KAUNO TECHNOLOGIJOS UNIVERSITETAS

FUNDAMENTALIŲJŲ MOKSLŲ FAKULTETAS

TAIKOMOSIOS MATEMATIKOS KATEDRA

Jonas Pokštas

PJAUSTYMO UŽDAVINIO ALGORITMŲ REALIZACIJA IR TYRIMAS

Magistro darbas

Vadovas
doc. dr. N. Listopadskis

KAUNAS, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS

FUNDAMENTALIŲJŲ MOKSLŲ FAKULTETAS

TAIKOMOSIOS MATEMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
prof. dr. J.Rimas
2007 06 06

PJAUSTYMO UŽDAVINIO ALGORITMŲ REALIZACIJA IR TYRIMAS

Taikomosios matematikos magistro baigiamasis darbas

Vadovas
( )doc.dr.N.Listopadskis
2007 06 03

Recenzentas
( ) doc. dr. Eimutis Karčiauskas
2007 06 01

Atliko
FMMM-5 gr. stud.
( ) J.Pokštas
2007 05 25

KAUNAS, 2007

KVALIFIKACINĖ KOMISIJA

Pirmininkas: Leonas Saulis, habil. dr., profesorius (VGTU)

Sekretorius: Eimutis Valakevičius, docentas (KTU)

Nariai: Algimantas Jonas Aksomaitis, profesorius (KTU)

Vytautas Janilionis, docentas (KTU)

Vidmantas Povilas Pekarskas, profesorius (KTU)

Rimantas Rudzkis, habil.dr., banko „NORD/LB“ vyriausiasis analitikas

Zenonas Navickas, profesorius (KTU)

Arūnas Barauskas, dr.,UAB „Elsis“ generalinio direktoriaus pavaduotojas

Pokštas J. Pjaustymo uždavinio algoritmų realizacija ir tyrimas: taikomosios matematikos magistro darbas / darbo vadovas doc. dr. N. Listopadskis, taikomosios matematikos katedra, fundamentaliųjų mokslų fakultetas, Kauno technologijos universitetas. Kaunas, 2007. - 67 p.

SANTRAUKA

Šiame darbe nagrinėjama negiljotininio, dvimačio, stačiakampių pjaustymo uždavinio atliekų minimizavimo problema ir jos sprendimo metodai. Dėl uždavinio kombinatorinio sudėtingumo neįmanoma tiksliai ir visais atvejais pateikti optimalų jo sprendinį, todėl pasirinkti apytiksliai sprendimo metodai.

Uždavinys sprendžiamas metaeuristiniais hibridiniais genetiniu ir modeliuojamo atkaitinimo algoritmais apjungtais su euristiniais „Žemiausio kairėn užpildymo“ ir „Žemiausio tarpo“, kuris yra originali „Geriausiai tinkamo“ metodo modifikacija. Taip pat realizuojami minėti euristiniai algoritmai atskirai nuo hibridinių. Atliekama šių metodų lyginamoji analizė bei jų parametrų ir pradinių sąlygų parinkimo įtakos tyrimas sprendinio kokybei. Suformuojama ir pateikiama metodika pjaustymo uždavinių sprendimui.

Šis darbas pristatytas trijose konferencijose (VI studentų konferencija, 2006; Matematika ir matematinis modeliavimas, 2006; Lietuvos matematikų draugijos XLVII konferencija, 2006). Taip pat išleisti trys straipsniai šios temos pagrindu [19], [20], [21] ir pateiktas vienas naujas Lietuvos matematikų draugijos XLVIII konferencijai.

Pokštas J. Implementation and analysis of cutting stock problem algorithms: Masters‘s work in applied mathematics / supervisor dr. assoc. prof. N. Listopadskis; Department of Applied mathematics, Faculty of Fundamental Sciences, Kaunas University of Technology. Kaunas, 2007. – 67 p.

SUMMARY

A non – guillotineable, two – dimensional, rectangular cutting stock problem is being introduced in this paper and its solving methods either. Due to the combinatorial complexity of a problem, it is impossible to solve it optimally for every instance. Consequently an aproximate methods have been chosen.

The problem is solved by metaheuristic genetic and simulated annealing methods hybridised with heuristic „Bottom Left Fill“ and „Lowest Gap“, which is an originally modified version of „Best Fit“ algorithm. The same heuristic algorithms are implemented separately from hybridised ones. A comparation analysis of these methods is done and the influence on solution quality depending on the selection of algorithms parameters and its initial conditions is considered. The methodology of solving cutting stock problems is being formulated and presented.

This work has been reported at the three conferences (6th Student‘s Conference, 2006; Mathematics and Mathematical Modelling, 2006; 47th Conference of Lithuanian Mathematician Association, 2006). In adittion, three articles have been published based on this topic [19], [20], [21] and one more arcticle is going to be presented at the 48th Conference of Lithuanian Mathematician Association.

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

1. ĮVADAS

1.1.PJAUSTYMO UŽDAVINIAI IR JŲ PRITAIKYMO GALIMYBĖS

Šiame darbe nagrinėsime pjaustymo uždavinį, kuris yra sprendžiamas daugelyje veiklos sričių ir pasireiškia kaip pagrindinis arba šalutinis uždavinys įvairiomis formomis tokiomis kaip:

- Stiklo, geležies, medienos, odos supjaustymo uždavinys aptinkamas pramoniniuose procesuose;
- Objektų išdėstymo ir talpinimo uždavinys logistikos procesuose;
- Efektyvaus patalpų padalijimo uždavinys architektūriniuose sprendimuose;
- Kompiuterio atminties valdymo (paskirstymo bei išskyrimo) uždavinys ir t.t.

Platesnė pjaustymo uždavinio klasifikacija suformuluota bei pateikta (H. Dyckhoff, 1990) ir (G. Wascher, 2006) veikaluose [1], [2]. Paprastai šios problemos literatūroje identifikuojamos kaip *pjaustymo ir pakavimo* uždaviniai (angl. *Cutting and Packing Problems*).

Taigi visus šiuos uždavinius sieja panašaus pobūdžio tikslai:

- Rasti geriausią supjaustymo šabloną, kad būtų mažiausias atliekų kiekis;
- Efektyviausiai išdėstyti prekes ant paletės transportavimui kiek galima didesniais kiekiais;
- Sukurti patalpų padalijimo schemą, efektyvesniam erdvės išnaudojimui;
- Optimaliai paskirstyti ir valdyti kompiuterinę atmintį, kad butų pagreitinti kompiuterinio skaičiavimo procesai ir pan.

Tokių užduočių sprendimo vykdymo metu, atliekamas optimizavimas. Kadangi paprastai susiduriama su diskrečiais dydžiais ir optimizavimo procesas grindžiamas elementariomis kombinatorinėmis (perstatymo ir pan.) operacijomis, tai pjaustymo ir pakavimo (toliau pjaustymo) uždavinys priskiriamas *kombinatorinio optimizavimo* (angl. *Combinatorial optimization*) užduočių klasei [1].

Šiems pjaustymo uždaviniams yra būdinga tai, kad didėjant jų apimčiai, visais atvejais tiksliai išspręsti juos per trumpą sprendimo laiko tarpą pasidaro nebeįmanoma. Todėl algoritmų sudėtingumo teorijoje tokios užduotys priskiriamos *NP sunkių* problemų sudėtingumo klasei (angl. *Nondeterministic polynomial hard*) [3]. Be to, kai kurių pjaustymo užduočių sprendimo variantuose reikalaujama operuoti sveikaisiais (arba natūriniais) skaičiais, taigi susiduriama su *sveikaskaičio programavimo* uždaviniu (angl. *Integer programming*) [4].

Dėl šių priežasčių yra ribotas tiksliųjų metodų panaudojimas, t.y. šie metodai taikomi, tačiau nedidelės apimties užduotims spręsti [5].

Sparčiai besivystant kompiuterinėms technologijoms, pradėtas simuliuoti pjaustymo procesas kompiuterio pagalba. Todėl atsivėrė plačios galimybės taikyti *euristinius* ir

metaeuristinius sprendimo metodus, kurie sėkmingai panaudoti apytiksliam pjaustymo uždavinių sprendimui [5], [6].

1.2.DARBO TIKSLAI

Šiame darbe nagrinėjamas *dvimatis* pjaustymo uždavinys, apibrėžtas 2.4. skyrelyje.

Pagrindinis darbo tikslas ištirti pasirinktus euristinius ir metaeuristinius metodus bei juos realizuoti tam, kad būtų galima išspręsti pasirinktą uždavinį ir atlikti algoritmų lyginamąją analizę.

Kadangi tai yra apytiksliai sprendimo metodai, priklausantys nuo įvairių parametrų, tai atlikti taip pat jų įtakos tyrimą sprendinio kokybei.

Suformuluoti pjaustymo uždavinių sprendimo metodiką

Papildomai pabandyti realizuoti tam tikrų metodų modifikacijas tam, kad galima būtų juos apjungti, t.y. hibridizuoti su kitais.

Taip pat panagrinėti, kokią įtaką sprendimo procesui turi skirtingi pradinių sąlygų parinkimo būdai.

Bendrojoje dalyje skaitytojas supažindinamas su kombinatorinio uždavinio bei algoritmų sudėtingumo teorijos pagrindiniais teiginiais, pateikiama pjaustymo uždavinio klasifikacija bei sprendimo metodų bendras aprašymas.

Tiriamojoje dalyje nagrinėjama sprendimo schema, algoritmų hibridzavimo principas ir jų realizacijų aprašymas pasirinktam uždaviniui spręsti bei tyrimo rezultatai.

Galiausiai diskusijos skyrelyje aptariami gauti rezultatai ir suformuluojamos išvados.

2. BENDROJI DALIS

2.1.KOMBINATORINIO OPTIMIZAVIMO UŽDAVINIO SAMPRATA

Daugumos praktinių ir teorinių optimizavimo uždavinių svarba susideda iš geriausios kintamųjų konfigūracijos paieškos tam, kad būtų pasiekti keliami tikslai. Spręsdami kombinatorinio optimizavimo uždavinį, ieškosime optimalaus sprendinio baigtinėje arba skaičiojoje, t.y. diskrečiojoje aibėje.

Kombinatorinis optimizavimo uždavinys *KOU* = (*X*, *P*, *Y*, *f*, *extr*) formaliai apibrėžiamas tokiu būdu, kur:

- *X* diskrečioji sprendinių aibė, kurioje apibrėžtos *f* ir *P;*
- *P* sprendinių, tenkinančių apribojimus, predikatas $P : X \to \{0,1\}$, čia $P = 0$, jei apribojimai netenkinami ir $P = 1$ atvirkščiai;
- Y tenkinančių apribojimus sprendinių aibė;
- Tikslo funkcija *f*, kur $f : Y \to R$;
- *extr* – tikslo funkcijos ekstremumas (minimumas ar maksimumas).

Y dažnai vadinama paieškos arba sprendinių, tenkinančių apribojimus, erdve. Tam, kad išspręsti kombinatorinio optimizavimo uždavinį, reikia rasti tokį sprendinį $y^* \in Y$, prie kurio tikslo funkcijos reikšmė būtų minimali (arba maksimali), t.y. $f(y^*) \le f(y)$ arba $f(y^*) \ge f(y), \forall y \in Y$.

Tuomet y^* vadinamas uždavinio globaliai optimaliuoju sprendiniu ir aibė $Y^* \subseteq Y$ - globaliai optimalių sprendinių aibe. Dažniausiai aibę Y^* sudaro vienas sprendinys.

2.2. UŽDAVINIŲ SUDĖTINGUMO KLASĖS

Algoritmų sudėtingumo teorijoje skiriamos dvi pagrindinės uždavinių sudėtingumo klasės *P* ir *NP*.

P (deterministic polynomial) – tai tokia sudėtingumo klasė, apjungianti uždavinius, kuriuos galima išspręsti per polinominį laiką – *m(n).*

NP (nondeterministic polynomial) – tai tokia sudėtingumo klasė, apjungianti uždavinius, kuriuos galima apytiksliai išspręsti per polinominį laiką – *m(n).*

Čia $m(n) = O(n^k)$, kur *k* – konstanta, priklausanti nuo uždavinio, kurio apimtis – *n*.

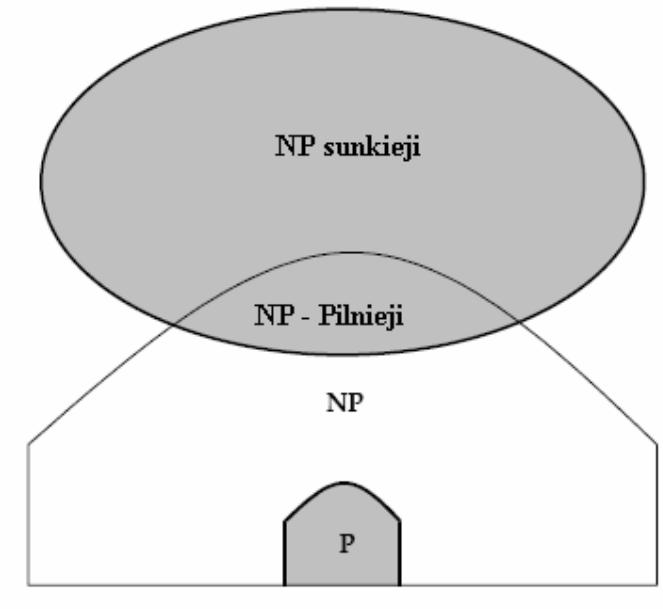
Tarkime, kad norime išspręsti uždavinį *X*, kai tuo tarpu jau žinome uždavinio *Y* sprendimo algoritmą ir galime sukonstruoti tokią funkciją *y = T(x)*, kur *x* ir *y* yra atitinkamai uždavinių *X* ir *Y* pradiniai duomenys. Jei algoritmas išsprendžia uždavinį *X* tada ir tik tada, kai išsprendžiamas uždavinys *Y* ir funkcija *T(x)* įvykdoma per polinominį laiką, tuomet laikome, kad uždavinys *X*

gali būti suvestas į uždavinį *Y* ir žymime $X \leq_p Y$. Kitais žodžiais tariant, tai reikštų, kad nėra sudėtingiau išspręsti uždavinį *X* nei *Y*.

Uždavinys *Y* yra *NP sunkus (NP hard)*, jeigu $X \leq_p Y, \forall X \in NP$. Jeigu *Y* yra *NP sunkus* uždavinys ir $Y \in NP$, tuomet *Y* vadinsime *NP pilnuoju (NP - Complete)* uždaviniu.

Taigi *NP sunkus* uždavinys *Y* gali ir nepriklausyti *NP* klasei, bet jis yra ne mažiau sudėtingas kaip ir kiekvienas uždavinys *X* iš *NP* (2.1 pav.). *NP pilnieji* yra patys sudėtingiausi klasės *NP* uždaviniai [7].

Jeigu kuris nors *NP pilnasis* uždavinys priklausytų sudėtingumo klasei *P*, tuomet galiotų lygybė *NP = P*. O tai reikštų, kad egzistuoja deterministinis polinominio laiko algoritmas, kuris visais atvejais optimaliai išspręstų bet kurį uždavinį iš *NP* klasės, tačiau dar niekam nepavyko sukurti tokio algoritmo, todėl manoma, kad $P \subseteq NP$.

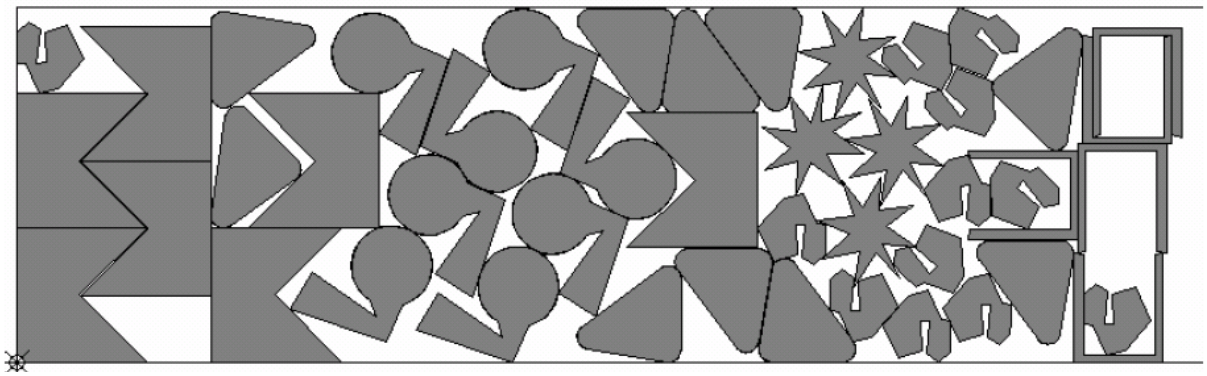


2.1 pav. Klasių sudėtingumo diagrama

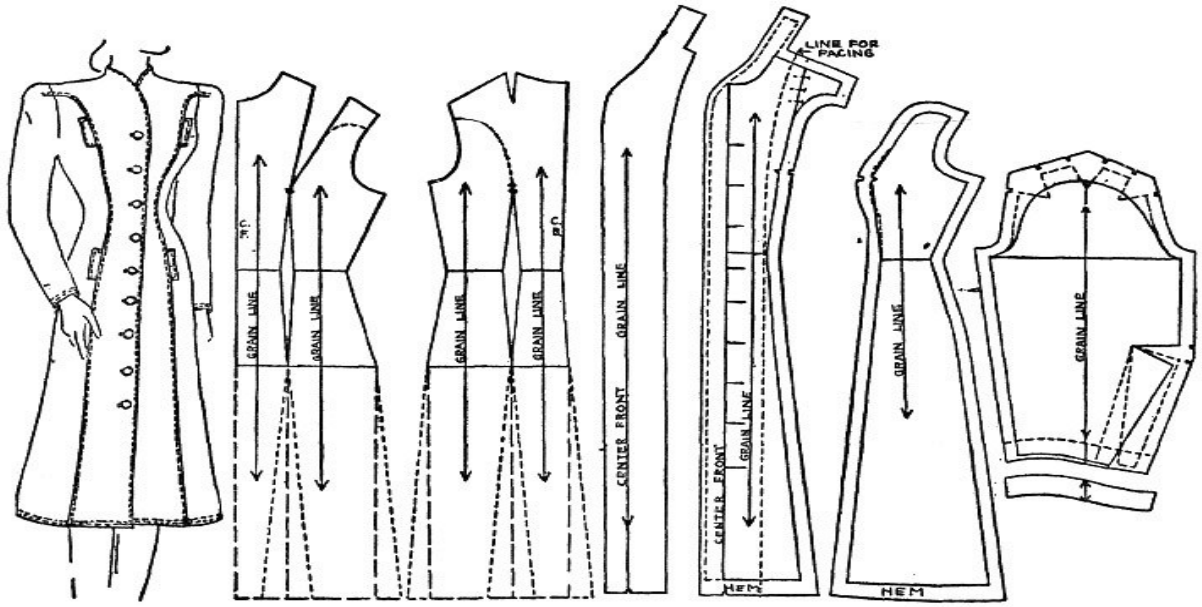
2.3.PJAUSTYMO UŽDAVINIŲ KLASIFIKACIJA

Pjaustymas – tai fizinio objekto arba jo dalies padalijimas į dvi ar keletą naujų dalių, panaudojant pjaustymo įrankį.

Akivaizdu, kad pjaustymo įrankiai gali būti labai įvairūs, taip pat ir pjaustymo objektai bei išpjovos gali būti visokiausių formų:



2.2 pav. Įvairių formų detalių pjaustymas



2.3 pav. Suknelės pjaustymo šablonų parengimo pavyzdys

Kadangi figūrų geometrija uždavinio kombinatoriniui sudėtingumui ir jo optimizavimo metodologijai esminio skirtumo nedaro, tai savo darbe susiaurinsim pjaustymo problemos klasę ir nagrinėsim atvejį, kai pjaustomi objektai ir iš jų gaunamos detalės yra stačiakampiai (kvadratai). Medžiagos prigimties tai pat nepaisysim, nes tai esminio skirtumo nesuteikia mūsų nagrinėjamai problemai.

Kai smulkios detalės (išpjovos) pjaustomos iš didelių objektų (lakštų), iškyla *atliekų minimizavimo problema*, t.y. kokiu būdu pjaustyti pasirinktą objektą, kad atliekų kiekis būtų mažiausias. Literatūroje tokia situacija dažnai identifikuojama pavadinimu - *žaliavų pjaustymo uždavinys* (Cutting Stock Problem, CSP).

Pjaustymo uždavinys gali būti klasifikuojamas pagal Dyckhoff pasiūlytą schemą [1]. Pradžioje išskiriamos dvi pagrindinės grupės:

- Orientuotas į išpjovą būdas (pjaustymo metu kiekviena išpjova išpjaunama individualiai)
- Orientuotas į šabloną būdas (pirmiausiai iš išpjovų yra sudaromi šablonai, kuriems nustatomi intensyvumai, užsakymui patenkinti)

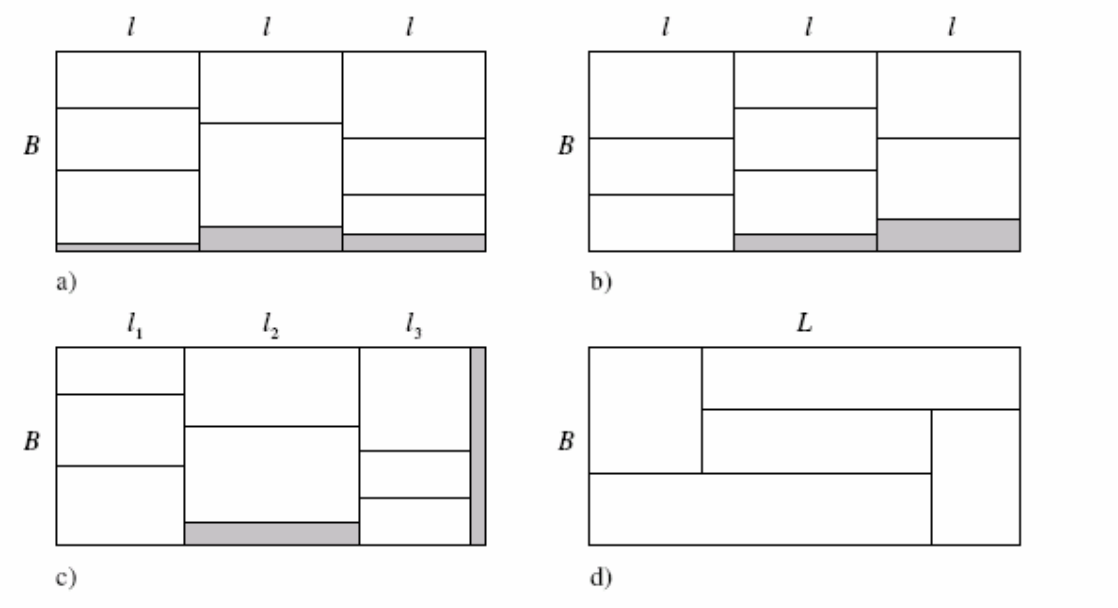
Be šių grupių apibrėžiamos keturios pagrindinės charakteristikos:

1. Dimensijų skaičius
    - 1 dimensija
    - 2 dimensijos
    - 3 dimensijos
    - N dimensijų (N > 3)
2. Žaliavų kiekis
    - Žaliavų deficitas
    - Žaliavų pakankamumas
3. Lakštų asortimentas
    - Vienas lakštas
    - Daug identiškų lakštų
    - Skirtingi lakštai
4. Išpjovų asortimentas
    - Nedaug išpjovų su skirtingomis dimensijomis
    - Daug išpjovų su skirtingomis dimensijomis
    - Daug išpjovų su panašiomis dimensijomis
    - Daug identiškų išpjovų

Svarbiausia charakteristika yra *dimensijų skaičius* – tai minimalus skaičius dimensijų, kurios yra reikšmingos sprendimo nustatymui. Paprasčiausias yra vienos dimensijos atvejis, tipinis pavyzdys būtų plieno strypų pjaustymas, kurių ilgis yra fiksuotas. Dviejų dimensijų atveju detalės išdėstomos objekte dviejų kintančių dimensijų atžvilgiu ir t.t.

Dviejų dimensijų atveju, pjaustymo procesas gali būti sudarytas iš atskirų žingsnių, kuriuose objektas pjaustomas į paobjekčius atitinkamais kampais, atsižvelgiant į ankstesnius pjūvius – tai *fazinis dvidimensinis* pjaustymas. Jei pjūviai daromi lygiagrečiai lakšto šonui, tuomet uždavinį vadinsime *ortogonaliuoju dvidimensiniu*. Pagaliau, pjūvį, kuris užima visą lakšto arba jo dalies, kuris ankstesnio pjūvio rezultatas, plotį – *giljotininiu*.

Skirtingos CSP pjaustymo procedūros pavaizduotos 2.4 paveikslėlyje. a) ir b) vienos dimensijos CSP atvejis, pjaustomo objekto B ilgis fiksuotas. Dviejų dimensijų CSP atveju c) išpjovų ilgiai skiriasi. Be to a) – c) atvejais turime *ortogonalųjį dvifazį giljotininį pjūvį*, o d) – pjūviai nėra *giljotininiai*.



2.4 pav. Pjūvių pavyzdžiai

Antroji charakteristika yra *žaliavų kiekis.* Pirmuoju atveju, visos žaliavos bus sunaudotos, bet ne visi užsakymai bus įvykdyti. Antruoju – visi užsakymai bus įvykdyti.

Trečioji charakteristika – lakštų asortimentas*,* gali būti suskirstyta į tris tipus. Pirmas, yra tiktai vienas didelis lakštas, antras – daug didelių lakštų su panašiomis dimensijomis, trečias – daug lakštų su skirtingomis dimensijomis.

Analogiškai suprantama ketvirtoji charakteristika - *išpjovų asortimentas.* Pirmu atveju - nedaug išpjovų su skirtingomis dimensijomis, antruoju – daug išpjovų su daug skirtingų dimensijų, trečiuoju – daug išpjovų su panašiomis dimensijomis, ketvirtuoju – daug identiškų išpjovų.

Net ir paprasčiausias CSP yra *NP pilnojo* sudėtingumo, tai reiškia, kad neegzistuoja deterministinis polinominio laiko algoritmas, kuris visais atvejais galėtų pateikti tikslų sprendimą. Dėl uždavinio kombinatorinio sudėtingumo prigimties, jo sprendimas dažniausiai reikalauja apytikslių sprendimo metodų.

2.4.PASIRINKTO UŽDAVINIO MATEMATINĖ FORMULUOTĖ

Nagrinėsime tokį pjaustymo uždavinį, kai lakštai, iš kurių bus pjaustomos figūros ir pačios figūros yra stačiakampiai (kvadratai). Atliekami pjūviai bus negiljotininiai, t.y. vykdant pjūvį, galimas jo krypties keitimas. Taip pat pjūvio vykdymo kryptys bus lygiagrečios vienai iš pjaustomo lakšto kraštinių. Papildomai atsiribosime ir nuo pjaustymo siūlės storio, tarsime, kad jis yra nykstamai mažas ir mūsų nagrinėjamo uždavinio sprendimui įtakos neturi.

Žinomas stačiakampių užsakymas, kuriuos reikės išpjauti: $\{(w_j,l_j,q_j)|j,w_j,l_j,q_j \in N\}$, čia w_j – plotis, l_j – ilgis, q_j – kiekis, $j=1,..,n$ ir lakštų (W,L), čia W – plotis, L – ilgis (pagrindinio stačiakampio, iš kurio bus išpjautos figūros). Organizuoti lakštų išpjaustymą taip, kad atliekų kiekis būtų mažiausias (arba kiek galima mažesnis). Pjaustymo metu galimas stačiakampio pasukimas 90 laipsnių kampu.

Tarkime $s_{i,j}$ $(i=1,..,m,\ j=1,..,n)$ reiškia kiekį išpjautų detalių (w_j,l_j), įvykdžius šabloną i, kurį suprasime kaip tam tikrą vieno lakšto išpjovimo būdą. Vienu šablonu galima išpjauti keletą lakštų, kurių skaičių žymėsime u_i.

$$\min \sum_{i=1}^{m} (WL - \sum_{j=1}^{n} s_{i,j} w_j l_j) \cdot u_i \tag{2.1}$$

$$\sum_{i=1}^{m} s_{i,j} u_i = q_j, j = 1,\ldots,n \tag{2.2}$$

$$w_i \le W, l_i \le L, i = 1,...,n \tag{2.3}$$

$$u_i \in N, i = 1,...,m \tag{2.4}$$

2.1 tikslo funkcijos reikšmė nurodo minimalų atliekų kiekį, įvykdžius pjaustymą. Apribojimu 2.2 reikalaujama, kad išpjautų stačiakampių kiekis atitiktų užsakymo kiekius q_j. 2.3 apribojimo prasmė yra tokia, kad pjaustomos detalės neišeitų iš lakšto ribų. Iš 2.4 išraiškos išplaukia, kad tai yra sveikaskaičio programavimo uždavinys. Taip pat akivaizdus papildomas reikalavimas, kad išdėstytos lakšte (išpjautos) detalės nesikirstų.

Iš uždavinio apibrėžimo išplaukia, kad lakštų (pradinių žaliavų) kiekis yra be galo didelis. Nagrinėsime taip pat situaciją, kai pradinių žaliavų kiekis yra ribotas, todėl įvesime dar vieną apribojimą:

$$\sum_{i=1}^{m} u_i \le k, k \in N \tag{2.5}$$

Apribojimu 2.5 reikalaujama, kad sudarytų šablonų bendras skaičius neviršytų tam tikrų iš anksto turimų lakštų atsargų.

Taigi, remiantis anksčiau pateikta Dyckhoff klasifikacija, pirmiausia uždavinys yra pagrįstas *orientuotu į šabloną būdu* (pradžioj yra sudaromi pjaustymo šablonai, kuriems nustatomas jų panaudojimo skaičius, užsakymui patenkinti). Tuomet, atsižvelgiant į vėlesnius punktus suklasifikuojame abu uždavinio variantus:

- 2 dimensijos | Žaliavų pakankamumas | Daug identiškų lakštų;
- 2 dimensijos | Žaliavų deficitas | Daug identiškų lakštų.

Pastaba. Kadangi nagrinėjamų metodų logika nepriklauso nuo išpjovų asortimento, tai ketvirtas segmentas į klasifikaciją neįtrauktas.

Pirmuoju atveju sandėlyje turime be galo daug medžiagos lakštų (žaliavų), todėl užsakymas visą laiką bus įvykdytas, antruoju – žaliavų ištekliai riboti, todėl gali būti situacijų, kai bus įvykdyta tik kažkuri dalis užsakymo.

Taigi atsižvelgiant į anksčiau išvardintus teiginius, susiduriame su *dvidimensiniu, negiljotininiu, ortogonaliuoju* pjaustymo uždaviniu. Kaip jau minėta anksčiau taip apibrėžtas uždavinys yra *NP sunkus*.

Verta priminti, kas buvo paminėta anksčiau, kad taip suformuluotas uždavinys yra analogiškas *talpinimo* uždaviniui (*Packing problems*), todėl tiriamojoje dalyje nagrinėjami algoritmai tinka tiek pjaustymo, tiek ir talpinimo problemoms spręsti.

2.5.PJAUSTYMO UŽDAVINIŲ METODŲ APŽVALGA

Dažniausiai literatūroje minimos keturios pagrindinės pjaustymo uždavinių sprendimo klasės.

Tikslieji metodai garantuoja optimalų uždavinių sprendimą, tačiau susiduriant su didesnės apimties uždaviniais sprendimo laikas pasidaro labai didelis, dėl tos priežasties, kad pjaustymo uždavinys yra *NP sunkus*. Gilmore ir Gomory (1961 m.) buvo vieni iš pirmųjų tyrinėtojų, kurie nagrinėjo tiksliuosius metodus pjaustymo uždaviniams spręsti [8]. Jie panaudojo tiesinio programavimo ideologiją ir gavo optimalius sprendinius, tačiau tik nedidėlės apimties uždaviniams. Christofides ir Whitlock (1977 m.) panaudojo metodą pagrįstą “medžio” tipo struktūra giljotininio[9], o Beasley (1985 m.) negiljotininio pjaustymo uždaviniui spręsti[10], vėlgi gauti optimalūs sprendiniai, tačiau tik labai nesudėtingiems uždaviniams.

Euristiniai metodai retai randa optimalų, bet dažnai pateikia pakankamai gerą bei priimtiną sprendimą per trumpą paieškos laiką. Euristiniai metodai yra labai priklausomi nuo uždavinio tipo, tai reiškia, kad jie naudoja specifišką informaciją apie konkrečius uždavinius, kuriems jie yra sukurti spręsti. Albano ir Sapuppo (1980 m.) pristatė euristinio “Žemiausio Kairėn Užpildymo” algoritmo idėjas [11]. Jakobs (1996 m.) panaudojo “Žemiausio kairėn” algoritmą, kuris ant didelio lakšto dėstė pagal tam tikrą logiką mažesnės figūras, t.y. organizavo jų

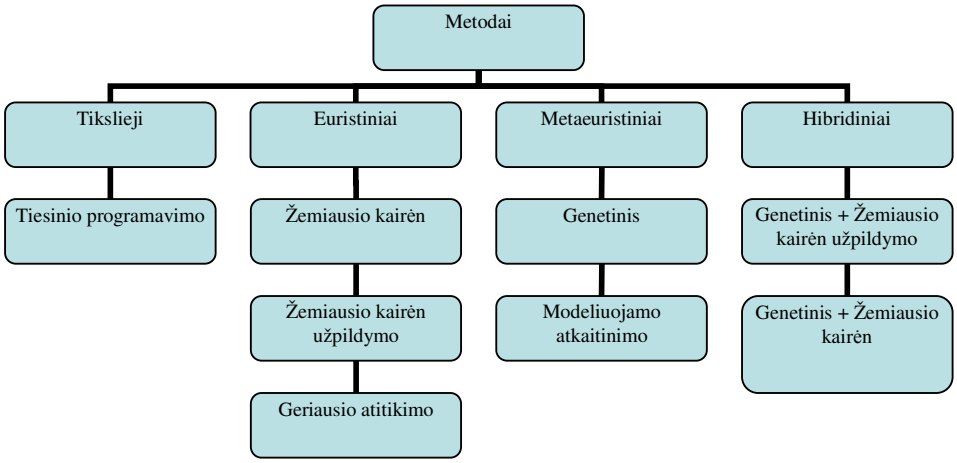
išpjaustymą [12]. Lui ir Teng (1999 m.) modifikavo šį algoritmą [13]. Burke ir Kendall (2004 m.) sukūrė "Geriausio atitikimo" metodą negiljotiniam orgononaliajam pjaustymo uždaviniui spręsti, kuris gaudavo gerus rezultatus su didelės apimties uždaviniais[5].

Metaeuristiniai metodai turi savybę nesuklysti lokaliai optimalaus sprendinio prasme, kaip tai gali atsitikti su tradicinėmis euristikomis. Metaeuristiniuose metoduose sprendinio paieškos procesas dažnai reguliuojamas žemesnio lygio euristikos. Dagli ir Hajakbari (1990 m.) panaudojo modeliuojamojo atkaitinimo metodą[14], Kroger (1995 m.) pritaikė genetinį algoritmą giljotininio pjaustymo uždaviniams spręsti, panaudodamas mažesnių stačiakampių apjungimą į vieną didesnį[6].

Hibridiniai metodai nėra tiksliai apibrėžti, nes juose dažniausiai apjungiami keli algoritmai. Kadangi kiekvienas metodas turi savų privalumų bei trūkumų, stengiamasi rasti tokią algoritmų kombinaciją, kurie bendroje sintezėje kiek galima labiau kompensuotų vienas kito trūkumus ir būtų galima pasinaudoti kiekvieno iš jų teikiamais savitais privalumais, sprendžiant uždavinį. Jakobs (1996 m.) apjungė genetinį ir "Žemiausio kairėn" algoritmus[15]. Hopper ir Turton (1999 m.) tą patį padarė su genetiniu ir "Žemiausio Kairėn Užpildymo" algoritmais[16].

Čia paminėti ir nurodyti tik vieni iš pagrindinių užsienio autorių darbai, kadangi ši problematikos sritis susilaukė nemažo susidomėjimo dėl savo praktinio pritaikymo įvairiose industrijos šakose. Kas liečia lietuvių autorius, Tomas Blažauskas yra nagrinėjęs vienmačio uždavinio pjaustymo euristinius algoritmus ir apgynęs šia tema disertaciją (2003 m.). Daugiau lietuvių autorių (nepaisant keleto bakalaurinių darbų ir straipsnių) nerasta, kad būtų pasižymėję pjaustymo tematika.

2.5 pav. Pateikiamos pagrindinės metodų klasės sutinkamos sprendžiant įvairius pjaustymo uždavinius bei keletas konkrečių algoritmų pavyzdžių, atstovaujančių tai klasei.



2.5 pav. Metodų klasifikacija

2.6.METAEURISTINIAI OPTIMIZAVIMO ALGORITMAI

2.6.1. GENETINIS ALGORITMAS

Genetiniai algoritmai (GA) ir jų sudarymo principas buvo pasiūlytas XX – ojo amžiaus 70 – aisiais metais Holland [17]. Pradedant Holland pirmuoju darbu, genetiniai algoritmai buvo sėkmingai išbandyti, sprendžiant įvairius kombinatorinio ir kitus optimizavimo uždavinius, pvz., tokius kaip tvarkaraščių sudarymo, duomenų talpinimo, biudžeto planavimo ir kt. Genetinių algoritmų veikimo principas pagrįstas evoliucijos, vykstančios gyvojoje gamtoje, t.y. natūraliosios atrankos proceso imitavimu. Pagrindinės sąvokos, kurios naudojamos modeliuojant biologinės evoliucijos procesus, yra „individas“ ir populiacija. „Individas“ yra tam tikras elementarus, daugiau neskaidomas vienetas, objektas. Didesnė ar mažesnė „individų“ grupė sudaro „populiaciją“. Dar vienas svarbus dalykas yra vadinamasis „individo tinkamumas“ – savotiška individo vertė. „Individo vertę“ galima traktuoti kaip „individo“ sugebėjimo sėkmingai prisitaikyti (prie aplinkos), išlikti ir reprodukuotis laipsnį. Šia prasme „vertingesnis“ (grynai biologiškai) yra tas „individas“, kuris sugeba geriausiai prisitaikyti (būti „stipresnis“ už kitus) ir, galbūt, palikti didesnį palikuonių („vaikų“) skaičių.

Optimizavime vietoje sąvokų „individas“, „populiacija“, „individo vertė“ naudojamos tradicinės įprastos sąvokos: „individui“ atitinka atskiras sprendinys, „populiacijai“ – sprendinių aibė (grupė, rinkinys), pagaliau, „individo vertė“ yra asocijuojama su tikslo funkcijos reikšme duotajam sprendiniui. Taip, kaip gyvosios gamtos evoliucijoje išlieka tik „vertingiausi“ („stipriausi“) „individai“, taip ir optimizuojant siekiama gauti kuo „geresnį“ sprendinį, t.y. sprendinį su kuo mažesne (ar didesne) tikslo funkcijos reikšme (žiūrint, koks optimizavimo uždavinys – minimizavimo ar maksimizavimo – yra sprendžiamas).

Genetiniuose algoritmuose individai yra vaizduojami sprendiniais, kurie yra koduojami simbolių eilutėmis – chromosomomis. Kiekvienas optimizuojamos funkcijos nežinomas kintamasis yra užkoduojamas simbolių, vaizduojančių genus, rinkiniu, kurie yra apjungiami į chromosomas. Optimizavimo metu yra ieškoma ne vieno galimo sprendinio, o jų aibės, t.y. sprendžiant uždavinį yra operuojama ne su viena chromosoma, o su jų populiacija.

Genetiniai algoritmai priklauso metaeuristinių metodų šeimai, kuriems būdingos tokios savybės:

1. Operuojama su viena ar keletu leistinų sprendinių „populiacijų“;
2. Atskiri sprendiniai iš vienos ar kelių „populiacijų“ parenkami tolimesniam nagrinėjimui, apdorojimui, teikiant pirmenybę tiems sprendiniams, kurie turi „geresnes“ tikslo funkcijos reikšmes;

3. Naudojamos euristinės procedūros, skirtos naujiems leistiniems sprendiniams formuoti, dalinant gautą populiaciją į sprendinių poras ir operuojant su šiomis poromis;
4. Naudojamos taisyklės naujiems sprendiniams generuoti iš atskirų anksčiau gautų sprendinių.

Paskutinių trijų savybių realizavimas leidžia populiaciją atnaujinti iteraciniu būdu, paliekant joje „geresniuosius“ sprendinius.

Formalizuojant genetinių algoritmų aprašymą, aukščiau minėtos bendrosios savybės susiejamos su atitinkamomis euristinėmis procedūromis. Taip (2) savybė susiejama su atrankos procedūra, (3) savybė vadinama kryžminimu, (4) savybė yra vadinama mutavimu.

Yra žinoma daug įvairių būdų, kaip atlikti „sprendinių – tėvų“ parinkimo, kryžminimo, mutavimo bei populiacijos sprendinių atnaujinimo procedūras pavyzdžiui, galima operuoti su viena didele sprendinių populiacija arba su keliomis mažesnėmis („lygiagrečiomis“, „sub - populiacijomis“ ir pan.). „Sprendinių - tėvų“ poros gali būti parenkamos, atsižvelgiant į jų tinkamumą, t.y., tikslo funkcijos reikšmę, ir į galimybes gauti „vertingus“ „sprendinius - palikuonis“. Kryžminimo bei mutavimo procedūros gali būti atliktos įvairiais būdais, beje, jos gali būti apjungtos į vieną procedūrą. Atrankos metu galima pakeisti visus einamosios populiacijos narius (sprendinius) naujai gautais „palikuonimis“ arba taikyti įvairius atrankos metodus, pavyzdžiui, turnyrinę, elitinę, proporcingumo atranką ir pan. Svarbu pažymėti, kad genetinio algoritmo realizacija labai priklauso nuo sprendinių kodavimo būdo.

Bendrais bruožais aptarsime genetinio algoritmo etapus.

Genetinė paieška pradedama sudarant pradinę populiaciją. Dažniausiai pradinė populiacija parenkama atsitiktinai, o po to ji yra optimizuojama kituose algoritmo žingsniuose, naudojant anksčiau minėtus genetinius operatorius. Jeigu sprendžiame pakankamai sudėtingą optimizavimo uždavinį, kartais verta dalį pradinės populiacijos individų generuoti taikant iš anksto apibrėžtą euristinį algoritmą, tarkim lokalios paieškos ir pan. Dažniausiai sutinkamos sprendinių – chromosomų koduotės yra dvejetainės arba sveikųjų skaičių (gali būti ir kitokios). Genu dažniausiai laikomas vienas atitinkamos koduotės simbolis (dvejetainis arba sveikasis skaičius), koduojantis vieną kintamąjį arba požymį.

Sudarius pradinę populiaciją, su ja atliekamos operacijos, modeliuojant evoliucinį procesą, kurio svarbiausi principai yra paveldimumas, mutacija ir atranka. Populiacijos evoliucionavimas dažniausiai modeliuojamas remiantis tokiomis taisyklėmis:

1. Chromosomos suskirstomos poromis, kurios su tam tikra tikimybe apsikeičia savo genų rinkinių dalimis, t.y. jos yra kryžminamos;
2. Kiekvienam populiacijos chromosomų genui taikomas mutacijos operatorius, kuris su tam tikra tikimybe gali pakeisti geną arba jo poziciją chromosomoje;

3. Kryžminimo ir mutavimo metu gautos chromosomos sudaro tarpinę populiaciją. Tikimybė chromosomai dalyvauti formuojant tarpinę populiaciją yra tuo didesnė, kuo aukštesnė ją atitinkanti tikslo funkcijos reikšmė;
4. Iš tarpinės populiacijos sudaroma nauja populiacija, atrenkant „geriausius“ individus.

Verta paminėti tai, kad čia pateikta tik viena iš galimų genetinio algoritmo vykdymo logikų. Kadangi pastaruoju metu yra sukaupta nemaža šių metodų panaudojimo praktikoje patirtis, atsiranda vis įvairesnių pasiūlymų, kaip galima būtų modeliuoti populiacijos evoliucionavimą.

Pagrindinės mutacijos ir kryžminimo funkcijos yra vertingos informacijos, esančios sprendinyje, perėmimas. Konkreti algoritmo realizacija turi užtikrinti potencialią galimybę gauti bet kokį sprendinį iš visų galimų sprendinių aibės, pasinaudojant tik mutacija ir kryžminu.

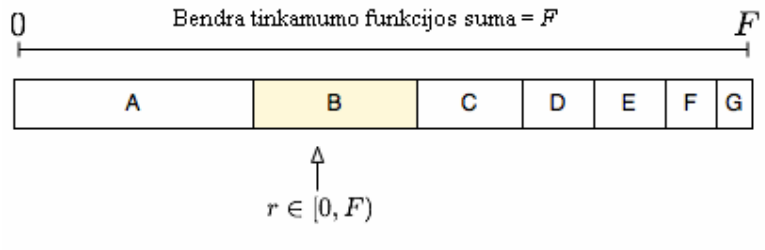
Kryžminimas leidžia efektyviai perrinkti didelį kiekį variantų (kadangi pradiniai sprendiniai dažniausiai formuojami atsitiktiniu būdu), tuo pat metu užtikrina tam tikrą sprendinių konvergavimą. Dažniausiai genetiniuose algoritmuose naudojama „dviejų tėvų“ schema, tačiau „palikuonių“ chromosomos gali būti generuojamos iš daugiau negu dviejų chromosomų. Yra naudojami įvairūs kryžminimo tipai: vieno taško, dviejų taškų, *m* – taškis ir vienalytis (homogeninis). Vieno taško kryžminime išrenkamos dvi chromosomos ir sugeneruojamas atsitiktinis kryžminimo taškas. Šis procesas pavaizduotas (2.6 pav.). Dviejų taškų kryžminimas skiriasi nuo vienataškio tik tuo, kad atsitiktinai parenkami du kryžminimo taškai ir chromosomos apsikeičia dalimis tarp tų taškų. *m* – taškio kryžminimo atveju yra parenkama *m* kryžminimo taškų. Šiais taškais chromosomos suskaldomos į *m+1* dalių ir apsikeičia tiktai lyginėmis arba nelyginėmis jų dalimis. Vienalyčio kryžminimo metu kiekvienas pirmojo tėvo genas turi 50 proc. šansą apsikeisti vietomis su atitinkamu antrojo tėvo genu.

2.6 pav. Vienataškis ir dvitaškis kryžminimas

Mutacija yra genetinis operatorius, operuojantis tik su viena chromosoma. Mutacijos operatorius dažniausiai pakeičia vieną chromosomos geną arba jo poziciją chromosomoje atsitiktiniu būdu. Jis yra pritaikomas visiems genams su tam tikra mutacijos tikimybe. Pagrindinė mutacijos užduotis yra suteikti galimybę perrinkti įvairius sprendinius arba pereiti prie tų sprendinių, kurių neįmanoma gauti kryžminimo būdu. Kita vertus, mutacija neturi sukelti chaoso ir išsigimimo populiacijoje, kurie gali įvykti, jei populiacijos tikimybė yra didelė.

Atranka, kuri yra svarbus evoliucinio proceso etapas, užtikrinantis chromosomų dalyvavimą formuojant būsimas kartas. Dalyvavimo tikimybės yra nevienodos ir priklauso nuo chromosomos vertės, kurią apsprendžia chromosomą atitinkanti tikslo funkcijos reikšmė. Todėl sprendiniai (chromosomos), kuriuos atitinka „geresnė“ tikslo funkcijos reikšmė, turi daugiau šansų dalyvauti formuojant tarpinę populiaciją. Priklausomai nuo atrankos operatorių, tarpinės populiacijos formavime gali dalyvauti ir sprendiniai, kuriuos atitinka „blogesnės“ tikslo funkcijos reikšmės, siekiant, kad šių sprendinių atranka gali lemti labai „gerų“ sprendinių atsiradimą tolesnės evoliucijos metu.

Išskiriami keli pagrindiniai atrankos būdai: turnyrinis, proporcingasis, elito ir kiti. Turnyrinėje atrankoje atsitiktinai pasirenkama iš populiacijos t chromosomų, iš kurių išrenkama geriausia. Dažniausiai turnyrai vyksta tarp dviejų individų (binariniai turnyrai, $t = 2$). Tokios atrankos metu populiacijos nereikia rūšiuoti, todėl šis algoritmas yra $O(n)$ sudėtingumo ir yra lengvai realizuojamas (n – individų skaičius populiacijoje). Proporcingoji atranka (2.7 pav.) paremta principu, kad galimybė būti išrinktam kiekvienam individui yra tiesiogiai (atvirkščiai) proporcinga jį atitinkančios tikslo funkcijos reikšmei. Šis atrankos principas suteikia galimybes atrinkti ir pačius „blogiausius“ sprendinius, taip užtikrinant genetinio fondo įvairovę.



2.7 pav. Proporcingosios atrankos schema

Elito atranka yra ignoruojantis įprastą eigą, tačiau labai efektyvus variantas, kai konstruojant naują populiaciją, leidžiama kelioms „geriausioms“ chromosomoms pereiti į naują populiacijai visai nepakitus.

Genetiniuose algoritmuose susiduriama su sprendinių „klonavimo“ problema, kai populiacijoje atsiranda daug vienodų „individų“. Tinkamai parinkus metodo parametrus, genetinis algoritmas turi užtikrinti galimybę perrinkti kiek įmanoma daugiau galimų sprendinio variantų, išvengiant didelio skaičiaus „klonų“. Tačiau vykdymo metu siekiama kuo greičiau aptikti optimalaus sprendinio traukos zoną ir patį sprendinį. Todėl genetiniuose algoritmuose svarbi algoritmo stabdymo ir galutinio sprendimo nustatymo problema. Dažniausios nutraukimo sąlygos:

1. Pasiektas nustatytas generacijų skaičius;

2. Tam skirtas biudžetas (laikas/pinigai) išnaudotas;
3. Pasiekta stabilizacijos būsena, kai nėra gaunama geresnių sprendinių;
4. Aukščiau minėtų priežasčių kombinacijos.

2.6.2. MODELIUOJAMOJO ATKAITINIMO ALGORITMAS

Šis metodas remiasi analogija su fizikiniu procesu – atkaitinimu (grūdinimu). Atkaitinimo modeliavimo metodas sukurtas remiantis analogijomis iš statistinės mechanikos, modeliuojant procesus sistemose, sudarytose iš didelio skaičiaus mažų diskrečiųjų dalelių. Modeliuojant tokios sistemos „atkaitinimą“, pradžioje sistemai yra suteikiama aukšta temperatūra, kuri palaipsniui mažinama, kol sistema „užsigrūdina“ (pereina į optimalią būseną). Šio modeliavimo pradininkai buvo (1953 m.) Metropolis, Rozenblutas ir kiti [18]. Jie pasinaudojo Bolcmano (eksponentiniu) pasiskirstymo dėsniu apibrėždami tikimybę, kad sistema, įvykus joje tam tikrai perturbacijai (perėjimui), pereis iš vieno energijos lygio (E_1) į kitą (E_2), kai temperatūra lygi T:

$$P(\Delta E,T)=\begin{cases}1, \Delta E<0\\ e^{\frac{-\Delta E}{c_B\cdot T}}, \Delta E\geq 0\end{cases} \qquad (2.6.)$$

čia $\Delta E = E_2 - E_1$, c_B - Bolcmano konstanta.

Modeliuojamojo atkaitinimo metodas taikomas diskretiems ir tolydiems uždaviniams spręsti. Svarbus šio metodo parametras yra temperatūra, mažėjanti su kiekvienu atkaitinimo proceso imitavimo žingsniu. Remiantis nuo temperatūros priklausančiu tikimybiniu patvirtinimo kriterijumi (2.6.) dar vadinamu Metropolio patvirtinimo kriterijumi, kiekviename algoritmo vykdymo žingsnyje yra priimamas surastas geresnis sprendinys, nei esamas bei su nedidele tikimybe gali būti priimtas blogesnis sprendinys. Ši savybė leidžia „ištrūkti“ iš lokaliųjų optimumo taškų, ieškant globaliojo optimumo.

Sistemos perturbacija vykdoma, generuojant „kaimyninį“ sprendinį, tokiu pat principu kaip yra vykdoma mutacija genetiniame algoritme. Skirtumas tik tas, kad pradžioje leidžiami didesni „kaimyninio“ sprendinio pokyčiai, o pabaigoje – mažesni, t.y. tikimybė vykdyti pakeitimus tiesiogiai priklauso nuo temperatūros reikšmės.

Modeliuojant sistemos „atkaitinimą“ yra svarbūs tokie veiksniai:

1. Pradinės (galutinės) temperatūros parinkimas;
2. Pusiausvyros testas;
3. Temperatūros mažinimo formulė.

Sistemos pradinė ir galutinė temperatūra apibrėžia ne ką kitą kaip bendrą bandymų (iteracijų) skaičių ir yra palaipsniui mažinama paprastai, kol tampa lygi nuliui (bet ne visą laiką). Jos reikšmė neturėtų būti nei per daug didelė nei per daug maža. Iš tikro, jei pradinė temperatūra

būtų pernelyg aukšta, tai „atkaitinimo“ procesas užsitęstų labai jau ilgai, be reikalo nagrinėjant daug „blogų“ sprendinių. Kita vertus, per daug žema pradinė temperatūra galėtų lemti per greitą konvergavimą į lokalų optimumą. Paprastai šios reikšmės nustatomos eksperimentiniu būdu.

Atkaitinimo modeliavimo algoritmuose gali būti panaudotas taip vadinamas ekvilibriumo (pusiausvyros) testas. Tai suprantama kaip stabili, be žymesnių fliuktacijų paieškos proceso būsena. Kombinatorinio optimizavimo uždavinių atveju pusiausvyros kriterijumi galėtų būti, pvz., tikslo funkcijos reikšmių svyravimų amplitudė (jeigu ji nedidelė, tai konstatuojama, kad ekvilibriumas pasiektas). Atsižvelgiant į tai, skiriami du „atkaitinimo“ tipai: homogeninis ir nehomogeninis. Pirmuoju atveju atliekama daug bandymų esant tai pačiai, fiksuotai temperatūrai; tai tęsiama tol, kol pasiekiama pusiausvyra – tuomet temperatūra sumažinama ir procesas kartojamas. Antruoju atveju temperatūra mažinama po kiekvieno įvykdyto bandymo, t.y. nevykdomas ekvilibriumo testas.

Modeliuojant „atkaitinimą“ galimi keli temperatūros mažinimo formulių variantai. Praktiškai taikomuose atkaitinimo algoritmuose plačiausiai naudojamos yra geometrinė formulė:

$$T_k = \alpha \cdot T_{k-1} \tag{2.7.}$$

ir Lundy – Mees formulė:

$$T_k = \frac{T_k}{1 + \beta \cdot T_{k-1}} \tag{2.8.}$$

Čia T_k - einamoji temperatūros reikšmė, $k = 1,2,\ldots n$, n - iteracijų skaičius, $0.8 \le \alpha \le 0.99$, $\beta << T_0$. Dažnai tam, kad palengvinti eksperimentavimą temperatūra yra tiesiogiai susiejama su iteracijų skaičiumi, tuomet jos mažinimo būdas yra akivaizdus – paprasčiausiai mažinant tikru dydžiu iteracijų skaičių.

Įdomu paminėti tai, kad kai kuriuose atkaitinimo modeliavimo algoritmų versijose temperatūra keičiama periodiškai, o ne monotoniškai mažinama. Taikoma taip vadinama „atkaitinimų“ ir „kaitinimų“ seka (reannealing), toks principas kartais suteikia papildomas lankstesnes, galimybes atkaitinimo modeliavimo algoritmams.

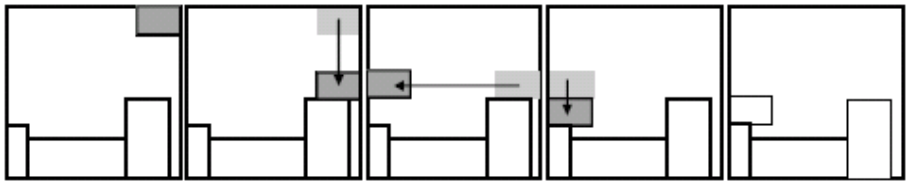
Pabaigos sąlygų variantai analogiški kaip ir anksčiau išvardinti genetiniam algoritmui.

2.7.EURISTINIAI STAČIAKAMPIŲ IŠDĖSTYMO ALGORITMAI

2.7.1. „ŽEMIAUSIO KAIRĖN“ IR „ŽEMIAUSIO KAIRĖN UŽPILDYMO“ ALGORITMAI

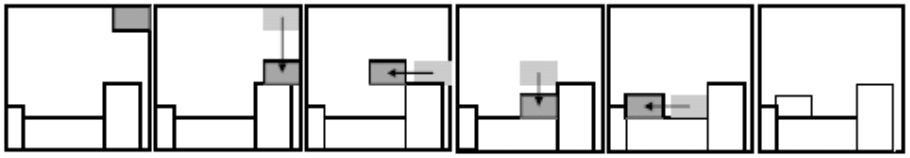
Euristiniai metodai pateikia pakankamai gerus per trumpą laiko tarpą sprendimus. Labiausiai dokumentuoti yra *„Žemiausio kairėn“* (ŽK) ir *„Žemiausio kairėn užpildymo“* (ŽKU*)* algoritmai.

Algoritmui paduodamas stačiakampių, kuriuos reikia išpjaustyti, sąrašas. Tuomet detalės yra dėliojamos ant lakšto (pjaustomo objekto) tokiu būdu: pirmiausia stačiakampis yra orientuojamas į viršutinį dešinį lakšto kampą, tada atliekami perkėlimo veiksmai iš pradžių į apačią, paskui į kairę.



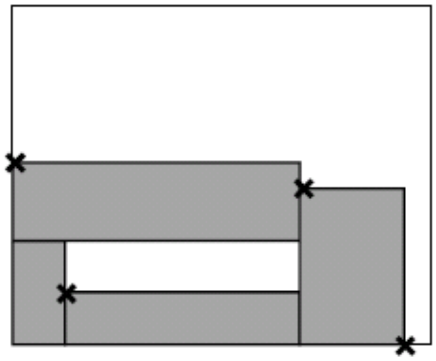
2.8 pav. ŽK veikimo principas

Sukurta pagerinta *ŽK* euristikos versija [13], suteikiančią judesiui žemyn prioritetą tokį, kad figūros perstumiamos į kairę tik tada, kai negalimas slydimas žemyn. Buvo parodyta, kad visąlaik egzistuoja mažiausiai viena stačiakampių seka, galinti duoti optimalų sprendimą.



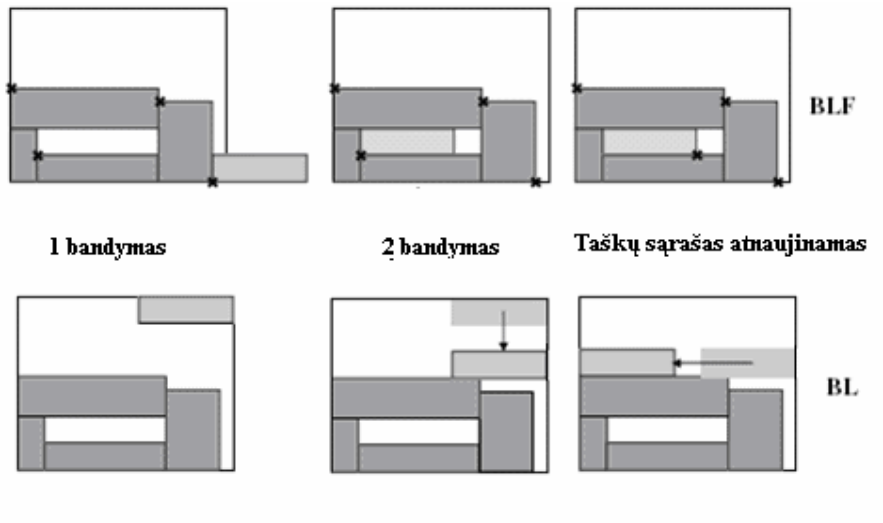
2.9 pav. Pagerinto ŽK veikimo principas

Antrasis *ŽKU* metodas yra taip pat modifikuota *ŽK* stačiakampių išdėstymo euristikos versija. Papildomai sudaromas sąrašas taškų, pradedant nuo apatinio kairiojo kampo, kurie parodo, kur būtų galima įdėti figūrą. Pjaustymo metu algoritmas pradedamas nuo žemiausio ir kairiausio taško, kuriame yra talpinama figūra. Tikrinama, ar talpinamas stačiakampis nesikerta su kokia nors kita anksčiau įdėta figūra arba lakšto kraštu. Jeigu nesikerta, figūra paliekama ir taškų sąrašas yra atnaujinamas, kitu atveju nagrinėjamas sekantis taškų sąrašo elementas (taškas), procesas tęsiamas tol, kol įmanoma įdėti figūrą taip, kad ši nesikirstų su kitomis.



2.10 pav. Galimos figūrų įdėjimo vietos

2.10 paveikslėlyje parodyti taškai, kuriose įmanoma talpinti sekančią figūrą (taškų sąrašas). Kaip minėjome, paieškos procesas pradedamas nuo žemiausiai kairiausio taško (situacija, kai taškai yra viename lygyje, imamas kairiausias taškas). Taigi akivaizdu, kad *ŽKU* metodas gali užpildyti susidariusias "skyles".



2.11 pav. ŽK ir ŽKU veikimo principų palyginimas

Susidariusi "skylė" niekada nebus užpildyta problemą sprendžiant *ŽK* metodu, tai yra *ŽKU* pranašumas. Verta paminėti, kad sprendinio kokybė priklauso nuo stačiakampių pateikimo eiliškumo. Nustatyta, kad *ŽK* algoritmo sudėtingumas yra $O(n^2)$, kai n pjaustomų detalių skaičius, atitinkamai *ŽKU* - $O(n^3)$. *ŽKU* sudėtingumas didesnis, todėl laiko sąnaudų prasme jisai dirba ilgiau, tačiau gautas sprendinys yra tikslesnis.

2.7.2. "GERIAUSIAI TINKAMO" ALGORITMAS

Burke ir Kendall (2004 m.) pristatė *"Geriausiai tinkamo"* (*"Best Fit"*) algoritmą [5]. Skirtingai nei *„Žemiausio kairėn"* ir *„Žemiausio kairėn užpildymo"* metodai, kurių veikimas priklauso nuo pradinės paduodamos užsakymo stačiakampių sekos, šis euristinis algoritmas savarankiškai pasirenka pjaustymui sekantį stačiakampį iš užsakymo sąrašo. Tai leidžia šiam metodui atlikti pakankamai informuotus sprendimus apie tai, kurį stačiakampį pasirinkti ir kurioje lakšto vietoje jį padėti. Iš principo pjaustymo samprata šiame algoritme keičiama į stačiakampių išdėstymo sampratą. Tai yra godi procedūra, kuri stengiasi pateikti geros kokybės detalių dėstinius lakštuose, remdamasi žemiausia esančia neužpildyta erdve (mūsų atveju plotu), į kurią stengiasi įdėti geriausiai atitinkančią figūrą. Yra keletas būdingų problemų tai, kas liečia algoritmo sudėtingumą ir sprendinio kokybę. Tam, kad rastume žemiausią tarpą (į kurį galima būtų įdėti figūrą), turime tirti lakšto struktūrą su jau išdėstytomis detalėmis, tai atlikus, privalome rasti geriausiai atitinkantį stačiakampį iš užsakymo sąrašo šiam tarpui. Abi šios operacijos turi būti vykdomos kiekviename algoritmo žingsnyje, todėl jos ženkliai prisideda prie algoritmo sudėtingumo.

Dėstymo proceso pradžioje turime tik tuščią žaliavos lakštą, todėl žemiausias galimas įdėjimo tarpas arba niša, bus visas lakšto ilgis (arba plotis). Kai pradedame figūrų dėstymo procesą, žemiausias tarpas keisis atitinkamai pagal padėtį ir ilgį. Geriausiai tarpą atitinkanti figūra yra visąlaik pasirenkama ir įdedama į jį. Galimi trys atvejai:

1. Stačiakampis tiksliai atitinka tarpą, t.y. tos pačios dimensijos (ilgis arba plotis)
2. Stačiakampio dimensija yra mažesnė nei tarpo
3. Nėra stačiakampio, kuris atitiktų tarpą

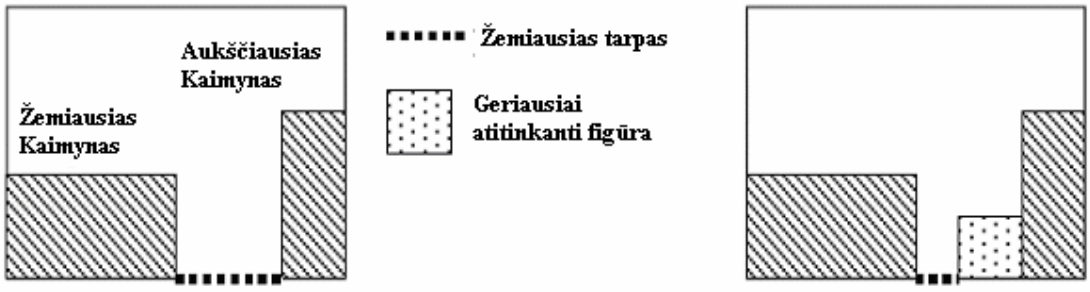
Pirmuoju atveju figūros įdėjimas yra paprastas, nes yra idealus atitikimas, jeigu yra kelios tokios figūros, tai pasirenkama didžiausią antrąją dimensiją turinti detalė,t.y. tokia, kurios plotas didžiausias. Antruoju atveju pasirenkama figūra, užimanti didžiausią tarpo dalį. Šis stačiakampis neužpildo viso tarpo, todėl reikia nustatyti kaip figūra turi būti talpinama tame tarpe. Tai vadinama *nišos užpildymo taktika*. Trečiuoju atveju esantis plotas interpretuojamas kaip *atliekų plotas.* Tai akivaizdu, jeigu nei viena iš užsakymo sąraše esančių figūrų neatitinka tarpo, tai būsimose iteracijose situacija išliks ta pati, todėl šis tarpas bus niekada neužpildytas. Skirtingai nuo *„Žemiausio kairėn"* metodo, šis algoritmas sukuria tik tokius atliekų plotus, kurių nebus įmanoma užpildyti vėlesnėse iteracijose ir skirtingai nuo *„Žemiausio kairėn užpildymo"* metodo, šis algoritmas nesaugo informacijos apie visus atliekų plotus kiekvienoje iteracijoje (arba skyles). Kitas šio algoritmo privalumas būtų toks, kad jo veikimui nereikia brangiai

kainuojančios laiko prasme „susikirtimo“ funkcijos, kuri atlieka susikirtimo testą tarp norimo įdėti stačiakampio ir jau išdėstytųjų figūrų. Akivaizdu, kad kuo daugiau figūrų jau yra išdėstyta, tų daugiau susikirtimo testų reikia atlikti, o tai įtakoja algoritmo veiksmo greitį blogąja prasme. *ŽK*, o ypač *ŽKU* atveju, šie metodai turi savybę vėlesnėse iteracijose lėtėti.

„Geriausiai tinkamo“ metode galimos trys *nišos (tarpo) užpildymo taktikos*, nurodančios kaip įdėti stačiakampį, kuris tiksliai neatitinka tarpo:

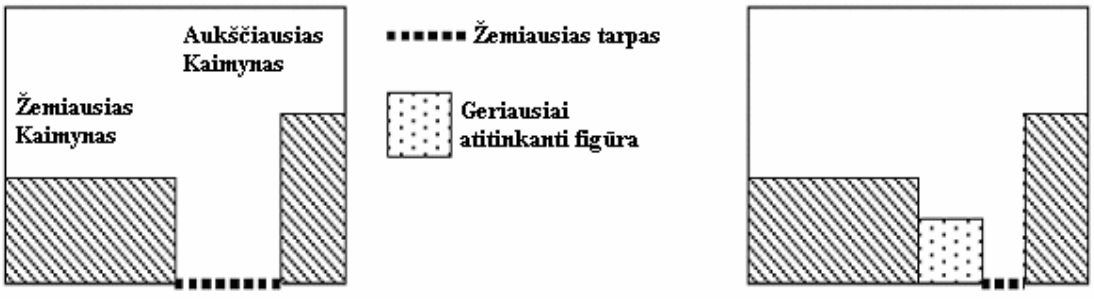
*“Kairiausio kaimyno”***.** Remiantis šia taktika, stačiakampis, tiksliai neatitinkantis tarpo, dedamas kairėje pusėje nišos. (Pastaba: galima ir dešiniausia užpildymo taktika, kuri bus veidrodinis atspindys kairiausiai).

“Aukščiausio kaimyno”. Remiantis šia taktika, nagrinėjami du stačiakampiai, kurie sudaro žemiausią tarpą. Figūra dedama prie aukščiausio stačiakampio, kuris sudaro tarpą. Jeigu žemiausią tarpą sudaro stačiakampis ir lakšto kraštas, tai figūra dedama prie lakšto krašto:



2.12 pav. Aukščiausio kaimyno tarpo užpildymo taktika

“Žemiausio kaimyno”. Remiantis šia taktika, nagrinėjami du stačiakampiai, kurie sudaro žemiausią tarpą. Figūra dedama prie žemiausio stačiakampio, kuris sudaro tarpą. Jeigu žemiausią tarpą sudaro stačiakampis ir lakšto kraštas, tai figūra dedama prie stačiakampio:



2.13 pav. Žemiausio kaimyno tarpo užpildymo taktika

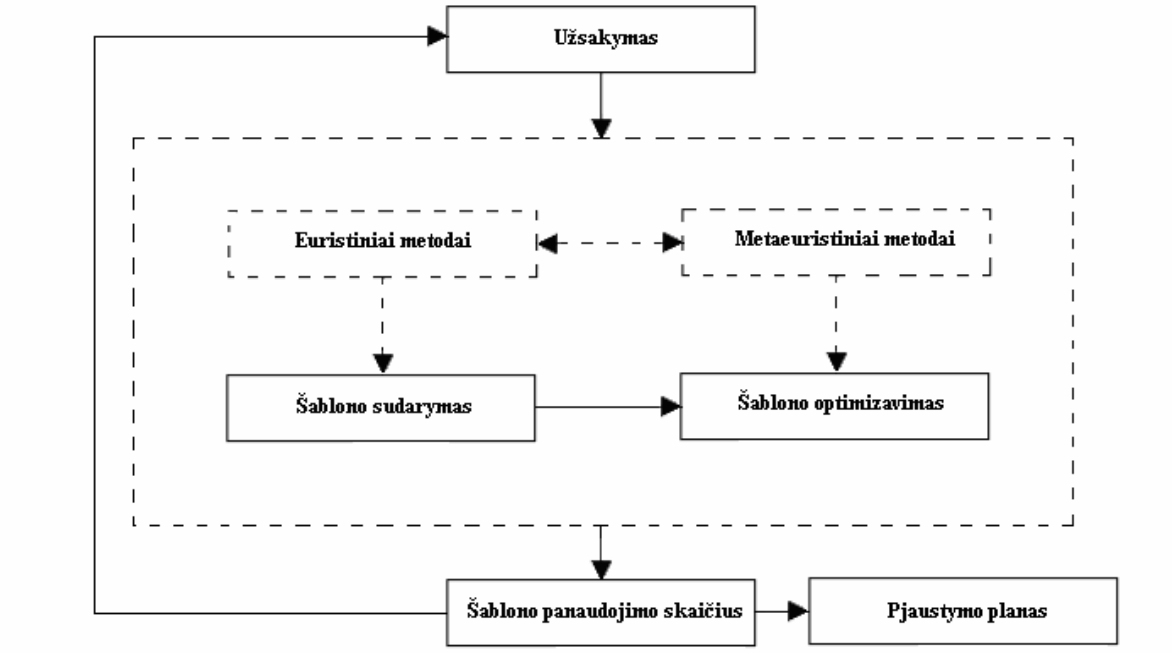
3. TIRIAMOJI DALIS

3.1.PJAUSTYMO UŽDAVINIO SPRENDIMO SCHEMA

Nagrinėsime pasirinktą (1.4. skyrelis) pjaustymo uždavinį. Kaip anksčiau minėta, pjaustymo uždavinys priklauso *NP sunkių* klasei, o tai reiškia, kad nėra greito algoritmo, kuris visais atvejais pateiktų optimalų (globaliai) sprendinį, todėl uždavinio sprendimas bus vykdomas apytikslių metodų – euristinių ir metaeuristinių – pagalba.

Pagrindinė uždavinio sprendimo schema pateikta 3.1 paveikslėlyje. Duotam figūrų užsakymui išpjauti, sudarysime pjaustymo šablonus. Pjaustymo šablonas suprantamas kaip tam tikras medžiagos lakšto išpjovimo būdas. Pjaustymo planas yra visų sudarytų šablonų visuma, kuri reprezentuoja visą užduoties sprendimą duotajam užsakymui. Įvykdžius pjaustymo planą, bus išpjauta atitinkamai - dalis užsakymo figūrų arba visos ir galios sąlyga, kad atliekų kiekis minimalus (arba lokaliai minimalus). Tai priklausys nuo to, koks bus pradinis lakštų kiekis ir kaip sėkmingai pavyks išspręsti uždavinį.

Šablono sudarymo užduotį vykdys euristiniai metodai. Negiljotininis pjaustymas yra analogija figūrų išdėstymui ant duoto medžiagos lakšto, todėl vėliau minimos sąvokos “išpjauti“, „išdėstyti“, „talpinti“ reiškia tą patį. Svarbu suvokti tai, kad euristinių pjaustymo metodų veikimas pagrįstas tam tikromis iš anksto nustatytomis taisyklėmis, kurias vykdant nuosekliai arba su tam tikru ciklišku gaunamas rezultatas. Nepaisant to, kad tam tikrais išskirtiniais atvejais tas rezultatas gali būti ir optimalus, tačiau tai yra tik vieno vykdymo, o ne paieškos proceso rezultatas. Dauguma euristinių pjaustymo metodų nedaro jokio sprendinių perrinkimo, nenagrinėja jų paieškos erdvės ir nelygina jų tarpusavyje, todėl šių metodų negalime vadinti optimizavimo algoritmais. Nepaisant to, sprendinio optimalumui jie daro didelę įtaką, nes jų pagalba konstruojamas pats sprendinys ir pateikiamas jo kokybės įvertinimas. Kad tai pavyktų padaryti, šie metodai turi būti labai specifiški ir priklausomi nuo pačio uždavinio ir jo formuluotės. Dėl šios priežasties euristiniai pjaustymo algoritmai praranda savo universalumą.



3.1 Bendra pjaustymo uždavinio sprendimo schema

Dėl minėtų euristinių pjaustymo algoritmų ypatumų, šablono optimizavimui pasirinkti metaeuristiniai metodai, kurie vykdo sprendinių perrinkimą ir palyginimą, taip pat yra universalūs ir tinka ne būtinai pjaustymo užduotims spręsti.

Kadangi euristiniai metodai labai specifiški, o metaeuristiniai pakankamai universalūs, tenka juos apjungti, nes šiame darbe pasirinkti metaeuristiniai (genetinis ir modeliuojamo atkaitinimo) algoritmai patys savarankiškai negalėtų išspręsti užduoties. Tuo tarpu, euristiniai gali tai padaryti. Todėl uždavinio sprendimas gali būti vykdomas ir be optimizavimo, tiesiog sudarant šabloną ir jį priskiriant pjaustymo planui, kartais netgi gaunami panašūs rezultatai lyginant su vykdymu optimizuojant.

Užsakymo detalių asortimentas būna labai įvairus. Gali tekti susidurti su dideliais kiekiais stačiakampių, kurių ilgis ir plotis vienodi, tokiu atveju bus didelis kiekis tų pačių šablonų. Kyla būtinybė tokius šablonus apjungti, t.y. panaudoti keletą kartų. Todėl sukonstruotam šablonui nustatysime jo panaudojimo skaičių.

Šabloną taikysim lakštams remdamiesi tokia logika:

1. Sukursime šabloną vienam lakštui, remdamiesi ankščiau išvardintomis šablono sudarymo procedūromis;
2. Tarkime, kad sudaryto šablono stačiakampių kiekiai $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$, čia $k \le n$, $i = 1,2,...,n$, kur n – stačiakampių tipų skaičius užsakyme;
3. Atitinkamai parenkame užsakymo detalių kiekius pagal tai, kokie detalių tipai yra išpjauti šablone $q_{i_1}, q_{i_2}, \ldots, q_{i_k}$;

4. Apskaičiuojame

$$m = \left\lfloor \min\left\{ \frac{q_{i_j}}{a_{i_j}} \right\} \right\rfloor \qquad (2.6.1.)$$

Čia $j = 1,2,\ldots,k$, $\lfloor\ \rfloor$ - apvalinimo žemyn operacija;

5. Vadinasi, sudarytą šabloną lakštams pjauti taikysim *m* kartų.

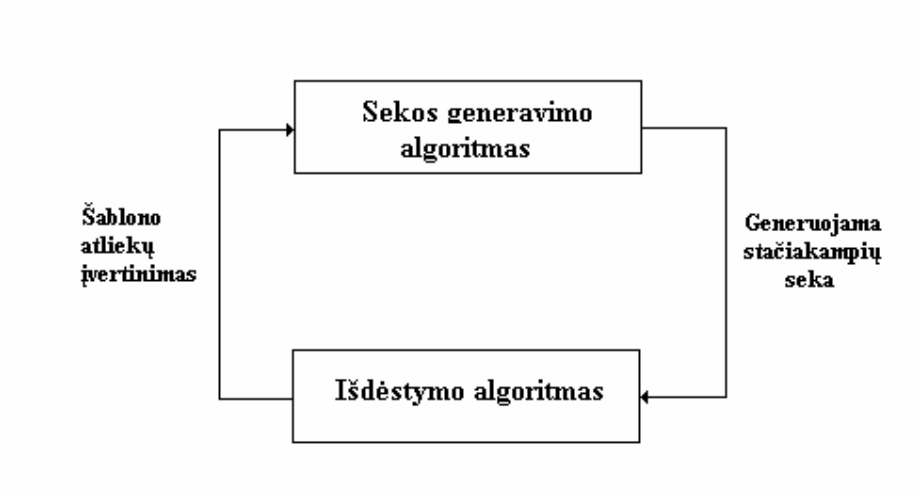
Šablono panaudojimo skaičių *m* nustatome, išdėsčius visus įmanomus stačiakampius ant medžiagos lakšto. Tuomet atitinkamai sumažinami užsakymo kiekiai ($q_{i_j} - a_{i_j}$).

3.2.HIBRIDINIO ALGORITMO PRINCIPAS

Pjaustymo šablono sudarymas grindžiamas dviem etapais:

- Generuojama stačiakampių seka;
- Stačiakampiai išdėstomi ant medžiagos lakšto.

Šiuos du etapus vykdys atskiri algoritmai, o galutinis rezultatas (pjaustymo šablonas) bus abiejų algoritmų darbo pasekmė. Nepaisant to, kad bus sprendžiamos iš pažiūros dvi skirtingos užduotys tuo pačiu metu, tačiau algoritmų veikimas priklausys vienas nuo kito. Todėl iškyla būtinybė apjungti algoritmus į vieną, t.y. juos hibridizuoti.



3.2 pav. Hibridinio algoritmo schema

Taigi pirmasis algoritmas generuos tam tikrą sprendinio variantą, o antrasis jį interpretuos ir pateiks jo kokybinį įvertinimą.

Pradinė stačiakampių seka bus parinkta atsitiktiniu būdu, panaudojant euristinį "Geriausio tinkamo" (GT) arba „Žemiausio kairėn užpildymo" algoritmus, sekančių sekų konstravimą vykdys metaeuristiniai genetinis (GA) ir modeliuojamojo atkaitinimo (MA) metodai.

Pjaustymas bus interpretuojamas kaip stačiakampių tam tikros sekos išdėstymo apribotame plote (kurio ilgis ir plotis yra ne kas kita kaip pradinių duotų žaliavos lakštų dimensijos) uždavinys. Šią problemą atskiru atveju (pradinės stačiakampių sekos generavimo atveju) spręs GT algoritmas, kitais atvejais euristiniai "Žemiausio Kairėn Užpildymo" (ŽKU) arba "Žemiausio Tarpo" (ŽT) metodai. Šie algoritmai taip pat pateiks kiekvienos sekos išdėstymo kokybinį įvertinimą, kuris apibrėžiamas formule:

$$\frac{\sum_{i=1}^{n_0} w_i \cdot l_i}{W \cdot L} \qquad (3.1.)$$

Čia n_0 – išdėstytų stačiakampių skaičius ant medžiagos lakšto, $n_0 \le n$, kur n – bendras stačiakampių skaičius (viso užsakymo), w_i, l_i, W, L atitinkamai apibrėžti 1.4. skyrelyje.

Taigi uždavinys bus sprendžiamas kompleksiškai apjungiant (hibridizuojant) tam tikrus aukščiau išvardintus algoritmus. Savo darbo tiriamojoje dalyje nagrinėsim 3 euristinius algoritmus:

1. GT;
2. ŽKU;
3. ŽT.

Taip pat 4 hibridinius algoritmus:

1. GA + ŽT (Genetinis algoritmas apjungtas kartu su „Žemiausio Tarpo" algoritmu ir t.t.);
2. GA + ŽKU;
3. MA + ŽT;
4. MA + ŽKU.

Šiame darbe minėti algoritmai buvo modifikuoti pradinėms sąlygoms generuoti panaudojus GT ir ŽKU algoritmą, nes iki tol dažniausiai jų parinkimas buvo vykdomas atsitiktiniu būdu. Taip pat ŽT algoritmas yra originali GT algoritmo modifikacija, pritaikyta atsitiktinių sekų atvejui.

3.3.PJAUSTYMO UŽDAVINIO ALGORITMŲ REALIZACIJA

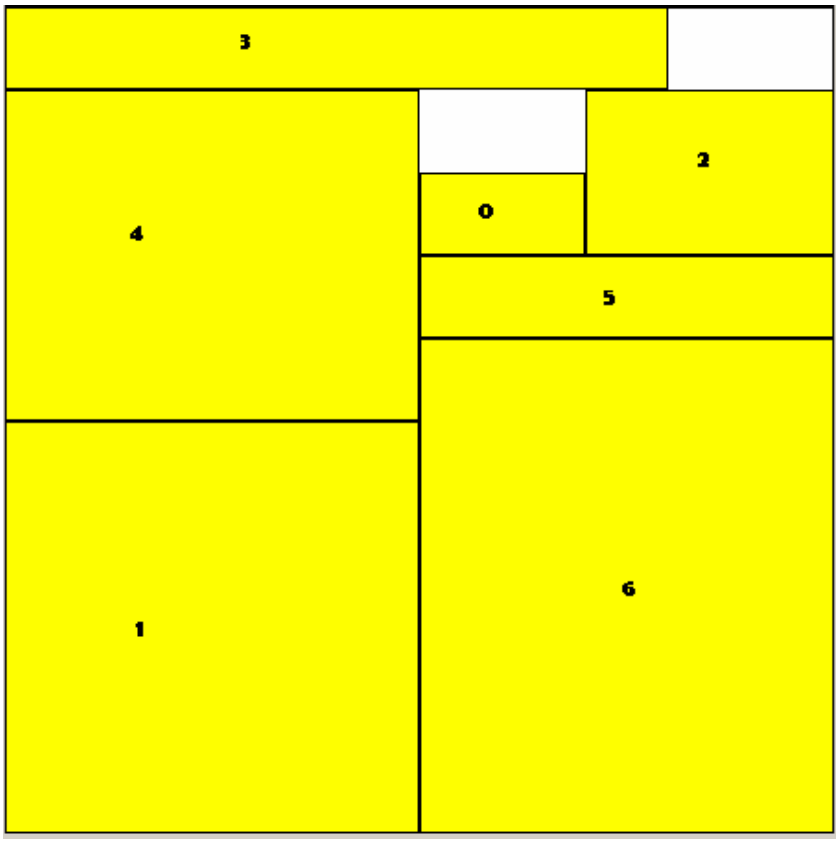
3.3.1. GENETINIS ALGORITMAS

Kaip minėta 1.6.1 skyrelyje, genetinio algoritmo įgyvendinimo schemos gali būti labai įvairios, todėl tinka daugeliui uždavinių spręsti. Nagrinėsime šio metodo pritaikymą 1.4. skyrelyje apibrėžtam pjaustymo uždaviniui.

Visi stačiakampiai, kuriuos reikės išpjauti, yra unikaliai indeksuojami, t.y. $(w_i, l_i) := j$, kur $i = 1,2,...,n$; $j = 0,1,...,N-1$; $N = \sum_{i=1}^{n} q_i$. Čia simbolis „:=“ žymi priskyrimo operaciją, w_i, l_i, q_i - apibrėžti 1.4. skyrelyje.

Taigi genas yra suprantamas kaip indekso reikšmė, kuri atitinka tam tikrą stačiakampį. Indekso reikšmių seka sudaro chromosomą arba „individą“, kuri reiškia tam tikrą pjaustymo šabloną (būdas išpjauti tam tikrą medžiagos lakštą). Tokiu būdu pjaustymo uždavinys modeliuojamas kaip sekos indeksų perstatymo uždavinys. Idėja paprasta - keičiant indeksų eiliškumą suformuluojama nauja seka, kuri atitinka tam tikrą sprendinį. Taip suformuojama chromosoma, kurią interpretuoja pasirinktas stačiakampių išdėstymo algoritmas, t.y. išdėsto, pateikia šablono grafinę interpretaciją bei grąžina atliekų kiekį, kuris yra (3.1.) kokybinis sprendinio – šablono įvertinimas.

Pavyzdžiui tarkime, kad mums reikia išpjauti 7 įvairaus dydžio stačiakampius iš medžiagos lakšto ir genetinis algoritmas sprendimo eigoje panaudojęs kryžminimo operatorių sukūrė „vaiką“ (chromosomą) – {1,6,4,5,0,2,3}. Kadangi chromosomos genai (sekos indeksai) atitinka tam tikrą stačiakampį, tai reškia ne ką kitą, kaip tam tikrą figūrų pjovimo eiliškumą. Šią indeksų seką toliau nagrinėja atitinkamas išdėstymo algoritmas, kuris pateikia taip vadinamą chromosomos interpretaciją. Kadangi chromosomą mes apibrėžiame kaip tam tikrą sprendinį - pjaustymo šabloną, tai gaunamas rezultatas yra išdėstyti stačiakampiai ant medžiagos lakšto arba, kitaip tariant, išpjauti iš to lakšto (pav. 3.3.) paveikslėlyje matyti išpjauti stačiakampiai apriboti juodomis linijomis ir nuspalvinti geltona spalva, likęs baltas plotas yra interpretuojamas kaip atliekos. Svarbu tai, kad chromosomoje yra talpinami viso užsakymo (visi) stačiakampiai, kuriuos reikės išpjauti. Mūsų nagrinėjamo pavyzdžio atveju medžiagos lakštas yra pakankamai didelis, kad sutalpintų viso užsakymo stačiakampius.



3.3 pav. Chromosomos samprata

Kai medžiagos lakštas nesutalpina visų užsakymo figūrų ir genetiniame algoritme operuojama su daugeliu chromosomų, susiduriame su kintamo ilgio chromosomomis. Tam kad supaprastinti šitą problemą, pasiūlysime kiekvienam medžiagos lakštui:

1. Chromosomoje saugoti viso užsakymo stačiakampių indeksus;
2. Chromosomą padalinti į du regionus: išpjautų ir neišpjautų stačiakampių indeksai.

3.3 paveikslėlyje duotas 11 stačiakampių užsakymas, taigi chromosomoje saugomi viso užsakymo stačiakampiai. Geltona spalva pažymėti stačiakampiai, kuriuos pavyko išpjauti (sutalpinti) iš duoto medžiagos lakšto. Balta spalva žymi stačiakampius, kurių nepavyko išpjauti, t.y. šablono atliekas.

3.4 pav. Chromosomos padalijimas į regionus

Taip apibrėžus chromosomą supaprastėja kryžminimo ir mutacijos operatorių logika. Algoritmo vykdymo pradžioje pasirenkamas tam tikras populiacijos dydis, kryžminimo ir mutacijos tikimybė, iteracijų (epochų) skaičius, elitinės atrankos buferio dydis. Tai yra pagrindiniai genetinio algoritmo parametrai, kurie nustatomi eksperimentiniu būdu. Šių parametrų nustatymas yra labai svarbus, nes tai lemia sprendinių kokybę. Kiekvienam pjaustymo

uždaviniui egzistuoja specifinis parametrų rinkinys, geriausiai išsprendžiantis problemą algoritmo darbo laiko trukmės ir sprendinio tikslumo prasme, todėl kiekvienam uždaviniui specifiškai neieškosime geriausios parametrų konfigūracijos. Tokiu atveju verta išskirti tam tikras grupes (pagal uždavinio apimties dydį) ir joms specifikuoti parametrų rinkinius.

Pradinę populiaciją generuosime dviem būdais:

1. Atsitiktiniu;
2. Kombinuotu.

Pirmasis dažniausiai pasitaikantis genetinių algoritmų panaudojimo praktikoje būdas. Pradinės populiacijos individai paprasčiausiai konstruojami mutacijos operatoriaus pagalba. Antrasis būdas paremtas algoritmų hibridizavimo idėja, vienas populiacijos individas bus sukonstruotas GT arba ŽKU algoritmo pagalba, o likusieji mutacijos.

Atranka vykdoma proporcingoji ir elitinė (1.6.1.1. skyrelis). Proporcingosios atrankos atveju pjaustymo šablonai (chromosomos) su mažesniu atliekų kiekiu turi didesnius šansus dalyvauti naujos kartos formavime, t.y. kiekvienai chromosomai $i = 1,2,...,PopDyd$ (čia $PopDyd$ – pasirinktas populiacijos dydis) nustatoma dalyvavimo tikimybė p_i:

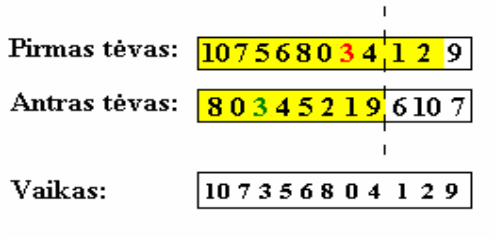
$$p_i = \frac{\frac{1}{a_i}}{\sum_{j=1}^{PopDyd} \frac{1}{a_j}} \qquad (3.2.)$$

Čia $a_i \neq 0$ yra atliekų kiekis įvykdžius i – *tąjį* šabloną. Jeigu šis dydis tampa lygus 0, tuomet procesas stabdomas, nes pasiektas optimalus stačiakampių išdėstymas. Elitinė atranka vykdoma: išrikiuojama populiacija pagal individų tinkamumą (atliekų kiekio mažėjimo kryptimi), pagal tai koks yra elitinės atrankos buferio dydis, atitinkamai perkeliamas skaičius individų kaip kandidatai į naują populiaciją be jokių pakeitimų.

Apibrėžus chromosomos struktūrą (3.4 pav.), pritaikytas vienalytis kryžminimo operatorius, kurio veikimo principas: parenkamos dvi chromosomos kryžminimui, paeiliui einama per jų genus ir vykdomas antrojo tėvo geno įterpimas su tikimybe 0,5. Paprastai generuojamas skaičius X ~ T[0,1], kai x < 0.5, tai vykdomas įterpimas. Vaikas, kuris bus kandidatas patekti į naują populiaciją, formuojamas pagal šias taisykles:

1. Įvyksta įterpimo įvykis, nustatoma įterpimo pozicija;
2. Tam, kad išvengti dublikatų, įterpiamo indekso reikšmė pašalinama iš pirmojo tėvo;

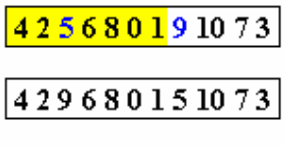
3. Antrojo tėvo reikšmė įterpiama pirmajam pagal nustatytą poziciją;
4. Taip modifikuotas pirmasis tėvas tampa vaiku.



3.5 pav. Kryžminimo schema

Kryžminimo schema pateikta (3.5 pav.). Atrankos metu pasirinktos dvi chromosomos, apibrėžiamas nagrinėjamų reikšmių regionas (juoda punktyrinė linija). Žalia spalva pažymėtas indeksas bus įterptas į pirmąjį tėvą, prieš tai vykdomas išėmimas (raudona spalva).

Mutacija vykdoma su tam tikra tikimybe vienai chromosomai ir yra pagrįsta, atsitiktinai pasirinktų genų sukeitimu vietomis. Keitime dalyvauja visi užsakymo genai nepriklausomai nuo to ar jie priklauso išpjautų ar neišpjautų stačiakampių regionui (2.2.1.4 pav.).

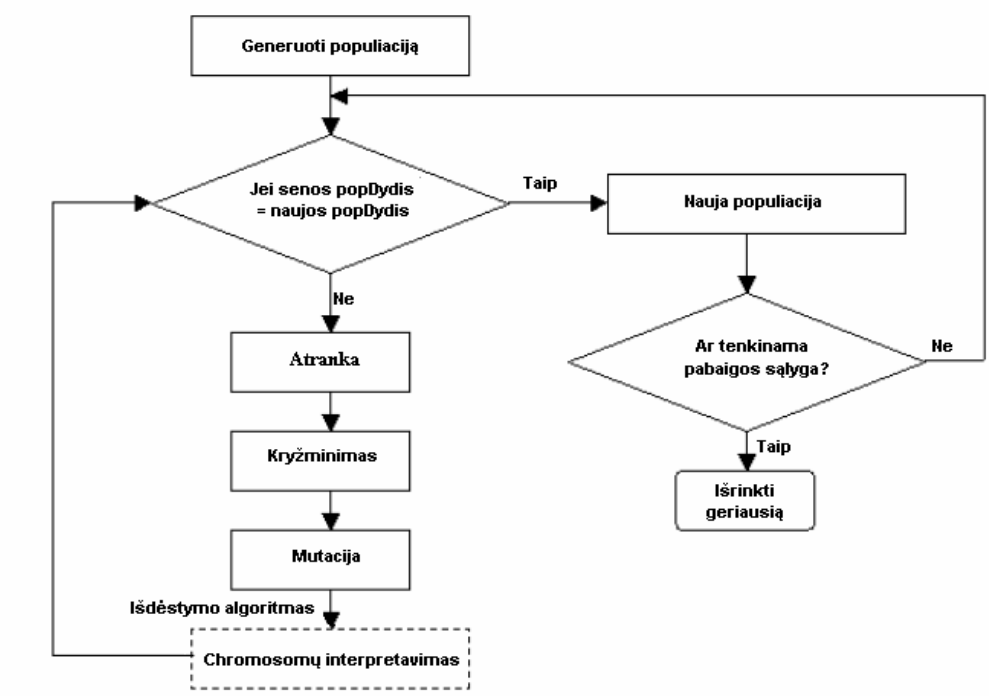


3.6 pav. Mutacijos schema

Perėjimas iš „tėvinės“ populiacijos prie naujos vyksta tokiu principu: „tėvinės“ ir kandidatų populiacijos nariai, gauti kryžminimo ir mutacijos pagalba, apjungiami kartu, išrikiuojami atliekų mažėjimo linkme, tuomet pirmieji *PopDyd* nariai tampa naujos populiacijos nariais.

Algoritmas baigia savo vykdymą po tam tikro iš anksto nustatyto epochų skaičiaus arba kai surandamas šablonas, kurio atliekų kiekis lygus 0.

(3.7 pav.) pateikiama bendra algoritmo schema. Vykdomi du ciklai vidinis skirtas naujų populiacijos sprendinių konstravimui ir išorinis naujoms populiacijoms generuoti.



3.7 pav. Bendra genetinio algoritmo schema

3.3.2. MODELIUOJAMO ATKAITINIMO ALGORITMAS

Teoriniai modeliuojamojo atkaitinimo pagrindai pateikti skyrelyje 1.6.2. Kas liečia sprendinio kodavimą - toks pat kaip ir chromosomos kodavimas genetiniame algoritme.

„Kaimyninis“ sprendinys generuojamas mutacijos pagalba, tik skirtumas nuo genetinio algoritmo mutacijos tas, kad mutacijos vykdymo tikimybė iš pradžių lygi 1 ir mažėjant sistemos temperatūrai, taip pat mažėja, t.y. $p_{kaim} = \frac{i}{n}$, čia n – nustatytas iteracijų skaičius, $n = n, n-1, ..., 0$.

Sistemos temperatūra tiesiogiai siejama su iteracijų skaičiumi. Taip pat nustatomas kitas dydis – vidinių iteracijų skaičius, kur leidžiama tęsti kaimyno paiešką su ta pačia fiksuota temperatūros reikšme (analogija ekvilibriumo testui aprašytam 1.6.2. skyrelyje). Taigi šis algoritmas turi du parametrus – iteracijų ir vidinių iteracijų skaičių. Vidinių iteracijų skaičius gali būti suprantamas panašiai, kaip genetinio algoritmo populiacijos dydis. Šie parametrai taip pat nustatomi eksperimentiniu būdu.

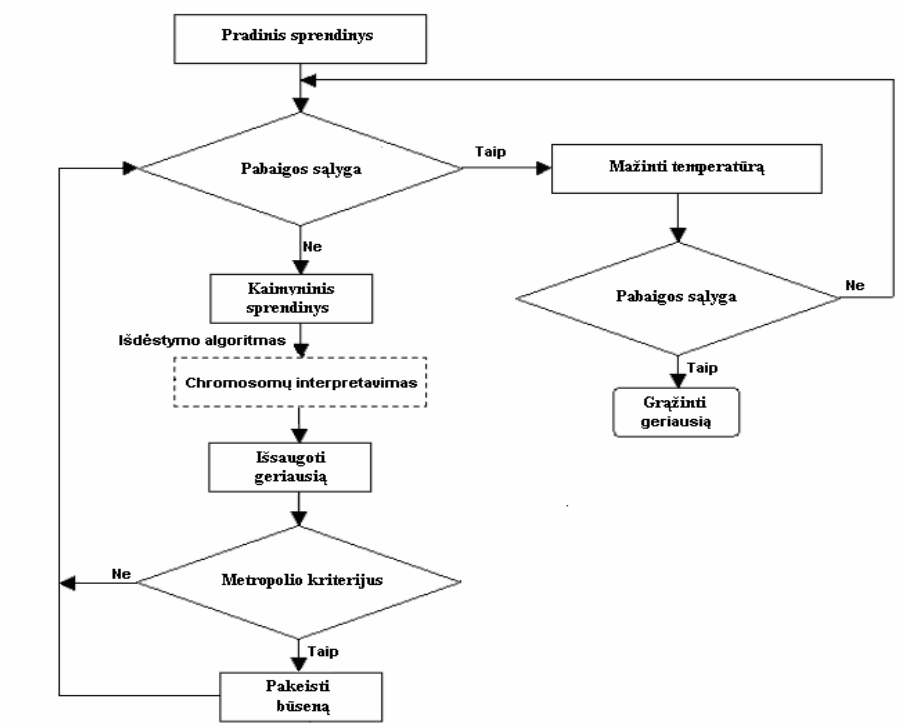
Pradinis sprendinys generuojamas atsitiktiniu būdu (mutacijos operatoriaus pagalba), kitas būdas, panaudojant GT arba ŽKU algoritmą, skirtumas tas, kad modeliuojamojo atkaitinimo algoritmas visą laiką operuoja vienu sprendiniu, o genetinis sprendinių populiacija.

Perėjimas iš esamo sprendinio į kaimyninį paremtas Metropolio patvirtinimo kriterijumi. Generuojamas atsitiktinis dydis *X~T[0,1]*, jeigu jo realizacija $x \le P$, kur

$$P(\Delta E,T)=\begin{cases}1,\Delta E<0\\ e^{\frac{-\Delta E}{T}},\Delta E\geq 0\end{cases} \qquad (3.3.)$$

Čia $\Delta E = a_{kaim} - a_0$, a_{kaim} kaimyninio šablono atliekos, a_0 einamo šablono atliekos; tuomet vykdomas perėjimas iš esamo šablono prie kaimyninio, kitu atveju nevykdomas. Jeigu sukonstruojamas geresnis sprendinys ($\Delta E < 0$), perėjimo tikimybė visą laiką P = 1, tačiau blogesni sprendiniai ($\Delta E \geq 0$) irgi gali būti priimami su perėjimo tikimybe $P(\Delta E,T) = e^{\frac{-\Delta E}{T}}$, kur $\lim_{T \to 0} P(\Delta E,T) = 0$. O tai reiškia, kad algoritmo vykdymo pabaigoje egzistuoja nykstamai maža tikimybė, kad blogesni sprendiniai bus priimti. Kai $\Delta E \geq 0$, tikimybės funkcija monotoniškai mažėja, mažėjant temperatūrai *T*. Taip pat viso vykdymo metu išsaugomas geriausias sugeneruotas sprendinys.

Algoritmo vykdymo metu sistema nuolat yra „aušinama“, kol galiausiai užsigrūdina, t.y. pereina į būseną, kuriai reikalingas minimalus vidinės energijos kiekis. Algoritmas baigia savo vykdymą kai sistemos temperatūra pasidaro lygi 0 arba kai surandamas šablonas, kurio atliekų kiekis lygus 0. Bendra algoritmo schema pateikta 3.8 paveikslėlyje.



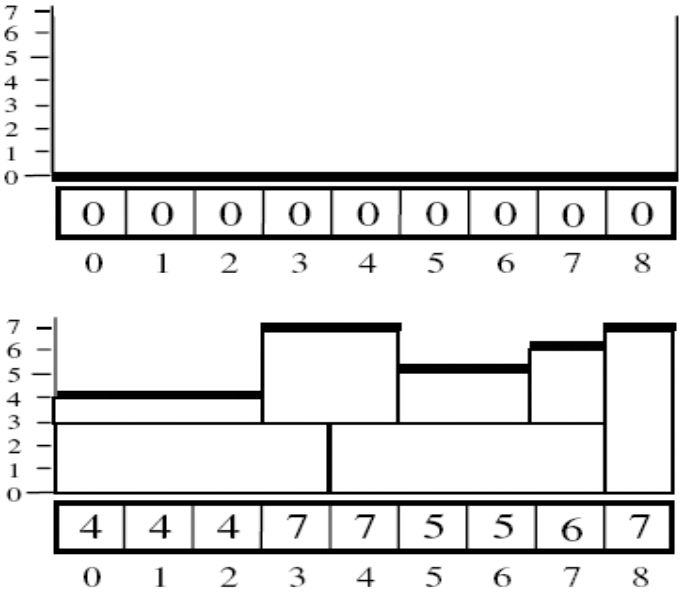
3.8 pav. Modeliuojamo atkaitinimo algoritmo schema

3.3.3. „GERIAUSIAI TINKAMO“ ALGORITMAS IR JO MODIFIKACIJA

Vykdant *„Geriausiai Tinkamo“* algoritmą, kiekvienos iteracijos metu turi būti įvykdytos dvi paieškos: geriausiai atitinkančios figūros ir žemiausiai esančio tarpo, į kurį galima būtų įdėti tą figūrą. Tai pakankamai brangios operacijos laiko prasmės atžvilgiu, todėl turi būti kiek įmanoma supaprastintos. Algoritmo pradiniai duomenys yra sąrašas bet kokia tvarka išsidėsčiusių stačiakampių, kuriuos reikia išpjauti iš lakšto, kurio išmatavimai taip yra pateikiami.

Kaip žinome, kai kurie algoritmai, tokie kaip *ŽK* ir *ŽKU* saugo informaciją apie padėties taškus, kurie nurodo vietą, kur galima butų įdėti detalę, be to naudoja „susikirtimo“ funkciją (overlap function) tam, kad išvengtų taip vadinamų detalių kolizijų. Todėl šitie metodai turi padėties taškų sąrašus, kurie turi būti pastoviai peržiūrimi, todėl kolizijų aptikimas yra pakankamai brangi operacija laiko prasmės atžvilgiu.

„Geriausiai Tinkamo“ algoritme medžiagos reprezentavimui panaudotas vienmatis *lygių masyvas*. Kiekvieno lygių masyvo elemento indeksas lygus koordinatei *x*, o pačio elemento reikšmė yra lygi jau išdėstytų detalių aukščiui koordinatės *x* atžvilgiu.



3.9 pav. Lygių masyvo samprata

3.9 paveikslėlyje pavaizduota situacija pjaustymo pradžioje ir kai jau išpjautos septynios detalės. Kaip matome lakšto ilgis yra devyni vienetai.

Taigi žemiausio tarpo koordinatę galime rasti, nustačius mažiausio lygių masyvo elemento reikšmės indeksą. Tarpo ilgis randamas suskaičiavus, kiek paeiliui yra viena kitai lygių masyvo elementų reikšmių. Remiantis 3.9 paveikslėliu matome, kad x = 0, o tarpo ilgis lygus 3.

Kita problema yra pradinis užsakymo stačiakampių sąrašas, kuris yra apibrėžiamas sveikųjų skaičių pora *(ilgis; plotis)*. Jeigu šis sąrašas nesurikiuotas, tuomet visi *n* stačiakampių turi būti patikrinti tam, kad rasti geriausiai atitinkančią tarpą figūrą. Problemos sprendimas būtų prieš pradedant pjaustyti vieną kartą surikiuoti visą užsakymo sąrašą, tai sumažintų detalių tikrinimo skaičių iki ½ *n* [5].

Iš pradžių pasukame kiekvieną stačiakampį, kurio ilgis yra mažesnis už plotį, pavyzdžiui:

{(3,5), (5,2), (1,1), (7,3), (1,2) } → { (5,3), (5,2), (1,1), (7,3), (2,1) }

Toliau stačiakampius rikiuojame ilgių ir pločių mažėjimo linkme, t.y. jeigu ilgiai lygūs tada pločių mažėjo kryptimi vykdome rikiavimą:

{ (5,3), (5,2), (1,1), (7,3), (2,1) } → { (7,3), (5,3), (5,2), (2,1), (1,1) }

Taip paruoštas stačiakampių sąrašas gali būti perduotas pjaustymo procedūrai, kuriai nereikės peržiūrinėti viso sąrašo kiekvienos iteracijos metu, tam kad rastų geriausiai atitinkančią figūrą.

Pavyzdžiui tarkime, kad mums reikia figūros 6 vienetų ilgio tarpui užpildyti. Tikrinamas pirmasis sąrašo stačiakampis (7,3), pastebėsime, kad šis stačiakampis galėtų užpildyti 3 tarpo vienetus, jei būtų pasuktas. Antrasis sąrašo stačiakampis (5,3) gali užimti 5 tarpo vienetus. Šia figūra paieška sąraše ir yra baigiama, nes visi likusieji elementai turi lygias arba mažesnes dimensijas už 5.

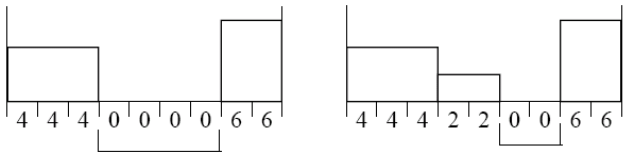
Tarkime, kad šį kartą turime 4 vienetų tarpą. Tikrinamas pirmasis sąrašo stačiakampis (7,3), jis gali užpildyti tarpą 3 vienetais, jeigu būtų pasuktas. Antrasis sąrašo stačiakampis (5,3), taip pat kaip ir pirmasis, tačiau pirmenybė teikiama didesnėms figūroms, todėl paieška tęsiama. Trečiasis sąrašo stačiakampis (5,2), tikrai nėra geresnis už pirmąjį. Paieška turi būti tęsiama, nes dar gali būti sąraše stačiakampių, kurių ilgis lygus 4 vienetams. Ketvirtasis sąrašo stačiakampis (2,1), šio stačiakampio ilgis yra blogesnis, nei pirmojo, todėl paieška nutraukiama ir grąžinamas pirmasis stačiakampis.

Jeigu randama detalė, kurios ilgis arba plotis atitinka tarpo ilgį, paieška nutraukiama ir grąžinama būtent ši detalė. Tai sumažina paieškos proceso laiką ir dėl sąrašo struktūros, leidžia pradžioj išpjauti didesnes detales. Bendrai yra geriau pjaustyti figūras su didesnėmis dimensijomis anksčiau, negu pabaigoje, nes jos gali išsikišti (sudaryti „bokštus“) ir smarkiai pabloginti sprendinio kokybę [5].

Pjaustymo etapo pradžioje tikrinamas lakštas, tam, kad rasti žemiausią tarpą (pradžioje x = 0, o tarpo ilgis lygus lakšto ilgiui). Tuomet peržiūrimas stačiakampių sąrašas tam, kad rasti geriausiai tinkantį stačiakampį. Rasta detalė įdedama atitinkamai remiantis pasirinkta nišos užpildymo taktika (mūsų atveju kairiausio kaimyno). Tuomet nustatomos stačiakampio koordinatės ir jis išimamas iš stačiakampio sąrašo. Galiausiai padidinamos atitinkamos lygių

masyvo elementų reikšmės stačiakampio pločio reikšmės dydžiu. Procesas tęsiamas: randamas žemiausias tarpas ir jo ilgis; randamas geriausiai jį atitinkantis stačiakampis, nustatomos jo koordinatės, išimamas iš stačiakampių sąrašo ir atnaujinamos atitinkamos lygių masyvo elementų reikšmės. Jeigu rastas stačiakampis nepilnai atitinka tarpą, nereikia iš naujo identifikuoti tarpo parametrų, nes pjaunamas stačiakampis yra tik dalis esamo tarpo, todėl jo koordinatė ir ilgis randamas tokiu būdu:

- Jei figūra padėta kairiame tarpo krašte: tarpo padėtis = tarpo padėtis + stačiakampio ilgis;
- Jei figūra padėta dešiniame tarpo krašte: tarpo padėtis = tarpo padėtis;
- Tarpo ilgis randamas: tarpo padėtis = tarpo padėtis - stačiakampio ilgis.

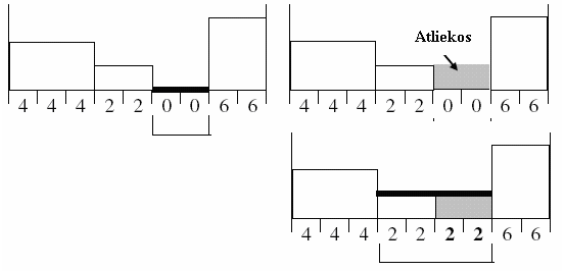


Tarpo padėtis = 3 Tarpo padėtis = 5

Ilgis = 4 Ilgis = 2

3.10 pav. Tarpo padėties nustatymo schema

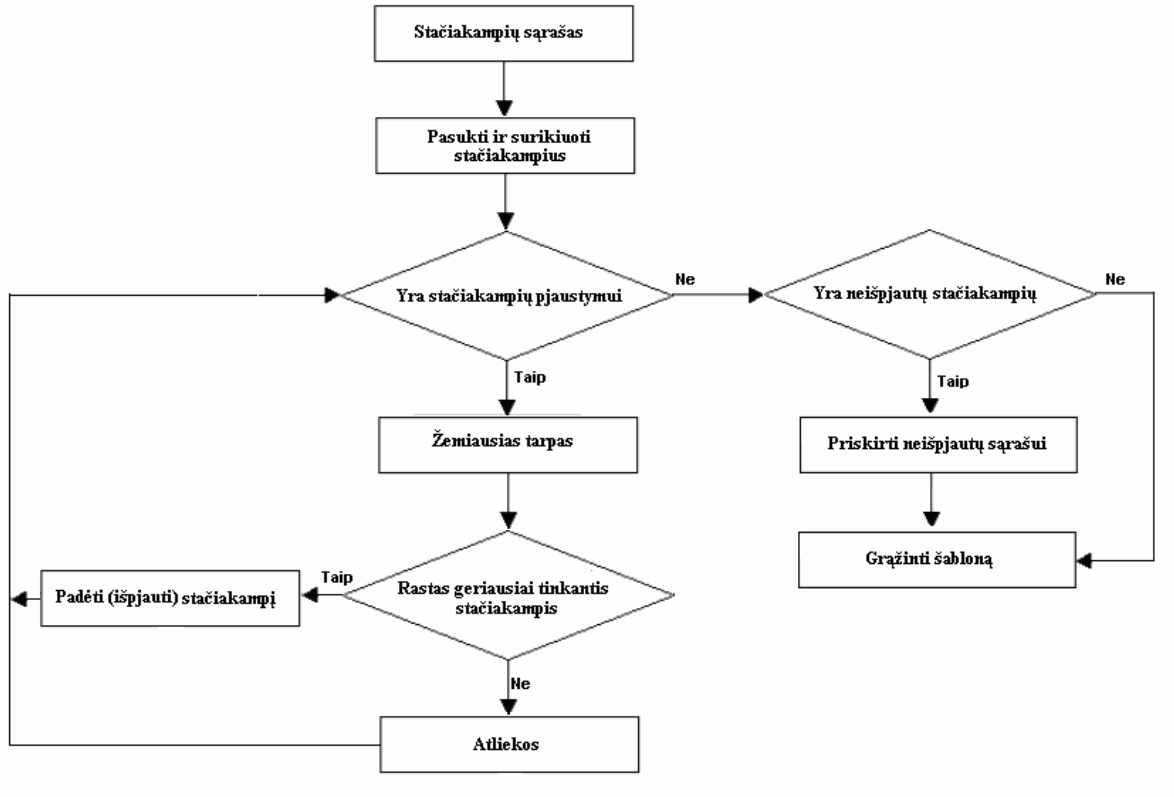
Jeigu nustatytam tarpui nerandame atitinkamo stačiakampio, tuomet esantis plotas interpretuojamas kaip atliekos, o tarpą atitinkantys lygių masyvo elementai padidinami iki žemiausio kaimyno reikšmės. Tarkime, kad rastas 2 vienetų ilgio tarpas, bet nerasta jam iš sąrašo detalės, kurios dimensijos būtų lygios arba mažesnės už 2. Tuomet tokia situaciją iliustruojame šiais paveikslėliais:



3.11 pav. Tarpo užpildymo schema

Kadangi tarpui nerandama figūra, tai tikrinami stačiakampiai sudarantys tą tarpą. Pirmojo lygis yra 2, o antrojo 6. Tuomet lygių masyvų elementai, atitinkantys tarpą, padidinami 2

vienetais, nes tai yra žemiausio kaimyno lygių reikšmė. Tuomet masyvas pertikrinamas, ieškant jo mažiausios reikšmės, nes po pakeitimo gali atsirasti lygio reikšmių mažesnių už 2, pjaustymo procesas tęsiamas įprastine tvarka. 3.12 paveikslėlyje pateikta bendra algoritmo schema.

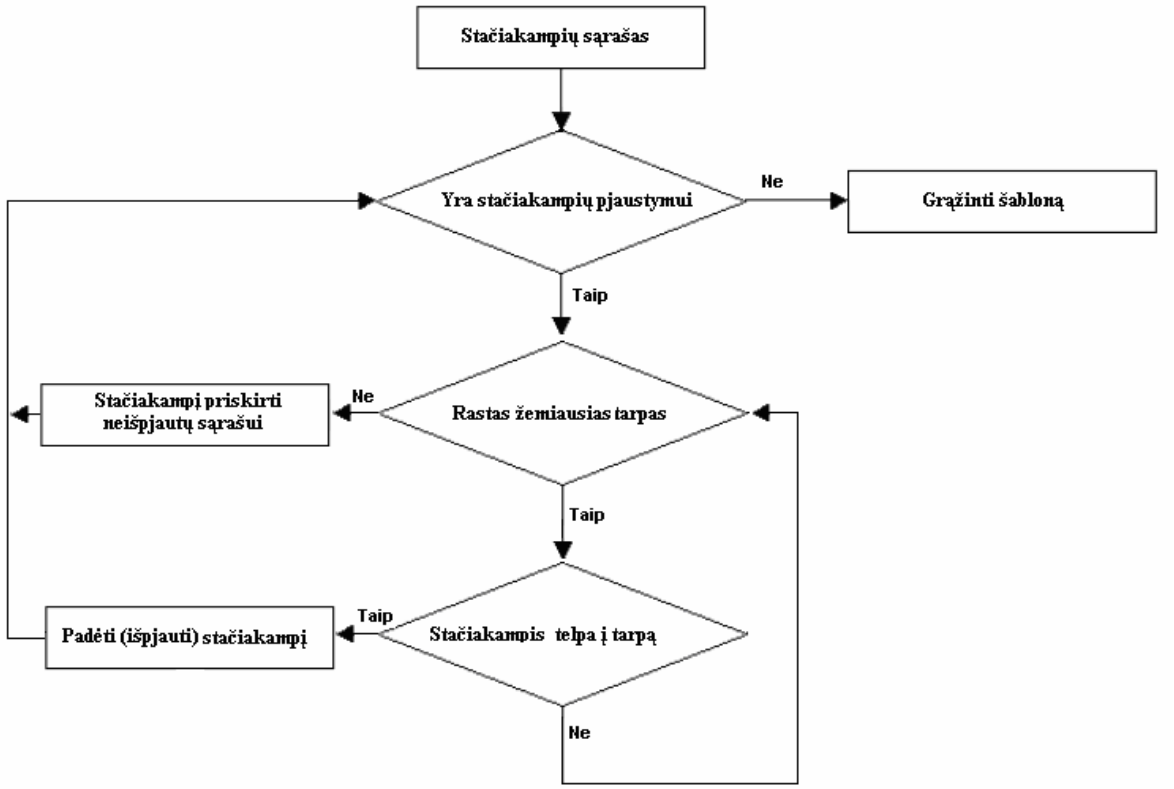


3.12 pav. „Geriausiai tinkamo“ algoritmo schema

Kadangi GT algoritmas pradinę stačiakampių seką surikiuoja ir tuomet organizuoja jų pjaustymą, rasdamas geriausiai tinkančią figūrą duotam tarpui, tai kiekvienam uždaviniui bus generuojamas vienareikšmiškas sprendimas. Tai reiškia, kad nuo stačiakampių sekos eiliškumo sprendimo rezultatas visiškai nepriklauso, o priklauso tik nuo pačių stačiakampių asortimento. Dėl šios priežasties šis metodas negali būti panaudotas kartu su genetiniu ir modeliuojamojo atkaitinimo algoritmais stačiakampių sekų interpretavimui. Vienintelis kombinuotas panaudojimo atvejis galimas, kai GT yra inkorporuojamas į bendrą hibridinio algoritmo schemą pradiniam sprendiniui konstruoti.

Tam galima būtų panaudoti GT metodą stačiakampių sekų interpretavimui buvo atliktos kelios modifikacijos. Pirmiausia atsisakyta figūrų sekos rikiavimo pagal ilgį ir plotį. Tačiau atlikus vien tik tokią modifikaciją, vis tiek bus seka interpretuojama vienareikšmiškai, nes algoritmas žemiausiam tarpui, visąlaik ieškos geriausiai atitinkančio stačiakampio, tai tik pailgins algoritmo veikimo laiką ir nieko naudingo daugiau neduos. Tuomet buvo atliktas dar vienas pakeitimas – atsisakyta geriausiai tinkančio stačiakampio paieškos duotam tarpui.

Taip naujai apibrėžta GT algoritmo modifikacija pagal savo veikimo pobūdį buvo pavadinta „Žemiausio Tarpo“ (ŽT) algoritmu. Šis metodas tenkino reikalavimą interpretuoti stačiakampių sekas nevienareikšmiškai atsižvelgiant į jų eiliškumą.



3.13 pav. „Žemiausio Tarpo“ algoritmo schema

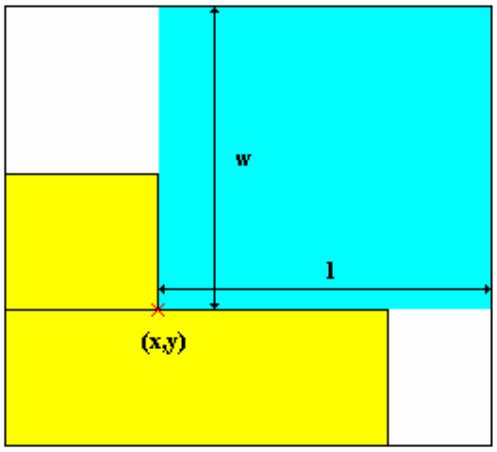
3.3.4. „ŽEMIAUSIO KAIRĖN UŽPILDYMO“ ALGORITMAS

Šis metodas taip pat naudojamas kartu su genetiniu ir modeliuojamojo atkaitinimo algoritmais, nes gali interpretuoti stačiakampių sekas priklausomai nuo jų eiliškumo.

Algoritmo vykdymo metu nagrinėjamas taškų, kur galima būtų įdėti stačiakampį sąrašas, pradedant nuo žemiausio ir kairiausio taško. Kiekvienas taškas aprašomas savo koordinatėmis (x,y), ilgiu *l* ir pločiu *w*, kurie apibrėžia laisvą erdvę, į kurią galima įdėti stačiakampį (3.13 pav.). Tikrinama ar įmanoma įdėti stačiakampį į tą erdvę, jei neįmanoma tuomet pereinama prie kito taško. Procesas tęsiamas tol, kol yra taškų (vietų) kur galima būtų patalpinti stačiakampį arba kol yra nepabandytų išpjauti stačiakampių. Stačiakampis laikomas neišpjautu, jeigu patikrinus visus taškus, nerandama jam patalpinimo vieta. Žinant lakšto ir išpjautų stačiakampių bendrą plotą, nesunkiai (algoritmo vykdymo pabaigoje) apskaičiuojamos atliekos.

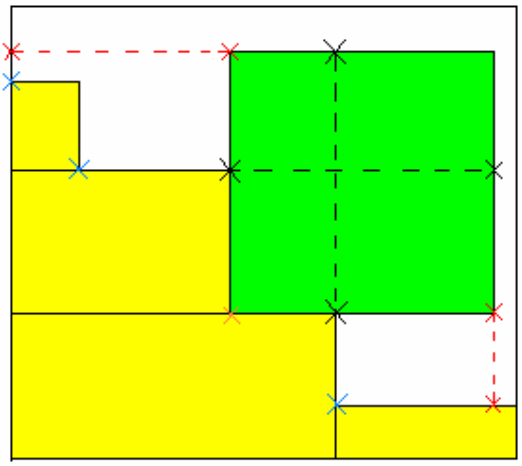
Kiekvieno stačiakampio įdėjimo (išpjovimo) metu vykdomas taškų sąrašo atnaujinimas. Pirmiausia randamas kairys viršutinis ir dešinys apatinis, tuomet jiems atitinkamai randami kairiausias ir žemiausias taškai (3.14 pav.). Šie taškai pažymėti raudona spalva, bus įtraukti į taškų sąrašą, jei prieš tai jame dar nebuvo. Žaliu fonu nuspalvintas stačiakampis , kuris bus

išpjautas įdėtas (išpjautas) einamuoju momentu. Stačiakampio atsiradimas įtakoja aplinkinius kaimyninius taškus, kurie pažymėti mėlyna spalva.



3.13 pav. Taško samprata

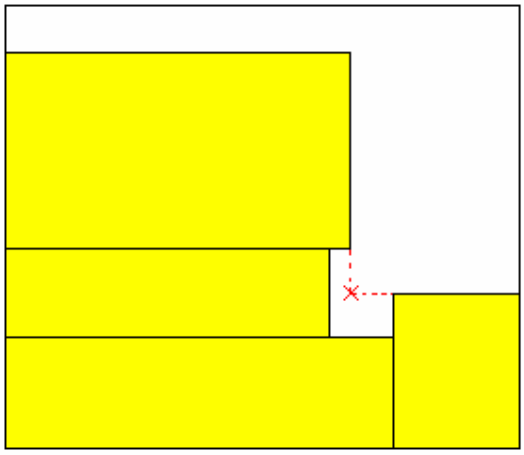
Tokio tipo taškai nagrinėjami tik iš kairės ir apatinės stačiakampio pusės ir yra atnaujinami jų atitinkamai, jei iš kairės tai ilgis *l*, jei iš apačios tai plotis *w*. Dar vieno tipo taškai turi būti (juoda spalva pažymėti ir punktyrais parodyta nauja vieta) atitinkamai perkelti per talpinamos figūros ilgį, jei taškas yra kairėje figūros kraštinėje (su sąlyga, kad dešinė kraštinė nesiliečia su jau išpjauta figūra ar lakšto kraštine) ir plotį, jei taškas randasi apatinėje stačiakampio kraštinėje (su sąlyga, kad viršutinė kraštinė nesiliečia su jau išpjauta figūra ar lakšto kraštine). Taškas esantis kairėje apatinėje stačiakampio viršūnėje (oranžinė spalva) yra eliminuojamas iš sąrašo.



3.14 pav.Taškų atnaujinimas

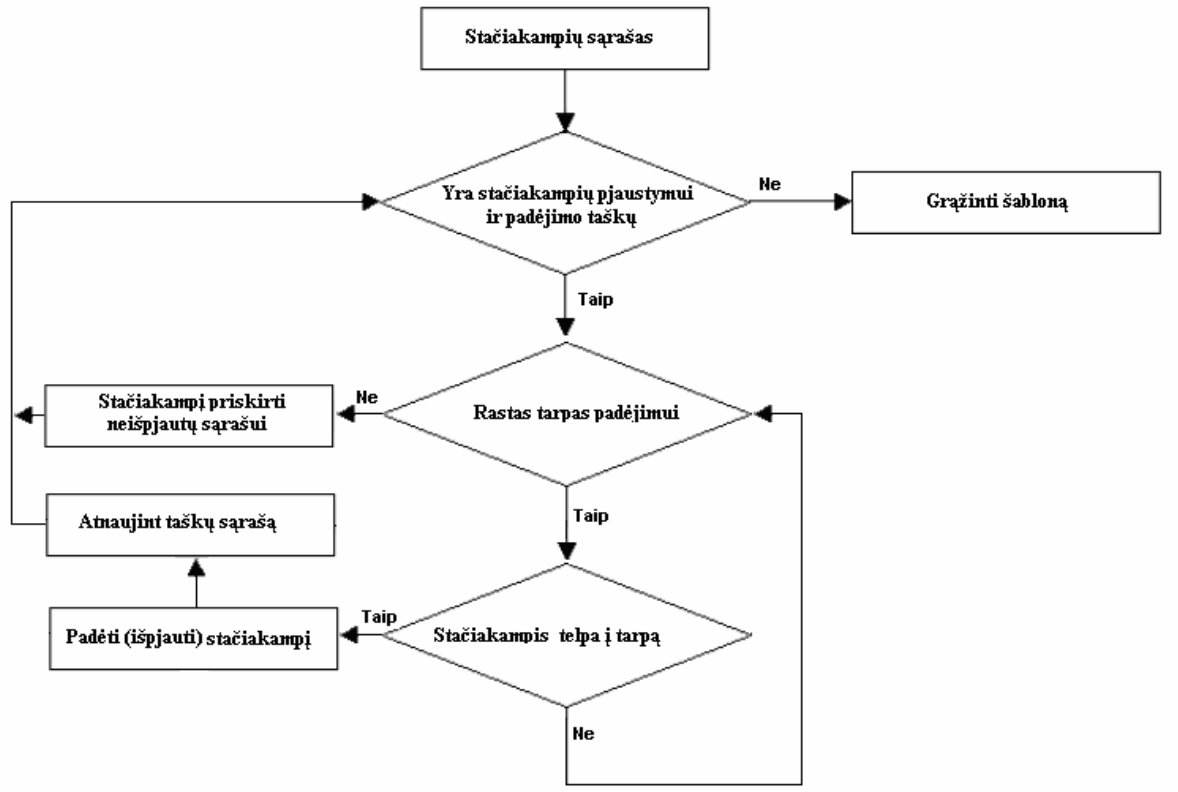
Taigi yra keturių tipų taškai, su kuriais atliekami skirtingi veiksmai. Tam kad palengvinti taškų paiešką, po kiekvienos įdėtos figūros, taškų sąrašas yra perrikiuojamas abiejų koordinačių didėjimo kryptimi.

Svarbu paminėti tai, kad šis algoritmo realizacijoj nenumatytas nagrinėjimas taip vadinamų *„kabančių“* taškų (3.15 pav.).



3.15 pav. „Kabantys“ taškai

Galiausiai pateiksime bendrą algoritmo schemą (3.16 pav.)



3.16 pav. ŽKU schema

3.4.TYRIMO REZULTATAI

3.4.1. EURISTINIŲ ALGORITŲ PALYGINIMAS

Tyrimo metu buvo nagrinėjami šie uždaviniai (3.1 lentelė). Šios užduotys buvo suformuluotos Hopper ir Turton [16], jų optimalūs sprendiniai žinomi „a priori“. Tai vienas iš plačiausiai naudojamų uždavinių rinkinių pjaustymo algoritmams testuoti.

Euristiniai algoritmai buvo patikrinti su visais uždaviniais (3.1 lentelė). Apskaičiuota procentinė atliekų ploto dalis:

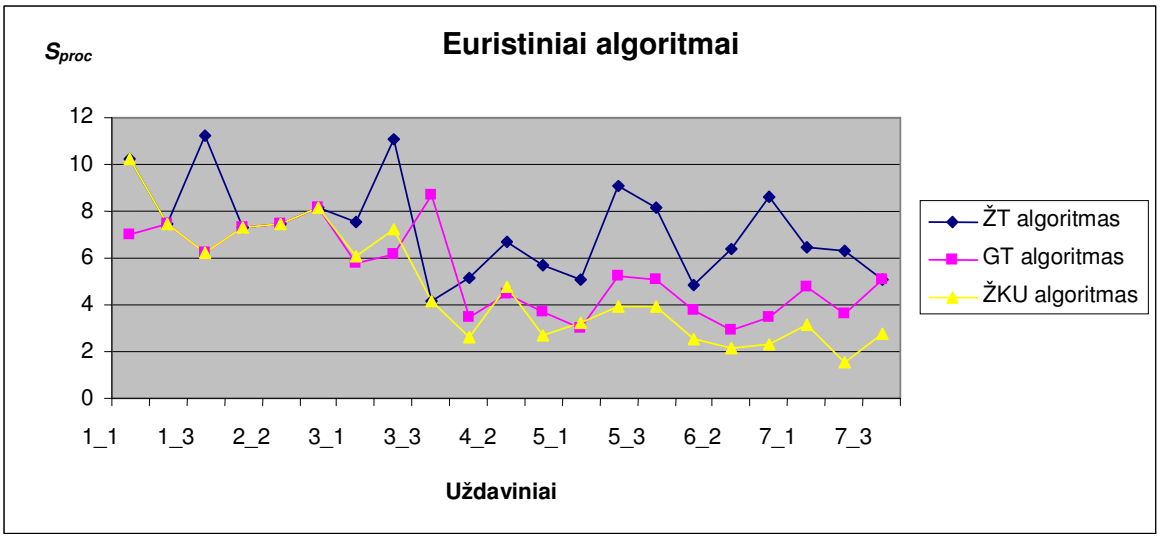
$$S_{proc} = \frac{S}{W \cdot L} \cdot 100\% \qquad (3.4.)$$

Čia S – gautas atliekų plotas (m^2), išpjovus medžiagos lakštą, čia W – lakšto plotis, L – lakšto ilgis. Gauti rezultatai (3.17 pav.)

3.1 lentelė

Uždaviniai

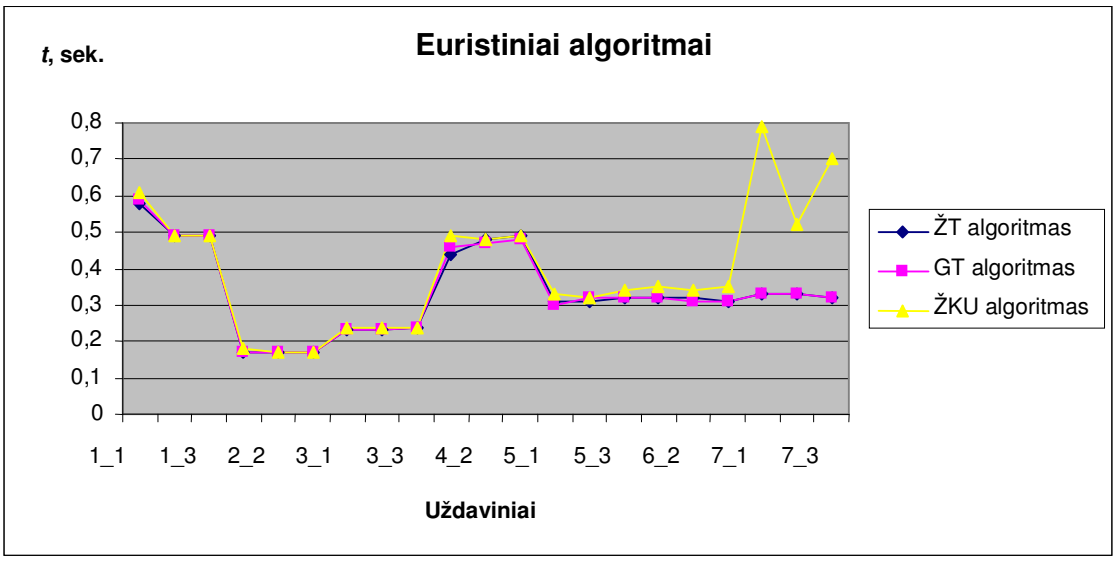
Uždavinys	Figūrų skaičius	Lakštų išmatavimai	
		Ilgis, m	Plotis, m
1_1	16	20	20
1_2	17	20	20
1_3	16	20	20
2_1	25	40	15
2_2	25	40	15
2_3	25	40	15
3_1	28	60	30
3_2	29	60	30
3_3	28	60	30
4_1	49	60	60
4_2	49	60	60
4_3	49	60	60
5_1	73	60	90
5_2	73	60	90
5_3	73	60	90
6_1	97	80	120
6_2	97	80	120
6_3	97	80	120
7_1	196	160	240
7_2	197	160	240
7_3	196	160	240



3.17 pav. Procentinė ploto atliekų dalis S_{proc}

Taip pat buvo stebimas algoritmo sprendimo laikas – *t*, išreikštas sekundėmis (3.18 pav.).

.



3.18 pav. Algoritmų sprendimo laikas *t*

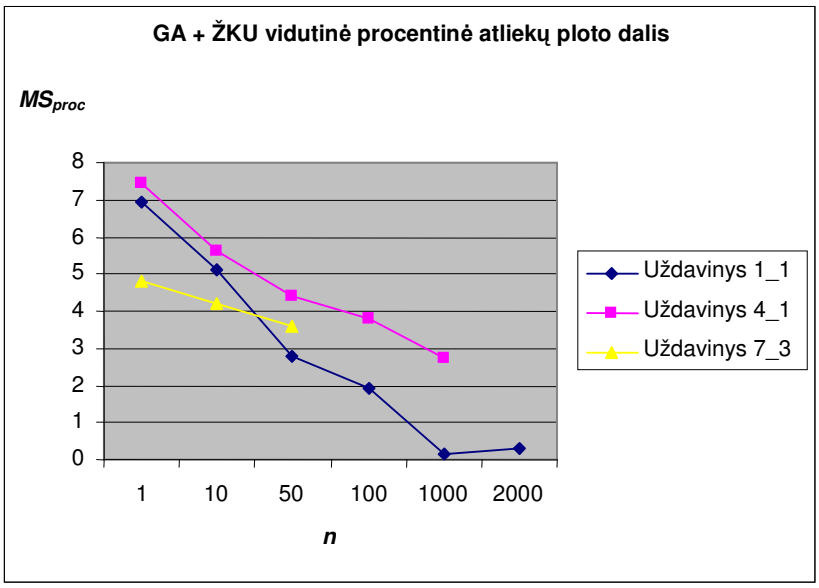
3.4.2. GENETINIO ALGORITMO PARAMETRŲ TYRIMAS

Pasirinkus tris uždavinius (3.1 lent.) buvo tiriama epochų skaičiaus *n* įtaka uždavinio vidutinei procentinei atliekų ploto daliai:

$$MS_{proc} = \frac{MS}{W \cdot L} \cdot 100\% \qquad (3.5.)$$

Čia $MS = \frac{\sum_{i=1}^{N} S_i}{N}$, kur N – stebėjimų skaičius, S_i – atliekų plotas (m^2), gautas stebėjimo momentu i, o W ir L atitinkamai lakšto plotis ir ilgis. Mūsų atveju N = 10. Sprendimo algoritmas – *GA + ŽKU* (genetinis hibridizuotas su ŽKU pjaustymo euristika, pradinė populiacija parenkama atsitiktinai).

Epochų skaičiaus tyrimas buvo vykdomas, kai kiti parametrai fiksuoti atitinkamai: populiacijos dydis - 10, kryžminimo tikimybė - 0.1, elitinės atrankos buferio dydis – 0.1.



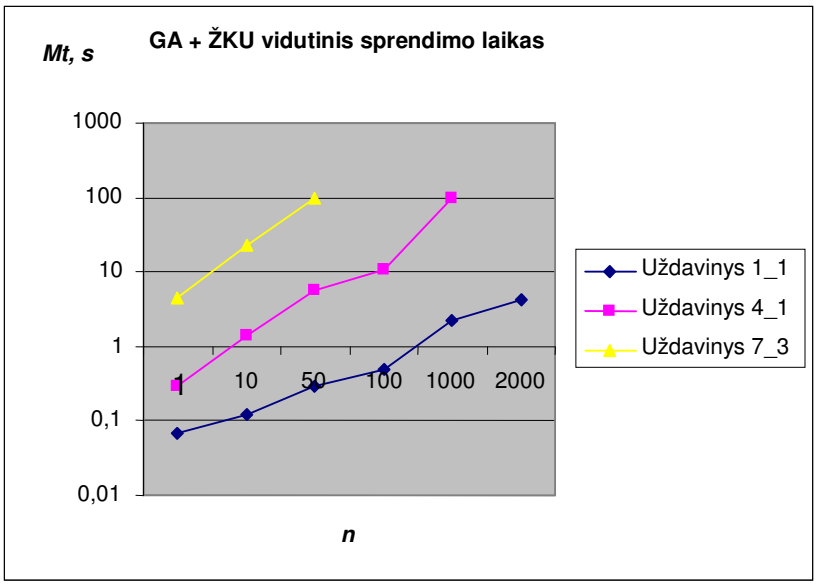
3.19 pav. GA + ŽKU MS_{proc} priklausomybė nuo *n*

Taip pat buvo stebimas algoritmo vidutinis sprendimo laikas – *Mt*, išreikštas sekundėmis.

$Mt = \frac{\sum_{i=1}^{N} t_i}{N}$, čia t_i algoritmo sprendimo laikas (sek.) stebėjimu i. Kadangi buvo gauti labai skirtingi vidutiniai sprendimo laikai, tai panaudota logaritminė skalė (grafiko ordinatė, 3.20 pav.).

Taip pat tirta sprendinio vidutinio procentinio atliekų ploto priklausomybė - MS_{proc} nuo populiacijos dydžio m (3.21 pav.) ir vidutinio sprendimo laiko Mt (3.22 pav.).

Fiksuoti parametrai pateikiami 3.2 lentelėje.

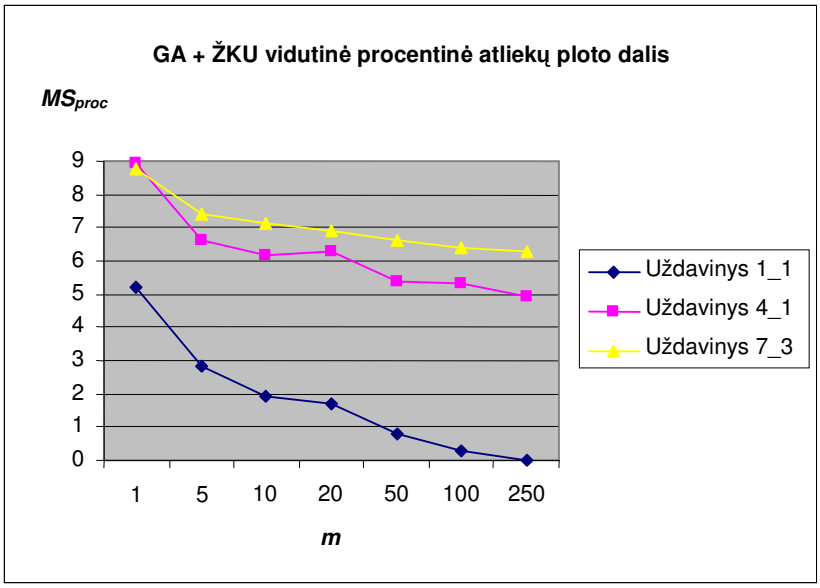


3.20 pav. GA + ŽKU *Mt* priklausomybė nuo *n*

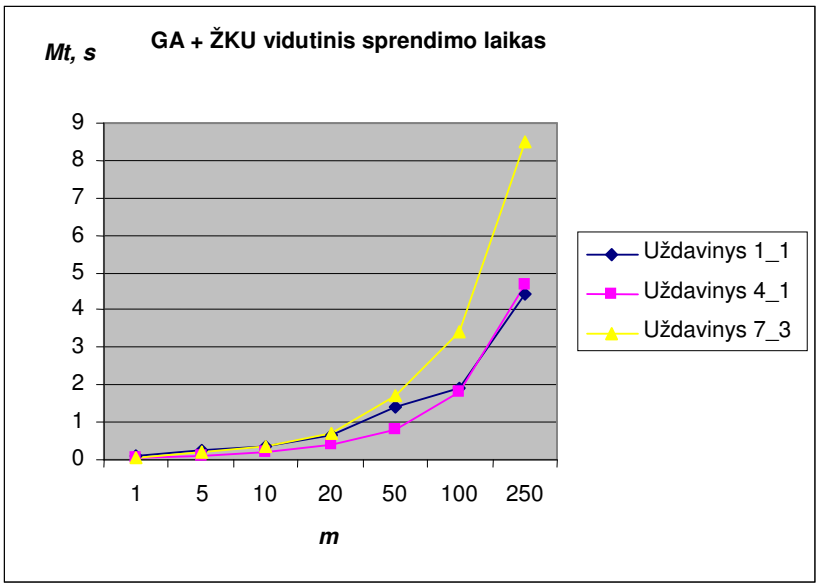
3.2 lentelė

Fiksuoti parametrai, populiacijos dydžio tyrimas

Uždavinys	Epochų skaičius	Mutacijos tikimybė	Elitinės atrankos tikimybė
1_1	100	0,1	0,1
4_1	50	0,1	0,1
7_3	10	0,1	0,1

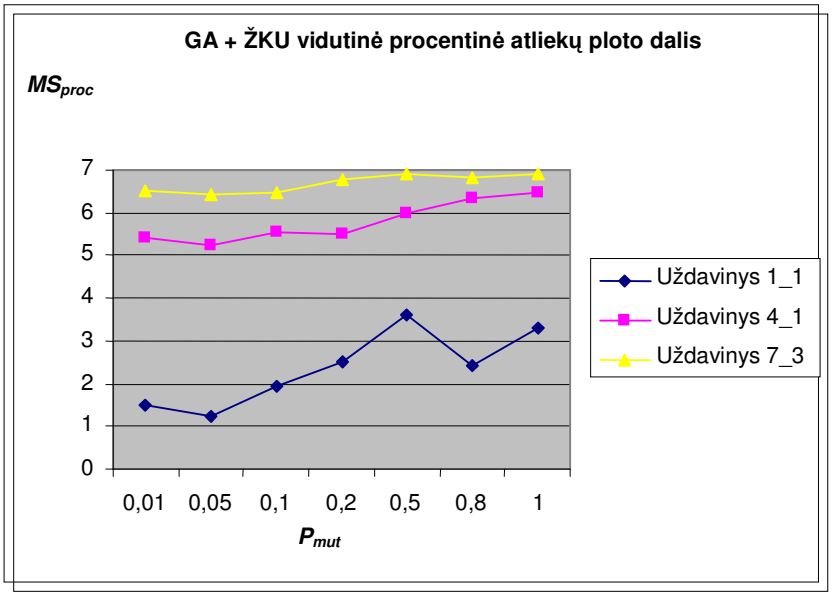


3.21 pav. GA + ŽKU MS_{proc} priklausomybė nuo *m*



3.22 pav. GA + ŽKU *Mt* priklausomybė nuo *m*

Galiausiai stebėta sprendinio vidutinio procentinio atliekų ploto priklausomybė - MS_{proc} nuo mutacijos tikimybės - P_{mut} (3.23 pav.) ir elitinės atrankos buferio dydžio - k (3.24 pav.).



3.23 pav. GA + ŽKU MS_{proc} priklausomybė nuo P_{mut}

3.4 lentelė

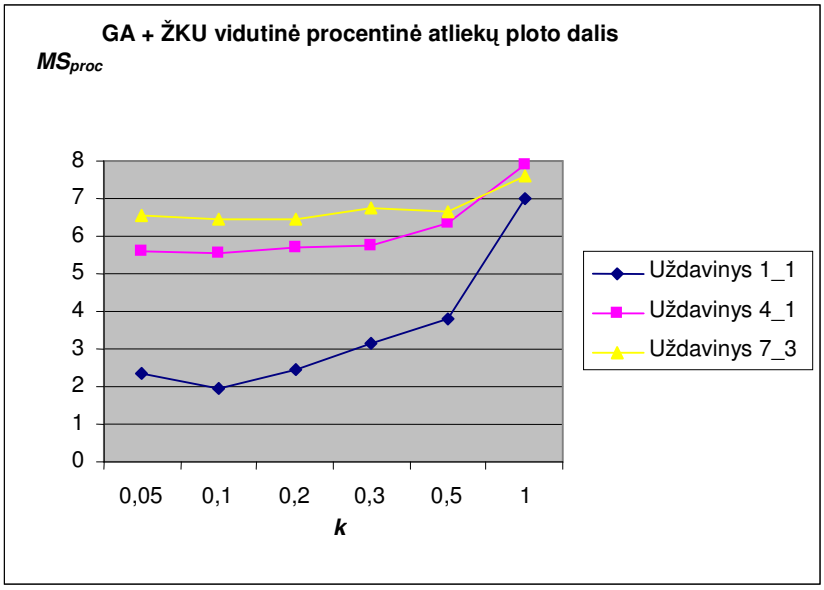
Fiksuoti parametrai, mutacijos tikimybės tyrimas

Uždavinys	Epochų skaičius	Populiacijos dydis	Elitinės atrankos tikimybė
1_1	100	50	0,1
4_1	50	50	0,1
7_3	10	50	0,1

3.5 lentelė

Fiksuoti parametrai, elitinės atrankos buferio dydžio tyrimas

Uždavinys	Epochų skaičius	Populiacijos dydis	Mutacijos tikimybė
1_1	100	50	0,1
4_1	50	50	0,1
7_3	10	50	0,1



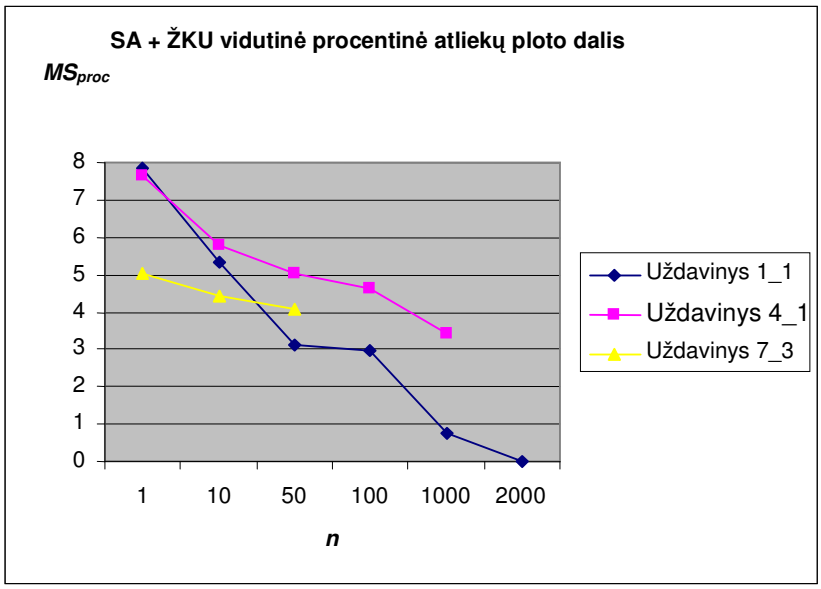
3.24 pav. GA + ŽKU MS_{proc} priklausomybė nuo k

3.4.3. MODELIUOJAMOJO ATKAITINIMO PARAMETRŲ TYRIMAS

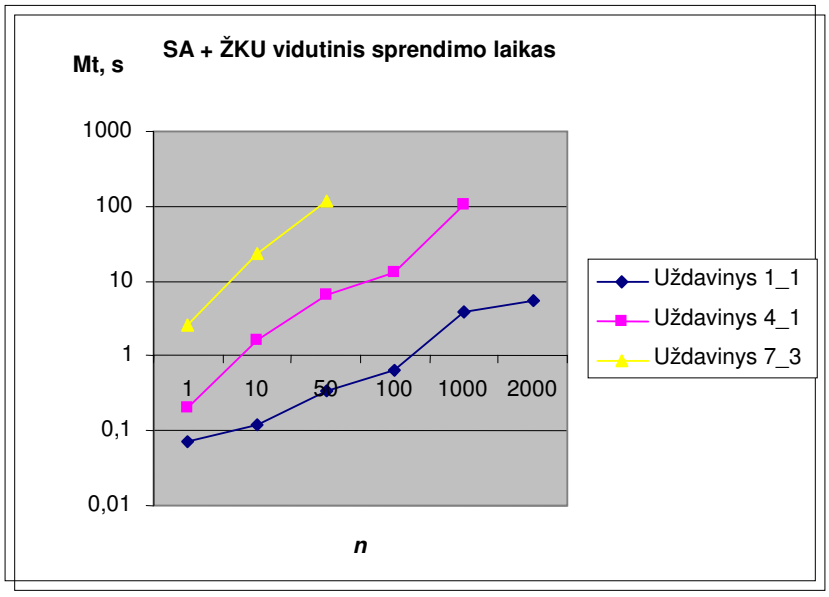
Analogiškai esant toms pačioms eksperimento sąlygoms buvo tirtas MA + ŽKU (modeliuojamojo atkaitinimo algoritmas hibridizuotas su žemiausio kairėn užpildymo euristika, kai pradinis sprendinys parenkamas atsitiktinai).

Buvo stebima temperatūros išreikštos tiesiogiai iteracijų skaičiumi n ir vidinių iteracijų skaičiaus m įtaka uždavinio vidutinei procentinei atliekų ploto daliai - MS_{proc} (3.25 pav.) ir (3.27 pav.) bei sprendimo vidutinei trukmei – Mt (3.26 pav.) ir (3.28 pav.).

Fiksuotas vidinių iteracijų skaičius lygus 10.



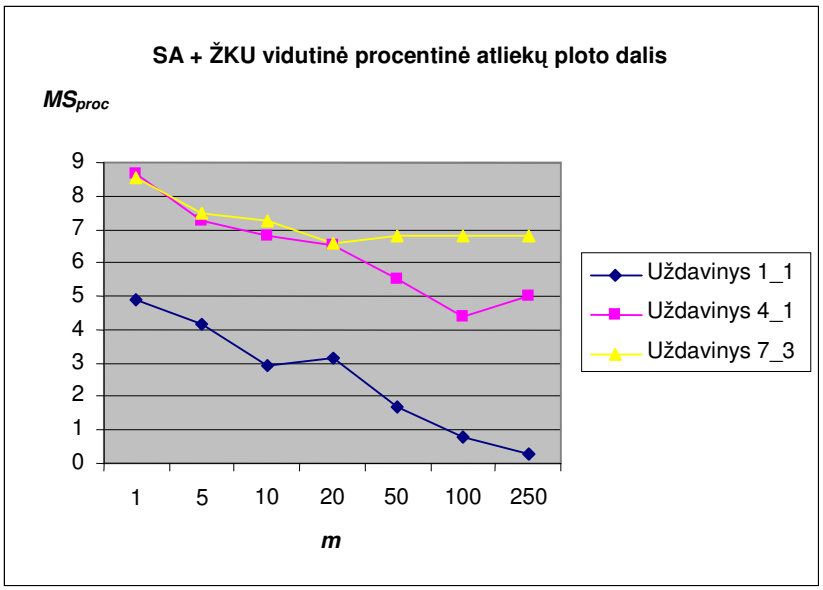
3.25 pav. MA + ŽKU MS_{proc} priklausomybė nuo *n*



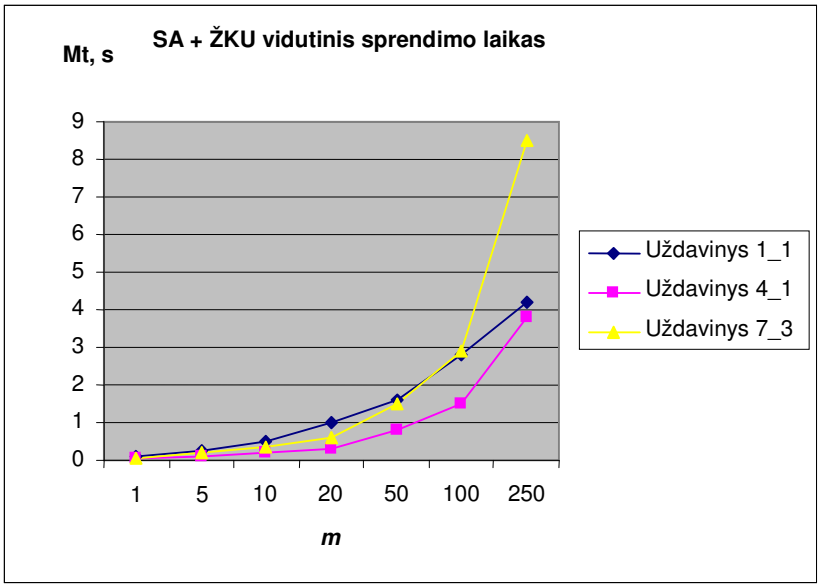
3.26 pav. MA + ŽKU *Mt* priklausomybė nuo *n*

Kadangi buvo gauti labai skirtingi vidutiniai sprendimo laikai, tai panaudota logaritminė skalė (grafiko ordinatė, 3.26 pav.).

Vidinių iteracijų skaičiaus tyrimui panaudotas fiksuotas temperatūros parametras (2.4.3.1 lent.).



3.27 pav. MA + ŽKU MS_{proc} priklausomybė nuo *m*



3.28 pav. MA + ŽKU *Mt* priklausomybė nuo *m*

3.6 lentelė

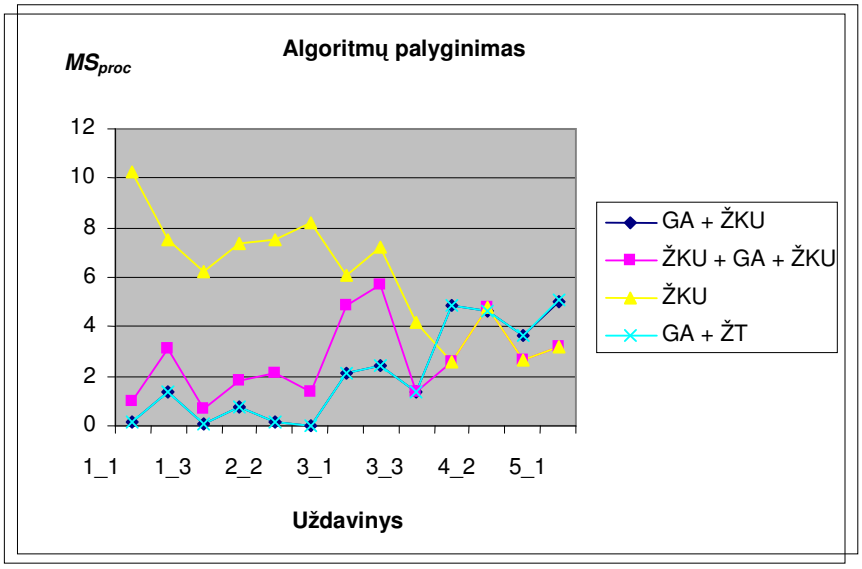
Fiksuoti parametrai, temperatūros dydžio tyrimas

Uždavinys	Temperatūra
1_1	100
4_1	50
7_3	10

3.4.4. BENDRAS ALGORITMŲ PALYGINIMAS

Palyginimui pasirinkti 13 uždavinių (žiūrėti 3.1. ir 3.8 lent.) ir 3 algoritmai: ŽKU („Žemiausio kairėn užpildymo“, kai pradinis stačiakampių sąrašas surikiuojamas ilgio ir pločio mažėjimo kryptimi), anksčiau minėtas GA + ŽKU ir ŽKU + GA + ŽKU (pradinė populiacija generuojama kombinuotai su ŽKU, t.y. vienas populiacijos individas sukonstruojamas ŽKU pagalba, kiti pasirenkami atsitiktinai). Apskaičiuoti MS, MS_{proc} (žiūrėti 3.4. formulę), S ir S_{proc} (žiūrėti 3.5. formulę), stebėjimų skaičius $N = 10$.

Gavome MS_{proc} priklausomybę nuo sprendžiamo uždavinio (3.29 pav.).



3.29 pav. Algoritmų palyginimas

Kiekvienai uždavinių grupei naudoti skirtingi parametrai (3.7 lent.).

3.7 lentelė

Algoritmų parametrai

Uždaviny s	Figūrų skaičiu s	Epochų skaičius	Populiacijos dydis	Mutacijos tikimybė	Elitinės atrankos tikimybė
1_1	16	1000	50	0,05	0,1
1_2	17	1000	50	0,05	0,1
1_3	16	1000	50	0,05	0,1
2_1	25	700	50	0,05	0,1
2_2	25	700	50	0,05	0,1
2_3	25	700	50	0,05	0,1
3_1	28	500	50	0,05	0,1
3_2	29	500	50	0,05	0,1
3_3	28	500	50	0,05	0,1
4_1	49	100	50	0,05	0,1
4_2	49	100	50	0,05	0,1
4_3	49	100	50	0,05	0,1
5_1	73	70	50	0,05	0,1

3.8 lentelė

Algoritmų sprendimo rezultatai

	GA + ŽKU		ŽKU+ GA + ŽKU		ŽKU		GA+ŽT	
Uždavinys	MS,m2	MS,proc.	MS,m2	MS,proc.	MS,m2	S,proc.	MS,m2	MS,proc.
1_1	0,6	0,15	4	1	4	10,25	0,6	0,15
1_2	5,4	1,35	12,4	3,1	12,4	7,5	5,4	1,35
1_3	0,4	0,1	2,6	0,65	2,6	6,25	0,4	0,1
2_1	7,6	0,76	18,6	1,86	18,6	7,33	7,6	0,76
2_2	1,3	0,13	20,9	2,09	20,9	7,5	1,3	0,13
2_3	0	0	14	1,4	14	8,17	0	0
3_1	35,3	2,1	82,2	4,89	82,2	6,06	35,3	2,1
3_2	41,8	2,4	99,3	5,71	99,3	7,22	41,8	2,4
3_3	23,4	1,39	22,5	1,34	22,5	4,17	23,4	1,39
4_1	174,2	4,84	93	2,58	93	2,58	174,2	4,84
4_2	166,7	4,63	171	4,75	171	4,75	166,7	4,63
4_3	132,6	3,68	97	2,69	97	2,69	132,6	3,68
5_1	270,7	5,01	173	3,2	173	3,2	273,8	5,07

4. PROGRAMINĖ REALIZACIJA IR INSTRUKCIJA VARTOTOJUI

Programa parašyta C++ kalba, reikalinga operacinė sistema Windows 98/ME/2000/XP. Jeigu bus sprendžiamos didelės apimties užduotys, tai pageidautinas pakankamai galingas kompiuteris.

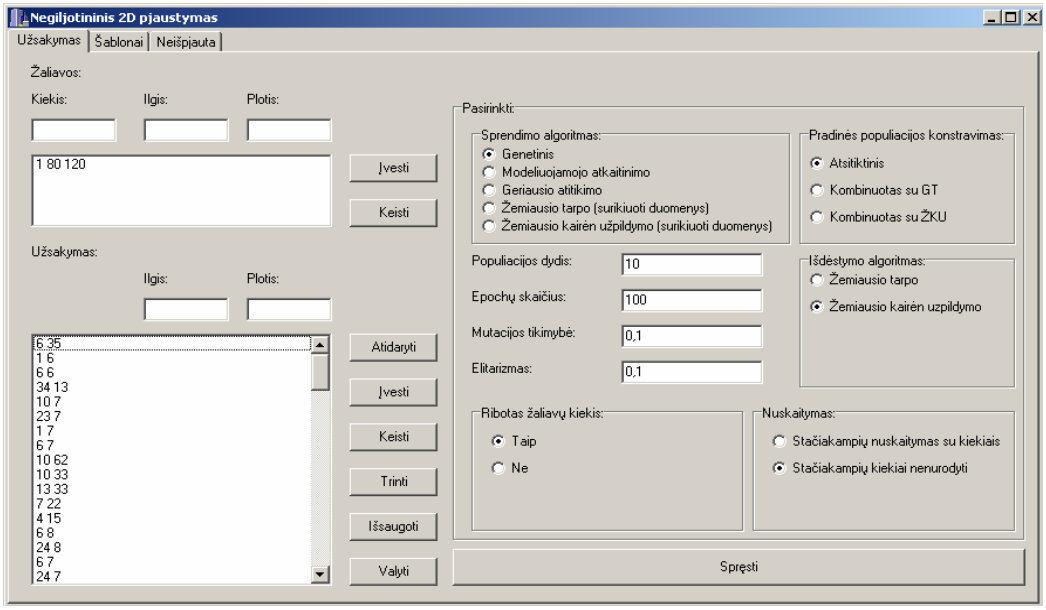
Programos vykdymas prasideda tiesioginiu duomenų įvedimu klaviatūra arba nuskaitymu iš failo, tai galima padaryti paspaudus mygtuką “Įvesti” (žiūrėti pav. 4.1 pav.) , jeigu vykdomas nuskaitymas iš failo, tai pasirinkti failą – spausti mygtuką “Atidaryti”. Mygtukai “Keisti”, “Trinti” skirti darbui su teksto koregavimu. “Išsaugoti” mygtuko paskirtis išsaugoti duomenų failą pasirinktoje kompiuterio vietoje.

Dešinėje lango pusėje yra parametrų ir algoritmų pasirinkimo laukas. “Sprendimo algoritmas” įgalina vartotoją pasirinkti sprendimo algoritmą galimi 5 algoritmų pasirinkimo variantai, atitinkamai skiltyje “Išdėstymo algoritmas” - du išdėstymo algoritmai. Viso bendrose kombinacijose galima įvykdyti 7 skirtingus algoritmus iš jų 4 su 3 skirtingais pradinių sąlygų generavimo būdais.

Sprendžiami du skirtingi uždavinių variantai:

- Žaliavų kiekis ribotas;
- Žaliavų kiekis pakankamas užsakymui įvykdyti.

Be to numatyti du nuskaitymo būdai iš failo formatu: “kiekis ilgis plotis” arba “ilgis plotis”, antruoju variantu bus laikoma, kad kiekis lygus vienetui.



4.1 pav. Pirmasis vartotojo sąsajos langas

Parametrų sąrašas pasikeičia automatiškai pasirinkus atitinkamą algoritmą. Genetinio algoritmo parametrai:

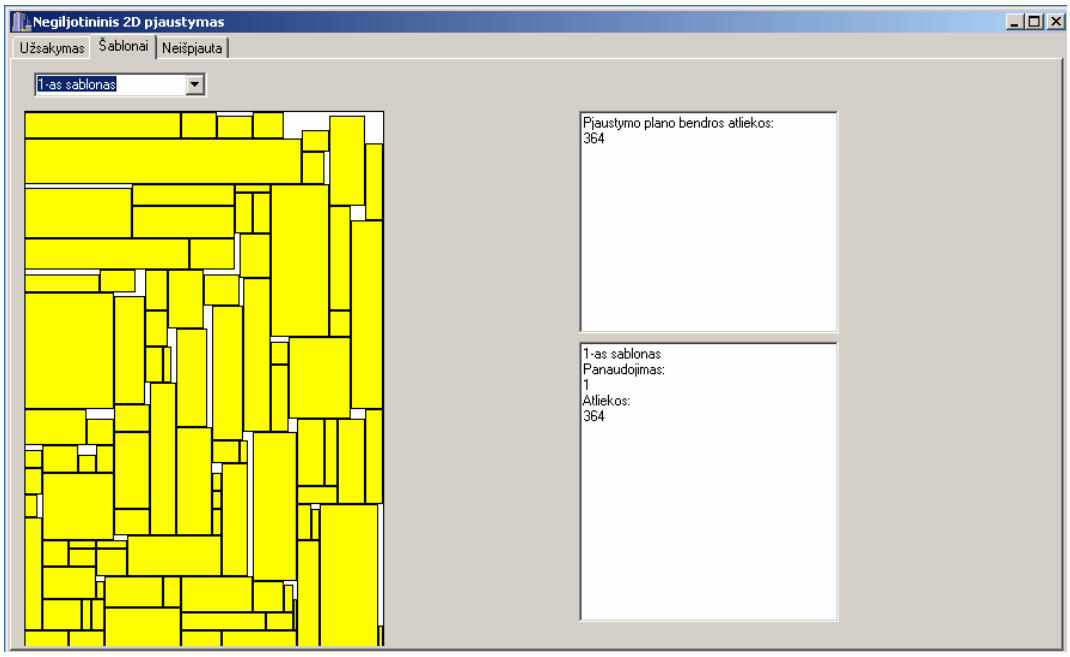
- Epochų skaičius;
- Populiacijos dydis;
- Mutacijos tikimybė;
- Elitinės atrankos buferio dydis;

Modeliuojamojo atkaitinimo (abiejuose variantuose nurodomi iteracijų skaičiai):

- Temperatūra;
- Energijos lygmuo.

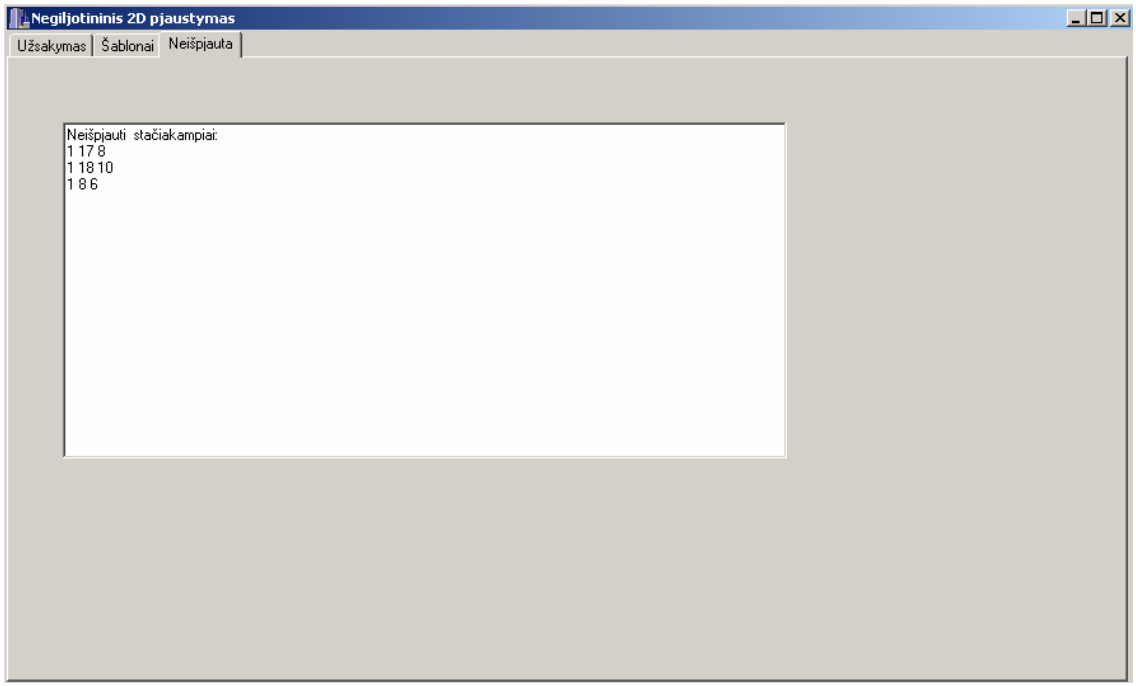
Programa vykdoma paspaudus mygtuką “Spręsti”. Gaunami sprendimo rezultatai (4.2 pav.).

Kairėje pusėje matome pjaustymo šablono grafinį vaizdą (4.2 pav.), galima pasirinkti kitą šabloną, jei toks egzistuoja, o dešinėje informaciją apie pjaustymo planą ir bendras atliekas bei duomenis apie pasirinktą šabloną ir jo atliekas, panaudojimo skaičių. Kadangi šiuo atveju pjaustymo planas sudarytas iš vieno šablono, tai informacija sutampa.



4.2 pav. Antrasis vartotojo sąsajos langas

Jeigu nepavyksta įvykdyti užsakymo, tai tie stačiakampiai ir jų kiekiai pateikiami trečiame vartotojo sąsajos lange (4.3 pav.).



4.3 pav. Trečiasis vartotojo sąsajos langas

5. DISKUSIJA

5.1. ALGORITMŲ PALYGINIMO TYRIMO APŽVALGA

Nagrinėjant euristinius algoritmus (2.4.1. skyrelis) paaiškėjo, kad geriausi rezultatai procentinės atliekų ploto dalies prasme gauti pritaikius ŽKU algoritmą. Verta paminėti tai, kad GT algoritmas pateikė panašaus tikslumo sprendinius, kai stačiakampių užsakymo dydis neviršijo 30 figūrų (3.17 pav.). Vėliau išryškėja ŽKU algoritmo pranašumas. Stačiakampių seka buvo surikiuojama figūrų išmatavimo mažėjimo tvarka, t.y. kad didesnių išmatavimų stačiakampiai būtų išdėstyti pirmieji. Dėl šios priežasties ŽKU algoritmas turi pranašumą prieš nagrinėtus analogus, nes jo vykdymo metu yra saugomas visų įmanomo stačiakampių padėjimo vietų sąrašas (1.10 pav.). Todėl, pirma išdėsčius didesnes figūras, atsiranda tarpų įdėti mažesnes, kurių algoritmas "nepamiršta", nes manipuliuoja minėtu sąrašu. Šios savybės neturi GT ir ŽT algoritmai, todėl jie veikia greičiau ypač, kai uždavinys sudėtingėja (3.17 pav.). ŽT metodo vykdymo metu gauti blogiausi rezultatai, nes šis algoritmas yra GT modifikacija, pritaikyta dirbti kombinuotai su genetiniais ir modeliuojamojo atkaitinimo metodais.

Apžvelgiant į euristinių algoritmų rezultatus paaiškėja labai įdomi pastarųjų savybė – sudėtingėjant uždaviniui pateikiamas vis geresnis sprendinys. Iš tikrųjų, kai užsakymo dydis buvo mažiau nei 30 figūrų, pasiekta atliekų procentinė ploto dalis 7 – 10 proc., tuo tarpu, kai užsakymas padidėjo iki 70 figūrų ir daugiau, gauta atliekų procentinė ploto dalis 2 – 5 proc. Taip pat sprendimo laikas buvo labai trumpas ir neviršijo 0,7 sekundės nagrinėtų užduočių atvejais. Vis dėl to sprendžiant labai didelės apimties užduotis (pvz. Į vieną lakštą telpa 1000 ir daugiau stačiakampių) vertėtų naudoti tik GT algoritmą, nes tarpui randanti geriausią stačiakampį procedūra išeikvoja mažiau kompiuterinio laiko, nei procedūra randanti stačiakampiui tinkamą tarpą (ŽT, ŽKU). Dėl šios priežasties ŽK ir ŽKU algoritmų sprendimo laikas gali labai padidėti, esant didelės apimties uždaviniams.

Tuo tarpu apjungus ŽT ir ŽKU algoritmus su genetiniu ir modeliuojamojo atkaitinimo metodais (GA + ŽKU, MA + ŽKU, GA + ŽT, MA + ŽT) gauti rezultatai byloja visiškai ką kitą. Sprendžiant mažos apimties uždavinius, šie algoritmai pademonstravo daug geresnius rezultatus vidutinės procentinės atliekų ploto dalies prasme, nei pavieniai euristiniai (3.29 pav.) per neilgą sprendimo laiką. Tuo tarpu, kai uždavinio apimtis viršijo 50 stačiakampių, geresni rezultatai jau gauti sprendžiant euristiniais metodais (tarkim ŽKU).

Buvo manoma, kad parinkus pradinės sąlygas hibridiniams GA + ŽKU arba MA + ŽKU algoritmams, panaudojant vieną iš euristinių metodų (ŽKU arba GT) pagerės sprendinio kokybė. Tačiau eksperimentai paneigė šią prielaidą (3.29 pav., 3.8 lent.). Pradinė populiacija buvo konstruojama tokiais būdais: vienas sprendinys sugeneruojamas ŽKU metodu, o kiti atsitiktinai,

sutrumpintai pažymėta ŽKU + GA + ŽKU; arba atsitiktinai GA + ŽKU. Lyginant rezultatus, esant mažos apimties uždaviniams, lyderiavo GA + ŽKU, o kai užsakymo dydis pasiekė 50 figūrų ribą (5_1 uždavinys) ŽKU + GA + ŽKU sprendinys sutapo su ŽKU, t.y. nepavyko pagerinti pradinio sprendinio. Atrodo, kad 50 stačiakampių yra ta riba skirianti euristinių ir metaeuristinių algoritmų panaudojimo galimybes (čia kalbama apie 50 stačiakampių telpančių į vieną lakštą).

Dar vienas įdomus pastebėjimas (3.29 pav. ir 3.8 lent.) GA + ŽT hibridinis algoritmas parodė tuos pačius rezultatus, kaip ir GA + ŽKU (išskyrus uždavinį 5_1), nors pavieniui ŽT ir ŽKU metodai veikė labai skirtingai (3.17 pav.). Tai paaiškinama panašia ŽT ir ŽKU veikimo logika : abu metodai ieško žemiausio ir kairiausio laisvo tarpo figūrai patalpinti. Nepaisant to, kad ŽT algoritmas turi trūkumų lyginant su ŽKU (nėra laisvų padėjimo vietų sąrašo), tie trūkumai yra kompensuojami genetinio algoritmo veikimu. Tai patvirtino ir sprendimo laikas, nors ŽT pavieniui veikia greičiau nei ŽKU (3.18 pav.), GA+ ŽT sprendimo laikas buvo ilgesnis nei GA + ŽKU. Tai paaiškinama greitesniu ŽKU konvergavimu, gebėjimu geriau sukonstruoti sprendinį iš pateiktos tos pačios stačiakampių sekos.

5.2.ALGORITMŲ PARAMETRŲ TYRIMO APŽVALGA

Tiriant genetinio algoritmo epochų skaičiaus ir modeliuojamojo atkaitinimo temperatūros įtaką sprendinio kokybei (3.19. ir 3.25 pav.), paaiškėjo, kad GA + ŽKU parodė truputį geresnius rezultatus nei MA + ŽKU vidutinės procentinės atliekų ploto dalies prasme. Tai paaiškinama tuom, kad MA algoritmas naudoja tik mutaciją sprendinio paieškai organizuoti, o GA dar papildomai ir kryžminimo operatorių, tokiu būdu suteikiamos GA metodui geresnės sprendinių perrinkimo galimybės.

Bendrai, tiek epochų skaičius GA, tiek temperatūra MA algoritmams turėjo didelę įtaką, kai užsakymo skaičius nėra didelis, o didesniems užsakymams poveikio reikšmė mažėjo ir labai pailgėjo sprendimo laikas (2.21 ir 2.27 pav.), todėl tyrimo su didesnės apimties užduotimis buvo atsisakyta, nes tikslas operuoti „realiu“ užduočių išsprendimo laiku. Nagrinėjant tris užduotis pastebėta, kad sprendimo laikas buvo pakankamai neilgas ir rezultatai geri, kai nustatėme uždaviniui 7_3 iteracijų skaičių 10, 4_1 uždaviniui – 50 ir 1_1 uždaviniui – 1000.

Nagrinėjant GA + ŽKU populiacijos dydį ir SA + ŽKU energijos lygmens lygį (3.21. ir 3.27 pav.) paaiškėjo, kad šis parametras ženkliai veikia sprendinio kokybę ir ne taip stipriai įtakoja sprendimo laiką kaip anksčiau tirtas parametras - nustatytas dydis 50.

Tiriant GA mutacijos tikimybę (3.23 pav.) paaiškėjo, kad geriausi rezultatai gauti, kai reikšmė pasirenkama 0,05. Tai paaiškinama tuom, kad genetiniam algoritmui mutacija reikalinga

tik tam, kad suteiktų įvairumo konstruojamiems sprendiniams. Dažnas jos panaudojimas įneštų per daug chaoso sprendimo procese.

Pastebėta, kad geriausi rezultatai gaunami, kai GA elitinės atrankos buferio dydis yra tarp 10 – 20 proc. (3.24 pav.). Iš tikrųjų, jei per didelė populiacijos dalis perkeliama į naująją be pakeitimų, tuomet lėtėja sprendinių atsinaujinimas.

Apibendrinant diskusijos skyrelį galima suformuluoti pjaustymo uždavinių sprendimo metodiką:

- Nedidelės apimties uždaviniams, kai į lakštą telpa mažiau nei 50 stačiakampių, verta naudoti hibridinius metaeuristinius algoritmus (GA + ŽKU, MA + ŽKU ir t.t.);
- Euristiniai algoritmai (ŽT, ŽKU, GT) naudojami didesnės apimties užduotims spręsti, kai daugiau nei 50 stačiakampių telpa į lakštą;
- Kai susiduriama su labai didelės apimties užduotimi (1000, 10000 ir daugiau stačiakampių telpančių į vieną lakštą) patartina naudoti euristinį GT algoritmą;
- Metaeuristinių algoritmų rekomenduojami parinkti parametrai atsižvelgiant į sprendžiamos užduoties dydį pateikti (3.7 lent.) (GA algoritmo epochų skaičius ir populiacijos dydis atitinka MA algoritmo temperatūrą ir energijos lygmenį).

IŠVADOS

1. Atlikus euristinių algoritmų analizę paaiškėjo, kad geriausi rezultatai pastarųjų vykdymo metu gaunami, sprendžiant didesnės (kai pjaustymo šabloną sudaro daugiau nei 50 figūrų) apimties užduotis.

2. Metaeuristiniai hibridiniai algoritmai labiau tinkami naudoti mažesnės apimties pjaustymo užduotims (kai pjaustymo šabloną sudaro mažiau nei 50 figūrų) optimizuoti.

3. Suformuluota pjaustymo uždavinių sprendimo metodika.

4. Metaeuristinių hibridinių algoritmų sprendimo laiko trukmę ir kokybę labai įtakoja parametrų parinkimas.
5. Sukurta originali „Geriausiai tinkamo“ algoritmo modifikacija pritaikyta kombinuotai dirbti su metaeuristiniais (genetiniu ir modeliuojamojo atkaitinimo) metodais.

6. Apjungiant skirtingų algoritmų veikimą, t.y. juos hibridizuojant, paaiškėjo, kad tam tikro algoritmo trūkumai gali būti kompensuoti kito algoritmo veikimu.

REKOMENDACIJOS

Darbe išnagrinėta nemaža euristinių ir metaeuristinių metodų įvairovė, nepaisant to egzistuoja dar viena tiksliųjų metodų klasė ir nemažai kitų algoritmų (tokių kaip metaeuristiniai tabu paieška, skruzdžių kolonijos, GRASP), kurie taip pat sėkmingai taikomi pjaustymo uždaviniams spręsti.

Nagrinėjamos pjaustomos figūros buvo stačiakampiai, nes taip supaprastėja tam tikri užduoties susijusios su figūrų išdėstymu aspektai, o optimizavimo logika išlieka labai panaši.

Užduotis buvo suformuluota dviejų matavimų erdvėje, tačiau sprendžiami uždaviniai ir trimatės erdvės atveju.

Dėl šių paminėtų priežasčių, lieka nemažai erdvės naujiems tyrimams, susijusiems su naujų algoritmų realizavimu, sudėtingesnių figūrų pjaustymu ir užduoties nagrinėjimu trimatėje erdvėje.

PADĖKOS

Dėkoju magistro darbo vadovui doc. dr. Narimantui Listopadskiui už pagalbą ir patarimus. Taip pat dėkoju Sauliui Lazaravičiui ir Juliui Simonavičiui už naudingas diskusijas kombinatorinio optimizavimo temomis.

LITERATŪRA

1. H. Dyckhoff, A typology of cutting and packing problems, European Journal of Operational Research 44 (1990) 145-159.
2. G. Wascher, H. Haubner, H. Schumann, An Improved Typology of Cutting and Packing Problems, Working Paper No. 24, Last Revision: 2006-01-16, Faculty of Economics and Management Magdeburg.
3. Garey, M. R., D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, CA . (1979).
4. E. Gomory, On the relations between integer and noninteger solutions to linear programs, Proceedings of the National Academy of Sciences of the United States of America 53 (1965) 260-265.
5. Burke, E. K. B., G. Kendall, G. Whitwell, A new placement heuristic for the orthogonal stock cutting problem. Oper. Res. 52(4) (2004) 655–671.
6. Kroger, B., Guillotineable bin packing: A genetic approach. Eur. J. Oper. Res. 84 (1995) 645–661.
7. Tovey, Tutorial on Computational Complexity. Interfaces, C.A. (2002) 32 (3), 30-61.
8. Gilmore, P. C., R. E. Gomory, A linear programming approach to the cutting stock problem. *Oper. Res.* (1961) 9 849–859.
9. Christofides, N., C. Whitlock, An algorithm for two-dimensional cutting problems. Oper. Res. (1977) 25(1) 30–44.
10. Beasley, J. E, An exact two-dimensional non-guillotine cutting tree search procedure. Oper. Res. (1985) 33(1) 49–64.
11. Albano, A., G. Sapuppo, Optimal allocation of two-dimensional irregular shapes using heuristic search methods. IEEE Trans. Systems, Man Cybernetics (1980) SMC-10 242–248.
12. Jakobs, S, On genetic algorithms for the packing of polygons. Eur. J. Oper. Res. (1996) 88 165–181.
13. Liu, D., H. Teng, An improved BL-algorithm for genetic algorithms of the orthogonal packing of rectangles. Eur. J. Oper. Res. (1999) 112 413–419.
14. Dagli, C. H., A. Hajakbari, Simulated annealing approach for solving stock cutting problem. Proc. IEEE Internat. Conf. Systems, Man, and Cybernetics. Los Angeles, CA, (1990) 221–223.
15. Jakobs, S, On genetic algorithms for the packing of polygons. Eur. J. Oper. Res. (1996) 88 165–181.

16. Hopper, E., B. C. H. Turton, A genetic algorithm for a 2D industrial packing problem. Comput. Indust. Engrg. (1999) 37 375–378.
17. Holland J. H, Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. Ann Arbor, MI: Univ. Of Michigan Press (1975).
18. Metropolis N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H., Teller E, Equation of State Calculation by Fast Computing Machines, J. of Chem. Phys., (1953) 21, p. 1087-1091.
19. Pokštas J., Listopadskis N., Euristinio pjaustymo uždavinio algoritmo realizacija, Matematika ir matematinis modeliavimas / Kauno technologijos universitetas. ISSN 1822-2757. Kaunas : Technologija. T. 2 (2006), p. 141-146.
20. Pokštas J., Listopadskis N., Negiljotininio pjaustymo uždavinio euristinių algoritmų analizė, Taikomoji matematika : VI studentų konferencijos pranešimų medžiaga / Kauno technologijos universitetas. ISBN 9955-25-044-5. Kaunas : Technologija, 2006, p. 57-59.
21. Pokštas J., Listopadskis N., Pjaustymo uždavinio algoritmo analizė, Lietuvos matematikos rinkinys : Lietuvos matematikų draugijos XLVII konferencijos mokslo darbai / Matematikos ir informatikos institutas, Lietuvos matematikų draugija, Vilniaus universitetas. ISSN 0132-2818. Vilnius : Matematikos ir informatikos institutas, 2006, T. 46, spec. nr, p. 384-389.

PRIEDAI

PROGRAMOS TEKSTAS

Klasių aprašai:

```
class Staciakampis
{
 public:
 Staciakampis()
 {
  ilgis = plotis = index = kiekis = x1 = y1 = x2 = y2 = 0;
 }
 int ilgis,
     plotis,
     index,
     kiekis;
 int x1,y1,
     x2,y2;
 int Plotas() {return ilgis* plotis;}
 void Pasukti() {int a = ilgis; ilgis = plotis; plotis = a;}
};
#include "Euristika.h"

//---------------------------------------
class Taskas
{
 public:
 int x,y,w,l;
 Taskas(int xx, int yy, int p, int i)
 {
  x =xx;
  y = yy;
  w = p;
  l = i;
 }
};
typedef Taskas* Tsk;
//-----------------------------------------------------
class BLF: public Euristika
{
 TList *Taskai;
 int NaudPl;
 bool ArYra(Taskas*, Taskas**);
 void RastiTarpoIlgPlot(Taskas*, Sablonas*);
 void Kairiausias(Taskas*, Sablonas*);
 void Zemiausias(Taskas*, Sablonas*);
 void IdetiSt(Staciakampis*, Taskas*, Sablonas*);
 void AtnaujintiTskSar(Taskas*, Staciakampis*,Sablonas*);
 public:
 PjaustymoPlanas* PjaustytiZKU();
 Sablonas* PjaustytiBLF(TList*);
 BLF(TList *D, int a, int b, bool c, int kiek) : Euristika(D, a, b, c, kiek)
 {
  Taskai = new TList();
 }
 virtual ~BLF()
 {
  for(int i = 0; i < Taskai->Count; i++)
   delete (Taskas*)Taskai->Items[i];
  delete Taskai;
 }
};
#include "Staciakampis.h"
#include "math.h"
//-------------------------------------------------------------
class Sablonas
{
 public:
 int Atliekos;
 /*
 long double Fitness()
 {
  long double fit;
  fit = expl(-(Atliekos));
```

```
 return fit;
}
*/
int Panaudojimas;
TList *Ispjauti;
TList *Neispjauti;
Sablonas()
{
 Atliekos = 0;
 Ispjauti = new TList();
 Neispjauti = new TList();
 Panaudojimas = 1;
}
~Sablonas()
{
 for(int i = 0; i < Ispjauti->Count; i++)
  delete (Staciakampis*)Ispjauti->Items[i];
 delete Ispjauti;
 delete Neispjauti;
}
};
#include "Sablonas.h"
//---------------------------------------
class PjaustymoPlanas
{
public:
int BendrosAtliekos()
{
 int sum = 0;
 for(int i = 0; i < Sablonai->Count; i++)
  sum += ((Sablonas*)Sablonai->Items[i])->Atliekos *
        ((Sablonas*)Sablonai->Items[i])->Panaudojimas;
 return sum;
}
TList *Neispjauti;
TList *Sablonai;
int BendrAtliekos;
PjaustymoPlanas()
{
 Neispjauti = new TList();
 Sablonai = new TList();
 BendrAtliekos = 0;
}
~PjaustymoPlanas()
{
 for(int i = 0; i < Neispjauti->Count; i++)
  delete (Staciakampis*)Neispjauti->Items[i];
 delete Neispjauti;
 for(int i = 0; i < Sablonai->Count; i++)
  delete (Sablonas*)Sablonai->Items[i];
 delete Sablonai;
}
};
#include <Classes.hpp>
#include "Pjaustymoplanas.h"
#include "values.h"
//---------------------------------------------------------
typedef Staciakampis* St;
class Euristika
{
int *Lygiai, *LygiaiT;
int* RastiTarpa(int*,int*,Staciakampis*,bool*);
void ZemiausiasTarpas(int*, int*, int*);
void Uzpildyti(int*, int*, int*);

public:
TList *DD;
PjaustymoPlanas* pl;
Staciakampis *Staciakamp;
TList *Pjaustymui;
bool RibotosZal;
int SabKiek, SablKiek;
TList *Uzsakymas, *UzsakymasGr, *SabGr;
int W,L;
PjaustymoPlanas* PjaustytiBF();
Sablonas* BF();
int SablonoPanaudojimas(Sablonas*);
TList* GrupuotiStac(TList*);
Sablonas* KopijuotiSab(Sablonas*);
```

```
 Sablonas* PjaustytiLG(TList*);
 PjaustymoPlanas* PjaustytiZT();
 Euristika(TList *D, int a, int b, bool c, int kiek)
 {
  Uzsakymas = new TList();
  for(int i = 0; i < D->Count; i++)
  {
   Staciakamp = new Staciakampis();
   *Staciakamp = *(Staciakampis*)D->Items[i];
   Uzsakymas->Add(Staciakamp);
  }
  W = a; L = b;
  pl = new PjaustymoPlanas();
  Pjaustymui = new TList();
  Lygiai = new int[L];
  LygiaiT = new int[L];
  RibotosZal = c;
  SabKiek = kiek;
  DD = D;
 }
 virtual ~Euristika()
 {
  for(int i = 0; i < Uzsakymas->Count; i++)
   delete (Staciakampis*)Uzsakymas->Items[i];
  delete pl;
  delete Uzsakymas;
  delete Pjaustymui;
  delete [] Lygiai;
  delete [] LygiaiT;
 }
};
#include "BLF.h"
//---------------------------------------------------
typedef Sablonas* Sab;
class GenetinisAlg : public BLF
{
 TList *Populiacija, *NPopuliacija, *inn;
 int IterSk, PopDyd, elit, Kryz;
 double TikKryz,
       TikMut;
 void GeneruotiPop();
 int Atranka();
 TList* KryzminimasDv(Sablonas*, Sablonas*);
 TList* KryzminimasTol(Sablonas*, Sablonas*);
 TList* Mutacija(TList*);

 public:

 TList *in;
 bool ArBLF;
 int Atsitiktinis;
 PjaustymoPlanas* AtrinktiGA();
 GenetinisAlg(TList *D, int W, int L, int popDyd, int iterSk, double tikKryz,
    double tikMut, double elite, bool RibotosZal, int SabKiek, bool ArBLf,
    int Ats, int Kryz) : BLF(D, W, L, RibotosZal, SabKiek)
 {
  Populiacija = new TList();
  NPopuliacija = new TList();
  PopDyd = popDyd; IterSk = iterSk;
  TikKryz = tikKryz; TikMut = tikMut;
  in = new TList();
  inn = new TList();
  elit = floor(PopDyd * elite);
  Atsitiktinis = Ats;
  ArBLF =  ArBLf;
 }
 virtual ~GenetinisAlg()
 {
  for(int i = 0; i < Populiacija->Count; i++)
   delete (Sablonas*)Populiacija->Items[i];
  delete Populiacija;
  for(int i = 0; i < NPopuliacija->Count; i++)
   delete (Sablonas*)NPopuliacija->Items[i];
  delete NPopuliacija;
  delete in;
  delete inn;
 }
};
#include "GenetinisAlg.h"
```

```
#include "stdio.h"
//---------------------------------------------------------------
class AtkaitinimoAlg : public GenetinisAlg
{
 TList* Kaimynas(TList*, TList*, double);
 int  MaxIter, MaxTempIter;
 public:
 PjaustymoPlanas* AtrinktiSA();
 AtkaitinimoAlg(TList *D, int W, int L, int popDyd, int iterSk, double tikKryz,
     double tikMut, double elite, bool RibotosZal, int SabKiek, bool ArBLf, int Ats, int Kryz)
               : GenetinisAlg(D, W, L, popDyd, iterSk, tikKryz, tikMut, elite,
                 RibotosZal, SabKiek, ArBLf, Ats, Kryz)
 {
  MaxIter = iterSk;
  MaxTempIter = popDyd;
 }
 virtual ~AtkaitinimoAlg()
 {
 }
};
//---------------------------------------------------------------
```

Metodų aprašai:

```
#include "Euristika.h"
//---------------------------------------------------------------------------
int __fastcall Kriter(void * Item1, void * Item2)
{
  if (St(Item1)->ilgis > St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis > St(Item2)->plotis)
    return -1;
  else
  if (St(Item1)->ilgis < St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis < St(Item2)->plotis)
    return 1;
  else
    return 0;
}
//---------------------------------------------------------------------------
Sablonas* Euristika::PjaustytiLG(TList *Ind)
{
 Sablonas *Sab = new Sablonas(); //Sablonas
 Staciakampis *St, *t;
 bool neispjauta = false;
 int  NaudPl = 0, Nuostoliai;
 int ilg, in, sum = 0;
 bool yra;
 Pjaustymui->Clear();
 for(int i = 0; i < L; i++)
  Lygiai[i] = LygiaiT[i] = 0;
 for(int i = 0; i < Ind->Count; i++)
  Pjaustymui->Add(Uzsakymas->Items[*(int*)Ind->Items[i]]);
 while(Pjaustymui->Count != 0)
 {
  St = (Staciakampis*) Pjaustymui->First();
  // Ilgis turi buti didesnis uz ploti
  if(St->ilgis < St->plotis)
   St->Pasukti();
  //Tarpo suradimas figurai
  Lygiai = RastiTarpa(&in, &ilg, St, &neispjauta);
  if(neispjauta)
  {
   Pjaustymui->Remove(St);
   Sab->Neispjauti->Add(St);
   continue;
  }
  t = new Staciakampis();
  *t = *St;
  NaudPl += t->Plotas();
  for(int j = 0; j < t->ilgis; j++)
  {
   Lygiai[in + j] += t->plotis;
   LygiaiT[in + j] = Lygiai[in + j];
  }
  t->x1 = in;
  t->y1 = Lygiai[in];
  t->x2 = in + t->ilgis;
  t->y2 = Lygiai[in] - t->plotis;
  Sab->Ispjauti->Add(t);
```

```
  St = (Staciakampis*)Pjaustymui->First();
  Pjaustymui->Remove(St);
 }
 for(int i = 0; i < L; i++)
  sum += LygiaiT[i];
 Sab->Atliekos = W*L - NaudPl;
 return Sab;
}
//---------------------------------------------------------------------------
int* Euristika::RastiTarpa(int *in, int *ilg, Staciakampis *St, bool *a)
{
 *a = false;
 for(int i = 0; i < L; i++)
  Lygiai[i] = LygiaiT[i];
 while(true)
 {
  ZemiausiasTarpas(in,ilg,Lygiai);
  if(St->ilgis <= *ilg && St->plotis <= W - Lygiai[*in])
   return Lygiai;
  else
   if(St->plotis <= *ilg && St->ilgis <= W - Lygiai[*in])
   {
    St->Pasukti();
    return Lygiai;
   }
   else
    if(*in == 0 && *ilg == L)
    {
     *a = true;
     return Lygiai;
    }
    else
     Uzpildyti(in,ilg,Lygiai);
  }
}
//---------------------------------------------------------------------------
void Euristika::ZemiausiasTarpas(int *in, int *ilg, int *mas)
{
 int i,x;
 *in = 0;
 x = mas[0]; *ilg = 1;
 for(i = 0; i < L; i++)
  if(mas[i] < x)
  {
   x = mas[i];
   *in = i;
  }
 i = *in;
 while(i < L-1 && mas[i] == mas[i + 1])
 {
  (*ilg)++;
   i++;
 }
}
//---------------------------------------------------------------------------
void Euristika::Uzpildyti(int *in, int *ilg, int *mas)
{
  if(*in == 0)
  {
   if(mas[*in + *ilg] < W)
   {
    for(int j = 0; j < *ilg; j++)
      mas[*in + j] = mas[*in + *ilg];
   }
   else
   {
    for(int j = 0; j < *ilg; j++)
      mas[*in + j] = W;
   }
  }
  if(*in + *ilg == L)
   {
    if(mas[*in - 1] < W)
    {
     for(int j = 0; j < *ilg; j++)
      mas[*in + j] = mas[*in - 1];
    }
    else
    {
```

```
      for(int j = 0; j < *ilg; j++)
        mas[*in + j] = W;
     }
    }
    else
     if(*in != 0 && mas[*in - 1] <= mas[*in + *ilg])
     {
      if(mas[*in - 1] < W)
      {
       for(int j = 0; j < *ilg; j++)
        mas[*in + j] = mas[*in - 1];
      }
      else
      {
       for(int j = 0; j < *ilg; j++)
        mas[*in + j] = W;
      }
     }
     else
     {
      if(mas[*in + *ilg] < W)
      {
       for(int j = 0; j < *ilg; j++)
        mas[*in + j] = mas[*in + *ilg];
      }
      else
      {
       for(int j = 0; j < *ilg; j++)
        mas[*in + j] = W;
      }
     }
}
//---------------------------------------------------------------------------
int Euristika::SablonoPanaudojimas(Sablonas *Sab)
{
 Staciakampis *St1, *St2;
 int kiek, minVal = MAXINT;
 UzsakymasGr = NULL;
 SabGr = NULL;
 SabGr = GrupuotiStac(Sab->Ispjauti);
 UzsakymasGr = GrupuotiStac(Uzsakymas);
 for(int i = 0; i < SabGr->Count; i++)
 {
  St1 = (Staciakampis*)SabGr->Items[i];
  for(int j = 0; j < UzsakymasGr->Count; j++)
  {
   St2 = (Staciakampis*)UzsakymasGr->Items[j];
   if(St2->ilgis == St1->ilgis && St2->plotis == St1->plotis ||
      St2->ilgis == St1->plotis && St2->plotis == St1->ilgis)
     {
       kiek = floor(St2->kiekis/St1->kiekis);
       if(minVal > kiek)
        minVal = kiek;
     }
   }
  }
  if(RibotosZal)
  {
   minVal = min(minVal, SabKiek - SablKiek);
   if(minVal == 0)
    return minVal;
  }
 // Uzsakymo atnaujinimas
 for(int i = 0; i < SabGr->Count; i++)
 {
  St1 = (Staciakampis*)SabGr->Items[i];
  kiek = 0;
  for(int j = 0; j < Uzsakymas->Count; j++)
  {
   St2 = (Staciakampis*)Uzsakymas->Items[j];
   if(St2->ilgis == St1->ilgis && St2->plotis == St1->plotis ||
      St2->ilgis == St1->plotis && St2->plotis == St1->ilgis)
     {
       kiek++;
       if(kiek <= St1->kiekis * minVal)
       {
        Uzsakymas->Remove(St2);
        delete St2;
        j--;
```

```
      }
     }
    }
   }
  for(int j = 0; j < Uzsakymas->Count; j++)
   ((Staciakampis*)Uzsakymas->Items[j])->index = j;
  if(SabGr != NULL)
   delete SabGr;
  if(UzsakymasGr != NULL)
   delete UzsakymasGr;
  return minVal;
}
//---------------------------------------------------------------------------
TList* Euristika::GrupuotiStac(TList *list)
{
 TList *temp = new TList();
 Staciakampis *St;
 int kiek, k = 0;
 for(int i = 0; i < list->Count; i++)
 {
  St = (Staciakampis*)list->Items[i];
  temp->Add(St);
 }
 for(int i = 0; i < list->Count; i++)
 {
  kiek = 0;
  for(int j = k; j < temp->Count; j++)
   if(((Staciakampis*)temp->Items[j])->ilgis ==
      ((Staciakampis*)list->Items[i])->ilgis &&
      ((Staciakampis*)temp->Items[j])->plotis ==
      ((Staciakampis*)list->Items[i])->plotis ||
      ((Staciakampis*)temp->Items[j])->ilgis ==
      ((Staciakampis*)list->Items[i])->plotis &&
      ((Staciakampis*)temp->Items[j])->plotis ==
      ((Staciakampis*)list->Items[i])->ilgis)
   {
     kiek++;
     St = (Staciakampis*)temp->Items[j];
     temp->Remove(St);
     j--;
   }
   if(kiek > 0)
   {
    St = (Staciakampis*)list->Items[i];
    St->kiekis = kiek;
    temp->Insert(k,St);
    k++;
   }
 }
 return temp;
}
//---------------------------------------------------------------------------
Sablonas* Euristika::KopijuotiSab(Sablonas* sab)
{
 Sablonas *kop = new Sablonas();
 Staciakampis *Stac;
 kop->Atliekos = sab->Atliekos;
 kop->Panaudojimas = sab->Panaudojimas;
 for(int j = 0; j < sab->Ispjauti->Count; j++)
 {
  Stac = new Staciakampis();
  *Stac = *(Staciakampis*)sab->Ispjauti->Items[j];
  kop->Ispjauti->Add(Stac);
 }
 for(int j = 0; j < sab->Neispjauti->Count; j++)
  kop->Neispjauti->Add(sab->Neispjauti->Items[j]);
 return kop;
}
//---------------------------------------------------------------------------
Sablonas* Euristika::BF()
{
 Sablonas *Sab = new Sablonas();
 Staciakampis *St, *t;
 bool neispjauta;
 int  NaudPl = 0, Nuostoliai;
 int ilg, in;
 Pjaustymui->Clear();
 for(int i = 0; i < L; i++)
  Lygiai[i] = 0;
```

```
for(int i = 0; i < Uzsakymas->Count; i++)
{
 St = (Staciakampis*) Uzsakymas->Items[i];
 if(St->ilgis < St->plotis)
  St->Pasukti();
 Pjaustymui->Add(St);
}
Pjaustymui->Sort(Kriter);
while(Pjaustymui->Count != 0)
{
 // Randamas zemiausias tarpas
 ZemiausiasTarpas(&in, &ilg, Lygiai);
 // Tikrinama, ar uzpildytas lakstas
 if(in == 0 && Lygiai[in] == W)
 {
  for(int i = 0; i < Pjaustymui->Count; i++)
   Sab->Neispjauti->Add(Pjaustymui->Items[i]);
  break;
 }
 // Tarpui surandama geriausiai atitinkanti figura
 neispjauta = true;
 for(int i = 0; i < Pjaustymui->Count; i++)
 {
  St = (Staciakampis*) Pjaustymui->Items[i];
  if(St->ilgis <= ilg && St->plotis <= W - Lygiai[in])
  {
   t = new Staciakampis();
   *t = *St;
   NaudPl += t->Plotas();
   for(int j = 0; j < t->ilgis; j++)
    Lygiai[in + j] += t->plotis;
   t->x1 = in;
   t->y1 = Lygiai[in];
   t->x2 = in + t->ilgis;
   t->y2 = Lygiai[in] - t->plotis;
   Sab->Ispjauti->Add(t);
   Pjaustymui->Remove(St);
   neispjauta = false;
   break;
  }
  else
   if(St->plotis <= ilg && St->ilgis <= W - Lygiai[in])
   {
    St->Pasukti();
    t = new Staciakampis();
    *t = *St;
    NaudPl += t->Plotas();
    for(int j = 0; j < t->ilgis; j++)
     Lygiai[in + j] += t->plotis;
    t->x1 = in;
    t->y1 = Lygiai[in];
    t->x2 = in + t->ilgis;
    t->y2 = Lygiai[in] - t->plotis;
    Sab->Ispjauti->Add(t);
    Pjaustymui->Remove(St);
    neispjauta = false;
    break;
   }
 }
 // Jeigu tokios figuros rasti nepavyko,
 // tuomet tas plotas interpretuojamas kaip atliekos
 if(neispjauta)
  Uzpildyti(&in, &ilg, Lygiai);
}
Sab->Atliekos = W*L - NaudPl;
return Sab;
}
//---------------------------------------------------------------------------
PjaustymoPlanas* Euristika::PjaustytiBF()
{
 Sablonas *sab;
 SablKiek = 0;
 while(Uzsakymas->Count != 0)
 {
  sab = BF();
  sab->Panaudojimas = SablonoPanaudojimas(sab);
  if(sab->Panaudojimas != 0)
   pl->Sablonai->Add(sab);
  SablKiek += sab->Panaudojimas;
```

```
  if(SablKiek == SabKiek)
   break;
 }
 return pl;
}
//---------------------------------------------------------------------------
PjaustymoPlanas* Euristika::PjaustytiZT()
{
 Sablonas *sab, *sab2 = new Sablonas();
 Staciakampis* St;
 int *ii;
 SablKiek = 0;
 while(Uzsakymas->Count != 0)
 {
  for(int i = 0; i < Uzsakymas->Count; i++)
  {
   St = (Staciakampis*) Uzsakymas->Items[i];
   if(St->ilgis < St->plotis)
    St->Pasukti();
  }
  Uzsakymas->Sort(Kriter);
  sab2->Ispjauti->Clear();
  for(int i = 0; i < Uzsakymas->Count; i++)
  {
   ii = new int;
   *ii = i;
   sab2->Ispjauti->Add(ii);
  }
  sab = PjaustytiLG(sab2->Ispjauti);
  sab->Panaudojimas = SablonoPanaudojimas(sab);
  if(sab->Panaudojimas != 0)
   pl->Sablonai->Add(sab);
  SablKiek += sab->Panaudojimas;
  if(SablKiek == SabKiek)
   break;
 }
 delete sab2->Ispjauti;
 return pl;
}
//---------------------------------------------------------------------------
#include "BLF.h"
//---------------------------------------------------
int __fastcall Palyginimui(void * Item1, void * Item2)
{
  if ( Tsk(Item1)->y < Tsk(Item2)->y
       || Tsk(Item1)->y == Tsk(Item2)->y
       && Tsk(Item1)->x < Tsk(Item2)->x )
    return -1;
  else
  if ( Tsk(Item1)->y > Tsk(Item2)->y
       || Tsk(Item1)->y == Tsk(Item2)->y
       && Tsk(Item1)->x > Tsk(Item2)->x )
    return 1;
  else
    return 0;
}
//---------------------------------------------------------------------------
int __fastcall Kriteri(void * Item1, void * Item2)
{
  if (St(Item1)->ilgis > St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis > St(Item2)->plotis)
    return -1;
  else
  if (St(Item1)->ilgis < St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis < St(Item2)->plotis)
    return 1;
  else
    return 0;
}
//---------------------------------------------------
Sablonas* BLF::PjaustytiBLF(TList *Ind)
{
 Sablonas *Sab = new Sablonas(); //Sablonas
 int k;
 Taskas *tsk;
 Staciakampis *St;

 NaudPl = 0;
 Pjaustymui->Clear();
```

```
 for(int i = 0; i < Taskai->Count; i++)
  delete (Taskas*)Taskai->Items[i];
 Taskai->Clear();

 for(int i = 0; i < Ind->Count; i++)
  Pjaustymui->Add(Uzsakymas->Items[*(int*)Ind->Items[i]]);
 tsk = new Taskas(0, 0, W, L);
 Taskai->Add(tsk);
 while(Pjaustymui->Count != 0)
 {
  St = (Staciakampis*) Pjaustymui->First();
  if(St->ilgis < St->plotis)
   St->Pasukti();
  Taskai->Sort(Palyginimui);
  k = 0;
  for(int i = 0; i < Taskai->Count; i++)
  {
   if(k == 1)
    break;
   tsk = (Taskas*)Taskai->Items[i];
   if(St->ilgis <= tsk->l && St->plotis <= tsk->w)
   {
    IdetiSt(St,tsk,Sab);
    delete(tsk);
    NaudPl += St->Plotas();
    k = 1;
   }
   else
    if(St->plotis <= tsk->l && St->ilgis <= tsk->w)
    {
     St->Pasukti();
     IdetiSt(St,tsk,Sab);
     delete(tsk);
     NaudPl += St->Plotas();
     k = 1;
    }
  }
  if(k == 0)
  {
   Sab->Neispjauti->Add(St);
   Pjaustymui->Remove(St);
  }
 }
 Sab->Atliekos = W*L - NaudPl;
 return Sab;
}
//---------------------------------------------------
void BLF::IdetiSt(Staciakampis* St, Taskas* Tsk, Sablonas *Sab)
{
 Staciakampis *t = new Staciakampis();
 Taskas *tsk, *tskp, *pt;
 *t = *St;
 t->x1 = Tsk->x;
 t->y1 = Tsk->y + St->plotis;
 t->x2 = Tsk->x + St->ilgis;
 t->y2 = Tsk->y;
 Sab->Ispjauti->Add(t);
 Taskai->Remove(Tsk);
 Pjaustymui->Remove(St);
 AtnaujintiTskSar(Tsk,t,Sab);
 if(Tsk->w - St->plotis != 0)
 {
  tsk = new Taskas(t->x1, t->y1, 0, 0);
  RastiTarpoIlgPlot(tsk,Sab);
  if(!ArYra(tsk, &pt))
  {
   if(tsk->w != 0 && tsk->l != 0)
   {
    Taskai->Add(tsk);
    tskp = new Taskas(t->x1, t->y1, 0, 0);
    Kairiausias(tskp,Sab);
    RastiTarpoIlgPlot(tskp,Sab);
    if(!ArYra(tskp, &pt))
     if(tskp->w != 0 && tskp->l != 0)
      Taskai->Add(tskp);
     else
     {
      if(tskp->w != 0 && tskp->l != 0)
      {
```

```
      pt->l = tskp->l;
      pt->w = tskp->w;
     }
     else
     {
      Taskai->Remove(pt);
      delete pt;
     }
     delete tskp;
    }
   }
  }
  else
  {
   if(tsk->w != 0 && tsk->l != 0)
   {
    pt->l = tsk->l;
    pt->w = tsk->w;
   }
   else
   {
    Taskai->Remove(pt);
    delete pt;
   }
   delete tsk;
  }
 }

 if(Tsk->l - St->ilgis != 0)
 {
  tsk = new Taskas(t->x2, t->y2, 0, 0);
  RastiTarpoIlgPlot(tsk,Sab);
  if(!ArYra(tsk, &pt))
  {
   if(tsk->w != 0 && tsk->l != 0)
   {
    Taskai->Add(tsk);
    tskp = new Taskas(t->x2, t->y2, 0, 0);
    Zemiausias(tskp,Sab);
    RastiTarpoIlgPlot(tskp,Sab);
    if(!ArYra(tskp, &pt))
     if(tskp->w != 0 && tskp->l != 0)
      Taskai->Add(tskp);
    else
    {
     if(tskp->w != 0 && tskp->l != 0)
     {
      pt->l = tskp->l;
      pt->w = tskp->w;
     }
     else
     {
      Taskai->Remove(pt);
      delete pt;
     }
     delete tskp;
    }
   }
  }
  else
  {
   if(tsk->w != 0 && tsk->l != 0)
   {
    pt->l = tsk->l;
    pt->w = tsk->w;
   }
   else
   {
    Taskai->Remove(pt);
    delete pt;
   }
   delete tsk;
  }
 }
}
//---------------------------------------------------
void BLF::Kairiausias(Taskas* tsk, Sablonas *Sab)
{
 int x = tsk->x;
```

```
 int kx = 0;
 TList *St = Sab->Ispjauti;
 Staciakampis *t;
 if(x == 0)
  return;
 while(x != 0)
 {
  for(int i = 0; i < St->Count; i++)
  {
   t = (Staciakampis*)St->Items[i];
   if(t->y2 <= tsk->y && tsk->y < t->y1 && x == t->x2)
   {
    kx = x;
    x = 1;
    break;
   }
  }
  x--;
 }
 tsk->x = kx;
}
//---------------------------------------------------
void BLF::Zemiausias(Taskas* tsk, Sablonas *Sab)
{
 int y = tsk->y;
 int ky = 0;
 TList *St = Sab->Ispjauti;
 Staciakampis *t;
 if(y == 0)
  return;
 while(y != 0)
 {
  for(int i = 0; i < St->Count; i++)
  {
   t = (Staciakampis*)St->Items[i];
   if(t->x1 <= tsk->x && tsk->x < t->x2 && y == t->y1)
   {
    ky = y;
    y = 1;
    break;
   }
  }
  y--;
 }
 tsk->y = ky;
}
//---------------------------------------------------
void BLF::AtnaujintiTskSar(Taskas* tsk, Staciakampis *St, Sablonas *Sab)
{
 Taskas *t, *ktsk, *pt;
 int x = tsk->x;
 int y = tsk->y;
 //Atnaujina zemiau esanciu tasku aukscius ir
 //kairiau esanciu tasku ilgius
 for(int i = 0; i < Taskai->Count; i++)
 {
  t = (Taskas*)Taskai->Items[i];
  if(t->y < tsk->y && t->x >= tsk->x && t->x < tsk->x+ St->ilgis)
  {
   if(t->y + t->w >= tsk->y)
   {
    t->w = tsk->y - t->y;
    if(t->x != tsk->x)
    {
     ktsk = new Taskas(t->x, tsk->y + St->plotis, 0, 0);
     RastiTarpoIlgPlot(ktsk,Sab);
     if(!ArYra(ktsk, &pt))
      if(ktsk->w != 0 && ktsk->l != 0)
      Taskai->Add(ktsk);
     else
     {
      if(ktsk->w != 0 && ktsk->l != 0)
      {
       pt->l = ktsk->l;
       pt->w = ktsk->w;
      }
      else
      {
       Taskai->Remove(pt);
```

```
      delete pt;
     }
     delete ktsk;
    }
   }
  }
 }
 if(t->x < tsk->x && t->y >= tsk->y && t->y < tsk->y + St->plotis)
 {
  if(t->x + t->l >= tsk->x)
  {
   t->l = tsk->x - t->x;
   if(t->y != tsk->y)
   {
    ktsk = new Taskas(tsk->x + St->ilgis, t->y, 0, 0);
    RastiTarpoIlgPlot(ktsk,Sab);
    if(!ArYra(ktsk, &pt))
     if(ktsk->w != 0 && ktsk->l != 0)
      Taskai->Add(ktsk);
    else
    {
     if(ktsk->w != 0 && ktsk->l != 0)
     {
      pt->l = ktsk->l;
      pt->w = ktsk->w;
     }
     else
     {
      Taskai->Remove(pt);
      delete pt;
     }
     delete ktsk;
    }
   }
  }
 }
}
//Isima arba perteklia taskus esancius staciakampio
//horizontalioje apatineje linijoje
for(int j = x; j < x + St->ilgis; j++)
{
 for(int i = 0; i < Taskai->Count; i++)
 {
  t = (Taskas*)Taskai->Items[i];
  if(t->x == j && t->y == tsk->y)
  {
   ktsk = new Taskas(t->x, t->y + St->plotis, 0,0);
   RastiTarpoIlgPlot(ktsk, Sab);

   if(!ArYra(ktsk, &pt))
    if(ktsk->w != 0 && ktsk->l != 0)
     Taskai->Add(ktsk);
   else
   {
    if(ktsk->w != 0 && ktsk->l != 0)
    {
     pt->l = ktsk->l;
     pt->w = ktsk->w;
    }
    else
    {
     Taskai->Remove(pt);
     delete pt;
    }
    delete ktsk;
   }
   Taskai->Remove(t);
   delete t;
   i--;
  }
 }
}
//Isima ir perkelia taskus esancius staciakampio
//vertikalioje kaireje linijoje
for(int j = y; j < y + St->plotis; j++)
{
 for(int i = 0; i < Taskai->Count; i++)
 {
  t = (Taskas*)Taskai->Items[i];
```

```
   if(t->x == tsk->x && t->y == j)
   {
    ktsk = new Taskas(t->x + St->ilgis, t->y, 0,0);
    RastiTarpoIlgPlot(ktsk, Sab);

    if(!ArYra(ktsk, &pt))
     if(ktsk->w != 0 && ktsk->l != 0)
      Taskai->Add(ktsk);
    else
    {
     if(ktsk->w != 0 && ktsk->l != 0)
     {
      pt->l = ktsk->l;
      pt->w = ktsk->w;
     }
     else
     {
      Taskai->Remove(pt);
      delete pt;
     }
     delete ktsk;
    }
    Taskai->Remove(t);
    delete t;
    i--;
   }
  }
 }
}
//---------------------------------------------------------
void BLF::RastiTarpoIlgPlot(Taskas* tsk, Sablonas *Sab)
{
 bool testi = true;
 int x = tsk->x;
 int y = tsk->y;
 TList *t = Sab->Ispjauti;
 Staciakampis *st;
 while(x < L && testi)
 {
  for(int i = 0; i < t->Count; i++)
  {
   st = (Staciakampis*)t->Items[i];
   if(st->y2 <= y && y < st->y1 && x == st->x1)
   {
    testi = false;
    x--;
    break;
   }
  }
  x++;
 }
 tsk->l = x - tsk->x;
 testi = true;
 x = tsk->x;
 while(y < W && testi)
 {
  for(int i = 0; i < t->Count; i++)
  {
   st = (Staciakampis*)t->Items[i];
   if(st->x1 <= x && x < st->x2 && y == st->y2)
   {
    testi = false;
    y--;
    break;
   }
  }
  y++;
 }
 tsk->w = y - tsk->y;
}
//---------------------------------------------------------
bool BLF::ArYra(Taskas *tsk, Taskas **pt)
{
 bool yra = false;
 Taskas *t;
 *pt = NULL;
 for(int i = 0; i < Taskai->Count; i++)
 {
  t = (Taskas*)Taskai->Items[i];
```

```
  if(t->x == tsk->x && t->y == tsk->y)
  {
   yra = true;
   *pt = t;
   break;
  }
 }
 return yra;
}
//---------------------------------------------------------
PjaustymoPlanas* BLF::PjaustytiZKU()
{
 Sablonas *sab, *sab2 = new Sablonas();                    ;
 Staciakampis* St;
 int *ii;
 SablKiek = 0;
 while(Uzsakymas->Count != 0)
 {
  for(int i = 0; i < Uzsakymas->Count; i++)
  {
   St = (Staciakampis*) Uzsakymas->Items[i];
   if(St->ilgis < St->plotis)
    St->Pasukti();
  }
  Uzsakymas->Sort(Kriteri);
  sab2->Ispjauti->Clear();
  for(int i = 0; i < Uzsakymas->Count; i++)
  {
   ii = new int;
   *ii = i;
   sab2->Ispjauti->Add(ii);
  }
  sab = PjaustytiBLF(sab2->Ispjauti);
  sab->Panaudojimas = SablonoPanaudojimas(sab);
  if(sab->Panaudojimas != 0)
   pl->Sablonai->Add(sab);
  SablKiek += sab->Panaudojimas;
  if(SablKiek == SabKiek)
   break;
 }
 delete sab2->Ispjauti;
 return pl;
}
//---------------------------------------------------------------------------
#include "GenetinisAlg.h"
#include "stdio.h"
//---------------------------------------------------------------------------

int __fastcall Kriterijus(void * Item1, void * Item2)
{
  if (Sab(Item1)->Atliekos < Sab(Item2)->Atliekos)
    return -1;
  else
  if (Sab(Item1)->Atliekos > Sab(Item2)->Atliekos)
    return 1;
  else
    return 0;
}
//---------------------------------------------------
int __fastcall Kriteri(void * Item1, void * Item2)
{
  if (St(Item1)->ilgis > St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis > St(Item2)->plotis)
    return -1;
  else
  if (St(Item1)->ilgis < St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis < St(Item2)->plotis)
    return 1;
  else
    return 0;
}
//---------------------------------------------------------------------------
PjaustymoPlanas* GenetinisAlg::AtrinktiGA()
{
 int k1, k2;
 Sablonas *Sab, *temp;
 SablKiek = 0;
 //FILE *F = fprintf("Log.txt","w");
 while(Uzsakymas->Count != 0)
```

```
{
 GeneruotiPop();
 for(int i = 0; i < IterSk; i++)
 {
  for(int k = 0; k < elit; k++)
   NPopuliacija->Add((Sablonas*)Populiacija->Items[k]);
  for(int j = elit; j < PopDyd; j++)
  {
   //FILE *F = fopen("Log.txt","w");

   k1 = Atranka();
   k2 = Atranka();
   /*
   //-----
   fprintf(F,"\n");
   for(int i = 0; i < ((Sablonas*)Populiacija->Items[k1])->Ispjauti->Count; i++)
     fprintf(F,"%d\n",((Staciakampis*)((Sablonas*)Populiacija->Items[k1])->Ispjauti->Items[i])->index );
   fprintf(F,"\n");
   for(int i = 0; i < ((Sablonas*)Populiacija->Items[k1])->Ispjauti->Count; i++)
     fprintf(F,"%d\n",((Staciakampis*)((Sablonas*)Populiacija->Items[k1])->Ispjauti->Items[i])->index );
   fprintf(F,"\n");
   fclose(F);
   */

   if(Kryz == 1)
    in = KryzminimasDv((Sablonas*)Populiacija->Items[k1],
                       (Sablonas*)Populiacija->Items[k2]);
   else
    in = KryzminimasTol((Sablonas*)Populiacija->Items[k1],
                       (Sablonas*)Populiacija->Items[k2]);

   in = Mutacija(in);

   if(ArBLF)
    Sab = PjaustytiBLF(in);
   else
    Sab = PjaustytiLG(in);

   NPopuliacija->Add(Sab);

   if(Sab->Atliekos == 0)
   {
     i = IterSk; j = PopDyd;
   }
  }
  for(int k = elit; k < Populiacija->Count; k++)
   NPopuliacija->Add(Populiacija->Items[k]);
  NPopuliacija->Sort(Kriterijus);
  Populiacija->Clear();
  for(int k = 0; k < PopDyd; k++)
   Populiacija->Add(NPopuliacija->Items[k]);
  for(int k = PopDyd; k < NPopuliacija->Count; k++)
   delete (Sablonas*)NPopuliacija->Items[k];
  NPopuliacija->Clear();
 }
 Populiacija->Sort(Kriterijus);
 temp = KopijuotiSab((Sablonas*)Populiacija->Items[0]);
 temp->Panaudojimas = SablonoPanaudojimas(temp);
 SablKiek += temp->Panaudojimas;
 if(temp->Panaudojimas != 0)
  pl->Sablonai->Add(temp);
 if(SablKiek == SabKiek)
  break;
}
return pl;
}
//---------------------------------------------------------------------------
void GenetinisAlg::GeneruotiPop()
{
 in->Clear();
 int *ind;
 Staciakampis *Staciakamp, *St;
 int kk = 0;
 Sablonas * Sab;
 for(int k = 0; k < Populiacija->Count; k++)
  delete (Sablonas*)Populiacija->Items[k];
 Populiacija->Clear();
 if(Atsitiktinis == 2)
 {
```

```
 for(int i = 0; i < Uzsakymas->Count; i++)
 {
  St = (Staciakampis*) Uzsakymas->Items[i];
  if(St->ilgis < St->plotis)
   St->Pasukti();
 }
 Uzsakymas->Sort(Kriteri);
}

for(int i = 0; i < Uzsakymas->Count; i++)
{
 if(Atsitiktinis == 2)
 {
  ind = new int;
  *ind = i;
 }
 else
  ind = &((Staciakampis *)Uzsakymas->Items[i])->index;
 in->Add(ind);
}

if(Atsitiktinis == 1)
{
 Populiacija->Add(BF());
 kk = 1;
}
else
 if(Atsitiktinis == 2)
 {
  Populiacija->Add(PjaustytiBLF(in));
  kk = 1;
 }

for(int k = kk; k < PopDyd; k++)
{
 for(int i = 0; i < in->Count; i++)
  in->Exchange(i,rand()%in->Count);
 if(ArBLF)
  Sab = PjaustytiBLF(in);
 else
  Sab = PjaustytiLG(in);
 Populiacija->Add(Sab);
}
}
//---------------------------------------------------------------------------
int GenetinisAlg::Atranka()
{
 /*
 //Atsitiktine atranka
 return rand() % PopDyd;
 */
 // Tikimybine atranka
 int  j = 0;
 int sumFit = 0, sumPart = 0;
 double ad;
 for(int i = 0; i < PopDyd; i++)
  sumFit += ((Sablonas*)Populiacija->Items[i])->Atliekos;
 ad = rand() % sumFit;
 while(true)
 {
  if(ad >= sumPart && ad < sumPart+((Sablonas*)Populiacija->Items[j])->Atliekos)
    return j;
   sumPart += ((Sablonas*)Populiacija->Items[j])->Atliekos;
   j++;
  }
}
//---------------------------------------------------------------------------
TList* GenetinisAlg::KryzminimasDv(Sablonas* s1, Sablonas* s2)
{
 int *aa, c1, c2;
 int kk, a = -1;
 int minLen;

 in->Clear();
 inn->Clear();

 for(int i = 0; i < s1->Ispjauti->Count; i++)
 {
  aa = &((Staciakampis*)s1->Ispjauti->Items[i])->index;
```

```
 in->Add(aa);
}
for(int i = 0; i < s1->Neispjauti->Count; i++)
{
 aa = &((Staciakampis*)s1->Neispjauti->Items[i])->index;
 in->Add(aa);
}

if((double)rand()/RAND_MAX < 1 - TikKryz)
 return in;

for(int i = 0; i < s2->Ispjauti->Count; i++)
{
 aa = &((Staciakampis*)s2->Ispjauti->Items[i])->index;
 inn->Add(aa);
}
for(int i = 0; i < s2->Neispjauti->Count; i++)
{
 aa = &((Staciakampis*)s2->Neispjauti->Items[i])->index;
 inn->Add(aa);
}

minLen = min(s1->Ispjauti->Count, s2->Ispjauti->Count);

if(minLen <= 2)
 return in;

while(true)
{
 c1 = rand()%(minLen-1);
 c2 = c1 + rand()%(minLen-c1);
 if(c2 - c1 >= 2 && c2 != c1)
  break;
}

for(int i = c1+1; i < c2; i++)
 for(int j = 0; j < inn->Count; j++)
  if(*(int*)in->Items[i] == *(int*)inn->Items[j])
  {
   inn->Delete(j);
   inn->Insert(j, &a);
  }

kk = c2;
for(int i = 0; i < in->Count; i++)
 if(*(int*)inn->Items[(i+c2) % inn->Count] != -1)
 {
  in->Delete(kk % inn->Count);
  in->Insert(kk % inn->Count, inn->Items[(i+c2) % inn->Count]);
  kk++;
 }

 return in;
}
//---------------------------------------------------------------------------
TList* GenetinisAlg::KryzminimasTol(Sablonas* s1, Sablonas* s2)
{
 int kk, *aa;
 int minLen;

 in->Clear();
 inn->Clear();

 for(int i = 0; i < s1->Ispjauti->Count; i++)
 {
  aa = &((Staciakampis*)s1->Ispjauti->Items[i])->index;
  in->Add(aa);
 }
 for(int i = 0; i < s1->Neispjauti->Count; i++)
 {
  aa = &((Staciakampis*)s1->Neispjauti->Items[i])->index;
  in->Add(aa);
 }

 if((double)rand()/RAND_MAX < 1 - TikKryz)
  return in;

 for(int i = 0; i < s2->Ispjauti->Count; i++)
 {
```

```
 aa = &((Staciakampis*)s2->Ispjauti->Items[i])->index;
 inn->Add(aa);
}
for(int i = 0; i < s2->Neispjauti->Count; i++)
{
 aa = &((Staciakampis*)s2->Neispjauti->Items[i])->index;
 inn->Add(aa);
}

minLen = min(s1->Ispjauti->Count, s2->Ispjauti->Count);

for(int i = 0; i < minLen; i++)
 if((double)rand()/RAND_MAX <= 0.5)
 {
  kk = *(int*)inn->Items[i];
  for(int j = 0; j < in->Count; j++)
   if(*(int*)in->Items[j] == kk)
   {
    in->Delete(j);
    break;
   }
  in->Insert(i,inn->Items[i]);
 }
return in;
}
//---------------------------------------------------------------------------
TList* GenetinisAlg::Mutacija(TList* indeks)
{
 int k1, k2;
 for(int i = 0; i < indeks->Count; i++)
  if((float)rand()/RAND_MAX  < TikMut)
  {
   k1 = rand()%indeks->Count;
   k2 = rand()%indeks->Count;
   indeks->Exchange(k1,k2);
  }
 return indeks;
}
//---------------------------------------------------------------------------
#include "AtkaitinimoAlg.h"
//---------------------------------------------------------------------------

int __fastcall Kriteri(void * Item1, void * Item2)
{
  if (St(Item1)->ilgis > St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis > St(Item2)->plotis)
    return -1;
  else
  if (St(Item1)->ilgis < St(Item2)->ilgis ||
      St(Item1)->ilgis == St(Item2)->ilgis && St(Item1)->plotis < St(Item2)->plotis)
    return 1;
  else
    return 0;
}
//---------------------------------------------------------------------
PjaustymoPlanas* AtkaitinimoAlg::AtrinktiSA()
{
 Sablonas *sab, *best_sab, *kaim_sab;
 int eps;
 int *ind;
 Staciakampis * St;
 SablKiek = 0;
 while(Uzsakymas->Count != 0)
 {
  in->Clear();

 if(Atsitiktinis == 2)
 {
  for(int i = 0; i < Uzsakymas->Count; i++)
  {
   St = (Staciakampis*) Uzsakymas->Items[i];
   if(St->ilgis < St->plotis)
    St->Pasukti();
  }
  Uzsakymas->Sort(Kriteri);
 }

 for(int i = 0; i < Uzsakymas->Count; i++)
 {
```

```
 if(Atsitiktinis == 2)
 {
  ind = new int;
  *ind = i;
 }
 else
  ind = &((Staciakampis *)Uzsakymas->Items[i])->index;
 in->Add(ind);
}

 if(Atsitiktinis == 0)
 {
  for(int i = 0; i < in->Count; i++)
   in->Exchange(i,rand() % in->Count);
  if(ArBLF)
   sab = PjaustytiBLF(in);
  else
   sab = PjaustytiLG(in);
 }
 else
  if(Atsitiktinis == 1)
   sab = BF();
  else
   if(Atsitiktinis == 2)
    sab = PjaustytiBLF(in);

 best_sab = KopijuotiSab(sab);
 for(int i = MaxIter; i > 0; i--)
 {
  for(int j = 0; j < MaxTempIter; j++)
  {
   if(ArBLF)
    kaim_sab = PjaustytiBLF(Kaimynas(sab->Ispjauti, sab->Neispjauti,
                        (double)i/MaxIter));
   else
    kaim_sab = PjaustytiLG(Kaimynas(sab->Ispjauti, sab->Neispjauti,
                        (double)i/MaxIter));
    eps = kaim_sab->Atliekos - sab->Atliekos;

    if(kaim_sab->Atliekos - best_sab->Atliekos < 0)
    {
     delete best_sab;
     best_sab = KopijuotiSab(kaim_sab);

    if((double)rand()/RAND_MAX < expl((double)-1*eps/i))
    {
     delete sab;
     sab = KopijuotiSab(kaim_sab);
    }

    if(best_sab->Atliekos == 0)
    {
     i = 0; j = MaxTempIter;
    }
   }
   delete kaim_sab;
  }
 }
 delete(sab);
 best_sab->Panaudojimas = SablonoPanaudojimas(best_sab);
 if(best_sab->Panaudojimas != 0)
  pl->Sablonai->Add(best_sab);
 SablKiek += best_sab->Panaudojimas;
 if(SablKiek == SabKiek)
  break;
 }
 return pl;
}
//---------------------------------------------------------------------
TList* AtkaitinimoAlg::Kaimynas(TList* indeks, TList *neisp, double tik)
{
 int k1, k2;
 in->Clear();
 for(int i = 0; i < indeks->Count; i++)
  in->Add(&((Staciakampis*)indeks->Items[i])->index);
 for(int i = 0; i < neisp->Count; i++)
  in->Add(&((Staciakampis*)neisp->Items[i])->index);
 for(int i = 0; i < in->Count; i++)
  if((double)rand()/RAND_MAX  < tik)
```

```
  {
   k1 = rand() % in->Count;
   k2 = rand() % in->Count;
   in->Exchange(k1,k2);
  }
 return in;
}
//---------------------------------------------------------------------
```