VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
KOMPIUTERINIO IR DUOMENŲ MODELIAVIMO KATEDRA

Magistro baigiamasis darbas

Tvarkaraščių sudarymo metodų tyrimas

Atliko:
Andrej Davidovič parašas

Vadovas:
Prof. dr. Tadas Meškauskas

Vilnius
2019

TURINYS

Santrauka

Tvarkaraščių sudarymo uždavinys apima pakankamai platų spektrą – tai ir mokymų įstaigų tvarkaraščių sudarymas, ir transporto tvarkaraščių sudarymas, ir registrų paskirstymas mikroprocesoriuose, ir Sudoku galvosūkis, ir kt. Tvarkaraščių sudarymo problema daugeliu atvejų suvedama į grafo spalvinimo uždavinį, kurio tikslas nuspalvinti grafą tokiu būdu, kad bet kurios grafo viršūnės sujungtos briauna turėtų skirtingas spalvas. Dėl šios priežasties darbe buvo tiriami įvairūs grafo spalvinimo algoritmai, kurių dėka sudaromi tvarkaraščiai. Įgyvendinti sekantys algoritmai nagrinėjamai problemai spręsti: pilno perrinkimo, godusis, atkaitinimo modeliavimo, genetinis ir Tabu paieškos. Pastarieji algoritmai ištirti trijų tipų: pilnų, reguliarių ir atsitiktinių – įvairaus dydžio sintetinių grafų pagalba. Įgyvendinta internetinė aplikacija, kuri leidžia švietimo įstaigoms sudaryti tvarkaraščius euristinio (atkaitinimo modeliavimo) algoritmo dėka.

Summary

Analysis of Scheduling Methods

Timetable scheduling is one of the most common tasks that people face every day. Timetable scheduling problem covers a wide range of tasks such as organizing the timetables of educational institutions; traffic flows; register allocation in microprocessors; finding the solution to the Sudoku puzzle, etc. Scheduling theory is closely related to graph theory – timetable scheduling problem is usually converted into a graph coloring approach, which purpose is to find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. Therefore, various graph coloring algorithms were used in this thesis to create timetables. The following algorithms were implemented to address the issue at hand: brute force, greedy, simulated annealing, genetic and Tabo search. These algorithms were investigated by developing three types of different size synthetic graphs: complete, regular and random. The analysis showed that the greedy algorithm has a fast computational speed. Compared to other heuristic methods this algorithm has always proved to be the fastest, but at the same time providing the worst results. Furthermore, algorithms comparison study found that simulated annealing method provides the most accurate and close to optimal solutions. Genetic algorithm analysis showed that it is enough to generate one new generation to find the optimal solution for complete graphs. Moreover, it was determined that crossover probability p_c has no effect on algorithm accuracy, but it affects the execution time of the algorithm: the higher the value of p_c is, the longer the calculation time will be. Finally, a web application was implemented that allows educational institutions to schedule timetables by using the heuristic (simulated annealing) algorithm.

ĮVADAS

Tvarkaraščių sudarymas yra vienas iš dažniausiai pasitaikančių uždavinių, su kuriuo, net ir nepastebėdami to, visi žmonės susiduria kas dieną – planavimo poreikis lydi juos visą gyvenimą. Kiekvienas žmogus užsiima asmeninio laiko planavimu, darbuose žinoma, kas turi būti baigta iki nustatyto termino, o taip pat įsivaizduojama, kiek laiko prireiks kiekvienam uždaviniui atlikti. Remiantis turima informacija ir derinant laike atliekamus darbus, sudaromas tvarkaraštis. Dažniausiai asmeninio tvarkaraščio sudarymas nesukelia sunkumų. Beveik visi žmonės vadovaujasi intuicija, bandydami viską daryti laiku.

Tvarkaraščių sudarymas atsiranda visur, kur planuojamas darbų atlikimo eiliškumas: skirstant darbus gamyboje, registrus mikroprocesoriuose, valdant informacijos srautus, sudarant lėktuvų tūpimo tvarkaraščius, planuojant aukštosiose mokyklose paskaitų laiką ir kitose srityse. Tvarkaraščio sudarymas yra pakankamai sudėtingas matematinis uždavinys, kurį galima apibrėžti taip: tai tam tikros baigtinės aibės paskirstymo procesas, kuris ribojamas tam tikrų resursų ištekliais. Sunkumai sudarant tvarkaraščius atsiranda tuomet, kai yra daug uždavinių, reikia atsižvelgti į daugybę papildomų sąlygų, sudaryti tvarkaraštį ne vienam asmeniui, bet visam kolektyvui, t. y. kai yra daugialypiai konfliktai ir būtina suderinti ne vieną interesą. Įsivaizduokite, pavyzdžiui, šimtus darbų ir dešimtis vykdytojų, kuriems reikia sudaryti tvarkaraštį. Sprendžiant tokius uždavinius, buvo parengtos bendrosios tvarkaraščių sudarymo rekomendacijos, principai ir metodai. Dėl to panašios užduotys pradėtos tirti specialiosios mokslo dalies rėmuose – tvarkaraščių teorijos.

Tvarkaraščių teorijos uždaviniai yra glaudžiai susiję su grafų teorija, o būtent su vienu iš jos grupės uždavinių – grafo spalvinimo problema. Grafų teorijoje grafų spalvinimas (angl. *Graph coloring*) yra ypatingas grafo žymėjimo atvejis. Grafo spalvinimas dar kitaip vadinamas grafo dažymu yra grupė uždavinių, kuriuose siekiama grafo elementams priskirti spalvas taip, kad būtų tenkinamos tam tikros sąlygos (paprastai – kad gretimi grafo elementai turėtų skirtingas spalvas). Iš tokių uždavinių dažniausiai naudojamas viršūnių spalvinimas, kai kiekvienai viršūnei priskiriama spalva taip, kad gretimos viršūnės turėtų skirtingas spalvas, kiek rečiau – briaunų spalvinimas, kai spalvos priskiriamos briaunoms.

Grafų spalvinimas naudojimas ne tik teoriniams uždaviniams spręsti, bet plačiai taikomas daugelyje praktinių sričių. Iš pradžių grafų dažymas buvo naudojamas geografiniams žemėlapiams sudaryti. Šiandien grafų spalvinimas (ypač dažymas naudojant mažiausią spalvų skaičių) naudojamas, kaip jau minėta, sudaryti lėktuvų skrydžių, aukštųjų mokyklų paskaitų ir kitokio pobūdžio tvarkaraščius, o taip pat paskirstyti registrus mikroprocesoriuose, išspręsti populiarųjį galvosūkį Sudoku ir kitose srityse (plačiau žr. 3.3 skyrelį).

Toliau šiame darbe bus išsamiai nagrinėjamas tvarkaraščio sudarymo uždavinys, kurio sprendimas remsis grafų spalvinimo problematika.

Darbo objektas

Tvarkaraščių sudarymo algoritmai, jų parametrų derinimo ir efektyvumo analizė.

Darbo problema

Tvarkaraščių sudarymas – sudėtingas kombinatorinio optimizavimo uždavinys, kuriuo metu būtina paskirstyti turimus išteklius laike taip, kad būtų tenkinami keliami reikalavimai. Pastaroji problema gali būti suvedama į grafo spalvinimo uždavinį, kurio tikslas nuspalvinti grafą tokiu būdu, kad bet kurios grafo viršūnės, sujungtos briauna, turėtų skirtingas spalvas. Dėl šios priežasties darbe bus tiriami įvairūs grafo spalvinimo (tvarkaraščių sudarymo) algoritmai.

Darbo tikslas

Ištirti tvarkaraščių sudarymo algoritmus bei pateikti rekomendacijas dėl jų panaudojamumo.

Darbo uždaviniai

Pastarasis tikslas apima šių pagrindinių uždavinių įgyvendinimą:

1. Atlikti tvarkaraščių sudarymo ir grafo spalvinimo problemos susijusių tyrimų analizę.
2. Išnagrinėti grafų teorijos pagrindines sąvokas, reikalingas spalvinimo problemai tirti.
3. Pristatyti grafo spalvinimo algoritmus, kurių pagalba sudaromi tvarkaraščiai.
4. Išskirti algoritmų vertinimo kriterijus.
5. Pagal minėtus vertinimo kriterijus atlikti išsamią algoritmų analizę.
6. Po atliktos analizės išrinkti geriausią metodą ir sukurti internetinę aplikaciją, kuri sudaro tvarkaraščius pasirinkto euristinio[1] algoritmo pagrindu.

Darbo struktūra

Darbas sudarytas iš penkių dalių. Pirmoje ir antroje darbo dalyse atitinkamai supažindinama su tvarkaraščių sudarymo problema ir pagrindinėmis grafo teorijos sąvokomis. Trečias skyrius skirtas detaliai išnagrinėti grafo spalvinimo problemą, pateikti galimus uždavinio sprendimo algoritmus ir aptarti grafų spalvinimo praktinį pritaikymą. Ketvirtame skyriuje pristatoma išsami grafo spalvinimo (tvarkaraščių sudarymo) algoritmų, kurie realizuoti *JetBrains PyCharm Community Edition 2018* programinės įrangos ir *Python* programavimo kalbos pagalba, analizė pagal išskirtus pagrindinius vertinimo kriterijus. Penktame skyriuje pateikiamas formalus mokymosi įstaigų tvarkaraščių sudarymo uždavinys ir pristatoma sukurta „aDa tvarkaraščiai“ internetinė aplikacija, kuri realizuota panaudojant *.NET Core 2.2* karkasą, *ReactJS* biblioteką (su *Redux* architektūra) ir *Bootstrap 4* karkasą, pastarajam uždaviniui spręsti. Be to, internetinė aplikacija įgyvendinta *Microsoft Visual Studio Professional 2017*, *Microsoft Visual Studio Code* programinės įrangos ir *C#*, *TypeScript* programavimo kalbų pagalba. „aDa tvarkaraščiai“ internetinę aplikaciją galima išbandyti adresu: http://78.60.221.238[2].

2 skyriaus 2.1 ir 2.2 poskyriai naudojami iš mokslo tiriamojo darbo, išskyrus kelis pakeitimus ir papildymus.

[1] Euristika – uždavinių sprendimo metodika, kai sprendimas gaunamas bandymų ir klaidų keliu. Tai gali būti ne optimalus, tačiau greitas ir mažiau sąnaudų reikalaujantis sprendimo būdas.

[2] Kadangi aplikacija yra aptarnaujama iš autoriaus nešiojamojo kompiuterio, tai yra galimybė, kad ji kai kuriuo momentu gali būti trumpam nepasiekiama, pavyzdžiui, dėl kompiuterio transportavimo priežasčių.

1. Tvarkaraščių sudarymo problema

Kalbant apie tvarkaraščių sudarymą nederėtų apsiriboti vien tik mokymo įstaigų ir darbo tvarkaraščiais (kaip yra įprasta manyti) – ši sąvoka apima žymiai platesnį spektrą. Tokie žinomi uždaviniai kaip registrų paskirstymas mikroprocesoriuose ir galvosūkis Sudoku taip pat priklauso tvarkaraščių kategorijai. Su tvarkaraščių sudarymo problema susiduriama visur, kur būtinas resursų paskirstymas. Bendrai planavimo idėja gali būti apibrėžta sekančiu būdu: paskirstyti susijusius išteklius laike, tenkinant įvairių tipų esminius ir pirmenybinius apribojimus, kuriais siekiama sukurti optimalų suderintą tvarkaraštį. Priklausomai nuo srities ir tvarkaraščių pobūdžio juos galima skirstyti į tokias tipines planavimo problemas:

- *Mokymo įstaigų tvarkaraščių sudarymas:*
  - *Paskaitų tvarkaraščio sudarymas* – paskaitos, kurioms pravesti naudojami bendri ištekliai (pavyzdžiui, auditorijos, dėstytojai ir kita), turi būti suderintos laike.
  - *Egzaminų tvarkaraščio sudarymas* – egzaminai, kurie vykdomi naudojant bendrus išteklius, turi būti suderinti laike.
- *Transporto tvarkaraščių sudarymas* – transporto priemonė turi būti priskirta reisams. Pavyzdžiui, lėktuvų skrydžių tvarkaraščio sudarymo atveju orlaiviai – skrydžiams.
- *Darbo grafikų sudarymas* – efektyviai sudėlioti darbo laiką dideliam darbuotojų ir įrenginių kiekiui, taip optimizuojant resursų sąnaudas.
- *Informacijos srautų valdymas kompiuterių tinkluose* – suplanuoti informacijos paketų perdavimo eiliškumą.

Taip pat tvarkaraščių uždavinius galima klasifikuoti ir pagal ieškomo sprendinio tipą, t. y. koks yra tvarkaraščio sudarymo tikslas:

- *Tvarkos nustatymo uždavinys.* Šiuose uždaviniuose jau yra nustatytas darbų paskirstymas pagal vykdytojus bei apibrėžti visi darbų parametrai (atlikimo trukmė, gavimo laikas ir kt.). Būtina sudaryti kiekvieno vykdytojo darbų atlikimo tvarkaraštį (arba eiliškumą).
- *Suderinamumo uždavinys.* Pagrindinis dėmesys šiuose uždaviniuose skiriamas darbų atlikimo trukmės, gavimo laiko ir kitų parametrų parinkimui.
- *Paskirstymo uždavinys.* Šie uždaviniai apima optimalaus darbų paskirstymo paiešką pagal vykdytojus.

Iš esmės daugelis tvarkaraščių sudarymo uždavinių yra optimizacinio pobūdžio, t. y. susideda iš pasirinkimo (nustatymo) tų sprendimų tarp leistinų tvarkaraščių, kuriais pasiekiama optimali tikslo funkcijos reikšmė. Paprastai optimalumas reiškia minimalią arba maksimalią tam tikros tikslo funkcijos reikšmę. Tvarkaraščio leistinumas suprantamas jo įgyvendinamumo prasme, o optimalumas – jo tikslingumo prasme.

1.1. Susijusių darbų analizė

Tvarkaraščių sudarymo uždavinių sprendimas kompiuterių pagalba turi pakankamai ilgą ir įvairią istoriją. 1967 m. grafo spalvinimas buvo pritaikytas paskaitų tvarkaraščio problemai [27]. Tais metais Welsh ir Powell [28] iliustravo ryšį tarp tvarkaraščių sudarymo ir grafų spalvinimo bei sukūrė naują bendrą grafų spalvinimo algoritmą, kuris leido efektyviau išspręsti (arba apytiksliai išspęsti) minimalaus spalvų skaičiaus problemą. Jiems taip pat sėkmingai pavyko spalvinti grafus, kylančius iš tvarkaraščių sudarymo problemų, o konkrečiau – iš egzaminų tvarkaraščių sudarymo. 1969 m. Wood pasiūlė grafo algoritmą [29], kuris naudojo dvi $n \times n$ matricas, čia n žymi grafo viršūnių skaičių; konfliktinė matrica C buvo naudojama norint parodyti, kurios viršūnių po-

ros turi būti nuspalvintos skirtingai dėl uždavinyje nustatytų apribojimų, ir panašumo matrica S – apibrėžti, kurios viršūnių poros turi būti tos pačios spalvos. 1981 m. Dutton ir Bingham pristatė du populiariausius euristinius grafų spalvinimo algoritmus. Peržiūrint visas spalvas vieną po kitos, formuojama klika[3] nuolat jungiant dvi viršūnes su labiausiai paplitusiomis gretimomis viršūnėmis. Pabaigoje identiškas dažymas pritaikomas visoms sujungtoms viršūnėms.

1986 m. Carter [6] savo tyrime apie egzaminų tvarkaraščius remėsi grafų spalvinimo algoritmais ir euristikomis bei parodė, kad grafo teorinis požiūris yra vienas populiariausių. Šis būdas buvo priimtas ir taikomas daugelyje švietimo įstaigų, kad išspręstų savo egzaminų tvarkaraščių sudarymo problemas. 1991 m. Johnson, Aragon, McGeoch ir Schevon [14] įgyvendino ir išbandė tris skirtingus grafų spalvinimo būdus su atkaitinimo modeliavimo metodu, pastebėdami, kad atkaitinimo modeliavimo algoritmai gali pasiekti gerų rezultatų, tačiau tik tuo atveju, jei leidžiama pakankamai didelė veikimo trukmė. 1992 m. Kiaer ir Yellen [15] straipsnyje aprašė euristinį algoritmą grafų spalvinimui, kurio pagalba siekė sudaryti universitetų paskaitų tvarkaraščius. Pasiūlyto algoritmo dėka naudojant svorinį grafą modeliuojamas uždavinys, kuriuo siekiama rasti kuo mažiausias grafo *k*-spalvinimo sąnaudas (k yra galimų laiko intervalų skaičius) su minimaliu konfliktų skaičiumi. 1994 m. Burke, Elliman ir Weare [5] pristatė planus dėl universitetų tvarkaraščių sudarymo sistemos, pagrįstos grafų spalvinimu ir apribojimų manipuliacijomis. Autoriai aprašė grafų spalvinimo ir patalpų paskirstymo euristinius metodus kartu iliustruodami kaip gali būti sujungti šie du elementai, kad būtų sukurtas tvarkaraščių sistemos pagrindas. Tyrėjai taip pat aptarė kelių bendrų tvarkaraščių sudarymo funkcijų tvarkymą sistemoje, visų pirma atsižvelgiant į egzaminų tvarkaraštį. 1995 m. buvo pristatytas grafų spalvinimo metodas, kurio tikslas – optimizuoti tvarkaraščių sudarymo uždavinių sprendimus. Bresina (1996 m.) buvo vienas iš ankstyvųjų mokslininkų, kuris naudojo šį būdą ir įnešė pokyčius į rankinį požiūrį, kuris buvo vykdomas universitetuose [4]. 2007 m. universiteto tvarkaraščiui buvo pateiktas alternatyvus grafų spalvinimo metodas, kuris dažymo proceso metu įtraukė ir patalpų paskirstymą. 2008 m. C++ programavimo kalba buvo sukurta Koala grafų spalvinimo biblioteka, apimanti daugelį praktinių grafų spalvinimo pritaikymų [8]. 2009 m. buvo pasiūlyti automatizuoti aproksimavimo algoritmai, skirti spręsti minimalaus viršūnių spalvų skaičiaus uždavinį. Be to, pastaraisiais metais mokslininkai tiria naujus alternatyvius metodus, skirtus planavimo problemoms spręsti siekiant geresnio rezultato.

1.2. Tvarkaraščių sudarymo algoritmų sudėtingumas

Tvarkaraščių sudarymui naudojama daug skirtingų algoritmų, kurie apdoroja apibrėžtus tvarkaraščio pradinius duomenis. Dauguma iš jų priklauso NP-sudėtingumo klasei (angl. *nondeterministic polynomial time*). Dėl šios priežasties tik mažos apimties uždaviniams gali būti rastas tikslus sprendinys per priimtiną skaičiavimų laiką [10]. Praktikoje paprastai ieškomas apytikris sprendinys, kuris būtų kuo artimesnis optimaliam. Taikant tam tikrus klasikinius programavimo algoritmus galima rasti praktiniu požiūriu priimtinus sprendinius. Šie sprendiniai gali būti pagerinti įvairiais euristiniais metodais, dauguma kurių paremti gamtos natūraliais procesais.

Atskiru atveju tvarkaraščių sudarymą galima apibūdinti kaip laiko intervalų priskyrimą visiems darbams esantiems sąraše tokiu būdu, kad kiekvienas darbas turėtų savo laiko intervalą, o darbai kurie negali vykti kartu – turėtų skirtingus laiko intervalus [22]. Paprastai pastarasis uždavinys suvedamas į grafo spalvinimo uždavinį (plačiau žr. 3 skyrių) ir yra eksponentinio sudėtingumo.

[3] Klika vadinamas grafo G poaibis, jeigu bet kurios dvi skirtingos, jei tokių yra, šio poaibio viršūnės yra sujungtos briauna.

Visgi, jei gaunamas intervalinis grafas[4], pastarojo spalvojimui taikomas polinominio sudėtingumo algoritmas. Polinominio sudėtingumo (angl. *polynomial time computable*) uždaviniai, kurių klasė žymima P, dažnai vadinami praktiškai išsprendžiamais uždaviniais, nes jų sprendimo laikas (polinominiais algoritmais) yra gana trumpas. Taikydami šiuos algoritmus galime ieškoti tikslių sprendinių.

Nors tvarkaraščių sudarymo uždavinių, kurie sprendžiami grafų pagalba, pakankamai daug, tačiau sprendžiant praktines situacijas keliami reikalavimai išplečia uždavinio sudėtingumą tiek, kad išspręsti uždavinį per priimtiną laiką nebepavyksta. Vis dėlto iš anksto atsižvelgus į metodo vykdymo laiko sudėtingumą galima nustatyti, ar taikomas problemos sprendimo algoritmas yra tinkamas turimam duomenų kiekiui.

Algoritmų sudėtingumas paprastai žymimas $O(f(n))$, čia $f(n)$ – funkcija parodanti, kaip auga algoritmui reikalingos atminties dydis arba vykdymo laikas didėjant pradiniams duomenims n. $O(1)$ reiškia, kad algoritmo sudėtingumas nepriklauso nuo duomenų kiekio, t. y. jis pastovus, konstantinis; $O(n)$ – priklauso tiesiškai nuo duomenų kiekio ir t. t. Metodai, kurių sudėtingumas nuo $O(1)$ iki $O(n^k)$, priklauso polinominio sudėtingumo klasei P. Tačiau egzistuoja daug svarbių taikomųjų uždavinių, kuriems kol kas sukurti tik eksponentinio $O(k^n)$ arba faktorialinio $O(n!)$ sudėtingumo algoritmai. Pastarieji priklauso NP-sudėtingumo klasei.

Šiame darbe sprendžiamas tvarkaraščių sudarymo uždavinys yra NP-sudėtingumo klasės. Be to, gaunamas grafas šiam uždaviniui spręsti nėra intervalinis, todėl paprastai nepavyks sumažinti uždavinio sudėtingumo. Norint sprendimą rasti per priimtiną laiką, būtina naudoti euristinius metodus, kurie, nors ir negarantuoja optimalaus sprendimo, bet tikėtina, kad ras jam artimą.

Optimizavimo uždaviniams spręsti gali būti taikomi įvairūs euristiniai paieškos algoritmai. Vieni iš žinomiausių tokių metodų pavyzdžių yra šie:

- godusis algoritmas (angl. *greedy algorithm*);
- atkaitinimo modeliavimo algoritmas (angl. *Simulated Annealing algorithm*);
- genetinis algoritmas (angl. *genetic algorithm*);
- paieška su draudimais algoritmas (angl. *Tabu Search algorithm*);
- skruzdžių kolonijos algoritmas (angl. *Ant Colony Systems algorithm*);
- spiečių algoritmai (angl. *Swarm algorithms*).

Kai kurios euristinės tvarkaraščių optimizavimo paradigmos detaliau aptariamos 3.2 skyrelyje.

[4]Intervalinis grafas – intervalų aibės sankirtų tiesėje grafas. Toks grafas turi po vieną viršūnę kiekvienam aibės intervalui ir po briauną tarp kiekvienos viršūnių poros, jei atitinkami intervalai susikerta. Kitaip tariant, intervalinio grafo viršūnės yra tam tikri intervalai tiesėje, o bet kurios dvi viršūnės yra sujungtos briauna tada ir tik tada, kai jas atitinkantys intervalai susikerta.

2. Grafų teorijos elementai

Grafų teorijos kaip matematinės disciplinos pradžia siejama su garsiuoju L. Oilerio 1736 m. nagrinėtu uždaviniu apie Karaliaučiaus (Kionigsbergo) tiltus. Tačiau maždaug šimto metų laikotarpyje šio uždavinio sprendinys buvo vienintelis grafų teorijos rezultatas. Didesnio susidomėjimo grafų teorija ir su ja susiję klausimai sulaukė sparčiai plėtojant elektros tinklų, kristalografijos, organinės chemijos ir kitų mokslo sričių tyrimus.

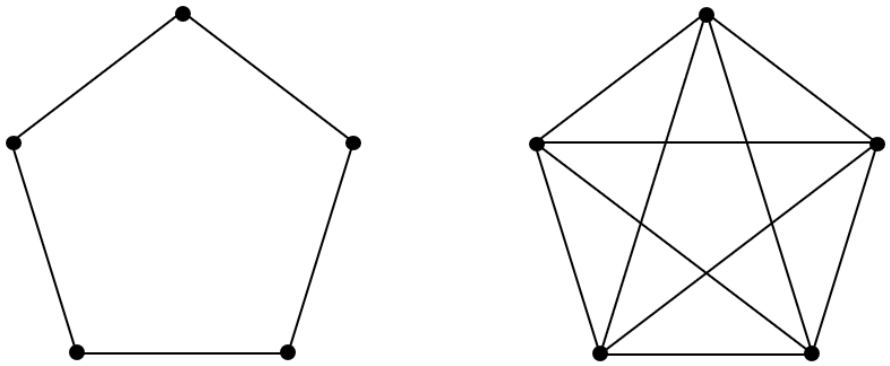
Grafų teorija aktyviai vystėsi ir pastaruoju metu grafai ir su jais susiję tyrimo metodai skirtingais lygmenimis prasiskverbė beveik į visas šiuolaikinės matematikos sritis. Bendrąja prasme grafas – tai reiškinys sutinkamas įvairiose gyvenimo srityse: planavimo ir valdymo teorijoje, tvarkaraščių sudaryme, ekonomikoje, elektros, telefono, kelių arba geležinkelio tinkluose ir daugelyje kitų sričių. Grafai padeda vizualizuoti objektų ar įvykių tarpusavio ryšius sudėtingose sistemose. Tam, kad turėtume tikslų įsivaizdavimą apie tyrimo objektą, būtina apibrėžti darbe naudojamas pagrindines sąvokas. Šis skyrius skirtas trumpai grafų teorijos apžvalgai: pateikiamas grafo apibrėžimas ir pagrindiniai jo elementai.

2.1. Grafo sąvoka

Sąvoka grafas yra gimininga sąvokoms grafinis, grafika. Iš tiesų, grafų modeliai turi paprastą ir suprantamą grafinę interpretaciją, kuri leidžia neišeinant iš griežtų matematinių rėmų vaizdžiai pavaizduoti įvairiausius objektus.

Norėdami apibūdinti įvairių sistemų, susidedančių iš tarpusavyje sujungtų elementų, struktūrą dažnai naudojamos grafinės schemos, kuriose elementai atvaizduojami taškais (apskritimais, stačiakampiais ir t. t.), o ryšiai tarp jų – linijomis. Kartu jie sudaro tai, ką matematikai vadina grafu. Pirmasis grafo terminą įvedė matematikas D. Kionigas 1936 m. Šiuo metu mokslinėje literatūroje sutinkami keli grafo sąvokos apibrėžimai. Formaliai grafą apibrėžti galima taip:

1 apibrėžimas. Grafas $G = (V, E)$ – tai aibių pora; čia V – netuščioji aibė, o aibė E yra aibės V visų galimų dvielemenčių poaibių aibė, t. y. $E \subset V \times V$. Aibė V yra vadinama viršūnių aibe, o E yra vadinama briaunų aibe [30]. Kiekvienas elementas $e \in E$ yra aibės V elementų pora (v_i, v_j); čia viršūnės v_i ir v_j vadinamos briaunos e galais.



1 pav. Nepilnojo ir pilnojo grafų pavyzdžiai. Iliustracija kairėje: nepilnas grafas (ne visos viršūnės sujungtos su kitomis viršūnėmis tarpusavyje). Iliustracija dešinėje: pilnas grafas (visos viršūnės turi tiesioginę briauną su likusiomis viršūnėmis).

Šis grafo apibrėžimas turi būti papildytas vienu svarbiu aspektu. Apibūdinant briauną galima atsižvelgti arba neatsižvelgti į dviejų galų išdėstymo tvarką. Jei pastaroji nėra svarbi, t. y. $(v_i, v_j) = (v_j, v_i)$, tai e vadinama neorientuota briauna. Priešingu atveju, kai išdėstymo tvarka svarbi, e vadinama orientuota briauna arba lanku. Jei grafas turi tik briaunas, tai jis vadinamas neorientuotuoju grafu, o orientuotuoju grafu, jei – tik lankus. Kai kuriais atvejais natūralu tirti mišrius grafus, turinčius tiek briaunas, tiek lankus.

Nagrinėjant grafą, pažymėtina ir grafo pilnumo savybė. Grafas vadinamas pilnuoju, kai visos viršūnės yra sujungtos su kitomis viršūnėmis tarpusavyje. Pilnas grafas su n viršūnių, turi $\frac{n(n-1)}{2}$ briaunų ir žymimas K_n. Ir atvirkščiai, nepilnajame grafe ne visos viršūnės turi tiesioginę briauną su likusiomis viršūnėmis. 1 pav. pateikti nepilnojo ir pilnojo grafų pavyzdžiai.

2.2. Keliai, trasos ir takai grafe. Grafo jungumo savoka

Nagrinėjant grafų teoriją taip pat būtina paminėti tokias sąvokas, kaip: kelias, trasa, takas. Kelias yra apibrėžiamas taip:

2 apibrėžimas. Grafo $G = (V, E)$ viršūnių seka: $v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_N}$ – vadinama keliu, jei bet kuriuos du gretimus šios sekos elementus jungia briauna, t. y. $(v_{i_j}, v_{i_{j+1}}) \in E$ su visais $1 \leq j \leq N-1$ [30].

Atitinkamai trasos ir tako apibrėžimai yra formuluojami tokiu būdu:

3 apibrėžimas. Trasa grafe apibrėžiama kaip kelias, kuris neturi pasikartojančių briaunų. Be to, uždara trasa vadinama grandine. Tuo tarpu takas grafe – kelias, kurio visos viršūnės yra skirtingos. Ciklas – tai uždaras takas.

Iš prieš tai apibrėžtų sąvokų išplaukia grafo jungumo sąvoka:

4 apibrėžimas. Grafas $G = (V, E)$ yra jungus, kai bet kuriai aibės V elementų porai (v_i, v_j) egzistuoja jas jungiantis kelias, t. y. toks kelias, kur v_i – pradinė viršūnė, o v_j – galinė [30].

Briauna, kurią pašalinus grafas tampa nejungiu, vadinama tiltu arba sąsmauka.

2.3. Grafo viršūnės laipsnis. Plokščias grafas. Reguliarus grafas

Toliau darbe taip pat bus naudojama grafo viršūnės laipsnio sąvoka, kuri formuluojama sekančiu būdu:

5 apibrėžimas. Grafo $G = (V, E)$ viršūnės $v_j \in V$ laipsniu $\deg(v_j)$ vadinsime iš viršūnės v_j išeinančių briaunų skaičių [30].

Viršūnė vadinama lapu arba kabančia (angl. *pendant*), jei jo laipsnis lygus vienetui. Pavyzdžiui, aukščiau pateiktų nepilno ir pilno grafų (žr. 1 pav.) kiekvienos viršūnės laipsnis atitinkamai yra $\deg(v_j) = 2$ ir $\deg(v_j) = 4$.

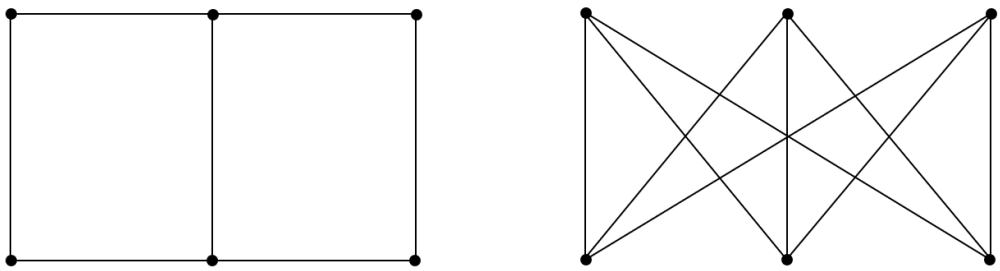
Visų grafo viršūnių laipsnių sumą randama pagal rankų paspaudimo teoremą (angl. *Handshaking Theorem*):

1 teorema. Tarkime, $G = (V, E)$ yra grafas. Tuomet visų grafo G viršūnių laipsnių suma yra lygi grafo briaunų skaičiui padaugintam iš dviejų:

$$\sum_{v_j \in V} \deg(v) = 2|E|$$

Kalbant apie grafo planarumo problemą, planarus arba plokščias grafas apibrėžiamas taip:

6 apibrėžimas. Grafas $G = (V, E)$ yra vadinamas planariuoju, jeigu jį galima pavaizduoti plokštumoje taip, kad jo briaunos nesikirstų niekur (neturėtų bendrų vidinių taškų), išskyrus viršūnių taškus [30].

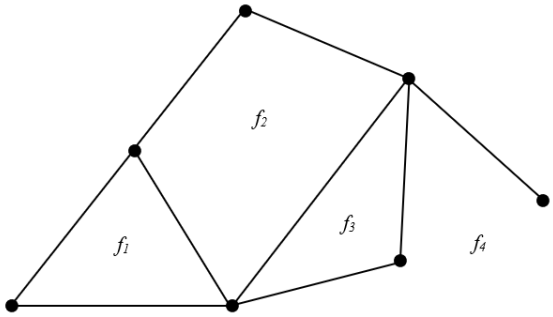


2 pav. Planariojo ir neplanariojo grafų pavyzdžiai. Iliustracija kairėje: planarus grafas (briaunos nesikerta niekur tarpusavyje). Iliustracija dešinėje: neplanarus grafas (kai kurios briaunos kertasi tarpusavyje – turi bendrų vidinių taškų).

Vienas iš neplanariojo grafo pavyzdžių yra jau nagrinėtas 5 viršūnių pilnasis grafas (žr. 1 paveikslo iliustraciją dešinėje). 2 paveikslo iliustracijoje dešinėje taip pat atvaizduotas neplanarus grafas, kuris vadinamas „komunaliniu grafu“ (angl. *utility graph*) arba dar Tomseno vardu. Tokio grafo pagalba yra iliustruojama Oilerio suformuota trijų komunalinių paslaugų problema (angl. *three utilities problem*), kuri gali būti įvardinta taip: tarkime, kad yra trys namai, kiekvienas iš kurių turi būti prijungtas prie dujų, vandens ir elektros šaltinių. Toliau keliamas klausimas, ar įmanoma tai padaryti vienoje plokštumoje, kad trijų minėtų šaltinių linijos nesikirstų. Įrodyta, kad toks uždavinys sprendimo neturi, t. y. grafas yra neplanarus.

Grafo planarumo uždavinio pritaikymo spektras platus. Viena iš sričių – elektroninių grandinių gamyba. Elektroninės grandinės spausdinamos plokštelėje, kuri pagaminta iš izoliacinės medžiagos. Kadangi spausdinamos grandinės yra neizoliuotos, jos neturi kirstis. Iš čia kyla klausimas, kaip schemoje turėtų būti išdėstomi kontaktai, kad plokštelėje galima būtų atspausdinti nesikertančias grandines.

Su grafų planarumo savybe taip pat susiduriama, sprendžiant žemėlapių spalvinimo uždavinį. Pastebėtina, kad jeigu grafas gautas iš žemėlapio, tai tokį grafą galima pavaizduoti plokštumoje taip, kad jo briaunos nesikirstų. Kitaip tariant, jis bus planarus.



3 pav. Plokštumoje pavaizduotas grafas turi 4 regionus: f_1, f_2, f_3 ir f_4.

Verta paminėti, kad plokščias grafas turi papildomą atributą, kuris vadinamas regionu arba veidu (angl. *face*).

7 apibrėžimas. Tarkime, kad grafas $G = (V, E)$ yra planarus. Tuomet plokštumos sritis apribotas grafo briaunomis vadinsime regionais arba veidais [24].

Pažymėtina, kad su planariais grafais atlikta daug tyrimų, kurių dėka gauti reikšmingi rezultatai, iš kurių vienas žinomiausių yra Oilerio daugiakampių formulė.

2 teorema. Tarkime, $G = (V, E)$ yra jungus planarus grafas. Tuomet galioja tokia lygybė:

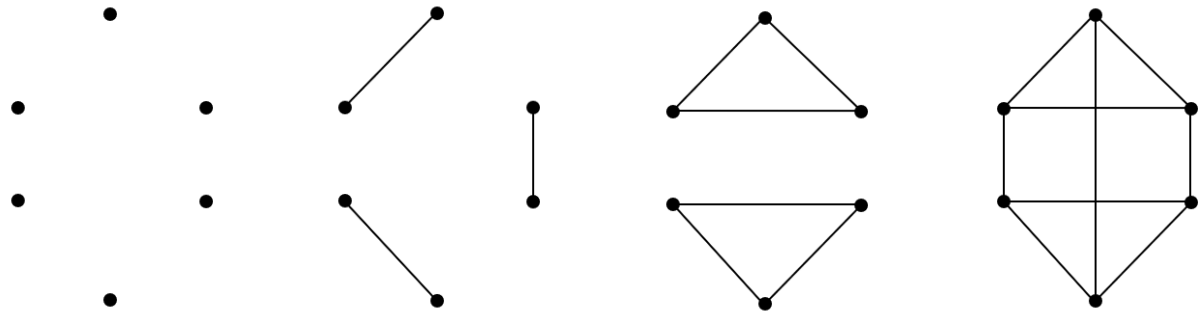
$$v - e + f = 2,$$

čia v yra grafo viršūnių skaičius, e – jo briaunų skaičius ir f – jo regionų skaičius.

Kalbant apie reguliarųjį grafą (angl. *regular graph*), jis apibrėžiamas taip:

8 apibrėžimas. Grafas $G = (V, E)$ yra vadinamas reguliariuoju, jei visų viršūnių laipsnis sutampa, t. y. kiekviena viršūnė turi vienodą kaimynų skaičių. Grafo reguliarumo laipsnis žymimas $r(G)$.

Reguliarūs grafai su $r(G) \leqslant 2$ klasifikuojami gana paprastai: 0-reguliarusis grafas sudarytas iš izoliuotų viršūnių – tuščias grafas; 1-reguliarusis grafas sudarytas iš izoliuotų briaunų; 2-reguliarusis grafas sudarytas iš išsklaidytų ciklų. Kai $r(G) > 2$, tai n-reguliarusis grafas dar vadinamas n-kubu. 4 paveiksle pateikti reguliarūs grafai.



(a) (b) (c) (d)

4 pav. Reguliariųjų grafų klasifikacija. Iliustracija *a*: 0-reguliarusis grafas. Iliustracija *b*: 1-reguliarusis grafas. Iliustracija *c*: 2-reguliarusis grafas. Iliustracija *d*: 3-reguliarusis (kubinis) grafas.

Verta paminėti, kad būtinos ir pakankamos k-reguliariojo grafo egzistavimo sąlygos yra $k+1 \leqslant n$ ir $n \times k$ – lyginis skaičius[5], čia n – grafo viršūnių skaičius, k – kiekvienos viršūnės laipsnis.

[5]Reguliariojo grafo visų briaunų skaičius lygus $\frac{n \times k}{2}$, todėl $n \times k$ privalo būti lyginis. Priešingu atveju (turint nelyginę sandaugą) gautume pusę briaunos, ko negali būti grafe.

3. Grafo viršūnių spalvinimo problema

Grafų teorija aktyviai vystėsi per pastaruosius kelis dešimtmečius. Tai yra susiję su tuo, kad grafų teorijos taikymo sritys sparčiai plečiasi, didėja užduočių, kurias galima išspręsti naudojant grafų teorijos metodus, skaičius. Pavyzdžiui, pagrindinė užduotis, susijusi su navigacija ar elektros, geležinkelio ir kitų tinklų tiesimu, yra surasti trumpiausią kelią grafe. Kitas aktualus uždavinys yra tvarkaraščių sudarymas. Pastarasis uždavinys apima bet kokio pobūdžio tvarkaraščių sudarymą: nuo mokyklos tvarkaraščio iki įmonėje atliekamų darbų sekos. Šio uždavinio apribojimai yra negalėjimas vienu metu atlikti užduočių (pamokų, darbo ir kt.) dėl tam tikrų priežasčių. Panaši problema nuolat kyla skirtingose situacijose, todėl jai spręsti reikalingi efektyvūs algoritmai.

Kalbant apie grafų spalvinimą, paprastai suprantama, kad kalba eina apie jų viršūnių spalvinimą, t. y. spalvų priskyrimas grafo viršūnėms tokiu būdu, kad bet kokios dvi viršūnės, kurios turi bendrą briauną, būtų skirtingų spalvų. Kadangi grafai, kuriuose yra kilpų[6], tokiu būdu negali būti nuspalvinti, jie nėra tyrimo objektas. Formaliai grafo spalvinimo uždavinys gali būti užrašytas taip:

9 apibrėžimas. Tarkime, kad turime grafą $G = (V, E)$ ir spalvų rinkinį C. Tuomet grafo nuspalvinimu yra vadinamas toks atvaizdis $f : V(G) \to C$, kad kiekvienai grafo briaunai $v_i v_j \in E(G)$ teisinga nelygybė $f(v_i) \neq f(v_j)$ [17]. Kartais toks spalvinimas vadinamas korektišku.

Viena iš sudėtingesnių grafų teorijos problemų yra grafo chromatinio skaičiaus (paprastai žymimas $\chi(G)$) nustatymas, t. y. minimalaus spalvų skaičiaus, reikalingo grafo viršūnėms nuspalvinti, radimas. Toks uždavinys priskiriamas NP-sunkumo uždavinių klasei. Šiam uždaviniui spręsti yra pasiūlyti įvairūs algoritmai, tačiau veiksmingo algoritmo paieška tęsiasi. Viršūnių spalvinimas leidžia modeliuoti daugelį planavimo problemų, o būtent grafo spalvinimo algoritmo pagalba gali būti išspręstas tvarkaraščių sudarymo uždavinys.

3.1. Ankstesnių tyrimų analizė

Nemažai tyrimų, kuriuose mokslininkai vertino grafų spalvinimo problemą ir jos sprendimo algoritmus, yra atlikta. Vienas iš tokių yra M. Aslan ir N. A. Baykan (2016) darbas, kuriame buvo vertinami įvairių euristinių algoritmų efektyvumas ir našumas grafo spalvinimo uždaviniui spręsti. Tyrime išanalizuoti pirmo tinkamo (angl. *First Fit*), išrikiavimo pagal didžiausią laipsnį (angl. *Largest Degree Ordering*), Welsh ir Powell, išrikiavimo pagal prisotinimo laipsnį (angl. *Saturated Degree Ordering*) ir rekursyviai pirmo didžiausio (angl. *Recursive Largest First*) algoritmai (toliau – atitinkamai FF, LDO, WP, SDO ir RLF), kurie literatūroje siūlomi viršūnių dažymo problemai nagrinėti. Algoritmų veiksmingumas buvo palygintas pagal sprendimo kokybę ir skaičiavimo laiką. Tyrimui atlikti autoriai pasirinko ne atsitiktinius grafus, bet specifinius, pavyzdžiui: registrų paskirstymo (angl. *Register Allocation*), Mycielski, knygos pavidalo (angl. *Book Graph*) ir kt. Eksperimentiniai rezultatai parodė, kad RLF ir SDO algoritmai yra pakankami grafo spalvinimo problemai, tuo tarpu kai FF metodas paprastai nepakankamas gauti artimą rezultatą optimaliam. Taip pat nustatyta, kad registrų paskirstymo, Mycielsky, knygos grafams WP algoritmas rasdavo geriausią sprendimą per trumpiausią laiką. Apibendrinant, autoriai priėjo prie išvados: visų pirma, reikia įsitikinti, kad sprendžiama problema yra artima grafo spalvinimo uždaviniui ir gali būti sugretinta su vienu iš standartizuotų grafų. Tuomet galima pritaikyti atitinkamus grafo spalvinimo algoritmus, kad būtų rastas geriausias sprendimas ir taip sutaupyti laiko [2].

[6] Kilpa – briauna, kurios pradžia ir galas sutampa.

Panašų tyrimą keleriais metais anksčiau atliko A. Mansuri, V. Gupta ir R. S. Chandel (2010). Šiame darbe buvo nagrinėjami keturi euristiniai algoritmai: FF, LDO, išrikiavimo pagal incidenčių laipsnį (angl. *Incident Degree Ordering*) (toliau – IDO) ir SDO – ir pasiūlytos jų modifikacijos. Autoriai tyrimui atlikti naudojo atsitiktiniu būdu sugeneruotus grafus, kurie turėjo skirtingus viršūnių skaičių ir tankumą. Gauti rezultatai parodė, kad pagal apskaičiuotą chromatinį skaičių prasčiausias buvo FF algoritmas. Empiriškai nustatyta, kad LDO naudodavo mažesnį spalvų skaičių nei IDO, kai tuo tarpu SDO naudodavo mažesnį spalvų skaičių už abu IDO ir LDO algoritmus. Taip pat autoriai darbe pasiūlė du hibridinius algoritmus, sujungdami LDO su IDO ir SDO su LDO. Atlikus eksperimentus modifikuotų algoritmų pagalba nustatyta, kad hibridiniai metodai yra geresni nei originalūs [20].

Grafų spalvimo problemą ir jos sprendimo metodus taip pat nagrinėjo H. Ayanegui ir A. Chavez-Aragon. Norėdami nustatyti sugeneruoto grafo chromatinį skaičių, tyrėjai taikė atkaitinimo modeliavimo algoritmą. Nors šiuo metodu buvo gaunami pakankamai neblogi rezultatai, tačiau sprendimai ne visada buvo optimalūs. Dėl to tam, kad gautų tikslesnius rezultatus, tyrėjai pasiūlė algoritmą, pagrįstą priimtino slenksčio (angl. *Threshold Accepting*) ir Davis-Putman metodais. Atlikti skaičiavimai parodė, kad tyrimo autorių pasiūlyta algoritmo modifikacija yra tikslesnė ir duoda geresnius rezultatus nei tradicinis atkaitinimo modeliavimo algoritmas [1].

Dar vieną euristinio pobūdžio grafų spalvinimo uždavinio sprendimo būdą savo darbe pristatė D. C. Porumbel1, J. Hao1 ir P. Kuntz (2009). Grafo spalvinimo problemai spręsti buvo pasirinktas klasikinis tabu paieškos algoritmas (toliau – TS). Tačiau siekdami gauti dar geresnius rezultatus, tyrėjai pabandė modifikuoti pastarąjį metodą ir pristatė euristinį pozicija vadovavimosi tabu paieškos (angl. *Position Guided Tabu Search*) algoritmą (toliau – PGTS). Šis metodas leido pagerinti klasikinio TS algoritmo rezultatus bet kuriam sugeneruotam grafui. Be to, tyrėjai pastebėjo, kad PGTS yra pakankamai efektyvus algoritmas lyginant su kitais metodais aprašytais literatūroje grafo spalvinimo problemai spręsti. Išskyrus kelis grafų atvejus, PGTS rasdavo geriausią spalvinimo variantą [26].

Grafo spalvinimo problema domėjosi ir J. O. Hajduk (2010). Kaip ir ankstesniame darbe autorius tirdamas grafo spalvinimo uždavinį, pritaikė TS algoritmą. Jis siekė nustatyti, kokią įtaką uždavinio sprendimui turi užduodami TS algoritmo parametrai. Darbe parodyta, kad įmanoma sumažinti TS parametrų skaičių nepakenkiant algoritmo veikimo našumui. Visgi pažymėtina, kad paliekami parametrai turi būti atrenkami atsargiai. Tyrėjas pasiūlė tabu paieškos mechanizmą, kuris gali būti naudojamas nustatyti, ar algoritmas yra įstrigęs paieškos srityje, kas gali būti nulemta netinkamo parametrų verčių pasirinkimo [12].

1991 m. L. Davis paskelbė savo darbą apie genetinio algoritmo taikymą grafo spalvinimo problemai spręsti [7]. Genetinis algoritmas – tai optimizavimo procedūros, kurios atkartoja tam tikros populiacijos ir sprendimų kandidatų evoliuciją, artėjant prie geriausio užduoties sprendimo. Tai reiškia, kad šios procedūros turi kelias priemones rekombinuoti sprendimus į naujus sprendimus ir pasirinkti dalį populiacijos, kuri laikoma blogesne už likusią populiacijos dalį, ir ją atmesti. Daviso siūlomas algoritmas kodavo sprendimus, kaip viršūnių perstatymus. Nors pasiūlyto metodo rezultatai ir buvo geresni už godžiųjų algoritmų, tačiau lyginant su kitais literatūroje minimais euristiniais algoritmais šis pasirodė esąs mažiau našus.

Genetinio algoritmo pagalba grafo spalvinimo problemą sprendė ir S. M. Douiri bei S. Elbernoussi (2014). Sujungę genetinį algoritmą su lokalios paieškos euristika (angl. *local search heuristic*), tyrėjai darbe pristatė genetinio algoritmo hibridą. Eksperimentiniai skaičiavimai, kuriems atlikti autoriai pasirinko 68 grafus, iš kurių 12 – turėjo po 900, 1000, 2000 ir 4000 viršūnių, parodė, kad jų pasiūlytu būdu gauti rezultatai buvo labai konkurencingi, lyginant su to meto labiausiai

žinomais rezultatais, paskelbtais literatūroje [9].

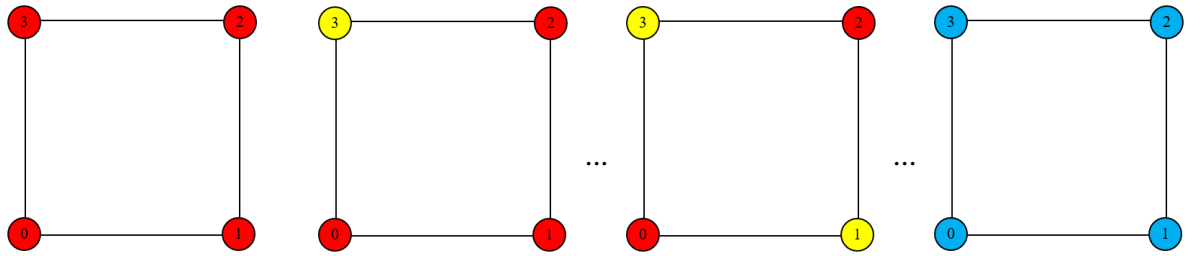
Apibendrinant prieš tai aptartus tyrimus, matyti, kad grafų spalvinimo problema yra aktuali ir plėtojama iki šių dienų. Remiantis minėtų ir kitų tyrėjų patirtimi, šiame darbe bus bandoma praplėsti nagrinėjamą temą pritaikant euristinius grafo spalvinimo metodus.

3.2. Grafų spalvinimo algoritmai

Atsižvelgiant į prieš tai minėtus darbus, galima teigti, kad egzistuoja nemažai algoritmų, skirtų viršūnių spalvinimo problemai spręsti. Šiame skyrelyje aprašomi kelių grafo spalvinimo algoritmų atlikimo etapai, kurie yra siūlomi literatūroje tvarkaraščių sudarymo problemai spręsti.

3.2.1. Pilno perrinkimo algoritmas

Pilnas perrinkimas (arba grubios jėgos metodas, angl. *brute force*) – matematinių uždavinių sprendimo būdas, kuris priklauso sprendimo paieškos metodų klasei, kai išbandomi visi galimi sprendiniai. Pilno perrinkimo sudėtingumas priklauso nuo visų galimų uždavinio sprendinių skaičiaus. Nors tik pilnas perrinkimas gali pateikti visada teisingą ir optimalų uždavinio sprendimą, tačiau, jeigu sprendinių erdvė yra labai didelė, tai šis metodas praranda prasmę, nes sprendimo ieškojimas gali užtrukti kelias valandas, dienas, metus ar net šimtmečius. Dėl šios priežasties šis algoritmas taikomas tik nedideliems grafams.



(a) (b) (c) (d)

5 pav. Pilno perrinkimo pavyzdys. Grafas turi 4 viršūnes, todėl galima sudaryti $4^4 = 256$ spalvų kombinacijų, t. y.: 0000, 0001, ..., 0101, ..., 3333[7]. Pasirenkamas pirmas spalvų derinys ir nuspalvinamos grafo viršūnės (*a* iliustracija). Kadangi bent viena iš kaimyninių viršūnių turi tą pačią spalvą, pirmasis spalvų derinys nėra sprendinys. Su antrąja kombinacija gaunama analogiška situacija (*b* iliustracija). Algoritmas užbaigiamas patikrinus visus įmanomus spalvų variantus (*d* iliustracijoje patikrintas paskutinis derinys). Sprendinys yra pirmas tinkamas variantas su mažiausiu spalvų skaičiumi (*c* iliustracija). Šiuo atveju chromatinis skaičius $\chi(G) = 2$.

Pilno perrinkimo algoritmas grafo viršūnių spalvinimo problemai spręsti atrodo taip:

1 žingsnis. Sudaromas visų įmanomų spalvų kombinacijų sąrašas. Jei grafas turi n viršūnių, tai iš viso galima sudaryti n^n spalvų kombinacijų.

2 žingsnis. Iteruojama per sudarytą spalvų kombinacijų sąrašą ir tikrinama, ar įmanoma su pasirinktuoju spalvų deriniu nuspalvinti grafą taip, kad bet kokios dvi viršūnės sujungtos tarpusavyje briauna turėtų skirtingą spalvą. Suradus pirmąją tokią spalvų kombinaciją, pastaroji išsaugoma. Paieška tęsiama toliau.

[7]Paprastumo dėlei spalvos žymimos skaičiais. Šiuo atveju: 0 – raudona, 1 – geltona, 2 – žalia, 3 – mėlyna.

3 žingsnis. Jei surandamas sekantis reikalavimus tenkinantis spalvų derinys, lyginamas pastarojo spalvų skaičius su paskutinio išsaugoto derinio spalvų skaičiumi (priešingu atveju pereiti prie 5 žingsnio). Jei pirmasis yra mažesnis, tai senoji išsaugota spalvų kombinacija pakeičiama naujai surastąja.

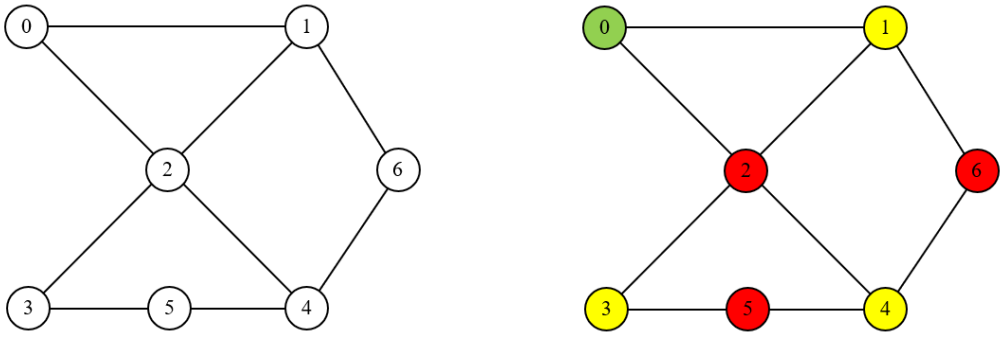
4 žingsnis. 3 žingsnis kartojamas tol, kol randama sekanti reikalavimus tenkinanti spalvų kombinacija.

5 žingsnis. Paskutinio išsaugoto derinio spalvų skaičius ir bus sprendimas, t. y. chromatinis skaičius.

3.2.2. Godusis algoritmas

Godusis algoritmas – tai eurisinių algoritmų klasei priskiriamas metodas, kuris kiekviename žingsnyje pasirenka tuo momentu geriausią variantą. Šis algoritmas paprastai yra efektyvus, tačiau kiekviename žingsnyje renkantis lokalųjį optimumą, nebūtinai gaunamas globalusis optimumas. Atsižvelgiant į godžiojo algoritmo savybes, pastarasis tinkamas grafo viršūnių spalvinimo problemai spręsti. Be to, šio metodo pateikiami rezultatai yra pakankamai konkurencingi lyginant su kitais žinomais algoritmais [19].

Literatūroje minimos kelios godžiojo metodo modifikacijos, kurios skirtos grafo dažymo problemai nagrinėti. Viena iš jų – išrikiavimo pagal didžiausią laipsnį (angl. *Largest Degree Ordering*) – pristatoma šioje darbo dalyje.



6 pav. Godžiojo algoritmo pritaikymo pavyzdys. Originalus grafas (iliustracija kairėje) ir tas pats grafas nuspalvintas godžiojo algoritmo pagalba (iliustracija dešinėje). Šiuo atveju chromatinis skaičius $\chi(G) = 3$.

Tarkime, kad turime grafą G, kurio viršūnių aibė aprašoma kaip $V = \{v_1, v_2, \ldots, v_n\}$, o viršūnių spalvų aibė – $C = \{c_1, c_2, \ldots, c_k\}$. Tuomet išrikiavimo pagal didžiausią laipsnį algoritmas vykdomas taip [2]:

1 žingsnis. Sukuriama tuščia spalvų aibė ir apskaičiuojamas kiekvienos grafo viršūnės laipsnis $\deg(v_j)$, čia $j = 1, 2, \ldots, n$. Visi apskaičiuoti viršūnių laipsniai pridedami prie grafo viršūnių laispnių sąrašo.

2 žingsnis. Iš grafo viršūnių laipsnių sąrašo pasirenkama didžiausią laipsnį turinti nenuspalvinta viršūnė (jei laipsniai sutampa, viršūnės renkamos jos numerio didėjimo tvarka). Pasirinktą viršūnę bandoma nuspalvinti viena iš spalvų iš spalvų aibės. Jei pastaroji tuščia arba joje nėra tinkamos spalvos, t. y. visos spalvos naudojamos kaimyninių viršūnių, apibrėžiama nauja spalva, kuri pridedama į spalvų aibę ir priskiriama pasirinktai viršūnei.

3 žingsnis. 2 žingsnis kartojamas tol, kol egzistuoja bent viena nenuspalvinta viršūnė. Priešingu atveju algoritmas nutraukiamas – chromatinis skaičius lygus spalvų aibėje esančių spalvų skaičiui.

1 lentelė. Grafo spalvinimo rezultatai gauti taikant godųjį (išrikiavimo pagal didžiausią laipsnį) algoritmą.

Viršūnės numeris	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Viršūnės laipsnių skaičius	2	3	4	2	3	2	2
Viršūnių pasirenkimo tvarka	4	2	1	5	3	6	7
Viršūnės spalvos	r_3	r_2	r_1	r_2	r_2	r_1	r_1

Šaltinis: sudaryta autoriaus.

Pastaba: r_1, r_2, r_3 žymi atitinkamai raudoną, geltoną, žalią spalvas 6 paveiksle.

3.2.3. Atkaitinimo modeliavimo algoritmas

Atkaitinimo modeliavimo algoritmas yra dar vienas euristinis metodas, kurio esmė paremta statistine fizika, o būtent atkaitinimu. Atkaitinimas apibrėžiamas kaip procesas, kurio metu homogeniškas metalinis kūnas išlydomas ir po to temperatūra lėtai mažinama, išlaikant ją artimą minimaliai. Metalinio kūno energija priklauso nuo atomų išsidėstymo jame. Vėstant kūnui, jis linkęs mažinti energijos kiekį, tačiau egzistuoja tikimybė pereiti į būseną su didesne energija nei esama. Kai pasiekiama šiluminė pusiausvyra, kūno būsena daugiau nebegali keistis [3]. Šiuo faktu ir grindžiamas atkaitinimo modeliavimo algoritmas.

Visgi egzistuoja problema, kad judant energijos mažinimo kryptimi yra galimybė įstrigti lokaliame minimume ir likti nepakankamai stabilioje būsenoje. Siekiant to išvengti, naudojamas algoritmas, kuriame atsižvelgiama į galimybę pereiti į būseną su didesne energijos verte.

Energijai yra gana sunku sukurti akivaizdų ekvivalentą optimizuojamoje sistemoje, tačiau tinkamai pasirinkus šį lygiavertį parametrą, galima gauti patikimus euristinius sprendimus, susijusius su optimizavimo uždaviniais, nes daugelis iš jų gali būti suvesti prie bet kokios minimizuojamos arba maksimizuojamos reikšmės [3].

Algoritmas atliekamas keliais etapais. Kiekvienas žingsnis susideda iš dabartinės būsenos atsitiktinio pakeitimo ir būsenos, esančios dabartinės kaimynystėje, sukūrimo. Kai tik gaunamas naujas sprendimas, generuojamas tikslo funkcijos atsakymas, kuris lemia, ar galima dabartinę būseną pakeisti ką tik sugeneruota. Jei tikslo funkcijos pokytis yra neigiamas, dabartinė būsena iš karto pakeičiama naująja, priešingu atveju pastaroji priimama, kai ji tenkina Metropolio[8] kriterijų. Jei šis kriterijus netenkinamas, dabartinė būklė nesikeičia. Norėdami pritaikyti algoritmą sprendžiamai užduočiai, reikia atlikti keturis esminius veiksmus:

- įvardinti galimus sprendimus;
- apsibrėžti tikslo funkciją;
- apsibrėžti kaimyninių būsenų sukūrimo mechanizmą;
- sukurti „vėsinimo“ mechanizmą.

[8]Metropolio kriterijus nurodo, kad jei skirtumas tarp tikslo funkcijų dabartinės ir naujosios būsenos verčių yra didesnis arba lygus nuliui, tuomet generuojamas tolygiai pasiskirstęs atsitiktinis skaičius δ, ir jeigu teisinga nelygybė $\delta \leqslant e^{-\frac{E_2-E_1}{T}}$ (čia E_2 yra nauja energija, o E_1 – sena energija), tai sukurta būsena yra laikoma dabartine.

Tam, kad „vėsinimas“ vyktų sklandžiai, būtina nustatyti keturis parametrus: pradinę temperatūrą, taisyklę, pagal kurią temperatūra keičiasi, iteracijų, kurios turi būti atliktos kiekviename žingsnyje, skaičių ir sustabdymo paieškos kriterijų.

Egzistuoja keletas būdų, kaip organizuoti „vėsinimo“ mechanizmą, dažnai naudojama geometrinė vėsinimo taisyklė. Remiantis šia taisykle, temperatūra apskaičiuojama pagal formulę $T_{i+1} = cT_i$, čia c – konstanta, kuri paprastai yra šiek tiek mažesnė už vienetą. Taip pat temperatūros funkcija gali būti užrašyta tokiu būdu $T(n) = \frac{T_{max}}{n}$, čia n – jau atliktų žingsnių skaičius.

Toliau pristatomi keturių spalvų uždavinio sprendimo rezultatai pritaikius atkaitinimo modeliavimo algoritmą. Tarkime, turime planarųjį grafą G (apie grafų planarumą žr. 2.3 skyrelį). Atsižvelgiant į keturių spalvų teoremą[9] (angl. *Four Color Theorem*), uždavinio sprendimo pradžioje kiekviena grafo G viršūnė atsitiktinai nuspalvinama į vieną iš keturių turimų spalvų. Būtina perspalvinti grafo viršūnes tomis pačiomis keturiomis spalvomis tokiu būdu, kad bet kokios dvi viršūnės sujungtos briauna nebūtų tos pačios spalvos. Tegul šiame uždavinyje energijos ekvivalentą atitinka briaunų skaičius, kurios jungia tos pačios spalvos viršūnes. Būsenų aibę vadinsime visų galimų viršūnių spalvinimo keturiomis spalvomis variantų sąrašą. Kaimyninėmis vadinsime dvi būsenas, kurios skiriasi ne daugiau kaip viena viršūnės spalva. Nustatomos maksimali ir minimali temperatūros, tegul $T_{max} = 20$, $T_{min} = 10^{-3}$. Kadangi nuspręsta apskaičiuoti esamą temperatūrą pagal formulę $T(n) = \frac{T_{max}}{n}$, minimali temperatūra turi būti griežtai didesnė už nulį. Prieš pradedant algoritmą žingsnio numeris yra vienetas. Toliau algoritmas vykdomas taip:

1 žingsnis. Atsitiktinai sugeneruojama būsena-kandidatė.

2 žingsnis. Atsitiktinai pasirenkama viena iš konfliktuojančių viršūnių ir nudažoma nauja spalva – taip gaunama kaimyninė būsena.

3 žingsnis. Skaičiuojamas energijų skirtumas.

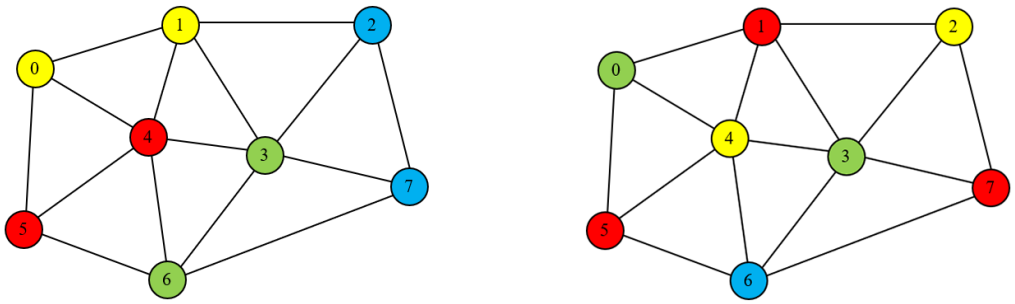
4 žingsnis. Jei skirtumas yra mažesnis nei nulis arba tenkinamas Metropolio kriterijus, pereinama į pasirinktą būseną.

5 žingsnis. Inkrementuojamas žingsnio numeris.

6 žingsnis. Apskaičiuojama esama temperatūra.

7 žingsnis. 2–6 žingsniai kartojami tol, kol dabartinė temperatūra yra aukštesnė nei minimali.

Pradines sąlygas ir galutinį rezultatą galima pamatyti 7 pav.



7 pav. Keturių spalvų uždavinys. Pradinė uždavinio būsena (iliustracija kairėje) ir galutinė būsena (iliustracija dešinėje).

[9] Matematinė teorema, teigianti, kad norint bet kokį planarųjį grafą nuspalvinti taip, kad bet kurios dvi grafo viršūnės sujungtos briauna nebūtų vienodos spalvos, užtenka keturių skirtingų spalvų. Teorema įrodyta 1976 m. panaudojant kompiuterius [30].

3.2.4. Genetinis algoritmas

Genetinis algoritmas yra euristinis paieškos algoritmas, naudojamas sprendžiant optimizavimo ir modeliavimo uždavinius atsitiktinai atrenkant, derinant ir keičiant ieškomus parametrus, naudojant mechanizmus analogiškus natūraliai atrankai gamtoje. Šie metodai yra paremti evoliuciniu gamtos modeliu, kai didėjant kartų skaičiui individai tampa vis tobulesni ir labiau prisitaikę prie aplinkos sąlygų. Be to, jie turi bendrą struktūrą ir darbo eigą, tačiau pagal konkrečią problemą jie skiriasi specifinėmis detalėmis. Algoritmas susideda iš tėvų atrankos (angl. *parent selection*), kryžminio (angl. *crossover*) ir mutacijos (angl. *mutation*) metodų [13]. Bendrai genetinis algoritmas atliekamas pagal sekančius žingsnius:

1 žingsnis. Atsitiktinai sukuriama populiacija.

2 žingsnis. Įvertinamas kiekvienos chromosomos tinkamumas ir pasirenkamos dvi geriausios.

3 žingsnis. Pritaikomas kryžminis (rekombinacijos) metodas naujoms (vaikinėms) chromosomoms gauti.

4 žingsnis. Kiekviena vaikinė chromosoma apdorojama mutacijos operatoriais.

5 žingsnis. Įvertinamas kiekvienos chromosomos tinkamumas: jei pasiekiamas optimalus sprendinys[10], algoritmas stabdomas, priešingu atveju atnaujinama populiacija ir kartojama viskas nuo 2 žingsnio, kol gaunamas optimalus sprendinys.

Toliau pateikiamas siūlomo genetinio algoritmo formalizavimas grafų spalvinimo uždaviniui spręsti [21]:

Genų seka. Genų seka $(g_0, g_1, \ldots, g_{n-1})$ reprezentuoja grafo viršūnių $V(G)$ spalvas, čia $g_i (0 \leqslant i \leqslant n-1)$ teigiamas sveikas skaičius.

Populiacijos sukūrimas. Rekombinacijos ir mutacijos tikimybės yra p_c ir p_m atitinkamai. g žymi kartos numerį, o P_g – atsitiktinį g kartos genų sekų sąrašą, t. y. populiaciją. Tuomet $N = |P_g|$ yra populiacijos dydis (g kartos chromosomų skaičius). Populiacijai P_0 $(g = 0)$ sukuriama N genų sekų.

Tikslo funkcija. Tarkime $f(G) > 0$ yra bendra tinkamumo (angl. *fitness*) funkcija grafui G, o taip pat skirtingų sveikųjų skaičių (šiuo atveju skirtingų spalvų) chromosomoje skaičius. Tuomet grafo spalvinimo uždavinio tikslas minimizuoti funkciją $f(G)$.

Konfliktiniai genai/briaunos. Chromosomos $s = (s_0, s_1, \ldots, s_i, \ldots, s_j, \ldots, s_{n-1})$ genai s_i ir s_j yra konfliktiniai tada ir tik tada, kai $s_i = s_j$ ir $i, j \in E(G)$, t. y. kaimyninės viršūnės turi tą pačią spalvą.

Tinkamumo įvertinimas ir pasirinkimas. Ruletės metodas (angl. *fitness proportionate selection* arba *roulette wheel selection*) naudojamas tam tikrų genų sekų pasirinkimui kiekvienoje kartoje g ir atliekamas taip:

- apskaičiuojama kiekvienos genų sekos i $(1 \leqslant i \leqslant N)$ tinkamumo funkcija $F(i)$ populiacijoje P_g kaip konfliktuojančių briaunų skaičius esantis toje sekoje;
- apskaičiuojama kiekvienos i chromosomos tikimybė: $p(i) = F(i) / \sum_{i=1}^{N} F(i)$;
- kiekvienai i chromosomai apskaičiuojamas: $E(i) = N \times p(i)$;
- pasirenkamos dvi geriausios genų sekos, kurių $E(i)$ reikšmės mažiausios.

[10] Kadangi genetinis algoritmas yra euristinis metodas, tai kalbant apie optimalų sprendimą turima omenyje, kad jis artimas optimaliam.

Kryžminis (rekombinacijos) metodas. Rekombinacijos metu yra generuojamos naujos genų sekos. Atsitiktinai pasirenkama tikimybė p_{cr}. Jei pastaroji yra didesnė nei rekombinacijos tikimybė p_c, tai pasirinktoms dviem geriausioms genų sekoms pritaikomas kryžminis metodas, kuriuo metu gaunamos naujos chromosomos (priešingu atveju pereinama prie mutacijos metodo).

Mutacijos metodas. Rekombinacijos metu gautos vaikinės chromosomos su atsitiktinai sugeneruota tikimybe p_{mr} paveikiamos mutacijos operatoriumi, jei pastaroji tikimybė didesnė nei mutacijos tikimybė p_m (priešingu atveju pereinama prie užbaigimo ir populiacijos atnaujinimo etapo).

Užbaigimas ir populiacijos atnaujinimas. Apskaičiuojama kiekvienos naujai gautos genų sekos tinkamumo funkcija $F(i^{'})$. Jei bent viena reikšmė lygi 0 (nėra konfliktinių genų), pereinama prie paskutinio etapo. Priešingu atveju padidinamas kartos numeris $g = g + 1$, populiacija P_g atnaujinama pakeičiant senas chromosomas naujai gautomis vaikinėmis chromosomomis – ir pereinama prie tinkamumo įvertinimo ir pasirinkimo etapo.

Optimalus sprendimas. Gautas optimalus sprendimas. Generacija stabdoma.

3.2.5. Tabu paieškos algoritmas

Tabu paieškos algoritmo pradininku laikomas F. Gloveras [11], kuris iš esmės pasiūlė naują lokalios paieškos schemą. Pagal ją algoritmas neapsistoja lokalaus optimumo taške, kaip tai nurodyta standartiniame lokalaus nusileidimo algoritme, bet leidžiama keliauti iš vieno lokalaus optimumo į kitą, tikintis tarp jų rasti globalų optimumą. Tabu paieškos atveju toleruojami sprendiniai, galintys pabloginti rezultatą, jeigu nėra tokių, kurie pagerintų jį (pavyzdžiui, kai paieška įstrigo griežtai lokaliame minimume). Pagrindinis mechanizmas, kuriuo dėka algoritmas gali išeiti iš lokalaus optimumo, yra draudimų sąrašas (toliau – tabu), neledžiantis paieškai grįžti prie anksčiau aplankytų sprendinių. Tabu paieškos vykdymo metu naudojamos atminties struktūros, kurios apibūdina aplankytus sprendinius arba tam tikrų taisyklių rinkinius. Jei galimas sprendimas anksčiau buvo aplankytas arba pažeidė taisyklę, jis pažymimas kaip „tabu“ (draudžiamas), kad į pastarąjį algoritmas pakartotinai neatsižvelgtų. Tabu paieškos algoritmo modifikacija skirta grafų spalvinimo problemai spręsti vadinama Tabucol.

Tabucol algoritmas pradedamas atsitiktinai generuojant k-spalvų derinį, kuriuo nuspalvinamas užduotas grafas G. Tai (beveik neabejotinai) nebus optimalus grafo nuspalvinimas. Tikslo funkcija f_c, kurią Tobucol metodu stengiamasi minimizuoti, yra lygi konfliktų skaičiui konfigūracijoje (spalvų derinyje) $f_c = \sum_{n=1}^{k} |\forall(i,j) \in E : c(i) = c(j)|$. Jei rastas toks spalvų derinys C, kad $f_c(C) = 0$, tai jis yra optimalus. Tabucol paieškos erdvė Ω, kurią sudaro visi įmanomi grafo G spalvinimo variantai: $|V|^k$. Konfigūracijos $C \in \Omega$ kaimynai C' gaunami keičiant vienos konfliktuojančios viršūnės spalvą.

Sugeneravus pradinį spalvų rinkinį Tabucol algoritmas vykdomas tol, kol surandamas tinkamas sprendinys arba pasiekiamas nustatytas laiko ar iteracijų limitas, o taip pat grąžinamas mažiausias surastas konfliktų skaičius. Kiekviena algoritmo iteracija susideda iš sekančio pakeitimo parinkimo (atitinka naujo spalvų derinio priskyrimą), jo taikymo ir tabu sąrašo sudarymo.

Sekančio pakeitimo pasirinkimas atliekamas taip: iš visų galimų pakeitimų išrenkami tie, kurių pagerinimas f_c yra didžiausias. Tuomet atsitiktine tvarka pasirenkamas vienas iš jų. Pasirinkus pakeitimą, pastarasis yra pritaikomas, kitaip tariant, pereinama prie kaimyninės konfigūracijos. Be to, jis neturėtų būti pritaikomas tam tikram iteracijų skaičiui. Pastarasis apspendžiamas tabu dydžiu T_l. T_l priklauso nuo tikslo funkcijos ir iteracijų, kai tikslo funkcija liko nepakitusi, skaičiaus m, t. y. $T_l = \alpha \times f_c(C) + random[0, A] + \lfloor \frac{m}{m_{max}} \rfloor$, čia $\alpha \in [0, 1]$ ir $\lfloor \frac{m}{m_{max}} \rfloor$ – komponentė, kuri padeda algoritmui pereiti į naują paieškos erdvės dalį, kai jis patenka į amžino ciklo būseną. Kaip greitai

tai įvyksta, priklauso nuo m_{max} dydžio [12]. Žemiau pateikiamas Tabucol algoritmo pseudokodas.

1 algoritmas. Tabu paieška (G, k)

Atsitiktinai generuojamas spalvų derinys C
while *konfliktųSkaičius* $\neq 0$ **and not** pasiektas nustatytas laiko ar iteracijų limitas **do**
  {Randamas ne tabu kaimynas C', kurio tikslo funkcijos pagerinimas yra didžiausias. C' yra tabu tada ir tik tada, kai pora (i, i'), atitinkanti pakeitimą $C \overset{C(i):=i'}{\longrightarrow} C'$, yra pažymėta tabu.}
  $C' \leftarrow$ *rastiDidžiausiąPagerinimą(N(C))*
  {Atliekamas pakeitimas}
  $C \leftarrow C'$
  if f_c nepasikeitė **then**
    inkrementuoti(m)
  else
    $m \leftarrow 0$
  end if
  $T_l \leftarrow \alpha \times f_c(C) + random[0, A] + m/m_{max}$
  Porą (i, i') pažymėti tabu T_l iteracijoms.
  if $f_c(C) < f_c(C_{best})$ nepasikeitė **then**
    $C_{best} \leftarrow C$
  end if
end while
return $f_c(C_{best})$

3.3. Praktinis pritaikomumas

Apart įprastų tvarkaraščių sudarymo grafų spalvinimo taikymo spektras praktinių problemų sprendimui yra labai platus. Kadangi daugeliu atvejų sudėtingas uždavinys negali būti išspręstas efektyviai (sprendimas pareikalautų labai didelių laiko sąnaudų), realizuojami algoritmai naudoja skirtingų rūšių apribojimus ar aproksimacijas, kad galėtų atlikti skaičiavimus per pakankamai trumpą laiką. Toliau aptariami keli potencialūs taikymo atvejai.

Dažnių paskirstymas radijo stotims. Tarkime, kad norime paskirstyti transliavimo dažnius tarp vietinių radijo stočių. Pagal grafo spalvinimo uždavinį kiekviena stotis atitinka grafo viršūnę, o dažniai – spalvas. Jei dvi stotys yra labai arti viena nuo kitos, jos sujungiamos briauna. Tokiu būdu šios stotys negali naudoti tą patį dažnį (turėti vienodą spalvą), nes priešingu atveju būtų susiduriama su dažnių trukdžiais. Praktikoje tokį grafą nebūtų pernelyg sunku nuspalvinti, kol turime pakankamai didelį dažnių kiekį, palyginti su stočių, esančių greta viena kitos, skaičiumi.

Sudoku galvosūkio sprendimas. Taip pat galime modeliuoti galvosūkį Sudoku, nustatant po vieną viršūnę kiekvienam langeliui. Populiariausios yra 9×9 langelių lentelės su 3×3 boksais. Spalvos yra devyni skaičiai, o kai kurie iš jų yra iš anksto priskirti tam tikroms viršūnėms. Dvi viršūnės yra sujungtos briauna, jei jų langeliai yra tame pačiame bokse, eilutėje arba stulpelyje. Galvosūkis yra išsprendžiamas, jeigu galime nuspalvinti šį grafą devyniomis spalvomis, atsižvelgiant į iš anksto priskirtas spalvas.

Žemėlapių spalvinimas. Tikriausiai viena iš seniausių spalvinimo problemų yra skirtingų šalių spalvojimas žemėlapyje. Šiuo atveju originalus žemėlapis yra grafas, kuriame regionai yra šalys. Norėdami sumodeliuoti spalvinimo problemą, turime tai konvertuoti į dvigubą grafą, kuriame

kiekvienas regionas yra viršūnė, o du regionai sujungiami briauna, kai šalys turi bendrą sieną. Šiuo atveju turime plokštųjį grafą, o tai žymiai palengvina spalvinimą.

Lėktuvų skrydžių tvarkaraštis. Tarkime, kad turime k orlaivius, kuriuos reikia priskirti n skrydžiams, o i-tojo skrydžio vykdomo laiko intervalas yra (a_i, b_i). Akivaizdu, kad jei du skrydžiai sutampa, tada negalime priskirti to paties orlaivio abiem skrydžiams. Konfliktinio grafo viršūnės atitinka skrydžius ir jos sujungiamos briauna tuo atveju, jei atitinkami laiko intervalai sutampa. Taigi toks konfliktinis grafas yra intervalinis grafas, kuris optimaliai gali būti nuspalvotas polinomialiu laiku.

Registrų paskirstymas. Spalvinimo uždavinio taikymas ypač svarbus kompiuterinėse technologijose, o būtent registrų paskirstyme. Kompiliatorius – kompiuterinė programa, kuri verčia vieną kompiuterinę kalbą į kitą. Norint pagerinti gaunamo kodo vykdymo laiką, vienas iš kompiliavimo optimizavimo būdų yra registrų paskirstymas, kuriame dažniausiai naudojami kompiliuojamos programos kintamieji saugomi procesoriaus greito veikimo registruose. Idealiu atveju kintamieji saugomi registruose taip, kad jie visi yra registruose jų naudojimo metu. Standartinis požiūris į šią problemą yra jos suvedimas į grafo spalvinimo uždavinį. Kompiliatorius kuria interferencinį grafą, kuriame viršūnės atitinka kintamuosius, o spalvos – registrus. Du kintamieji yra sujungiami briauna, jei jie yra naudojami tuo pačiu metu ir dėl to negali dalintis registru.

4. Kompiuteriniai skaitiniai eksperimentai

Ketvirtame skyriuje paminėta programinė įranga ir programavimo kalba, kurios pagalba buvo realizuoti grafo spalvinimo algoritmai, kuriais sprendžiamas tvarkaraščio sudarymo uždavinys. Taip pat šioje dalyje aptariamas sintetinių grafų generavimas. Paskutiniame poskyryje atliekama išsami algoritmų efektyvumo analizė.

4.1. Eksperimentų vykdymo eiga

Šis skyrelis skirtas aprašyti kompiuterinių eksperimentų, atliktų tyrimo metu, eigą.

Programinė įranga

JetBrains PyCharm Community Edition 2018 programinės įrangos ir *Python* programavimo kalbos pagalba buvo realizuoti šie grafo spalvinimo algoritmai: pilno perrinkimo, godusis, atkaitinimo modeliavimo, genetinis ir Tabu paieškos – skirti tvarkaraščių sudarymo problemai spręsti. Pastarųjų metodų analizė atlikta *Microsoft Windows 10 OS* aplinkoje. Vartotojo kompiuterio pagrindiniai parametrai:

- *Procesorius:* Inter(R) Core(TM) i5-8250U CPU @ 1.60 GHz 1.80 GHz
- *RAM kiekis:* 16 GB
- *Disko dydis:* 465 GB

Sintetinių grafų generavimas

Siekiant atlikti išsamią minėtų algoritmų analizę, pasinaudota *Python NetworkX 2.2* biblioteka, kurios pagalba buvo generuojami trijų tipų įvairaus dydžio sintetiniai grafai:

- ***pilnieji*** – įvestis: *grafo viršūnių skaičius*; išvestis: *pilnas grafas* (plačiau žr. 2.1 skyrių).
- ***reguliarieji*** – įvestis: *grafo viršūnių skaičius* ir *kiekvienos viršūnės laipsnis*; išvestis: *reguliarus grafas* (plačiau žr. 2.3 skyrių).
- ***atsitiktiniai*** – įvestis: *grafo viršūnių skaičius* ir *grafo visų briaunų skaičius*; išvestis: *atsitiktinis grafas su nurodytų viršūnių ir briaunų skaičiumi*.

Grafo duomenų struktūra

Tam, kad įgyvendinti algoritmai sėkmingai išspręstų grafų spalvinimo uždavinį, t. y. apskaičiuotų chromatinį skaičių, būtina perrašyti grafo duomenis į tekstinį failą[11], kuris bus nuskaitomas programos. Kitaip tariant, reikia pasirinkti tinkamą duomenų struktūrą. Šiame darbe naudojamas gretimumo sąrašas (angl. *Adjacency List*), kurio struktūra susideda iš eilučių su viršūnių numeriais. Pirmasis numeris eilutėje žymi pradžios viršūnę, o sekantys eilutės numeriai – viršūnes, su kuriomis jungiasi pradžios viršūnė, t. y. sudaro briaunas. Pavyzdžiui, grafo, kuris pavaizduotas 8 paveiksle, briaunas: 0-1, 0-5, 0-6, 1-5, 2-5, 3-5, 3-6, 4-5, 4-6, 5-6 – galima užrašyti sekančiu gretimumo sąrašu:

1 eilutė. 0 1 5 6

2 eilutė. 1 5

[11] Generavimo metu grafo duomenys įrašomi pateikta struktūra į tekstinį failą, kurį nuskaito atitinkamas algoritmas ir skaičiuoja chromatinį skaičių.

3 eilutė. 2 5

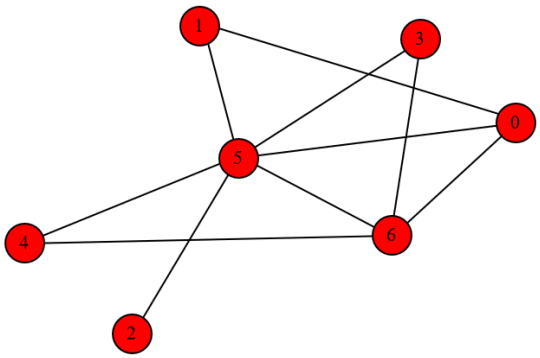
4 eilutė. 3 5 6

5 eilutė. 4 5 6

6 eilutė. 5 6

7 eilutė. 6

Tokiu pavidalu grafas užrašomas į tekstinį failą ir perduodamas nuskaityti realizuotiems algoritmams, kurie išsprendžia grafų spalvinimo uždavinį, pateikdami sprendinį, t. y. chromatinį skaičių.



8 pav. Sintetinio grafo pavyzdys. Atsitiktinis grafas su 7 viršūnėmis ir 10 briaunų.

4.2. Eksperimentų rezultatai

Kaip minėta, grafo spalvinimo algoritmai: pilno perrinkimo, godusis, atkaitinimo modeliavimo, genetinis ir Tabu paieškos – ištirti sintetiškai sugeneruotų grafų pagalba. Pastarųjų metodų efektyvumo analizė atlikta pagal du pagrindinius kriterijus: sprendimo tikslumą ir skaičiavimų greitaveiką. Taip pat pagal (1) formulę apskaičiuojamas ir nurodomas kiekvieno sugeneruoto grafo briaunų tankumas (angl. *the edge density*) D.

$$D = \frac{2 \times E}{V \times (V - 1)} \tag{1}$$

čia E yra grafo briaunų skaičius, o V – viršūnių skaičius.

Rezultatai pateikti trijų tipų lentelėse (priklausomai nuo grafų), kuriose laikas suklasifikuotas į geriausią, blogiausią ir vidutinį laiką (nurodytas sekundėmis). Į šį laiką įeina tik algoritmo vykdymas. Taip pat lentelėse nurodytas ir algoritmų sprendimas, t. y. apskaičiuotas chromatinis skaičius $\chi(G)$. Sudaryti paveikslai, iš kurių matyti, kaip algoritmų skaičiavimo trukmė priklauso nuo grafų viršūnių ir briaunų skaičiaus bei tankumo. Kiekvienas algoritmas vykdomas su skirtingo dydžio grafais po 10 kartų.

4.2.1. Pilno perrinkimo algoritmo analizė

Kaip minėta, pilno perrinkimo algoritmas nepriklauso euristinių paieškos algoritmų klasei. Be to, šis metodas tinkamas tik nedideliems grafams. Atliekant skaitinius eksperimentus, maksimalus grafo dydis, kuriam esant dar pavyko apskaičiuoti chromatinį skaičių per priimtiną laiką (skaičiavimai užtruko vidutiniškai apie 55 minutes) – 9 viršūnių pilnas grafas (žr. 2 lentelę). Dėl šios

priežasties rezultatai, gauti pilno perrinkimo algoritmu, nepalyginami su kitų metodų rezultatais, kurie apskaičiuoti su žymiai didesniais grafais.

2 lentelė. Rezultatai gauti taikant pilno perrinkimo algoritmą pilniems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
br_f_graph_5	5	10	1	0.0156	0.0313	0.0203	5
br_f_graph_7	7	21	1	5.6250	5.8594	5.7311	7
br_f_graph_8	8	28	1	130.0937	133.9531	131.2668	8
br_f_graph_9	9	36	1	3313.8644	3328.7093	3321.2070	9

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

2 lentelėje pateikti skaičiavimai atlikti pilniems grafams. Matyti, kad grafui turint tik 5 viršūnes pilno perrinkimo algoritmas chromatinį skaičių apskaičiuoja pakankamai greitai – vidutiniškai per 0.0203 sekundės. Tačiau padidinus grafo dydį tik keliomis viršūnėmis, skaičiavimo laikas išauga dešimtimis, šimtais ar tūkstančiais kartų. Žinant pilnojo grafo apibrėžimą (plačiau žr. 2.1 skyrių), nesunku suvokti, kad chromatinis skaičius yra lygus grafo viršūnių skaičiui – tai įrodo ne tik pilnojo perrinkimo metodas, bet ir sekantys euristiniai algoritmai.

3 lentelė. Rezultatai gauti taikant pilno perrinkimo algoritmą reguliariems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
br_r_graph_5_2	5	5	0.5	0.0156	0.0469	0.0250	3
br_r_graph_7_2	7	7	0.33	6.5000	6.7188	6.6344	3
br_r_graph_7_4	7	14	0.66	6.0469	6.3750	6.1813	3
br_r_graph_8_2	8	8	0.29	150.6094	154.9532	152.7208	2
br_r_graph_8_4	8	16	0.57	138.7344	142.7500	140.6498	3
br_r_graph_8_6	8	24	0.86	130.6407	133.3125	132.5904	4

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

3 ir 4 lentelėse atitinkamai pateikti reguliarių ir atsitiktinių grafų rezultatai, kai grafo tankumas, viršūnių ir briaunų skaičius iš 3 lentelės sutampa su 4 lentelės atitinkamu grafu. Šia analize siekta nustatyti, kokią įtaką skaičiavimo laikui ir chromatiniam skaičiui turi briaunų pasiskirstymas, nepaisant to, kad grafų dydis ir tankumas vienodas. Lyginant nereguliarų briaunų pasiskirstymą su reguliariu, vienu atveju skaičiavimo trukmė sumažėdavo ir būdavo gaunamas mažesnis chromati-

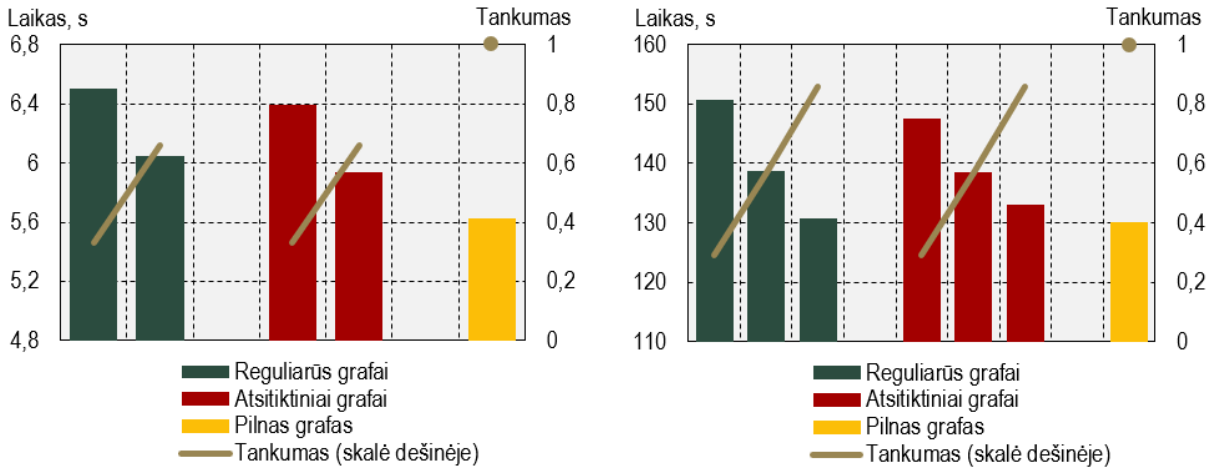
nis skaičius, pavyzdžiui: *br_f_graph_7_2* ir *br_f_graph_7_7* grafai – kitu atveju skaičiavimo trukmė padidėdavo ir būdavo gaunamas didesnis chromatinis skaičius, pavyzdžiui: *br_f_graph_8_6* ir *br_f_graph_8_24* grafai. Vadinasi, atsitiktinumo faktoriaus poveikis rezultatams nevienareikšmiškas.

4 lentelė. Rezultatai gauti taikant pilno perrinkimo algoritmą atsitiktiniams grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
br_a_graph_5_5	5	5	0.5	0.0156	0.0313	0.0222	3
br_a_graph_7_7	7	7	0.33	6.3906	6.6562	6.4797	2
br_a_graph_7_14	7	14	0.66	5.9375	6.1094	6.0188	4
br_a_graph_8_8	8	8	0.29	147.6094	150.6407	148.9779	2
br_a_graph_8_16	8	16	0.57	138.5938	143.1093	140.1732	4
br_a_graph_8_24	8	24	0.86	133.0937	134.6718	134.1920	6

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.



9 pav. Pilno perrinkimo algoritmo analizė. Iliustracija kairėje: vidutinis skaičiavimų laikas su 7 viršūnių reguliariais, atsitiktiniais ir pilnu grafais su skirtingu tankumu – 0.33, 0.66 ir 1. Iliustracija dešinėje: vidutinis skaičiavimų laikas su 8 viršūnių reguliariais, atsitiktiniais ir pilnu grafais su skirtingu tankumu – 0.29, 0.57, 0.86 ir 1.

Atlikta pilno perrinkimo algoritmo analizė parodė, kad kuo tankesnis tas pats grafas, tuo skaičiavimo laikas yra trumpesnis (žr. 9 pav.). Iš 9 paveikslo matyti, kad briaunų skaičius grafe (tankumas) atvirkščiai proporcingas skaičiavimo trukmei. Tai galima paaiškinti sekančiu būdu: kadangi pilnojo perrinkimo algoritmas tikrina visas įmanomas spalvų kombinacijas, tai esant tankesniam grafui greičiau surandamos konfliktuojančios viršūnės, todėl ir greičiau atmetamas einamasis spalvų derinys ir keliaujama prie sekančio spalvų varianto.

4.2.2. Godžiojo algoritmo analizė

Kadangi godusis algoritmas priklauso euristinių paieškos metodų klasei, tai skaitinius eksperimentus galima atlikti su žymiai didesniais grafais. Padarius kelias pradines algoritmo iteracijas, nuspręsta, kad minimalus grafo dydis, nuo kurio bus eksperimentuojama, yra 100 viršūnių, maksimalus – 5000[12].

5 lentelė. Rezultatai gauti taikant godųjį algoritmą pilniems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
f_graph_100	100	4950	1	0.0	0.0157	0.0016	100
f_graph_200	200	19900	1	0.0	0.0156	0.0047	200
f_graph_300	300	44850	1	0.0	0.0157	0.0141	300
f_graph_500	500	124750	1	0.0312	0.0469	0.0328	500
f_graph_1000	1000	499500	1	0.1250	0.1719	0.1391	1000
f_graph_2000	2000	1999000	1	0.5781	0.5782	0.5781	2000
f_graph_5000	5000	12497500	1	3.8437	3.8906	3.8656	5000

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

6 lentelė. Rezultatai gauti taikant godųjį algoritmą reguliariems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
r_g_100_17	100	850	0.17	0.0	0.0157	0.0016	9
r_g_200_90	200	9000	0.45	0.0	0.0156	0.0031	32
r_g_300_80	300	12000	0.27	0.0	0.0156	0.0031	27
r_g_500_46	500	11500	0.09	0.0	0.0156	0.0047	17
r_g_1000_400	1000	200000	0.40	0.0469	0.0625	0.0594	102
r_g_2000_1152	2000	1152000	0.58	0.3278	0.3495	0.3404	266
r_g_5000_250	5000	625000	0.05	0.2812	0.2969	0.2922	61

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

[12]Grafų dydžiai pasirinkti remiantis ankstesnių darbų autorių patirtimi. Maksimalus viršūnių skaičius pasirinktas atsižvelgiant ir į kitų euristinių algoritmų greitaveiką tam, kad galima būtų atlikti metodų palyginamumo analizę.

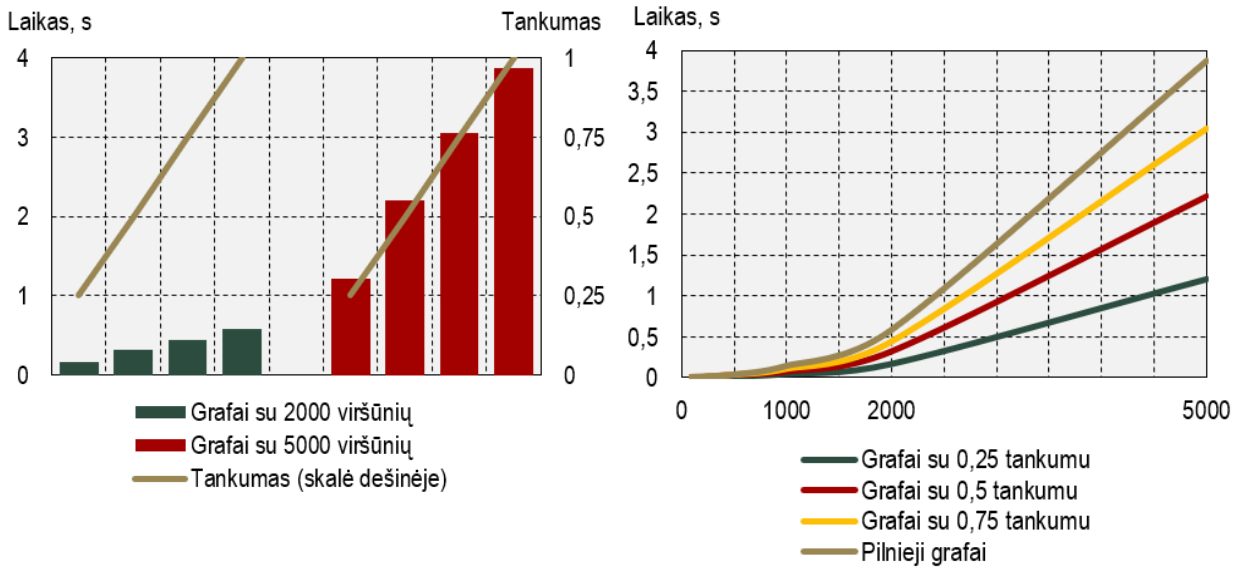
Iš 5, 6 ir 7 lentelių matyti, kad godusis algoritmas pasižymi sparčia skaičiavimų greitaveika. Turint pilnąjį grafą net su 2000 viršūnių rezultatas gaunamas mažiau nei per sekundę (vidutiniškai per 0.5781 sekundės), o su 5000 viršūnių – vidutiniškai tik per 3.8656 sekundės. Be to, šis metodas visais atvejais pateikia tikslų chromatinį skaičių pilniems grafams. Kalbant apie chromatinio skaičiaus nustatymą reguliariuose ir atsitiktiniuose grafuose, esant tam pačiam grafo tankumui, tačiau skirtingam briaunų pasiskirstymui (pirmuoju atveju tolygus, antruoju – atsitiktinis), gaunamas didesnis arba mažesnis chromatinis skaičius, pavyzdžiui: *r_g_100_17* ir *a_graph_100* grafai, *r_g_2000_1152* ir *a_graph_2000* grafai.

7 lentelė. Rezultatai gauti taikant godųjį algoritmą atsitiktiniams grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius $\chi(G)$**
a_graph_100	100	850	0.17	0.0	0.0156	0.0016	10
a_graph_200	200	9000	0.45	0.0	0.0157	0.0047	30
a_graph_300	300	12000	0.27	0.0	0.0157	0.0047	26
a_graph_500	500	11500	0.09	0.0	0.0157	0.0063	17
a_graph_1000	1000	200000	0.40	0.0469	0.0625	0.0578	97
a_graph_2000	2000	1152000	0.58	0.3281	0.3594	0.3359	259
a_graph_5000	5000	625000	0.05	0.2812	0.2969	0.2891	58

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.



10 pav. Godžiojo algoritmo analizė. Iliustracija kairėje: vidutinis skaičiavimų laikas su 2000 ir 5000 viršūnių grafais su skirtingu tankumu – 0.25, 0.5, 0.75 ir 1. Iliustracija dešinėje: vidutinio skaičiavimų laiko priklausomybė nuo grafo dydžio (viršūnių skaičiaus), kai grafo tankumas pastovus.

Iš 10 paveikslo abiejų iliustracijų matyti, kad, priešingai nei pilnojo perrinkimo algoritmo atveju, kuo didesnis grafo tankumas, tuo skaičiavimų trukmė didesnė. Be to, didinant grafo viršūnių skaičių, kai jo tankumas nesikeičia, skaičiavimų laikas taip pat ilgėja. Tai galima paaiškinti sekančiu būdu: tankesnis grafas reiškia, kad jo viršūnės turi daugiau briaunų, tuo pačiu ir kaimynų – vadinasi, vykdant godžiojo algoritmo žingsnius, pastarajam iteruojant per kiekvieną viršūnę tenka tikrinti daugiau jos kaimynų ir tik tada priskirti jai spalvą.

4.2.3. Atkaitinimo modeliavimo algoritmo analizė

Kaip minėta anksčiau, atkaitinimo modeliavimo algoritmas taip pat priklauso euristinių paieškos metodų klasei, kuris remiasi gamtos natūraliais procesais. Skaitiniams eksperimentams atlikti buvo pritaikytas eksponentinis temperatūros vėsinimas ir remiantis ankstesniais darbais pasirinkti sekantys parametrai [18]: $\alpha = 0.99$, $T_{min} = 0.1$, maksimalus leistinas algoritmo iteracijų skaičius lygus 100000 (iš pastarųjų koeficientų išskaičiuota ir T_{max}).

8 lentelė. Rezultatai gauti taikant atkaitinimo modeliavimo algoritmą pilniems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
f_graph_100	100	4950	1	0.0534	0.0691	0.0617	100
f_graph_200	200	19900	1	0.4547	0.5484	0.4969	200
f_graph_300	300	44850	1	1.6519	1.8059	1.7008	300
f_graph_500	500	124750	1	8.1019	9.2763	8.6932	500
f_graph_1000	1000	499500	1	74.30469	79.2424	76.4004	1000

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Iš 8, 9 ir 10 lentelių matyti, kad atitinkamai didžiausias nagrinėtas pilnas, reguliarus ir atsitiktinis grafas sudarytas iš 1000 viršūnių. Toks dydis pasirinktas, siekiant apskaičiuoti artimą optima-

9 lentelė. Rezultatai gauti taikant atkaitinimo modeliavimo algoritmą reguliariems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
r_g_100_17	100	850	0.17	0.2007	0.8248	0.3997	7
r_g_200_90	200	9000	0.45	7.3624	18.7736	10.19687	26
r_g_300_80	300	12000	0.27	10.2704	16.1831	12.3473	23
r_g_500_46	500	11500	0.09	7.9032	12.9417	10.5321	15
r_g_1000_400	1000	200000	0.40	612.2358	633.2235	623.7371	95

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

liam sprendinį per pakankamai trumpą (priimtiną) laiką.

Lyginant du vienodo dydžio grafus, kai viršūnių ir briaunų skaičius vienodas (iš čia seka, kad tankumas taip pat sutampa), tačiau pirmojo briaunų pasiskirstymas tolygus (reguliarus grafas), antrojo – atsitiktinis, atkaitinimo modeliavimo algoritmo vykdymo trukmė daugeliu atvejų buvo trumpesnė, nustatant chromatinį skaičių reguliariuose grafuose (žr. 9 ir 10 lenteles). Tačiau briaunų pasiskirstymas (tolygus ar atsitiktinis), esant vienodam grafų dydžiui ir tankumui, neturėjo įtakos apskaičiuojant chromatinį skaičių – abiem atvejais buvo gaunamas tas pats sprendinys.

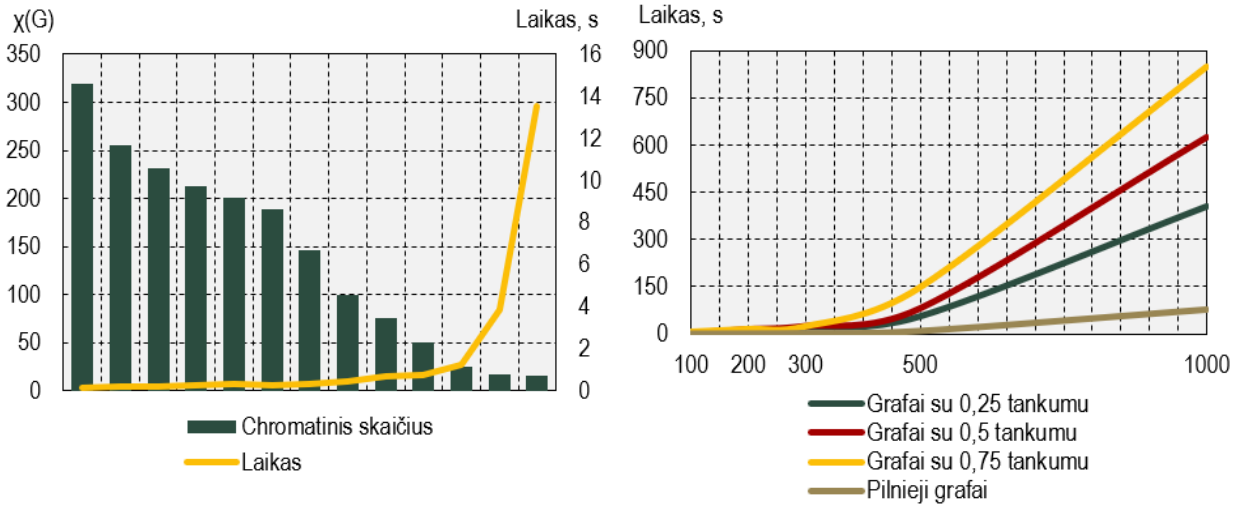
10 lentelė. Rezultatai gauti taikant atkaitinimo modeliavimo algoritmą atsitiktiniams grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius $\chi(G)$**
a_graph_100	100	850	0.17	0.3634	1.0812	0.6987	7
a_graph_200	200	9000	0.45	6.8796	18.8604	12.8196	26
a_graph_300	300	12000	0.27	9.4603	24.4075	14.5466	23
a_graph_500	500	11500	0.09	9.9533	18.2886	13.5479	15
a_graph_1000	1000	200000	0.40	527.7796	612.4545	568.0248	95

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Atliekant atkaitinimo modeliavimo algoritmo analizę pastebėta, kad kiekvienos lentelės rezultatus laiko prasme galima žymiai pagerinti, tačiau tuo pačiu labai pabloginti tikslumo prasme (išskyrus pilnų grafų atvejį, tikslumas nenukenčia). Šio metodo pradinėje atlikimo stadijoje privaloma sugeneruoti atsitiktinį spalvų derinį, todėl būtina užduoti iš kelių spalvų pastarasis sudaromas, t. y. fiktyviai nustatyti chromatinį skaičių, kuris pasibaigus algoritmui gali būti patikslintas, page-



11 pav. Atkaitinimo modeliavimo algoritmo analizė. Iliustracija kairėje: vidutinio skaičiavimų laiko ir chromatinio skaičiaus priklausomybė (skaičiavimai atlikti su 500 viršūnių grafu, kai tankumas lygus 0.09). Iliustracija dešinėje: vidutinio skaičiavimų laiko priklausomybė nuo grafo dydžio (viršūnių skaičiaus), kai grafo tankumas pastovus.

rintas arba likti nepakitęs. Parinkus pakankamai didelį (šiek tiek mažesnį (negalioja pilniems grafams), lygų arba didesnį už grafo viršūnių skaičių) – algoritmas pateiks sprendinį per trumpą laiką, tačiau tolimą nuo optimalaus. Aprašytai situacijai pademonstruoti buvo pasirinktas atsitiktinis grafas *a_graph_500* iš 10 lentelės. Sudarant pradinį spalvų derinį iš spalvų, kurių yra tiek pat, kiek ir viršūnių, t. y. 500, chromatinis skaičius apskaičiuotas per 0.1406 sekundės ir lygus 319; iš 319 spalvų – per 0.1874 sekundės ir lygus 256; iš 256 spalvų – per 0.2029 sekundės ir lygus 232; ir t. t. Eksperimentas buvo tęsiamas tol, kol gautas mažiausias chromatinis skaičius, kurį dar pavyko išskaičiuoti algoritmo pagalba. Minėto eksperimento rezultatai atvaizduoti 11 paveikslo iliustracijoje kairėje. Kaip matyti, skaičiuojant tikslesnį (artimą optimaliam) chromatinį skaičių, algoritmo vykdymo trukmė ilgėja. Pastebėtina, kad tokia išvada negalioja pilnam grafui, t. y. generuojant pradinį spalvų rinkinį iš spalvų, kurių yra daugiau nei viršūnių grafe, visada apskaičiuojamas optimalus chromatinis skaičius ir per žymiai trumpesnį laiką. 9 ir 10 lentelėse pateikti geriausi rezultatai tikslumo prasme – apskaičiuoti mažiausi įmanomi chromatiniai skaičiai sugeneruotiems grafams. 8 lentelėje rezultatai pateikti, generuojant pradinį spalvų rinkinį iš spalvų skaičiaus lygaus grafo viršūnių skaičiui.

Kalbant apie grafo dydžio (viršūnių skaičiaus), kai pastarojo tankumas išlieka pastovus, ir algoritmo vykdymo trukmės priklausomybę, tai matomas tiesioginis ryšys (žr. 11 paveikslo iliustraciją dešinėje). Be to, galima pastebėti, kad kuo tankesnis grafas, tuo skaičiavimų laikas taip pat ilgesnis. Tačiau, kai tankumas lygus 1, t. y. turime pilną grafą, algoritmo vykdymo trukmė mažesnė nei kitais atvejais. Tai galima paaiškinti tokiu būdu: visų pirma, užduodamas didesnis fiktyvus chromatinis skaičius (parametras, kuris nusako iš kelių spalvų generuojamas pradinis atsitiktinis spalvų derinys), o kaip minėta aukščiau tai sąlygoja spartesnį skaičiavimą. Antra, turint pilną grafą greičiau surandamos konfliktuojančios viršūnės, todėl dalis patikrinimų yra praleidžiama.

4.2.4. Genetinio algoritmo analizė

Žinoma, kad genetinis algoritmas yra paremtas evoliuciniu gamtos modeliu, kuomet didėjant kartų skaičiui individai tampa vis tobulesni ir labiau prisitaikę prie aplinkos sąlygų. Įgyvendinant šį metodą būtina pasirinkti rekombinacijos tikimybę p_c, kuri paprastai yra tarp 0.6 ir 1.0, bei mutacijos tikimybę p_m, kuri paprastai artima 0 [16]. Remiantis atlikta ankstesnių darbų literatūros analize, vykdant kompiuterinius eksperimentus su genetiniu algoritmu, pasirinkti sekantys parametrai: populiacija $P_g = 120$, $p_c = 0.7$, $p_m = 0.15$, maksimalus kartų skaičius 20000. Be to, buvo vykdomi vieno taško rekombinacijos ir pavienės mutacijos (angl. *Single Mutation*) metodai.

11 lentelė. Rezultatai gauti taikant genetinį algoritmą pilniems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Vidutinis kartų skaičius $\bar{g}$**	**Chromatinis skaičius $\chi(G)$**
f_graph_100	100	4950	1	0.2969	0.3125	0.3078	1	100
f_graph_200	200	19900	1	1.1561	1.1733	1.1645	1	200
f_graph_300	300	44850	1	2.6875	2.7370	2.7103	1	300
f_graph_500	500	124750	1	8.3906	8.4937	8.4202	1	500
f_graph_1000	1000	499500	1	38.3667	39.2499	38.6101	1	1000

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Dėl tos pačios priežasties, kaip ir atkaitinimo modeliavimo algoritmo analizės atveju, didžiausias nagrinėtas pilnas, reguliarus ir atsitiktinis grafas sudarytas iš 1000 viršūnių, tai matyti atitinkamai iš 11, 12 ir 13 lentelių. Nustatant chromatinį skaičių pilniems grafams nepriklausomai nuo jų dydžio, užtenka sugeneruoti vieną naują kartą, kad būtų gautas optimalus sprendinys (žr. 11 lentelę). Iš 11 lentelės taip pat matyti, kad algoritmo skaičiavimo laikas yra tiesiogiai proporcingas grafo dydžiui.

12 lentelė. Rezultatai gauti taikant genetinį algoritmą reguliariems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Vidutinis kartų skaičius $\bar{g}$**	**Chromatinis skaičius $\chi(G)$**
r_g_100_17	100	850	0.17	1.7692	9.3516	3.7140	115.3	7
r_g_200_90	200	9000	0.45	10.2190	38.1183	17.568	104.3	27
r_g_300_80	300	12000	0.27	33.3092	202.5107	60.1227	174.9	23
r_g_500_46	500	11500	0.09	30.1173	111.6723	47.6686	116	15
r_g_1000_400	1000	200000	0.40	560.6320	691.3202	615.1323	87.1	96

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Siekiant įvertinti briaunų pasiskirstymo įtaką sprendinio radimui kaip ir ankstesnių algoritmų atveju chromatinis skaičius buvo nustatomas vienodo dydžio ir tankumo atitinkamai reguliariems ir atsitiktiniams grafams. Iš 12 ir 13 lentelių matyti, kad briaunų pasiskirstymas (tolygus ar atsitiktinis) nedaro jokios įtakos apskaičiuotam chromatiniam skaičiui. Tuo tarpu vidutinis genetinio algoritmo vykdymo laikas ir vidutinis kartų skaičius daugeliu atvejų buvo mažesni reguliariuose grafuose.

13 lentelė. Rezultatai gauti taikant genetinį algoritmą atsitiktiniams grafams.

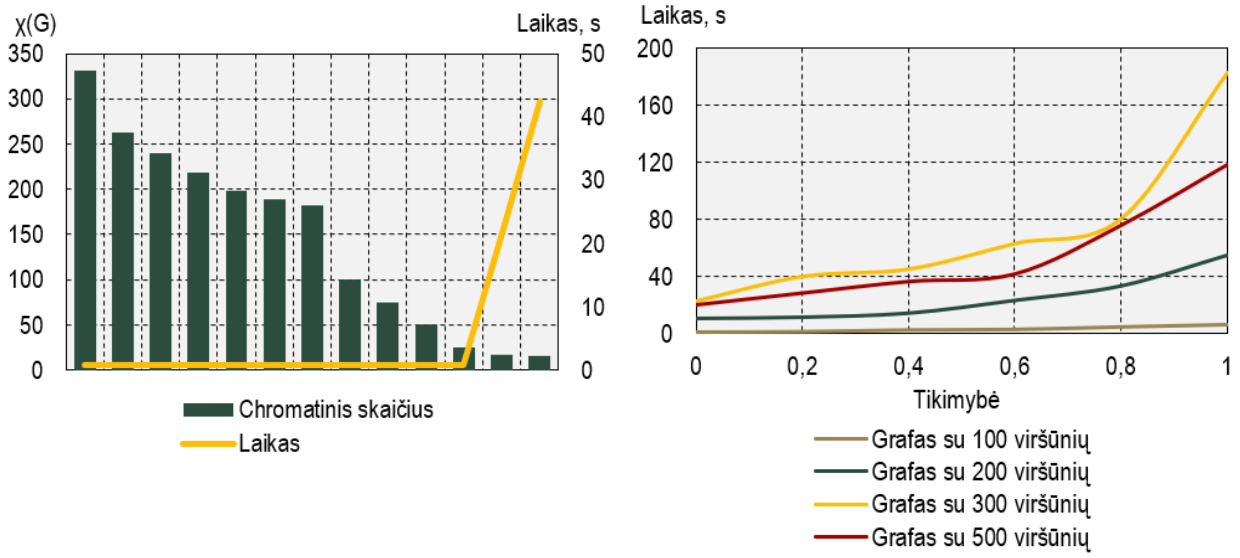
Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Vidutinis kartų skaičius $\bar{g}$**	**Chromatinis skaičius $\chi(G)$**
a_graph_100	100	850	0.17	2.5119	7.1940	5.2821	149.9	7
a_graph_200	200	9000	0.45	20.7174	38.1863	27.3478	104.4	27
a_graph_300	300	12000	0.27	34.4434	134.0884	63.0208	169.1	23
a_graph_500	500	11500	0.09	32.0491	68.4794	42.3671	111.2	15
a_graph_1000	1000	200000	0.40	521.8465	631.4832	582.0238	87.2	96

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Analogiškai atkaitinimo modeliavimo metodui vykdant genetinį algoritmą pradinėje atlikimo stadijoje taip pat būtina užduoti parametrą, kuris nusako iš kelių spalvų generuojama pradinės populiacijos kiekviena chromosoma. Iš 12 paveikslo iliustracijos kairėje matyti, kad ir šiam algoritmui būdinga atvirkštinė skaičiavimų laiko ir chromatinio skaičiaus priklausomybė, t. y. siekiant nustatyti tikslesnį chromatinį skaičių tam pačiam grafui (šiuo atveju *a_graph_500* grafui iš 13 len-

telės), užduodant mažesnį skaičių, nusakantį iš kelių spalvų sudaroma kiekviena chromosoma, algoritmo vykdymo trukmė auga. Tačiau pastaroji išvada netaikytina pilnų grafų atveju: generuojant pradinę populiaciją iš spalvų, kurių yra daugiau nei viršūnių grafe, visada apskaičiuojamas optimalus chromatinis skaičius ir per žymiai trumpesnį laiką. 12 ir 13 lentelėse pateikti geriausi rezultatai tikslumo prasme – apskaičiuoti mažiausi įmanomi chromatiniai skaičiai sugeneruotiems grafams. 11 lentelėje rezultatai pateikti, generuojant pradinę populiaciją iš spalvų skaičiaus lygaus grafo viršūnių skaičiui.



12 pav. Genetinio algoritmo analizė. Iliustracija kairėje: vidutinio skaičiavimų laiko ir chromatinio skaičiaus priklausomybė (skaičiavimai atlikti su 500 viršūnių grafu, kai tankumas lygus 0.09). Iliustracija dešinėje: genetinio algoritmo vykdymo trukmės priklausomybė nuo pasirinktos rekombinacijos tikimybės p_c.

12 paveikslo dešinėje iliustracijoje pateikta skaičiavimų laiko ir rekombinacijos tikimybės p_c priklausomybė. Matyti, kad pastaroji yra tiesiogiai proporcinga – kuo pasirinkta didesnė tikimybė, tuo algoritmo vykdymas ilgesnis. Tačiau chromatinio skaičiaus įverčiui tai įtakos neturi. Be to, kaip ir atkaitinimo modeliavimo algoritmo atveju, skaičiavimų laikas taip pat ilgėja didinant grafo dydį (viršūnių skaičių), kai pastarojo tankumas nekinta, arba didinant briaunų skaičių (tankumą), kai viršūnių skaičius išlieka pastovus.

4.2.5. Tabu paieškos algoritmo analizė

Anksčiau minėta, kad šis metodas remiasi tam tikrų sprendinių uždraudimo idėja, o pagrindinis jo skiriamasis bruožas – tabu sąrašas (angl. *tabu list*) arba draudimų sąrašas. Duoto algoritmo valdantieji (pagrindiniai) parametrai yra aplinkos didumas ir tabu sąrašo ilgis. Kompiuteriniams eksperimentams atlikti buvo pasirinkti sekantys parametrai: aplinkos didumas – 100, tabu sąrašo ilgis lygus grafo viršūnių skaičiui ir leistinas iteracijų skaičius 10^5.

Siekiant išlaikyti darbo vientisumą ir algoritmų palyginamumą, atliekant Tabu paieškos algoritmo analizę, kaip ir ankstesniais dviem atvejais, didžiausias nagrinėtas pilnas, reguliarus ir atsitiktinis grafas sudarytas iš 1000 viršūnių, tai matyti atitinkamai iš 14, 15 ir 16 lentelių. Be to, ir šiuo atveju turint tokio dydžio grafą sprendinys dar apskaičiuojamas per pakankamai trumpą (priimtiną) laiką.

14 lentelė. Rezultatai gauti taikant Tabu paieškos algoritmą pilniems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
f_graph_100	100	4950	1	0.0198	0.0342	0.0249	100
f_graph_200	200	19900	1	0.1629	0.1901	0.1792	200
f_graph_300	300	44850	1	0.6203	0.6797	0.6442	300
f_graph_500	500	124750	1	3.2602	3.5405	3.3837	500
f_graph_1000	1000	499500	1	29.0609	31.0340	29.9013	1000

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Iš 15 ir 16 lentelių matyti, kad Tabu paieškos algoritmo atveju lyginant du vienodo dydžio grafus (viršūnių ir briaunų skaičius sutampa), tačiau su skirtingu briaunų pasiskirstymu (pirmuoju atveju tolygus, antruoju – atsitiktinis), gaunamas tas pats chromatinis skaičius. Tačiau reguliariuose grafuose vidutinis skaičiavimų laikas visada buvo trumpesnis.

15 lentelė. Rezultatai gauti taikant Tabu paieškos algoritmą reguliariems grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
r_g_100_17	100	850	0.17	0.0312	0.5156	0.2079	8
r_g_200_90	200	9000	0.45	18.4035	75.1734	38.5767	27
r_g_300_80	300	12000	0.27	9.9063	34.7221	19.5449	24
r_g_500_46	500	11500	0.09	25.1741	107.5596	47.3389	15
r_g_1000_400	1000	200000	0.40	1254.3358	1363.2294	1312.5667	96

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Analogiškai atkaitinimo modeliavimo metodui vykdant Tabu paieškos algoritmą pradinėje atlikimo stadijoje taip pat privaloma sugeneruoti atsitiktinį spalvų rinkinį, todėl būtina užduoti parametrą, kuris nusako iš kelių spalvų pastarasis generuojamas. Iš 13 paveikslo iliustracijos kairėje matyti, kad šiam algoritmui taip pat būdinga atvirkštinė skaičiavimų laiko ir chromatinio skaičiaus priklausomybė: siekiant nustatyti tikslesnį chromatinį skaičių tam pačiam grafui (šiuo atveju *a_graph_500* grafui iš 16 lentelės), užduodant mažesnį skaičių, nusakantį iš kelių spalvų sudaromas pradinis spalvų variantas, algoritmo vykdymo trukmė auga. Tačiau ši išvada netaikytina pilniems grafams: generuojant pradinį spalvų rinkinį iš spalvų, kurių yra daugiau nei viršūnių grafe, visada apskaičiuojamas optimalus chromatinis skaičius ir per žymiai trumpesnį laiką. 15 ir 16 lentelėse pateikti geriausi rezultatai tikslumo prasme – apskaičiuoti mažiausi įmanomi chroma-

tiniai skaičiai sugeneruotiems grafams. 14 lentelėje rezultatai pateikti, generuojant pradinį spalvų rinkinį iš spalvų skaičiaus lygaus grafo viršūnių skaičiui.

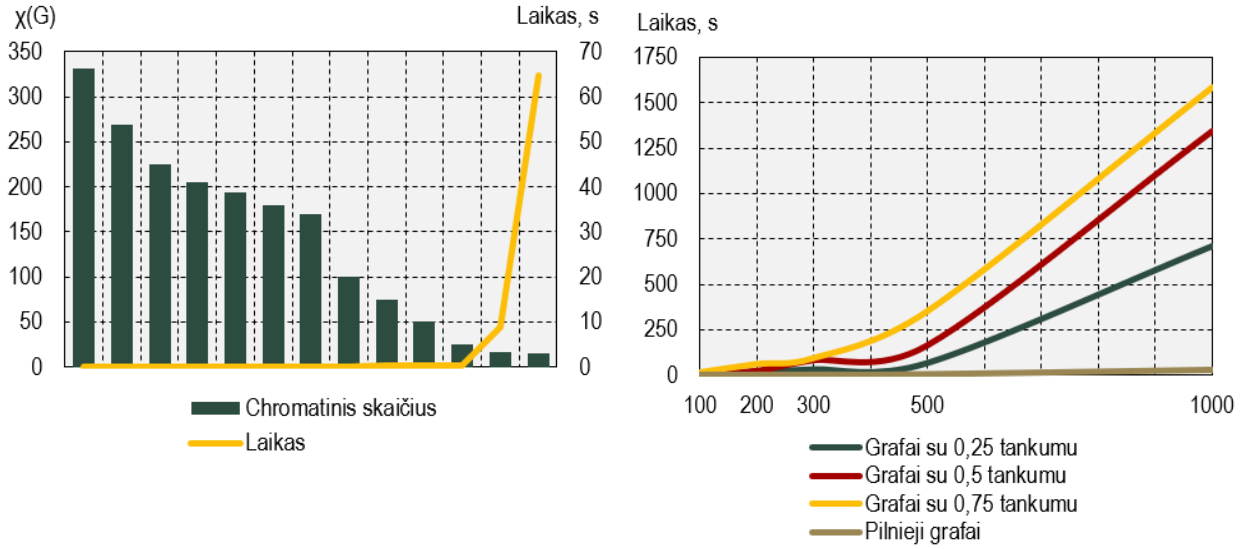
16 lentelė. Rezultatai gauti taikant Tabu paieškos algoritmą atsitiktiniams grafams.

Grafas	**V**	**E**	**D**	**Geriausias laikas**	**Blogiausias laikas**	**Vidutinis laikas**	**Chromatinis skaičius** $\chi(G)$
a_graph_100	100	850	0.17	0.0625	2.5339	0.8206	8
a_graph_200	200	9000	0.45	13.6889	76.5858	49.0877	27
a_graph_300	300	12000	0.27	12.8276	43.8674	30.0172	24
a_graph_500	500	11500	0.09	49.5422	91.9701	64.6610	15
a_graph_1000	1000	200000	0.40	1084.5248	1625.8799	1342.1494	96

Šaltinis: sudaryta autoriaus.

Pastaba: laikas pateiktas sekundėmis. Algoritmas vykdytas su visais grafais po 10 kartų.

Kalbant apie algoritmo vidutinės vykdymo trukmės ir grafo dydžio (kai tankumas yra pastovus) priklausomybę, ir čia matomas tiesioginis ryšys (žr. 13 paveikslo iliustraciją dešinėje). Be to, galima pastebėti, kad kuo tankesnis grafas, tuo skaičiavimų laikas taip pat ilgesnis. Tačiau, kai tankumas lygus 1, t. y. turime pilną grafą, algoritmo vykdymo trukmė mažesnė. Kaip ir atkaitinimo modeliavimo algoritmo atveju, tai galima paaiškinti tokiu būdu: visų pirma, užduodamas didesnis parametras, kuris nusako iš kelių spalvų generuojamas pradinis atsitiktinis spalvų derinys, o kaip minėta aukščiau tai sąlygoja spartesnį skaičiavimą. Antra, turint pilną grafą greičiau surandamos konfliktuojančios viršūnės, todėl dalis patikrinimų praleidžiama.



13 pav. Tabu paieškos algoritmo analizė. Iliustracija kairėje: vidutinio skaičiavimų laiko ir chromatinio skaičiaus priklausomybė (skaičiavimai atlikti su 500 viršūnių grafu, kai tankumas lygus 0.09). Iliustracija dešinėje: vidutinio skaičiavimų laiko priklausomybė nuo grafo dydžio (viršūnių skaičiaus), kai grafo tankumas pastovus.

4.2.6. Euristinių algoritmų rezultatų palyginamoji analizė

Šioje darbo dalyje atliekamas euristinių algoritmų tarpusavio palyginimas, t. y. pagal pasirinktus vertinimo kriterijus nustatoma, kuris iš metodų turi sparčiausią skaičiavimų greitaveiką ir artimiausią optimaliam sprendinį.

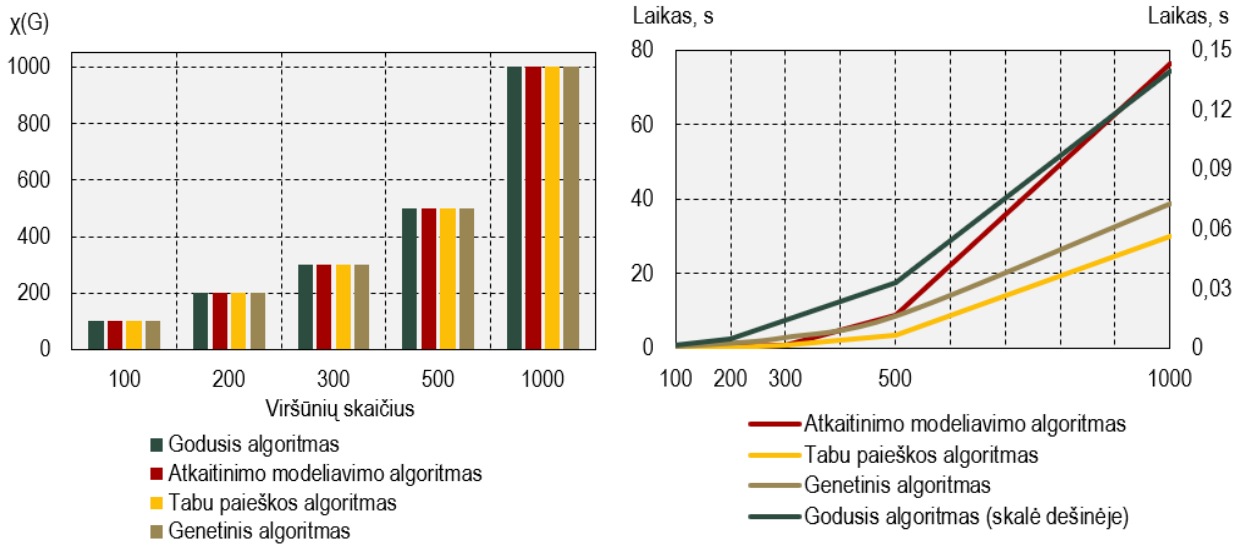
17 lentelė. Algoritmų palyginamumo analizė pilniems grafams.

Grafas	V	E	D	Godusis algoritmas		Atkaitinimo modeliavimo algoritmas		Genetinis algoritmas		Tabu paieškos algoritmas	
				s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$
f_graph_100	100	4950	1	0.0016	100	0.062	100	0.308	100	0.025	100
f_graph_200	200	19900	1	0.0047	200	0.497	200	1.164	200	0.179	200
f_graph_300	300	44850	1	0.0141	300	1.701	300	2.710	300	0.644	300
f_graph_500	500	124750	1	0.0328	500	8.693	500	8.420	500	3.384	500
f_graph_1000	1000	499500	1	0.1391	1000	76.40	1000	38.61	1000	29.90	1000

Šaltinis: sudaryta autoriaus.

Pastaba: oranžine spalva paryškinti langeliai žymi geriausią skaičiavimų laiką, žalsva – mažiausią chromatinį skaičių.

Iš pradžių panagrinėkime trivialų pilnų grafų atvejį. Iš 17 lentelės matyti, kad kiekvienas algoritmas nustato optimalų chromatinį skaičių, tačiau visais atvejais godusis algoritmas apskaičiuodavo sprendinį žymiai greičiau. Kaip jau žinoma, atkaitinimo modeliavimo, genetinio ir Tabu paieškos algoritmų skaičiavimų atlikimą pilniems grafams galima paspartinti ir gauti tą patį rezultatą pradiniame algoritmų vykdymo etape užduodant didesnį skaičių, nusakantį iš kelių spalvų sudaromas pradinis spalvų rinkinys ar pradinės populiacijos kiekviena chromosoma, nei grafo viršūnių skaičius. Tačiau ir tada godžiojo algoritmo skaičiavimų trukmė yra trumpiausia. Tuo tarpu beveik



14 pav. Algoritmų palyginamumo analizė pilniems grafams. Iliustracija kairėje: apskaičiuotas chromatinis skaičius euristiniais algoritmais skirtingo dydžio grafams. Iliustracija dešinėje: algoritmų vykdymo laikų priklausomybė nuo grafo viršūnių skaičiaus.

visais atvejais genetinis algoritmas pasirodė esąs lėčiausias nustatant chromatinį skaičių pilniems grafams.

14 paveiksle iliustracijoje dešinėje pavaizduota algoritmų vykdymo laikų priklausomybė nuo grafo viršūnių skaičiaus. Kaip matyti, pastaroji yra tiesioginė, t. y. didėjant pilnojo grafo dydžiui visų algoritmų skaičiavimų trukmė taip pat ilgėja.

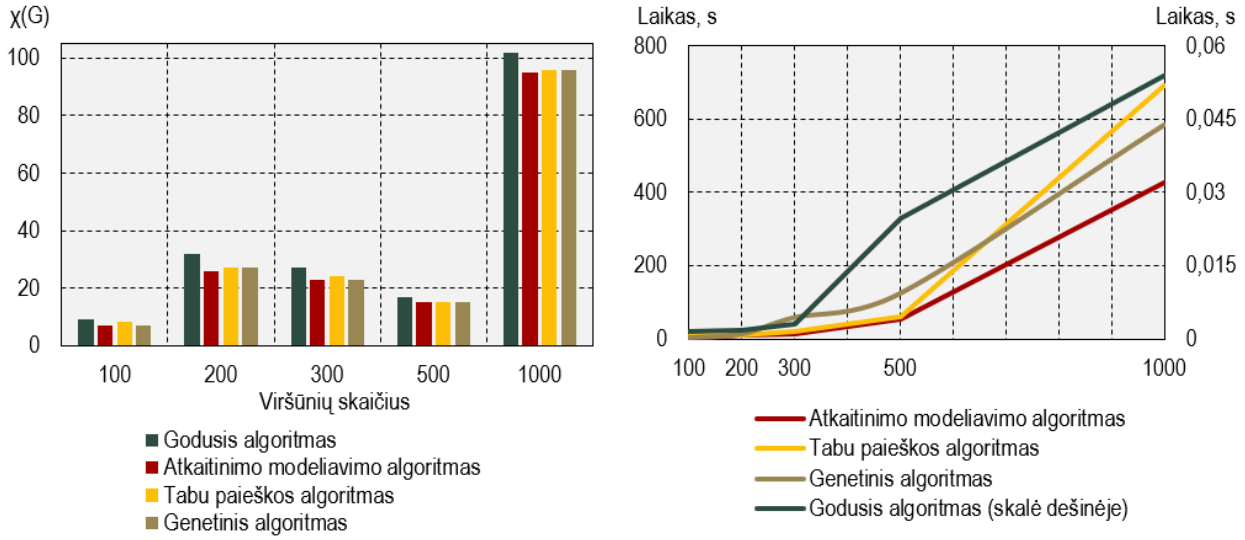
18 lentelė. Algoritmų palyginamumo analizė reguliariems grafams.

Grafas	**V**	**E**	**D**	**Godusis algoritmas**		**Atkaitinimo modeliavimo algoritmas**		**Genetinis algoritmas**		**Tabu paieškos algoritmas**	
				s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$
r_g_100_17	100	850	0.17	0.002	9	0.4	7	3.71	7	0.21	8
r_g_200_90	200	9000	0.45	0.003	32	10.2	26	17.57	27	38.58	27
r_g_300_80	300	12000	0.27	0.003	27	12.35	23	60.12	23	19.55	24
r_g_500_46	500	11500	0.09	0.005	17	10.53	15	47.67	15	47.34	15
r_g_1000_400	1000	200000	0.40	0.059	102	623.7	95	615.1	96	1312.6	96

Šaltinis: sudaryta autoriaus.

Pastaba: oranžine spalva paryškinti langeliai žymi geriausią skaičiavimų laiką, žalsva – mažiausią chromatinį skaičių.

Kalbant apie reguliarius grafus, t. y. situaciją, kada yra galimybė nustatyti kiekvienai grafo viršūnei vienodą kaimynų skaičių, iš 18 lentelės ir 15 paveikslo iliustracijoje kairėje matyti, kad visais atvejais artimiausią optimaliam rezultatą pateikdavo atkaitinimo modeliavimo algoritmas. Tikslumo prasme tiek atkaitinimo modeliavimo, tiek genetinis, tiek Tabu paieškos metodai buvo pranašesni už godųjį algoritmą – nagrinėtų grafų chromatinis skaičius vidutiniškai 4.2, 3.8 ir 3.4 spalvos atitinkamai gavosi mažesnis. Tačiau vėlgi greičiausiai skaičiavimus atlikdavo godusis algoritmas. Atliekant eksperimentus su reguliariais grafais Tabu paieškos algoritmas buvo beveik visada lėtesnis už atkaitinimo modeliavimo ir genetinį metodus.



15 pav. Algoritmų palyginamumo analizė reguliariems grafams. Iliustracija kairėje: apskaičiuotas chromatinis skaičius euristiniais algoritmais skirtingo dydžio grafams. Iliustracija dešinėje: algoritmų vykdymo laikų priklausomybė nuo grafo viršūnių skaičiaus, kai tankumas yra ∼0.25.

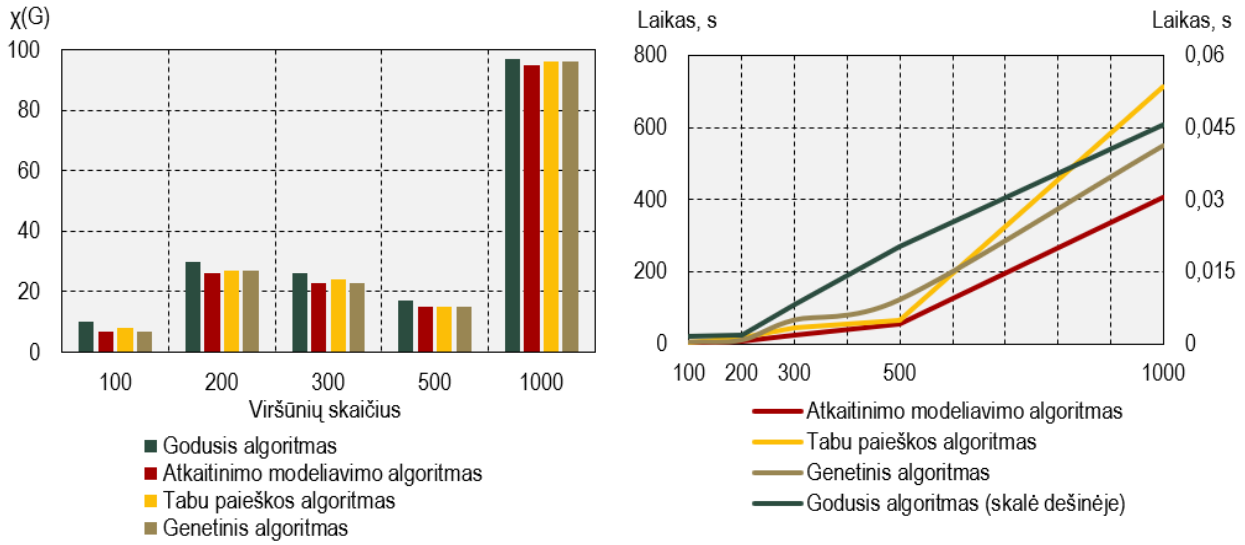
19 lentelė. Algoritmų palyginamumo analizė atsitiktiniams grafams.

Grafas	V	E	D	Godusis algoritmas		Atkaitinimo modeliavimo algoritmas		Genetinis algoritmas		Tabu paieškos algoritmas	
				s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$	s	$\chi(G)$
a_g_100	100	850	0.17	0.002	10	0.70	7	5.28	7	0.82	8
a_g_200	200	9000	0.45	0.005	30	12.82	26	27.35	27	49.09	27
a_g_300	300	12000	0.27	0.005	26	14.55	23	63.02	23	30.02	24
a_g_500	500	11500	0.09	0.006	17	13.55	15	42.37	15	64.66	15
a_g_1000	1000	200000	0.40	0.058	97	568.02	95	582.02	96	1342.1	96

Šaltinis: sudaryta autoriaus.

Pastaba: oranžine spalva paryškinti langeliai žymi geriausią skaičiavimų laiką, žalsva – mažiausią chromatinį skaičių.

Nagrinėjant atsitiktinių grafų atvejį, kuomet kiekvienai grafo viršūnei kaimynų skaičius parenkamas atsitiktinai, iš 19 lentelės ir 16 paveikslo iliustracijoje kairėje matyti, kad kaip ir reguliarių grafų eksperimentų metu geriausią rezultatą pateikdavo atkaitinimo modeliavimo algoritmas. Tikslumo prasme tiek atkaitinimo modeliavimo, tiek genetinis, tiek Tabu paieškos metodai buvo pranašesni už godųjį algoritmą – nagrinėtų grafų chromatinis skaičius vidutiniškai 2.6, 2.4 ir 2 spalvos atitinkamai gavosi mažesnis. Tačiau vėlgi greičiausiai skaičiavimus atlikdavo godusis algoritmas.



16 pav. Algoritmų palyginamumo analizė atsitiktiniams grafams. Iliustracija kairėje: apskaičiuotas chromatinis skaičius euristiniais algoritmais skirtingo dydžio grafams. Iliustracija dešinėje: algoritmų vykdymo laikų priklausomybė nuo grafo viršūnių skaičiaus, kai tankumas lygus 0.25.

Prieš tai atlikti eksperimentai implikuoja, kad atkaitinimo modeliavimo algoritmo pagalba apskaičiuotas chromatinis skaičius nepilniems grafams yra tiksliausias. Tuo tarpu pastarojo, genetinio ir Tabu paieškos algoritmų vykdymo laikai yra ilgesni, tačiau tai kompensuoja gaunami tikslesni rezultatai nei godžiojo algoritmo pagalba. Priklausomai nuo iškeltų tikslų, tenka rinktis vieną ar kitą metodą – dėl tikslesnio sprendinio turi būti aukojamas laikas ir atvirkščiai.

5. Tvarkaraščių sudarymo internetinė aplikacija

Švietimo įstaigų tvarkaraščio sudarymas – tai gerai žinomas bendresnės tvarkaraščio problemos pavyzdys. Atsižvelgiant į tai, kaip gerai sudarytas tvarkaraštis, priklauso studentų mokomosios medžiagos įsisavinimas, dėstytojų darbo efektyvumas, racionalus materialinių išteklių naudojimas. Kokybiškai sudaryti tvarkaraščiai turėtų užtikrinti tolygų studentų grupių ir dėstytojų apkrovimą. Be to, šiais laikais sudarant tvarkaraščius siekiama optimizuoti mokymosi, psichologinę ir fizinę naštą bei kitas naujoves.

Šiame skyriuje aptariamos Lietuvoje labiausiai naudojamos pusiau automatinės sistemos, leidžiančios sudaryti tvarkaraščius, suformuluojamas mokymosi įstaigų tvarkaraščių sudarymo uždavinys ir pristatoma internetinė aplikacija, kuri išsprendžia pastarąjį uždavinį euristinio algoritmo dėka.

5.1. Esamų tvarkaraščių sudarymo programų apžvalga

Šiais laikais sutinkama gana daug sistemų, kurios leidžia automatizuotai sudaryti tvarkaraščius. Tokios sistemos gali būti bendrojo naudojamo arba specializuotos. Specializuotų programų dėka dažniausiai sudaromi bendrojo lavinimo mokyklos arba aukštosios mokyklos (universiteto ar kolegijos) tvarkaraščiai. Šios priemonės taip pat skiriasi savo funkcionalumu ir lankstumu:

- *automatinės* – kai vartotojas tik įveda reikiamus duomenis, o visą tvarkaraščio sudarymo procesą atlieka programa;
- *pusiau automatinės* – kai pagal įvestus duomenis programa sugeneruoja tvarkaraštį, kuris pateikiamas vartotojui korekcijoms atlikti;
- *pagalbinės* – kurios tvarkaraščio negeneruoja, tačiau sukuria vartotojui patogią aplinką tvarkaraščiui sudaryti.

Susipažinus su keliais švietimo įstaigoms siūlomais produktais, toliau pateikiamos dvi dominuojančios Lietuvoje pusiau automatinės sistemos, leidžiančios sudaryti tvarkaraščius: programa „aSc Timetables“ ir programa „Mimosa“. Šios priemonės sugeneruoja tvarkaraštį pagal įvestus duomenis ir leidžia jį koreguoti vartotojui.

5.1.1. Programa „aSc Timetables“

Programą „aSc Timetables“ yra sukūrusi Slovakijos IT kompanija „Applied Software Consultants s.r.o.“. Šios sistemos platintojų interneto svetainėje nurodyta, kad kol kas tai vienintelė rinkoje tvarkaraščių kūrimo programa lietuvių kalba, kuria naudojasi daugiau nei 300 mokymo įstaigų Lietuvoje [31]. Programos „aSc Timetables“ dėka galima atlikti šias pagrindines tvarkaraščio sudarymo užduotis:

1. Įvesti ir išsaugoti pagrindinius duomenis: klases (grupes), mokytojus (dėstytojus), kabinetus (auditorijas), pamokas (paskaitas), savaitės dienas kada vyksta pamokos ar užsiėmimai ir kt.
2. Užduoti įvairius apribojimus: laiką, kada gali vykti pamokos ar paskaitos, mokytojų ar dėstytojų darbo laiką, sąryšius tarp skirtingų disciplinų (kaip jos turi būti paskirstytos per savaitę, kokios disciplinos negali eiti viena po kitos, kada turi vykti ir pan.), dėstomų dalykų skaičių ir daugelį kitų dalykų.
3. Sukurti tvarkaraštį: programa naudoja algoritmą, leidžiantį jai sudaryti tvarkaraštį per keliolika minučių, sudedant į tvarkaraštį visas pamokas ir užsiėmimus. Vėliau galima tvarkaraštį redaguoti, keičiant sąlygas ar net rankiniu būdu išdėstant užsiėmimus. Taip pat programa

turi testavimo įrankius problemai surasti, tuo atveju, jeigu pasirodo, kad tvarkaraščio sukurti neįmanoma dėl klaidingai suvestų duomenų ar dėl įvestų pernelyg griežtų sąlygų.

4. Atspausdinti kiekvienos grupės, kabineto ar mokytojo (dėstytojo) tvarkaraštį, ar bendrus tvarkaraščius. Taip pat tvarkaraščius galima eksportuoti $Excel$ lentelių (naudojant modulį „Tvarkaraščiai Internete“) ar $html$ formatu ir patalpinti švietimo įstaigos interneto puslapyje ar vietiniame kompiuterių tinkle.

5.1.2. Programa „Mimosa“

Tvarkaraščių sudarymo programa sukurta Suomijos kompanijos „Mimosa Software Ltd.“. Naujausia programos „Mimosa“ 7.2.0 versija pasirodė 2018 m. rugsėjo 16 d. Pasak kūrėjų šiandien programinė įrangą „Mimosa“ tiekiama į visų rūšių mokymo įstaigas – nuo vaikų darželių iki universitetų – daugiau nei 80 šalių [32].

Ši programa leidžia sudaryti tvarkaraščius tiek rankiniu būdu, tiek automatiškai. Kuriant tvarkaraščius rankiniu būdu, vartotoją visuomet lydi „Mimosa“ vadovas, kuris pataria, kaip planuoti, pašalinti, perkelti paskaitas į tvarkaraščius. Taip pat galima anuliuoti ir perrašyti visus arba pasirinktus pakeitimus ir grįžti į bet kurį ankstesnį kūrimo etapą. Programa padeda išvengti klaidų, nes sudaryta galimybė stebėti, kas įvyksta visuose kituose tvarkaraščiuose, kai pakeičiamas vienas iš jų. Kai pasirenkamas automatinis sudarymo būdas, kompaktiški ir tvarkingi tvarkaraščiai gaunami per kelias minutes. Žinoma, vartotojas visada gali pasirinkti, ką koreguoti ar optimizuoti ir kaip tai padaryti, be to, galima derinti automatines ir rankines užduotis. Galima išvardinti tokias pagrindines šios programos funkcijas:

- trys tvarkaraščio sudarymo būdai (rankinis, automatinis, mišrus);
- galimybė automatiškai sudaryti keletą variantų tvarkaraščių ir pasirinkti geriausią;
- įvesti ir išsaugoti pagrindinius duomenis apie klases, mokytojus, kabinetus, pamokas ir kt.;
- tvarkaraščių konsolidavimas;
- optimizavimo kriterijų pasirinkimas;
- tvarkaraščio galimumo patikrinimas;
- tvarkaraščius eksportuoti $Excel$ lentelių ar $html$ formatu ir patalpinti internete;
- kitos funkcijos.

5.2. Mokymosi įstaigų tvarkaraščių sudarymo uždavinys

Tvarkaraščių teorija yra užsiėmimų tvarkaraščių sudarymo pagrindas. Ji plačiai taikoma tiek organizuojant darbą įmonėse, tiek – uždavinius švietimo įstaigose. Užsiėmimų tvarkaraštis formaliuoju požiūriu yra užsiėmimų išdėstymas laiko skalėje, atsižvelgiant į jiems iš anksto keliamus reikalavimus. Šiuos reikalavimus formuoja patys mokymosi proceso dalyviai bei rašytiniai dokumentai. Pradiniai šio proceso duomenys yra:

- G – mokymosi srautai, susidedantys iš besimokančių grupių ar pogrupių, kurie susidaro dėl grupių dalinimosi į atskirus vienetus.
- M – dėstytojai, pagrindinis poveikio mokymosi srautams mechanizmas.
- D – mokymosi disciplinos, pagrįstos mokymosi planu, kurį sudaro įvairių rūšių užsiėmimai.
- K – auditorijos, patalpos, kuriose vedami užsiėmimai.

Planavimo užduotis formuluojama kaip visų pradinių planavimo proceso duomenų variantų perrinkimas (Dekarto aibė $R = \{G \times M \times D \times K \times z\}$, čia z – užsiėmimų vykdymo laikai, nustatant optimumą pagal atitikimo tvarkaraščiui keliamiems reikalavimams kriterijų). Iš čia seka išvada

apie suformuluotos užduoties atlikimo sudėtingumą, nes priklausomai nuo jos sprendimo apimties derinių skaičius auga eksponentiškai, o tai padaro šį uždavinį NP-pilnuoju. Tačiau šis požiūris ne visada yra teisingas, nes jau pradinio pasirengimo planavimui metu derinių kiekis sumažinamas dėstytojo, srauto, auditorijos (arba galimos auditorijos) ir vedamo užsiėmimo pagal teminį planą apjungimo dėka.

Uždavinys apie tvarkaraščio sudarymą su apribojimais gali reikšti užsiėmimų paskirstymą laike tokiu būdu, kad tuo pačiu laiku nebūtų paskirtos paskaitos vienai grupei, vienam dėstytojui arba vienoje auditorijoje. Toks uždavinys lengvai suvedamas į grafų spalvinimą su sąlyga, kad visos auditorijos yra žinomos visoms grupėms, kiekvieną paskaitą veda paskirtas dėstytojas ir yra žinomos auditorijos, kuriose turi būti vedamos paskaitos. Šiuo atveju tvarkaraštis gali būti atvaizduotas grafo pavidalu, kuriame viršūnės atitinka užsiėmimus, o briaunomis sujungtos viršūnės, atitinkančios vienos grupės paskaitas, vienoje auditorijoje arba su vienu dėstytoju. Po tokio grafo nuspalvinimo tvarkaraštis sudaromas pagal principą: paskaitos, atitinkančios tos pačios spalvos viršūnes, gali būti pravestos vienu metu.

Aukščiau aptartą atvejį galima iliustruoti sekančiu pavyzdžiu: tarkime, kad turime dvi grupes, kiekvienai iš kurių būtina pravesti po keturis užsiėmimus. Kiekvieną užsiėmimą veda vienas iš keturių dėstytojų vienoje iš trijų auditorijų. 20 lentelėje pateikta užsiėmimų tvarka, t. y. koks dėstytojas kuriai grupei kurioje auditorijoje veda užsiėmimą – būtina sudaryti tvarkaraštį, kad dėstytojas ir besimokanti grupė tuo pačiu metu neturėtų kelių užsiėmimų bei vienu metu patalpoje nevyktų keli užsiėmimai. Pateiktus 20 lentelės duomenys galima atvaizduoti grafu[13], kuris pavaizduotas 17 paveikslo iliustracijoje kairėje. Pritaikius euristinį (nagrinėjamu atveju atkaitinimo modeliavimo) algoritmą išsprendžiamas grafo spalvinimo uždavinys, kurio sprendinys ir yra galutinis sudarytas tvarkaraštis – su sąlyga, kad iš anksto yra nurodyta, kuri spalva reiškia atitinkamą laiką, pavyzdžiui: raudona – pirmadienis 8 val., žalia – antradienis 8 val. ir t. t. (žr. 17 paveikslo iliustraciją dešinėje)

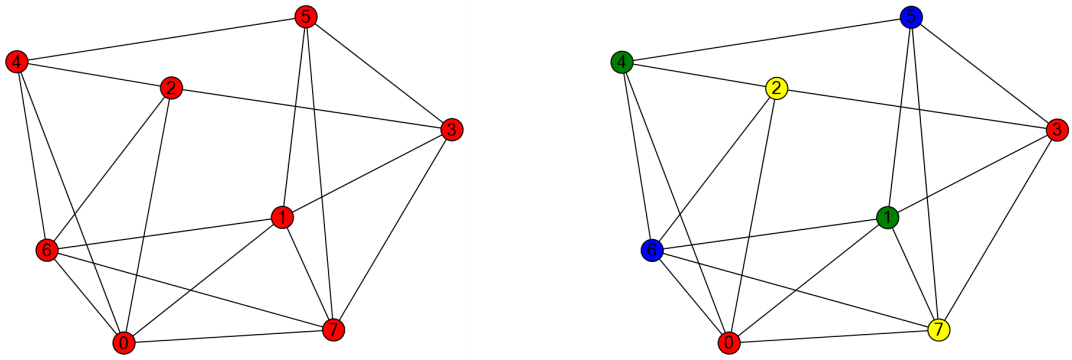
20 lentelė. Tvarkaraščių sudarymo uždavinys.

Nr.	Grupė	Dėstytojas	Disciplina	Auditorija
0	G1	M1	D1	K1
1	G2	M1	D1	K1
2	G1	M2	D2	K2
3	G2	M2	D2	K2
4	G1	M3	D3	K3
5	G2	M3	D3	K3
6	G1	M4	D4	K1
7	G2	M4	D4	K1

Šaltinis: sudaryta autoriaus.

Pastaba: G1 ir G2 yra atitinkamai pirma ir antra grupės, M1, M2, M3 ir M4 – atitinkamai pirmas, antras, trečias ir ketvirtas dėstytojai, D1, D2, D3, D4 – dėstytojų mokomos disciplinos, K1, K2, K3 – atitinkamai pirma, antra ir trečia auditorijos.

[13] Viršūnė yra trijų kintamųjų: grupės, dėstytojo, auditorijos – kombinacija, kuri turi kaimyninę viršūnę tuomet, kai bent vienas kintamasis sutampa.



17 pav. Tvarkaraščių sudarymo uždavinys. Iliustracija kairėje: pagal 20 lentelę sudarytas konfliktinis grafas. Iliustracija dešinėje: išspręstas grafo spalvinimo uždavinys, kur kiekviena spalva reiškia paskaitos laiką, pavyzdžiui: raudona – pirmadienis 8 val., žalia – antradienis 8 val. ir pan.

Žymiai sudėtingesnė uždavinio versija gaunama, kai kiekvienai paskaitai nustatoma konkretaus tipo auditorija, o bendras kiekvieno tipo auditorijos skaičius ribotas. Pavyzdžiui, kompiuterinėse auditorijose negalima pravesti dešimt užsiėmimų, jei jų yra tik penkios. Tokios formuluotės uždavinio sudėtingumas pasireiškia tuo, kad panašūs apribojimai grafe negali būti atvaizduoti ryšių pavidalu. Pastarieji uždaviniai reikalauja papildomo apdorojimo arba sprendimo metu, arba nuspalvinus grafą.

Taip pat kaip spalvinimo problemą galime modeliuoti egzaminų tvarkaraščio sudarymą. Dviejų mokymosi dalykų egzaminai neturėtų būti paskirti tuo pačiu laiku, jei yra studentų, kurie lanko abu dalykus. Taigi galima modeliuoti tai kaip grafą, kuriame kiekvienas dalykas atitinka viršūnę, o dalykai yra sujungiami briauna, jei jie dalijasi studentais. Tuomet kyla klausimas, ar galime nuspalvinti grafą su k spalvomis, čia k yra egzaminų laikų skaičius mūsų tvarkaraštyje. Egzaminų tvarkaraščio sudarymo uždavinyje iš tikrųjų galime tikėtis atsakymo „ne“, nes norint panaikinti konfliktus reikės pernelyg didelio egzaminų laikų skaičiaus. Taigi reali praktinė problema yra sekanti: kiek mažai studentų turime išimti iš paveikslo (t. y. pateikti specialius konfliktinius egzaminus), kad būtų galima išspręsti spalvinimo problemą su pagrįsta (priimtina) k reikšme.

Apskritai kiekvienam tvarkaraščiui keliami tam tikri sudarymo reikalavimai, kurių privaloma laikytis. Jei tvarkaraštis neatitinka sudarymo taisyklių, jis laikomas netinkamu. Toliau išvardijami pagrindiniai reikalavimai [25]:

1. Užsiėmimus vedantis asmuo vienu metu gali turėti tik vieną užsiėmimą;
2. Besimokančiųjų grupė tuo pačiu metu gali turėti tik vieną užsiėmimą;
3. Vienu metu patalpoje gali vykti tik vienas užsiėmimas;
4. Konkretūs užsiėmimai gali vykti tik tam tikrose patalpose, kurios yra pritaikytos mokomo dalyko specifikai (laboratorijos, kompiuterių klasės, sporto salė ir pan.);
5. Patalpoje užsiėmimas gali vykti tik tada, kai joje telpa visi besimokantieji, turintys tą užsiėmimą.

5.3. Internetinės aplikacijos realizacija

Šis skyrelis skirtas aprašyti sukurtą internetinę aplikaciją, kuri leidžia švietimo įstaigoms sudaryti tvarkaraščius, o sprendimo branduolį sudaro atkaitinimo modeliavimo algoritmas.

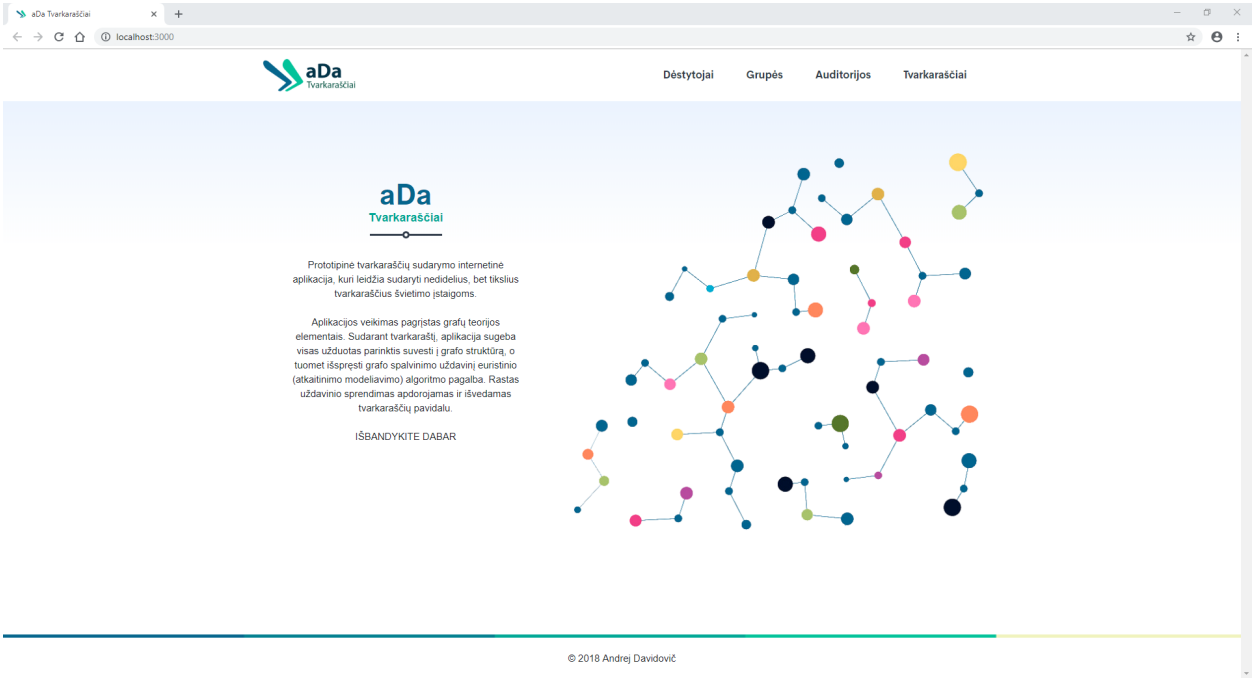
Programinė įranga

Internetinė aplikacija realizuota *Microsoft Visual Studio Professional 2017*, *Microsoft Visual Studio Code* programinės įrangos ir *C#* (naudota vidiniam (angl. *back-end*) programavimui), *TypeScript* (naudota išoriniam (angl. *front-end*) programavimui) programavimo kalbų pagalba, panaudojant *.NET Core 2.2* karkasą[14], *ReactJS* biblioteką (su *Redux* architektūra) ir *Bootstrap 4* karkasą. Aplikacijos dizainui plėtoti taip pat pritaikyta ir *SASS* technologija bei bandomoji *Photoshop CC 2018* programa.

Sukurta sistema nenaudoja duomenų bazės[15], kaip alternatyva duomenys saugomi naršyklės *Local Storage*, imituojant reliacinę duomenų bazę, o kiekvienas įrašas – objektas, kuris turi identifikacinį numerį (toliau – ID).

Apie internetinę aplikaciją „aDa tvarkaraščiai“

„aDa tvarkaraščiai“ – šio darbo metu sukurta prototipinė tvarkaraščių sudarymo internetinė aplikacija, kuri leidžia sudaryti nedidelius, bet tikslius tvarkaraščius švietimo įstaigoms. Sudarant tvarkaraštį, aplikacija sugeba visas užduotas parinktis suvesti į grafo struktūrą, o tuomet išspręsti grafo spalvinimo uždavinį atkaitinimo modeliavimo algoritmo pagalba. Rastas uždavinio sprendimas apdorojamas ir išvedamas tvarkaraščių pavidalu. Kitaip tariant ši aplikacija sprendžia uždavinį suformuluotą 5.2 skyrelyje. 18 paveiksle pateiktas „aDa tvarkaraščiai“ aplikacijos pradžios langas[16]. Įgyvendintą internetinę aplikaciją galima išbandyti nurodytu adresu: http://78.60.221.238.



18 pav. „aDa tvarkaraščiai“ aplikacijos pradžios langas.

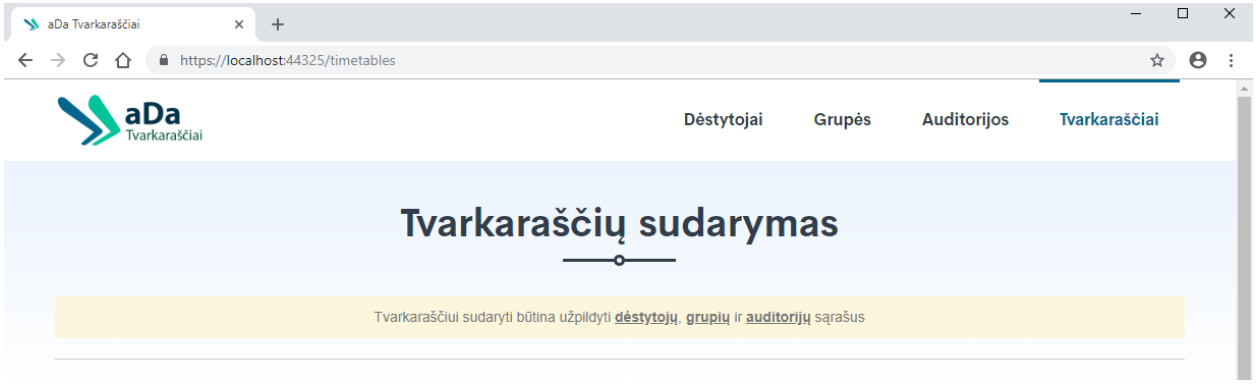
Duomenų įvedimas

Norint sudaryti tvarkaraštį per „aDa tvarkaraščiai“, būtina užpildyti dėstytojų, grupių ir auditorijų sąrašus (žr. 19 pav.). Atitinkamai aplikacijos „Dėstytojai“, „Grupės“ ir „Auditorijos“ languose

[14] Pagrindinė priežastis pasirenkant šį karkasą – suderinamumas ir paprastas programų vystymasis daugelyje operacinių sistemų.

[15] Ateityje planuojama plėsti aplikacijos funkcionalumą tuo pačiu prijungiant ir duomenų bazę.

[16] Kitus aplikacijos langus galima pamatyti darbo prieduose (žr. B priedą).



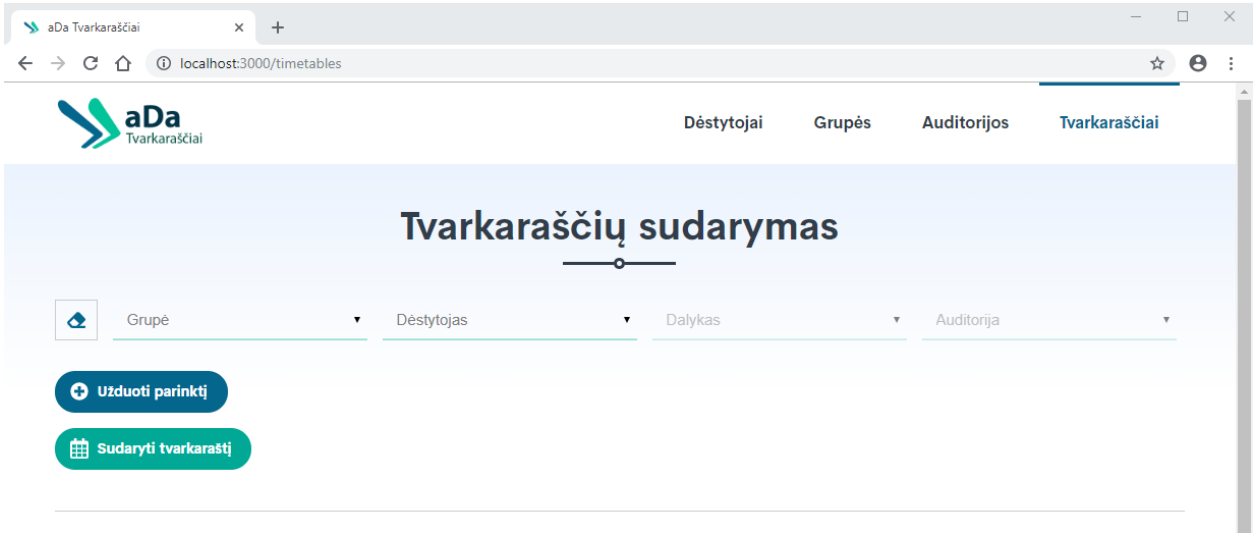
19 pav. Informacinis „aDa tvarkaraščiai“ aplikacijos pranešimas, kad būtina turėti užpildytus dėstytojų, grupių ir auditorijų sąrašus, norint sudaryti tvarkaraštį.

tam yra sukurtos įvedimo formos. Pastarosioms įgyvendinti buvo panaudota *Redux Form* technologija, kuri palengvina įvedamų duomenų validavimą ir pateikimą. Prie kai kurių įvedimo laukų pridėtas informacinis ženklas, ant kurio užvedus pateikiama papildoma informacija, pavyzdžiui, kaip teisingai turi būti užpildytas atitinkamas laukas.

Kaip minėta anksčiau, aplikacija duomenų saugojimui nenaudoja duomenų bazės, todėl, kai patvirtinama forma, sukuriamas objektas su unikaliu ID ir *JSON* formatu išsaugomas naršyklės *Local Storage* atitinkamame lauke. Be to, pridėtas įrašas iš karto matomas vartotojui sąraše esančiam šalia formos (duomenys atnaujinami realiu laiku be puslapio perkrovimo). Vartotojo patogumui kiekviena suvestų duomenų lentelė turi rūšiavimo ir filtravimo funkcionalumą (žr. B priedą), todėl galima greitai surasti ieškomą dėstytoją, grupę ar auditoriją. Paminėtina, kad esant poreikiui įrašas gali būti ištrintas iš sąrašo paspaudus pašalinimo mygtuką, kuris pridėtas prie kiekvienos duomenų eilutės iš dešinės pusės[17].

Tvarkaraščio sudarymas ir išsaugojimas

Kai dėstytojų, grupių ir auditorijų sąrašai yra užpildyti, aplikacija įgalina parinkčių sudarymą tvarkaraščio generavimui (žr. 20 pav.). Pridėjus parinkti būtina pasirinkti grupę, dėstytoją, discipliną ir auditoriją. Pastarosios dvi parinktys priklauso nuo pirmųjų dviejų: auditorijos išskleidžiamas



20 pav. „aDa tvarkaraščiai“ aplikacijos parinkčių sudarymas tvarkaraščio generavimui.

[17] Ateityje planuojama pridėti ir įvestų duomenų atnaujinimo funkcionalumą.

sąrašas yra neaktyvus tol, kol nėra pasirinkta grupė. Be to, leidžiama pasirinkti tik tą auditoriją, kurioje telpa pasirinktos grupės studentų skaičius. Panaši situacija ir su disciplinos išskleidžiamuoju sąrašu: pastarasis tampa aktyvus tik tuo atveju, kai yra pasirinktas dėstytojas. Be to, rodomos tik tos disciplinos, kurias gali dėstyti pasirinktas dėstytojas. Užduota parinktis gali būti ištrinta paspaudus pašalinimo mygtuką, kuris sukuriamas su kiekviena parinktimi, iš kairės pusės.

Svarbu paminėti, kad pridedant parinktis tas pats dėstytojas, grupė ar auditorija negali būti naudojamas daugiau nei 30 kartų. Priešingu atveju programos posistemė tvarkaraščio sudarymo metu vartotojui grąžins klaidos pranešimą, kad vienas iš pasirinkimų viršija nurodytą limitą. Nustatytas toks skaičius atsižvelgiant į realias sąlygas: tarkime, kad vienos paskaitos trukmė yra 2 akademinės valandos ir jos vyksta nuo 8 val. iki 20 val. Tuomet maksimalus kiekvieno dėstytojo, grupės ir auditorijos užimtumas per dieną – 6 užsiėmimai. Vadinasi, per savaitę 30 (5 darbo dienos po 6 užsiėmimus). Be to, yra sudaromos mokomosios programos, kurios nurodo, kiek užsiėmimų turi kiekvienas dėstytojas ir grupė – šis skaičius neviršija nustatyto aplikacijoje.

Pridėjus visas reikiamas parinktis ir paspaudus „Sudaryti tvarkaraštį“ mygtuką, nusiunčiama užklausa į programos posistemės logiką, kuri visas gautas parinktis priveda prie grafo struktūros ir išsprendžia grafo spalvinimo uždavinį atkaitinimo modeliavimo algoritmo pagalba. Sprendimas yra grąžinamas į vartotojo sąsają, kur išvedamas lentelių pavidalu (žr. 21 pav.). Paminėtina, kad sudarytas tvarkaraštis turi dviejų tipų tvarkaraščius: grupių ir dėstytojų – kurie, savo ruožtu, dar išskirstyti į kiekvienos grupės ir dėstytojo atskyrus tvarkaraščius.

Vardenis Pavardenis (Profesorius)

	Pirmadienis	Antradienis	Trečiadienis	Ketvirtadienis	Penktadienis
08:00-10:00	Netiesinio modeliavimo metodai Kompiuterinis modeliavimas (Magistrantūros studijos, I kursas, 2 grupė) 109 kab.				Netiesinio modeliavimo metodai Kompiuterinis modeliavimas (Magistrantūros studijos, I kursas, 1 grupė) 419 kab.
10:00-12:00	Netiesinio modeliavimo metodai Kompiuterinis modeliavimas (Magistrantūros studijos, I kursas, 1 grupė) 419 kab.		Netiesinio modeliavimo metodai Kompiuterinis modeliavimas (Magistrantūros studijos, I kursas, 2 grupė) 109 kab.		
12:00-14:00					
14:00-16:00					
16:00-18:00					
18:00-20:00					

21 pav. „aDa tvarkaraščiai“ aplikacijos sudaryto dėstytojo tvarkaraščio pavyzdys.

Internetinė aplikacija leidžia išsaugoti sukurtą tvarkaraštį: paspaudus „Išsaugoti tvarkaraštį“ mygtuką, atsidaro modalinis dialogas, kuriame vartotojo prašoma suvesti sudaryto tvarkaraščio pavadinimą. Išsaugotas tvarkaraštis pridedamas į tvarkaraščių sąrašą. Be to, yra galimybė peržiūrėti arba ištrinti kiekvieną tvarkaraštį esantį sąraše.

Pagrindiniai aplikacijos trūkumai

Kaip minėta, sukurta aplikacija yra prototipinės versijos, t. y. labiau parodomoji nei paruošta masiniam naudojimui, ir ateityje planuojama plėsti bei tobulinti jos funkcionalumą, tačiau šiuo metu galima išskirti šias pagrindines problemas, su kuriomis susiduriama naudojant „aDa tvarkaraščiai“:

- Vartotojo interfeiso netobulumas: sudarant didelius tvarkaraščius reikia pridėti daug parinkčių, todėl aplikacijos tvarkaraščių sudarymo langas slenkasi žemyn. Dėl šios priežasties vartotojas netenka galimybės matyti jau pridėtų parinkčių ir gali sukurti parinkčių dublikatų, taip sugeneruojant klaidingą tvarkaraštį. Vadinasi, kuriant didelius tvarkaraščius būtina patikrinti, ar nėra pridėta parinkčių dublikatų, jeigu jų neturi būti. Kitas būdas – sudarinėti tvarkaraščius labiau orientuotus į dėstytojus, t. y. iš pradžių suvesti pirmojo dėstytojo vedamas disciplinas, paskui antrojo, trečiojo ir t. t., arba į grupes – iš pradžių pridėti pirmosios grupės užsiėmimus, po to antrosios, trečiosios ir t. t.
- Duomenų bazės nebuvimas: kadangi šiuo metu aplikacija nenaudoja duomenų bazės, tai kiekvienas sukurtas tvarkaraštis yra atskira „istorija“, t. y. sudarius pirmąjį tvarkaraštį, o paskui generuojant tokį patį ir antrąjį (galima sudaryti ir kitą, tačiau panaudojant kelis tuos pačius dėstytojus arba grupes, arba auditorijas) nėra galimybės iš karto (sudarymo metu) patikrinti, ar naujas tvarkaraštis nekonfliktuoja su jau turimais ir pranešti apie tai vartotojui, kad reikia patikslinti arba pakeisti vieną, kitą parinktį. Praktiškai tai gali būti realizuota ir dabar, tačiau patikrinimas vyktų vartotojo pusėje, kas iš geros programavimo praktikos yra negerai, nes tai pablogina programos veikimą (angl. *performance*). Vadinasi, siekiant dabar išvengti minėtos problemos reikia kurti didesnius tvarkaraščius.
- Galimas ilgas tvarkaraščių sudarymas: iš atliktos 4.2.3 skyrelyje atkaitinimo modeliavimo algoritmo analizės žinoma, kad didėjant grafo dydžiui ilgėja ir metodo skaičiavimo trukmė, todėl labai didelių tvarkaraščių generavimas gali užtrukti.

Aplikacijos testavimas

Siekiant atlikti realizuotos aplikacijos testavimą, buvo suvesti testiniai duomenys, o tuomet generuojami įvairaus dydžio tvarkaraščiai: paprastai pridedant nuo 100 iki 200 parinkčių. Be to, stengtasi atkartoti realius atvejus: tvarkaraščiai buvo sudaromi srautais, pavyzdžiui: visų I kurso magistrantūros studijų grupių tvarkaraščiai; visų II kurso bakalauro studijų grupių tvarkaraščiai ir pan. Visais atvejais sugeneruoti tvarkaraščiai, kurie neturėjo nei vieno konflikto. Po atlikto sistemos testavimo galima teigti, kad:

- „aDa tvarkaraščiai“ turi pakankamai paprastą, tačiau tuo pat metu ir patrauklią akiai vartotojo sąsają;
- internetine aplikacija nesunku naudotis: aišku, kam skirtas kiekvienas langas, kaip teisingai suvedami visi reikalingi duomenys, pridedamos parinktys ir sudaromi bei išsaugomi tvarkaraščiai;
- įgyvendinta „aDa tvarkaraščiai“ aplikacija sugeba greitai ir tiksliai (be konfliktų) sudaryti nedidelius tvarkaraščius švietimo įstaigoms. Be to, ši sistema suteikia galimybę generuoti ir didesnės apimties tvarkaraščius, tačiau tai gali pareikalauti ilgesnio laiko;
- norint lengviau dirbti su dideliais duomenų kiekiais, reikalingas aplikacijos vartotojo sąsajos tobulinimas.

IŠVADOS

1. Atliktų kompiuterinių skaitinių eksperimentų rezultatai patvirtino, kad pilno perrinkimo algoritmas tinkamas tik nedideliems grafams – maksimalus grafo dydis, kuriam esant dar pavyko apskaičiuoti chromatinį skaičių per priimtiną laiką (skaičiavimai užtruko vidutiniškai apie 55 minutes) – 9 viršūnių pilnas grafas. Vadinasi, pilnojo perrinkimo metodas nėra tinkamas didelių tvarkaraščių sudarymui.

2. Atlikta algoritmų analizė parodė, kad godusis algoritmas pasižymi sparčia skaičiavimų greitaveika: turint pilnąjį grafą net su 5000 viršūnių rezultatas gaunamas vidutiniškai tik per 3.87 sekundės. Lyginant su kitais euristiniais metodais, pastarasis visais atvejais pasirodė esąs greičiausias, tačiau tuo pat metu ir pateikiantis prasčiausius rezultatus (žr. 18 ir 19 lenteles). Be to, šio metodo skaičiavimo tikslumui nemažą įtaką daro briaunų pasiskirstymas: turint du vienodo dydžio grafus, kai viršūnių ir briaunų skaičius vienodas, tačiau pirmojo briaunų pasiskirstymas tolygus, antrojo – atsitiktinis, chromatinis skaičius galėjo skirtis iki 10 spalvų (žr. 6 ir 7 lenteles). Apibendrinant galima teigti, kad godusis algoritmas tinkamas tada, kada būtina skubiai sudaryti didelį tvarkaraštį (pavyzdžiui: šimtus darbų paskirstyti dešimtims vykdytojų), paaukojant pastarojo tikslumą.

3. Algoritmų palyginamumo tyrimo dėka nustatyta, kad atkaitinimo modeliavimo algoritmas pateikia tiksliausius, artimus optimaliam sprendinius (žr. 18 ir 19 lenteles). Be to, pagal greitaveiką šis metodas buvo greičiausias tarp kitų euristinių metodų, išskyrus godųjį algoritmą (žr. 15 ir 16 paveikslų iliustracijas dešinėje). Vadinasi, kuomet siekiama sudaryti kuo tikslesnį tvarkaraštį, o laiko sąnaudos nėra labai svarbios, t. y. sudarymo trukmė yra antraeilis dalykas, turėtų būti naudojamas atkaitinimo modeliavimo algoritmas.

4. Genetinio algoritmo analizė parodė, kad užtenka sugeneruoti vieną naują kartą, kad būtų surastas optimalus sprendinys pilniems grafams. Be to, nustatyta, kad algoritmo tikslumui rekombinacijos tikimybė p_c poveikio neturi, tačiau daro įtaką algoritmo vykdymo laikui: kuo pasirinkta didesnė p_c reikšmė, tuo skaičiavimų trukmė ilgesnė (žr. 12 paveikslo iliustraciją dešinėje). Genetinis algoritmas daugeliu atvejų sugeba atkartoti atkaitinimo modeliavimo metodu gautą rezultatą, tačiau tai užtrunka žymiai ilgiau (žr. 18 ir 19 lenteles). Apibendrinant galima teigti, kad ir genetinio algoritmo pagalba galima generuoti pakankamai tikslius tvarkaraščius, paaukojant sudarymo laiką.

5. Išanalizavus tvarkaraščių sudarymo algoritmus, nustatyta, kad Tabu paieškos algoritmas tikslumo prasme pasirodė esąs pranašesnis (pateikdavo mažesnį chromatinį skaičių) tik už godųjį algoritmą. Be to, šis metodas daugeliu atvejų buvo lėtesnis už atkaitinimo modeliavimo ir genetinį algoritmus. Tabu paieškos metodas gali būti naudojamas sudėtingesnių tvarkaraščių sudarymui, tačiau yra greitesnių ir tikslesnių metodų.

6. Atliktas realizuotos „aDa tvarkaraščiai“ internetinės aplikacijos, veikiančios atkaitinimo modeliavimo algoritmo pagrindu, testavimas parodė, kad ši sistema sugeba greitai ir tiksliai (be konfliktų) sudaryti nedidelius tvarkaraščius švietimo įstaigoms, kai parinkčių kiekis yra intervale nuo 100 iki 200. Be to, ši sistema suteikia galimybę generuoti ir didesnės apimties tvarkaraščius, tačiau tai gali pareikalauti ilgesnio laiko. Kalbant iš vartotojo sąsajos pusės, tai puikus, aiškus ir lengvai naudojamas įrankis su paprastu, tačiau tuo pat metu ir akį traukiančiu dizainu.

Ateities tyrimų gairės

Be abejo, atlikta analizė yra nedidelis žingsnis nagrinėjant tvarkaraščių sudarymo problemą. Siekiant patobulinti ir praplėsti tyrimą, jis gali būti pratęstas keliomis kryptimis:

1. Pamėginti įtraukti daugiau euristinių metodų: skruzdžių kolonijos algoritmą, spiečių algoritmą ir kt.
2. Pabandyti apjungti kelis euristinius grafų spalvinimo algoritmus, taip sukonstruojant hibridinį metodą, kuris galėtų pagerinti rezultatų kokybę lyginant su turimais.
3. Praplėsti ir pagerinti „aDa tvarkaraščiai“ internetinės aplikacijos funkcionalumą, ištaisant vartotojo interfeiso netobulumas, prijungiant duomenų bazę – kitaip tariant, panaikinti aprašytus 5.3 skyrelyje esamus sistemos trūkumus.

LITERATŪROS ŠALTINIAI

[1] Ayanegui, H., Chavez-Aragon, A. *A complete algorithm to solve the graph coloring problem*, CEUR Workshop Proceedings, **533**, 107–117, 2009, http://ceur-ws.org/Vol-533/09_LANMR09_06.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-07).

[2] Aslan, M., Baykan, N. A. *A Performance Comparison of Graph Coloring Algorithms*, International Journal of Intelligent Systems and Applications in Engineering, 2016, http://dergipark.gov.tr/download/article-file/254140 (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-06).

[3] Belianova, M. A. *Metod imitacii otzhiga i ego primenenie pri reshenii optimizacionnyx zadach*, Molodiozhnyj nauchno-texnicheskij vestnik, 2016, http://sntbul.bmstu.ru/doc/836892.html (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-13).

[4] Bresina, J. *Heuristic-Biased Stochastic Sampling*, Proceedings of the 13th National Conference on Artificial Intelligence, 271–278, 1996, https://www.aaai.org/Papers/AAAI/1996/AAAI96-041.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-28).

[5] Burke, E. K., Elliman, D. G., Weare, R. F. *A University Timetabling System Based on Graph Colouring and Constraint Manipulation*, Journal of Research on Computing in Education, **27**(1), 1–18, 1994, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.2734&rep=rep1&type=pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-28).

[6] Carter, M. W., Laporte, G., Lee, S. Y. *Examination Timetabling: Algorithmic Strategies and Applications*, Journal of the Operational Research Society, **47**(3), 373–383, 1996.

[7] Davis, L. *Order-based genetic algorithms and the graph coloring problem*, Handbook of Genetic Algorithms, 72–90, 1991.

[8] Dobrolowski, T., Dereniowski, D., Kuszner, L. *Koala Graph Coloring Library: An Open Graph Coloring Library for Real World Applications*, Proceedings of the 2008 1st International Conference on Information Technology, IT, Gdansk, Poland, 1–4, 2008, https://www.researchgate.net/publication/4370415_Koala_graph_coloring_library_An_open_graph_coloring_library_for_real-world_applications (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-28).

[9] Douiri, S. M., Elbernoussi, S. *Solving the graph coloring problem via hybrid genetic algorithms*, Journal of King Saud University – Engineering Sciences, **4**, 114–118, 2015, https://ac.els-cdn.com/S1018363913000135/1-s2.0-S1018363913000135-main.pdf?_tid=517be02b-9af2-4360-b836-04c2\220d8778&acdnat=1538931616_2fac2cbf379b81f4649f7bbb2c0d4bd9 (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-07).

[10] Felinskas, G. *Euristinių metodų tyrimas ir taikymas ribotų išteklių tvarkaraščiams optimizuoti*, daktaro disertacija, Vilnius: VDU Matematikos ir informatikos institutas, 2007.

[11] Glover, F. *Tabu search – Part I*, ORSA Journal on Computing, **1**(3), 190–206, 1989, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.4060&rep=rep1&type=pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-18).

[12] Hajduk, J. O. *An analysis of tabu search for the graph coloring problem*, Utrecht University, 2010, www.cs.uu.nl/education/scripties/pdf.php?SID=INF/SCR-2009-095 (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-07).

[13] Hindi, M. M., Yampolskiy, R. V. *Genetic Algorithm Applied to the Graph Coloring Problem*, J. B. Speed School of Engineering, 2011, http://ceur-ws.org/Vol-841/submission_10.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-14).

[14] Johnson, D. S., Aragon, C. R., McGeoch, L. A., Schevon, C. *Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning*, Operations Research, **39**(3), 378–406, 1991, https://pdfs.semanticscholar.org/22ea/d55c99d537754ee377c284319d1f0ce4cdd1.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-28).

[15] Kiaer, L., Yellen, J. *Weighted Graphs and University Timetabling*, Computers and Operations Research, **19**(1), 59–67, 1992.

[16] Kumar, R., Gill, S., Kaushik, A. *A Crossover Probability Distribution in Genetic Algorithm Under process Scheduling Problem*, International Journal of Advanced Research in Computer Science, **1**(4), 505–508, 2010. http://www.ijarcs.info/index.php/Ijarcs/article/viewFile/206/197 (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-12-12).

[17] Lidicky, B. *Graph coloring problems*, Prague, 2011, https://orion.math.iastate.edu/lidicky/pub/thesis.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-06).

[18] Lukasik, S., Kokosinski, Z., Swieton, G. *Parallel Simulated Annealing Algorithm for Graph Coloring Problem*, International Conference on Parallel Processing and Applied Mathematics, 229–238, 2007, http://riad.pk.edu.pl/~zk/pubs/PPAM07_2.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-12-04).

[19] Mahmoudi, S., Lotfi, S. *Modified cuckoo optimization algorithm (MCOA) to solve graph coloring problem*, Applied soft computing, **33**, 48–64, 2015.

[20] Mansuri, A., Gupta, V., Chandel, R. S. *Coloring Programs in Graph Theory*, Int. Journal of Math. Analysis, **4**(50), 2473–2479, 2010, http://www.m-hikari.com/ijma/ijma-2010/ijma-49-52-2010/

mansuriIJMA49-52-2010.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-06).

[21] Marappan, R., Sethumadhavan, G. *A New Genetic Algorithm for Graph Coloring*, Fifth International Conference on Computational Intelligence, Modelling and Simulation, 2013, https://zapdf.com/a-new-genetic-algorithm-for-graph-coloring.html (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-15).

[22] Sakalauskas, G. *Tvarkaraščio realizacija žiniatinklio terpėje taikant grafo viršūnių spalvojimo algoritmus*, baigiamasis bakalauro darbas, Vilnius: VU Matematikos ir informatikos fakultetas, 2011.

[23] Matuliauskas, K. *Grafų teorija*, pakaitų konspektas, 2010, http://kestutis.matuliauskas.lt/K.Matuliauskas_Grafu-teorija_v1.2.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-09-30).

[24] Moss, K. *Coloring problems in graph theory*, Iowa State University, 2017, https://lib.dr.iastate.edu/cgi/viewcontent.cgi?referer=https://www.google.lt/&httpsredir=1&article=6390&context=etd (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-06).

[25] Navakauskaitė, L. *Tvarkaraščio sudarymo uždaviniai*, Vilnius: VU Matematikos ir informatikos fakultetas, 2016.

[26] Porumbel1, D. C., Hao1, J., Kuntz, P. *Position-Guided Tabu Search Algorithm for the Graph Coloring Problem*, Learning and Intelligent Optimization, 148–162, 2009, http://cedric.cnam.fr/~porumbed/papers/paperLion.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-07).

[27] Redl, T. A. *University Timetabling via Graph Coloring: An Alternative Approach*, Congressus Numerantium, **187**, 174–186, 2007, https://pdfs.semanticscholar.org/66d8/a322a70822226b899288af708231378cf432.pdf (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-10-28).

[28] Welsh, D. J. A., Powell, M. B. *An upper bound for the chromatic nmber of a graph and its application to timetabling problems*, The Computer Journal, **10**(1), 85–86, 1967.

[29] Wood, D. C. *A system for computing university examination timetables*, The Computer Journal, **110**(1), 41–47, 1968.

[30] Zacharovas, V. *Grafų teorija*, pakaitų konspektas, 2018, http://www.mif.vu.lt/~vytzach/cgi-bin/sq.php?grafu_teorija (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-09-30).

[31] aSc Timetables, http://www.ibn.lt/asc-tvarkarasciai-kompiuterine-tvarkarasciu-sudarymo-programa (puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-12-20).

[32] Mimosa Software,
http://www.mimosasoftware.com/help/mimosa_help_overview.htm
(puslapio adresas gali būti keičiamas, paskutinį kartą tikrintas 2018-12-20).

PRIEDAI

Dokumentą sudaro du priedai. A priede pateiktas programos kodo fragmento listing'as, kuriame yra realizuoti: pilno perrinkimo algoritmas, godusis algoritmas, atkaitinimo modeliavimo algoritmas, genetinis algoritmas ir Tabu paieškos algoritmas. B priede parodyti internetinės „aDa tvarkaraščiai“ aplikacijos pagrindiniai langai.

A. Programos kodo fragmento listing'as

```
def brute_force_algorithm(g):
    """
    Graph coloring with brute force algorithm

    :param g:
        Graph
    :return:
        Nodes with color and chromatic number
    """
    nodes_count = len(list(g.nodes()))
    chr_number = nodes_count
    colorized_nodes = dict()
    final_col_nodes = dict()

    # Count of all color combinations
    color_comb_count = nodes_count**nodes_count

    for comb in range(color_comb_count):
        correct_comb = True
        color_comb = get_color_combination(comb, nodes_count,
            color_comb_count)
        color_comb = tuple(color_comb)

        for index, color in enumerate(color_comb):
            colorized_nodes[index] = color

        # Iteration through all nodes
        for node in g.nodes():
            incorrect_comb = False

            # Iteration through all node's neighbors
            for neighbor in g.neighbors(node):
                if colorized_nodes[node] == colorized_nodes[neighbor]:
                    incorrect_comb = True
                    break

            if incorrect_comb:
                correct_comb = False
                break

        if correct_comb:
            if not final_col_nodes or chr_number > len(list(set(
                colorized_nodes.values()))):
                final_col_nodes = dict(colorized_nodes)
                chr_number = len(list(set(colorized_nodes.values())))

    return final_col_nodes, chr_number


def greedy_algorithm(g):
    """
    Graph coloring with greedy algorithm
```

```
    :param g:
        Graph
    :return:
        Nodes with color and chromatic number
    """
    nodes = dict()

    for node in g.nodes():
        nodes[node] = len(g[node])

    # Nodes sorting by neighbors count
    sorted_nodes = list(dict(sorted(nodes.items(), key=lambda value:
        value[1], reverse=True)).keys())
    colorized_nodes = dict()
    colorized_nodes[sorted_nodes[0]] = 0

    # Set colors to nodes
    for node in sorted_nodes[1:]:
        available_colors = [True] * len(sorted_nodes)

        for neighbor in g.neighbors(node):

            # Color is marked as unavailable if it has already been
                assigned
            if neighbor in colorized_nodes.keys():
                color = colorized_nodes[neighbor]
                available_colors[color] = False

        for color in range(len(available)):
            if available_colors[color]:
                colorized_nodes[node] = color
                break

    colors = list(set(colorized_nodes.values()))
    chr_number = len(colors)

    return colorized_nodes, chr_number


def simulated_annealing_algorithm(g, colors_count, initial_temp,
    min_temp, alpha):
    """
    Graph coloring with simulated annealing algorithm

    :param g:
        Graph
    :param colors_count:
        Number of colors for graph coloring
    :param initial_temp:
        Initial temperature
    :param min_temp:
        Minimal temperature
    :param alpha:
        Speed coefficient of temperature cooling
```

```
    :return:
        Nodes with color and chromatic number
    """
    colors = list(range(colors_count))
    nodes_count = len(list(g.nodes()))

    # Generation of random graph coloring
    colorized_nodes = dict()

    for i in range(nodes_count):
        colorized_nodes[i] = colors[randrange(0, len(colors))]

    temperature = initial_temp
    fitness = 0

    while True:
        conflicted_nodes = set()
        no_conflicts = True

        # Finding of conflicting nodes
        for i in range(nodes_count):
            if i in conflicted_nodes:
                continue

            for neighbor in g.neighbors(i):
                if colorized_nodes[i] == colorized_nodes[neighbor]:
                    conflicted_nodes.add(i)
                    conflicted_nodes.add(neighbor)
                    no_conflicts = False

            if not no_conflicts:
                break

        # Stop algorithm if valid graph coloring is found
        if no_conflicts is True:
            break

        # Random choosing of conflicted node
        conflicted_nodes = list(conflicted_nodes)
        conflicted_node = conflicted_nodes[randrange(0, len(
            conflicted_nodes))]

        # New graph coloring
        new_colorized_nodes = colorized_nodes.copy()

        # Add new color for chosen conflicted node
        valid_colors = set(colors)

        for n in g.neighbors(conflicted_node):
            if new_colorized_nodes[n] in valid_colors:
                valid_colors.remove(new_colorized_nodes[n])

        valid_colors = list(valid_colors)

        if len(valid_colors) > 0:
```

```
                new_colorized_nodes[conflicted_node] = valid_colors[
                    randrange(len(valid_colors))]
            else:
                new_color = colors[randrange(0, len(colors) - 1)]

                if new_colorized_nodes[conflicted_node] == new_color:
                    new_color = colors[-1]

                new_colorized_nodes[conflicted_node] = new_color

            # Fitness of graph coloring
            fitness = calculate_fitness(g, nodes_count, colorized_nodes,
                fitness)
            new_fitness = calculate_fitness(g, nodes_count,
                new_colorized_nodes)

            if new_fitness > fitness:
                colorized_nodes = new_colorized_nodes
                fitness = new_fitness
            else:
                p = random()
                diff_fitness = fitness - new_fitness

                # Metropolis condition
                if p < exp(-diff_fitness / temperature):
                    colorized_nodes = new_colorized_nodes
                    fitness = new_fitness

            # Cooling of temperature
            temperature *= alpha

            if temperature < min_temp:
                break

    if no_conflicts is False:
        return None
    else:
        colors = list(set(colorized_nodes.values()))
        chr_number = len(colors)
        return colorized_nodes, chr_number


def genetic_algorithm(g, colors_count, max_generations, population_size)
    :
    """
    Graph coloring with genetic algorithm

    :param g:
        Graph
    :param colors_count:
        Number of colors for graph coloring
    :param max_generations:
        Maximum number of generations
    :param population_size:
        Size of population
```

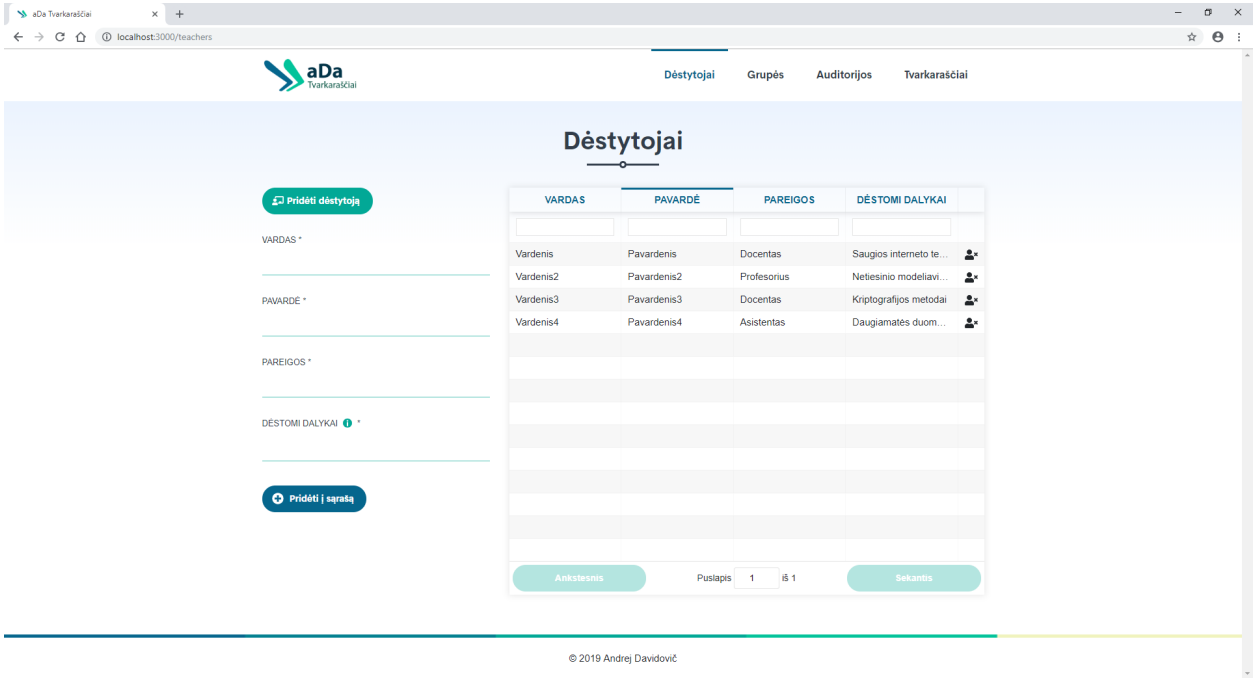
```
    :return:
        Nodes with color and chromatic number
    """
    colors = list(range(colors_count))
    nodes_count = len(list(g.nodes()))
    population = []
    fitness = []
    generation = 0

    # Generation of population
    for i in range(population_size):
        chromosome = dict()

        for j in range(nodes_count):
            chromosome[j] = colors[randrange(len(colors))]

        # Calculate chromosome fitness
        chromosome_fitness = calculate_chromosome_fitness(g, nodes_count
            , chromosome)

        population.append(chromosome)
        fitness.append(chromosome_fitness)

    sort_population(population, fitness, population_size)

    while fitness[0] != 0 and generation < max_generations:
        half_population_size = int(population_size / 2)
        children = []
        new_fitness = []

        for i in range(half_population_size):
            parents = get_parents(population, fitness, population_size)

            p_cr = random()
            if p_cr >= p_c:
                child = crossover(parents, nodes_count)
            else:
                child = population[i + half_population_size]

            p_mr = random()
            if p_mr >= p_m:
                mutate(child, g, nodes_count, colors)

            # Calculate new chromosome fitness
            new_chromosome_fitness = calculate_chromosome_fitness(g,
                nodes_count, child)

            children.append(child)
            new_fitness.append(new_chromosome_fitness)

        for i in range(half_population_size):
            population[i + half_population_size] = children[i]
            fitness[i + half_population_size] = new_fitness[i]

        sort_population(population, fitness, population_size)
```

```
        generation += 1

    print('Generations: {}'.format(generation))

    if fitness[0] != 0:
        return None
    else:
        colors = list(set(population[0].values()))
        chr_number = len(colors)
        return population[0], chr_number


def tabu_algorithm(g, colors_count, repeats, max_iteration):
    """
    Graph coloring with tabu search algorithm

    :param g:
        Graph
    :param colors_count:
        Number of colors for graph coloring
    :param repeats:
        Number of repeats for improvements
    :param max_iterations:
        Iterations limit
    :return:
        Nodes with color and chromatic number
    """
    colors = list(range(colors_count))
    nodes_count = len(list(g.nodes()))
    tabu_size = nodes_count

   # Generation of random graph coloring
    colorized_nodes = dict()

    for i in range(nodes_count):
        colorized_nodes[i] = colors[randrange(0, len(colors))]

    tabu = deque()
    intention = dict()
    iterations = 0
    conflicts_count = None
    conflicted_nodes = set()

    while iterations < max_iterations:
        if len(conflicted_nodes) == 0:

            # Finding of conflicting nodes
            for i in range(nodes_count):
                if i in conflicted_nodes:
                    continue

                for neighbor in g.neighbors(i):
                    if colorized_nodes[i] == colorized_nodes[neighbor]:
                        conflicted_nodes.add(i)
```

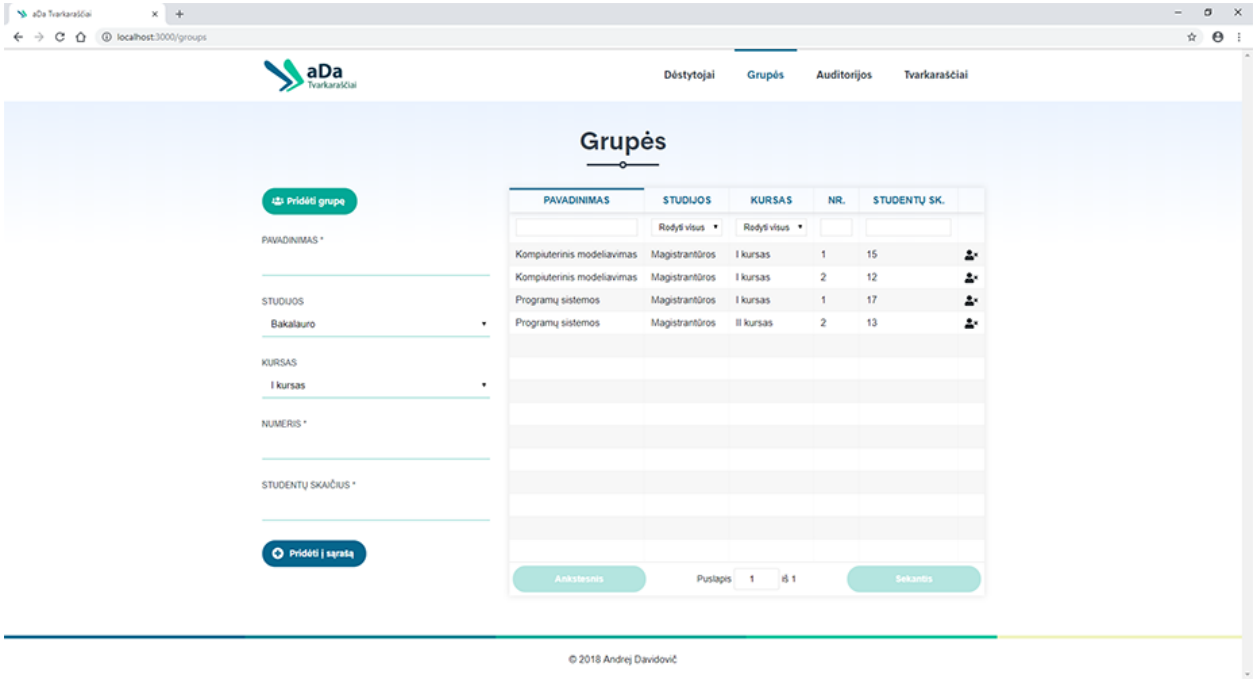
```
                conflicted_nodes.add(neighbor)

        conflicted_nodes_list = list(conflicted_nodes)
        conflicts_count = len(conflicted_nodes_list)

        # Stop algorithm if valid graph coloring is found
        if conflicts_count == 0:
            break

        conflicted_node = None
        new_colorized_nodes = None
        new_conflicted_nodes = None

        for r in range(repeats):

            # Random choosing of conflicted node
            conflicted_node = conflicted_nodes_list[randrange(0, len(
                conflicted_nodes_list))]

            # New graph coloring
            new_colorized_nodes = colorized_nodes.copy()

            # Add new color for chosen conflicted node
            valid_colors = set(colors)

            for n in g.neighbors(conflicted_node):
                if new_colorized_nodes[n] in valid_colors:
                    valid_colors.remove(new_colorized_nodes[n])

            valid_colors = list(valid_colors)

            if len(valid_colors) > 0:
                new_color = valid_colors[randrange(len(valid_colors))]
            else:
                new_color = colors[randrange(0, len(colors) - 1)]

                if new_colorized_nodes[conflicted_node] == new_color:
                    new_color = colors[-1]

            new_colorized_nodes[conflicted_node] = new_color
            new_conflicted_nodes = set()

            # Finding of new conflicting nodes
            for i in range(nodes_count):
                if i in new_conflicted_nodes:
                    continue

                for neighbor in g.neighbors(i):
                    if new_colorized_nodes[i] == new_colorized_nodes[
                        neighbor]:
                        new_conflicted_nodes.add(i)
                        new_conflicted_nodes.add(neighbor)

            new_conflicts_count = len(list(new_conflicted_nodes))
```

```
            # Check if found better coloring of graph
            if new_conflicts_count < conflicts_count:
                if new_conflicts_count <= intention.setdefault(
                    conflicts_count, conflicts_count - 1):
                    intention[conflicts_count] = new_conflicts_count - 1

                    if (conflicted_node, new_color) in tabu:
                        tabu.remove((conflicted_node, new_color))
                else:
                    if (conflicted_node, new_color) in tabu:
                        continue
                break

        tabu.append((conflicted_node, colorized_nodes[conflicted_node]))
        colorized_nodes = new_colorized_nodes.copy()
        conflicted_nodes = new_conflicted_nodes.copy()

        if len(tabu) > tabu_size:
            tabu.popleft()

        iterations += 1

    if conflicts_count != 0:
        return None
    else:
        colors = list(set(colorized_nodes.values()))
        chr_number = len(colors)
        return colorized_nodes, chr_number
```

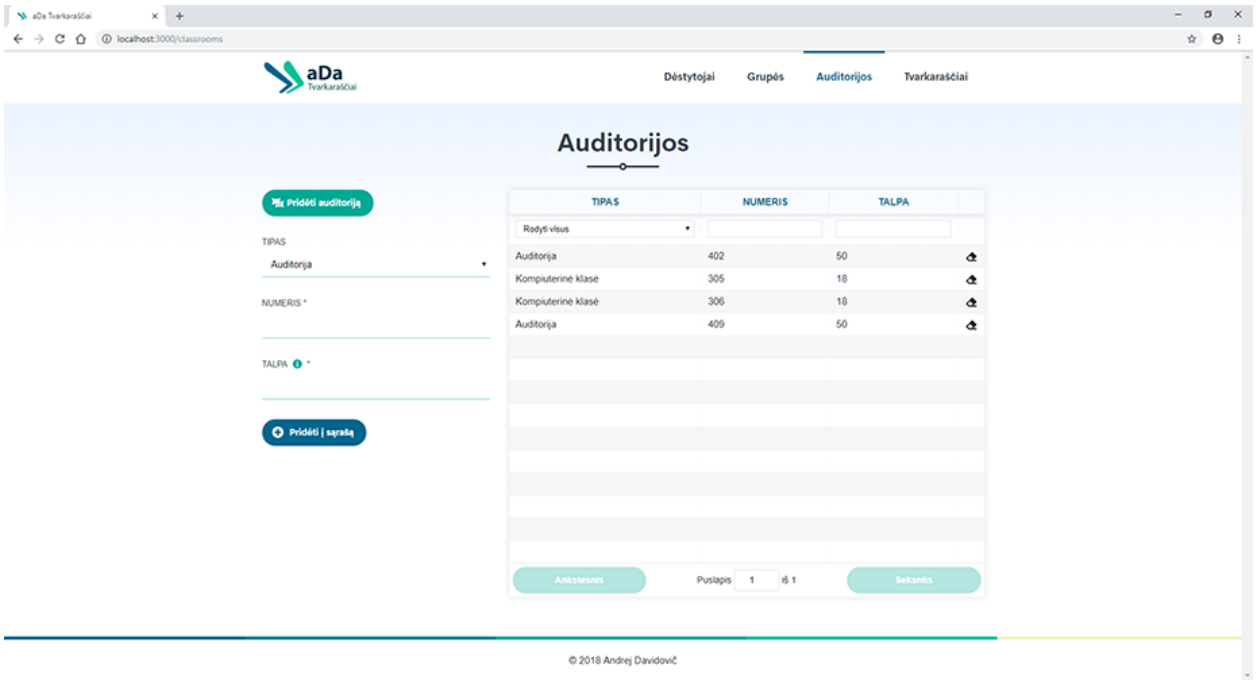
B. Internetinės aplikacijos pagrindiai langai



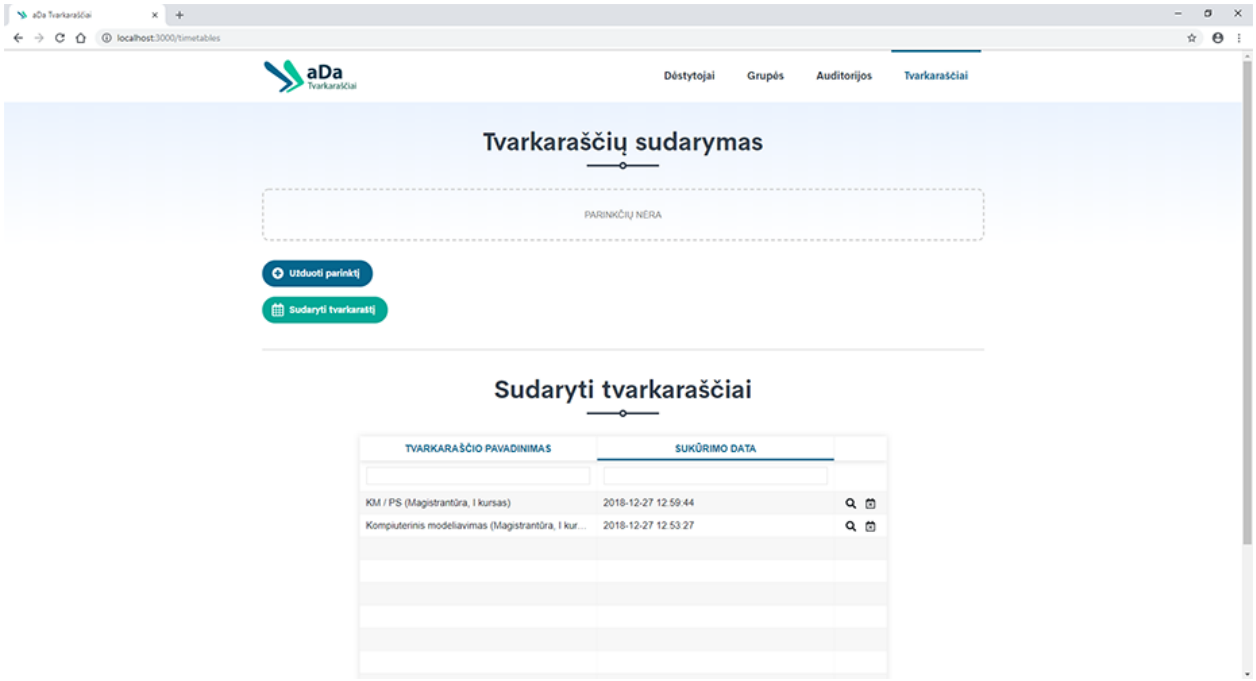
22 pav. „aDa tvarkaraščiai“ aplikacijos „Dėstytojai“ langas.



23 pav. „aDa tvarkaraščiai“ aplikacijos „Grupės“ langas.



24 pav. „aDa tvarkaraščiai“ aplikacijos „Auditorijos“ langas.



25 pav. „aDa tvarkaraščiai“ aplikacijos „Tvarkaraščiai“ langas.