustatomas matricos dydis, ir po to įvedama matrica privalo tą dydį atitikti. `@names` eilučių galima ir visai nenaudoti, tuomet įvedama tik matrica, pvz.:

```
[ 0 12  4 32]
[ 1  0 16 28]
[17  4  0 43]
[ 3  9 32 17]
```

Kiekviena matricos eilutė privalo prasidėti „`[`“, o pasibaigti „`]`“ simboliu, o skaičiai tarpusavyje turi būti atskirti tarpais arba kableliais. Matricos dydis bus nustatytas pagal pirmąją eilutę, ir bus reikalaujama atitinkamo eilučių ir stulpelių skaičiaus likusioje įvestyje.

Tuščios eilutės ir papildomi tarpai faile yra galimi, ir yra ignoruojami. Įvesties faile galima įtraukti ir komentarus, jie prasideda ‘#’ ženklu kiekvienoje eilutėje ir yra taip pat ignoruojami.

Programos korektiškas veikimas buvo testuojamas Linux (Debian) aplinkoje MIF serveryje .

5. Eksperimentai su sprendžiamos klasės uždaviniais

Eksperimentų aplinka:

Asmeninis nešiojamasis kompiuteris Lenovo Y50-70 (2015)
Procesorius: i5-4210H 2.9 GHz
Operatyvioji atmintis: 16GB 1600MHz DDR3
Išorinė atmintis: SAMSUNG 860 EVO 500GB MZ-76E500

5.1. Lietuvos miestai ir rajonai

Pirmiausia, realizavus algoritmą, jis buvo pritaikytas Lietuvos miestų ir rajonų centrų atstumų lentelei[1] – 20 × 20 matmenų simetriškai matricai (A.2 priedas `input.example.txt`).

Programos paleidimas:

```
python3 tsp_branch_bound.py input.example.txt
```

Išvestis:

```
1 -> 3 -> 17 -> 19 -> 16 -> 20 -> 12 -> 2 -> 11 -> 7 -> 14 -> 4 -> 10 -> 13 -> 8
-> 15 -> 5 -> 18 -> 9 -> 6 -> 1

Cost = 1370

Alytus -> Druskininkai -> Varėna -> Vilnius -> Švenčionys -> Visaginas -> Rokiškis
-> Biržai -> Panevėžys -> Kėdainiai -> Šiauliai -> Joniškis -> Mažeikiai ->
Skuodas -> Klaipėda -> Šilutė -> Jurbarkas -> Vilkaviškis -> Marijampolė -> Kaunas
-> Alytus
```

Gavome optimalų maršrutą ir jo kainą `Cost`, lygią 1370 km**.**

Programos vykdymo laikas: 0.06665 s. (vidurkis iš 10-ies kartų).

5.2. Priklausomybė nuo viršūnių skaičiaus

Buvo atliktas tyrimas, kaip algoritmo vykdymo laikas priklauso nuo viršūnių skaičiaus. Lankų skaičius šiame tyrime fiksuotas – $n(n-1)$. Pradinis viršūnių skaičius buvo 3, ir jis buvo didinamas. Atlikti keli tyrimo variantai – pirmasis, kai svorių reikšmės kito nuo 1 iki 10, ir antrasis, kai jos kito nuo 1 000 iki 1 000 000. Su kiekvienu viršūnių skaičiumi buvo atlikta 10 testinių paleidimų, apskaičiuota vidutinė programos vykdymo trukmė, vidutinė vienos iteracija trukmė ir vidutinis iteracijų skaičius. Testams su skirtingais viršūnių skaičiais atlikti buvo pasitelktos pagalbinės Linux komandinės eilutės komandos (`for` ciklas).
Rezultatai:

[1] Viršūnės didinamos nuo 3 iki 207 (svorių reikšmės: nuo 1 iki 10).

Testo paleidimas:

```
python3 tsp_branch_bound.py -c $i -w 1 10 -r 1 -t 10 -s
```

kur `$i` - vis didinamas viršūnių skaičius. Linux komanda, palaipsniui didinant viršūnių skaičių:

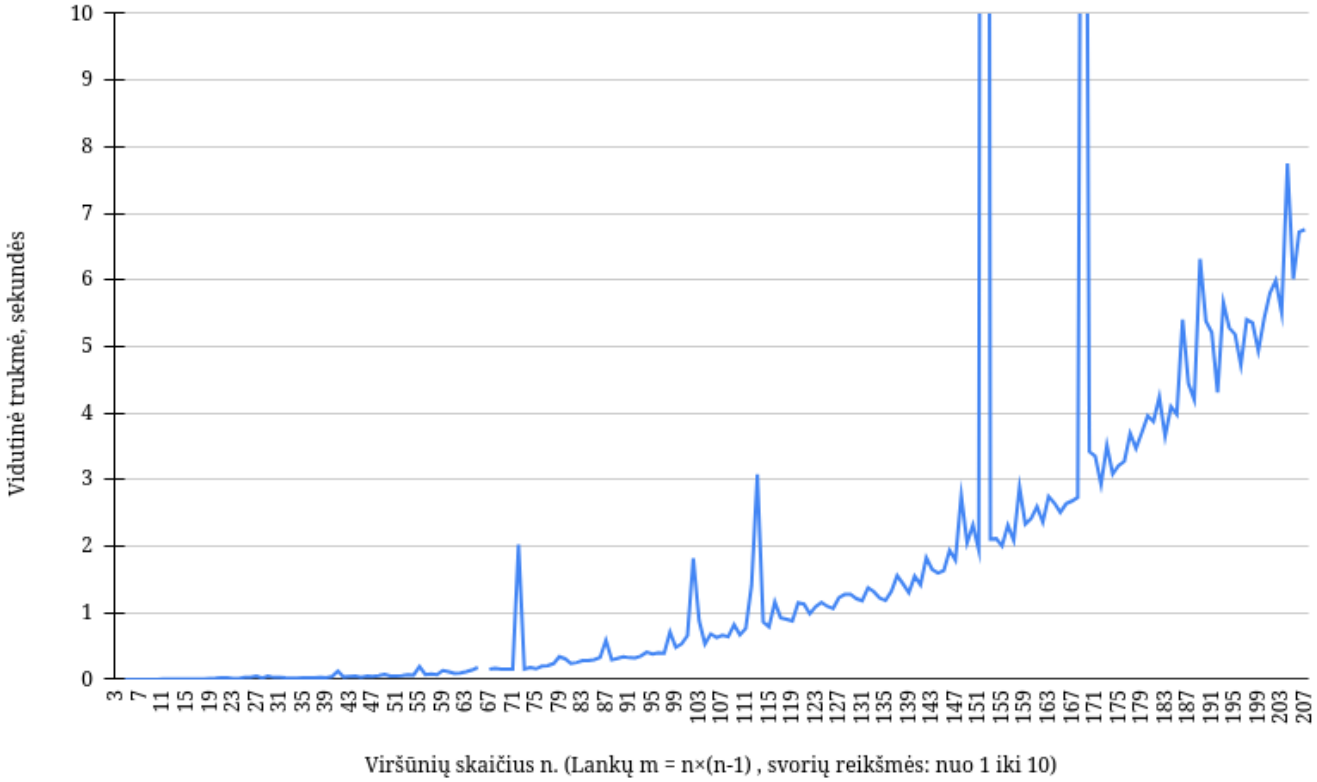
```
for ((i=3;i<=207;i++));
do
    ./tsp_branch_bound.py -c $i -w 1 10 -r 1 -t 10 -s
    if [ $? -ne 0 ]; then exit $?; fi
done
```

Visos Linux komandos, naudotos tyrimų metu, yra kiekviename rezultatų aplanke esančiuose failuose `run_tests.sh` (priedas A.4).

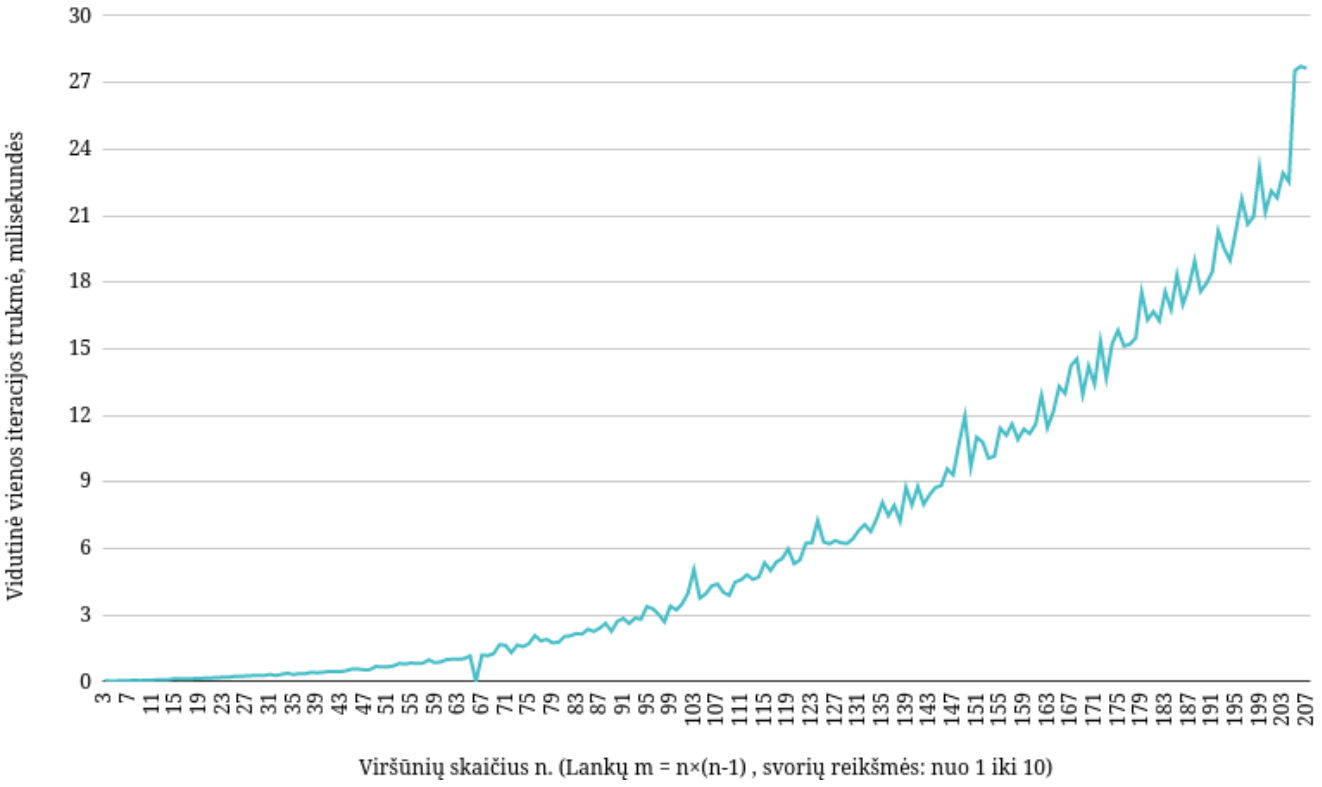
Visi grafiko duomenys detaliai pateikti priede A.5 (Excel faile).

Gauti tokie grafikai:

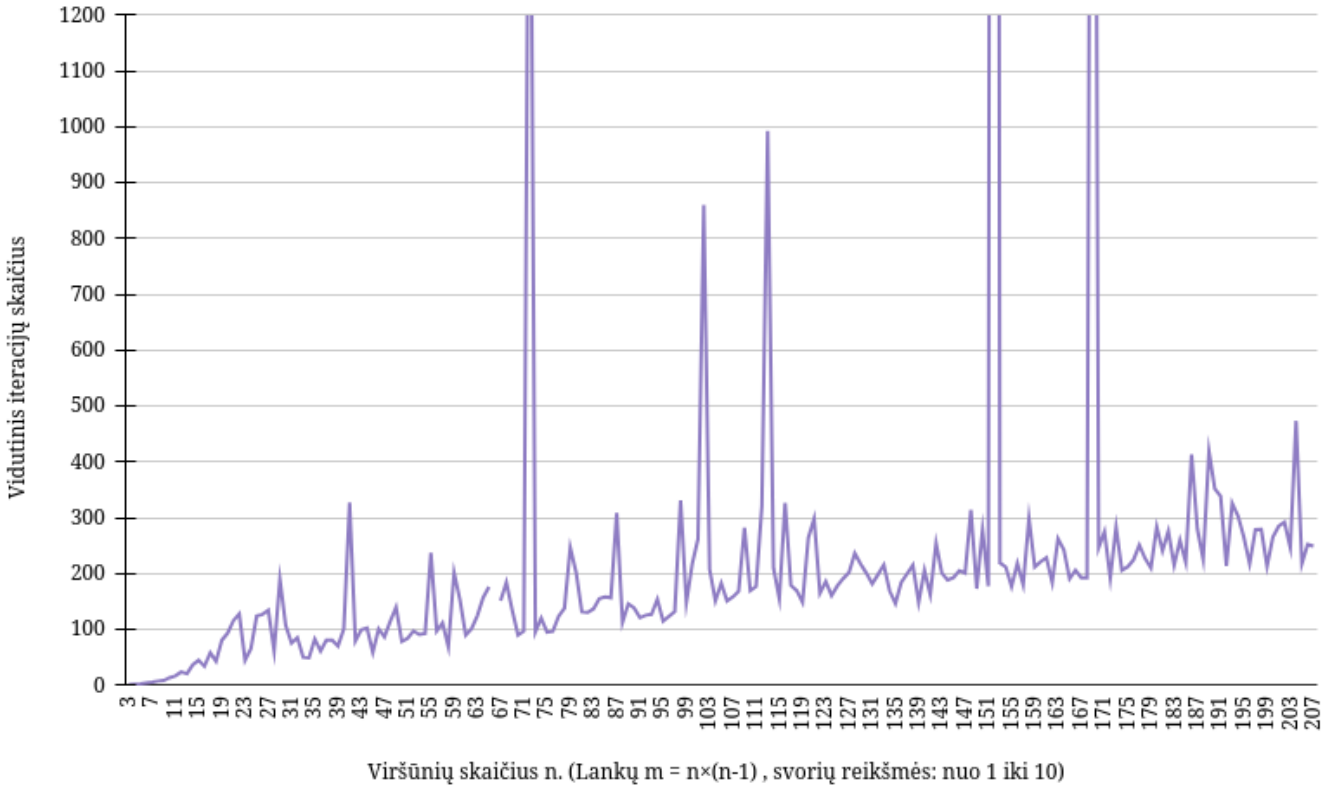
Grafikas nr. 1



Grafikas nr. 2



Grafikas nr. 3



Iš pirmų dviejų grafikų matome priklausomybę tarp viršūnių skaičiaus ir vykdymo laiko, tiek bendro, tiek vienos iteracijos trukmės. Abu jie nurodo priklausomybę, panašią į kvadratinės funkcijos parabolę (jos dalį su teigiamais argumentais). Bendrojo vykdymo laiko atveju (grafikas nr. 1) apytikslė funkcija,

atitinkanti grafiką, galėtų būti $t = \frac{1}{6000}n^2$ nuo viršunių skaičiaus n^2. Maža konstanta priekyje reiškia, kad funkcijos reikšmė kyla itin lėtai sulig viršūnių skaičiumi, tačiau asimptotiškai sudėtingumas yra kvadratinis. Tai yra neblogas rezultatas, žinant, kad blogiausiu atveju uždaviniu sudėtingumas yra $(n-1)!$. Plačiau apie sudėtingumą aptarsime kiek vėliau, (6.2.) skyriuje. Antrajame grafike funkcija taip pat galėtų būti kvadratinė, tik su kiek didesne, bet vistiek daug mažesne už nulį, konstanta.

Tuo tarpu trečiajame grafike matome vidutinio iteracijų skaičiaus priklausomybę nuo viršūnių skaičiaus – iteracijų skaičius palaipsniui taip pat didėja. Ši priklausomybė yra panašesnė į tiesinę, ir, natūralu, kad tuomet vienos iteracijos vykdymo laikui didėjant kvadratiškai (grafikas nr. 2), atitinkamai kvadratiškai ir didėja bendras vykdymo laikas, esant tiesiniam iteracijų skaičiaus didėjimui.

Grafikuose tam tikrą netvarką duoda skirtingi atsitiktinai sugeneruoti įvesties duomenys. Didžiausi nukrypimai matomi grafike nr. 3. Iš tiesų, iteracijų skaičius labiausiai varijuoja, priklausomai nuo duomenų, nes čia yra ir sėkmės elementas, kaip greitai šakų ir rėžių metodu atrasime geriausią tinkamą pomedį, ir kiek netinkamų pomedžių reikės peržiūrėti.

Grafikuose nr.1 ir nr.3 matome, kad kelios reikšmės netilpo į atvaizduotą grafiką, kitaip jis būtų labai pernelyg daug ištemptas vertikaliai, tad pateikimo patogumo dėlei jos buvo paliktos už ribų. Šios reikšmės yra pridėtame Excel faile. Grafike nr. 1 jos viršijo aplink jas esančias reikšmes nuo 10 iki 25 kartų, grafike nr. 3 – nuo 30 iki 70. Tai reiškia, kad net ir su palyginus nedideliais duomenimis šis algoritmas, žinoma, gali nesielgti dėsningai pagal grafiką, ir altikti keliasdešimt kartų daugiau iteracijų, kartu gaunant ir keliasdešimt kartų ilgesnį vykdymo laiką, o įvesties duomenims didėjant, tokie užtrukimai gali būti ir dar labiau santykinai didesni.

Tęsiant šią tyrimo dalį, buvo padidintos galimos svorių reikšmės: minimali reikšmė – 1 000, maksimali – 1 000 000, ir vėl tikrinama, kaip vykdymas keičiasi didėjant viršūnių skaičiui.

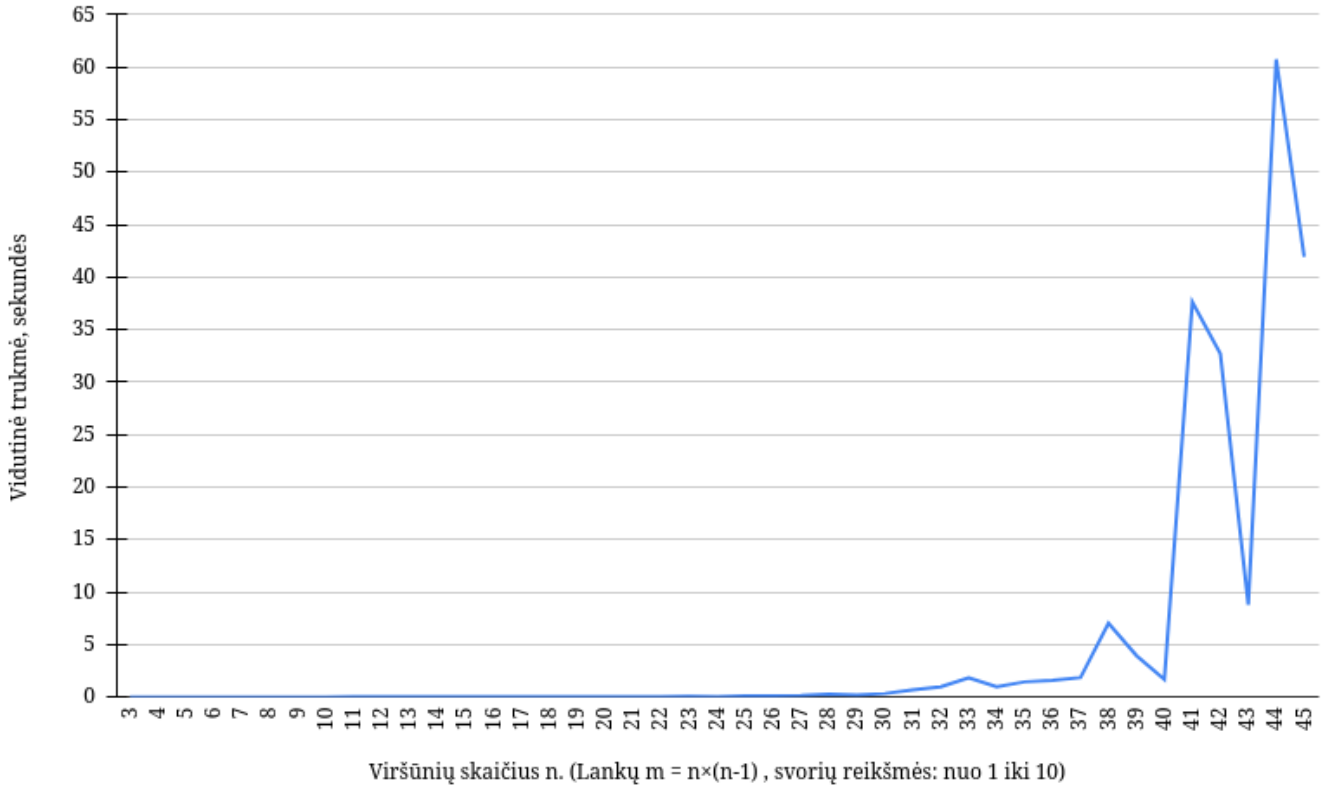
Testo paleidimas:

```
python3 tsp_branch_bound.py -c $i -w 1 1000 -r 1 -t 1000000 -s
```

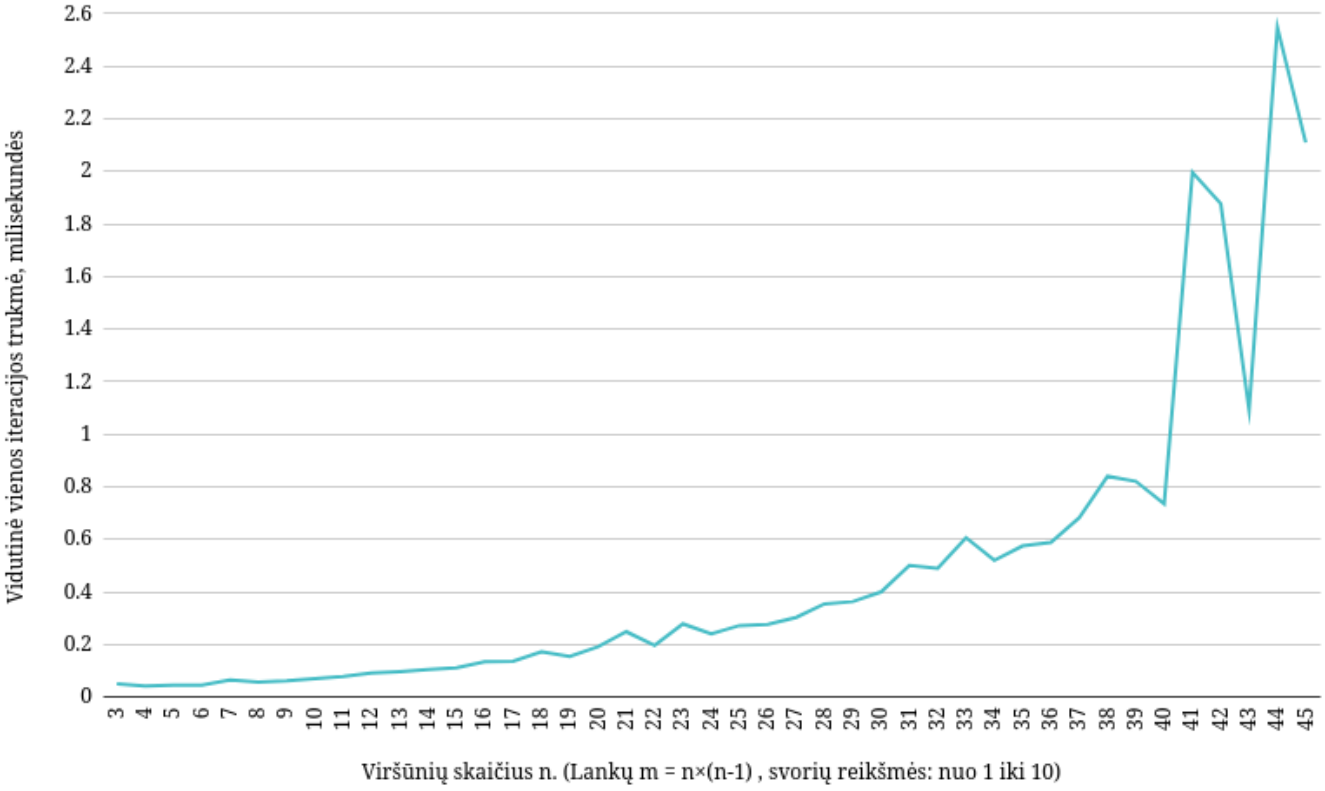
kur `$i` vėlgi yra didinamas viršūnių skaičius. Daug testų iš eilės paleisti buvo naudojama ta pati Linux `shell` komanda – `for` ciklas.

Gauti tokie grafikai:

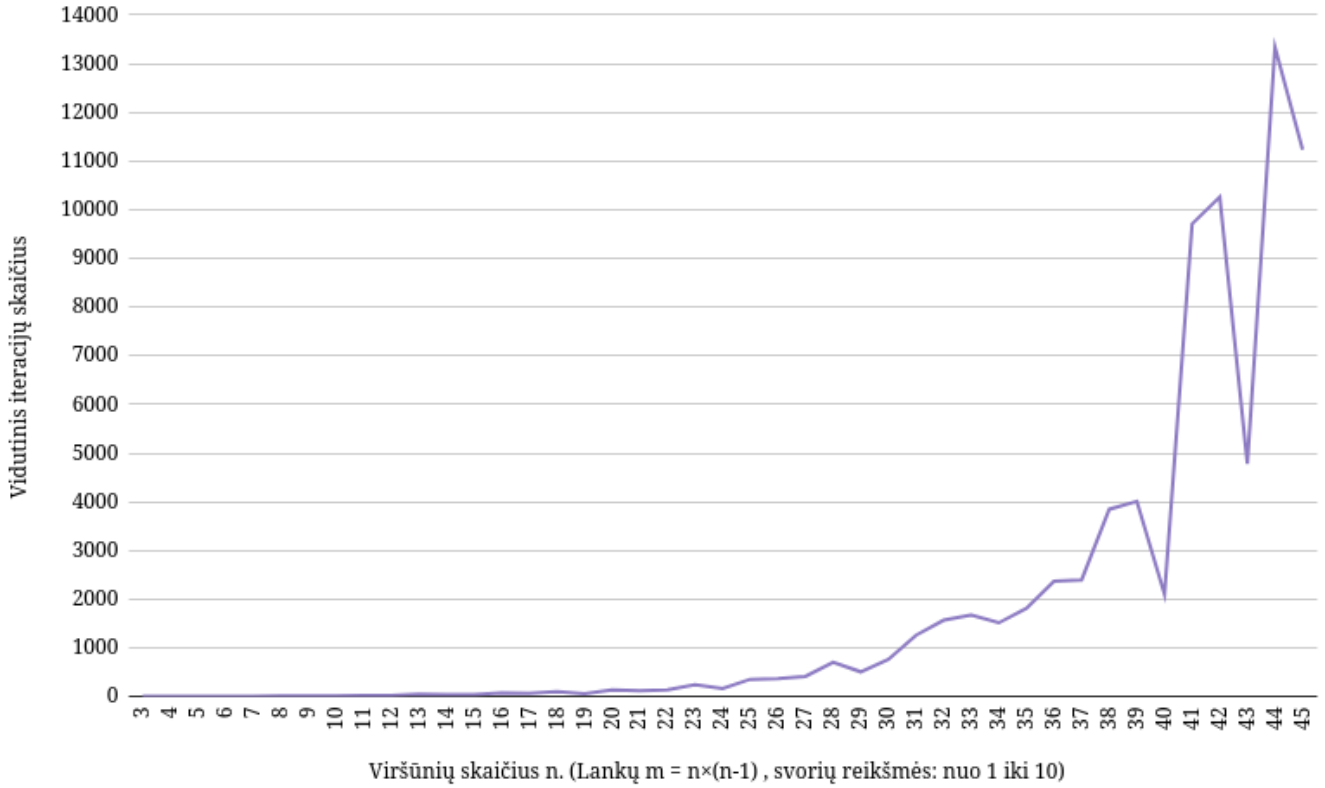
Grafikas nr. 4



Grafikas nr. 5



Grafikas nr. 6



Štai šiuo atveju galime matyti, kad, padidinus svorių reikšmes, pirklausomybė smarkiai skiriasi nuo pirmojo atvejo. Ties 45 viršūnėmis buvo sustota, nes programos vykdymo trukmė jau pradėjo viršyti 1 minutę, o 10 pakartotinių vykdymų su vienu viršūnių skaičiumi tuomet jau viršija 10 minučių. Nors iš pateiktos grafiko dalies priklausomybė gal ir atrodo kvadratinė, tačiau matome labai smarkų vykdymo laiko pakilimą pradedant nuo 40 viršūnių, kas leidžia daryti prielaidą, kad tuomet vykdymo laikas galimai pradeda didėti eksponentiškai. Vykdymo laiko grafikus gana akivaizdžiai paaiškina trečiasis, iteracijų skaičiaus grafikas, kuriame matome, kad atitinkamai nuo 30 viršūnių pradeda ryškiai didėti iteracijų skaičius, o nuo 40 viršūnių dar staigiau šauna į viršų. Tai rodo, kad šis įgyvendintas šakų ir rėžių algoritmas su didesnėmis svorių reikšmėmis duoda daug didesnį vykdymo laiką, kuomet viršūnių skaičius padidėja iki tam tikros ribos (35-40 viršūnių). Šio pastebėjimo išplėtimas, vėlgi, aptartas tolimesniame (6.2) skyriuje.

5.4. Priklausomybė nuo svorių reikšmių

Esant fiksuotiems viršūnių skaičiams – 10, 20, 30, 40, 50, 100, sulig kiekvienu iš jų buvo keičami lankų svorių reikšmių intervalai. Minimali jų reikšmė išlikdavo lygi 1, tačiau maksimalią reikšmę didindavome nuo 10 iki 10 000 000. Lankų skaičius vėlgi buvo fiksuotas – $n(n-1)$.

Testo paleidimas:

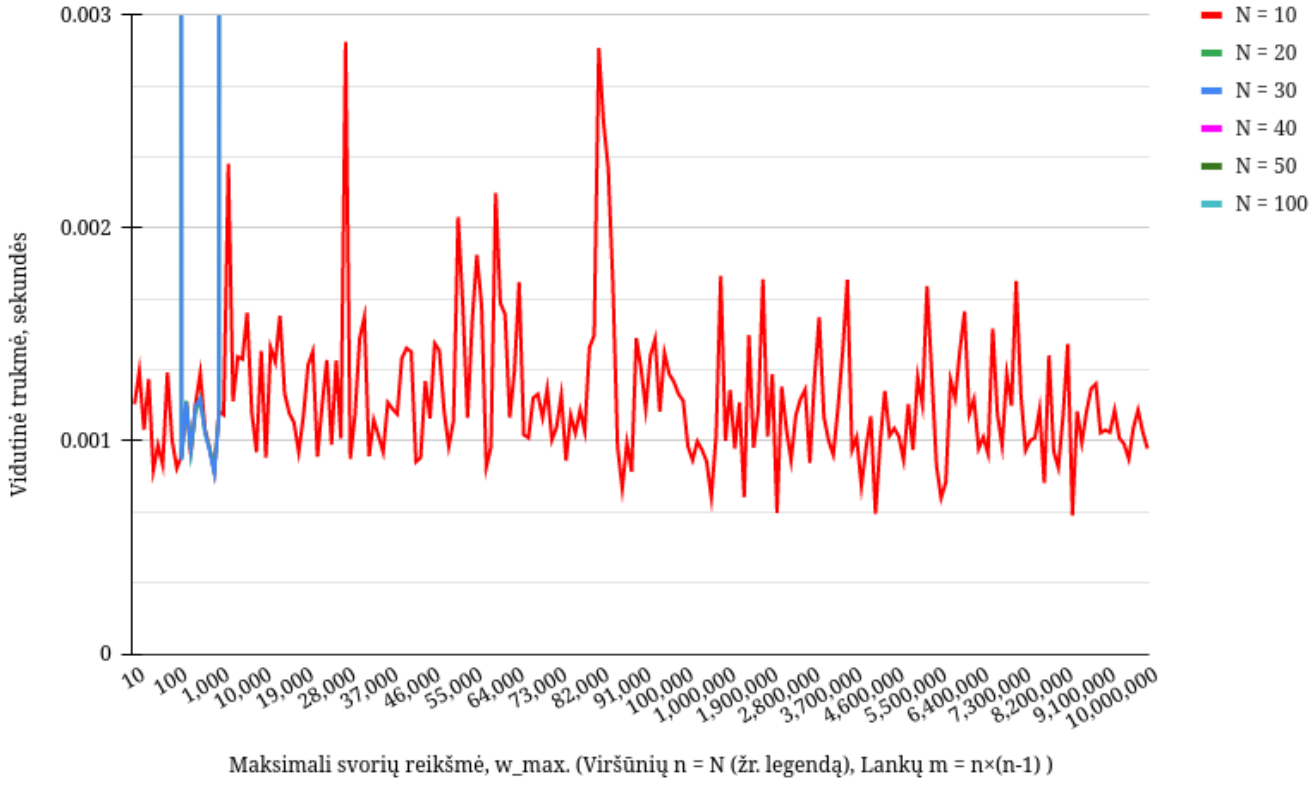
```
./tsp_branch_bound.py -c $c -w 1 $i -r 1 -t 10 -s
```

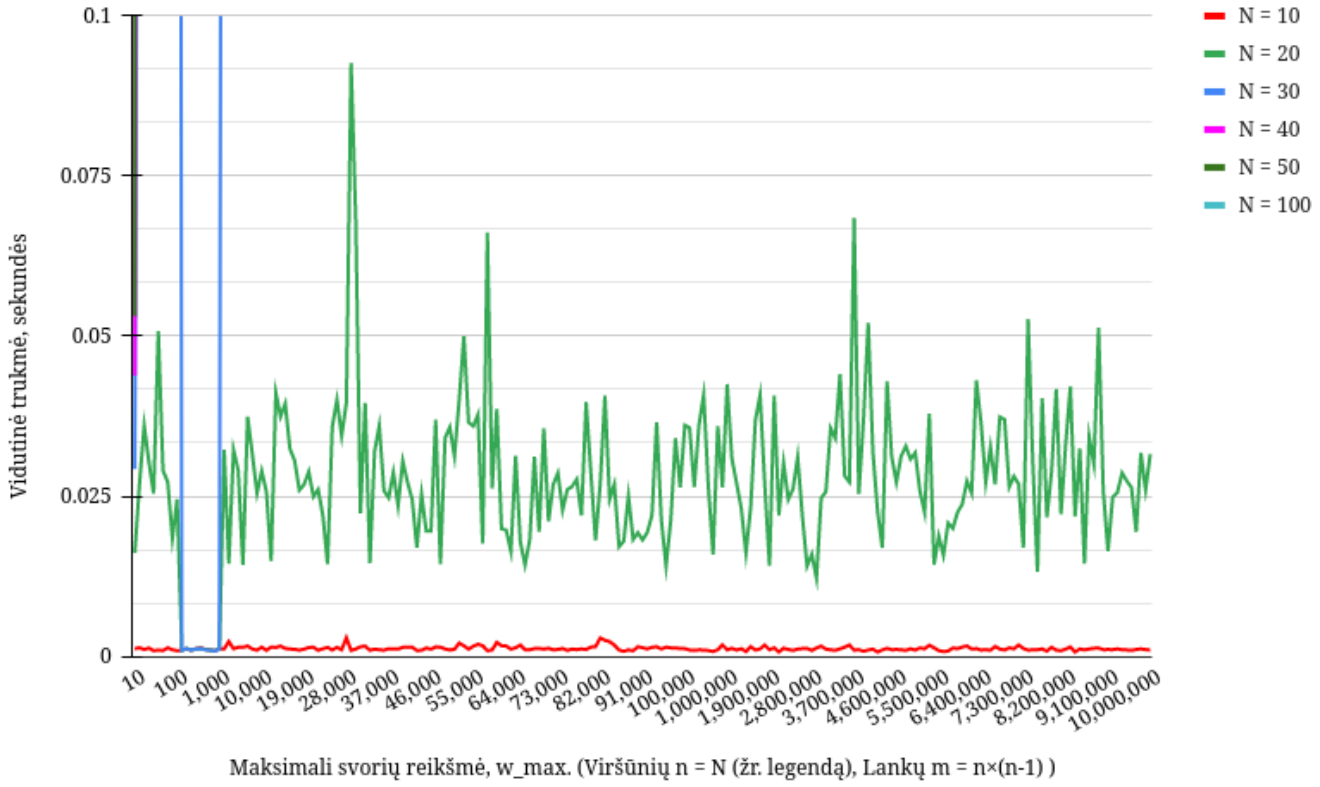
kur `$c` yra pasirinktas viršūnių skaičius, o `$i` – maksimali svorių reikšmė.

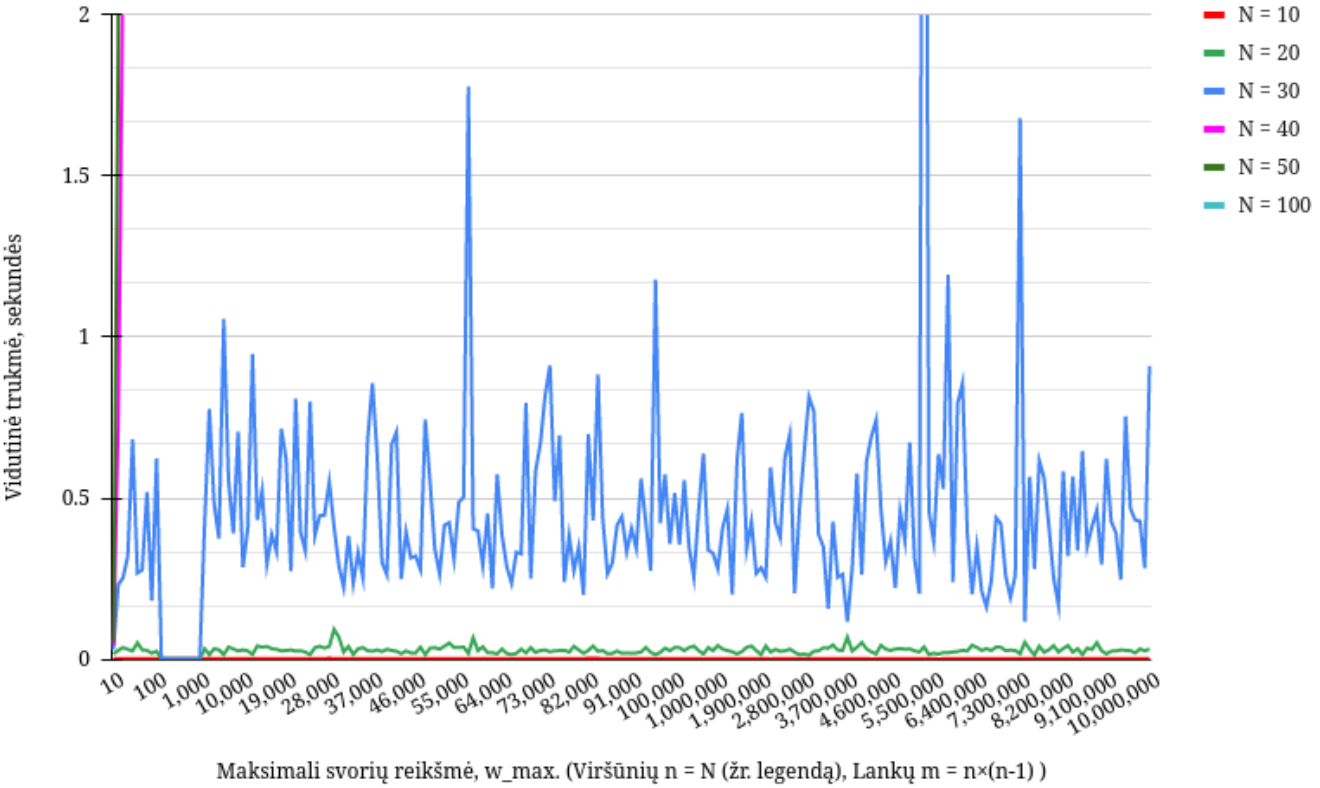
Gautuose rezultatuose gavosi tokia situacija, kad daug didesnį pokyti vykdymo laikui turėjo pats viršūnių padidinimas, nei svorių reikšmių didinamas, fiksuojant vieną viršūnęs reikšmę. Visa šio tyrimo informacija buvo vėlgi sutalpinta į tris grafikus, tačiau kiekviename iš jų skirtingomis spalvomis buvo nubrėžtos šešios kreivės, atitinkančios skirtingą viršūnių skaičių nuo 10 iki 100. Patogumo dėlei, atvaizduosime kiekvieną iš šių grafikų per tris-keturis dalinius to grafiko paveiksliukus, keičiant jo priartinimą (tiksliau sakant - nustatant maksimalią reikšmių srities reikšmę). Pilni grafikai yra pateikti A.5 priede, Excel faile.

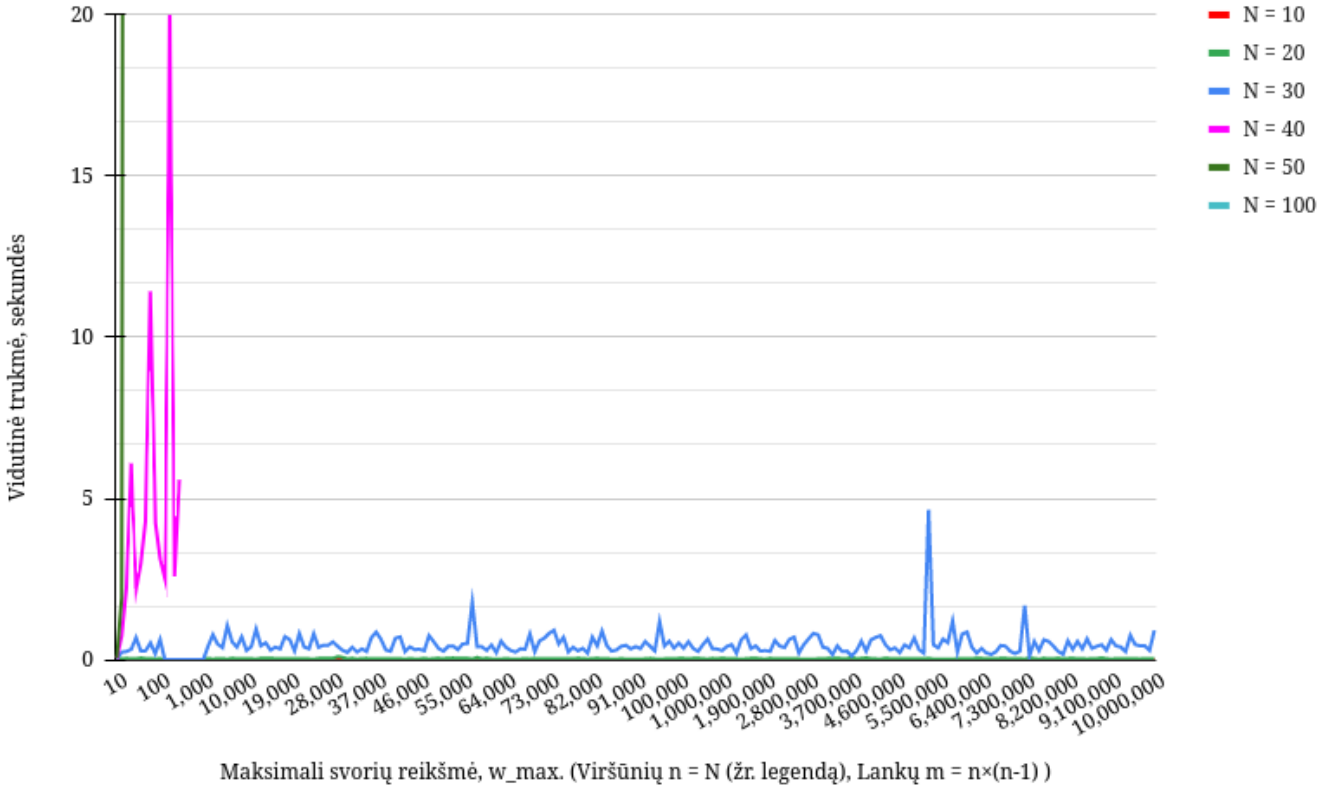
Gauti grafikai:

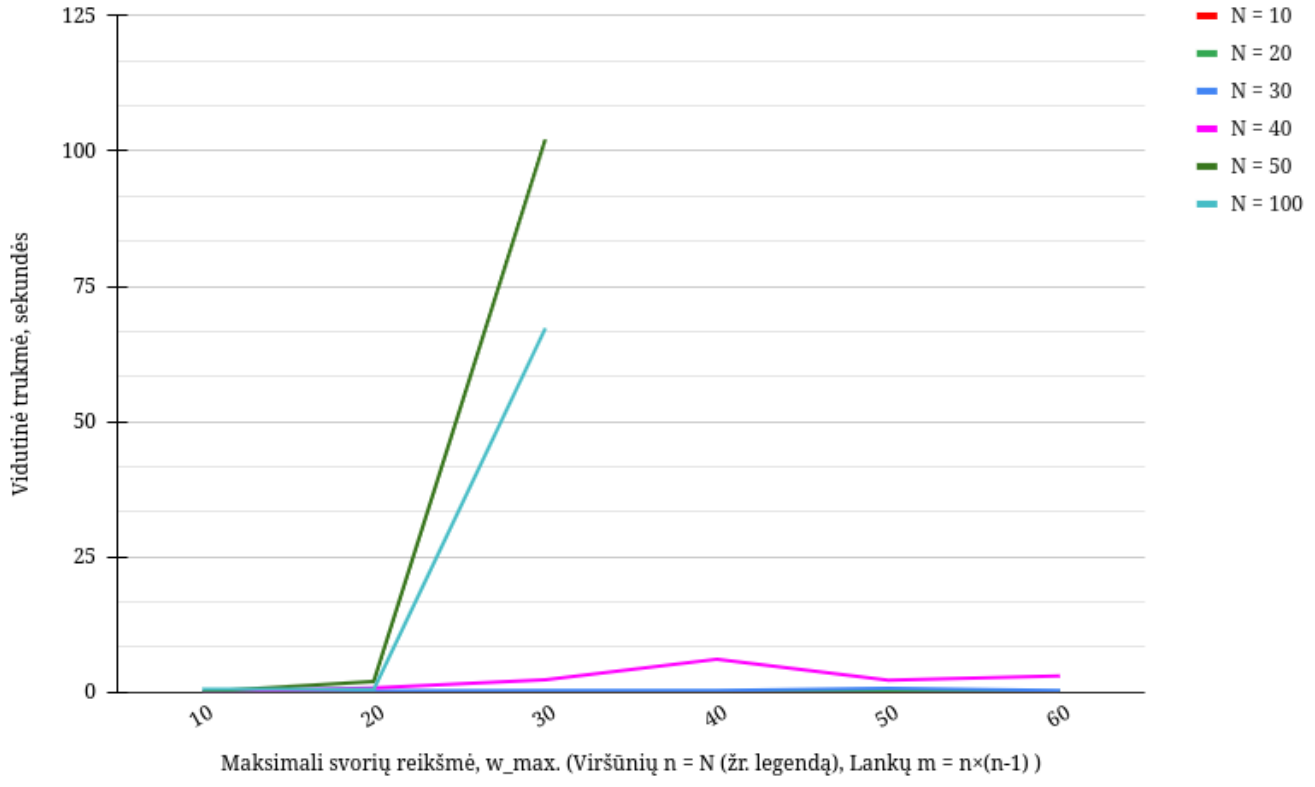
Grafikas nr. 7 – vidutinė trukmė (susideda iš žemiau pateiktų paveikslėlių):



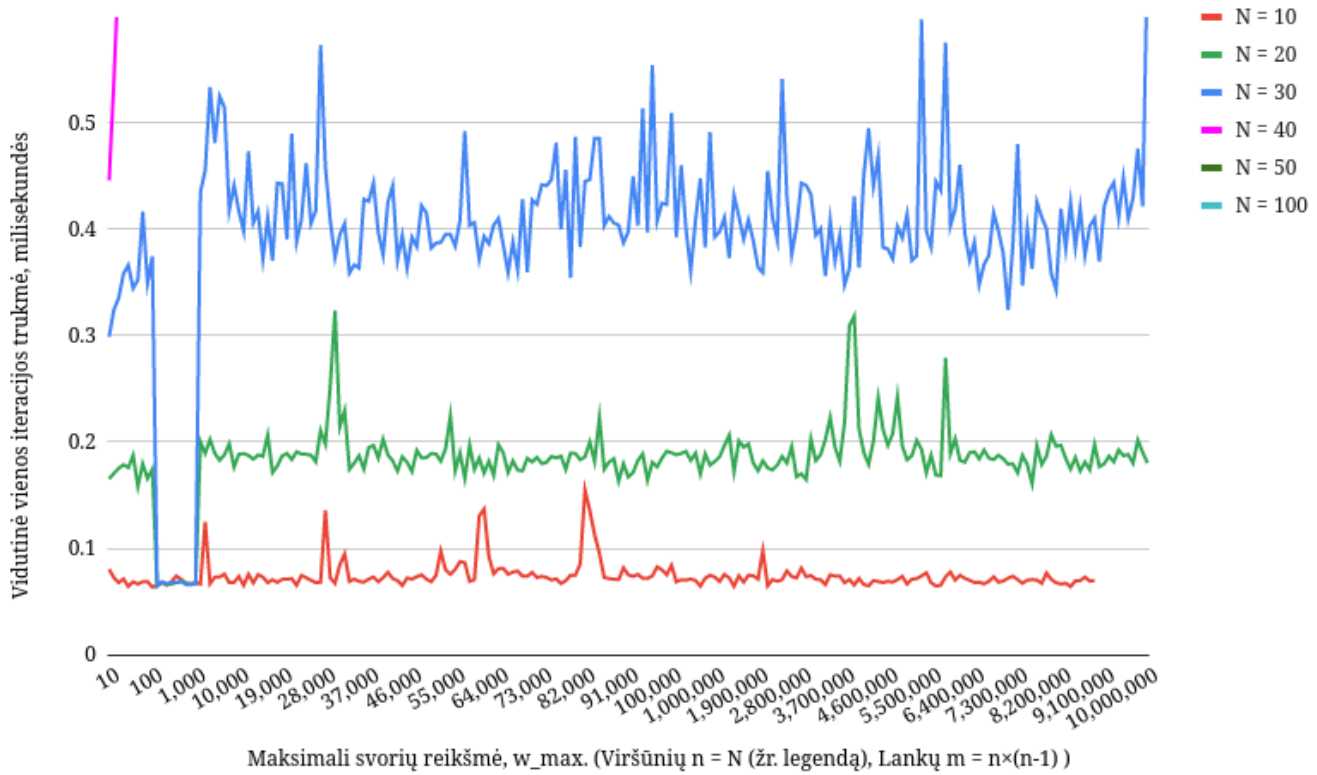

Vidutinė trukmė, sekundės
0.1
0.075
0.05
0.025
0
10
100
1,000
10,000
19,000
28,000
37,000
46,000
55,000
64,000
73,000
82,000
91,000
100,000
1,000,000
1,900,000
2,800,000
3,700,000
4,600,000
5,500,000
6,400,000
7,300,000
8,200,000
9,100,000
10,000,000
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))

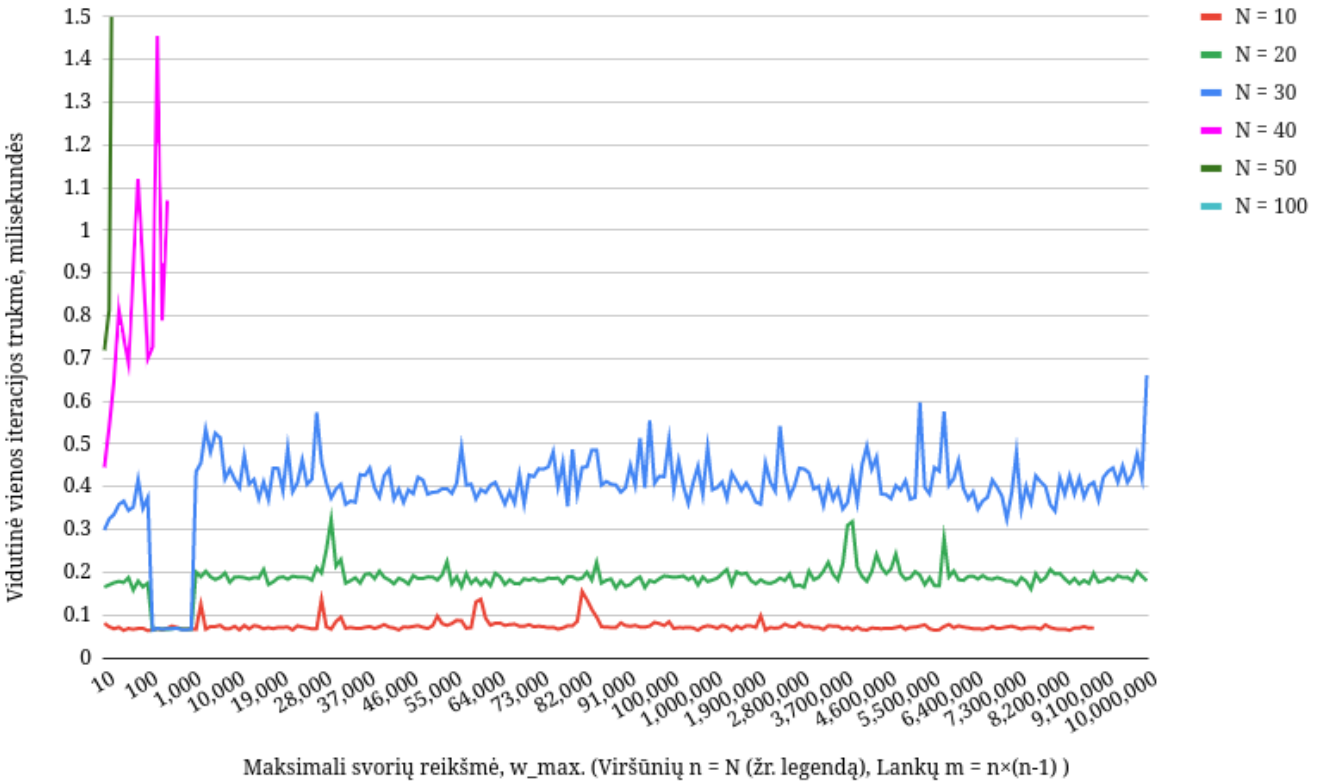

Vidutinė trukmė, sekundės
2
1.5
1
0.5
0
10
100
1,000
10,000
19,000
28,000
37,000
46,000
55,000
64,000
73,000
82,000
91,000
100,000
1,000,000
1,900,000
2,800,000
3,700,000
4,600,000
5,500,000
6,400,000
7,300,000
8,200,000
9,100,000
10,000,000
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))

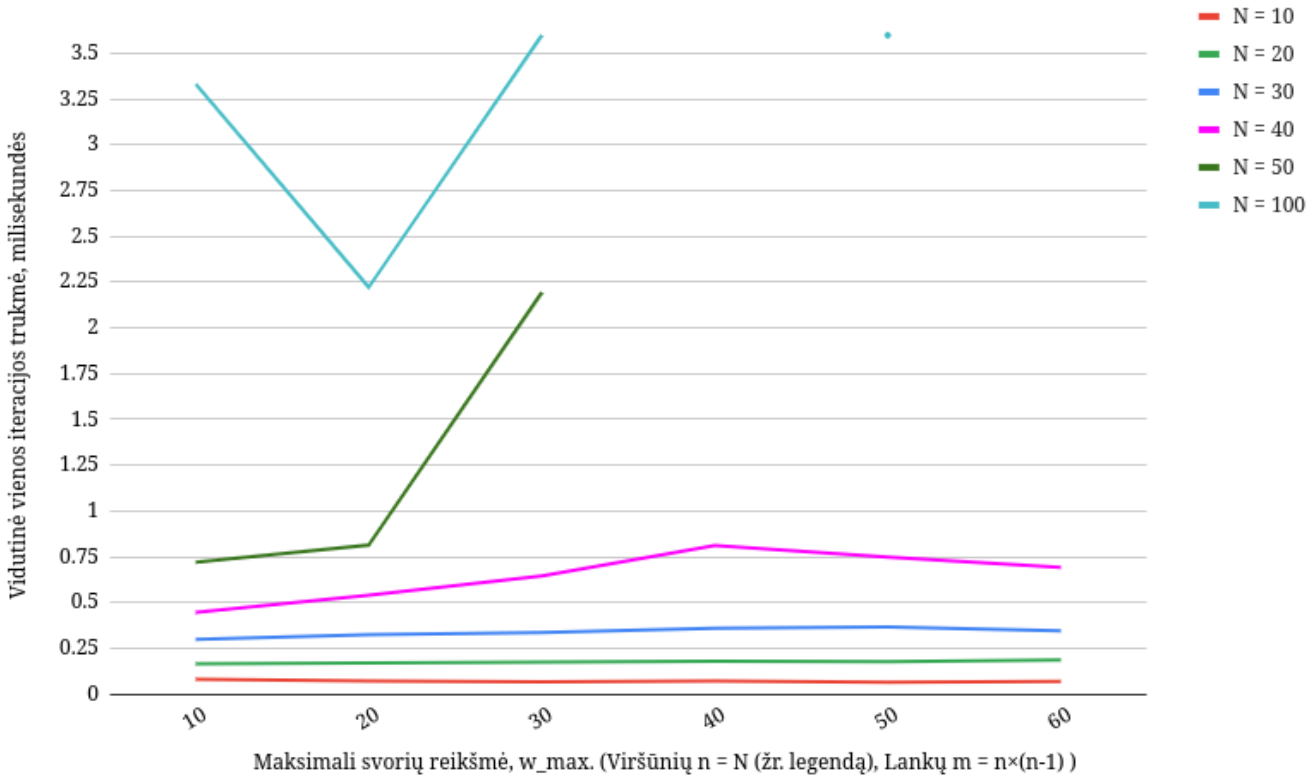

20
15
10
5
0
Vidutinė trukmė, sekundės
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
10
100
1,000
10,000
19,000
28,000
37,000
46,000
55,000
64,000
73,000
82,000
91,000
100,000
1,000,000
1,900,000
2,800,000
3,700,000
4,600,000
5,500,000
6,400,000
7,300,000
8,200,000
9,100,000
10,000,000
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))


125
100
75
50
25
0
Vidutinė trukmė, sekundės
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
10
20
30
40
50
60
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))

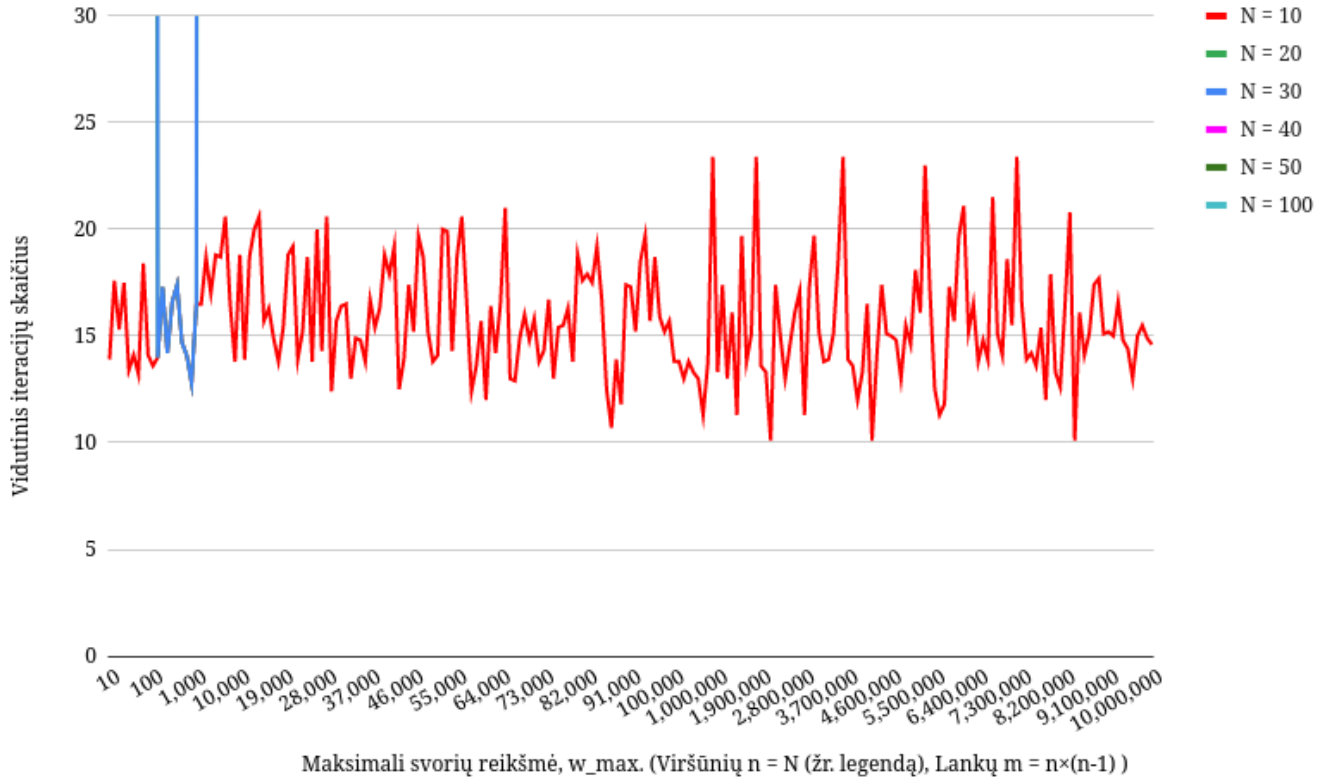
Grafikas nr. 8 – vidutinė vienos iteracijos trukmė (susideda iš žemiau pateiktų paveikslėlių):

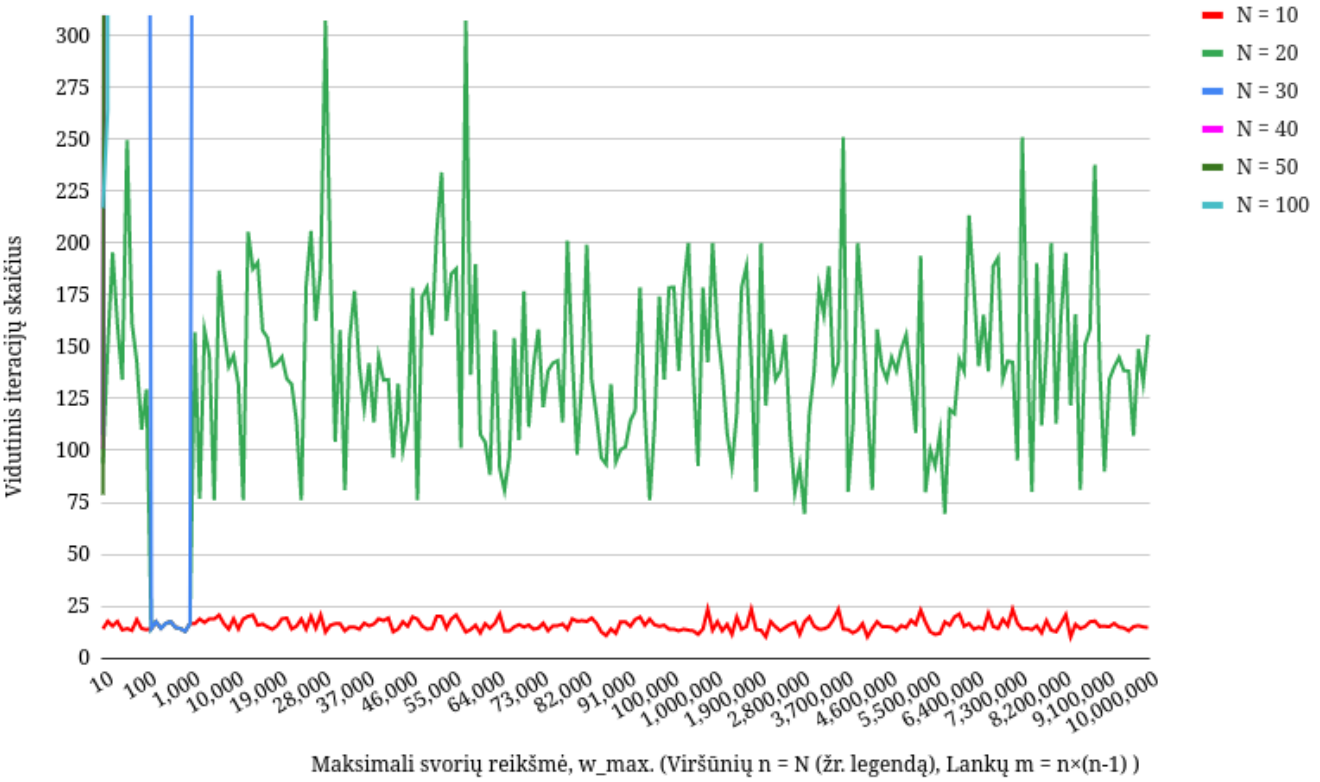


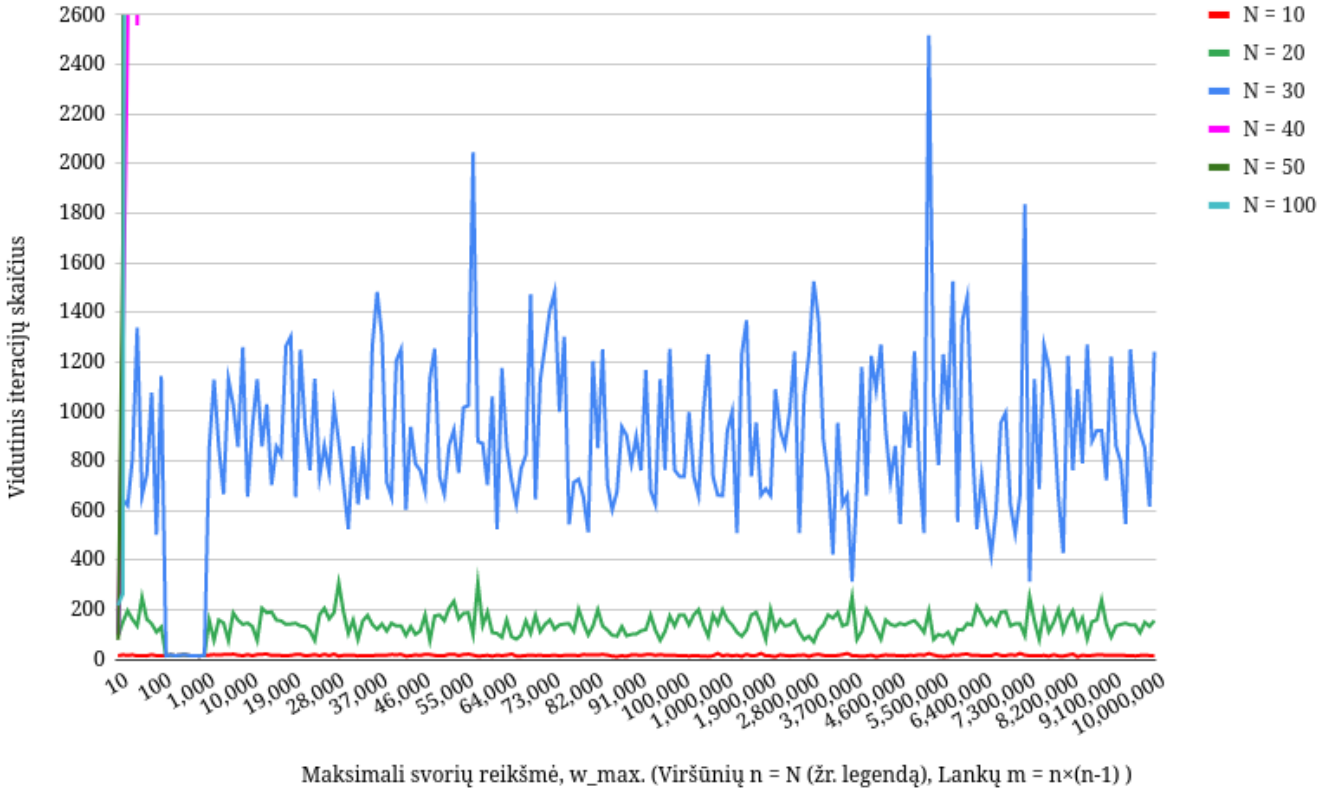


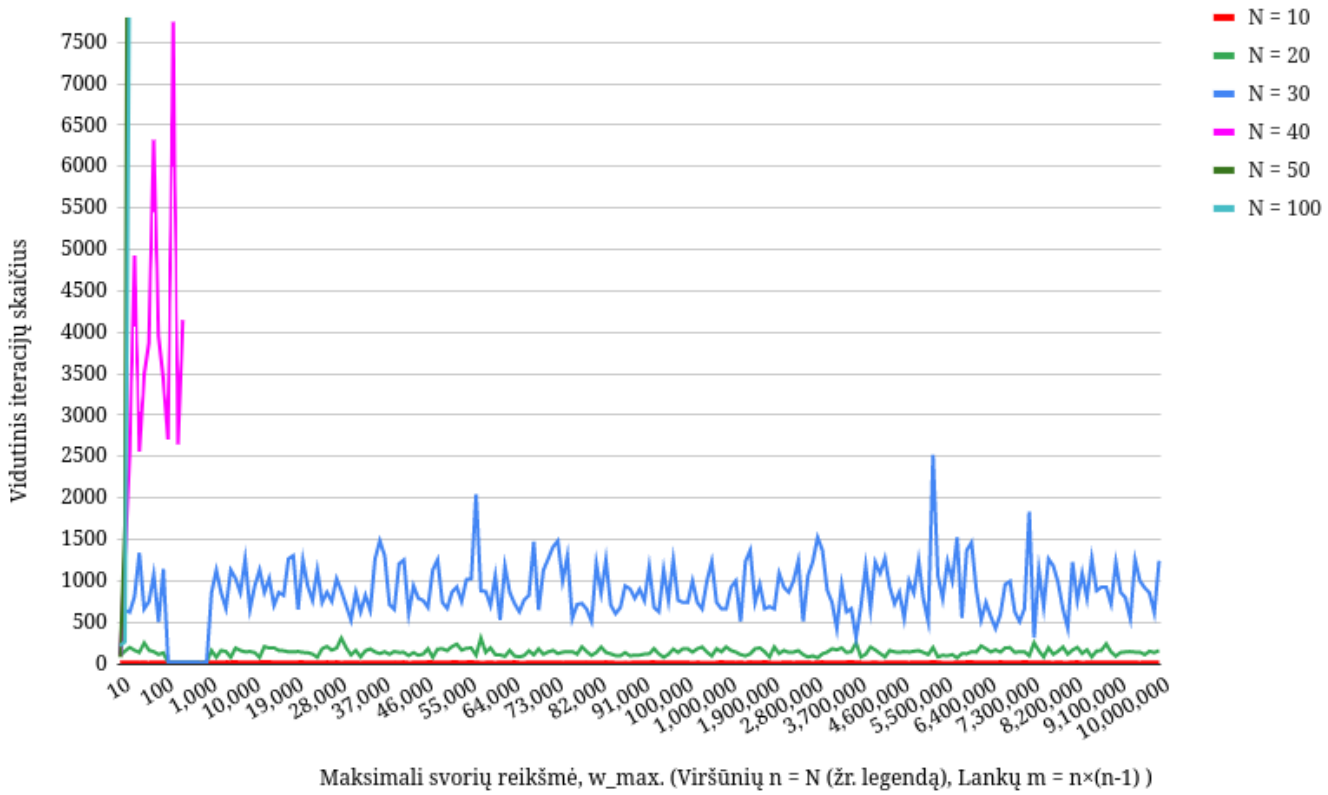

N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
Vidutinė vienos iteracijos trukmė, milisekundės
3.5
3.25
3
2.75
2.5
2.25
2
1.75
1.5
1.25
1
0.75
0.5
0.25
0
10
20
30
40
50
60
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))

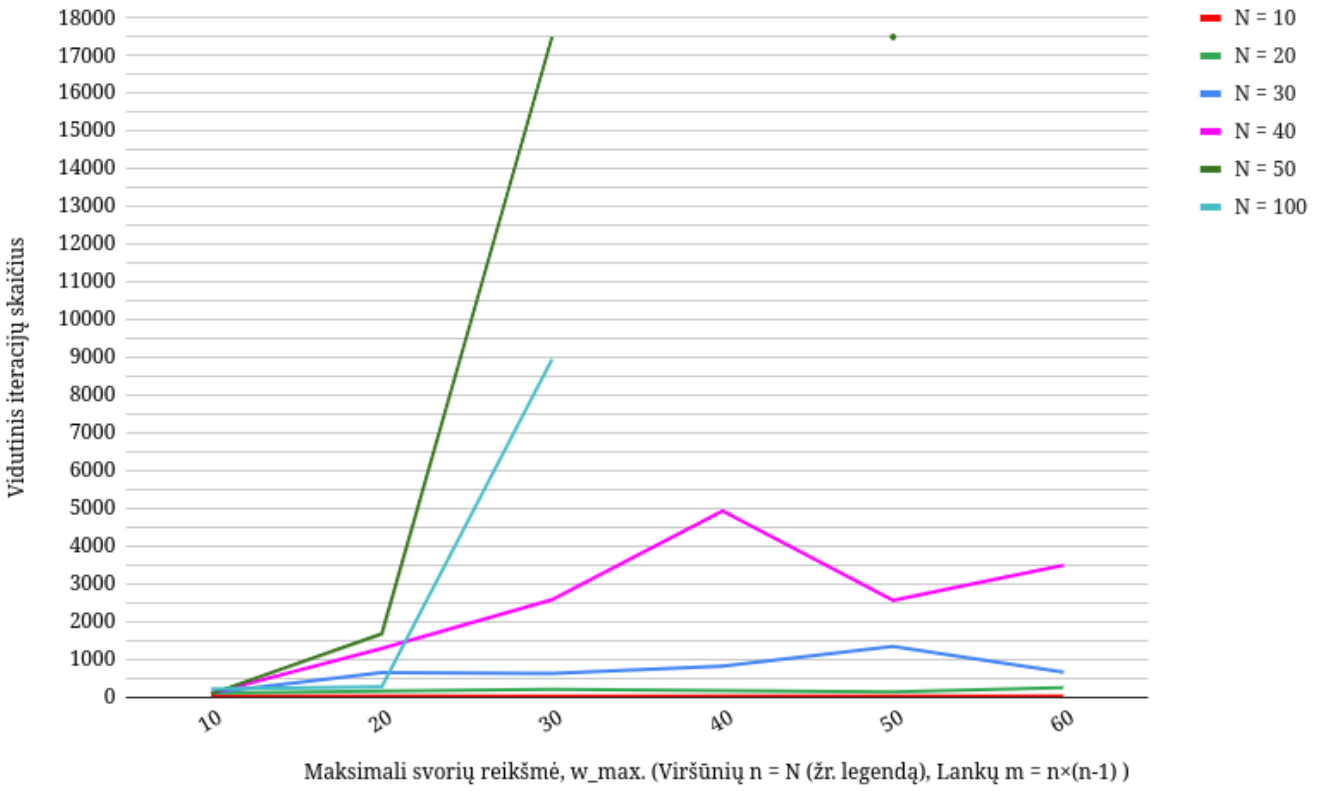
Grafikas nr. 9 – vidutinis vienos iteracijos skaičius (susideda iš žemiau pateiktų paveikslėlių):






Vidutinis iteracijų skaičius
2600
2400
2200
2000
1800
1600
1400
1200
1000
800
600
400
200
0
10
100
1,000
10,000
19,000
28,000
37,000
46,000
55,000
64,000
73,000
82,000
91,000
100,000
1,000,000
1,900,000
2,800,000
3,700,000
4,600,000
5,500,000
6,400,000
7,300,000
8,200,000
9,100,000
10,000,000
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))


Vidutinis iteracijų skaičius
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
10
100
1,000
10,000
19,000
28,000
37,000
46,000
55,000
64,000
73,000
82,000
91,000
100,000
1,000,000
1,900,000
2,800,000
3,700,000
4,600,000
5,500,000
6,400,000
7,300,000
8,200,000
9,100,000
10,000,000
N = 10
N = 20
N = 30
N = 40
N = 50
N = 100
Maksimali svorių reikšmė, w_max. (Viršūnių n = N (žr. legendą), Lankų m = n×(n-1))



Iš šių grafikų matyti, kad su viršūnių skaičiais 10, 20, ir 30 svorių padidinimas didelės įtakos programai nedaro. Įskaitant duomenų išsibarstymą dėl atsitiktinai sugeneruotų matricų, vykdymo laikas ir iteracijų skaičius daugmaž išlieka pastovus. Tačiau skirtumas padidinus viršūnes nuo 10 iki 20, ir iki 30 yra akivaizdžiai matomas, kuo jau ir įsitikinome pirmojoje tyrimo dalyje (5.1.). Sudėtingesnė situacija yra tuomet, kai viršūnių skaičius pasiekia 40, 50 ir 100. Čia vykdymo laikai ne tik skiriasi, kaip ir turėtų, nuo mažesnių viršūnių skaičiaus atvejų, tačiau parodo staigų kilimą padidinus maksimalią svorių reikšmę nuo 10 iki 20, ir iki 30, ypač 50-ies ir 100 viršūnių atveju. Šie vykdymo laikai viršijo 1 minutę, o su didesnėmis svorių reikšmėmis užtruko dar ilgiau, todėl tolesni tyrimai su didinant svorius nebuvo atliekami dėl didelių laiko sąnaudų. Iš to, kaip pradėjo elgits kreivės N = 50, N = 100, ir iš to, kaip ilgai veikė tolimesnių svorių reikšmių testai, galima daryti prielaidą, kad su šiuo viršūnių skaičiumi jau svoriai pradeda daryti žymesnę įtaką. Tai galėtų būti paaiškinta tuo, kad svoriams turint daugiau skirtingų reikšmių prastinant matricą vis rečiau gauname, kad suprastinus vieną eilutę ar stulpelį, kiti papildomi elementai irgi tampa lygūs 0. Tačiau tai yra gan silpna prielaida, ir labai galimas dalykas, kad daugiau laiko paskyrus šiems testams pasirodys, kad ir su 50 bei 100 viršūnių svorių kitimas didelės įtakos neduos, o grafiko kreivė svyruos aukštyn-žemyn taip pat, kaip ir likusios kreivės.

6. Realizuotų algoritmų sudėtingumo analizė

6.1. Teorinis sudėtingumas

Operacijų skaičius (nuo viršūnių skaičiaus *n*) pagal programos bloką:

[1] Viršūnių perrinkimas ir 0-nių elementų reikšmių nustatymas į `None` (begalybę) – n^2.

[2] Matricos eilučių ir stulpelių prastinimas –n^2.

[3] Rėžio pokyčio radimas – n^4. Turime $c_1 n \times (n + n) = c_2 n^2$, kur $c_1 n$– skaičius elementų, kuriems apskaičiuojamas rėžio pokytis. Taip pat, blogiausiu atveju šis blokas yra kartojamas tol, kol randamas kelias, nesudarantis dalinių ciklų. Tokiu būdu tikriname kažkokį tai skaičių galimų kelių $c_3 n$, taigi $c_2 n^2 \times c_3 n = cn^3$. Dalinių ciklų patikrinimas užimą $c_4 n$ skaičių operacijų, taigi gauname cn^4.

[4] Sudėtis – 1 veiksmas.

[5] Matricos prastinimas –n^2.

[6] Matricos dydžio patikrinimas – 1 veiksmas.

[7] Reikia visada pridėti tik du kelius – konstanta c.

[8] Palyginimas – 1 veiksmas.

[9] Naujos viršūnės pasirinkimas iš prioritetinės eilės – 1 veiksmas.

[10] Palyginimai – 2 veiksmai.

[11] Kai reikia vėl koreguoti C matricą, kylame nuo viršūnės *X* paieškos medžiu į viršų (*n*), kiekvienoje viršūnėje atliekame *c* veiksmų su joje nurodytu įeinančiu arba neįeinančiu keliu, ir turime *cn*. Tačiau taip pat turime pridėti visus įėjusius kelius į dabartinį kaupiamą maršrutą, kas užtruks dar papildomai *n* žingsnių (nes bus tikrinama dėl dalinių ciklų), taigi sudėtingumas išauga iki n^2. Pabaigoje dar turime suprastinti matricą, tačiau nuo to šio bloko sudėtingumas (nepaisant konstantos) nesikeičia ir išlieka n^2 .

Taigi, jeigu programoje reikalingą pomedį (maršrutą) randame vieną iš pirmųjų, sudėtingumas bus O(n^4). Tačiau svarbiausias aspektas yra tai, kad KPU uždavinio sudėtingumas priklauso nuo duomenų, ir blogiausiu atveju gali tekti perrinkti visus $(n - 1)!$ maršrutų[5]. Šiame darbe realizuotai programai taip pat galioja ši savybė.

Taip pat, verta paminėti, kad eksperimentiškai yra apskaičiuota, kad atsitiktinei atstumų tarp miestų matricai vidutinis sudėtingumas yra $O(1.26^n)$[5].

6.2. Praktinis sudėtingumas

Praktiškai ištyrėme, kad didinant viršūnių skaičių nuo 3 iki 207 vykdymo laikas didėjo su kvadratine priklausomybe. Pagal iteracijų skaičių sprendžiame, kad maršrutų perrinkta nebuvo itin daug, o santykinai vis tiek nedidelis testuotų viršūnių skaičius galimai lėmė, kad **[3]** bloke nedažnai reikėjo imti ir tikrinti kitus potencialius kelius ir jiems vėl perskaičiuoti rėžio pokyčius, todėl **[3]** bloko sudėtingumas, o kartu ir viso algoritmo, buvo kur kas arčiau n^2 žingsnių, nei n^4.

Tačiau turėjome ir kelis atvejus, kuomet iteracijų skaičius, o kartu ir vykdymo laikas labai smarkiai išsiskyrė nuo gretima esančių rezultatų, ir tai jau matomai pasireiškė tie atvejai, kai buvo perrenkama daug daugiau maršrutų, iki kol buvo rastas geriausias.

7. Išvados ir pastebėjimai

Kompiuterine programa buvo realizuotas konkretus paieškos su grįžimu algoritmas – šakų ir rėžių metodas – keliaujančio pirklio uždaviniui spręsti, ištirtas jo sudėtingumas teoriškai ir praktiškai (eksperimento būdu).

Per šį pavyzdį buvo dar labiau susipažinta su efektyviais kombinatoriniais algoritmais bei algoritmų konstravimo metodais. Tačiau kur kas daugiau buvo išmokta apie paties programavimo praktikas realizuojant šį algoritmą. Dar kartą pasitvirtino kelių kertinių dalykų svarba – darbu žingsių planavimas, atominių pokyčių fiksavimas (naudojant *Git* sistemą šiuo atveju), atskirų programos funkcijų prototipavimas ir testavimas (*unit tests*) prieš integruojant jas į pagrindinę kodo dalį.

Taip pat buvo dirbama su duomenų struktūromis ir jų realizacija šiam algoritmui vaizduojant kombinatorinius objektus kompiuterio atmintyje ir operuojant su jais. Čia vėlgi buvo svarbu suplanuoti iš anksto ne tik darbo žingsnius, t.y. bendrą algoritmo struktūrą ir programinį jos prototipą, bet ir apsibrėžti duomenų struktūras. Viena iš esminių pamokų buvo bandant sprendinių paieškos medį realizuoti paprastu sąrašu. Iš pradžių tai buvo daroma vardan paprastumo, tačiau tikrinant su tuo susijusias klaidas programoje ir vis koreguojant sąrašo apibrėžimą ir funkcijas, galiausiai buvo grįžta prie medžio struktūros, realizuotos viršūnių klasėmis. Tokiu atveju atsirado ir daugiau aiškumo, ir mažiau galimybių klaidoms išlįsti.

Sukurtas programos naudojimo interfeisas su papildoma dokumentacija užtruko nemažai laiko, ir priminė apie šios dalies svarbą programoje, nes kiekviena programa, kad ir kokį efektyvų algoritmą realizuotų, galiausiai turės savo vartotoją, kuriam turėtų būti aišku, kaip programa naudotis.

Įgyvendinus algoritmą, buvo atliktas eksperimentas su Lietuvos miestų ir rajonų centrų atstumais, o po to ir atlikti eksperimentai keičiant viršūnių skaičių ir svorių reikšmes. Dar kartą pavyko įsitikinti, kad eksperimentas turi būti pakartojamas ir turėti aiškias instrukcijas, kaip tą padaryti. Buvo įgyta patirties atliekant eksperimentus, užtvirtinti pagrindiniai programų leidimo eksperimentų reikalavimai – daugiau nei vienas programos paleidimas, kuomet gaunami reikšmių vidurkiai, testavimo aplinkos aprašymas (kompiuterio parametrų specifikacija).

Buvo labai naudinga patirtis įvertinti savo algoritmo teorinį sudėtingumą ir vėliau palyginti jį su gautais praktiniais rezultatais. Atsiskleidė papildomi praktinio eksperimento vertinimo niuansai, kiekvienas nukrypimas ar rezultatas, kurio mažiau tikėtasi, turėjo būti interpretuojamas, pagrindžiamas.

Šio tyrimo praplėtimui būtų galima panaudoti galingesnę kompiuterinę techniką ir atlikti daugiau testų su didesniais viršūnių ir svorių skaičiais, parinkti daugiau kombinacijų tarp šių reikšmių, ir taip pat, skiriant tam daugiau laiko, atlikti kiekvieną testą po daugiau nei 10 kartų (pvz. 100 kartų), tuomet reikšmių vidurkiai labiau atitiktų testuojamus parametrus.

Literatūra ir šaltiniai

Trumpiniai:

[WIKI_TSP]: https://en.wikipedia.org/wiki/Travelling_salesman_problem [žiūrėta 2021-05-23]

Sąrašas:

1. Atstumų tarp Lietuvos miestų lentelė. Lietuvos HSM duomenų archyvas. http://www.lidata.eu/en/index.php?file=files/mokymai/stat/stat.html&course_file=stat_II_10_1.html [žiūrėta 2021-05-23]

2. [WIKI_TSP]: Sekcija “History”.

3. P. Crescenzi, V. Kann. A compendium of NP optimization problems. https://www.csc.kth.se/~viggo/wwwcompendium/node106.html#5296 https://www.csc.kth.se/~viggo/wwwcompendium/node105.html#5257 [žiūrėta 2021-05-23]

4. Šaltinio autoriai ir pavadinimas taip pat kaip ir [3]. https://www.csc.kth.se/~viggo/wwwcompendium/node104.html#5232 [žiūrėta 2021-05-23]

5. V. Dičiūnas, *Algoritmų Analizės Pagrindai*, 2005, skyrelis 3.4. https://drive.google.com/drive/u/0/folders/1xHdGUk08GYkVUFf6SuCfuGt0ie5olc2x [žiūrėta 2021-05-03]

6. [WIKI_TSP]: Įvadinė sekcija.

7. R. Radharamanan, L.I.Cho, *A branch and bound algorithm for the travelling salesman and the transportation routing problems,* Computers & Industrial Engineering Volume 11, Issues 1–4, 1986, pp. 236-240, https://www.sciencedirect.com/science/article/abs/pii/0360835286900859 [žiūrėta 2021-05-28]

8. С. Гудман, С. Хидетниеми, *Введение в разработку и анализ алгоритмов*, Мир, Москва, 1981, pp. 129–143 (rusų k.).
   Tai yra vertimas iš S. E. Goodman, S. T. Hedetniemi, *Introduction to the design and analysis of algorithms*, McGraw-Hill, New York, 1977.
   https://catalog.princeton.edu/catalog/SCSB-5950792 [žiūrėta 2021-05-31]

9. R. Čiegis, *Duomenų Struktūros, Algoritmai ir jų Analizė*, Technika, Vilnius, 2007, pp. 262–266 https://www.ebooks.vgtu.lt/product/duomen-struktros-algoritmai-ir-j-analiz [žiūrėta 2021-05-23]

10. E. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, New Jersey, 1977, pp. 121–126.
https://biu.primo.exlibrisgroup.com/discovery/fulldisplay?vid=972BIU_INST:972BIU&search_scope=MyInstitution&tab=LibraryCatalog&docid=alma990002570450205776&lang=en&context=L&adaptor=Local%20Search%20Engine&query=sub,exact,Combinatorial%20analysis%20--%20Data%20processing,AND&mode=advanced
[žiūrėta 2021-05-31]

A Priedai

A.1 Programos vykdomasis failas `tsp_branch_bound.py`

A.2 Pavyzdinis įvesties failas `input.example.txt`

A.3 Skirtingi įvesties failai, su kuriais buvo atliekami programos korektiškumo testai, pateikti `input_tests/` aplanke. Iš viso – 10 failų, kartu su failu `run_input_tests.sh` su Linux komandinės eilutės instrukcijomis testams (`for` ciklu). Šiam failui paleisti reikia nukopijuoti vykdomąjį failą į šią direktoriją.

A.4 Aplankas `experiment_data/` su šiais jame esančiais aplankais kiekvienai tyrimo daliai:

```
test_vertices/
test_arcs/
test_weights/
```

kuriuose yra `.out` plėtinio failai su programos išvestimis ir pagalbinis failas `run_tests.sh` su Linux komandinės eilutės instrukcijomis, kurių pagalba buvo leidžiami testai. Šiam failui paleisti reikia nukopijuoti vykdomąjį failą į minėtą(as) direktoriją(as).

A.5 Excel failas su tyrimo rezultatais ir grafikais `experiment.xlsx`