KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Dovydas Šopa

DVIMAČIŲ IŠDĖLIOJIMO UŽDAVINIŲ SPRENDIMAS NAUDOJANT LYGIAGREČIUS SKAIČIAVIMUS GRAFINIAME PROCESORIUJE

Baigiamasis magistro projektas

Vadovas
Doc. dr. T. Blažauskas

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

DVIMAČIŲ IŠDĖLIOJIMO UŽDAVINIŲ SPRENDIMAS NAUDOJANT LYGIAGREČIUS SKAIČIAVIMUS GRAFINIAME PROCESORIUJE

Baigiamasis magistro projektas

Programų sistemų inžinerija (kodas 621E16001)

Vadovas
(parašas) Doc. dr. T. Blažauskas
(data)

Recenzentas
(parašas) Doc. dr. A. Ostreika
(data)

Projektą atliko
(parašas) Dovydas Šopa
(data)

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Informatikos fakultetas

(Fakultetas)

Dovydas Šopa

(Studento vardas, pavardė)

Programų sistemų inžinerija, 621E16001

(Studijų programos pavadinimas, kodas)

„Dvimačių išdėliojimo uždavinių sprendimas naudojant lygiagrečius skaičiavimus grafiniame procesoriuje“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 16 m. gegužės d.

Kaunas

Patvirtinu, kad mano, **Dovydo Šopos**, baigiamasis projektas tema „Dvimačių išdėliojimo uždavinių sprendimas naudojant lygiagrečius skaičiavimus grafiniame procesoriuje“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

TURINYS

Šopa, Dovydas. Dvimačių išdėliojimo uždavinių sprendimas naudojant lygiagrečius skaičiavimus grafiniame procesoriuje. *Magistro* baigiamasis projektas / vadovas doc. dr. Tomas Blažauskas; Kauno technologijos universitetas, Informatikos fakultetas.

Mokslo kryptis ir sritis: programų sistemų inžinerija, objektų skaičiavimų teorija, dėliojimo uždavinys, programinės įrangos programavimas.

Reikšminiai žodžiai: *objektų išdėliojimo uždavinys, grafinis procesorius, CUDA, lygiagretūs skaičiavimai.*

Kaunas, 2016. 49 p.

SANTRAUKA

Objektų išdėliojimas yra klasikinis optimizavimo uždavinys. Šis uždavinys įdomus ne tik teoriškai, bet turi ir daug praktinių pritaikymų: straipsnių išdėliojimas laikraštyje taip, kad reikėtų kuo mažiau lapų; detalių pjaustymas lakštuose taip, kad liktų kuo mažiau nepanaudotų medžiagų.

„Nvidia“ ištobulinta CUDA technologija leidžia atlikti didelį kiekį skaičiavimų naudojant grafinį procesorių. Grafinis procesorius pritaikytas atlikti tuos pačius veiksmus su skirtingais duomenimis daug kartų, o objektų išdėliojimo uždavinyje būtent tai ir daroma.

Šiame darbe nagrinėjamas dvimatis objektų išdėliojimo atvejis, kai objektai yra orientuoti stačiakampiai, objektų sąrašas žinomas iš anksto, o lakštai yra vienodo dydžio. Taip pat bus panaudojamos CUDA suteikiamos galimybės šiam uždaviniui spręsti lygiagrečiai. Pateikiamas evoliucinis euristinis modifikavimo tipo algoritmas aprašytam uždaviniui spręsti, bei pasiūlytą algoritmą realizuojančios sistemos aprašymas. Galiausiai pateikiamas siūlomo algoritmo greitaveiką ir tikslumą vertinantis eksperimentas bei gauti rezultatai.

Šopa, Dovydas. *Solving 2D Bin Packing Problem Using Graphics Processing Unit*: *Master‘s* thesis in software engineering / supervisor assoc. prof. Tomas Blažauskas. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: computer science, software engineering, theory of computation, bin packing problem, programming systems.

Key words: bin packing problem, graphics processing unit, CUDA, parallel calculations.

Kaunas, 2016. 49 p.

SUMMARY

Bin packing problem is classic optimization problem. This problem is interesting not only theoretically, but also has many practical applications. F. e. placing articles in newspaper so that the amount of papers would be minimal. Or putting boxes in trucks so that the amount of trucks would be minimal.

“Nvidia” perfected their CUDA technology which allows to perform a large amount of computation using graphics processing unit. Graphics processing unit is adapted to perform the same actions with different data many times, and that is exactly what bin packing problem is all about.

This work is focused on solving two-dimensional bin packing problem where objects cannot be rotated, objects are rectangular shape, objects are given offline and all bins are same dimensions. CUDA will also be used to address this challenge in parallel. An evolutionary heuristic algorithm which applies modifications is presented as well. Finally, experiment for measuring the speed and accuracy of the proposed algorithm are presented, as well as the results obtained.

1. ĮŽANGA

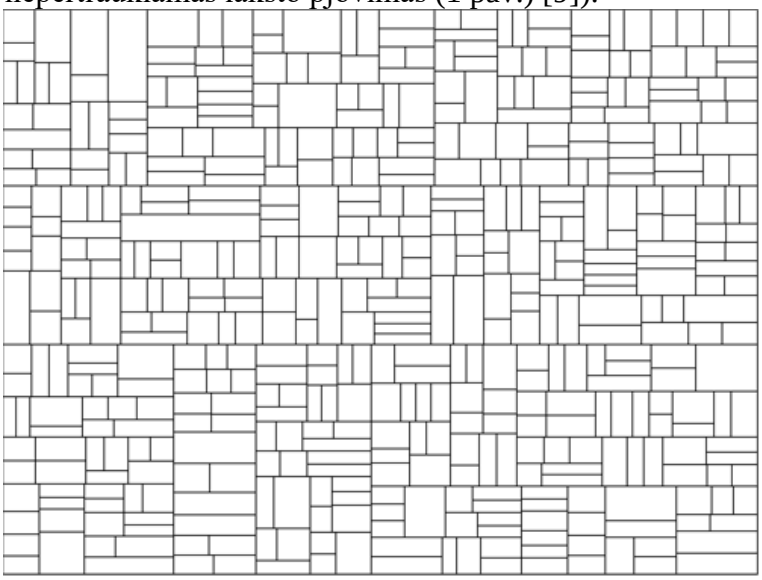
Objektų išdėliojimas (angl. *bin packing problem*) yra klasikinis optimizavimo uždavinys, kuris pirmą kartą suformuluotas 1939 m. [1]. Šis uždavinys įdomus ne tik teoriškai, bet turi ir daug praktinių pritaikymų: straipsnių išdėliojimas laikraštyje taip, kad reikėtų kuo mažiau lapų; detalių pjaustymas lakštuose taip, kad liktų kuo mažiau nepanaudotų medžiagų; dėžių išdėliojimas sunkvežimiuose taip, kad reikėtų kuo mažiau sunkvežimių.

Gerai žinoma, kad tai nedeterministiškai polinomiškai (toliau – NP) sunkus uždavinys [2]. Pirmiausia tokie uždaviniai pradėti nagrinėti kaip vienmačiai, o po to buvo praplėsti iki daugiamačių (dD, kur $1 \leq d$). Klasikinė šio uždavinio formuluotė skamba taip: visos dėžutės turi ilgius 1, o objektų ilgiai yra intervale (0; 1] kiekvienai dimensijai. Objektus į dėžutes reikia sudėlioti taip, kad būtų panaudota kuo mažiau dėžučių. Dvimačiu atveju matematinis šio uždavinio apibrėžimas skamba taip: numatytoje dvimatėje koordinačių erdvėje A, kurios dydis R^2, reikia surasti tokią stačiakampių aibę V, kad bet kuriai stačiakampių porai (v; v') iš aibės V sankirtų $v \cap v'$ aibė būtų tuščia.

NP uždaviniai pasižymi tuo, kad jų sprendimui reikia didesnio nei polinominio laiko. Vadinasi, šio uždavinio sprendimo trukmės apatinė riba (angl. *lower bound*) yra $\omega(n^k)$, kur n – įvesties parametras, susijęs su duomenų kiekiu; k – bet koks skaičius.

Šis uždavinys turi daug variantų:

1. Objektai gaunami po vieną, ir visas jų sąrašas nėra žinomas iš anksto (angl. *online*) [3] arba iš anksto žinomas visas objektų sąrašas (angl. *offline*) [4];
2. Objektai gali būti sukinėjami [5] arba yra orientuoti ir negali būti sukinėjami [3];
3. Objektų forma gali būti bet kokia [6] arba apribota (pvz., stačiakampiai [4], kvadratai [7]);
4. Dėžutės, į kurias dedami objektai, gali būti vienodo [3] arba skirtingo dydžio [8];
5. Gali būti papildomų apribojimų objektų dėliojimui (pvz., sprendžiant dvimatį uždavinį gali būti reikalingas nepertraukiamas lakšto pjovimas (1 pav.) [9]).



1 pav. Dvimatis objektų išdėliojimas, kai reikalingas nepertraukiamas pjovimas [10]

Žinoma, galimos įvairios išvardytų uždavinio variantų kombinacijos.

1.1. Dokumento paskirtis

Šiame darbe nagrinėjamas dvimatis objektų išdėliojimo atvejis, kai objektai yra orientuoti stačiakampiai, negali būti vartomi, objektų sąrašas žinomas iš anksto ir lakštai yra vienodo dydžio. Taip pat bus panaudojamos CUDA (angl. *Compute Unified Device Architecture*) suteikiamos galimybės šiam uždaviniui spręsti lygiagrečiai.

Antrame šio dokumento skyriuje pateikiama dvimačių išdėliojimo algoritmų analizė, kurioje apžvelgiami šiuo metu egzistuojantys algoritmai, jų privalumai ir trūkumai. Trečiame skyriuje nurodomi realizuotos sistemos projekto esminiai aspektai. Ketvirtame skyriuje pateikiamas siūlomas evoliucinis euristinis modifikavimo tipo algoritmas, aprašoma, kaip algoritmas yra pritaikomas

ilgiausiai trunkančius skaičiavimus atlikti grafiniame procesoriuje (toliau – GPU). Taip pat šiame skyriuje pateikiamas sistemos realizacijos kokybės įvertinimas. Penktame šio dokumento skyriuje aprašomas atliekamas eksperimentinis tyrimas ir pateikiami jo rezultatai. Tai pat aptariamos tolesnės sistemos tobulinimo galimybės. Galiausiai, pateikiamos darbą apibendrinančio išvados.

1.2. Darbo tikslas

Suformuluoti dvimatį objektų išdėliojimo uždavinį sprendžiantį evoliucinį euristinį algoritmą, kuris išnaudotų CUDA suteikiamas lygiagretumo galimybes. Atlikti realizuoto algoritmo tyrimą ir palyginti jo gaunamus rezultatus su kitais evoliuciniais euristiniais algoritmais.

1.3. Mokslinis naujumas

1. Pasiūlytas evoliucinis euristinis modifikavimo tipo algoritmas, kuris pritaikytas skaičiavimams CUDA.

1.4. Uždaviniai

1. Išanalizuoti egzistuojančius objektų išdėliojimo uždavinį sprendžiančius algoritmus.
2. Suformuoti evoliucinį euristinį modifikavimo tipo algoritmą, kuris būtų pritaikytas skaičiavimams GPU.
3. Suprojektuoti ir sukurti siūlomą algoritmą realizuojančią sistemą.
4. Atlikti realizuotos sistemos tyrimą ir palyginti gautus rezultatus su jau egzistuojančių evoliucinių euristinių algoritmų gautais rezultatais.

2. ANALITINĖ DALIS

Kadangi iki šiol nerastas optimalus NP sunkių uždavinių sprendimas, mokslininkai ieško vis geresnio objektų išdėliojimo uždavinio sprendinio per kuo trumpesnį laiką. Šiame skyriuje apžvelgsime keleto tipų siūlomus algoritmus:

1. Tikslieji algoritmai. Jų paskirtis – surasti tikslų uždavinio sprendinį. Šio tipo algoritmai gauna tikslų uždavinio atsakymą, tačiau tam sugaištama daug laiko.
2. Euristiniai algoritmai. Tai algoritmai, kurie grąžina daugeliu atvejų patenkinamą sprendinį. Tačiau šio tipo algoritmai negali garantuoti, kad visuomet ras gerą sprendinį ar kad tai padarys greitai.
3. Evoliuciniai euristiniai algoritmai. Tai algoritmai, kurie operuoja žemesnės eilės euristiniais algoritmais. Evoliucinių euristinių algoritmų tikslas – kiekviename žingsnyje pasirinkti labiausiai tinkantį euristinį algoritmą.
4. Lygiagretiems skaičiavimams pritaikyti algoritmai. Paprastai pasirenkamas vienas iš anksčiau paminėtų algoritmų tipų ir pritaikomas skaičiavimams lygiagrečiai.

2.1. Tikslieji algoritmai

Šiame skyrelyje apžvelgsime kelis tiksliuosius algoritmus. Šio tipo algoritmai padeda rasti tikslų uždavinio atsakymą, tačiau tam sugaištama daug laiko. Apžvelgsime objektų išdėliojimo uždavinio performulavimą į tinklinį grafą ir lakšto užpildymo algoritmą.

2.1.1. Uždavinio performulavimas į tinklinį grafą

A. Quilliotas ir H. Toussaintas [4] pasiūlė dvimatį išdėliojimo uždavinį performuluoti į uždavinį su tinkliniu grafu. Visas jų darbas yra orientuotas į matematinius įrodymus, kaip iš pradinės užduoties sudaryti tinklinį grafą be ciklų. Performulavę uždavinį kaip tinklinį grafą, jie pasiūlo paprastą sprendimą, kurio esmė – tinklinio grafo formavimas taip, kad susidarytų kuo trumpesnis kritinis kelias. Šis algoritmas neduoda iki šiol žinomų geriausių rezultatų, tačiau autoriai to ir nesiekia. Jų tikslas – uždavinio performulavimas. Uždavinys virsta matematiniu grafų teorijos uždaviniu. Tolimesniuose darbuose A. Quilliotas ir H. Toussaintas žada ieškoti būdų, kaip pasiekti optimalų sprendinį, naudojant tokį uždavinio formulavimą.

2.1.2. Lakšto užpildymo algoritmas

R. E. Korf [11] pasiūlė pilno lakštų užpildymo algoritmą. Pirmiausia, jis sudėlioja visus objektus į lakštus naudodamas geriausiai tinkančio objekto mažėjimo tvarka (angl. *best fit decreasing*, toliau – BFD) algoritmą. Tuomet tikrinama, į kiek lakštų objektai turėjo tilpti. Jei rastas sprendinys sutampa su optimaliu, tuomet BFD rastas sprendinys grąžinamas kaip uždavinio sprendinys. Jei sprendinys nėra optimalus, tuomet BFD rastą sprendinį bandoma pagerinti naudojant šakų ir ribų (angl. *branch and bound*, toliau – BB) algoritmą.

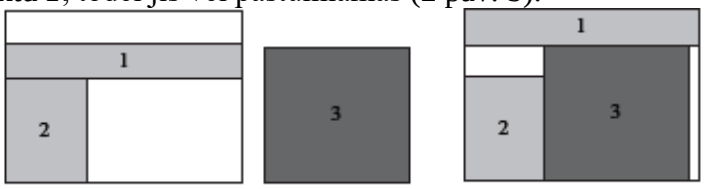
2.2. Euristiniai algoritmai

Šiame skyrelyje apžvelgsime keletą euristinių algoritmų. Šio tipo algoritmai grąžina daugeliu atvejų patenkinamą sprendinį, tačiau negali garantuoti, kad visuomet ras gerą sprendinį ar kad tai padarys greitai. Apžvelgsime suspaudimo, nuoseklų, pasikartojimais grįstą, lokalios paieškos ir konstrukcinį algoritmus.

2.2.1. Suspaudimo algoritmas

Kai kurie autoriai (Z. Zhang, S. Guo, W. Zhu, W. C. Oon ir A. Lim [12]) objektų išdėliojimo uždavinį pasiūlė spręsti minimizuojant nepanaudotą vietą. Nepanaudotos vietos minimizavimas automatiškai mažina reikiamų lakštų kiekį, o tai ir yra pagrindinis uždavinio tikslas. Minimizavimas vyksta stengiantis suspausti objektus kaip galima labiau vienas prie kito. Algoritmo pavyzdys pateiktas 2 pav. Objektas *1* dedamas į apatinį kairį kampą. Šis objektas sukuria 3 papildomus taškus (objekto viršūnės) naujo objekto įterpimui. Objektas *2* dėjimui jau turi 4 vietas. Pirmiausia vėl bandoma dėti į pirmąjį tašką (apatinį kairįjį kampą). Objektas *2* persidengia tik su objektu *1*, todėl šį bandoma pastumti. Pastūmus objektą *1*, gaunama 2 pav. a dalyje pateikta situacija. Naujo objekto įdėjimui jau

yra 7 taškai. Tuomet bandoma įdėti objektą *3*. Vėl bandoma dėti į apatinį kairįjį kampą. Objektas *3* persidengia su dviem objektais, todėl bandoma dėti į antrąjį tašką. Objektas *3* antrame taške persidengia tik su objektu *1*, todėl jis vėl pastumiamas (2 pav. b).

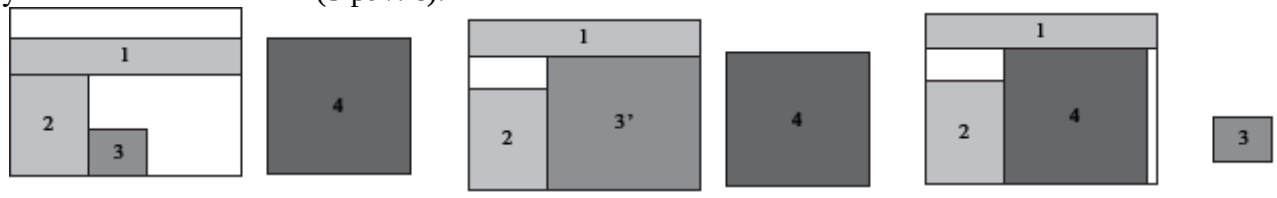


(a) Usable space is fragmented, item 3 cannot be packed

(b) Item 1 is pushed upwards to make room for item 3

2 pav. Elementų suspaudimas [12]

Taip pat atkreipiamas dėmesys į objektų keitimą vietomis (3 pav.). Jei bandoma įdėti objektą, tačiau jis nebetelpa (3 pav. a), tuomet tikrinama ar būtų galima šį objektą sukeisti su jau įdėtu mažesniu objektu. Tam kiekvieną mažesnį objektą, pradedant mažiausiu (3 pav. objektas *3*), bandoma praplėsti (3 pav. b). Jei gautas plotas didesnis arba lygus norimo įdėti objekto plotui, tuomet objektai yra sukeičiami vietomis (3 pav. c).



(a) Item 4 cannot be loaded

(b) Inflating item 3

(c) Replacing item 3 by item 4

3 pav. Elementų keitimas vietomis [12]

Šis algoritmas imituoja normalų žmogaus elgesį: žmonės, siekdami padaryti vietos kitiems objektams, stumdo juos, jei yra galimybė, keičia mažus objektus didesniais. Šio algoritmo veikimo laikas yra $O(n^2)$.

2.2.2. Nuoseklus algoritmas

Autoriai (Y. P. Cui, Y. Cui ir T. Tang [9]) pateikia nuoseklų euristinį algoritmą dvimačio išdėliojimo uždavinio sprendimui. Algoritmas generuoja iš anksto apibrėžtą išdėstymo variantų kiekį, kiekvieną kartą objektus vertindamas skirtingomis vertėmis. Iš pradžių kiekvieno objekto vertė yra prilyginama jo plotui. Po kiekvieno plano sudarymo, objektų vertės keičiamos naujomis, atsižvelgiant į objekto dydį ir medžiagos panaudojimą plane. Objekto vertė didinama, jei jis sunkiai įpakuojamas arba blogai dera dabartiniame plane. Geriausias planas pasirenkamas kaip uždavinio sprendinys.

2.2.3. Pasikartojimais grįstas algoritmas

F. Brandão ir J. P. Pedroso [13] patobulina standartinius pirmo tinkančio mažėjimo tvarka (angl *first fit decreasing*, toliau – FFD) ir BFD algoritmus.

FFD algoritmas formuluojamas taip (4 pav.):

1. Objektai surikiuojami ploto mažėjimo tvarka.
2. Objektai į lakštus dedami po vieną, pradedant didžiausiu. Kiekvienam objektui ieškomas pirmas lakštas, į kurį jis tiktų.
3. Jei toks lakštas randamas jau panaudotų lakštų sąraše, tuomet objektas dedamas į šį lakštą, ir likęs laisvas lakšto plotas sumažinamas. Priešingu atveju yra imamas naujas lakštas.

```
input: m – number of different lengths; l – set of lengths; b – demand for each length; L - roll length
output: R – number of rolls needed; Sol – list of assignments
function FFD(m, l, b, L):
    l ← reverse(sort(l));    //sort lengths in decreasing order
    Sol ← [ ];
    R ← 1;
    Rem ← [L];
    for i ← 1 to m do    //for each length
        k' ← 1;
        for j ← 1 to b_{l_i} do    //for each piece of length l_i
            assigned ← False;
            for k ← k' to R do    //try each roll
                if Rem[k] > l_i then    //if there is enough space
                    Rem[k] ← Rem[k] − l_i;
                    Sol.append(l_i → roll k);
                    assigned ← True;
                    k' ← k;
                    break;
            if not assigned then    //if the piece was not assigned to any roll
                R ← R + 1;
                k' ← R;
                Rem.append(L − l_i);
                Sol.append(l_i → roll R);
    return (R, Sol);
```

4 pav. Pirmo tinkančio objekto mažėjimo seka algoritmas [13]

BFD algoritmas formuluojamas taip (5 pav.):

1. Objektai surikiuojami ploto mažėjimo tvarka.
2. Objektai į lakštus dedami po vieną, pradedant didžiausiu. Kiekvienam objektui ieškomas lakštas su mažiausiu laisvu plotu, į kurį objektas tilptų.
3. Jei toks lakštas randamas jau panaudotų lakštų sąraše, tuomet objektas dedamas į šį lakštą, o likęs laisvas lakšto plotas sumažinamas. Priešingu atveju imamas naujas lakštas.

```
input: m – number of different lengths; l – set of lengths; b – demand for each length; L – roll length
output: R – number of rolls needed; Sol – list of assignments
function BFD(m, l, b, L):
    l ← reverse(sort(l));    //sort lengths in decreasing order
    Sol ← [ ];
    R ← 1;
    Rem ← [L];
    for i ← 1 to m do    //for each length
        for j ← 1 to b_{l_i} do    //for each piece of length l_i
            S ← {k | 1 ≤ k ≤ R, Rem[k] ≥ l_i};
            if S ≠ ∅ then
                best ← argmin_{k ∈ S} Rem[k];
                Rem[best] ← Rem[best] − l_i;
                Sol.append(l_i → roll best);
            else    //if there is no roll with enough available space
                R ← R + 1;
                Rem.append(L − l_i);
                Sol.append(l_i → roll R);
    return (R, Sol);
```

5 pav. Geriausiai tinkančio objekto mažėjimo seka algoritmas [13]

Autoriai nagrinėja atvejį, kai objektų yra labai daug, tačiau mažas unikalių objektų kiekis. Jų dėliojimas paremtas pasikartojimais, t. y. jei yra daug vienodų objektų ir vienas lakštas jau suformuotas optimaliai (turint nedidelį kiekį skirtingų objektų tai lengva įgyvendinti), tuomet kitų lakštų nebereikia formuoti iš naujo. Su likusiais nepanaudotais objektais vėl vykdomas FFD arba BFD algoritmas. Žinoma, toks sprendimo būdas yra tinkamas tik turint nedidelį unikalių objektų kiekį ir vienodo dydžio lakštus.

2.2.4. Lokalios paieškos algoritmas

J. Bang-Jensenas ir R. Larsenas [8] sprendžia dvimačio objektų išdėliojimo uždavinio atvejį, kai lakštai yra skirtingų matmenų ir uždavinio sprendinys turi būti rastas greitai. Tam jie pasirinko dinaminio programavimo ir lokalios paieškos metodus (6 pav.). Pirmiausia, dinaminio programavimo principu, visi objektai išdėliojami į lakštus. Tam naudojamas godus metodas. Gautas dinaminio programavimo atsakymas perduodamas lokaliai paieškai, kuri sukuria mažesnes užduotis ir optimizuoja gautą sprendinį. Paieška pirmiausia tikrina, ar likusios nepanaudotos vietos dydis yra pakankamas, norint potencialiai atlaisvinti bent vieną lakštą. Jei tokia situacija randama, tuomet šiuos lakštus bandoma perdėlioti, stengiantis atlaisvinti nepanaudotą vietą. Naujas sprendinys priimamas jei pavyksta sutaupyti lakštų arba lokalizuoti nepanaudotą vietą. Nepanaudotos vietos kriterijus įtrauktas norint, kad lokali paieška greičiau išeitų iš lokalių minimumų.

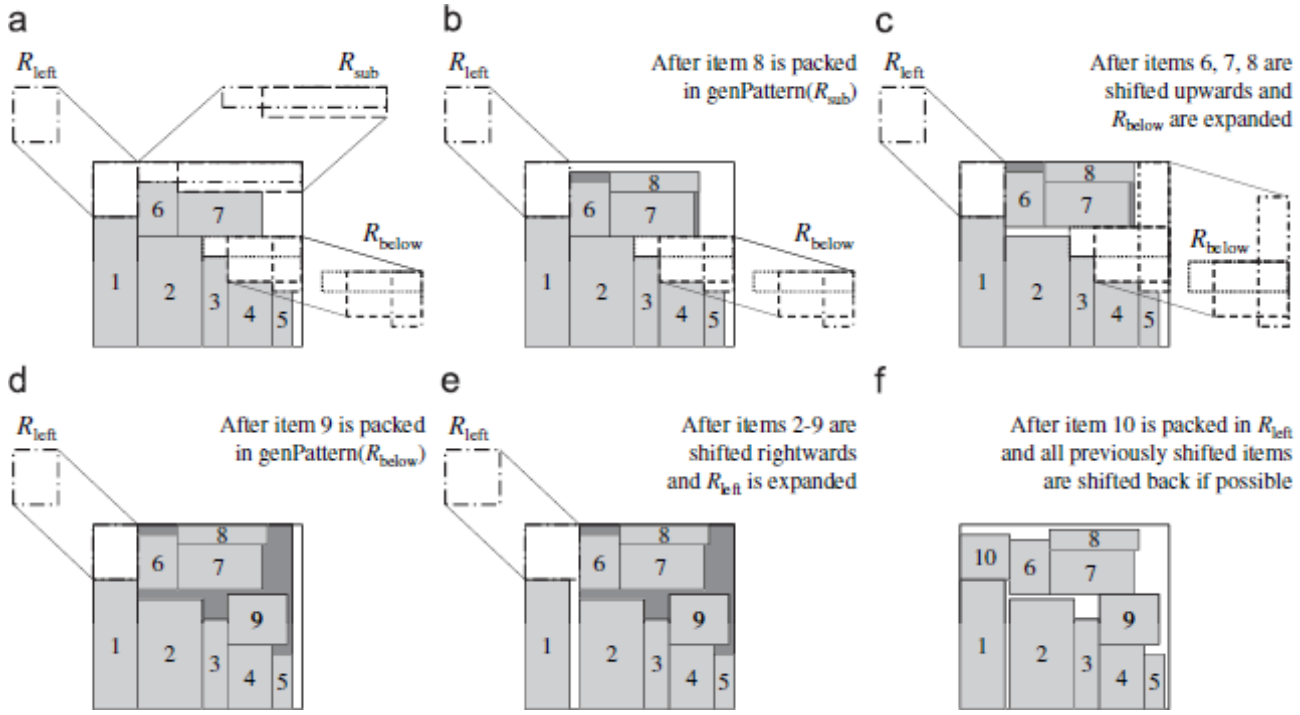
```
Data: An initial solution sol.
nbReleased ← 2;
while The allotted time has not expired and the solution is not
verifiably optimal do
    X ← selectNextSubset(nbReleased);
    subS ← reassign items from X optimally;
    if cost(subS) < cost(X) then
        Reassign items from X in sol according to their assignment in subS;
    end
    if All releasable subsets have been explored since last improvement
    then
        nbReleased ← nbReleased + 1;
    end

end
return sol
```

6 pav. Lokalios paieškos algoritmas [8]

2.2.5. Konstrukcinis algoritmas

C. Charalambousas ir K. Fleszaras [14] pateikia konstrukcinį euristinį algoritmą. Jo esmė – užpildyti lakštą kaip galima pilniau ir tuomet pereiti prie kito lakšto. Vieno lakšto užpildymas atliekamas pagal 7 pav. pateiktą pavyzdį:

1. Iš kairės į dešinę sudedami aukščiausi objektai (7 pav. a). Tuomet lieka 3 laisvos sritys: R_{sub}, R_{below}, R_{left}.
2. Pirmiausia pagal pirmą žingsnį pildomas R_{sub} sritis (7 pav. b).
3. Kai R_{sub} užpildoma, tada visi objektai, esantys virš R_{below}, yra pastumiami iki viršutinės lakšto sienos (7 pav. c). Taip padidinama R_{below} plotas.
4. R_{below} pildoma pagal pirmą žingsnį (7 pav. d).
5. Kai R_{below} sritis užpildoma, tada visi objektai, esantys į dešinę nuo R_{left}, yra pastumiami iki dešinės lakšto sienos (7 pav. e). Taip padidinamas R_{left} plotas.
6. R_{left} sritis pildoma pagal pirmą žingsnį (7 pav. f).
7. Objektai, kurie prieš tai buvo stumiami į viršų ir į dešinę, stumiami kairėn ir žemyn (7 pav. f).



7 pav. Vieno lakšto užpildymas naudojant konstrukcinį algoritmą [14]

2.3. Evoliuciniai euristiniai algoritmai

Šiame skyrelyje apžvelgsime keletą evoliucinių euristinių algoritmų. Tai algoritmai, kurie operuoja žemesnės eilės euristiniais algoritmais. Evoliucinių euristinių algoritmų tikslas – kiekviename žingsnyje pasirinkti labiausiai tinkantį euristinį algoritmą. Šio tipo algoritmai yra populiariausi dvimačio išdėliojimo uždavinio sprendimui, nes duoda gerą sprendinį per trumpą laiką. Išskiriami du evoliucinių euristinių algoritmų tipai: pasirinkimo ir modifikavimo. Pirmuoju atveju stengiamasi iš turimos euristinių algoritmų aibės išsirinkti geriausiai konkrečiam uždavinio atvejui ar žingsniui tinkantį algoritmą, antruoju – iš turimų algoritmų ir taisyklių išvesti naujas taisykles, pagal kurias evoliucinis euristinis algoritmas turėtų dirbti. Apžvelgsime evoliucinį mažiausio tarpo užpildymo, aukštesnės eilės euristikos su tikslumu grįsta mokymosi sistema, atkaitinimo modeliavimo (angl. *simulated annealing*, toliau – SA) ir binarinės paieškos, genetinį grupavimo su valdomu genų perdavimu, bei pasirinkimo evoliucinius euristinius algoritmus.

2.3.1. Evoliucinis mažiausio tarpo užpildymo algoritmas

C. Bluma ir V. Schmidcius [15] pasiūlė modifikuoti patobulintą mažiausio tarpo užpildymo (angl. *improved Lowest Gap Fill*, toliau – LGFi) algoritmą. Standartinis LGFi algoritmas susideda iš dviejų dalių:

1. Pasirengimo dalyje objektai išrikiuojami nedidėjančio ploto seka. Vienodo ploto objektai rikiuojami pagal nedidėjantį absoliutinį ilgio ir pločio skirtumą.
2. Pakavimo dalyje objektai yra dedami į lakštus. Tai iteratyvus procesas:
   2.1. Pirmiausia nustatomas apatinis kairiausias kampas, į kurį objektas gali būti dedamas.
   2.2. Tuomet apskaičiuojami 2 tarpai, susiję su dabartine pozicija:
      2.2.1. Horizontalus tarpas, apibūdinantis atstumą nuo pasirinkto taško iki dešiniosios sienos arba iki arčiausio iš dešinės pusės esančio objekto kairės sienos.
      2.2.2. Vertikalus tarpas, apibūdinantis atstumą nuo pasirinkto taško iki viršutinės sienos arba iki arčiausio iš viršaus esančio objekto apatinės sienos.
   2.3. Pasirenkamas mažesnis iš tarpų (horizontalaus ir vertikalaus).
   2.4. Jei mažesnis horizontalus tarpas, tuomet nesupakuotų objektų sąraše ieškoma labiausiai užpildančio objekto pagal plotį, priešingu atveju – pagal ilgį.
   2.5. Jei randamas visą vietą užpildantis objektas, tuomet jis pasirenkamas. Priešingu atveju pasirenkamas pirmasis tinkamas objektas. Jei nerandamas nei vienas objektas, tuomet

nepanaudota vieta skelbiama šiukšle (šiukšle laikomas visas plotis į kairę ir aukštis iki artimiausio kaimyninio objekto arba iki lakšto viršaus, jei tokio objekto nėra).

8 pav. pateiktas evoliucinis LGFi algoritmas. Pirmiausia, tikimybiškai sugeneruojama p_{size} kiekis sekų. Tuomet kiekvienoje iteracijoje atliekamas geriausių sekų kryžminimas. Dviejų sekų (s ir s^c) kryžminimas vykdomas taip:

1. 3 rodyklės (k, l ir r) nustatomos į atitinkamai 3 sekų pradžias (s, s^c ir s^{off} (šioje sekoje bus saugomas atsakymas).
2. Jeigu $s_k = s_l^c$, tuomet $s_r^{off} \lll s_k$. Tada r padidinamas 1, o rodyklės k ir l perstumiamos į dešinę iki artimiausios pozicijos, kurioje esantis objektas dar nėra įdėtas į s^{off}.
3. Jeigu $s_k \neq s_l^c$, tuomet dedamas objektas pasirenkamas atsitiktinai. Didesnė tikimybė (0,75) duodama objektui, kurio populiacija yra geresnė. Šį kartą r vėl padidinamas 1, tačiau perstumiama tik ta rodyklė, kurios elementas buvo pasirinktas.
4. Galiausiai, s^{off} įvertinamas naudojant LGFi algoritmą. Jei gauta geresnė arba tokia pati reikšmė, tuomet nauja seka įdedama į populiaciją. Priešingu atveju paliekama senoji.
5. Po kryžminimo atliekamas populiacijos papildymas naujais elementais naudojant tą patį algoritmą kaip ir pradinės populiacijos sukūrimui.

Algorithm 1 Evolutionary Algorithm for the 2BP (EA-LGFi)

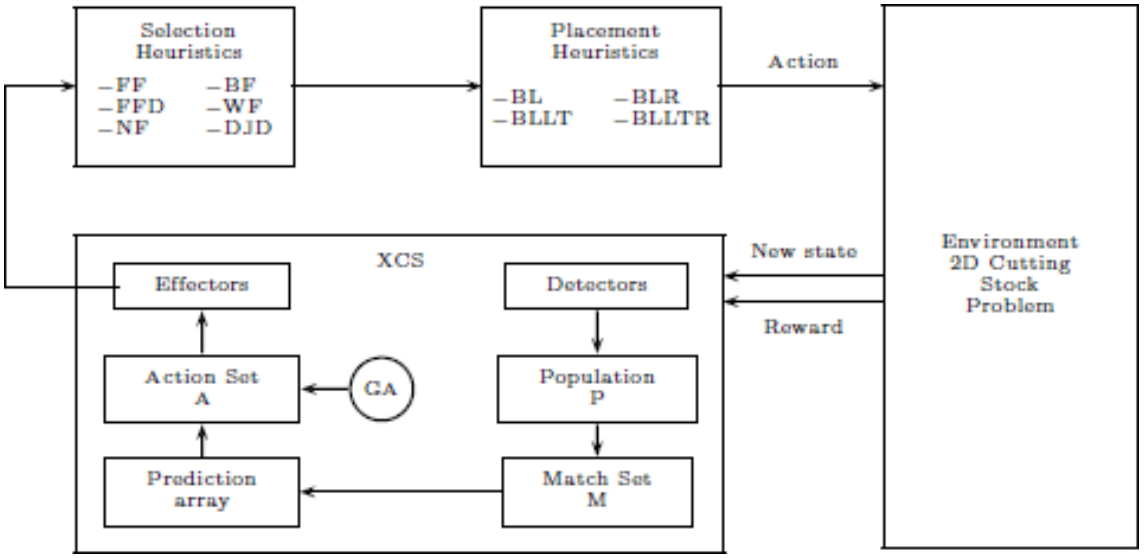
```
1: input: values for parameters p_size, c_rate, κ and δ
2: P := GenerateInitialPopulation(p_size, κ)
3: while stopping criterion not met do
4:     P' := Crossover(P, c_rate, δ)
5:     P := AddNewSolutions(P', p_size, κ)
6: end while
7: output: best solution found
```

8 pav. Evoliucinis mažiausio tarpo užpildymo algoritmas [15]

2.3.2. Aukštesnės eilės euristikos su tikslumu grįsta mokymosi sistema

Autoriai (H. Terashima-Marí, C. J. Farías-Zárate, P. M. Ross ir M. Valenzuela-Rendón [5]) pasiūlė sujungti kelis euristinius algoritmus į vieną evoliucinį euristinį modelį. Evoliucinis euristinis modelis turėtų nuspręsti, kurį žemesnės eilės euristinį algoritmą naudoti. Kito algoritmo pasirinkimas yra dinaminis procesas ir priklauso nuo ankstesniame žingsnyje panaudoto algoritmo gautų rezultatų bei dabartinės paieškos erdvės. Iš viso pasirinkti 8 objekto pasirinkimo euristiniai algoritmai ir 2 objekto dėjimo algoritmai (autoriai nagrinėja atvejį, kai objektus galima sukioti, todėl iš viso gaunama 40 veiksmų kombinacijų). Autoriai pasiūlė 2 evoliucinius euristinius modelius, kurie skirtingais būdais išrenka tolesnį veiksmą.

Pirmasis modelis (9 pav.) turi pradinių taisyklių aibę, kurią periodiškai atnaujina naudodamas genetinį algoritmą. Taip gaunama sistema, kuri gali lengviau reaguoti į aplinkos pokyčius. Tada, pagal gautą aplinkos būseną, išrenkamos tinkamos pasirinkimo ir dėjimo euristikos.

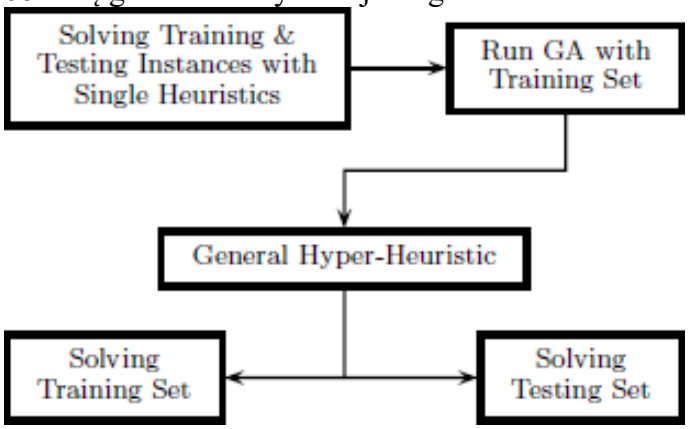


9 pav. Aukštesnės eilės euristika su tikslumu grįsta mokymosi sistema [5]

Antrasis modelis (10 pav.) supaprastintai pateikiamas taip:

1. Nustatyti dabartinę būseną;
2. Surasti artimiausią tinkamą euristiką;
3. Pritaikyti pasirinktą euristiką;
4. Atnaujinti būseną;
5. Kartoti žingsnius 1–4 tol, kol gaunamas uždavinio sprendinys.

Genetinis algoritmas ieško euristikos, kuri geriausiai apibūdina dabartinę sistemos būseną. Tam, pirmiausia, išsprendžiami testiniai uždaviniai su visomis paprastomis euristikomis. Kiekvieno testinio atvejo geriausias sprendinys paliekamas. Tuomet genetinis algoritmas vykdomas su papildomais testiniais duomenimis, kad būtų galima sudaryti naujas algoritmo išrinkimo taisykles.



10 pav. Sprendimo modelis [5]

2.3.3. Atkaitinimo modeliavimo ir binarinės paieškos evoliucinis euristinis algoritmas

Kai kurie autoriai (S. Hong, D. Zhang, H. C. Lau, X. X. Zeng ir Y. W. Si [16]) nagrinėja objektų išdėliojimo uždavinį, kur dalis objektų yra fiksuotos krypties, o dalis gali būti sukinėjami. Uždavinio sprendimui jie pasiūlė SA ir binarine paieška paremtą evoliucinį euristinį algoritmą. Šiuos algoritmus autoriai papildė judėjimo atgal (angl. *backtracking*) algoritmu ir patobulinta taškų skaičiavimo strategija. Pirmiausia vykdomas grįžtamojo ryšio algoritmas skirtingo dydžio lakštų pakavimui. Tuomet, naudojant kelias iteracijas ir taikant patobulintą paiešką, optimizuojamas sprendinys. Visa tai daroma tol, kol randamas optimalus sprendinys arba pasiekiamas nurodytas laiko limitas.

2.3.4. Genetinis grupavimo algoritmas su valdomu genų perdavimu

Autoriai (M. Quiroz-Castellanos, L. Cruz-Reyes, J. Torres-Jimenez, C. S. Gómez, H. J. F. Huacuja ir A. C. F. Alvim [17]) sprendžia vienmatį objektų išdėliojimo uždavinį. Jie pasiūlė genetinį grupavimo algoritmą su valdomu genų perdavimu (angl. *grouping genetic algorithm with*

controlled gene transmission, toliau – GGA-CGT). GGA-CGT naudoja efektyvias euristikas geriausių genų perdavimui tam, kad išvestų aukštos kokybės sprendinį. Apibendrintas algoritmas pateiktas 11 pav.

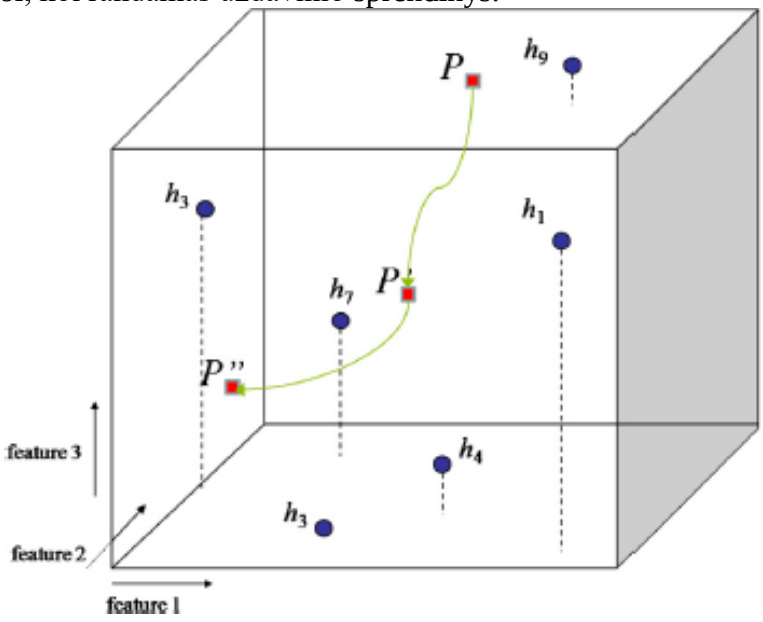
```
procedure GGA-CGT
1   Generate an initial population P with FF-ų;
2   while generation < max_gen and Size(best_solution) > L_2 do
3      Select n_c individuals to cross by means of Controlled_Selection;
4      Apply Gene_Level_Crossover+FFD to the n_c selected individuals;
5      Apply Controlled_Replacement to introduce progeny;
6      Select n_m individuals and clone elite solutions by means of Controlled_Selection;
7      Apply Adaptive_Mutation+RP to the best n_m individuals;
8      Apply Controlled_Replacement to introduce clones;
9      Updated the global_best_solution;
10  end;
end GGA-CGT.
```

11 pav. Genetinis grupavimo algoritmas su valdomu genų perdavimu [17]

Pirmiausia sugeneruojama pradinė populiacija *P*. Tada maksimalų leistiną iteracijų skaičių arba kol randamas optimalus sprendinys, vyksta populiacijos *P* rekombinacija ir mutacija 2 fazėmis. Pirmojoje fazėje išrenkami geriausi individai, atliekamas jų genų sumaišymas, ir surandami palikuonys. Antrojoje fazėje geriausi individai yra klonuojami ir paimami genetiniams pakeitimams. Atlikus pakeitimus, klonai keičia dalį populiacijos *P* individų. Atnaujinamas geriausias rastas sprendinys ir ciklas pradedamas iš naujo.

2.3.5. Pasirinkimo evoliucinis euristinis algoritmas

Kai kurie autoriai (E. López-Camacho, H. Terashima-Marin, P. Ross ir G. Ochoa [18]) pasiūlė evoliucinį euristinį algoritmą, kuris tinka ne tik stačiakampiams objektams, bet ir kitokiems poligonams. Metodo esmė – kiekviename žingsnyje pasirinkti labiausiai tinkantį euristinį algoritmą. Savo metodą jie apibūdina 12 pav. pateiktu 3-mačiu kubu. Iš pradžių sudaromas n-matis kubas, kuriame kiekvienas taškas *h* atitinka vieną euristiką. Tada, pradedant pradine uždavinio būsena *P*, ieškoma artimiausios euristikos (pvz., atvejis h_9), kuri transformuoja uždavinį į būseną *P'*. Taip procesas vykdomas tol, kol randamas uždavinio sprendinys.



12 pav. Euristinio algoritmo pasirinkimo 3-matis kubas [18]

Euristinių algoritmų įvertinimas sudarytas iš 6 žingsnių:

1. Pradiniai duomenų rinkiniai išspręsti su visais euristiniais algoritmais;
2. Kiekvienu atveju veikimo sparta apibūdinama R^6 (naudojami 6 algoritmai) aibės normalizuotu vektoriumi;

3. Visi gauti sprendiniai suskirstyti į 8 grupes pagal panašumą;
4. Kiekvienam uždavinio variantui buvo nustatyta po 23 pagrindines savybes;
5. Koreliuojančių savybių pašalinimas;
6. Labiausiai tinkančių savybių pasirinkimas.

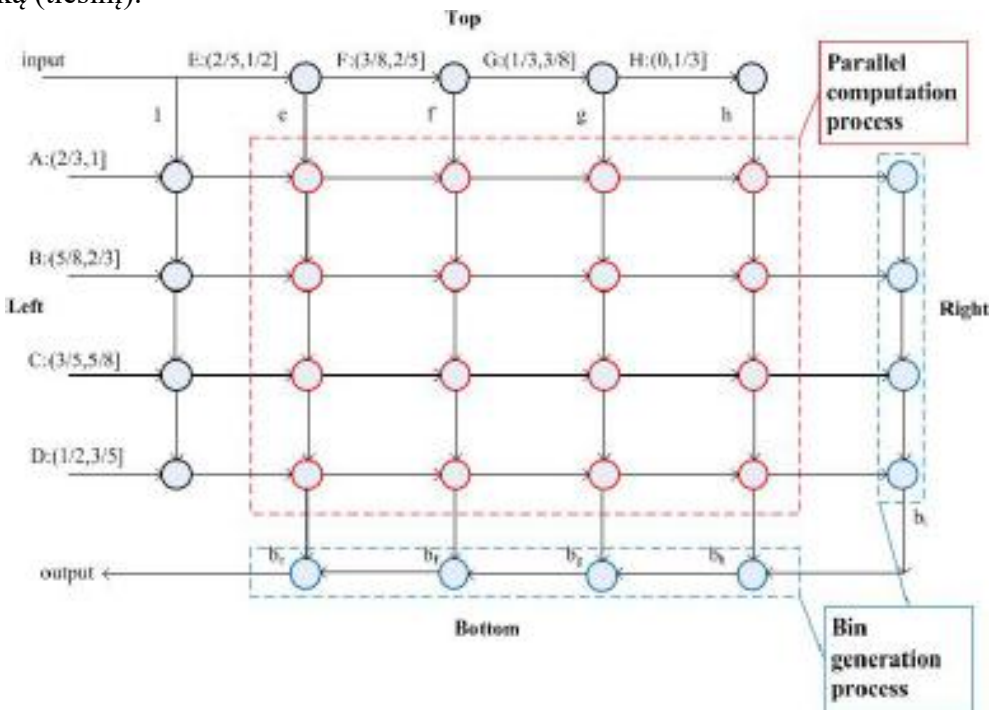
Taip jie sudarė 10 svarbiausių savybių rinkinį (12 pav. virstų 10-mačiu kubu). Galiausiai, kiekvienoje uždavinio būsenoje reikia nustatyti visų savybių reikšmes ir pasirinkti artimiausią euristinį algoritmą.

2.4. Uždavinio sprendimas naudojant lygiagrečius skaičiavimus

Šiame skyrelyje apžvelgsime siūlomus lygiagretiems skaičiavimams pritaikytus dvimačio išdėliojimo uždavinio sprendimo algoritmus. Pirmiausia apžvelgsime algoritmą, kuris yra pritaikytas skaičiavimams aplinkoje, sudarytoje iš daug procesorių (toliau – CPU). Po to apžvelgsime kaip dideliems skaičiavimams bandoma išnaudoti GPU.

2.4.1. Lygiagretūs skaičiavimai naudojant daug procesorių

X. Zhao ir H. Shenas [19] sprendžia kvadratinių objektų išdėliojimo uždavinį. Kadangi objektai ir lakštai yra kvadratiniai, uždavinys virsta vienmačiu. Objektai, didesni nei ½ lakšto dydžio, negali būti sudėti kartu (atitinkamai jei objektas, kurio dydis ½ lakšto dydžio jau yra lakšte, tuomet į jį negali tilpti 2 objektai, kurių dydžiai didesni nei ¼ lakšto dydžio). Pasinaudodami šiuo principu, autoriai išdalina visus objektus 8 CPU (13 pav. juodi apskritimai). Naudodami 16 skaičiavimams (13 pav. raudoni apskritimai) ir 8 lakštų valdymui (13 pav. mėlyni apskritimai) skirtus CPU jie atlieka skaičiavimus. 13 pav. rodyklėmis parodyta, kurie CPU siejasi tarpusavyje. Ši sistemą rezultatą gauną per $\Theta(n)$ laiką (tiesinį).



13 pav. Lygiagretus dvimačio dėliojimo uždavinio sprendimas [19]

2.4.2. Lygiagretūs skaičiavimai naudojant grafinį procesorių

Keli autorių kolektyvai (V. Boyer, D. E. Baz ir M. Elkihel [20] bei M. C. Feier, C. Lemnaru ir R. Potolea [21]) pabandė kuprinės uždavinį (angl. *knapsack problem*) spręsti naudodami GPU. Šis uždavinys glaudžiai susijęs su objektų išdėliojimo uždaviniu. Jie naudojo „Nvidia“ sukurtą CUDA technologiją. CUDA technologija yra paremta vienos instrukcijos daugeliui gijų pagrindu (angl. *single instruction multiple threads*, toliau – SIMT). SIMT pasižymi tuo, kad kelios gijos vykdo tuos pačius veiksmus su skirtingais duomenimis. Tai leidžia sutaupyti laiko instrukcijoms įkrauti.

Abu autorių kolektyvai pateikė euristinius algoritmus kuprinės uždaviniu spręsti. Pirmojo autorių kolektyvo sudarytas algoritmas GPU veikė 26 kartus greičiau nei CPU, antrojo – net 67 kartus greičiau.

Reiktų atkreipti dėmesį į tai, kad abu autorių kolektyvai naudojo tik savo pasirinktus duomenų rinkinius. Tai neleidžia palyginti jų sprendinio korektiškumo ir tikslumo su kitais algoritmais.

2.5. Dabartinių sprendimų įvertinimas

Atlikus apžvalgą pastebėta, jog šiuo metu populiariausi yra evoliuciniai euristiniai algoritmai. Taip yra dėl to, kad euristiniai algoritmai pritaikyti specifiniams uždavinio atvejams (pvz., turint daug pasikartojančių objektų naudojami, pasikartojimų ieškantys algoritmai). Deja, tokių algoritmų veikimo sparta arba sprendinio tikslumas gerokai krenta, jei uždavinys nėra iš jiems tinkamų uždavinių aibės.

Evoliucinių euristinių algoritmų tikslas – sujungti geriausias euristinių algoritmų savybes. Šiuo metu išskiriami du evoliucinių euristinių algoritmų tipai: pasirinkimo ir modifikacijų. Pirmuoju atveju stengiamasi iš turimos euristinių algoritmų aibės išsirinkti geriausiai konkrečiam uždavinio atvejui ar žingsniui tinkantį algoritmą, antruoju – iš turimų algoritmų ir taisyklių išvesti naujas taisykles, pagal kurias evoliucinis algoritmas turėtų dirbti.

Pastebėta, kad skiriamas gana mažas dėmesys lygiagrečių algoritmų paieškai. X. Zhao ir H. Sheno [19] darbe matoma, kad, naudojant lygiagrečius skaičiavimus, galima paspartinti uždavinio sprendimą (aišku, jie nagrinėja vienmatį atvejį).

Atsižvelgiant į autorių, tyrinėjusių kuprinės uždavinio pritaikymą skaičiavimams GPU, pasiektus rezultatus, galime spręsti, kad objektų išdėliojimo uždavinys puikiai tinka skaičiavimams GPU. Šie autoriai apsiriboja euristinių algoritmų taikymu. Kadangi šiuo metu geriausius rezultatus pasiekia evoliuciniai euristiniai algoritmai, todėl bus kuriamas evoliucinis euristinis modifikavimo algoritmas ir analizuojamas jo veikimas GPU.

3. PROJEKTINĖ DALIS

Šiame skyriuje pateikti sukurtos programinės įrangos (toliau – PĮ) projektinės dokumentacijos esminiai aspektai. Architektūra pateikiama šiose diagramose:

1. Panaudos atvejų (toliau – PA) diagrama. Šioje diagramoje naudojami simboliai:
   1.1. Aktorius (žmogeliukas) – kuriamą PĮ naudojantis žmogus ar sistema. Apačioje nurodomas jo pavadinimas;
   1.2. Stačiakampis – kuriamos PĮ ribos;
   1.3. Ovalas – PĮ teikiamas funkcionalumas aktoriui. Ovalo viduje nurodomas PA pavadinimas.
2. Komponentų diagrama. Šioje diagramoje naudojami simboliai:
   2.1. Stačiakampis – PĮ komponentas. Stačiakampio viduje nurodomas komponento pavadinimas;
   2.2. Linija su burbuliuku – komponento teikiama sąsaja;
   2.3. Linija su lankeliu – naudojama kito komponento sąsaja.
3. Klasių diagrama. Šioje diagramoje naudojami simboliai:
   3.1. Stačiakampis apvalintais kampais su tekstu *Class* – klasė (angl. *class*). Viršuje nurodomas klasės pavadinimas:
      3.1.1. Klasėje aprašomi komponentai:
         3.1.1.1. Kintamieji (angl. *fields*);
         3.1.1.2. Savybės (angl. *properties*);
         3.1.1.3. Metodai (angl. *methods*).
      3.1.2. Klasių variantai:
         3.1.2.1. Paryškintomis sienomis – uždara (angl. *sealed*) klasė. Ši klasė negali būti paveldėta;
         3.1.2.2. Brūkšninėmis sienomis – abstrakti (angl. *abstract*) klasė. Šią klasę reikia paveldėti ir įgyvendinti (angl. *implement*) trūkstamus metodus.
   3.2. Stačiakampis apvalintais kampais su tekstu *Interface* po pavadinimu – sąsaja (angl. *interface*). Viršuje nurodomas sąsajos pavadinimas. Sąsajoje naudojami komponentai:
      3.2.1. Savybės;
      3.2.2. Metodai.
   3.3. Stačiakampis su tekstu *Struct* – struktūra (angl. *struct*). Viršuje nurodomas struktūros pavadinimas. Sąsajoje naudojami komponentai:
      3.3.1. Kintamieji;
      3.3.2. Metodai.
   3.4. Stačiakampis su tekstu *Enum* – išvardijimas (angl. *enum*). Išvardijimo struktūra, skirta konstantoms su pavadinimais saugoti. Viršuje nurodomas išvardijimo pavadinimas.
   3.5. Rodyklė su apskritimu – klasės įgyvendinama sąsaja.
   3.6. Ryšiai tarp klasių:
      3.6.1. Su viena rodykle gale rodo, kad naudojamas vienas objektas;
      3.6.2. Su dviem rodyklėmis gale rodo, kad naudojamas daugiu nei vienas objektas.
   3.7. Modifikatoriai:
      3.7.1. Jei prie komponento nėra nurodyta jokio modifikatoriaus, tai šis elementas yra atviras (angl. *public*);
      3.7.2. Jei prie elemento yra širdis (♥), tai šis elementas yra atviras tik šios bibliotekos elementams (angl. *internal*);
      3.7.3. Jei prie elemento yra žvaigždutė (✷), tai šis elementas yra apsaugotas (angl. *protected*);
      3.7.4. Jei prie elemento yra spyna (🔒), tai šis elementas yra privatus (angl. *private*).
   3.8. Metodo aprašymas:
      3.8.1. Skliausteliuose paprastai nurodomi metodui perduodami parametrai, tačiau šiose diagramose skliausteliai palikti tušti. Toks sprendimas priimtas, nes, sudėjus visus metodų priimamus parametrus, diagramos taptų neįskaitomomis;

3.8.2. Po dvitaškio pateikiamas metodo grąžinamo objekto tipas.

4. Sekų diagrama. Šioje diagramoje naudojami simboliai:
   4.1. Stačiakampis su punktyrine linija žemyn ir žmogeliuku viršuje – aktoriaus;
   4.2. Stačiakampis su punktyrine linija žemyn – objektas;
   4.3. Stačiakampis ant aktoriaus ar objekto linijos – elemento gyvavimo laikas. Šiuo metu elementas egzistuoja (objekto atveju turi būseną, nėra sunaikintas);
   4.4. Tekstas stačiakampių viduje prieš dvitaškį – objekto pavadinimas;
   4.5. Tekstas stačiakampio viduje po dvitaškio – klasės, kurios veiksmus nurodytas objektas atlieka, pavadinimas.
   4.6. Naudojamos rodyklės:
      4.6.1. Paprasta linija su pilnavidure rodykle gale – objekto metodo kvietimas. Jei rodyklėje yra tekstas *Setter*, tuomet tai objekto savybių nustatymas;
      4.6.2. Paprasta linija su paprasta rodykle gale – metodo kvietimas asinchroniškai. Nėra laukiama atsakymo, darbas tęsiamas toliau;
      4.6.3. Punktyrinė linija su rodykle gale – grįžimas iš metodo. Tekstas nurodo grąžinamą kintamąjį;
      4.6.4. Stačiakampis su tekstu *loop* – ciklas, kuris vykdomas tol, kol galioja po tekstu *loop* nurodyta sąlyga.

5. Bendradarbiavimo diagrama. Šioje diagramoje naudojami simboliai:
   5.1. Aktorius (žmogeliukas) – kuriamą PĮ naudojantis žmogus ar sistema. Apačioje nurodomas jo pavadinimas;
   5.2. Stačiakampis – objektas;
   5.3. Rodyklės su tekstu – vykdomi veiksmai. Veiksmų eiliškumą nurodo numeracija;
   5.4. Žvaigždutė po numerio nurodo, kad veiksmas gali būti kartojamas daugiau nei vieną kartą.

6. Duomenų bazės (toliau – DB) modelis. Šioje diagramoje naudojami simboliai:
   6.1. Stačiakampis atitinta DB lentelę;
   6.2. Pirmoje eilutėje pateiktas DB lentelės pavadinimas;
   6.3. Toliau pateikiami lentelės atributai:
      6.3.1. Atributai su tekstu *PK* – pirminiai raktai;
      6.3.2. Po atributo pavadinimo eina atributo tipas.

3.1. Architektūros tikslai ir apribojimai

Dvimatis išdėliojimo uždavinys – klasikinis optimizavimo uždavinys. Kadangi iki šiol nesukurtas efektyvus algoritmas, garantuojantis greitą optimalaus sprendinio radimą, naudojamos įvairios euristikos. Atsižvelgiant į tai, architektūra turėtų būti sudaryta taip, kad būtų galima atnaujinti ir plėsti sukurtą PĮ.

Skaičiavimai yra imlūs laikui, todėl kuriama PĮ turėtų veikti lygiagrečiai, nestabdydama dabartinio proceso darbo. Kuriama PĮ su išorinėmis sistemomis bendraus žinutėmis. Bus reikalingos skaičiavimų pradžios iniciavimo ir skaičiavimų nutraukimo žinutės, tačiau jomis nebus perduodami PĮ duomenys. Žinutėse perduodamos nuorodos į DB lenteles, kuriose saugoma informacija.

PĮ projektuojama dviem įrankiais: „Microsoft Visual Studio 2015“ ir „Microsoft Visio 2013“. „Microsoft Visual Studio 2015“ naudojama klasių diagramai braižyti, o visos kitos diagramos braižomos „Microsoft Visio 2013“. Toks klasių diagramos braižymo išskirtinumas pasirinktas todėl, kad PĮ bus kuriama „.NET“ platforma. Kadangi „Microsoft Visual Studio 2015“ aplinkoje, braižant klasių diagramą, automatiškai kuriamos klasės, metodai ir kiti elementai, kartu su diagrama bus sukurtas ir pradinis PĮ šablonas.

3.2. Panaudos atvejų vaizdas

PA vaizdas pateiktas 14 pav., o aprašymas – 1 lentelėje.

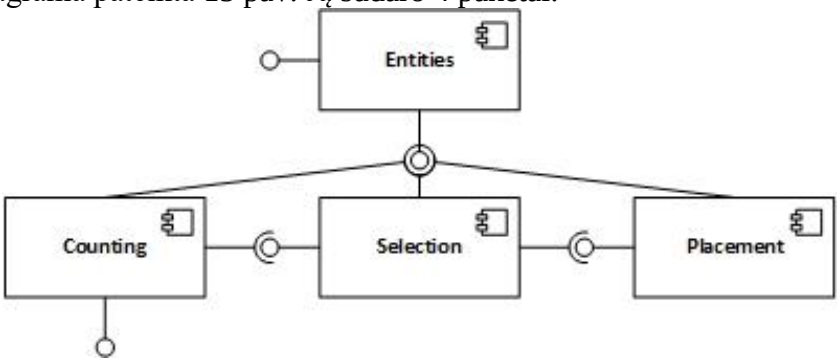
14 pav. Panaudos atvejų diagrama

1 lentelė. Skaičiavimų iniciavimo panaudojimo atvejo aprašymas

ID	1
Tikslas	Inicijuoti skaičiavimus ir sudėlioti objektus kaip galima geriau ir greičiau.
Aktoriai	„Optitecha“ sistema
Ryšiai su kitais PA	–
Nefunkciniai reikalavimai	500 objektų reikia sudėlioti per 5 min.
Prieš-sąlygos	„Optitecha“ sistema informaciją apie objektus įrašė į specialią DB lentelę.
Sužadinimo sąlyga	Skaičiavimų iniciavimo iškvietimas.
Po-sąlyga	Rezultatai įrašomi į specialią DB lentelę ir „Optitecha“ sistema informuojama apie darbo pabaigą.
Pagrindinis scenarijus	1. „Optitecha“ sistema inicijuoja skaičiavimus. 2. PĮ pasiima duomenis iš DB. 3. Uždavinio sprendimas. 4. Rezultatai įrašomi atgal į DB. 5. PĮ informuoja „Optitecha“ sistemą apie darbo pabaigą.
Alternatyvus scenarijus	–

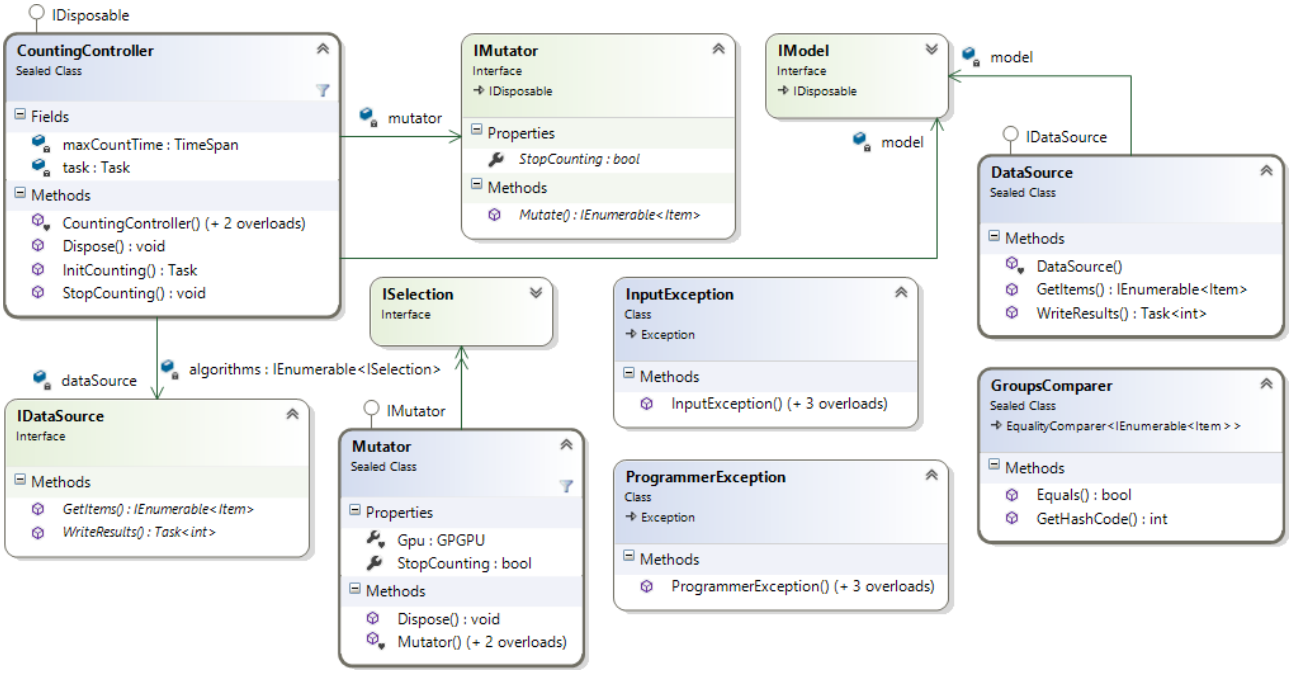
3.3. Sistemos statinis vaizdas

PĮ paketų diagrama pateikta 15 pav. Ją sudaro 4 paketai:



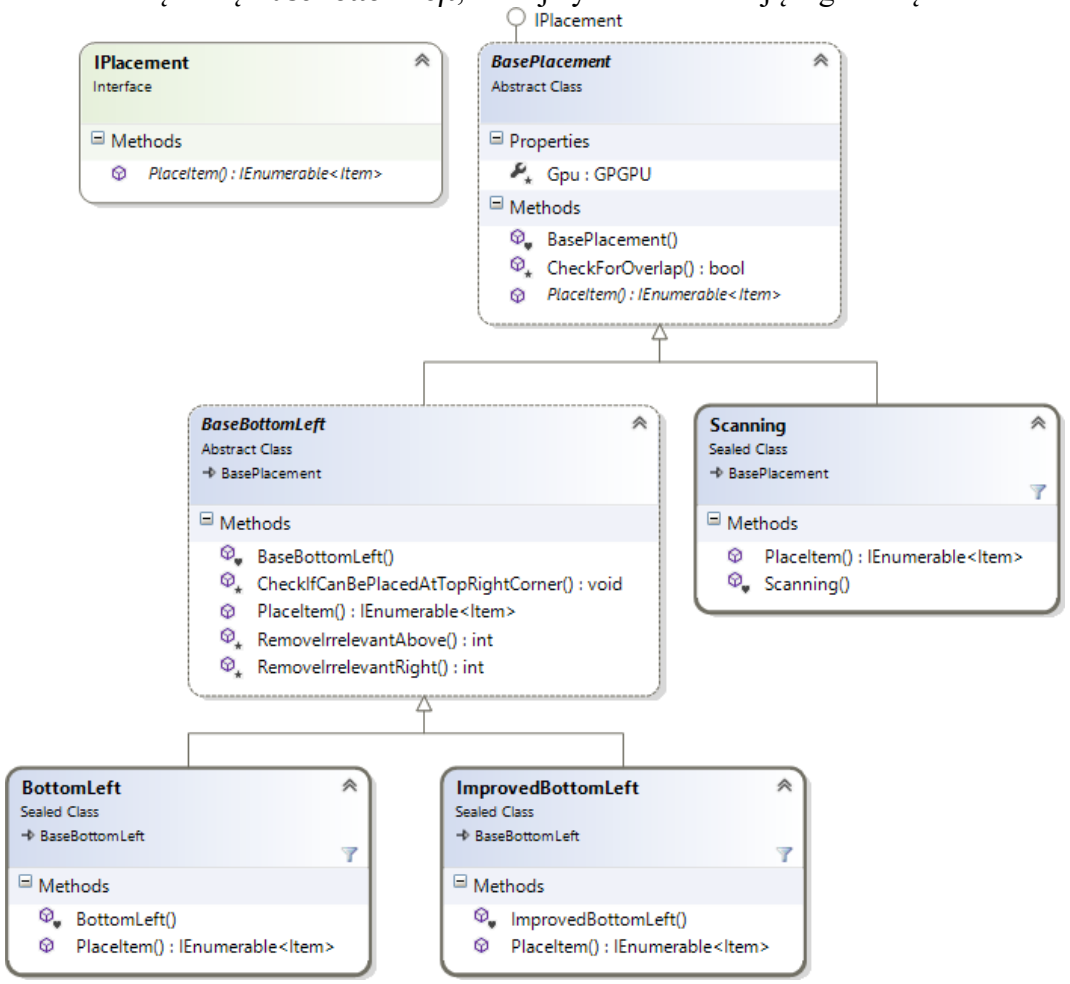
15 pav. Paketų diagrama

1. *Counting* (16 pav.) – už skaičiavimus atsakingas paketas. Tai pagrindinis sistemos paketas, kuriame yra skaičiavimų valdymo klasė (*CountingController*). Per šios klasės sąsają bus bendraujama su išorinėmis sistemomis. Taip pat yra *DataSource* klasė, kuri yra skirta bendrauti su DB, ir *Mutator* klasė, kuri skirta objektų išdėliojimo uždaviniui spręsti.



16 pav. *Counting* paketo klasių diagrama

2. *Placement* (17 pav.) – paketas, kuriame yra skaičiavimams naudojami dėjimo algoritmai. Visi dėjimo algoritmai įgyvendina *IPlacement* sąsają. Taip pat *BottomLeft* ir *ImprovedBottomLeft* klasės turi tėvinę klasę *BaseBottomLeft*, kurioje yra bendri abiejų algoritmų metodai.



17 pav. *Placement* paketo klasių diagrama

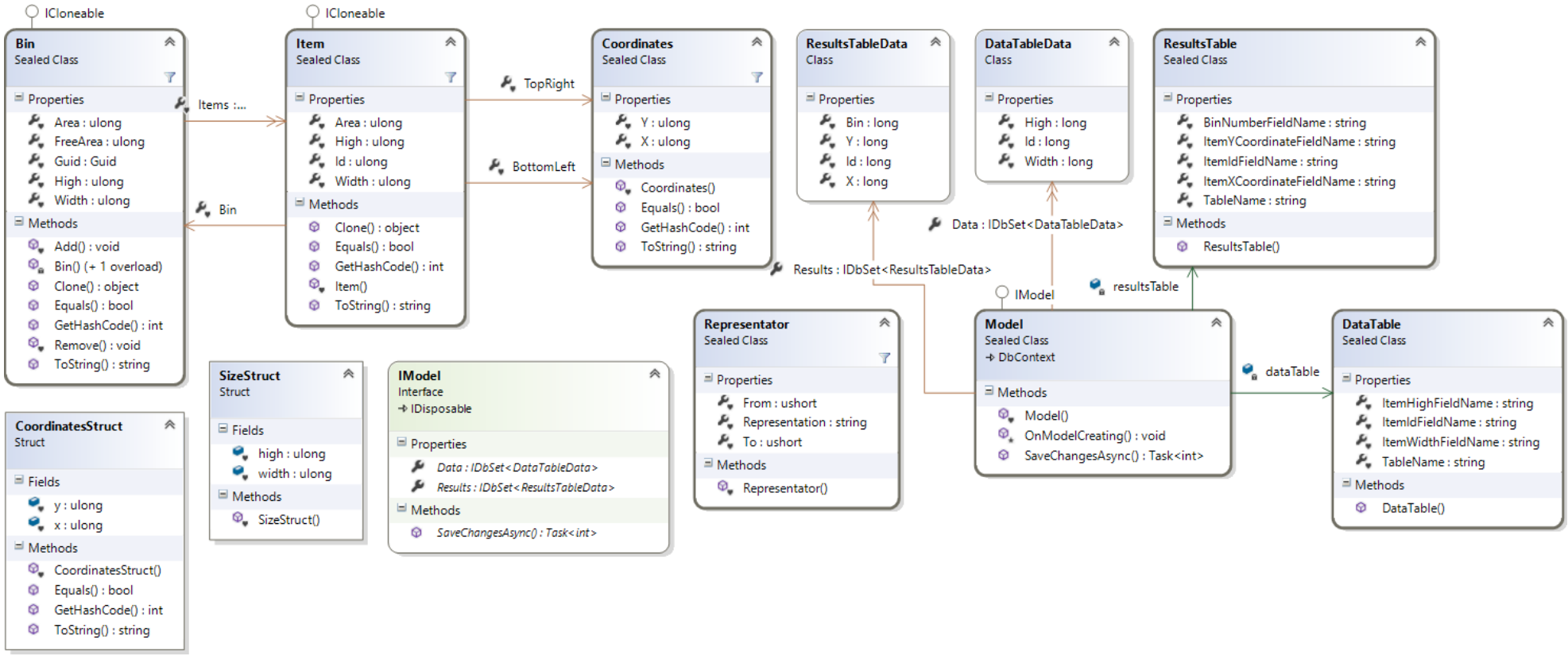
3. *Entities* (18 pav.) – paketas, kuriame yra naudojamos duomenų struktūros. Kai kurios duomenų struktūros yra panašios, pvz., *Coordinates* ir *CoordinatesStruct*, *Item* bei *Bin* ir

SizeStruct. Toks klasių informacijos kopijavimas į struktūras reikalingas, nes CUDA nedirba su klasių objektais. CUDA palaiko tik primityvius duomenų tipus, struktūras ir masyvus.

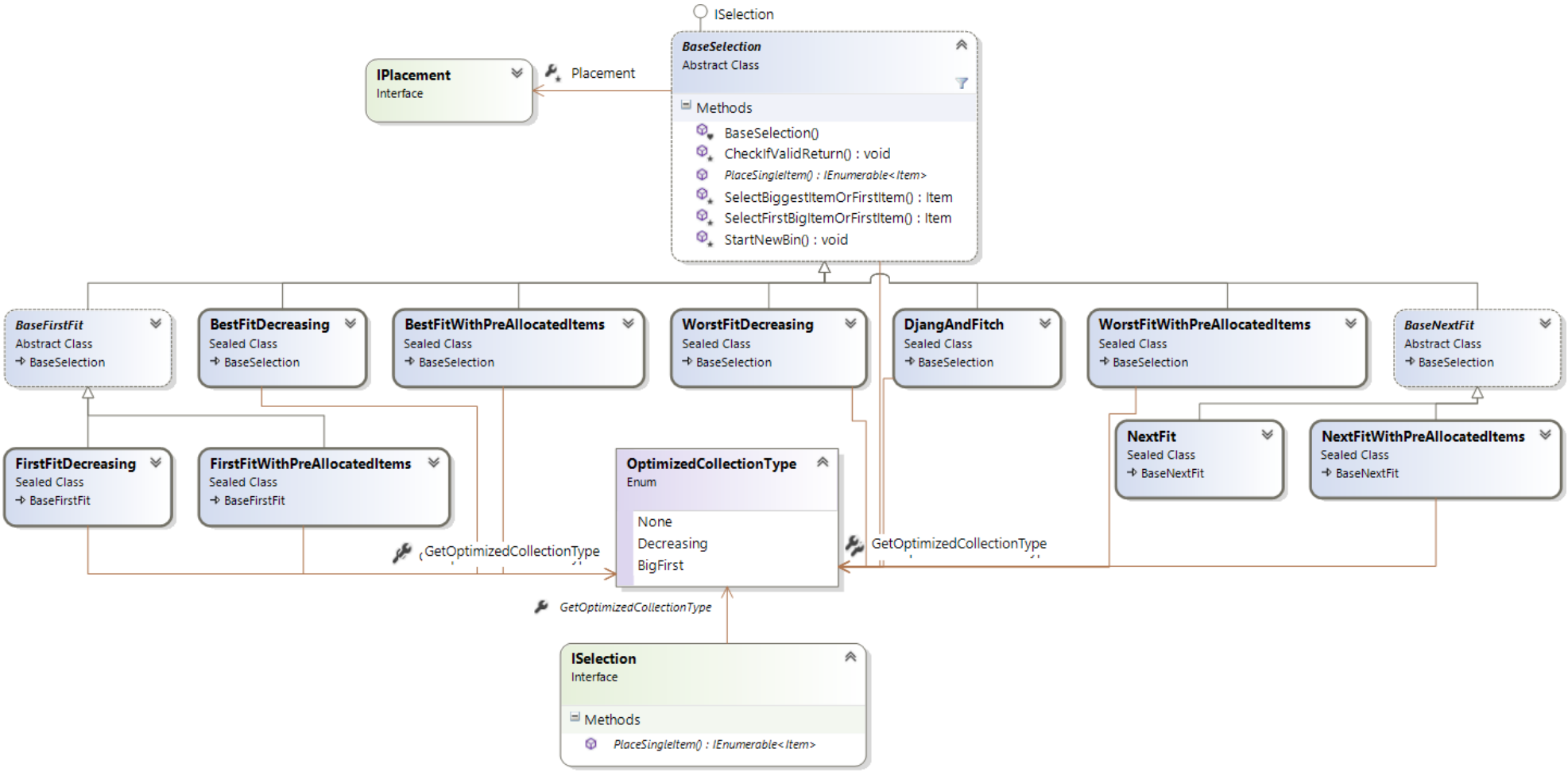
4. *Selection* (19 pav.) – paketas, kuriame yra skaičiavimams naudojami išrinkimo algoritmai. Visi išrinkimo algoritmai įgyvendina *ISelection* sąsają. Taip pat *FirstFitDecreasing* ir *FirstFitWithPreAllocatedItems* turi tėvinę klasę *BaseFirstFit*, bei *NextFit* ir *NextFitWithPreAllocatedItems* turi tėvinę klasę *BaseNextFit*, kuriose yra bendri algoritmų metodai.

3.4. Sistemos dinaminis vaizdas

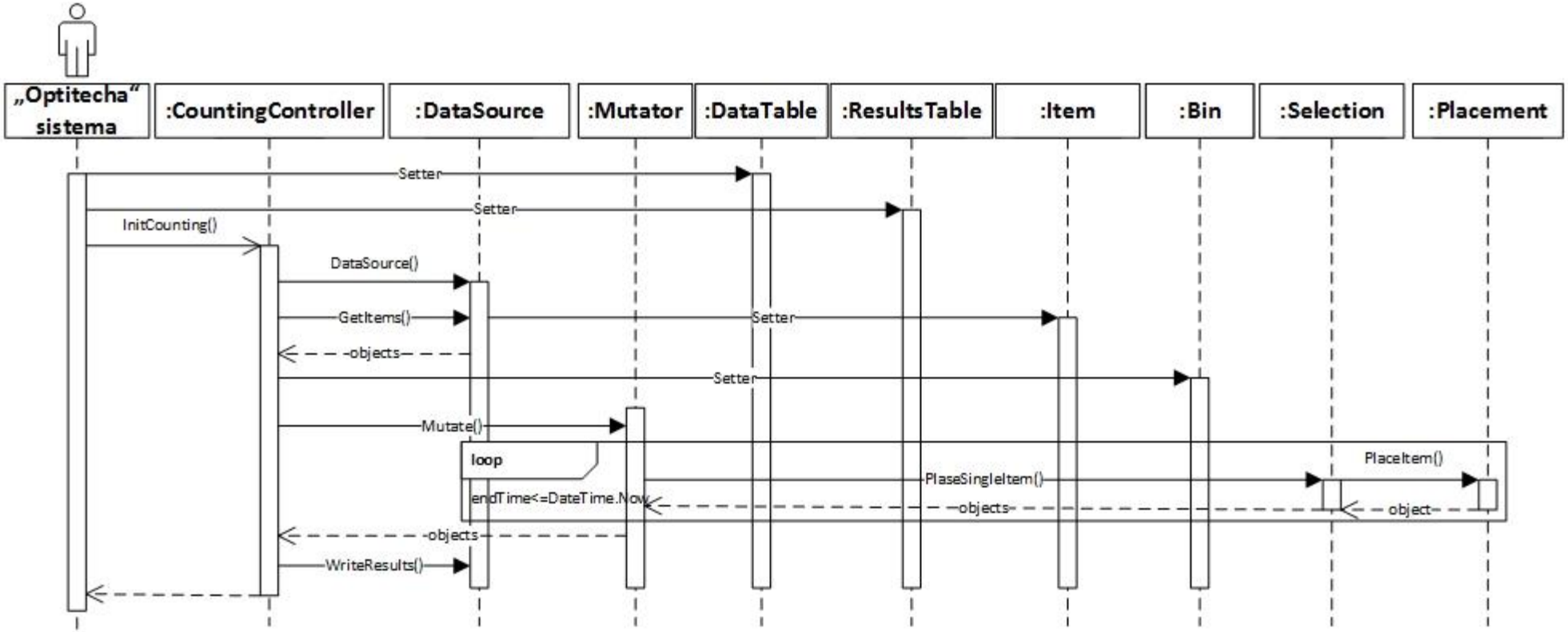
PĮ skaičiavimų iniciavimo sekų diagrama pateikta 20 pav. Joje matome, kad aktorius inicijuoja skaičiavimus ir toliau gali tęsti darbą – baigusi darbą sistema jį informuos. Pirmiausia vykdomas duomenų nuskaitymas iš DB. Tuomet su iš DB nuskaitytais duomenimis vykdomas evoliucinis euristinis algoritmas (*Mutate*). Skaičiavimai vykdomi tol, kol randamas optimalus sprendinys, baigiasi aktoriaus nurodytas skaičiavimų laikas arba sprendinys nėra pagerinamas iš anksto numatytą iteracijų skaičių.



18 pav. *Entities* paketo klasių diagrama

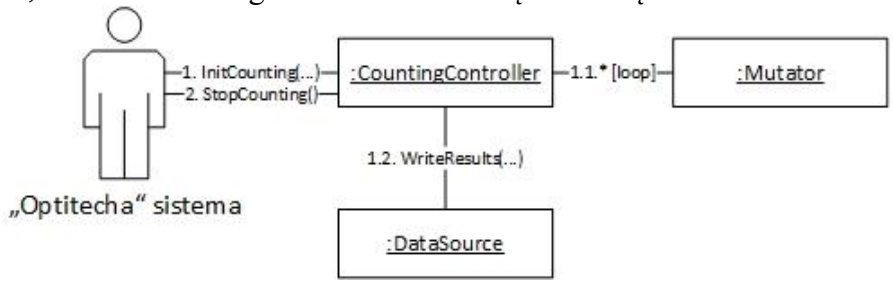


19 pav. *Selection* paketo klasių diagrama



20 pav. Skaičiavimų iniciavimo sekų diagrama

Darbo nutraukimo galimybė parodyta bendradarbiavimo diagrama (21 pav.). Ji glaudžiai susijusi su sekų diagrama (20 pav.). Iniciavus skaičiavimus (*InitCounting*), *CountingController* nuskaito reikiamus duomenis iš DB. Tuomet, vykdant evoliucinį euristinį algoritmą, aktorius gali sustabdyti procesą naudodamas *StopCounting* metodą. Aktoriui sustabdžius skaičiavimus, baigiamas dabartinis skaičiavimų ciklas, ir iki šiol rastas geriausias rezultatas įrašomas į DB.

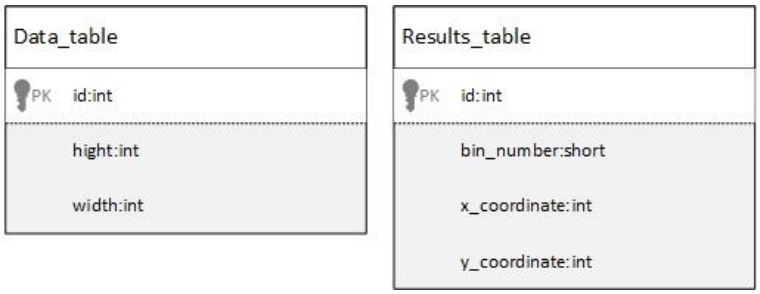


21 pav. Skaičiavimų sustabdymo bendradarbiavimo diagrama

3.5. Duomenų vaizdas

Rekomenduojama DB diagrama pateikta 22 pav. Darbui reikalingos dvi lentelės:

1. *Data_table* – duomenų lentelė, kurioje pateikiami objektų išmatavimai:
   1.1. *id* – objekto identifikatorius (toliau – ID);
   1.2. *hight* – objekto aukštis;
   1.3. *width* – objekto plotis.
2. *Results_table* – rezultatų lentelė, kurioje įrašomi gauti skaičiavimų rezultatai:
   2.1. *id* – objekto ID;
   2.2. *bin_number* – lakšto, į kurį objektas dedamas, numeris. Šį numerį sugeneruoja PĮ. Lakštų numeravimas pradedamas nuo 0;
   2.3. *x_coordinate* – x (horizontali) koordinatė lakšte, kurioje yra kairysis apatinis objekto kampas;
   2.4. *y_coordinate* – y (vertikali) koordinatė lakšte, kurioje yra kairysis apatinis objekto kampas.



22 pav. Duomenų bazės modelis

Tai privalomi DB lentelių laukai, tačiau vartotojui paliekama laisvė. Kadangi tiek lentelių, tiek laukų vardai yra vartotojo užduodami inicijuojant skaičiavimo procesą, vartotojas gali:

1. Naudoti vieną lentelę duomenims saugoti;
2. Keisti lentelių pavadinimus;
3. Keisti atributų pavadinimus;
4. Keisti atributų tipus.

Atliekant atributų tipų pakeitimus, reikėtų atsižvelgti į tai, kad visi atributai turi būti sveiki skaičiai, t. y., galima keisti atributų tipus tarp „short“, „int“ ir „long“, tačiau negalima vartoti „nvarchar“, „double“ ir kitų ne sveiko skaičiaus tipų.

3.6. Projektinės dalies apibendrinimas

PĮ buvo kuriama taip, kad ji būtų patogi tiek ją palaikančiam personalui, tiek ją naudojantiems aktoriams. Palaikančiam personalui darbą palengvina tai, kad PĮ dalys nėra griežtai surištos – naujo išrinkimo ar įdėjimo algoritmas iš esmės nekeičia kitų sistemos dalių (užtektų įtraukti šį algoritmą į

galimų algoritmų sąrašą evoliuciniame euristiniame algoritme). Iš kitos pusės, paties evoliucinio euristinio algoritmo keitimas visiškai neveikia išrinkimo ir dėjimo algoritmų. Taip pat kiekviena klasė konstruktoriuose (angl. *constructor*) reikalauja ne konkrečios klasės objekto, o sąsajos. Tai užtikrina lengvesnį klasių testavimą bei galimybę sukurti naujas sąsają realizuojančias klases.

Aktoriams leidžiama turėti pakankamai laisvą DB struktūrą. Tai labai palengvina integraciją su jau egzistuojančiomis sistemomis. Vartotojams nereikėtų keisti DB struktūros vien tik tam, kad galėtų atlikti skaičiavimus sukurta PĮ. Ši sistema darbą vykdo lygiagrečiai. Tai leidžia naudoti PĮ nestabdant dabartinio proceso darbo, kas ypač aktualu, norint leisti vartotojo sąsajai veikti toliau.

4. TYRIMO DALIS

Šiame skyriuje aprašomas siūlomas evoliucinis euristinis algoritmas bei jo pritaikymas lygiagretiems skaičiavimams CUDA aplinkoje. Taip pat šiame skyriuje aprašomos priemonės, kurių buvo imtasi norint užtikrinti sukurtos PĮ kokybę.

4.1. Siūlomas evoliucinis euristinis algoritmas

Kai kurie objektų išdėliojimo uždaviniai pasižymi savybėmis, dėl kurių specifinės euristikos juos sprendžia greitai ir tiksliai. Tačiau nėra vienos euristikos, kuri puikiai spręstų visus galimus atvejus. Evoliucinio euristinio algoritmo idėja yra sujungti paprastus euristinius algoritmus, taip bandant išvengti kiekvieno iš algoritmų silpnų vietų.

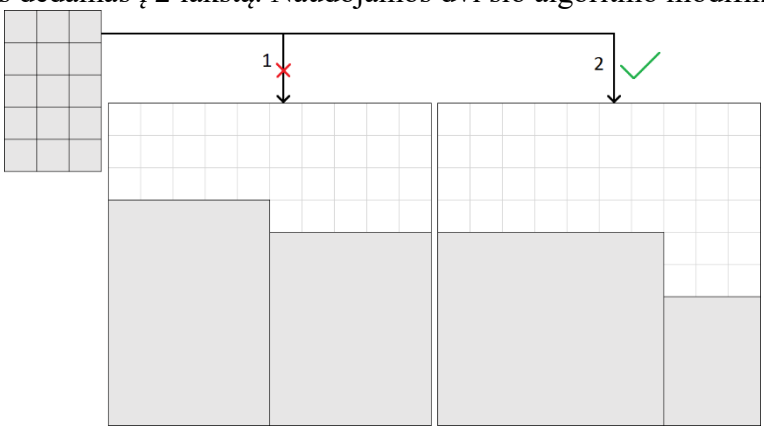
Evoliuciniai euristiniai algoritmai problemą nagrinėja ir iš kitos pusės. Jei euristiniai algoritmai operuoja duomenimis (objektais ir lakštais, į kuriuos šie objektai turi būti sudėti), tai evoliuciniai euristiniai algoritmai operuoja žemesnės eilės algoritmais (algoritmais, iš kurių reikia pasirinkti).

Siūlomas evoliucinis euristinis algoritmas remiasi kitų autorių mintimis [5, 15, 16, 17, 18]. Dauguma siūlomo evoliucinio euristinio algoritmo naudojamų euristinių algoritmų nėra nauji, tačiau pasirenkama specifinė algoritmų aibė, arba jie modifikuoti.

4.1.1. Naudojami euristiniai algoritmai

Vienmačiu objektų išdėliojimo uždavinio atveju euristikų tikslas yra išrinkti objektą ir talpyklą, į kurią šis objektas bus dedamas. Dvimačiu atveju prisideda papildomas kintamasis – reikia rasti poziciją, į kurią objektas bus dedamas lakšte. Todėl naudojami dviejų tipų euristiniai algoritmai: išrinkimo (išrenka objektą ir lakštą, į kurį objektas bus dedamas) ir dėjimo (išrenka poziciją lakšte, kur objektas bus dedamas). Naudojami pasirinkimo algoritmai:

1. Pirmo tinkančio (angl. *first fit*, 23 pav.) – atidarytus lakštus nagrinėja iš eilės, ieškant pirmo, į kurį tinka objektas. 23 pav. matome, kad lakštai nagrinėjami pradedant *1*. Į šį lakštą objektas netelpa, todėl jis dedamas į *2* lakštą. Naudojamos dvi šio algoritmo modifikacijos:

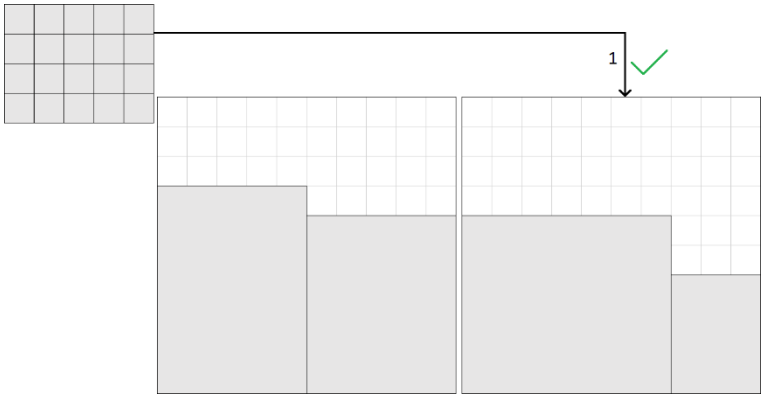


23 pav. Pirmo tinkančio algoritmas

1.1. Ploto mažėjimo seka – objektai nagrinėjami nuo didžiausio, iki mažiausio;

1.2. Su iš anksto paskirtais objektais [17] – visi objektai, kurie yra didesni nei pusė lakšto dydžio, iš karto išdėliojami į lakštus.

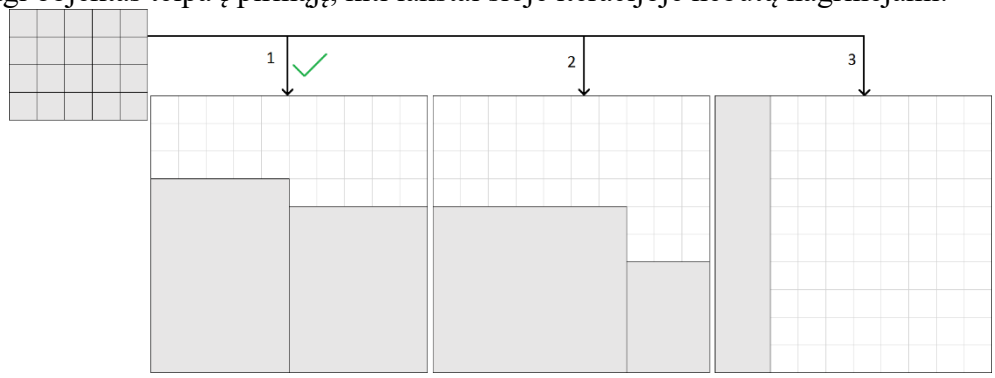
2. Kito tinkančio (angl. *next fit*, 24 pav.) – objekto dėjimui nagrinėjamas tik paskutinis naudotas lakštas. Jei objektas tinka, tuomet dedama į jį, priešingu atveju – imamas naujas lakštas. 24 pav. matome, kad pirmas lakštas net nebėra nagrinėjamas (nors objektas ir tilptų). Naudojamos dvi šio algoritmo modifikacijos:



24 pav. Kito tinkančio algoritmas

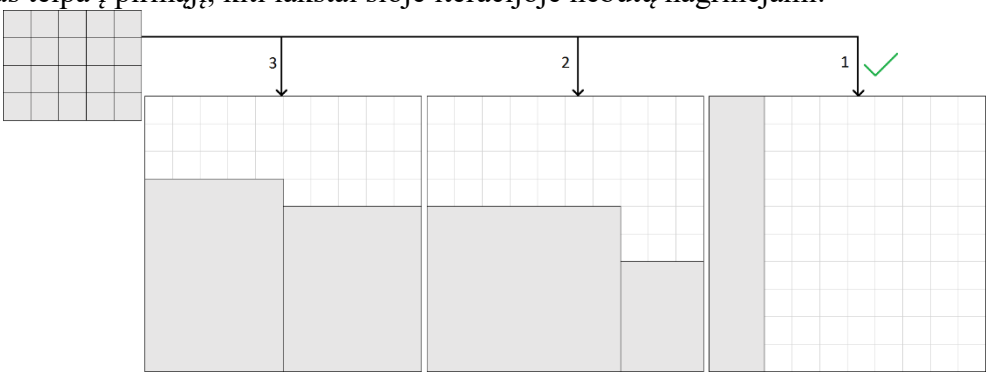
2.1. Standartinis – objektai ir lakštai nagrinėjami tokia seka, kokia yra gauti;
2.2. Su iš anksto paskirtais objektais.

3. Geriausiai tinkančio (angl. *best fit*, 25 pav.) – lakštai išrikiuojami pagal likusio laisvo ploto didėjimo seką. Šia seka jie nagrinėjami, siekiant rasti pirmą į kurį objektas telpa. Naudojama objektų ploto mažėjimo modifikacija. 25 pav. numeracija rodo, kuria seka lakštai nagrinėjami. Kadangi objektas telpa į pirmąjį, kiti lakštai šioje iteracijoje nebūtų nagrinėjami.



25 pav. Geriausiai tinkančio algoritmas

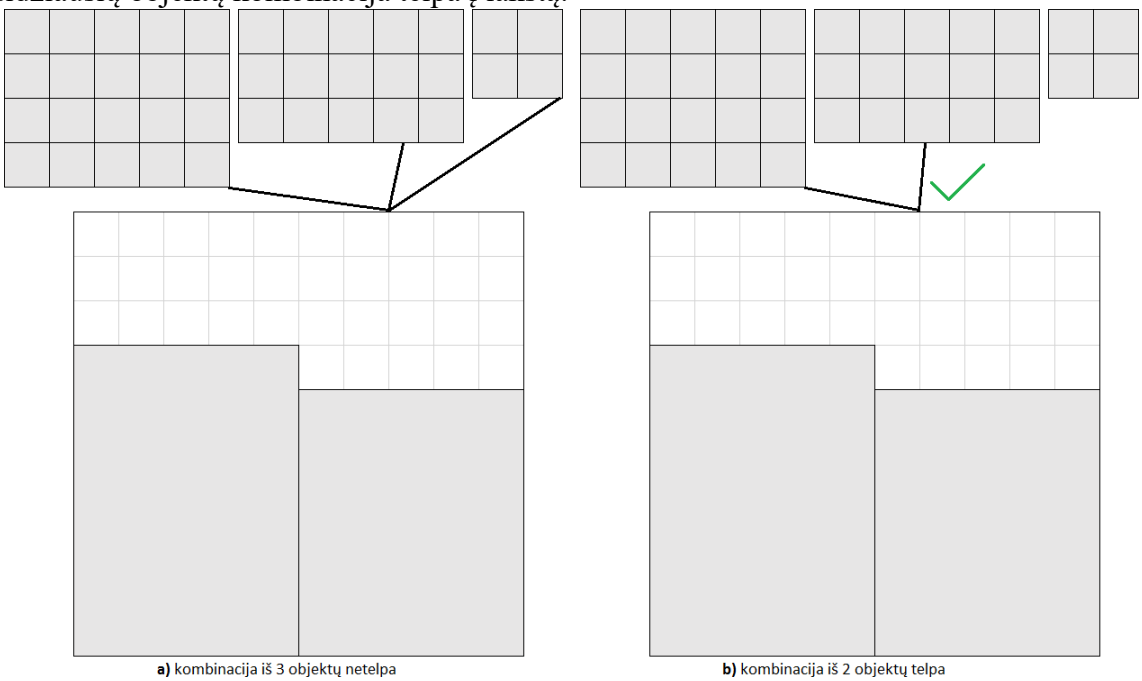
4. Blogiausiai tinkančio (angl. *worst fit*, 26 pav.) – lakštai išrikiuojamos pagal likusio laisvo ploto mažėjimo seką. Šia seka jie nagrinėjami, siekiant rasti pirmą į kurį telpa objektas (lakštai nagrinėjami priešinga seka, nei geriausiai tinkančio algoritme). Naudojama objektų ploto mažėjimo modifikacija. 26 pav. numeracija rodo, kuria seka lakštai nagrinėjami. Kadangi objektas telpa į pirmąjį, kiti lakštai šioje iteracijoje nebūtų nagrinėjami.



26 pav. Blogiausiai tinkančio algoritmas

5. Django ir Fitcho (27 pav.) – algoritmas užpildo 1/n lakšto dalį naudodamas pirmo tinkančio algoritmą. Tuomet jis ieško objektų iki trijų (gali naudoti 1, 2 arba 3) kombinacijos, kuri sudarytų didžiausią plotą ir dar tilptų į lakštą. Jei kelios kombinacijos turi vienodą plotą, pirmenybė teikiama tai, kuri turi mažiau elementų. Jei ir tada yra kelios kombinacijos, pirmenybė teikiama tai, kuri turi didžiausią objektą (jei didžiausias objektas sutampa, tada ta pati sąlyga galioja antram arba trečiam pagal dydį objektui). Jei nei viena kombinacija negali būti įdėta į lakštą –

imamas naujas. Naudojamos trys šio algoritmo modifikacijos (užpildymas iki 1/4, 1/3 ir 1/2 [18]). Šiame algoritme objektai nagrinėjami ploto mažėjimo seka. 27 pav. a dalyje matome, kad pirmoji kombinacija iš trijų objektų į lakštą netelpa. 27 pav. b dalyje matome, kad dviejų didžiausių objektų kombinacija telpa į lakštą.

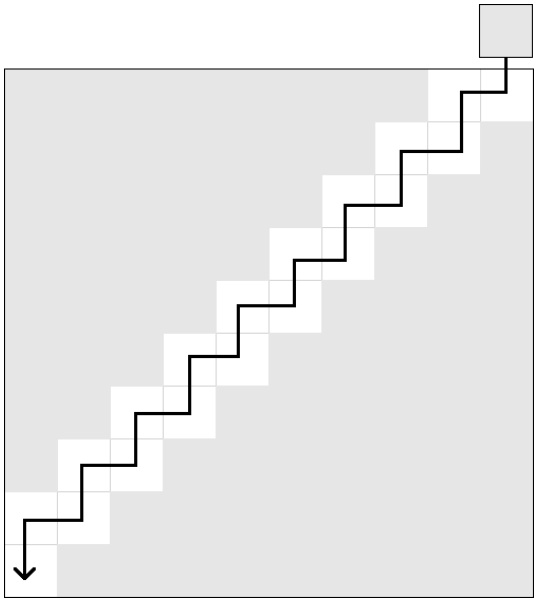


27 pav. Django ir Fitcho algoritmo kombinacijos išrinkimas

Django ir Fitcho algoritmas yra vienintelis, kuris vienu pasirinkimu gali išrinkti daugiau nei vieną objektą.

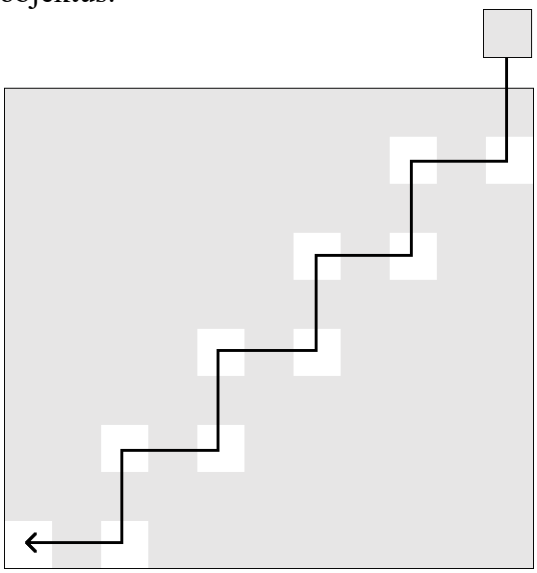
Objektų dėjimui pasirinkti algoritmai, kurie orientuoti į dėjimą apatiniame kairiajame kampe. Visi algoritmai orientuoti į panašų dėjimą, kad būtų išlaikytas sprendinio stabilumas (jei dalis algoritmų dėtų nuo kairiojo kampo, o dalis – nuo dešiniojo, tada viduryje gali likti tarpų, kurie bus nepatogios formos ir kuriuos bus sunku užpildyti). Naudojami dėjimo algoritmai:

1. Apatinio kairiojo kampo (28 pav.). Objektas įdedamas į viršutinį dešinįjį kampą. Tuomet, naudojant stūmimo žemyn ir kairėn veiksmus, objektas stumiamas tol, kol pasiekia kitą objektą arba lakšto kraštą. 28 pav. matome, kaip objektas gali būtų stumiamas nuo viršutinio dešiniojo į apatinį kairįjį kampą.



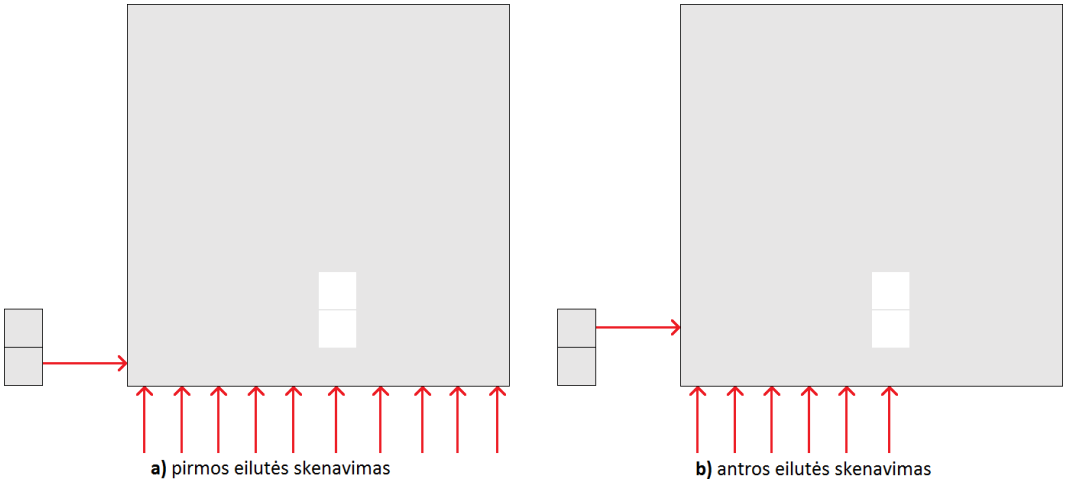
28 pav. Apatinio kairiojo kampo algoritmas

2. Patobulinto apatinio kairiojo kampo (29 pav.). Objektas, kaip ir apatinio kairiojo kampo algoritme, įdedamas į viršutinį dešinįjį kampą. Tuomet, skirtingai nuo apatinio kairiojo kampo algoritmo, objektas stumiamas ne tik iki artimiausio objekto krašto. Pristūmus objektą iki kito objekto procesas nėra nutraukiamas – tikrinama, ar už šio yra laisvos vietos, į kurią naujas objektas tilptų. Jei ši vieta randama – objektas perkeliamas per jau egzistuojantį objektą. 29 pav. matome, kaip objektas gali būtų stumiamas nuo viršutinio dešiniojo į apatinį kairįjį kampą peršokant per jau įdėtus objektus.



29 pav. Patobulintas apatinio kairiojo kampo algoritmas

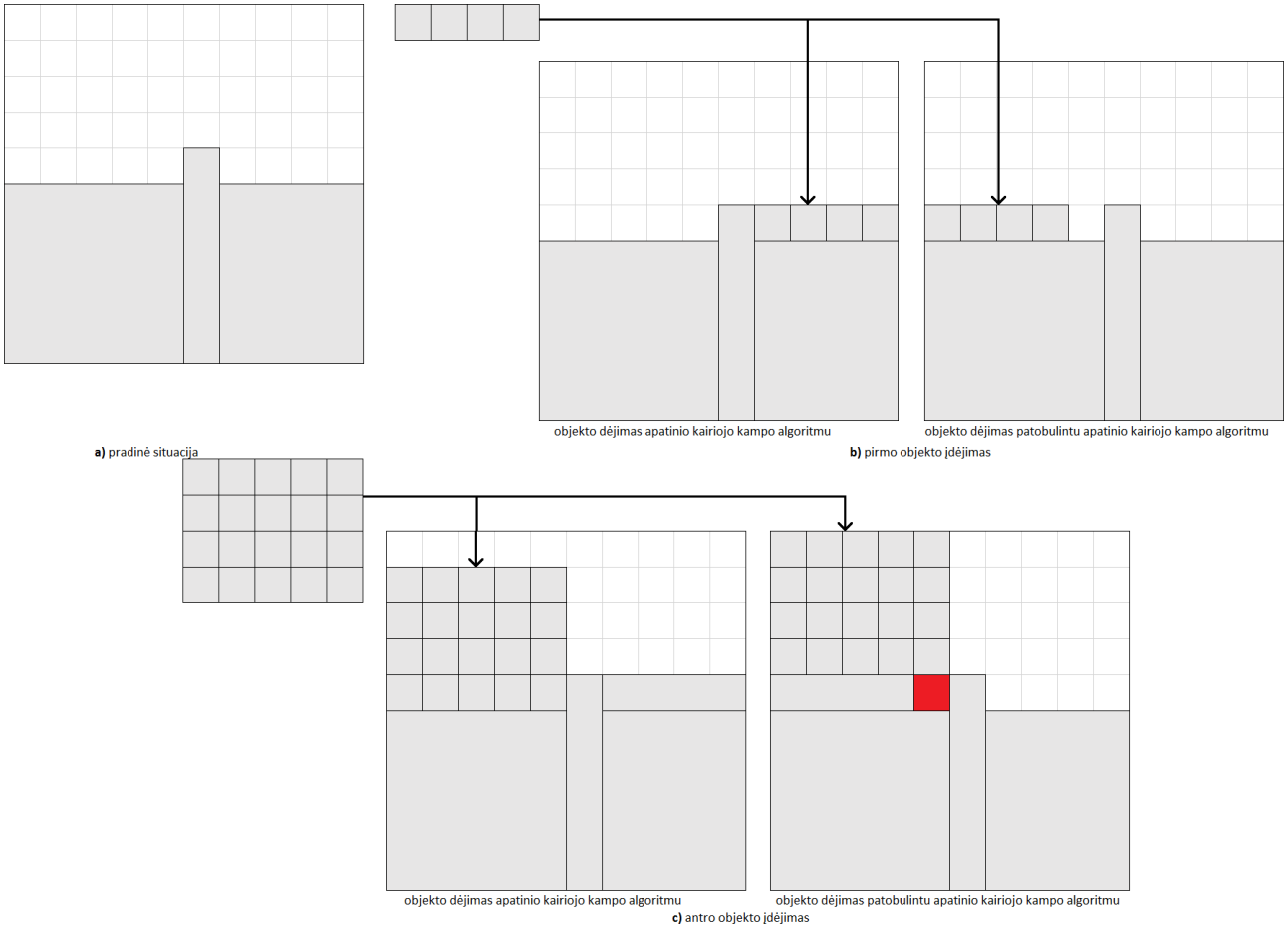
3. Skenavimo (30 pav.). Šio algoritmo tikslas – skenuoti visas pozicijas iš kairės į dešinę bei iš apačios į viršų ir ieškoti pirmos, kurioje gali būti padėtas apatinis kairysis objekto kampas. 30 pav. a dalyje matome pirmos eilutės skenavimą. Šioje eilutėje nėra tinkamos pozicijos objektui įdėti. 30 pav. b dalyje matome antros eilutės skenavimą. Šioje eilutėje yra pozicija, į kurią objekto apatinis kairysis kampas gali būti įdėtas. Pasiekus šią poziciją paieška stabdoma.



30 pav. Skenavimo algoritmas

Taip iš viso gautos 27 algoritmų kombinacijos (9 pasirinkimo ir 3 dėjimo algoritmai).

Gali iškilti klausimas, ar reikalingas apatinio kairiojo kampo algoritmas, kai turime patobulintą apatinio kairiojo kampo algoritmą? Vieną iš atvejų, kai apatinio kairiojo kampo algoritmas gauna geresnius rezultatus nei patobulintas apatinio kairiojo kampo algoritmas, pateiktas 31 pav. Šio paveikslo a dalyje nurodyta pradinė situacija. 31 pav. b dalyje pateiktas pirmo objekto įdėjimas. Pirmasis abiejų algoritmų žingsnis vienodas – jie objektą įdeda į tą vietą, į kurią objektą įdeda apatinio kairiojo kampo algoritmas. Tuomet patobulintas apatinio kairiojo kampo algoritmas randa laisvos vietos už išsikišusio objekto ir perkelia dedamą objektą į šią laisvą vietą. 31 pav. c dalyje pateiktas antro objekto įdėjimas. Atėjus didesniam objektui, apatinio kairiojo kampo algoritmas turi vietos objektui įdėti. Kita vertus, patobulintas apatinio kairiojo kampo algoritmas nebeturi tokios geros vietos objektui įdėti, todėl sukuriamas raudona spalva pažymėtas plyšys, kurį gali būti labai sunku užpildyti. Būtent dėl tokių atvejų reikalingi abu objektų įdėjimo algoritmai.



31 pav. Apatinio kairiojo kampo ir patobulinto apatinio kairiojo kampo algoritmų palyginimas

4.1.2. Evoliucinis euristinis algoritmas

Evoliucinį euristinį algoritmą galima aprašyti taip (32 pav.):

1. Paruošiama 20 uždavinio kopijų.
2. Kiekvienai kopijai nustatoma dabartinė uždavinio būsena.
3. Jei rastai būsenai jau yra pasirinkta algoritmų kombinacija – ji ir naudojama. Priešingu atveju pasirenkama atsitiktinė kombinacija iš anksčiau aprašytų.
4. Šis procesas kartojamas tol, kol sudedami visi objektai.
5. Įvertinamas kiekvieno iš gautų sprendinių tinkamumas (angl. *fitness*).
6. Naudojant genetinį algoritmą, išrenkama dalis geriausią sprendinį radusių algoritmų kombinacijų:
   6.1. Išrenkama 80 % algoritmų kombinacijų. Kombinacijos renkamos ruletės principu. Kiekvienam variantui, atsižvelgiant į jo tinkamumą, skiriamas intervalas intervale [0; 1). Tada generuojamas atsitiktinis skaičius šiame intervale. Pasirenkama algoritmų kombinacija, kuriai priskirtame intervale yra sugeneruotas skaičius.
   6.2. Atliekama rekombinacija. Iš algoritmų kombinacijų sąrašo pasirenkamos dvi atsitiktinės ir su 50 % atliekama rekombinacija. Jei rekombinacija įvyksta, tuomet atsitiktinėje vietoje sumaišomi naudoti algoritmai, jei ne – kombinacijos perduodamos tokios, kokios yra.
   6.3. Su 10 % tikimybe atliekama mutacija. Įvykus mutacijai, 20 % atsitiktinių algoritmų kombinacijų išmetama.
7. Šis procesas kartojamas tol, kol uždavinio sprendinys yra nepagerinamas 10 iteracijų.

```
while Uždavinio sprendinys nepagerinamas 10 kartų
    Paruošti 20 uždavinio kopijų
    foreach kopija in kopijos
        while Ne visi daiktai sudėlioti
            Nustatyti dabartinę uždavinio būseną
            if Būsena, kuriai algoritmas jau pasirinktas
                Naudoti jau pasirinktą algoritmą
            else
                Atsitiktinai išsirinkti naują algoritmą
    Apskaičiuoti kiekvieno sprendinio tinkamumą T
    Geriausių individų išrinkimas (žr. 6.1)
    Geriausių individų rekombinacija (žr 6.2)
    Atliekama atsitiktinių individų mutacija (žr. 6.3)
```

32 pav. Siūlomas evoliucinis euristinis algoritmas

Sprendinio tinkamumas T apskaičiuojamas taip:

$$T = \frac{\sum_{i=1}^{M}\left(\frac{\sum_{j=1}^{N} s_j}{N}\right)_i^2}{M}; \quad (1)$$

čia M – panaudotų dėžučių kiekis uždaviniui spręsti; N – išdėliotų objektų kiekis; S_j – vieno objekto plotas.

Uždavinio būsena koduojama eilute, kurią sudaro 23 simboliai:

1. Pirmi 6 simboliai yra skirti likusių objektų aukščiui identifikuoti. Pagal objekto aukščio santykį su lakšto aukščiu objektai sugrupuojami į tris grupes: (0; 1/3], (1/3; 1/2] ir (1/2; 1].
2. Kiti 6 (nuo 7 iki 12) simboliai yra skirti likusių objektų pločiui identifikuoti. Pagal objekto pločio santykį su lakšto pločiu objektai sugrupuojami į tris grupes: (0; 1/3], (1/3; 1/2] ir (1/2; 1].
3. Kiti 8 (nuo 13 iki 20) simboliai yra skirti likusių objektų plotui identifikuoti. Pagal objekto ploto santykį su lakšto plotu objektai sugrupuojami į 4 grupes: (0; 1/4], (1/4; 1/3], (1/3; 1/2] ir (1/2; 1].
4. Paskutiniai 3 (nuo 21 iki 23) simboliai skirti likusių neįdėtų objektų kiekio santykiui su pradiniu objektų kiekiu identifikuoti.

Kiekvienas iš šių identifikatorių gauna reikšmę pagal jo atitikimą vienam iš intervalų procentais. Pirmoms trims grupėms šifruoti naudojamas 2 simbolių kodavimas, kur „00“ atitinka intervalą [0; 10], „01“ – (10; 25], „10“ – (25; 50], „11“ – (50; 100]. Likusių objektų šifravimui naudojamas 3 simbolių kodavimas, kur „000“ atitinka intervalą [0; 12,5], „001“ – (12,5; 25], „010“ – (25; 37,5], „011“ – (37,5; 50], „100“ – (50; 62,5], „101“ – (62,5; 75], „110“ – (75; 87,5], „111“ – (87,5; 100].

4.1.3. Uždavinio lygiagretinimas

Kadangi didžiąją sprendimo laiko dalį sudaro tinkamos pozicijos lakšte ieškojimas, todėl nuspręsta būtent šią dalį realizuoti naudojant CUDA technologijas. Tam visi 3 objektų dėjimo algoritmai pritaikyti lygiagretiems skaičiavimams CUDA.

Apatinio kairiojo kampo ir patobulintą apatinio kairiojo kampo algoritmus galima lygiagrečiai vykdyti tokiame gijų skaičiuje, kiek objektų tuo metu yra lakšte. Stūmimo žemyn žingsnyje kiekviena gija pirmiausia identifikuoja, ar objektas aktualus paieškai (bent dalis objekto turi būti po dedamu objektu). Tada kiekviena gija pažymi jos tikrinamo objekto užimamą aukščio intervalą (šiame intervale jau yra objektas, todėl naujas nebegali būti dedamas). Galiausiai, viena gija (esant dideliems objekto ir lakšto matmenų skirtumams, šis veiksmas tai pat padalinamas kelioms gijoms po lygiai) peržiūri pažymėtą informaciją ir patikrina, ar yra nepažymėtas reikiamo dydžio intervalas. Atitinkami veiksmai vykdomi ir objektą stumiant į kairę.

Skenavimo algoritmo įgyvendinimas šiek tiek skiriasi. Čia gijų kiekis pasirinktas pagal lakšto aukštį. Kiekviena gija nagrinėja galimybę, kad apatiniu kairiuoju objekto dėjimo kampu bus viena iš jos nagrinėjamos eilutės pozicijų. Tinkamo taško eilutėje radimui taikomas toks pat principas kaip ir vietai, į kurią objektas dedamas apatinio kairiojo kampo ir patobulinto apatinio kairiojo kampo algoritmuose, nustatyti.

4.2. Programinės įrangos kokybės užtikrinimas

Šiame skyrelyje apžvelgiamos priemonės, kurių buvo imtasi norint užtikrinti sukurtos PĮ kokybę. Taip pat apžvelgiami kokybės užtikrinimo priemonių pasiekti rezultatai.

4.2.1. Programinės įrangos kokybės užtikrinimo priemonės

Jau kuris laikas kuriamos sistemos tiek savo dydžiu, tiek sudėtingumu pasiekė tokį lygį, kad praktiškai neįmanoma jų sukurti be klaidų. Norint pateikti aukštos kokybės produktą, jo veikimą reikia patikrinti dar prieš atiduodant vartotojui. Šiuos tikslus pasiekti padeda testavimas. Gerai ištestuotas produktas bus kokybiškesnis ir sunkiau pažeidžiamas.

Egzistuojantys testai apsaugo nuo netyčinių klaidų modifikuojant kuriamą PĮ ar jos komponentą. Dažnai pasitaiko, kad modifikuojant vieną kuriamos PĮ komponentą, gali nugriūti, atrodytų, visiškai nesusijęs komponentas. Geri testai tuojau pat parodytų tokį netyčinį sugriovimą ir nurodytų, kur ieškoti klaidos.

Gerai paruošti testai puikiai tinka kaip kuriamos PĮ ar komponento dokumentacija. Tokios PĮ kūrimo metodologijos kaip testavimų grįstas kūrimas (angl. *test driven development,* toliau – TDD) iš viso atsisako projektavimo ir naudoja testus vietoj projektinės dokumentacijos. Tokie testai atspindi PA ir užsakovui yra pakankamai lengvai suprantami.

PĮ bus kuriama naudojant TDD, t. y., bus parašomi testai kiekvienam metodui ir tik po to rašomas pats metodas. Toks testavimo metodas pasirinktas todėl, kad jis leidžia išgauti geresnę architektūrą. Taip pat reiktų paminėti, kad kuriama PĮ realizuoja matematinius algoritmus, kuriuos stengiamasi optimizuoti. Testais padengtą programinį kodą kur kas lengviau ir saugiau optimizuoti nei testais nepadengtą programinį kodą.

Kūrimo metu bus naudojamas statinės analizės įrankis. Statinės analizės įrankiai padeda aptikti neteisingą naudojamo karkaso elementų panaudojimą, parašytą, bet nenaudojamą ar neprieinamą kodą, kintamuosius. Šio įrankio naudojimo tikslas – išlaikyti aukštą parašyto programinio kodo kokybę.

4.2.2. Kokybės užtikrinimo rezultatai

Kuriant PĮ, buvo parašyti 406 testai. Šie testai padengė daugiau nei 95 % programinio kodo. Testais nepadengtas tik DB modelio sukūrimas pagal vartotojo pateiktus duomenis. Ši dalis nebuvo padengta testais, nes modelio saugojimo objektas nėra pritaikytas testavimui.

Taip pat dalis kodo buvo išmesta iš kodo padengimo skaičiavimo. Šią dalį sudaro:

1. Objekto duomenų atlaisvinimo (angl. *dispose*) metodai, nes jie atsakingi už privačių laukų atlaisvinimą. Kadangi testai negali prieiti prie privačių laukų, tai atlaisvinimo procesas negali būti ištestuotas. Šių metodų teisingumu rūpinasi statinė kodo analizė.
2. „Model“ klasės savybės ir metodai, nes jie atsakingi už tiesioginį bendravimą su DB. Šių elementų testavimas apima integracinį, o ne vienetų testavimą.
3. Naudojamos vidinės išimtys, nes naudojamas paveldėjimas iš standartinio išimties tipo ir neturima jokio nuosavo kodo. Vadinasi, šių klasių testavimas yra nereikalingas.
4. Kodas, kuris yra pergeneruojamas į GPU programinį kodą. Šis kodas nėra vykdomas tiesiogiai (vykdoma GPU kodo versija), todėl šios dalies padengimo analizė yra neįmanoma. Žinoma, šiai kodo daliai testai yra parašyti – tiesiog neįmanoma apskaičiuoti šių testų padengimo.

Statinė analizė papildomų klaidų nerado. Į išimčių sąrašą pridėtas tik vienas pranešimas: „CA1062:Validate arguments of public methods“. Ši išimtis padaryta atsižvelgiant į testų gautą rezultatą. Metodas, kuriam priskirta ši išimtis, dėl klasių paveldėjimo patikrinimą atlieka tėvinėje klasėje ir konkrečioje realizacijoje tampa perteklinis. Vadinasi, šis pranešimas yra klaidingas teigimas (angl. *false positive*).

5. EKSPERIMENTINĖ DALIS

Šiame skyriuje aprašomi atlikti eksperimentai ir galimi PĮ patobulinimai.

5.1. Atlikti eksperimentai

Eksperimento tikslas – palyginti pasiūlyto evoliucinio euristinio modifikavimo tipo algoritmo su jau egzistuojančio evoliucinio euristinio algoritmo veikimo charakteristikas. Bus lyginamas rasto sprendinio tikslumas ir laikas, per kurį sprendinys gautas.

Algoritmas realizuotas naudojant „.NET 4.6“ technologijas bei „Cudafy.NET“ biblioteką „.NET“ kodui transformuoti į CUDA kodą. Tyrimai daryti nešiojamame kompiuteryje, kuris turi Intel® Core™ i7 4710HQ 2,5 GHz procesorių, 8 GB darbinės atminties (RAM) ir Nvidia GeForce GTX 850M GPU. Palyginimui pasirinktas C. Bluma ir V. Schmido pasiūlytas algoritmas [15]. Šis algoritmas šiuo metu pasiekia vienus iš geriausių rezultatų. Pasirinkti pirmų penkių klasių (I–V) duomenys. Informacija apie šias duomenų klases pateikta 2 lentelėje. Kiekvienoje klasėje objektų kiekis yra iš aibės {20, 40, 60, 80, 100}. Kiekvienai klasei ir kiekvienam objektų kiekiui pateikiama po 10 duomenų rinkinių. Taip iš viso gaunama 250 duomenų rinkinių algoritmams palyginti.

2 lentelė. Duomenų rinkinių klasės

Klasė	Objekto plotis	Objekto aukštis	Lakšto plotis	Lakšto aukštis
I	[1; 10]	[1; 10]	10	10
II	[1; 10]	[1; 10]	30	30
III	[1; 35]	[1; 35]	40	40
IV	[1; 35]	[1; 35]	100	100
V	[1; 100]	[1; 100]	100	100

Žinoma, negalima tiesiogiai palyginti C. Bluma ir V. Schmido gautų sprendimo laikų su šiame darbe pasiūlyto evoliucinio euristinio algoritmo sprendimo laikais, kadangi kompiuterių parametrai labai skyrėsi. Šie laikai pasirinkti kaip atskaitos taškas, kurį CUDA technologijomis realizuotas algoritmas turėtų įveikti.

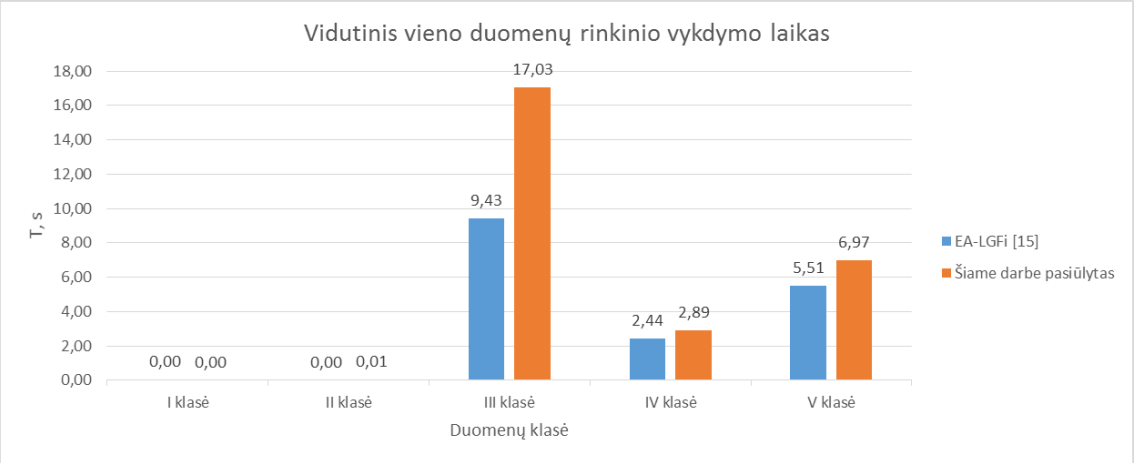
Rezultatų patikimumui užtikrinti buvo imtasi kelių priemonių. Visų pirma, naudojamas kompiuteris buvo atjungtas nuo interneto. Taip pat buvo išjungtos visos programos ir procesai, kurių nereikia normaliam operacinės sistemos darbui užtikrinti. Galiausiai, skaičiavimai su kiekvienu duomenų rinkiniu buvo vykdomi bent po 2 kartus. Jei abiejų vykdymų laikai buvo panašūs – imamas šių laikų vidurkis. Priešingu atveju skaičiavimai su duomenų rinkiniu buvo vykdomi trečią kartą. Tada buvo atsisakoma labiausiai nutolusio rezultato ir pateikiamas 2 artimesnių rezultatų vidurkis.

Gauti rezultatai apibendrinti 3 lentelėje. Kiekvienoje eilutėje pateikiami 10 duomenų rinkinių jungtiniai rezultatai. Pirmame stulpelyje pateikiamas objektų kiekis klasėje, antrame – geriausias iki šiol žinomas šių 10 duomenų rinkinių sprendimo rezultatas (LB, angl. *lower bound*). Rezultatų palyginimas pateiktas dviem stulpeliais: pirmajame pateikiamas algoritmo gautas rezultatas, o antrajame – laikas, per kurį rezultatas pasiektas. Paskutinėje eilutėje pateikiama visų reikalingų lakštų kiekių suma ir vidutis vieno duomenų rinkinio vykdymo laikas.

3 lentelė. Rezultatų palyginimas

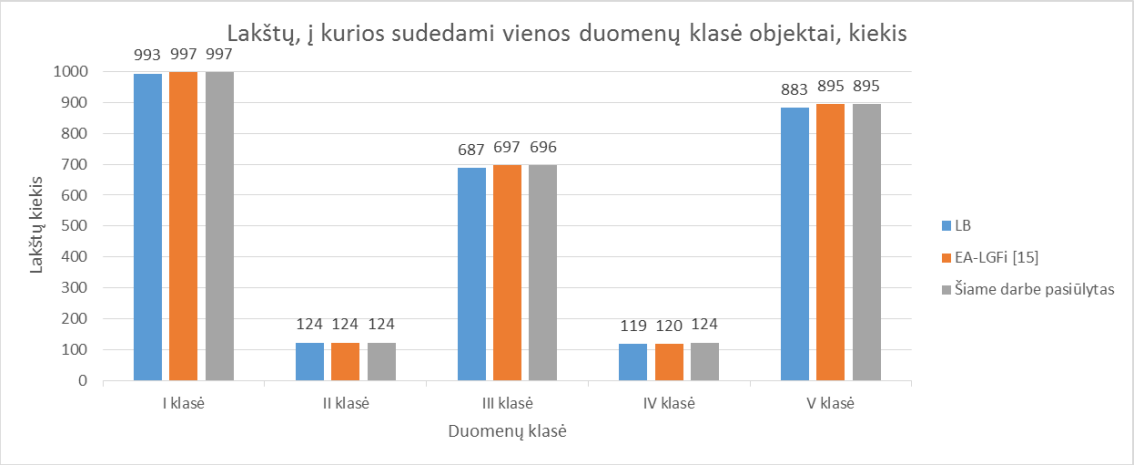
Uždavinys	LB	EA-LGFi [15]		Šiame darbe pasiūlytas	
		Rezultatas	Laikas (s)	Rezultatas	Laikas (s)
I klasė					
20	71	71	0,00	71	0,00
40	134	134	0,00	134	0,00
60	197	200	0,01	200	0,00
80	274	275	0,00	275	0,00
100	317	317	0,00	317	0,00
II klasė					
20	10	10	0,00	10	0,01
40	19	19	0,00	19	0,01
60	25	25	0,00	25	0,01
80	31	31	0,00	31	0,01
100	39	39	0,00	39	0,01
III klasė					
20	51	51	0,02	51	0,04
40	92	94	0,01	94	0,05
60	136	139	0,27	139	1,02
80	187	189	20,68	189	39,85
100	221	224	26,17	223	44,21
IV klasė					
20	10	10	0,00	10	0,02
40	19	19	0,00	19	0,02
60	23	23	12,18	25	0,03
80	30	31	0,00	32	14,36
100	37	37	0,00	38	0,04
V klasė					
20	65	65	0,00	65	0,01
40	119	119	0,03	119	0,04
60	179	180	0,14	180	0,17
80	241	247	0,03	247	0,03
100	279	284	27,33	284	34,62
Rezultatai	2806	2833	3,47	2836	5,39

33 pav. pateiktas vidutinis vieno duomenų rinkinio vykdymo laikas pagal duomenų klases. Kaip matome, nei vienoje duomenų klasėje šiame darbe pasiūlytas evoliucinis euristinis algoritmas neaplenkia C. Bluma ir V. Schmido algoritmo.



33 pav. Vidutinis vienos duomenų klasės vykdymo laikas

34 pav. pateiktas lakštų, į kuriuos sudedami vienos duomenų klasės objektai, kiekis. Galima pastebėti, kad šiame darbe pasiūlytas algoritmas C. Bluma ir V. Schmido algoritmui nusileidžia tik IV klasėje. I, II ir V duomenų klasėse abu algoritmai objektus sudėlioja į tokį patį lakštų kiekį. III duomenų klasėje šiame darbe pasiūlytas algoritmas objektus išdėlioja netgi geriau nei C. Bluma ir V. Schmido pasiūlytas algoritmas.



34 pav. Lakštų, į kuriuos sudedami vienos duomenų klasės objektai, kiekis

Palyginus rezultatus galima pastebėti, kad CUDA vykdyti skaičiavimai nusileidžia C. Bluma ir V. Schmido gautiems rezultatams. Pirmiausia, tai privertė suabejoti realizacijos korektiškumu. Dėl to buvo atliktas papildomas realizuotos programos tyrimas naudojant „ANTS Performance Profiler“. Šis įrankis leidžia išanalizuoti, kurios programos vietos vykdomos ilgiausiai. Atlikus šią analizę pastebėta, kad apie 60 % laiko programa praleidžia siųsdama duomenis tarp CPU ir GPU. Tai visiškai neefektyvu. Jei atmestumėme šiuos 60 % laiko, tada vidutinis vykdymo laikas nuo 5,39 s nukristų iki 2,16 s. Iš šių rezultatų galima spręsti, kad reikia tobulinti algoritmo realizaciją.

Jei atkreiptumėme dėmesį į evoliucinio euristinio algoritmo gaunamų rezultatų tikslumą, pastebėtumėme, kad jis C. Bluma ir V. Schmido algoritmui nusileidžia tik 3 lakštais – jų algoritmas objektus sudėlioja į 2833 lakštus, o šiame darbe pasiūlytas – į 2836. Tai rodo, kad pats evoliucinis euristinis algoritmas yra geras, tačiau jo realizacija nėra optimali.

5.2. Patobulinimų galimybės

Patobulinimų galimybes galime išskirti į 2 dalis:

1. Funkcionalumo patobulinimo galimybės. Tai naujas funkcionalumas, kuriuo PĮ šiuo metu nepasižymi.
2. Realizacijos patobulinimai. Su realizacija susiję pakeitimai, kurie turėtų pagerinti sistemos greičio charakteristiką.

5.2.1. Funkcionalumo patobulinimas

Šiame darbe nagrinėjamas dvimatis objektų išdėliojimo atvejis, kai objektai yra orientuoti stačiakampiai, negali būti vartomi, objektų sąrašas žinomas iš anksto ir lakštai yra vienodo dydžio. Uždavinio formuluotė ir apribojimai puikiai atspindi, kur galima tobulinti PĮ:

1. Leisti sukinėti objektus;
2. Leisti laisvą objektų formą;
3. Lesti gauti objektus po vieną;
4. Leisti skirtingo dydžio lakštus;
5. Leisti papildomus apribojimus (pvz., nepertraukiamas lakšto pjovimas, apsauginių zonų aplink objektus palikimas);
6. Ne tik dvimačio uždavinio sprendimas.

Nors kai kurie pakeitimai, pvz., apsauginių zonų aplink objektus palikimas, reikalautų tik PĮ pakeitimų, tačiau yra ir tokių, dėl kurių reiktų peržiūrėti visą evoliucinį euristinį algoritmą. Pavyzdžiui, leidimas turėti skirtingo dydžio lakštus ir šiuo metu naudojamas išrinkimo algoritmas su iš anksto paskirtais objektais. Jei objektas sudaro daugiau nei pusę vieno lakšto ploto, tai nereiškia, kad į šį lakštą objektą ir reikėtų dėti. Galbūt yra kitas lakštas, kurį objektas užpildo visu 100 %. Vadinasi, ne visi galimi patobulinimai yra lengvai įgyvendinami.

Šiuo metu kuriama tik objektų dėliojimo uždavinį sprendžianti biblioteka. T. y., nėra skiriamas didelis dėmesys vartotojo sąsajai. Buvo sukurta tik demonstracinio tipo vartotojo sąsaja pristatymui. Jei būtų norima šią PĮ naudoti, jai reikėtų vartotojo sąsajos arba integracijos į jau esamą produktą.

5.2.2. Realizacijos patobulinimas

Visų pirma, reikėtų ne tik dėjimo, bet ir išrinkimo euristinius algoritmus pritaikyti skaičiavimams CUDA. Taip pat reikėtų ir patį evoliucinį euristinį algoritmą pritaikyti skaičiavimams CUDA. Tuomet vienintelis bendravimas tarp CPU ir GPU būtų pradinių duomenų perdavimas ir rezultatų pasiėmimas.

Reikėtų atsisakyti „Cudafy.NET“ bibliotekos naudojimo PĮ transformavimui į CUDA suprantamą programinį kodą. Kuriant PĮ kelis kartus buvo susidurta su situacija, kai ši biblioteka sugeneruoja neoptimalų ar net neteisingą programinį kodą. Pastebėti neteisingai sugeneruotą programinį kodą leido iš anksto pasirašyti testai. Jie parodė, kad tas pats kodas, veikdamas CPU ir GPU, gauna skirtingus rezultatus.

Jei bus atsisakoma „Cudafy.NET“ bibliotekos naudojimo, tuomet reikės naudoti tiesioginę CUDA teikiamą sąsają, kuri yra pritaikyta „C++“ programavimo kalbai. Tokiu atveju reikėtų arba perrašyti kuriamą PĮ naudojant „C++“, o ne „.NET“ technologiją, arba naudoti „C++/CLI“.

6. IŠVADOS

1. Išanalizavus literatūroje rastus objektų išdėliojimo uždavinį sprendžiančius algoritmus nustatyta, kad geriausius rezultatus su optimaliomis laiko sąnaudomis pasiekia evoliuciniai euristiniai algoritmai. Taip pat buvo pastebėta, kad evoliucinis euristinis modifikavimo tipo algoritmas geriau prisitaiko prie nestandartinių uždavinių. Dėl šių priežasčių buvo nuspręsta suformuluoti evoliucinį euristinį modifikavimo tipo algoritmą.
2. Išanalizavus literatūrą pastebėta, kad labai mažai dėmesio kreipiama objektų išdėliojimo uždavinio sprendimui lygiagrečiai. Nors buvo atlikti keli bandymai, tačiau nei vienas iš jų nenaudojo evoliucinio euristinio algoritmo. Dėl to buvo nuspręsta pritaikyti siūlomą evoliucinį euristinį algoritmą lygiagretiems skaičiavimams CUDA.
3. Suprojektavus ir įgyvendinus PĮ galime pastebėti, kad ji yra kokybiška. PĮ kokybę atspindi jos padengimas testais (95 %) bei klaidų nerandanti statinė analizė. Tai leidžia tikėtis, kad sistema veiks be didelių klaidų. Taip pat aukštos kokybės PĮ bus lengva palaikyti ir tobulinti.
4. Atlikus eksperimentą ir palyginus gautus rezultatus su jau egzistuojančių evoliucinių euristinių algoritmų gaunamais rezultatais galime pastebėti, kad realizuotas algoritmas pasiekia panašius rezultatus. Nuo šiuo metu geriausius rezultatus pasiekiančio C. Bluma ir V. Schmido algoritmo atsiliekama tik 3 lakštais (C. Bluma ir V. Schmido algoritmas objektus sudėlioja į 2833 lakštus, o šiame darbe pasiūlytas – į 2836).
5. Atlikus eksperimentą pastebėta, kad skaičiavimai trunka ilgiau nei kitų evoliucinių euristinių algoritmų. Šiame darbe pasiūlytas algoritmas vieną duomenų rinkinį vidutiniškai dėlioja 5,39 s, o C. Bluma ir V. Schmido algoritmas – 2,16 s. Reikėtų nepamiršti, kad eksperimentai su C. Bluma ir V. Schmido algoritmu buvo atliekami kur kas prastesniame kompiuteryje. Tai verčia suabejoti realizacijos korektiškumu.
6. Atlikus papildomą sukurtos PĮ analizę naudojant profiliavimo įrankį pastebėta, kad didžioji darbo laiko dalis sugaištama perduodant duomenis tarp CPU ir GPU (apie 60 %). Tai rodo, kad pasirinkta algoritmo realizacija nėra optimali.
7. Šia tema buvo parašytas straipsnis (1 priedas). Jis buvo pristatytas XXI tarpuniversitetinėje tarptautinėje magistrantų ir doktorantų konferencijoje „Informacinė visuomenė ir universitetinės studijos“ (IVUS, 2016) 2016 m. balandžio 28 d.

7. LITERATŪRA

[1] KANTOROVICH, L. V. Mathematical Methods of Organizing and Planning Productions. *Management Science*, 1960, vol. 6, no. 4, p. 366–422.

[2] GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman and Company, 1979.

[3] BURCEA, M.; WONG, P. W. H.; YUNG, F. C. C. *Online Multi-dimensional Dynamic Bin Packing of Unit-Fraction Items.* Liverpool: Department of Computer Science, University of Liverpool, 2013.

[4] QUILLIOT, A.; TOUSSAINT, H. About Casting 2D-Bin Packing into Network Flow Theory. In *HAL* [interaktyvus]. 2010 [žiūrėta 2016-05-13]. Prieiga per internetą: <https://hal.archives-ouvertes.fr/hal-00679046/document>

[5] TERASHIMA-MARIN, H.; FARÍAS-ZÁRATE, C. J.; ROSS, P. M.; VALENZUELA-RENDÓN, M. Comparing Two Models to Generate Hyper-heuristics for the 2D-Regular Bin-Packing Problem. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York: ACM, 2007, p. 2182–2189.

[6] MARTINEZ-SYKORA, A.; ALVAREZ-VALDES, R.; BENNELL, J.; TAMARIT, J. M. Constructive procedures to solve 2-dimensional bin packing problems. *Omega*, 2015, vol. 12, no. 1, p. 15–32.

[7] HARREN, R.; STEE, R. van; JANSEN, K.; PRÄDEL, L.; SCHWARZ, U. M. *Two for One: Tight approximation of 2D Bin Packing*. Kiel: Universität Kiel, Institut für Informatik, 2013.

[8] BANG-JENSEN, J.; LARSEN, R. Efficient algorithms for real-life instances of the variable size bin packing problem. *Computers & Operations Research*, 2012, vol. 39, no. 11, p. 2848–2857.

[9] CUI, Y.; CUI, Y.; TANG, T. Sequential heuristic for the two-dimensional bin-packing problem. *European Journal of Operational Research*, 2015, vol. 240, no. 1, p. 43–53.

[10] MUMFORD, C. L.; WANG, P. Y. Cutting, Packing and VLSI Layout [interaktyvus]. 2011 [žiūrėta 2016-04-15]. Prieiga per internetą: <http://users.cs.cf.ac.uk/C.L.Mumford/Research%20Topics/layout/Outline.html>

[11] KORF, R. E. A New Algorithm for Optimal Bin Packing. In *Proceedings of the 18th National Conference on Artificial intelligence (AAAI 2002). US,* 2002, p. 731–736.

[12] ZHANG, Z.; GUO, S.; ZHU, W.; OON, W. C.; LIM, A. Space Defragmentation Heuristic for 2D and 3D Bin Packing Problems. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence.* Palo Alto, 2011.

[13] BRANDÃO, F.; PEDROSO, J. P. Fast Pattern-based Algorithms for Cutting Stock. *Computers & Operations Research*, 2014, vol. 48, no. 1, p. 69–80.

[14] CHARALAMBOUS, C.; FLESZAR, K. A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts. *Computers & Operations Research*, 2011, vol. 38, no. 10, p. 1443–1451.

[15] BLUMA, C.; SCHMIDC, V. Solving the 2D Bin Packing Problem by Means of a Hybrid Evolutionary Algorithm. *Procedia Computer Science*, 2013, vol. 18, no. 1, p. 899–908.

[16] HONG, S.; ZHANG, D.; LAU, H. C.; ZENG, X. X.; SI, Y. A hybrid heuristic algorithm for the 2D variable-sized bin packing problem. *European Journal of Operational Research*, 2014, vol. 238, no. 1, p. 95–103.

[17] QUIROZ-CASTELLANOS, M.; CRUZ-REYES, L.; TORRES-JIMENEZ, J.; FRAIRE HUACUJA, H. J.; ALVIM, A. C. F.; GÓMEZ, C. S. A grouping genetic algorithm

with controlled gene transmission. *Computers & Operations Research*, 2015, vol. 55, no. 1, p. 52–64.

[18] LÓPEZ-CAMACHO, E.; TERASHIMA-MARIN, H.; ROSS, P.; OCHOA, G. A unified hyper-heuristic framework for solving bin packing problems. *Expert Systems with Applications*, 2014, vol. 41, no. 15, p. 6876–6889.

[19] ZHAO, X.; SHEN, H. A Parallel Algorithm for 2D Square Packing. In *Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*. Washington: IEEE Computer Society, 2013, p. 179–183.

[20] BOYER, V.; BAZ, D. E.; ELKIHEL, M. Solving knapsack problems on GPU. *Computers & Operations Research*, 2012, vol. 19, no. 1, p. 42–47.

[21] FEIER, M. C.; LEMNARU, C.; POTOLEA, R. Solving NP-Complete Problems on the CUDA Architecture Using Genetic Algorithms. In *10th International Symposium on Parallel and Distributed Computing, 6–8 July 2011, Cluj Napoca, Romania : proceedings*. Los Alamitos, Calif.: Conference Publishing Services, IEEE Computer Society, 2011, p. 278–281.

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

1. BB (angl. *branch and bound*) – šakų ir ribų algoritmas.
2. BFD (angl. *best fit decreasing*) – geriausiai tinkančio objekto algoritmas, kuriame objektai nagrinėjami ploto mažėjimo seka.
3. CPU (angl. *central processing unit*) – procesorius.
4. CUDA (angl. *Compute Unified Device Architecture*) – skaičiavimams skirta vieninga įrenginių architektūra.
5. DB – duomenų bazė.
6. Euristinis algoritmas – tai algoritmas, kuris grąžina patenkinamą sprendinį daugeliu atvejų, tačiau jis negali garantuoti, kad visada ras gerą sprendinį ar kad tai padarys greitai.
7. Evoliucinis euristinis algoritmas – tai algoritmas, kuris operuoja žemesnės eilės euristiniais algoritmais. Evoliucinio euristinio algoritmo tikslas – kiekviename žingsnyje pasirinkti labiausiai tinkantį euristinį algoritmą.
8. FFD (angl. *first fit decreasing*) – pirmo tinkančio objekto algoritmas, kuriame objektai nagrinėjami mažėjimo seka.
9. GGA-CGT (angl. *Grouping Genetic Algorithm with Controlled Gene Transmission*) – genetinis grupavimo algoritmas su valdomu genų perdavimu.
10. GPU (angl. *graphics processing unit*) – grafinis procesorius.
11. ID – identifikatorius.
12. LGFi (angl. *improved lowest gap fill*) – patobulintas mažiausio tarpo užpildymo algoritmas.
13. NP – nedeterministiškai polinomiškai.
14. PA – panaudos atvejis.
15. PĮ – programinė įranga.
16. SA (angl. *simulated annealing*) – atkaitinimo modeliavimas.
17. SIMT (angl. *single instruction multiple threads*) – viena instrukcija daugeliui gijų.
18. TDD (angl. *test driven development*) – testavimu grįstas kūrimas.

9. PRIEDAI

9.1. 1 priedas. Dvimačių supjaustymo uždavinių sprendimas naudojant grafinį procesorių

Dvimačių supjaustymo uždavinių sprendimas naudojant grafinį procesorių

Dovydas Šopa
Programų inžinerijos katedra
Kauno technologijos universitetas
Kaunas, Lietuva

***Santrauka* – straipsnyje analizuojamas dvimačių supjaustymo uždavinių sprendimas naudojant grafinį procesorių. Pasiūlytas algoritmas naudoja evoliucinį euristinį modifikavimo tipo algoritmą, kuris naudoja euristinius algoritmus sprendiniui gauti. Pateikiamas pasiūlyto algoritmo ir realizacijos gautų rezultatų palyginimas su evoliuciniu euristiniu algoritmu, kuris skaičiavimus vykdo standartiniame procesoriuje.**

Raktiniai žodžiai – daiktų išdėliojimo uždavinys, grafinis procesorius, optimizavimas, CUDA.

I. Įvadas

Objektų išdėliojimo uždavinys yra klasikinis optimizavimo uždavinys, kuris pirmą kartą suformuotas 1939 m. [1]. Šis uždavinys yra įdomus ne tik teoriškai, bet turi daug praktinių pritaikymų: straipsnių išdėliojimas laikraštyje taip, kad reikėtų kuo mažiau lapų; detalių pjaustymas lakštuose taip, kad liktų kuo mažiau nepanaudotų medžiagų; dėžių išdėliojimas sunkvežimiuose taip, kad reikėtų kuo mažiau sunkvežimių.

Gerai žinoma, kad tai nedeterministiškai polinomiškai (toliau – NP) sunkus uždavinys [2]. Pirmiausia šie uždaviniai pradėti nagrinėti kaip vienmačiai, o po to buvo praplėsti iki daugiamačių. Klasikinė šio uždavinio formuluotė skamba taip: visos dėžutės turi ilgius 1, o daiktų ilgiai yra intervale (0*;* 1] kiekvienai dimensijai. Daiktus į dėžutes reikia sudėlioti taip, kad būtų panaudota kuo mažiau dėžučių. Dvimačiu atveju matematinis šio uždavinio apibrėžimas skamba taip: duotoje dvimatėje koordinačių erdvėje *A*, kurios dydis R^2, reikia surasti tokią stačiakampių aibę *V*, kad bet kuriai stačiakampių porai $(v; v')$ iš aibės *V* sankirtų $v \cap v'$ aibė būtų tuščia [3].

NP sudėtingumo uždaviniai pasižymi tuo, kad jų sprendimui reikia didesnio nei polinominio laiko. Tai reiškia, kad šio uždavinio sprendimo trukmės apatinė riba (angl. *Lower bound*) yra $\omega(n^k)$, kur n – įvesties parametras, susijęs su duomenų kiekiu; k – bet koks skaičius.

Čia gali padėti grafiniai procesoriai (toliau – GPU). GPU gamintojai paskutiniu metu ištobulino šiuos procesorius, kad jie būtų tinkami didelės apimties skaičiavimams. Čia labai prisidėjo NVIDIA, kuri sukūrė CUDA (angl. *Compute Unified Device Architecture*). CUDA technologijos yra paremtos vienos instrukcijos daugeliui gijų pagrindu (angl. *Single instruction multiple threads*, toliau – SIMT). SIMT pasižymi tuo, kad kelios gijos vykdo tuos pačius veiksmus su skirtingais duomenimis. Tai leidžia sutaupyti laiko instrukcijoms įkrauti.

Šiame darbe nagrinėjamas objektų išdėliojimo dvimatis atvejis, kai objektai yra orientuoti stačiakampiai, negali būti vartomi, objektų sąrašas žinomas iš anksto ir lakštai yra vienodo dydžio. Bus bandoma išnaudoti CUDA suteikiamas galimybes šiam uždaviniui spręsti lygiagrečiai.

Šio darbo antrame skyriuje nurodomi susiję darbai. Trečiame skyriuje pateikiamas siūlomas evoliucinis euristinis algoritmas. Ketvirtame skyriuje pateikiamas atliktas tyrimas ir gauti rezultatai, o penktame skyriuje išvados ir tolimesni darbai.

II. Susiję darbai

Kadangi optimalus algoritmas NP sunkiems uždaviniams dar nėra sukurtas, literatūroje galima rasti daugybę straipsnių, kuriuose mokslininkai siūlo savo algoritmus. Galima pastebėti, kad geriausius rezultatus pasiekia evoliuciniai euristiniai algoritmai [4, 5, 6, 7, 8]. Šių algoritmų pagrindinė idėja – priklausomai nuo situacijos pasirinkti vieną iš paprastų euristinių algoritmų.

Literatūroje galima pastebėti dviejų tipų evoliucinius euristinius algoritmus: pasirinkimo [7] ir modifikavimo [8]. Kadangi modifikavimo algoritmai gali lengviau prisitaikyti prie specifinių uždavinių, šiam darbui pasirinktas modifikavimo tipo evoliucinis euristinis algoritmas.

X. Zhao ir H. Shen [9] pabandė sudaryti algoritmą, kuris objektų išdėliojimo uždavinį spręstų lygiagrečiai. Jie pasiūlė algoritmą, kuris pritaikytas 32 procesorių sistemai. Buvo sprendžiamas specifinis uždavinio atvejis (objektai kvadratiniai) ir pasiektas $\Theta(n)$ skaičiavimo laikas.

Praktikoje tokia procesorių sistema nepraktiška. Tokiems skaičiavimams būtų tikslinga naudoti CUDA. NP sudėtingumo uždavinius jau buvo bandoma spręsti naudojant GPU [10, 11]. Šie sprendimai parodo, kad CUDA puikiai tinka tokio tipo uždaviniams (rezultatai pagerėja apie 50 kartų).

Sprendimai, pritaikyti CUDA, apsiribojo genetinių algoritmų naudojimu. Kadangi nuosekliuose skaičiavimuose geriausius rezultatus pasiekia evoliuciniai euristiniai algoritmai, todėl jų pritaikymas lygiagretiems skaičiavimams naudojant CUDA gali pasiekti dar geresnius rezultatus. Būtent tai šiame darbe ir nagrinėjama – modifikavimo evoliucinis euristinis algoritmas, pritaikytas skaičiavimams CUDA.

III. Siūlomas evoliucinis euristinis algoritmas

Kai kurie objektų išdėliojimo uždaviniai pasižymi savybėmis, dėl kurių specifinės euristikos juos sprendžia greitai ir tiksliai. Tačiau nėra vienos euristikos, kuri puikiai spręstų

visus galimus atvejus. Evoliucinio euristinio algoritmo idėja yra sujungti paprastus euristinius algoritmus, taip bandant išvengti kiekvieno iš algoritmų silpnų vietų.

Taip pat evoliuciniai euristiniai algoritmai problemą nagrinėja ir iš kitos pusės. Jei euristiniai algoritmai operuoja duomenimis (objektais ir lakštais, į kuriuos šie objektai turi būti sudėti), tai evoliuciniai euristiniai algoritmai operuoja žemesnės eilės algoritmais (algoritmais, iš kurių reikia pasirinkti).

Siūlomas evoliucinis euristinis algoritmas remiasi kitų autorių mintimis [4, 5, 6, 7, 8]. Dauguma siūlomo evoliucinio euristinio algoritmo naudojamų euristinių algoritmų nėra nauji, tačiau pasirinkta specifinė algoritmų aibė, arba jie modifikuoti.

A. Naudojami euristiniai algoritmai

Vienmačiu objektų išdėliojimo uždavinio atveju euristikų tikslas yra išrinkti objektą ir talpyklą, į kurią šis objektas bus dedamas. Dvimačiu atveju prisideda papildomas kintamasis – reikia rasti poziciją, į kurią bus dedamas objektas lakšte. Dėl šios priežasties naudojami dviejų tipų euristiniai algoritmai: išrinkimo (išrenka objektą ir lakštą, į kurį objektas bus dedamas) ir dėjimo (išrenka poziciją lakšte, kur objektas bus dedamas). Naudojami pasirinkimo algoritmai:

1. Pirmo tinkančio (angl. *first fit*) – atidarytus lakštus nagrinėja iš eilės, ieškant pirmo, į kurį tinka objektas. Naudojamos dvi šio tipo algoritmo modifikacijos:
    1.1. ploto mažėjimo seka – objektai nagrinėjami nuo didžiausio, iki mažiausio;
    1.2. su iš anksto paskirtais objektais [6] – visi objektai, kurie yra didesni nei pusė lakšto dydžio, iš karto išdėliojami į lakštus.
2. Kito tinkančio (angl. *next fit*) – objekto dėjimui nagrinėjamas tik paskutinis naudotas lakštas. Jei objektas tinka – dedama į jį, priešingu atveju – imamas naujas lakštas. Naudojamos dvi šio tipo algoritmo modifikacijos:
    2.1. standartinis – objektai ir lakštai nagrinėjami tokia seka, kokia yra gauti;
    2.2. su iš anksto paskirtais daiktais.
3. Geriausiai tinkančio (angl. *best fit*) – lakštai išrikiuojami pagal likusio laisvo ploto didėjimo seką. Šia seka jie nagrinėjami, siekiant rasti pirmą į kurį objektas telpa. Naudojama objektų ploto mažėjimo modifikacija.
4. Blogiausiai tinkančio (angl. *worst fit*) – lakštai išrikiuojami pagal likusio laisvo ploto mažėjimo seką. Šia seka jie nagrinėjami, siekiant rasti pirmą į kurį telpa objektas (lakštai nagrinėjami priešinga seka, nei geriausiai tinkančio algoritme). Naudojama objektų ploto mažėjimo modifikacija.
5. Django ir Fitcho – algoritmas užpildo 1/*n* lakšto dalį naudodamas pirmo tinkančio algoritmą. Tada jis ieško objektų iki trijų (gali naudoti 1, 2 arba 3) kombinacijos, kuri sudarytų didžiausią plotą ir dar tilptų į lakštą. Jei kelios kombinacijos turi vienodą plotą, pirmenybė teikiama tai, kuri turi mažiau elementų. Jei ir tada yra kelios kombinacijos, pirmenybė teikiama tai, kuri turi didžiausią objektą (jei didžiausias objektas sutampa, tada ta pati sąlyga galioja antram arba trečiam pagal dydį objektui). Jei nei viena kombinacija negali būti įdėta į lakštą – imamas naujas. Naudojamos trys šio algoritmo modifikacijos (užpildymas iki 1/4, 1/3 ir 1/2) [7]. Šiame algoritme objektai nagrinėjami ploto mažėjimo seka.

Django ir Fitcho algoritmas yra vienintelis, kuris vienu pasirinkimu gali išrinkti daugiau nei vieną objektą.

Objektų dėjimui pasirinkti algoritmai, kurie orientuoti į dėjimą apatiniame kairiajame kampe. Visi algoritmai orientuoti į panašų dėjimą, kad būtų išlaikytas sprendinio stabilumas (jei dalis algoritmų dėtų nuo kairiojo kampo, o dalis nuo dešiniojo, tada viduryje gali likti tarpų, kurie bus nepatogios formos ir kuriuos bus sunku užpildyti). Naudojami dėjimo algoritmai:

1. Apatinio kairiojo kampo. Objektas įdedamas į viršutinį dešinįjį kampą. Tada, naudojant stūmimo žemyn ir kairėn veiksmus, objektas stumiamas tol, kol pasiekia kitą objektą arba lakšto kraštą.
2. Patobulinto apatinio kairiojo kampo. Objektas, kaip ir apatinio kairiojo kampo algoritme, įdedamas į viršutinį dešinįjį kampą. Tada, skirtingai nuo apatinio kairiojo kampo algoritmo, objektas stumiamas ne tik iki artimiausio objekto krašto. Pristūmus objektą iki kito objekto procesas nėra nutraukiamas – tikrinama, ar už šio yra laisvos vietos, į kurią naujas objektas tilptų. Jei ši vieta randama – objektas perkeliamas per jau egzistuojantį objektą.
3. Skenavimo. Šio algoritmo tikslas skenuoti visas pozicijas iš kairės į dešinę ir iš apačios į viršų ir ieškoti pirmos, kurioje gali būti padėtas apatinis kairysis objekto kampas.

Taip iš viso gautos 27 algoritmų kombinacijos (9 pasirinkimo ir 3 dėjimo algoritmai).

B. Evoliucinis euristinis algoritmas

Evoliucinį euristinį algoritmą galima aprašyti tokiais žingsniais (1 figūra):

1. Paruošiama 20 uždavinio kopijų.
2. Kiekvienai kopijai nustatoma dabartinė uždavinio būsena.
3. Jei rastai būsenai jau yra pasirinkta algoritmų kombinacija – ji ir naudojama. Priešingu atveju pasirenkama atsitiktinė kombinacija iš anksčiau aprašytų.
4. Šis procesas kartojamas, kol sudedami visi objektai.

```
while Uždavinio sprendinys nepagerinamas 10 kartų
    Paruošti 20 uždavinio kopijų
    foreach kopija in kopijos
        while Ne visi daiktai sudėlioti
            Nustatyti dabartinę uždavinio būseną
            if Būsena, kuriai algoritmas jau pasirinktas
                Naudoti jau pasirinktą algoritmą
            else
                Atsitiktinai išsirinkti naują algoritmą
        Apskaičiuoti kiekvieno sprendinio tinkamumą
        Geriausių individų išrinkimas
        Geriausių individų rekombinacija
        Atliekama atsitiktinių individų mutacija
```

1 figūra. Naudojamas evoliucinis euristinis algoritmas

5. Įvertinamas kiekvieno iš gautų sprendinių tinkamumas (angl. *fitness*).
6. Naudojant genetinį algoritmą, išrenkama dalis geriausią sprendinį radusių algoritmų kombinacijų:

6.1. Išrenkama 80 % algoritmų kombinacijų. Kombinacijos renkamos ruletės principu. Kiekvienam variantui, atsižvelgiant į jo tinkamumą, skiriamas intervalas intervale [0; 1). Tada generuojamas atsitiktinis skaičius šiame intervale. Pasirenkama algoritmų kombinacija, kuriai priskirtame intervale yra sugeneruotas skaičius.

6.2. Atliekama rekombinacija. Iš algoritmų kombinacijų sąrašo pasirenkamos dvi atsitiktinės ir su 50 % atliekama rekombinacija. Jei rekombinacija įvyksta, tuomet atsitiktinėje vietoje sumaišomi naudoti algoritmai, jei ne – kombinacijos perduodamos tokios, kokios yra.

6.3. Su 10 % tikimybe atliekama mutacija. Įvykus mutacijai, 20 % atsitiktinių algoritmų kombinacijų išmetama.

7. Visas šis procesas kartojamas tol, kol uždavinio sprendimas nepagerinamas 10 iteracijų.

Sprendinio tinkamumas *T* apskaičiuojamas taip:

$$T = \frac{\sum_{l=1}^{M} \left(\frac{\sum_{j=1}^{N} S_j}{N} \right)_i^2}{M} \quad (1)$$

M – sunaudotų dėžučių kiekis uždaviniui spręsti. *N* – išdėliotų objektų kiekis. S_j – vieno objekto plotas.

Uždavinio būsena koduojama eilute, kurią sudaro 23 simboliai:

1. Pirmi 6 simboliai yra skirti likusių objektų aukščiui identifikuoti. Pagal objekto aukščio santykį su lakšto aukščiu objektai sugrupuojami į tris grupes: (0; 1⁄3], (1⁄3; 1⁄2] ir (1⁄2; 1].
2. Kiti 6 (nuo 7 iki 12) simboliai yra skirti likusių objektų pločiui identifikuoti. Pagal objekto pločio santykį su lakšto pločiu objektai sugrupuojami į tris grupes: (0; 1⁄3], (1⁄3; 1⁄2] ir (1⁄2; 1].
3. Kiti 8 (nuo 13 iki 20) simboliai yra skirti likusių objektų plotui identifikuoti. Pagal objekto ploto santykį su lakšto plotu objektai sugrupuojami į 4 grupes: (0; 1⁄4], (1⁄4; 1⁄3], (1⁄3; 1⁄2] ir (1⁄2; 1].
4. Paskutiniai 3 (nuo 21 iki 23) simboliai yra skirti likusių neįdėtų objektų kiekio santykiui su pradiniu objektų kiekiu identifikuoti.

Kiekvienas iš šių identifikatorių gauna reikšmę pagal jo atitikimą vienam iš intervalų procentais. Pirmoms trims kategorijoms šifruoti naudojamas 2 simbolių kodavimas, kur „00“ atitinka intervalą [0; 10], „01“ – (10; 25], „10“ – (25; 50], „11“ – (50; 100]. Likusių objektų šifravimui naudojamas 3 simbolių kodavimas, kur „000“ atitinka intervalą [0; 12,5], „001“ – (12,5; 25], „010“ – (25; 37,5], „011“ – (37,5; 50], „100“ – (50; 62,5], „101“ – (62,5; 75], „110“ – (75; 87,5], „111“ – (87,5; 100].

C. Uždavinio lygiagretinimas

Kadangi didžiąją sprendimo laiko dalį sudaro tinkamos pozicijos lakšte ieškojimas, todėl nuspręsta būtent šią dalį realizuoti naudojant CUDA technologijas. Tam visi 3 objektų dėjimo algoritmai pritaikyti lygiagretiems skaičiavimams.

Apatinio kairiojo kampo ir patobulintą apatinio kairiojo kampo algoritmus galima lygiagrečiai vykdyti tokiame gijų skaičiuje, kiek objektų tuo metu yra lakšte. Stūmimo žemyn žingsnyje kiekviena gija pirmiausiai identifikuoja, ar objektas aktualus paieškai (bent dalis objekto turi būti po dedamu objektu). Tada kiekviena gija pažymi jos tikrinamo objekto užimamą aukščio intervalą (šiame intervale jau yra objektas, todėl naujas nebegali būti dedamas). Galiausiai viena gija (esant dideliems objekto ir lakšto matmenų skirtumams, šis veiksmas taip pat padalinamas kelioms gijos po lygiai) peržiūri pažymėtą informaciją ir patikrina, ar yra nepažymėtas reikiamo dydžio intervalas. Atitinkami veiksmai vykdomi ir objektą stumiant į kairę.

Skenavimo algoritmo įgyvendinimas šiek tiek skiriasi. Čia gijų kiekis pasirinktas pagal lakšto aukštį. Kiekviena gija nagrinėja galimybę, kad apatiniu kairiuoju objekto dėjimo kampu bus viena iš jos nagrinėjamos eilutės pozicijų. Tinkamo taško eilutėje radimui taikomas toks pat principas, kaip ir vietai, į kurią objektas dedamas apatinio kairiojo kampo ir patobulinto apatinio kairiojo kampo algoritmuose, nustatyti.

IV. Rezultatai

Algoritmas realizuotas naudojant *.NET* 4.6 technologijas bei *Cudafy.NET* biblioteką *.NET* kodui transformuoti į CUDA kodą. Tyrimai daryti nešiojamame kompiuteryje, kuris turi Intel® Core™ i7-4710HQ 2,5 GHz procesorių, 8 GB darbinės atminties (RAM) ir NVIDIA GeForce GTX 850M GPU. Palyginimui pasirinktas C. Bluma ir V. Schmido pasiūlytas algoritmas [8]. Šis algoritmas šiuo metu pasiekia vienus iš geriausių rezultatų. Pasirinkti pirmų penkių klasių (I–V) duomenys. Informacija apie šias duomenų klases pateikta I lentelėje. Kiekvienoje klasėje objektų kiekis yra iš aibės {20, 40, 60, 80, 100}. Kiekvienai klasei ir kiekvienam objektų kiekiui pateikiama po 10 duomenų rinkinių. Taip iš viso gaunama 250 duomenų rinkinių algoritmams palyginti.

Žinoma, negalima tiesiogiai palyginti C. Bluma ir V. Schmido gautų sprendimo laikų su šiame darbe pasiūlyto algoritmo sprendimo laikais, kadangi kompiuterių parametrai labai skyrėsi. Šie laikai pasirinkti kaip atskaitos taškas, kurį CUDA technologijomis realizuotas algoritmas turėtų įveikti.

I LENTELĖ. Duomenų rinkinių klasės

Klasė	Objekto plotis	Objekto aukštis	Lakšto plotis	Lakšto aukštis
I	[1; 10]	[1; 10]	10	10
II	[1; 10]	[1; 10]	30	30
III	[1; 35]	[1; 35]	40	40
IV	[1; 35]	[1; 35]	100	100
V	[1; 100]	[1; 100]	100	100

Gauti rezultatai apibendrinti II lentelėje. Kiekvienoje eilutėje pateikiami 10 duomenų rinkinių jungtiniai rezultatai. Pirmame stulpelyje pateikiamas objektų kiekis klasėje, antrame – geriausias iki šiol žinomas šių 10 duomenų rinkinių sprendimo rezultatas (LB, angl. *Lower bound*). Rezultatų palyginimas pateiktas dviem stulpeliais: pirmajame pateikiamas

algoritmo gautas rezultatas, o antrajame – laikas, per kurį rezultatas pasiektas. Paskutinėje eilutėje pateikiama visų reikalingų lakštų kiekių suma ir vidutis vieno duomenų rinkinio vykdymo laikas.

Rezultatų patikimumui užtikrinti buvo imtasi kelių priemonių. Visų pirma, naudojamas kompiuteris buvo atjungtas nuo interneto. Taip pat buvo išjungtos visos programos ir procesai, kurių nereikia normaliam operacinės sistemos darbui užtikrinti. Galiausiai, skaičiavimai su kiekvienu duomenų rinkiniu buvo vykdomi bent po 2 kartus. Jei abiejų vykdymų laikai buvo panašūs – imamas šių laikų vidurkis. Priešingu atveju skaičiavimai su duomenų rinkiniu buvo vykdomi trečią kartą. Tuomet buvo atsisakoma labiausiai nutolusio rezultato ir pateikiamas 2 artimesnių rezultatų vidurkis.

Palyginus rezultatus galima pastebėti, kad CUDA vykdyti skaičiavimai nusileidžia C. Bluma ir V. Schmido gautiems rezultatams. Pirmiausia tai privertė suabejoti realizacijos korektiškumu. Dėl to buvo atliktas papildomas realizuotos programos tyrimas naudojant *ANTS Performance Profiler*. Šis įrankis leidžia išanalizuoti, kurios programos vietos vykdomos ilgiausiai. Atlikus šią analizę pastebėta, kad apie 60 % laiko programa praleidžia siųsdama duomenis tarp CPU ir GPU. Tai visiškai neefektyvu. Jei atmestumėme šiuos 60 % laiko, tada vidutinis vykdymo laikas nuo 5,39 s nukristų iki 2,16 s. Šie rezultatai verčia susimąstyti apie reikalingus realizacijos pakeitimus.

V. Išvados ir tolimesni darbai

Ši evoliucinio euristinio algoritmo realizacija yra neefektyvi. Dėl nuolatinio duomenų perdavimo tarp CPU ir GPU labai smarkiai nukenčia vykdymo laikas. Reikia sumažinti bendravimą tarp CPU ir GPU. Tam reikia ne tik atskirų programos dalių (dabartiniu atveju tik dėjimo algoritmų), bet viso evoliucinio euristinio algoritmo pritaikymo skaičiavimas GPU. Tai ir bus tolimesnis darbas.

Taip pat reikėtų atsisakyti *Cudafy.NET* bibliotekos ir tiesiogiai naudoti CUDA teikiamą sąsają. Sistemos kūrimo metu pastebėtos kelios bibliotekos klaidos, dėl kurių sugeneruotas kodas kartais būna neteisingas arba neoptimalus.

II LENTELĖ. Rezultatų palyginimas

Uždavinys	LB	EA-LGFi [8]		Šiame darbe pasiūlytas	
		Rezultatas	*Laikas (s)*	*Rezultatas*	*Laikas (s)*
I klasė					
20	71	71	0,00	71	0,00
40	134	134	0,00	134	0,00
60	197	200	0,01	200	0,00
80	274	275	0,00	275	0,00
100	317	317	0,00	317	0,00
II klasė					
20	10	10	0,00	10	0,01
40	19	19	0,00	19	0,01
60	25	25	0,00	25	0,01
80	31	31	0,00	31	0,01
100	39	39	0,00	39	0,01
III klasė					
20	51	51	0,02	51	0,04
40	92	94	0,01	94	0,05
60	136	139	0,27	139	1,02
80	187	189	20,68	189	39,85
100	221	224	26,17	223	44,21
IV klasė					
20	10	10	0,00	10	0,02
40	19	19	0,00	19	0,02
60	23	23	12,18	25	0,03
80	30	31	0,00	32	14,36
100	37	37	0,00	38	0,04
V klasė					
20	65	65	0,00	65	0,01
40	119	119	0,03	119	0,04
60	179	180	0,14	180	0,17
80	241	247	0,03	247	0,03
100	279	284	27,33	284	34,62
Rezultatai	2806	2833	3,47	2836	5,39

VI. Naudota literatūra

[1]. L. V. Kantorovich, „Mathematical Methods of Organizing and Planning Production“, „Management Science“, Maryland, Vol. LX, No. 4, INFORMS, p. 366–422, 1960.

[2]. M. R. Garey ir D. S. Johnson, „Computers and Intractability: A Guide to the Theory of NP Completeness“, New York: W.H. Freeman and Company, 1979.

[3]. A. Quilliot ir H. Toussaint, „About Casting 2D Bin Packing into Network Flow Theory“, HAL, 2010.

[4]. H. Terashima-Marín, C. J. Farías-Zárate, P. M. Ross ir M. Valenzuela-Rendón, „Comparing Two Models to Generate Hyper-heuristics for the 2D Regular Bin Packing Problem“, įtraukta į „Proceedings of the 9th annual conference on Genetic and evolutionary computation“, New York, ACM, 2007, p. 2182–2189.

[5]. S. Hong, D. Zhang, H. C. Lau, X. Zeng ir Y. Si, „A hybrid heuristic algorithm for the 2D variable sized bin packing problem“, „European Journal of Operational Research“, Vol. CCXXXVIII, No. 1, p. 95–103, 2014.

[6]. M. Quiroz-Castellanos, L. Cruz-Reyes, J. Torres-Jimenez, C. Gómez S., H. J. Fraire Huacuja ir A. C. Alvim, „A grouping genetic algorithm with controlled gene transmission“, „Computers & Operations Research“, Vol. LV, p. 52–64, 2015.

[7]. E. López-Camacho, H. Terashima-Marin, P. Ross ir G. Ochoa, „A unified hyper heuristic framework for solving bin packing problems“, „Expert Systems with Applications“, Vol. XLI, No. 15, p. 6876–6889, 2014.

[8]. C. Bluma ir V. Schmidc, „Solving the 2D bin packing problem by means of a hybrid evolutionary algorithm“, „Procedia Computer Science“, Vol. XVIII, p. 899–908, 2013.

[9]. X. Zhao ir H. Shen, „A Parallel Algorithm for 2D Square Packing“, įtraukta į „Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies“, Washington, IEEE Computer Society, 2013, p. 179–183.

[10]. V. Boyer, D. El Baz ir M. Elkihel, „Solving knapsack problems on GPU“, „Computers & Operations Research“, Vol. XIX, p. 42–47, 2012.

[11]. M. Calid Feier, C. Lemnaru ir R. Potolea, (2010), „Solving NP-Complete Problems on the CUDA Architecture using Genetic Algorithms“, „10th International Symposium on Parallel and Distributed Computi