KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS INŽINERIJOS KATEDRA

LINAR CHABIBULIN

DINAMINIŲ KELIO PAIEŠKOS ALGORITMŲ TYRIMAS

Magistro darbas

Darbo vadovas
lekt. dr. K. Jankauskas

KAUNAS, 2013

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS INŽINERIJOS KATEDRA

LINAR CHABIBULIN

DINAMINIŲ KELIO PAIEŠKOS ALGORITMŲ TYRIMAS

Magistro darbas

Darbo vadovas:
lekt. dr. K. Jankauskas
2013-05-24

Recenzentas:
doc. dr. T. Blažauskas
2013-05-24

Atliko:
IFM-1/1 gr. studentas
Linar Chabibulin
2013-05-24

KAUNAS, 2013

Dinaminių kelio paieškos algoritmų tyrimas

Santrauka

Dinaminiai kelio paieškos algoritmai apjungia euristinės ir plečiamos (*angl. incremental*) paieškos metodus, sprendžiant eiles panašių paieškos uždavinių tiek žinant visą informaciją apie aplinką, tiek neturint jokios informacijos. Yra trys plečiamą paiešką naudojančių algoritmų klasės. Šiame darbe pateikiama trumpa dinaminių kelio paieškos algoritmų, naudojančių plečiamos paieškos metodus analizė. Pagrindinis darbo tikslas – visų trijų plečiamos paieškos klasių algoritmų, kuriuos naudojant gaunamas optimalus kelio paieškos sprendinys aplinkose, kur perėjimų tarp viršūnių svoriai gali didėti ir mažėti, palyginimas. Algoritmai lyginami trijose skirtingose situacijose: stacionarioje, judėjimo link tikslo (*angl. goal – directed*) bei judančio taikinio (*angl. moving – target*). Tyrimo rezultatai parodė, jog A* ir FSA* nepasiekiamų viršūnių kiekiui esant ~16000 yra ~23,6% našesni už GAA* ir trečios plečiamos klasės algoritmus, o pasiekiamumą keičiančių viršūnių kiekiui esant ~8000 – 42,3%. Nepasiekiamų viršūnių kiekiui kintant nuo 1000 iki 16000 trečios plečiamos klasės algoritmai yra vidutiniškai ~58,7% našesni už GAA* ir ~54% našesni už A* ir FSA*. Pasiekiamumą keičiančių viršūnių kiekiui kintant nuo 500 iki 8000 trečios plečiamos klasės algoritmai yra vidutiniškai ~69,3% našesni už GAA* ir ~47,8% našesni už A* ir FSA*.

Raktiniai žodžiai

Dinaminiai kelio paieškos algoritmai, plečiama paieška, euristinė paieška, A*, Bendro planavimo A*, Lengvo planavimo D*, Pakraščių taupymo A*, adaptyvus A*.

Analysis of dynamic path finding algorithms

Summary

Dynamic path finding algorithms combine heuristic and incremental search methods to solve a series of similar search tasks in both known and unknown environments. There are three classes of incremental search algorithms. In this document we provide a brief summary of dynamic path finding algorithms, that uses incremental search methods, but its main focus is on comparing main algorithms of all three incremental classes, that are guaranteed to give optimal solution in environment where action costs can increase and decrease over time, and showing their strong and weak sides. The algorithms are compared in three different situations: stationary, goal – directed and moving – target. At the end of the document conclusions are given based on performed work. In this paper, research showed that A* and FSA* are ~23,6% more efficient than GAA* and third incremental class algorithms when the amount of untraversable cells is ~16000 and ~42,3% more efficient when the amount of traversability changing cells is ~8000. When the amount of untraversable cells is between 1000 and 16000, the third incremental class algorithms are ~58,7% more efficient than GAA* and ~54% – than A* and FSA*. When the amount of traversability changing cells is between 500 and 8000, the third incremental class algorithms are ~69,3% more efficient than GAA* and ~47,8% – than A* and FSA*.

Keywords

Dynamic path finding algorithms, incremental search, heuristic search, A*, Longlife planning A*, D* Lite, Fringe saving A*, Adaptive A*.

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

Terminų ir santrumpų žodynas

Euristinė funkcija	funkcija paremta apytiksliais įverčiais.
Statinė aplinka	aplinka, kurioje viršūnių pasiekiamumas laikui bėgant nesikeičia
Dinaminė aplinka	aplinka, kurioje viršūnių pasiekiamumas laikui bėgant keičiasi
Plečiama paieška	(*angl. incremental search*) paieška, dabar atliekamos paieškos pagreitinimui naudojantys prieš tai buvusių paieškų duomenis.
Autonominė sistema	savarankiška sistema, nereikalaujanti žmogaus įsikišimo.
FSA*	(*angl. Fringe – Saving A**) – pakraščių taupymo A*.
AA*	(*angl. Adaptive A**) – adaptyvus A*.
GAA*	(*angl. Generalized Adaptive A**) – apibendrintas adaptyvus A*.
LPA*	(*angl. LongLife planning A**) – bendro planavimo A*.
D*Lite	lengvo planavimo D*
ADA*	(*angl. Anytime dynamic A**) – bet kurio laiko dinaminis A* .
MT – D*Lite	(*angl. Moving target – D* Lite*) – Judančio taikinio – lengvo planavimo D*.
GFRA*	(*angl. Generalized Fringe – Retrieving A**) – apibendrintas pakraščių atstatymo A*.
PAA*	(*angl. Path – adaptive A**) – kelio – adaptyvus A*.
TAA*	(*angl. Tree adaptive A**) – medžio struktūrą naudojantis adaptyvus A*.
RTAA*	(*angl. Real – time adaptive A**) – realaus laiko adaptyvaus A*.
LRTA*	(*angl. Learning real time A**) – besimokantį realaus laiko A*.
RTD*	(*angl. Real – time D**) – Realaus laiko D*.
STL	(*angl. Standard template library*) **–** C++ biblioteka.

ĮVADAS

Problemos aktualumas

Viena iš dirbtinio intelekto sričių tiria žiniomis paremtus paieškos metodus, kurie gali spręsti paieškos uždavinius didelėse aplinkose. Didžioji šių metodų tyrimų dalis skirta spręsti vienetinėms (*angl. one-shot*) paieškoms. Tačiau paieška dažnai būna pasikartojantis procesas, kai reikia išspręsti eiles panašių paieškos uždavinių, nes, tikroji situacija yra truputį kitokia, negu buvo manyta, arba nes situacija keičiasi laikui bėgant. Pavyzdžiui, kelio planavimo metu besikeičiančios eismo sąlygos. Todėl, norint visada matyti trumpiausią kelią, pasikeitus eismo sąlygom, reikia perplanuoti turimą kelią. Tokiais atvejais dauguma paieškos metodu kelią perplanuoja nuo pradžių. Plečiamos paieškos metodai randa sprendimus eilėms panašių uždavinių daug greičiau, nei skaičiuojant kiekvieną uždavinį nuo pradžių. Be to, plečiamos paieškos metodų surasti sprendimai yra ne ilgesni, nei sprendžiant uždavinį atskirai.

Plečiamą paiešką naudojantys dinaminiai kelio paieškos algoritmai plačiai naudojami mobiliuosiuose robotuose, autonominių transporto priemonių navigacijos sistemose. Tokios navigacijos sistemos yra bandytos marsaeigiuose "Opportunity" ir "Spirit"[1].

Tyrimo sritis ir objektas

Dinaminiai kelio paieškos algoritmai sprendžia prielaidomis grįstas kelio planavimo problemas, įskaitant atvejus, kai roboto uždavinys yra pasiekti duotą tikslą nežinomoje aplinkoje. Apie aplinką yra padaromos prielaidos ir surandamas trumpiausias kelias tarp esamos pozicijos ir tikslo. Roboto judėjimui naudojamas surastas kelias. Kai aptinkama nauja informacija apie aplinką, ji yra pridedama prie turimos ir jei būtina – perskaičiuojamas trumpiausias kelias tarp esamos pozicijos bei tikslo. Procesas kartojamas kol pasiekiamas tikslas arba nustatoma, kad tikslas yra nepasiekiamas. Tikrinant aplinką kliūtys tikėtina bus aptinkamos dažnai, todėl kelio perskaičiavimas privalo būti greitas. Ne visų dinaminių algoritmų veikimas yra vienodai efektyvus. Be kliūčių pokyčių turi būti atsižvelgta į starto ir tikslo pozicijos (jeigu jos kinta).

Tyrimo tikslas ir uždaviniai

Tyrimo tikslas – ištirti pagrindinius dinaminius paieškos algoritmus, kuriuose gaunamas optimalus (ne ilgesnį negu A*) kelio paieškos sprendimas aplinkose, kur perėjimų tarp viršūnių svoriai gali didėti ir mažėti.

Tyrimo uždaviniai:

1. Pasirinkti dinaminius kelio paieškos algoritmus tyrimui.
2. Išskirti aplinkas, kuriose kiekvienas algoritmas turi pranašumą kitų algoritmų atžvilgiu.

1. PROBLEMINĖS SRITIES ANALIZĖ

1.1. Analizės tikslas

Išanalizuoti pagrindinius ir dažniausiai naudojamus dinaminius paieškos algoritmus, įvertinti jų privalumus ir trūkumus ir išsirinkti tinkamiausius tyrimui.

1.2. Analizės metodai

Kelio radimo algoritmai yra viena iš dirbtinio intelekto šakų ir didžiojoje dalyje knygų apie dirbtinį intelektą visa sritis apžvelgiama labai plačiai. Knygose apie žaidimų kūrimą kelio radimo algoritmai apžvelgiami daug plačiau, tačiau žaidimuose dinaminiai algoritmai nenaudojami, nes jie reikalauja daug resursų, todėl knygose nėra pakankamai informacijos. Tiesioginio priėjimo prie tokių sistemų, kur naudojami dinaminiai kelio paieškos algoritmus nėra, todėl pagrindinis ir vienintelis patikimas informacijos šaltinis yra moksliniai straipsniai.

1.3. Tyrimo objekto analizė

1.3.1. Euristinė paieška

Euristinė paieška – tai kelio paieškos technika, kuri naudoja euristinę funkciją, suteikiančią papildomą informaciją apie paieškos algoritmo vykdomą sekantį žingsnį. Euristinė funkcija – tai funkcija paremta apytiksliais įverčiais ir generuojanti apytikslę reikšmę. Todėl rezultato tikslumas gautas naudojant euristinę paiešką priklauso nuo euristinės funkcijos apsirašymo. Dauguma paieškos problemų yra eksponentinio sudėtingumo prigimties, todėl panaudojant euristines funkcijas galimas algoritmų sudėtingumo sumažinimas iki $O(n^c)$ sudėtingumo [2].

Euristinė funkcija kelio paieškose gali apimti informaciją apie būsenas, perėjimus tarp jų, tikslų charakteristikas ir t.t. Tačiau dažniausiai euristine funkcija bandoma įvertinti kelio svorį tarp nagrinėjamos ir tikslo viršūnių.

1.3.2. Plečiama paieška

Naudojant plečiamos paieškos algoritmus, dažniausiai rezultatas yra gaunamas greičiau, negu sprendžiant kiekvieną kelio paieškos uždavinį nuo pradžių, kadangi dabar atliekamos paieškos pagreitinimui naudojami prieš tai buvusių paieškų duomenis. Praeitų paieškų duomenys turi būti atnaujinami lokaliai tose vietose, kur viršūnių pasiekiamumas pasikeitė. Šie plečiamos paieškos algoritmai sutrumpina paieškos laiką ir pagreitina trumpiausio kelio radimą kiekvienoje sekančioje iteracijoje nenagrinėdami nepakitusių viršūnių. Yra trys plečiamos paieškos algoritmų klasės. Pirmos klasės paieškos algoritmuose A* paieška pradedama vykdyti nuo tos vietos, kur dabartinė A* paieška skiriasi nuo praėjusios. Antros klasės algoritmuose naudojamos pastovios, euristinės funkcijos generuojamos, reikšmės *h*. Viršūnės *h* reikšmė yra pastovi tikslo viršūnės atžvilgiu, jeigu ji tenkina tris sąlygas (2.1):

$$\begin{aligned} &H(s',s') = 0 \\ &H(s,s') < c(s,a) + H(succ(s,a),s') \\ &s \neq s' \end{aligned} \qquad (2.1)$$

čia $H(s,s)$ yra euristinės funkcijos generuojama reikšmė tarp dviejų nurodytų viršūnių, $c(s,a)$ yra svoris iš viršūnės s atliekant veiksmą a, $succ(s,a)$ nurodo viršūnę, į kurią bus nukeliauta iš viršūnės s atlikus veiksmą a. Trečios klasės algoritmuose atnaujinamos viršūnių *g* (svorio tarp starto ir nagrinėjamos viršūnės) reikšmės, tai lyg paieškos medžio transformacija iš praėjusios paieškos į dabartinės paieškos [3].

1.3.3. A* algoritmas

A* yra dažniausiai naudojamas kelio paieškos algoritmas, kuris randa optimalų ir pilną kelio paieškos uždavinio sprendinį (jeigu naudojama euristinė funkcija yra priimtina). A* algoritme naudojama geriausias – pirmas (*angl. best – first*) paieškos taktiką. A* algoritme kelio paieška susiaurinama naudojant tinkamą euristinę funkciją, kuri nukreipia paiešką link tikslo viršūnės. A* yra optimaliai našus, nes jame yra išnagrinėjamos visos viršūnės, kurių svoris yra mažesnis negu optimalaus kelio.

A* pseudo kodas:

```
Procedure ShortestPath
//in: s_start - starto viršūnė
//in: s_goal - tikslo viršūnė
// fn: AVS.Iterpti(s) - viršūnės s įterpimas į surikiuotą atvirų viršūnių sąrašą
// fn: AVS.Paimti() - iš atvirų viršūnių sąrašo grąžinama  bei ištrinama pirma viršūnė
//fn: UVS.Iterpti(s) - į uždarų viršūnių sąrašą įterpiama s viršūnė

begin
{01}  g(s_start) = 0; Parent(s_start) = NULL;
{02}  AVS = ø;
{03}  UVS = ø;
{04}  AVS.Iterpti(s_start);
{05}  while AVS ≠ ø do begin
{06}     s = AVS.Paimti();
{07}     UVS.Iterpti(s);
{08}     if (s == s_goal) then
{09}        return "kelias rastas";
{10}     else
{11}        forall s kaimynams s' do begin
{12}           if ((s'nėra AVS sąraše and s'nėra UVS sąraše) Or g(s)+1 < g(s')) then
{13}              g(s') = g(s) + 1;
{14}              tėvas(s') = s;
{15}              AVS.Iterpti(s');
{16}           endif;
{17}        endfor;
{18}     endif;
{19}  endwhile;
{20}  return "kelio nėra";
end.
```

A* viršūnė yra sudaryta iš keturių reikšmių: (1) *h* – euristinės funkcijos apskaičiuota apytikslė reikšmė, kuri nusako svorį tarp nagrinėjamos viršūnės bei tikslo viršūnės. (2) *g* – svoris tarp starto ir nagrinėjamos viršūnės. (3) *f* – suminis svoris tarp starto ir nagrinėjamos viršūnės bei euristinės funkcijos reikšmės. (4) *tėvas* – rodyklė į tėvo viršūnę paieškos medyje. A* turi du duomenų sąrašus, kuriuose talpinami duomenis apie viršūnių būsenas: atvirų viršūnių (toliau – *AVS*) ir uždarų viršūnių (toliau – *UVS*). *AVS* sąraše talpinamos visos viršūnės, kurios yra tinkamos nagrinėjimui, *UVS* sąraše talpinamos visos jau išnagrinėtos viršūnės.

Iš *AVS* sąrašo yra ištrinama viršūnė su mažiausia f reikšme (*žr. ShortestPath eil. 6*), įkeliama į *UVS* sąrašą (*žr. ShortestPath eil. 7*) ir jos visos kaimyninės viršūnės įkeliamos į AVS sąrašą (*žr. ShortestPath eil. 11-15*). Ši procedūra kartojama kol *AVS* sąraše nelieka viršūnių (*žr. ShortestPath eil. 20*) arba išnagrinėjama tikslo viršūnė (*žr. ShortestPath eil. 13*). Jeigu kelias egzistuoja, tai jis atsekamas einant per viršūnių tėvų rodykles pradedant nuo tikslo viršūnės [4].

A* yra našus sprendžiant pavienius paieškos uždavinius, bet tikėtina, kad nebus toks našus sprendžiant eiles panašių kelio paieškos uždavinių. Dinaminėse aplinkose kiekvieną kartą pasikeitus viršūnių pasiekiamumui algoritmas naują paieškos uždavinį turi spręsti nuo pradžių. Laiko atžvilgiu tai ne visada geras sprendimas.

1.3.4. Pakraščių taupymo A* algoritmas

Pakraščių taupymo A* (*angl. Fringe – Saving A*)* yra pirmos plečiamos klasės A* algoritmo versija, kuri pakartotinai suranda trumpiausią kelią nuo fiksuotos starto iki fiksuotos tikslo viršūnės, jeigu pasikeičia viršūnių pasiekiamumas. Atliekant naują paiešką yra naudojami prieš tai buvusios paieškos *AVS* ir *UVS* sąrašų duomenys. Ir A* paieška vykdoma ne nuo pradžių, o tik nuo to momento kai aptinkama viršūnė kurios pasiekiamumas skiriasi nuo prieš tai buvusioje paieškoje.

Pakraščių taupymo A* pseudo kodas:

```
Procedure UpdTravers
//in:Iteracija - nagrinėjamos iteracijos numeris
//in: UžBlokId - masyvas, kuriame talpinamos kiekvienos iteracijos m reikšmė
//local: LaikinUžBlokId -anksčiausiai išnagrinėtos viršūnės, kuri gali įtakoti naują paiešką nagrinėjimo eilės numeris .
// fn: ArPanaudojama(s) - grąžina ar viršūnė gali būti pakartotinai panaudota
// out: m - mažiausias išnagrinėjimo numeris viršūnės, kuri  nebus atrinkta pakartotinam panaudojimui

begin
{01}  LaikinUžblokId = ∞;
{02}  forall virsunes s, kurių pasiekiamumas pasikeitė do begin
{03}     if (s yra užblokuota) then
{04}        if (ArPanaudojama(s)) then
{05}           if (NagrinėjimoId(s)< LaikinUžblokId) then
{06}              LaikinUžblokId = NagrinėjimoId(s);
{07}           endif;
{08}        endif;
{09}     else
{10}        Parent(s) = NULL;
{11}        forall s kaimynams s' do begin
{12}           if (ArPanaudojama(s')) then
{13}              if (NagrinėjimoId(s') + 1 < LaikinUžblokId) then
{14}                 LaikinUžblokId = NagrinėjimoId(s') + 1;
{15}              endif;
{16}           endif;
{17}        endfor;
{18}     endif;
{19}  endfor;
{20}  for i = 1 to Iteracija do begin
{21}     if (LaikinUžblokId < UžblokId [i]) then
{22}        UžblokId [i] = LaikinUžblokId;
{23}     endif;
{24}  endfor;
{25}  m = UžblokId[Iteracija];
end.
```

```
Procedure RetrieveFringe
//in:Iteracija - nagrinėjamos iteracijos numeris
//in: s_start - starto viršūnė
//in: s_goal - tikslo viršūnė
// fn: AVS.Iterpti(s) - viršūnės s įterpimas į surikiuotą AVS sąrašą .
// fn: ArPanaudojama (s) - grąžina ar viršūnė gali būti pakartotinai panaudota
begin
{26}  AVS = ø;
{27}  Iteracija = Iteracija + 1;
{28}  s = s_goal;
{29}  while Not ArPanaudojama(tevas(s)) do begin
{30}     s = tevas(s)
{31}     if (s == s_start) then
{32}        return "kelio nėra";
{33}     endif;
{34}  endwhile;
{35}  Stumti s aplink sritį, kurioje yra visos viršūnės s' tenkinančios ArPanaudojama
   (s') sąlygą.
{36}     forall s kaimynams s', kurie tenkina ArPanaudojama(s') sąlygą do begin
{37}        GenIteracija (s') = Iteracija;
{38}        if (s yra neužblokuota) then
{39}           if (tevas(s) == NULL Or (Not ArPanaudojama(tevas(s)))) then
{40}              forall s kaimynams s' do begin
{41}                 if (ArPanaudojama(s')) then
{42}                    tevas(s) = s';
{43}                    g(s) = g(s') + 1;
{44}                    break;
{45}                 endif;
{46}              endfor;
{47}           endif;
{48}           GenIteracija(s) = Iteracija;
{49}           AVS.Iterpti(s);
{50}        endif;
{51}     endfor;
end.
```

Be reikšmių naudojamų A* algoritme, FSA* viršūnėje yra naudojamos papildomos trys reikšmės: (1) *GenIteracija* – iteracijos, kurioje buvo sugeneruota viršūnė numeris. (2) *NagrinėjimoIteracija* – iteracijos, kurioje buvo išnagrinėta viršūnė numeris. (3) *NagrinėjimoId* – kuriuo numeriu buvo išnagrinėta viršūnė.

FSA* algoritme tarp visų paieškų yra atliekamos dvi procedūros: *UpdTravers* (*žr*. *UpdTravers*) ir *RetrieveFringe* (*žr*. *RetrieveFringe*). Naudojant pirmąją procedūrą yra nustatoma reikšmė *m*, tokia kad sekančioje paieškoje būtų išnagrinėtos (būtų sudėtos į naują *UVS* sąrašą) bent visos viršūnės *s*, kurių *NagrinėjimoId(s)* yra mažiau už *m* reikšmę, tokia pačia tvarka kaip ir prieš tai buvusioje paieškoje. Jeigu pasikeitus pasiekiamumui viršūnė *s′* tapo pasiekiama, tai *m* bus lygi tos viršūnės išnagrinėjimo numeriui (2.2) Jeigu – neužblokuota, tai minimaliai tos viršūnės vaikų išnagrinėjimo reikšmei padidintai vienetu (2.3). Jeigu pasiekiamumą pakeitė daugiau negu viena viršūnė, tai *m* pasirenkama minimaliausia iš gautų m reikšmių.

$$m = NagrinejimoId(s') \tag{2.2}$$

$$m = 1 + \min_{s'' \in Succ(s')} NagrinejimoId(s'') \tag{2.3}$$

čia *succ*(*s*) nurodo viršūnes, į kurias galima nukeliauti iš viršūnės *s*.

Tada tikrinamas naujas *UVS* sąrašas. Jeigu jame yra tikslo viršūnė, tai paieška nevykdoma, nes trumpiausias kelias buvo surastas praeitoje paieškoje. Jeigu starto viršūnė yra pasiekiama, tai paieška nevykdoma, nes kelias neegzistuoja. Jeigu starto viršūnė pasiekiama, bet jos nėra atstatytame *UVS* sąraše, tai paieška vykdoma nuo pradžių. Naujame *AVS* sąraše talpinamos neužblokuotos viršūnės, kurios negali būti pakartotinai panaudojamos bei riboja naujame *UVS* sąraše esančias viršūnes. Visų pirma, einant viršūnių tėvų rodyklėmis nuo tikslo viršūnės surandama viršūnė, kuri iš pakartotinai panaudojamos tampa nepanaudojama bei riboja vieną iš viršūnių esančių naujame *UVS* sąraše (*žr. RetrieveFringe eil. 29-32*). Tada einant aplink naują *UVS* sąrašą ribojančią sritį randamos likusios naujam AVS sąrašui priklausančios viršūnės (*žr. RetrieveFringe eil. 35-49*) [3][5].

1.3.5. Adaptyvus A* algoritmas

Adaptyvus A* (*angl. Adaptive A*)* yra antros plečiamos klasės algoritmas, kuris atnaujina viršūnių euristines reikšmes, naudodamas turimą informaciją iš praeitų paieškų. Jis yra sukurtas spręsti tiek stacionarius, tiek judėjimo link tikslo, tiek judančio taikinio uždavinius, tačiau veikia tik situacijose, kai perėjimų tarp viršūnių svoriai tik didėja.

Adaptyvaus A* pseudo kodas:

```
Procedure InitializeState
//in: s - nagrinėjama viršūnė
//in: kelioIlgis(n) - n-tosios paieškos ilgis
//in: delta(n) - atstumas tarp pirmos ir n-tosios paieškos tikslo viršūnių
// out:s - nagrinėjama viršūnė

begin
{01}  if (paieška(s) ≠ counter and paieška(s) ≠ 0)
{02}     if (g(s) + h(s) < kelioIlgis(search(s)))
{03}        h(s) = kelioIlgis(paieška(s)) - g(s);
{04}     h(s) = h(s) - (delta(coutner) - delta(paieška(s)));
{05}     h(s) = max(h(s), H(s, s_goal));
{06}     g(s) = ∞;
{07}  elseif (paieška(s) = 0)
{08}     g(s) = ∞;
{09}     h(s) = H(s, s_goal);
{10}  endif;
{11}  paieška(s) = coutner;
end.
```

```
Procedure Main
//in: kelioIlgis(n) - n-tosios paieškosilgis
//in: delta(n) - atstumas tarp pirmos ir n-tosios paieškos tikslo viršūnių
//in: s_goal - tikslo viršūnė
//local: s_newgoal - nauja tikslo viršūnė

begin                    ...
{18}  s_newgoal reikšmei priskiriam naują tikslo viršūnę
{19}  if (s_goal ≠ s_newgoal)
{20}     InitializeState(s_newgoal);
{21}     if (g(s_newgoal) + h(s_newgoal) < kelioIlgis(counter))
{22}        h(s_newgoal) = kelioIlgis(counter) - g(s_newgoal);
{22}     endif;
{23}     delta(counter + 1) = delta(counter) + h(s_newgoal);
{24}     s_goal = s_newgoal;
{25}  else
{26}     delta(counter + 1) = delta(counter);
{27}  endif;
{28}  counter = coutner + 1;
{29}  pakiečiam viršūnių pasiekiamumą
end.
```

Adaptyvaus A* viršūnėje, be pagrindinių keturių reikšmių, yra naudojama pagalbinė paieškos reikšmė, kuri nusako iteraciją, kurioje paskutinį kartą buvo panaudota viršūnė.

Viršūnėms tapus nepasiekiamomis, jų *h* reikšmės išlieka pastovios, tačiau joms tampant pasiekiamoms, jų *h* reikšmės ne būtinai išliks pastovios, todėl jos turi būti atnaujintos. Algoritme *h* reikšmės atnaujinamos tik tada, kai jos yra reikalingos paieškos metu, inicijuojant viršūnę (*žr. InitializeState*), taip taupant resursus. Viršūnės iniciavimui naudojamos paieška, *counter* – dabartinės paieškos numeris, *delta* – visų tikslo viršūnių pokyčių suma, ir *kelioIligs(x)* – x-osios paieškos rastas kelio ilgis. Tikslo viršūnės pasikeitimas įtakoja visų kitų viršūnių *h* reikšmes ir skaitinė reikšmė atspindinti tą pokytį talpinama delta kintamajame. Jis perskaičiuojamas po kelio radimo, kaskart pasikeitus tikslo viršūnei (*žr. Main eil. 26/28*) [6].

1.3.6. Apibendrintas adaptyvus A* algoritmas

Apibendrintas adaptyvus A* (*angl. Generalized Adaptive A*)* yra adaptyvaus A* algoritmo apibendrinta versija, kur viršūnių h reikšmės perskaičiuojamos po kiekvienos paieškos naudojant *consistency* procedūrą (*žr. Consistency*). Tai yra vienas iš algoritmų sukurtas spręsti judančio taikinio problemą (kai keičiasi ir startas, ir tikslas).

Kadangi *h* reikšmėms atnaujinti naudojama tam skirta procedūra, tai iš viršūnės iniciavimo funkcijos yra pašalinami visi veiksmai, kurių metu yra atnaujinama *h* reikšmė. *Consistency* procedūra patikrina visas viršūnes, kurios tampa pasiekiamos ir atnaujina jų *h* reikšmes, kad jos tenkintų pastovumo sąlygą [7].

Apibendrinančio adaptyvaus A* *Consistency* procedūros pseudo kodas:

```
Procedure Consistency
//in: s_goal - tikslo viršūnė
// fn: InitState(s) - sukuriama viršūnė s, jeigu ji jau yra sukurta, tai paima ir grąžina turimą
// fn: AVS.Ištrinti(s) - iš AVS sąrašo ištrinama viršūnė s
// fn: AVS.Iterpti(s) - viršūnės s įterpimas į surikiuotą AVS sąrašą
// fn: AVS.Paimti() - iš AVS grąžinama bei ištrinama pirma viršūnė

begin
{01}  AVS = ø;
{02}  forall perėjimams iš viršūnės s į s', kurių perėjimo svoris sumažėjo do begin
{03}     InitState(s);
{04}     InitState(s');
{05}     if (h(s) > h('s) + 1) then
{06}        h(s) = h('s) + 1;
{07}        if (s yra Open sąraše) then
{08}           AVS.Ištrinti(s);
{09}        endif;
{10}        AVS.Iterpti(s);
{11}     endif;
{12}  endfor;
{13}  while AVS ≠ ø do begin
{14}     s' = AVS.Paimti();
{15}     forall s ≠ s_goal and s kaimynams s'
{16}        InitState(s);
{17}        if (h(s) > h('s) + 1) then
{18}           h(s) = h('s) + 1;
{19}           if (s yra AVS sąraše) then
{20}              AVS.Ištrinti(s);
{21}           endif;
{22}           AVS.Iterpti(s);
{23}        endif
{24}     endfor;
end.
```

1.3.7. Bendro planavimo A* algoritmas

Bendro planavimo A* (*angl. LongLife planning A*)* yra trečios plečiamos klasės A* algoritmo versija.

Ilgalaikio planavimo A* pseudo kodas:

```
Procedure LPACalculateKey
//in:s -nagrinėjama  viršūnė
//in:s_goal - tikslo viršūnė
// out: f(s) ir g(s)

begin
{01}  return[min(g(s), rhs(s))+h(s,s_goal); min(g(s), rhs(s))];
end.
```

```
Procedure LPAUpdateVertex
//in:s -nagrinėjama  viršūnė
//in:s_start - starto viršūnė
//fn: AVS.Iterpti(s, f) - viršūnės s įterpimas į AVS  su rakto reikšme  f
// fn:AVS.Ištrinti(s) - ištrina viršūnę s iš AVS sąrašo.
begin
{02}  if (s ≠ s_start) then
{03}     rhs(s) = min_(s kaimynų s')(g(s') + 1);
{04}  endif;
{05}  if (s yra AVS sąraše) then
{06}     AVS.Ištrinti(s);
{07}  endif;
{08}        endif;
{09}  if (g(s) ≠ rhs(s)) then
{10}     AVS.Iterpti(s, CalculateKey(s));
{11}  endif;
end.
```

```
Procedure LPAShortestPath
//in:s_goal - tikslo viršūnė
//in:s_start - starto viršūnė
// fn: AVS.MazPirmum() - grąžina mažiausią  pirmumo rodiklį
// fn:AVS.Paimti() - iš AVS grąžinama bei ištrinama pirma viršūnė

begin
{12}  while AVS.MazPirmum() < CalculateKey(s_goal) Or rhs(s_goal) ≠ g(s_goal)  do begin
{13}     s = AVS.Paimti();
{14}     if (g(s) > rhs(s)) then
{15}        g(s) = rhs(s);
{16}        forall s kaimynams s' do begin
{17}           UpdateVertex(s');
{18}        endfor;
{19}     else
{20}        g(s) = ∞;
{21}        forall s ir s kaimynams s' do begin
{22}           UpdateVertex(s');
{23}        endfor;
{24}     endif;
end.
```

Bendro planavimo A* algoritmo viršūnėje naudojamos tokios pat reikšmės kaip ir A* algoritmo(h reikšmė turi būti pastovi) bei papildoma *rhs* – reikšmė, kuri yra labiau informuotas, g reikšme paremtas atstumo nuo starto įvertis (pvz. 2.4).

$$rhs(s) = 0, \text{ kai } s = s_{start} \qquad (2.4)$$

$$rhs(s) = \min_{(s\_kaimynams\_s')}(g(s') + 1)$$

Jeigu viršūnės *g* ir *rhs* reikšmės sutampa, tai viršūnė yra pastovi. Jeigu visos viršūnės yra pastovios, tai galima atsekti trumpiausią kelią nuo starto iki tikslo viršūnės einant nuo tikslo viršūnės į mažiausią g reikšmę turinčią tėvinę viršūnę. Tačiau pasikeitus svoriams tarp viršūnių patikslinamos tik tam tikros viršūnės, kurios tampa nepastovios. Jos atrenkamos naudojant euristinę funkciją. *AVS* sąraše talpinamos tik nenuoseklios viršūnės, t.y. tos viršūnės, kurias reikia koreguoti. Iš *AVS* sąrašo yra paimamos ir nagrinėjamos viršūnės turinčios mažiausią rakto reikšmę (*žr. LPAShortestPath eil. 13-24*). Viršūnės rakto reikšmė nusakoma viršūnės vektoriaus $k(s) = [k_1(s); k_2(s)]$ (*žr. LPACalculateKey eil. 01).* Pradinėje iniciavime visų viršūnių *g* ir *rhs* reikšmėms priskiriama begalybė, o starto *g* reikšmėms 0, taip užtikrinama, kad starto viršūnė bus nepastovios ir pirmoji paieškos iteracija prasidės nuo nurodytos starto viršūnės kaip paprasta A* paieška. Nepastovios viršūnės koreguojamos dviejuose algoritmo vietose: tarp kelio paieškų bei kelio paieškos metu. Tarp kelio paieškų, pasikeitus aplinkai išrenkamos ir pakoreguojamos nepastovios viršūnės bei jų raktų reikšmės (*žr. LPAUpdateVertex*). Kelio paieškos metu gali būti aptiktos dviejų rūšių nepastovios viršūnės: lokaliai daugiau pastovios (2.5) arba lokaliai mažiau pastovios (2.6). Pirmuoju atveju *g* reikšmei priskiriama *rhs* reikšmė, tada ji tampa pastovi (*žr. LPAShortestPath eil. 15*). Antruoju atveju g reikšmei priskiriama begalybė, tada ji tampa arba pastovi arba lokaliai daugiau pastovi (*žr. LPAShortestPath eil. 20*). Bet kokiu atveju pakoregavus nepastovias viršūnes, korekcijos turi būti atliktos ir jų kaimyninėms viršūnėms, kadangi jų nepastovumas gali įtakoti ir kaimynines viršūnes [8].

$$g(s) > rhs(s) \qquad (2.5)$$

$$g(s) < rhs(s) \qquad (2.6)$$

1.3.8. D* algoritmas

D* – dinaminė A* algoritmo versija. Pirmoji D* paieška yra tokia pati kaip ir A* algoritme. Pasikeitus aplinkai yra perskaičiuojamas kelias panaudojant kai kurias anksčiau sugeneruoto paieškos medžio dalis, medį pakoreguojant tik tose vietose, kur viršūnių pasiekiamumas pasikeitė.

D* viršūnės turi tokias pat reikšmes kaip ir A* bei taip pat naudoja du sąrašus: *AVS* ir *UVS*. D* nuo A* skiriasi tuo, kad kelio paieška pradedama vykdyti ne nuo starto viršūnės, bet nuo tikslo. Kai viršūnių pasiekiamumas pasikeičia surastas kelias naudojamas kaip pradinis ir kai aptinkama pasiekiama viršūnė arba sekanti viršūnė yra kelio pabaiga, tai kaip starto viršūnė nustatoma

paskutinė nagrinėta viršūnė ir perskaičiuojamos visos viršūnių esančių tarp naujos starto ir tikslo reikšmės. Naujos reikšmės palyginamos su senomis. Jeigu naujas suminis svoris yra mažesnis už seną, tai viršūnės būsena pakeičiame į *žemesnė* ir pirmumas vėlesniuose skaičiavimuose duodamas naujoms reikšmėms, priešingu atveju į *didesnė* [9].

1.3.9. Lengvo planavimo D* algoritmas

Lengvo planavimo D* (*angl. D*Lite*) algoritmas yra išvestas iš LPA* algoritmo ir yra skirtas situacijoms, kai paieškų metu kinta starto viršūnės pozicija. Jis yra žymiai trumpesnis lyginant su D* algoritmu, jis naudoja tik vieną kriterijų viršūnių pirmumo lyginimo metu. Taip pat nedaro jokių prielaidų apie viršūnių pasiekiamumo kitimą, ar viršūnės užblokuojamos ar atblokuojamos, ar jos keičiasi arti starto ar toli. Lengvo planavimo D* paieška vykdoma nuo tikslo link starto viršūnės.

Lengvo planavimo D* pseudo kodas:

```
Procedure DLiteCalculateKey
//in:s -nagrinėjama  viršūnė
//in:s_goal - tikslo viršūnė
// out: f(s) ir g(s)

begin
{01}  return[min(g(s), rhs(s))+h(s,s_goal)+k_m; min(g(s), rhs(s))];
end.
```

```
Procedure DLiteShortestPath
//in:s_goal - tikslo viršūnė
//in:s_start - starto viršūnė
//local: k_old - mažiausia pirmumo eilės rakto reikšmė
// fn: AVS.MazPirmum() - grąžina mažiausią  pirmumo rodiklį
// fn:AVS.Paimti() - iš AVS grąžinama bei ištrinama pirma viršūnė
// fn: AVS.Iterpti(s, f) - viršūnės s įterpimas į AVS  su rakto reikšme  f

begin
{02}  while AVS.MazPirmum() < CalculateKey(s_start) Or rhs(s_start) ≠ g(s_start)  do begin
{03}     k_old = AVS. MazPirmum();
{04}     s = AVS.Paimti();
{05}     if (k_old < CalculateKey(s))
{06}        AVS.Iterpti(s, CalculateKey(s))
{07}     else if (g(s) > rhs(s)) then
{08}        g(s) = rhs(s);
{09}        forall s kaimynams s' do begin
{10}           UpdateVertex(s');
{11}        endfor;
{12}     else
{13}        g(s) = ∞;
{14}        forall s ir s kaimynams s' do begin
{15}           UpdateVertex(s');
{16}        endfor;
{17}     endif;
end.
```

Starto viršūnei pakeitus poziciją, reikėtų perskaičiuoti visų viršūnių esančių pirmumo eilėje raktų reikšmes (norint jas padaryti nuosekliomis), bet tai gali užimti daug laiko, jeigu pirmumo eilėje yra daug viršūnių. Kadangi tik pirmoji iš dviejų rakto reikšmių priklauso nuo euristinės funkcijos ir startui artėjant prie tikslo ji mažėja, tai norint išlaikyti viršūnių išsidėstymo tvarką pirmumo eilėje nepakitusią reikia pokytį pridėti pakitusioms viršūnėms. Tam naudojamas naujas kintamasis k_m – visų pokyčių suma, naudojamas skaičiuojant viršūnių rakto reikšmes (*žr. DLiteCalculateKey*) [10].

1.3.10. Judančio taikinio – lengvo planavimo D* algoritmas

Judančio taikinio – lengvo planavimo D* (*angl. Moving target – D* Lite*) skirtas atvejams kai keičiasi ne tik aplinka, bet ir abi – starto ir tikslo viršūnės. MT – D*Lite yra lengvo planavimo D*Lite algoritmo praplėtimas, naudojantis apibendrinto pakraščių atstatymo A* (*angl. Generalized Fringe – Retrieving A**) [11] veikimo principą pakartotinai perskaičiuojant trumpiausią kelią. D*Lite gali būti pritaikytas tokiems atvejams kiekvieną kartą pastumiant aplinką priklausomai nuo starto

pozicijos taip, kad ji liktų stacionari (jei startas pajuda vienu vienetu į viršų, tai aplinka pastumiama vienetu žemyn), tačiau tokiu atveju yra prarandama didžioji praeitose paieškose gauta informacijos dalis. Tokiu atveju D*Lite algoritmas gali veikti lėčiau negu A*. Apibendrintas pakraščių atstatymo A* algoritmas gali spręsti judančio taikinio uždavinius, bet tik statinėse aplinkose. MT – D*Lite aplinką šiek tiek pakoreguoja, bet jos pozicijos nekeičia.

Judančio taikinio – lengvo planavimo D*pseudo kodas:

```
Procedure Deletion
//local: IVS- sąrašas, kuriame talpinamos visos praeitos paieškos kelio medžio viršūnės, kurios nėra poaibis medžio, kurio šaknis s_start
// fn: AVS.MazPirmum() - grąžina mažiausią  pirmumo rodiklį
// fn:AVS.Ištrinti(s) - ištrina viršūnę s iš AVS sąrašo.
// fn: AVS.Iterpti(s, f) - viršūnės s įterpimas į AVS  su rakto reikšme  f

begin
{01}  IVS = ø;
{02}  parent(s_start)= NULL;
{03}  forall s priklauso paieškos medžiui, bet ne poaibis medžio, kurio šaknis yra s_start do begin
{04}     parent(s)= NULL;
{05}     g(s) = ∞;
{06}     rhs(s) = ∞;
{07}     if (s yra AVS sąraše) then
{08}        AVS.Ištrinti(s);
{09}     endif;
{10}  endfor;
{11}  forall s yra IVS sąraše do begin
{12}     forall s kaimynams s' do begin
{13}        if (rhs(s) > g(s') + 1) then
{14}           rhs(s) > g(s') + 1;
{15}           parent(s) = s';
{16}        endif;
{17}     endfor;
{18}     if (rhs(s) < ∞) then
{19}        AVS.Iterpti(s, CalculateKey(s));
{20}     endif;
{21}  endfor;
end.
```

MT – D*Lite algoritme, kaip ir GFRA* algoritme naudojamas papildomas ištrintų viršūnių sąrašas (toliau *IVS*), kuriame talpinamos viso kuriame talpinamos visos praeitos paieškos kelio medžio viršūnės, kurios nėra poaibis medžio, kurio šaknis s_{start}. Iš pirmo, s_{start} tėvo viršūnei priskiriama *NULL* reikšmė (*žr. Deletion eil. 2*). Tada visų viršūnių, esančių *IVS* sąraše *g* ir *h* reikšmėms priskiriama begalybė, o *tėvo* reikšmei – *NULL* bei jeigu jos yra *AVS* sąraše, jos yra ištrinamos (*žr. Deletion eil. 3 – 10*). Paskutinis žingsnis yra visų viršūnių, kurios gali būti įtakotos anksčiau atliktų veiksmų, *rhs* bei *tėvo* reikšmių atnaujinimas (*žr. Deletion eil. 11 – 21*) [12].

1.3.11. Kelio – adaptyvus A* algoritmas

Kelio – adaptyvus A* (*angl. Path – adaptive A**) naudoja adaptyvų A* algoritmą kelio radimui aplinkose, kai pradžioje nėra turima jokios informacijos apie galimas kliūtis ir daroma prielaida, kad kliūčių nėra (*angl. free space assumption*). Algoritmas pakartotinai panaudoja rasto kelio pradžią, kuri išlieka nepakitusi pasikeitus viršūnių pasiekiamumui. Kas kartą yra randamas trumpiausias kelias bei juo einama kol aptinkama, kad perėjimo svoris į vieną iš viršūnių, įeinančių į trumpiausią kelią, padidėja. Tada perskaičiuojamas kelias nuo turimos viršūnės iki kelių sekančių nepakitusių trumpiausio kelio viršūnių naudojant pateiktą euristinę funkciją bei tęsiamas keliavimas trumpiausiu keliu per viršūnę su mažiausia euristine reikšme[13].

1.3.12. Bet kurio laiko dinaminis A* algoritmas

Bet kurio laiko dinaminis A* (*angl. Anytime dynamic A**) yra euristinę paiešką bei bet kurio laiko (*angl. anytime*) planavimą naudojantis algoritmas. Jis buvo sukurtas apjungiant lengvo planavimo D* [8] bei ARA* [14] algoritmus. Iš pirmo yra surandamas egzistuojantis, tačiau mažai tikėtina trumpiausias kelias. Apjungiant anksčiau minėtus du metodus nuolatos yra atnaujinamas rastas kelias mažinant infliacijos faktorius, taip apskaičiuojant naujus kelius su tikslesnėmis ribomis, kol nepraeina iš anksto skaičiavimams duotas laiko intervalas. Kai aplinkoje įvyksta pokyčiai

įtakojantys perėjimų tarp viršūnių svorius, tos viršūnės įterpiamos į *AVS* sąrašą naudojant D*Lite algoritme naudojamą pirmumo raktą. *AVS* sąrašo viršūnės nagrinėjamos, kol kelias pasiekia nurodytą optimalumą [15].

1.3.13. Medžio struktūra naudojantis adaptyvus A* algoritmas

Medžio struktūrą naudojantis adaptyvus A* (*angl. Tree adaptive A**) yra AA* algoritmo apibendrintas variantas, kuris yra skirtas spręsti judėjimo link tikslo uždaviniams aplinkose, apie kurias neturima jokios informacijos, daroma prielaida, kad kliūčių nėra bei kai perėjimų tarp viršūnių svoriai gali tik didėti. Kitaip negu Kelio PAA* algoritmas, TAA* pakartotinai panaudoja visų anksčiau rastų kelių pradžias. Algoritmas apjungia antros ir trečios klasės plečiamų euristinių klasių veikimo principus. Algoritmo pakartotinai panaudojamas paieškos medis yra panašus į trečios klasės plečiamų euristinių algoritmų (kaip D*Lite) naudojamą paieškos medį, kuris yra gaunamas atliekant atbulinę A* paiešką, užtikrinant paieškos medžio pradžios pastovumą. TAA* algoritme yra atliekama tiesioginė A* paieška [16].

1.3.14. Realaus laiko adaptyvus A* algoritmas

Realaus laiko adaptyvaus A* (*angl. Real – time adaptive A**) algoritme naudojamas bet kurio laiko metodas, kuriuo galima pasirinkti lokalią paieškos sritį ir greitai atnaujinti joje esančių viršūnių euristines reikšme. RTAA* yra panašus į LRTA* [17]. Jie skiriasi tik euristinių reikšmių atnaujinimo metodu. LRTA* kiekvienos išnagrinėtos viršūnės euristinę reikšmę pakeičia atstumo nuo viršūnės iki sugeneruotos, bet dar neišnagrinėtos viršūnės s ir viršūnės s euristinės reikšmės minimalia suma, tarp visų sugeneruotų, bet dar neišnagrinėtų viršūnių. RTAA* euristinei reikšmei priskiriama viršūnės, kuri turėjo būti nagrinėjama, kai buvo nutraukta paieška, f reikšmės bei dabar nagrinėjamos viršūnės g reikšmės skirtumas [18].

1.3.15. Realaus laiko D* algoritmas

Realaus laiko D* (*angl. Real – time D**) skiriasi nuo kitų realaus laiko algoritmų, nes yra atliekama paprasta dinaminė globali paieška, kartais ją pertraukiant, kad perskaičiuoti starto poziciją, atsižvelgiant į realaus laiko paiešką. Globali paieška leidžia įvertinti viršūnių pasiekiamumų pokyčius už lokalios paieškos srities, taip randant kelio nukirtimus. Apjungiant abi paieškas visada yra gražinamas rezultatas, net jeigu pilnas kelias nebuvo rastas. Dinaminiai paieškai atlikti gali būti naudojamas D*Lite arba ADA* algoritmas, o realaus laiko paieškai – LRTA* arba LSS – LRTA* [19]. RTD* yra dvikryptis paieškos algoritmas, nes dinaminė globali paieška atliekama nuo tikslo, bandant pasiekti starto viršūnę, o realaus laiko lokali paieška atliekama nuo starto, bandant pasiekti tikslo viršūnę [20].

1.4. Pasirinktų algoritmų pagrindimas

Dinaminių kelio paieškos algoritmų tyrimui buvo pasirinkti šeši algoritmai: A*, FSA*, GAA*, LPA*, D*Lite ir MT – D*Lite. A* algoritmas pasirinktas, nes tai yra dažniausiai naudojamas kelio paieškos algoritmas bei jį modifikuojant buvo gaunami visi kiti šiame tyrime nagrinėjami algoritmai. FSA* yra vienintelis pirmos plečiamos klasės algoritmas. GAA* yra vienintelis antros plečiamos klasės algoritmas, kuris gali rasti optimalų kelią aplinkoje, kur perėjimų tarp viršūnių svoriai gali didėti ir mažėti. Kitų antros plečiamos klasės algoritmų (AA*, PAA*, TAA*) tokioje aplinkoje rastas kelias nebus optimalus, nes juose nevykdomi jokie pataisymai, kai viršūnė tampa pasiekiama (kai perėjimo į viršūnę svoris sumažėja) ir todėl algoritmas negali aptikti atsiradusių kelio sutrumpinimų. LPA*, D*Lite ir MT – D*Lite yra trečios plečiamos klasės algoritmai ir yra skirti atitinkamai stacionarioje, judėjimo link tikslo bei judančio taikinio situacijose. D* nėra įtrauktas į tiriamų algoritmų sąrašą, nes yra įrodyta, kad lengvo planavimo D* algoritmas visais atvejais veikia ne lėčiau negu D*Lite. Taip pat į tiriamų algoritmų sąrašą nėra įraukti ADA*, RTAA* ir RTD* algoritmai, nes jie yra skirti spręsti kelio paieškos uždavinius, kai yra duotas laiko intervalas per kurį turi būti randamas sprendinys, todėl šių algoritmų randami keliai ne visada būna optimalūs.

2. PROJEKTINĖ DALIS

Šiame skyriuje aprašomi pasirinktų algoritmų žingsnių realizacija.

2.1. Duomenų talpinimo struktūros

AVS, bl_array, unbl_array sąrašai buvo realizuoti naudojant *STL vector* konteinerį. *AVS* buvo realizuotas kaip rikiuotas sąrašas (pagal *f* reikšmes), kuriame buvo talpinamos viršūnių x ir y koordinatės.

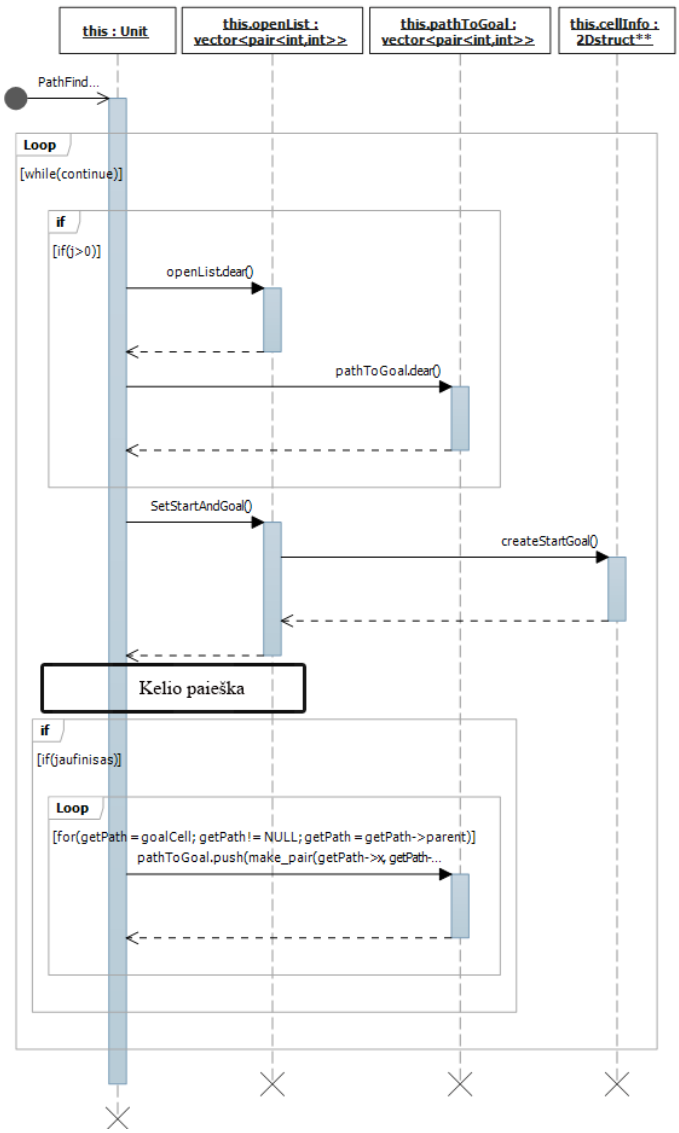
Viršūnės informacija (*g, h, f* reikšmės, rodyklė į *tėvo* viršūnę ir t.t.) talpinama struktūriniame elemente. Visi viršūnių struktūriniai elementai talpinamai aplinkos dydžio matricoje, kur pirmas indeksas nurodo x koordinatę, o antras – y koordinatę.

UVS sąrašo atsisakyta ir informacija apie viršūnių būseną (atvira, uždara, nesugeneruota) talpinama viršūnės struktūriniame elemente.

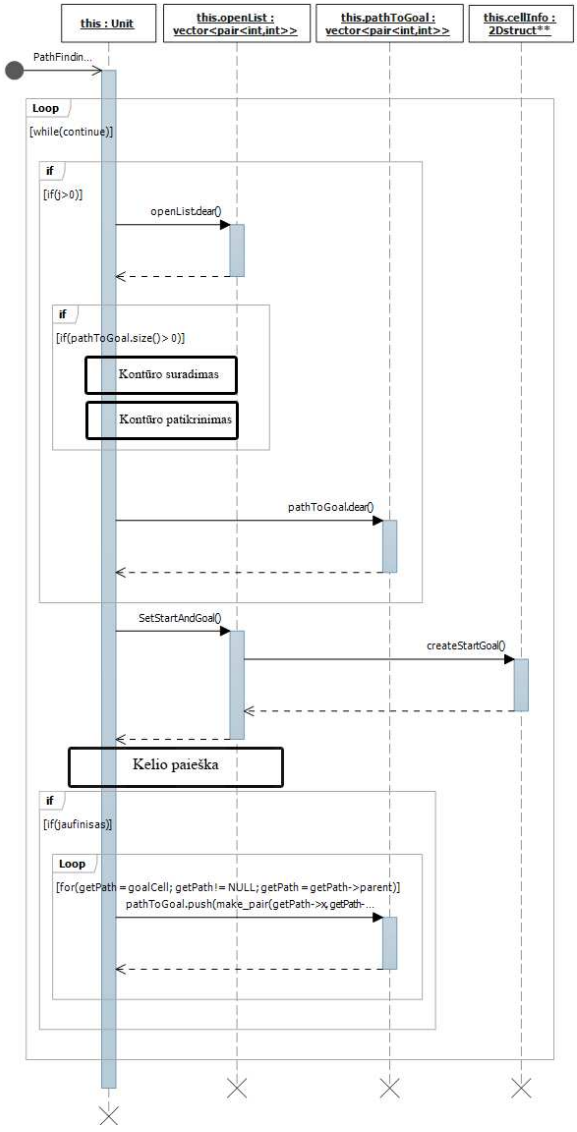
2.2. A* ir FSA* algoritmų žingsnių realizacijos detalizavimas

A* algoritmo realizacijos apibendrinta sekų diagrama pavaizduota 2.1 pav., FSA* – 2.2 pav. Paryškinti sekų diagramų žingsniai bus aprašyti poskyriuose.

While ciklas yra vykdomas, kol nėra pasiekta 100 skaičiavimų statinėje situacijoje arba kol starto viršūnė nelygi tikslo viršūnei judėjimo link tikslo ir judančio taikinio situacijose. *openList.clear()* ir *pathToGoal.clear()* kreipiniai atitinkamai išvalo *AVS* ir kelią talpinantį sąrašus (šios operacijos atliekamos visose išskyrus pirmąją kelio paiešką). *setStartAndGoal()* – sukuria starto ir tikslo viršūnes ir į *AVS* sąrašą įdeda starto viršūnės koordinates. Paskutiniu *if* sakiniu tikrinama ar yra pasiekta tikslo viršūnė. Paskutiniu *for* ciklu yra atsekamas kelias einant į nagrinėjamos viršūnės *tėvo* viršūnę pradedant nuo tikslo ir jas įrašant į *pathToGoal* sąrašą. FSA* algoritmo dalyje esančioje po *AVS* sąrašo išvalymo yra tikrinama ar praeitoje paieškoje buvo surastas kelias. Jeigu kelias buvo surastas, tai vykdomi *Kontūro suradimo* ir *Kontūro patikrinimo* žingsniai.



2.1 pav. A* realizacijos apibendrinta sekų diagrama



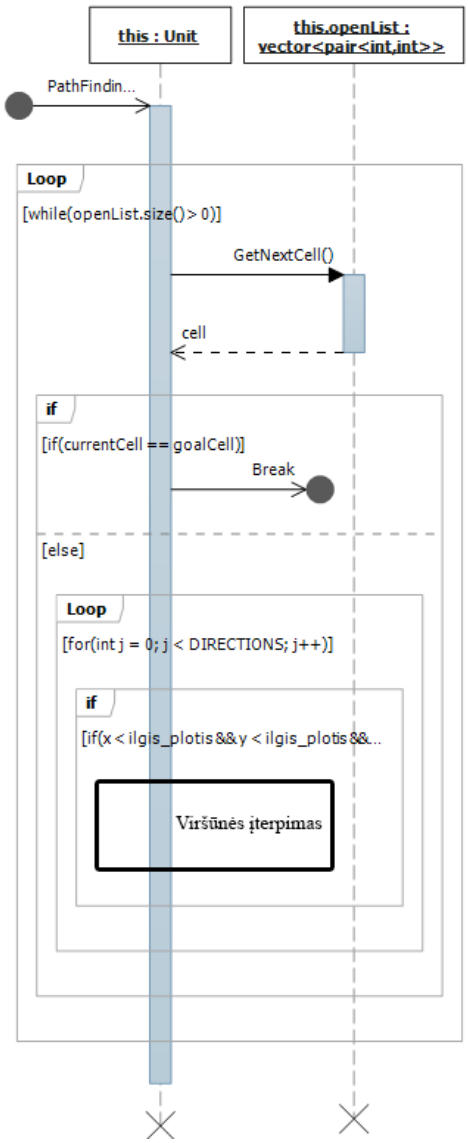
2.2 pav. FSA* realizacijos apibendrinta sekų diagrama

2.2.1. Kelio paieškos ir viršūnės įterpimo žingsniai

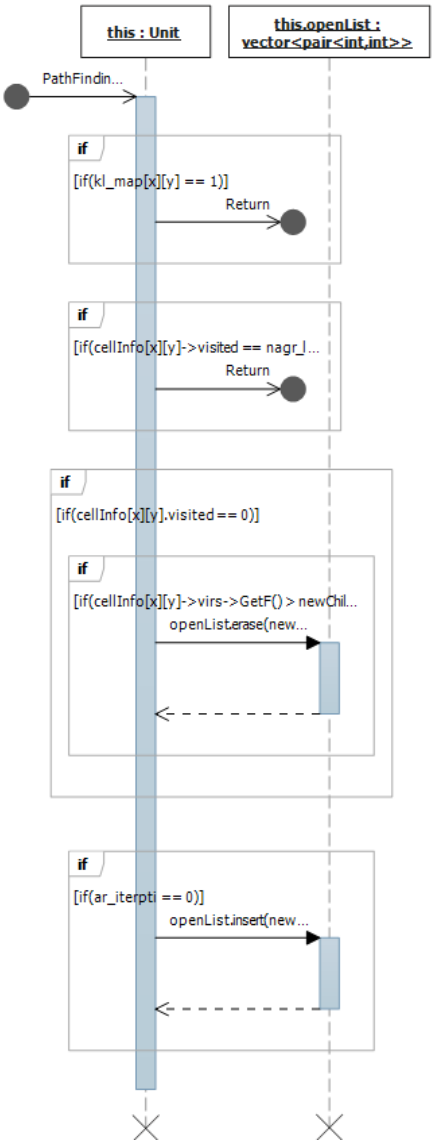
Kelio paieškos žingsnyje (žr. **Error! Reference source not found.**) yra vykdomas ciklas kol *AVS* sąrašas nėra tuščias. Jo metu iš AVS sąrašo yra paimama pirmoji viršūnė ir patikrinama ar tai nėra tikslo viršūnė. Jeigu tai yra tikslo viršūnė, tai ciklas yra stabdomas, jeigu ne, tai yra apeinamos visos kaimyninės viršūnės, prieš tai patikrinant ar jos nėra už aplinkos ribų.

Viršūnės įterpimo žingsnyje (žr. 2.4 pav.) tikrinama ar viršūnė turėtų būti įterpta į *AVS* sąrašą. Šioje dalyje esantys *if* sakiniai atlieka tokius veiksmus (aprašomi eilės tvarka):

- Tikrinama ar viršūnė yra nepasiekiama, jeigu taip, tai pereinama prie sekančios kaimyninės viršūnės.
- Tikrinama ar viršūnė buvo aplankyta dabartinėje paieškoje, jeigu taip, tai pereinama prie sekančios kaimyninės viršūnės.
- Tikrinama ar viršūnė yra *AVS* sąraše.
- Tikrinama ar viršūnės *f* reikšmė yra mažesnė už *AVS* sąraše esančios viršūnės, jeigu taip, tai viršūnė yra trinama iš *AVS* sąrašo.
- Tikrinama ar viršūnė turi būti įterpta į *AVS* sąrašą.



2.3 pav. *Kelio paieškos* dalies sekų diagrama



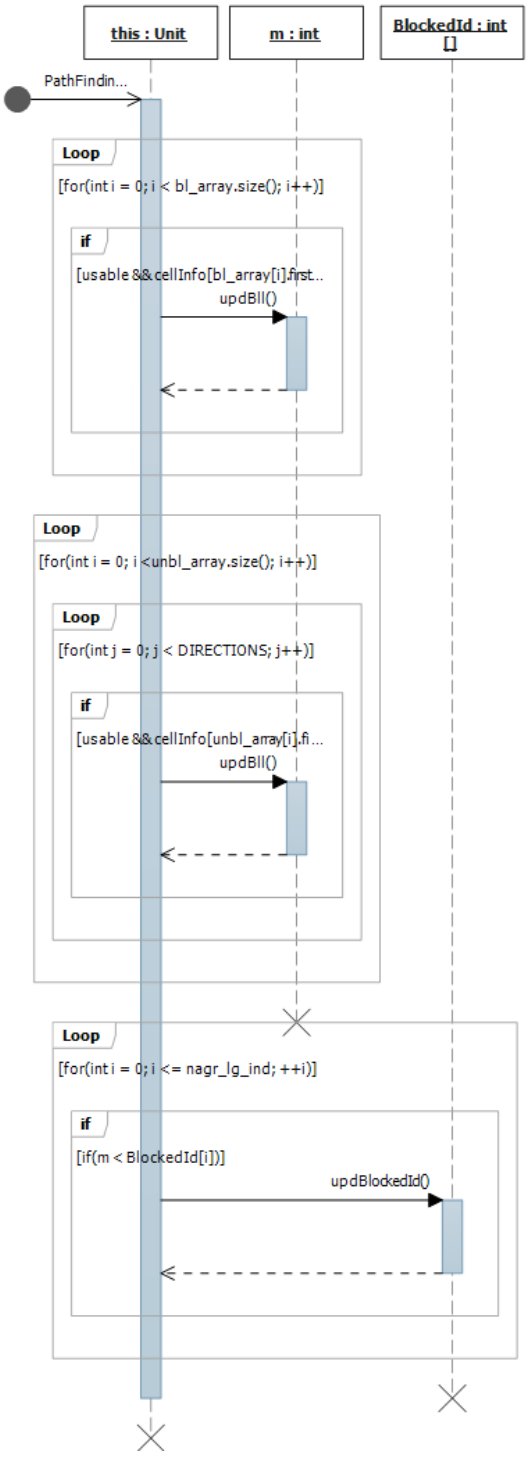
2.4 pav. *Viršūnių įterpimo* dalies sekų diagrama

2.2.2. Kontūro suradimo ir kontūro patikrinimo žingsniai

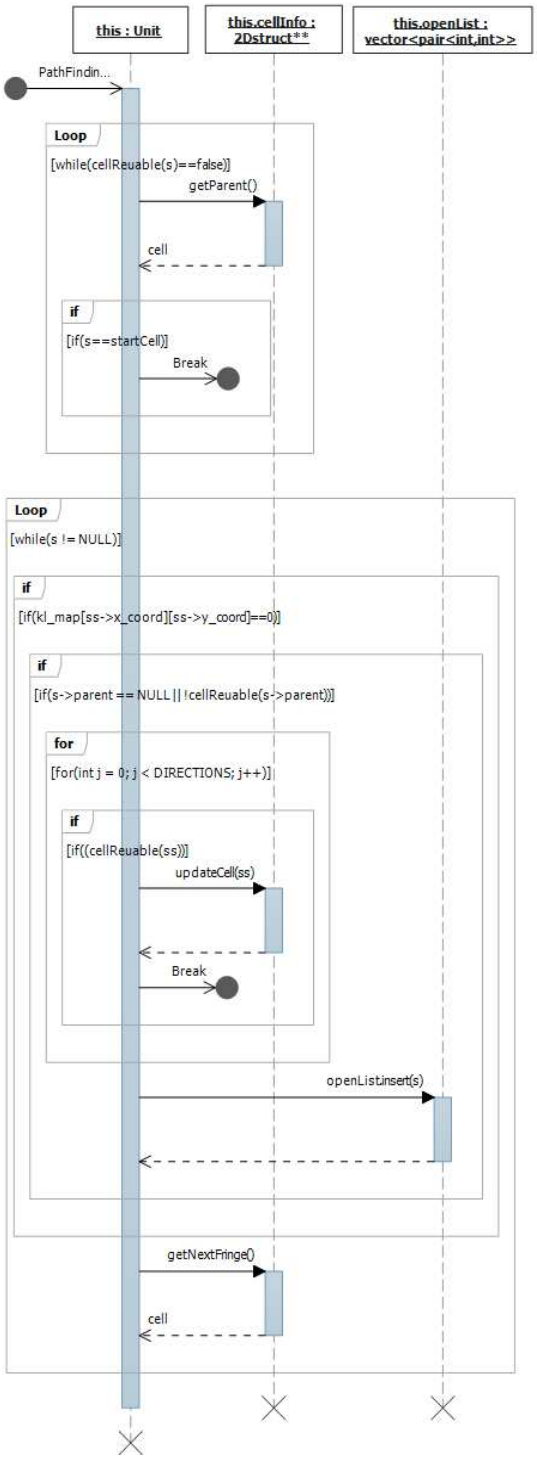
Kontūro suradimo žingsnyje yra (žr. 2.5 pav.) yra surandama minimali išnagrinėjimo eilės numerio reikšmė *m* viršūnės, kuri pakeitė pasiekiamumą ir turi įtakos surastam keliui. Pirmasis *for* ciklas yra vykdomas, kol patikrinamas *bl_array* sąrašas, kuriame talpinamos visos pasiekiamomis tapusios viršūnės, antrasis – kol patikrinami visi *unbl_array* sąraše, kuriame talpinamos visos nepasiekiamomis tapusios viršūnės, esančių viršūnių kaimynai. Pirmu ir antru *if* sakiniais tikrinama ar viršūnė yra mažiausią išnagrinėjimo eilės numerį ir surastam keliui įtaką turini viršūnė, jeigu taip, tai *m* reikšmė yra atnaujinama. Paskutiniame cikle yra atnaujinamos visų paieškų minimalios *m* reikšmės.

Kontūro patikrinimo žingsnyje (žr. 2.6 pav.), pirmame *while* cikle yra einama surastu keliu pradedant nuo tikslo viršūnės, kol aptinkama pirmoji viršūnė, kuri gali būti pakartotinai panaudota arba kol yra pasiekiama starto viršūnė (tokiu atveju praeitos paieškos kelias negali būti pakartotinai panaudotas ir *Kontūro patikrinimo* žingsnyje veiksmai toliau nėra atliekami). Antrasis *while* ciklas yra skirtas srities, kurioje yra visos pakartotinai panaudojamos viršūnės, kontūro viršūnėms apeiti. Srities viduje esančių viršūnių visos kaimyninės viršūnės yra arba pakartotinai panaudojamos, arba užblokuotos. Ant srities kontūro esančių viršūnių kaimynų tarpe yra ir viršūnių, kurios negali būti pakartotinai panaudotos. Ciklo pabaigoje esantis *getNextFringe()* kreipinys grąžina ant kontūro esančią kaimyninę viršūnę, t.y. turinčių dviejų rūšių kaimynų. Jeigu kaimynas jau buvo surastas, t.y.

buvo apeitas pilnas ratas aplink sritį, tai gražinama *NULL* reikšmė. Prieš kontūro viršūnę įterpiant į *AVS* sąrašą, patikrinama ar ji yra pasiekiama bei ar jos *tėvo* viršūnė yra pakartotinai panaudojama. Jeigu *tėvo* viršūnė netenkina pastarosios sąlygos arba *tėvo* viršūnė neegzistuoja, tada *tėvo* viršūnei priskiriama viena kontūro viršūnės kaimyninė viršūnė tenkinanti sąlygas.



2.5 pav. *Kontūro suradimo* dalies sekų diagrama

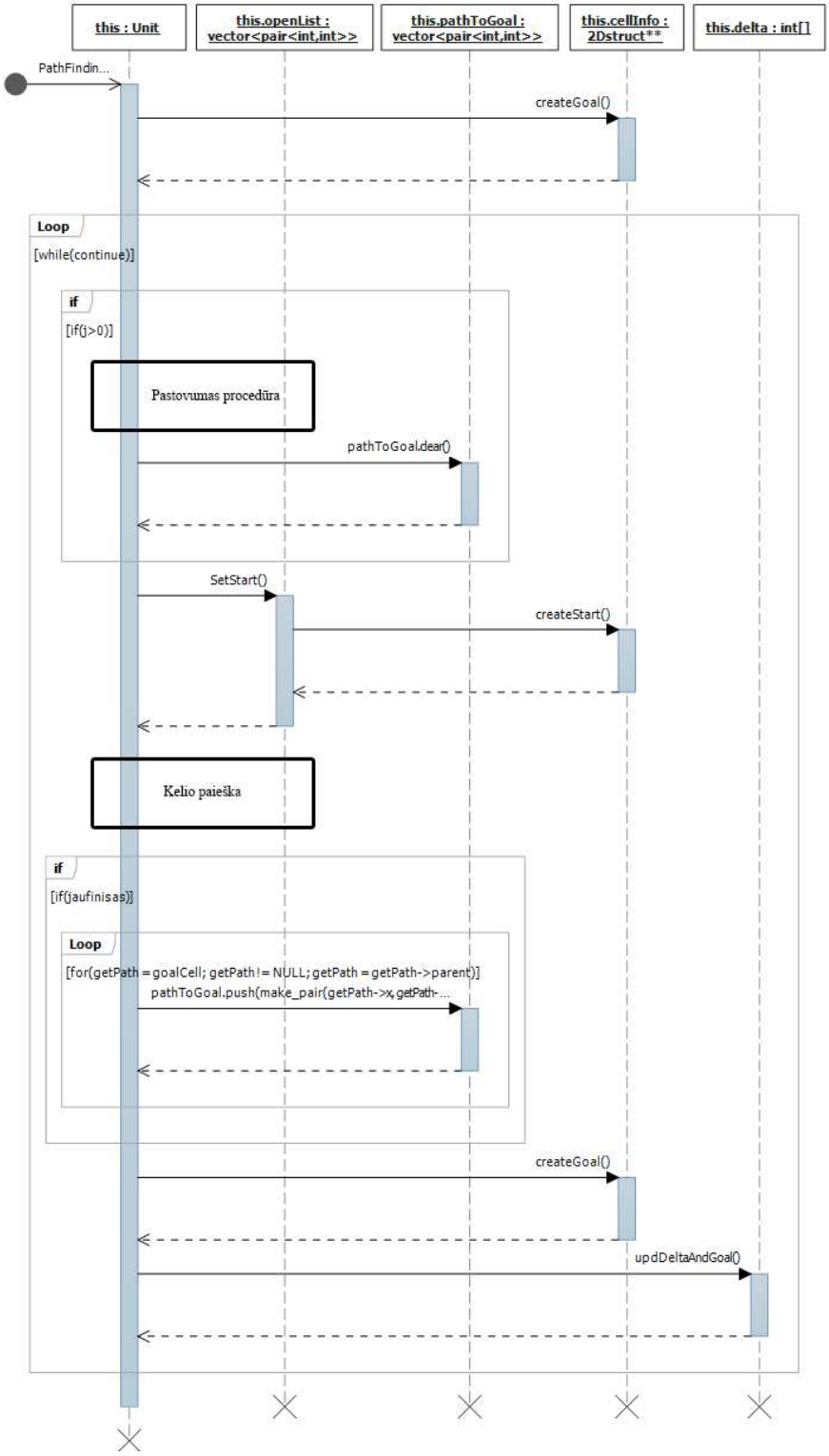


2.6 pav. *Kontūro patikrinimo* dalies sekų diagrama

2.3. GAA* algoritmo žingsnių realizacijos detalizavimas

GAA*, skirtingai negu A*, starto ir tikslo viršūnių sukūrimui turi atskirus metodus. Starto viršūnė sukuriame paieškos pradžioje, o tikslo – prieš tai buvusios paieškos pabaigoje (jeigu tai pirmoji paieška, tai tikslo viršūnė sukuriama pradžioje), nes naujos tikslo viršūnės reikia atnaujinant

delta (tikslo kitimo įvertis) atnaujinimui. Taip pat GAA* visose paieškose išskyrus pirmoje yra atnaujinamos pasiekiamomis tapusių viršūnių *h* reikšmės naudojant *pastovumo procedūrą*. *Kelio paieškos* ir *viršūnių įterpimo* dalys yra identiškos A* skyriuje (žr. 2.2.1 skyriuje) aprašytoms dalims.

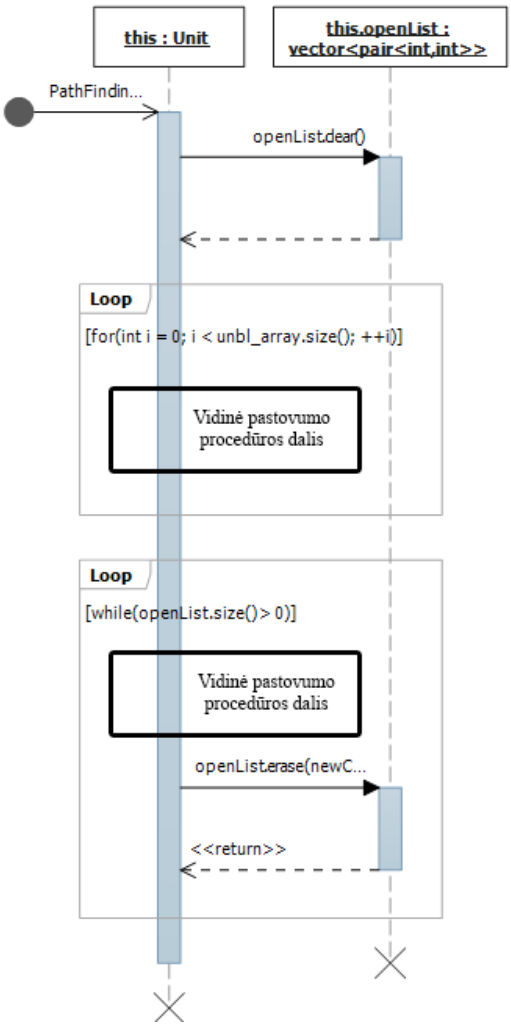


2.7 pav. GAA* realizacijos apibendrinta sekų diagrama

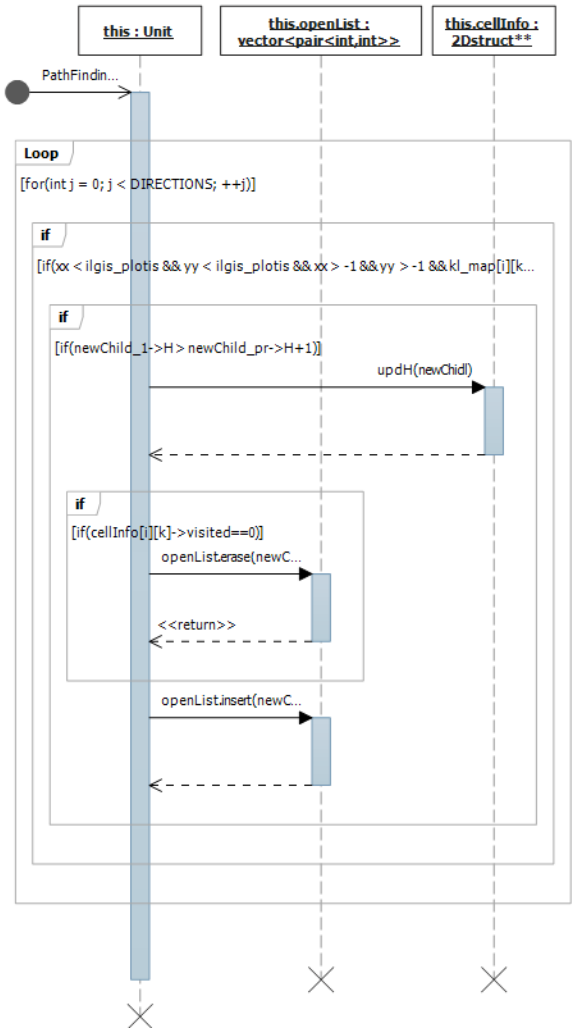
2.3.1. Pastovumo procedūros žingsniai

Pastovumo procedūros dalyje (žr. 2.8 pav.) prieš viršūnių tikrinimą yra išvalomas *AVS* sąrašas. Viršūnių tikrinimas atliekamas dviem etapais: pirmajame patikrinamos visos *unbl_array* sąraše esančios viršūnės, antrajame – *AVS* sąraše esančios viršūnės, patikrinus ištrinant jas iš *AVS* sąrašo. Abiem atvejais naudojami veiksmai pavaizduoti *vidinėje pastovumo procedūros* dalyje (žr. 2.9 pav.),

kurioje viršūnės *h* reikšmė yra lyginama su visomis pasiekiamų kaimyninių viršūnių *h* reikšmėmis ir priskiriama mažiausia reikšmė. Jeigu viršūnės *h* reikšmė pasikeitė, tai viršūnė yra įterpiama į *AVS* sąrašą. Po atliktų žingsnių turim viršūnes su atnaujintomis *h* reikšmėmis bei tuščią *AVS* sąrašą, paruoštą sekančiai paieškai.



2.8 pav. *Pastovumo procedūros* dalies sekų diagrama

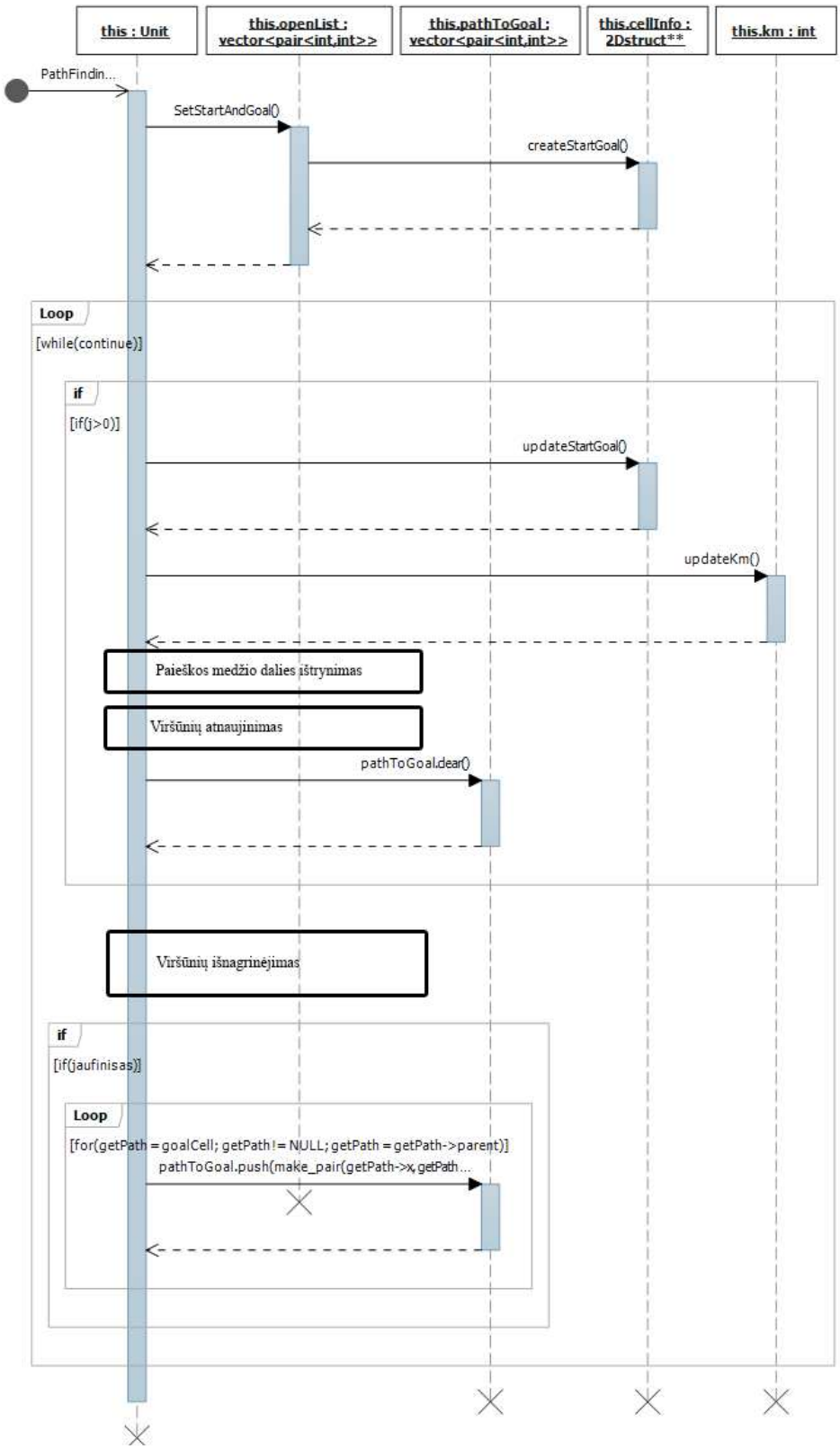


2.9 pav. *Vidinės pastovumo procedūros* dalies sekų diagrama

2.4. LPA*, D*Lite ir MT – D*Lite algoritmų žingsnių realizacijos detalizavimas

LPA*, D*Lite ir MT – D*Lite yra trečios plečiamos klasės algoritmai atitinkamai pritaikyti kiekvienai iš trijų situacijų (statinei, judėjimo link tikslo ir judančio taikinio). 2.10 pav. yra pavaizduota MT – D*Lite sekų diagrama, tačiau joje yra visi žingsniai reikalingi tiek D*Lite, tiek LPA* algoritmams.

Starto ir tikslo viršūnių sukūrimas ir starto viršūnės (D*Lite atveju tikslo viršūnės) koordinačių įterpimas į *AVS* sąrašą (*setStartGoal()*) atliekamas tik vienintelį kartą, pačioje algoritmo pradžioje, nes naujai paieškai yra naudojamas jau turimas *AVS* sąrašas. *UpdateStartGoal()* kreipinys pakeičia starto ir tikslo (D*Lite atveju tik starto) viršūnes (jų poziciją). *UpdateKm()* kreipinys pakoreguoja *km* reikšmę prie jos pridedant euristinės funkcijos apskaičiuotą reikšmę tarp praeitos ir šios paieškų tikslo viršūnių (D*Lite atveju starto viršūnių). LPA* atveju šie du kreipiniai nėra vykdomi, kadangi nei starto nei tikslo pozicijos nekinta. *Paieškos medžio dalies ištrynimo* dalyje esantys žingsniai yra atliekami tik MT – D*Lite algoritme.

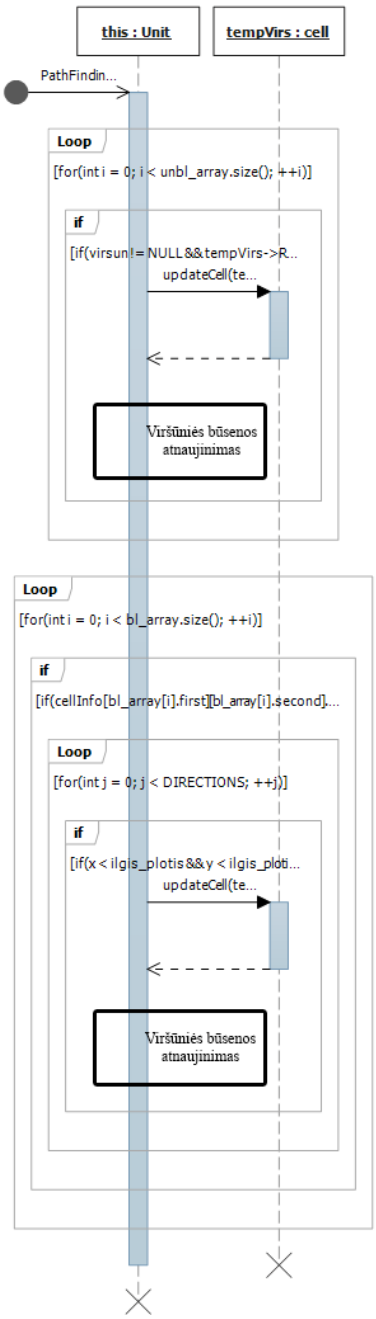


2.10 pav. MT – D*Lite realizacijos apibendrinta sekų diagrama

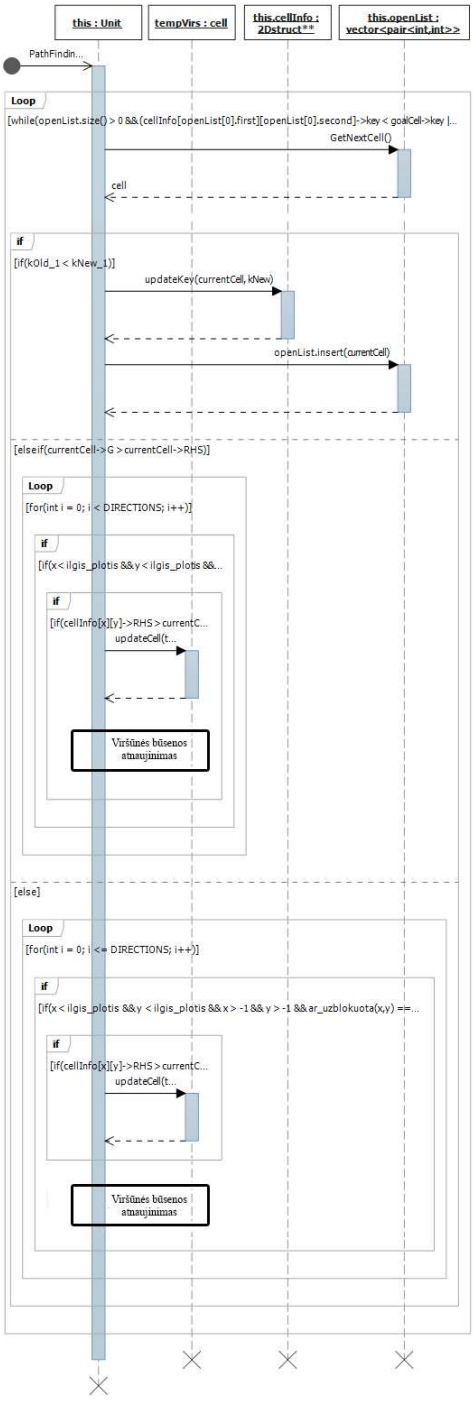
2.4.1. Viršūnių atnaujinimo ir viršūnių išnagrinėjimo žingsniai

Viršūnių atnaujinimo dalis (žr. 2.11 pav.) yra vykdoma tarp kelio paieškų ir joje yra atnaujinamos visos pasiekiamumą pakeitusios viršūnės bei jų kaimyninės viršūnės, kurios gali būti įtakotos pokyčių. Šią dalį sudaro du pagrindiniai *for* ciklai. Pirmajame cikle yra atnaujinamos visos pasiekiamomis tapusių viršūnių *rhs* bei *tėvo* viršūnių reikšmės naudojant *updateCell()* kreipinį bei vykdoma *viršūnės būsenos atnaujinimo* dalis. Antrajame cikle tikrinama nepasiekiamomis tapusių viršūnių kaimyninės viršūnės. Kadangi prieš tai vykdytose paieškose nesugeneruotos viršūnės neturi

jokios įtakos kaimyninėms viršūnėms, tai prieš pradedant tikrinti kaimynines viršūnės, patikrinama ar nepasiekiama tapusi viršūnė buvo sugeneruota. Paskutiniame *if* sakinyje tikrinama ar kaimyninės viršūnės *tėvas* yra nagrinėjama nepasiekiama tapusi viršūnė, jeigu taip, tai kaimyninė viršūnė yra atnaujinama.



2.11 pav. *Viršūnių atnaujinimo* dalies sekų diagrama



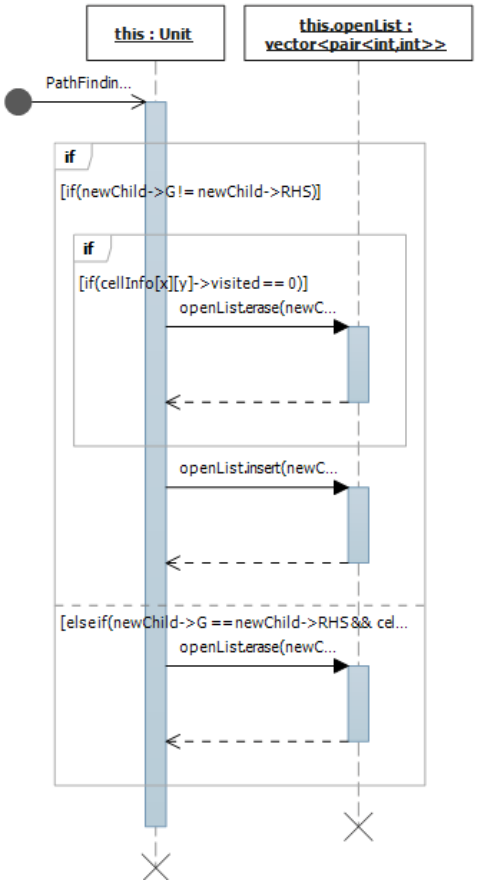
2.12 pav. *Viršūnių išnagrinėjimo* dalies sekų diagrama

Viršūnių išnagrinėjimo dalyje (žr. 2.12 pav.) pagrindinis ciklas vykdomas kol *AVS* sąrašas nėra tuščias ir yra tenkinama viena iš dviejų sąlygų: nagrinėjamos viršūnės bendra įverčio reikšmė yra mažesnė už tikslo viršūnės arba tikslo viršūnės *rhs* reikšmė > *g* reikšmę. D*Lite atveju *while* cikle tikslo viršūnė pakeičiame starto viršūne. Pirmasis *if* sakinio bloke esantys žingsniai, tik statinėse situacijose veikiančiame LPA* algoritme nėra atliekami. Jame tikrinama ar nagrinėjamos viršūnės turima bendra įverčio reikšmė yra mažesnė už bendrą įverčio reikšmę gautą įvertinus kintančios viršūnės poziciją. Jeigu sąlyga tenkinama, tai viršūnė įterpiama į *AVS* sąrašą su nauja įverčio reikšme. Jeigu sąlyga netenkinama, tai pereinama prie antros sąlygos – tikrinama ar nagrinėjamos

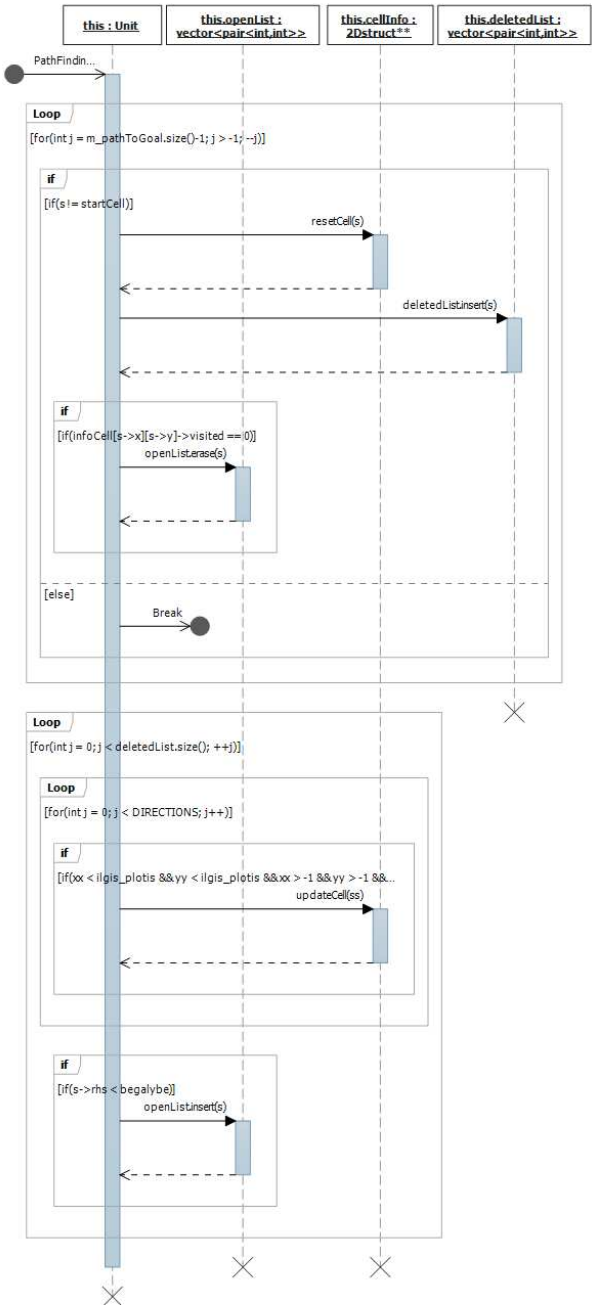
viršūnės *rhs* > *g*. Jeigu sąlyga tenkinama, tada tikrinama ar viršūnės kaimyno *rhs* reikšmė yra didesnė už nagrinėjamos viršūnės *g* reikšmę, padidinta vienetu (perėjimu tarp viršūnių svoriu). Jeigu taip, tai kaimyninė viršūnė yra atnaujinama ir atliekami veiksmai aprašyti *viršūnės būsenos atnaujinimo* dalyje. Jeigu antra sąlyga netenkinama – atliekami paskutiniame cikle esantys žingsniai, kurie yra identiški antroje sąlygoje atliekamiems žingsniams, tik šalia kaimyninių viršūnių yra patikrinama ir pati nagrinėjama viršūnė bei *viršūnės būsenos atnaujinimo* dalis yra vykdoma neatsižvelgiant į paskutinį *if* sakinį.

2.4.2. Viršūnės būsenos atnaujinimo ir Paieškos medžio dalies ištrynimo žingsniai

Viršūnės būsenos atnaujinimo dalyje (žr. 2.13 pav.) pirmuoju *if* sakiniu tikrinama ar viršūnė nėra pastovi (*rhs* ≠ *g*). Jeigu sąlyga tenkinama, tai viršūnė įterpiama į *AVS* sąrašą, prieš tai ištrynus viršūnę iš *AVS* sąrašo, jeigu ji tuo metu buvo patalpinta jame. Jeigu sąlyga netenkinama, tikrinama ar viršūnė yra pastovi ir ar ji yra įterpta į *AVS* sąrašą. Pastovios viršūnės yra ištrinamos iš *AVS* sąrašo.



2.13 pav. *Viršūnės būsenos atnaujinimo* dalies sekų diagrama



2.14 pav. *Paieškos medžio dalies ištrynimo* dalies sekų diagrama

Paieškos medžio dalies ištrynimo dalyje (žr. 2.14 pav.) pirmo *for* ciklo metu yra tikrinamos į kelią įeinančios viršūnės, pradedant nuo pradinės (nuo paskutinės sąraše įrašytos). Ciklas yra nutraukiamas jeigu aptinkama naujos paieškos starto viršūnė. Ciklo metu naudojant *resetCell()* kreipinį viršūnių *rhs* ir *g* reikšmėms priskiriama begalybė, o *tėvo* viršūnei *NULL* bei viršūnė yra ištrinama iš *AVS* ir įterpiama į *IVS* sąrašą. Antruoju ir trečiuoju *for* ciklais atnaujinamos visų *IVS* sąraše esančių viršūnių kaimynų *rhs* ir *tėvo* viršūnių reikšmės.

3. EKSPERIMENTO DALIS

3.1. Eksperimento metodologija

Šiame darbe yra atliekami pagrindinių dinaminių kelio paieškos algoritmų tyrimai dinaminėse aplinkose. Tyrimai atliekami trijose skirtingose situacijose: stacionarioje, judėjimo link tikslo bei judančio taikinio. Stacionarioje situacijoje yra atliekama šimtas kelio perskaičiavimo iteracijų. Judėjimo link tikslo bei judančio taikinio situacijose skaičiavimai atliekami kol yra pasiekiama tikslo viršūnė. Starto pozicija kinta dviem vienetais, tikslo – vienu. Algoritmų našumą nusakančios charakteristikos: vienos iteracijos vidutinis skaičiavimo laikas, vidutinis išnagrinėtų viršūnių kiekis ir vidutinis įterpimų į *AVS* sąrašą skaičius. Našumą nusakančios charakteristikos kiekvienai situacijai apskaičiuojamos išvedant vidurkį iš dešimties tos pačios testavimo aplinkos paleidimų. Kadangi algoritmų veikimo principas visose situacijose yra panašus, tai galutiniai rezultatai yra gaunami išvedant vidurkį iš visų trijų situacijų. Trys trečios plečiamos klasės algoritmai (LPA*, D*Lite ir MT – D*Lite) tyrime bus naudojami kaip vienas (toliau III I.K.A.)

Šiame darbe vykdomų tyrimų sąrašas:

1. Algoritmų našumo priklausomybė nuo aplinkos dydžio.
2. Algoritmų našumo priklausomybė nuo atstumo tarp starto ir tikslo viršūnių.
3. Algoritmų našumo priklausomybė nuo nepasiekiamų viršūnių kiekio.
4. Algoritmų našumo priklausomybė nuo pasiekiamumą keičiančių viršūnių kiekio.
5. Algoritmų našumo priklausomybė nuo nekintančių viršūnių kiekio aplink starto ir tikslo viršūnes.
6. Algoritmų našumas, kai tarp starto ir tikslo viršūnių kelias neegzistuoja (atliekamas tik statinėje situacijoje).

3.2. Įrangos aprašymas

Visų testavimų metu yra naudojama vienoda aparatūrinės bei programinės įrangos konfigūracija . Algoritmų našumas tiesiogiai priklauso nuo naudojamos įrangos, todėl juos testuojant kitoje aplinkoje galimas kitokių rezultatų gavimas.

Tyrime naudojama įrangą:

- Centrinis procesorius: Intel(R) Core(TM) i7-2600K CPU @3.40GHz.
- Operatyvioji atmintis: DDR3 Kingston HyperX Genesis 4x4GB 1600MHz CL9.
- Vaizdo plokštė: GeForce GTX 460 1024MB.
- Operacinė sistema: Microsoft Windows 7 64 bitų.

3.3. Testavimo aplinkos

Testavimo aplinkos generuojamos naudojant tinklo (*angl. grid*) struktūrą. Tinklo elementas atitinka viršūnės elementą, naudojamą kelio paieškoje. Viršūnė turi po aštuonis kaimynus. Perėjimai tarp greta esančių viršūnių lygūs vienetui. Kuriant aplinkas buvo atsižvelgta į šiuos faktorius: aplinkos dydį, pradinių nepasiekiamų viršūnių kiekį, atstumą tarp starto ir tikslo viršūnių, pasiekiamumą keičiančių viršūnių kiekį bei nekintančių viršūnių aplink starto ir tikslo viršūnes kiekį.

Nekintantys parametrai: aplinkos dydis – 200x200, atstumas tarp starto ir tikslo viršūnių – 200, nepasiekiamų viršūnių kiekis – 8000, pasiekiamumą keičiančių viršūnių kiekis – 2000, nekintančių viršūnių aplink starto ir tikslo viršūnes kiekis – (0, 0).

3.1 lentelė Aplinkų kintančių parametrų reikšmės

Tyrimas	Kintantis parametras	Kintančių parametrų reikšmės
1	Aplinkos dydis	100x100, 200x200, 300x300, 400x400, 600x600, 800x800
	Atstumo tarp starto ir tikslo viršūnių	100
	Nepasiekiamų viršūnių kiekis	3000
	Pasiekiamumą keičiančių viršūnių kiekis	1500
2	Atstumo tarp starto ir tikslo viršūnių	50, 100, 150, 200
3	Nepasiekiamų viršūnių kiekis	1000, 2000, 4000, 8000, 16000
4	Pasiekiamumą keičiančių viršūnių kiekis	500, 1000, 2000, 4000, 8000
5	Nekintančių viršūnių kiekis aplink starto ir tikslo viršūnes	(0, 0), (2000, 0), (0, 2000), (1000, 1000)

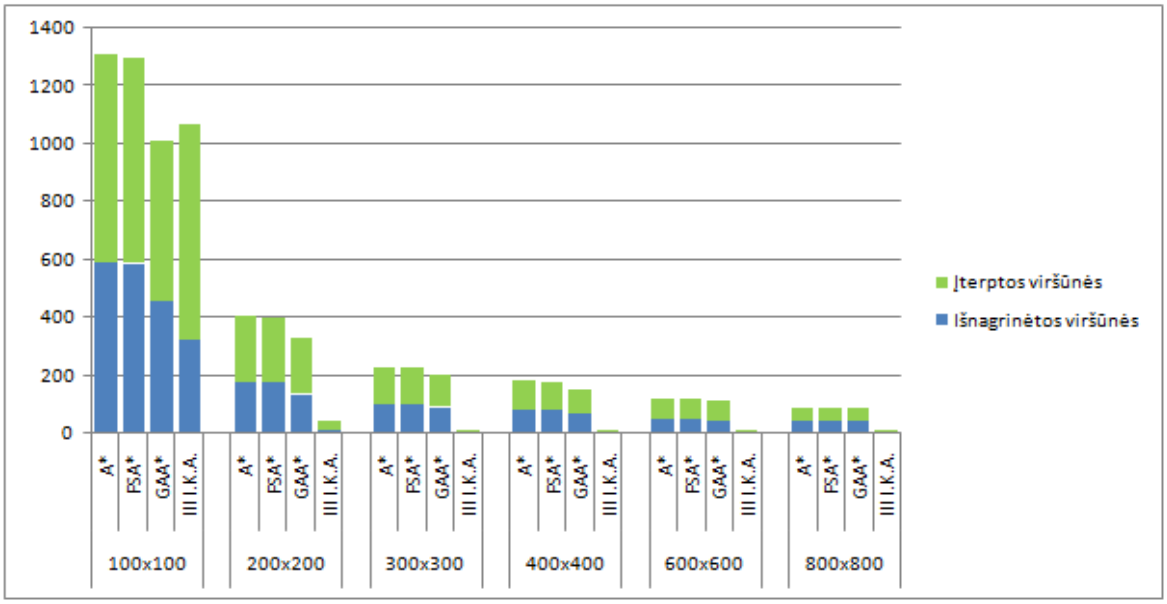
3.4. Eksperimentų rezultatai

Dėl A* ir FSA* algoritmų veikimo panašumų ir nekintančių viršūnių kiekio aplink starto ir tikslo viršūnes pasirinkimo galima tikėtis nedidelių skirtumų tarp algoritmų našumo charakteristikų. Trečios plečiamos klasės algoritmai, naują paiešką pradėdami ne su tuščiu *AVS* sąrašu didžiojoje dalyje tyrimų gali būti daugiau negu du kartus našesni negu kiti tyrime naudojami algoritmai.

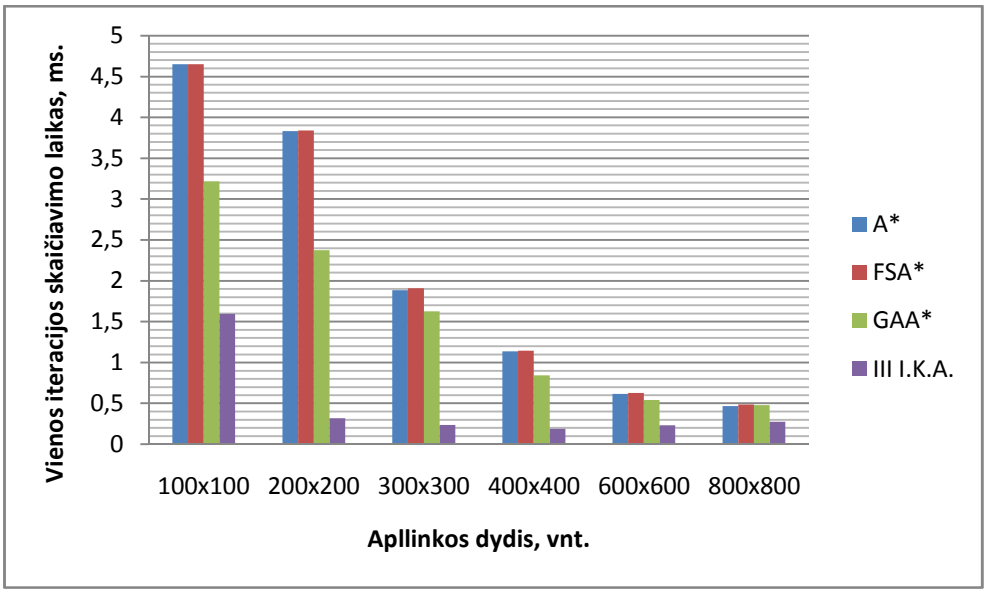
3.4.1. Našumo priklausomybės nuo aplinkos dydžio tyrimas

Aplinkos dydžiui mažėjant, visų nagrinėjamų algoritmų našumo charakteristikos didėja (žr. 3.1 pav. ir 3.2 pav.), nes esant didesnei aplinkai nepasiekiamos viršūnės turi didesnį išsidėliojimo plotą, taip sumažindamos tikimybę turėti įtakos kelio paieškai. Aplinkai padidėjus nuo 100x100 iki 800x800 A* ir FSA* išnagrinėtų viršūnių kiekis sumažėjo ~92,5%, įterptų viršūnių skaičius sumažėjo ~93,5%, skaičiavimo laikas sumažėjo ~ 90,7%, GAA* charakteristikos atitinkamai sumažėjo ~90,2%, ~91,5% ir~ 85%. Aplinkos dydis neturi įtakos A* ir GAA* išnagrinėtų ir į sąrašą įterptų viršūnių tarpusavio santykiui. GAA* visada išnagrinės ir į sąrašą įterps ne daugiau viršūnių negu A*. Aplinkos dydžiui didėjant skaičiavimo laiko skirtumas tarp GAA* ir A* mažėja bei esant sąlyginai didelei aplinkai, A* skaičiavimams skirtas laikas yra mažesnis negu GAA*, nes skirtumas tarp išnagrinėtų ir įterptų viršūnių yra nežymus, o GAA* algoritme yra papildomai vykdoma *pastovumo* procedūra.

Trečios klasės plečiamų algoritmų atveju esant sąlyginai didelei aplinkai (mūsų atveju didesnei nei 200) aplinkos dydis neturi didelės įtakos, nes yra pakoreguojamos tik tos viršūnės, kurioms įtakos turi pasiekiamumą pakeitusios viršūnės. Esant didelei aplinkai yra didelė tikimybė, kad viršūnėms, kurias reiktų pakoreguoti, nereikės atlikti jokių korekcijų, nes jos dar nebuvo sugeneruotos praeitų paieškų. Tuo tarpu esant mažai aplinkai tikimybė yra maža, todėl aplinkai padidėjus nuo 100x100 iki 200x200 charakteristikos atitinkamai sumažėja ~95,5%, ~95,6% ir ~80%. Visais nagrinėtais atvejais III I.K.A. rodė didesnį našumą už kitus tyrime naudojamus algoritmus.



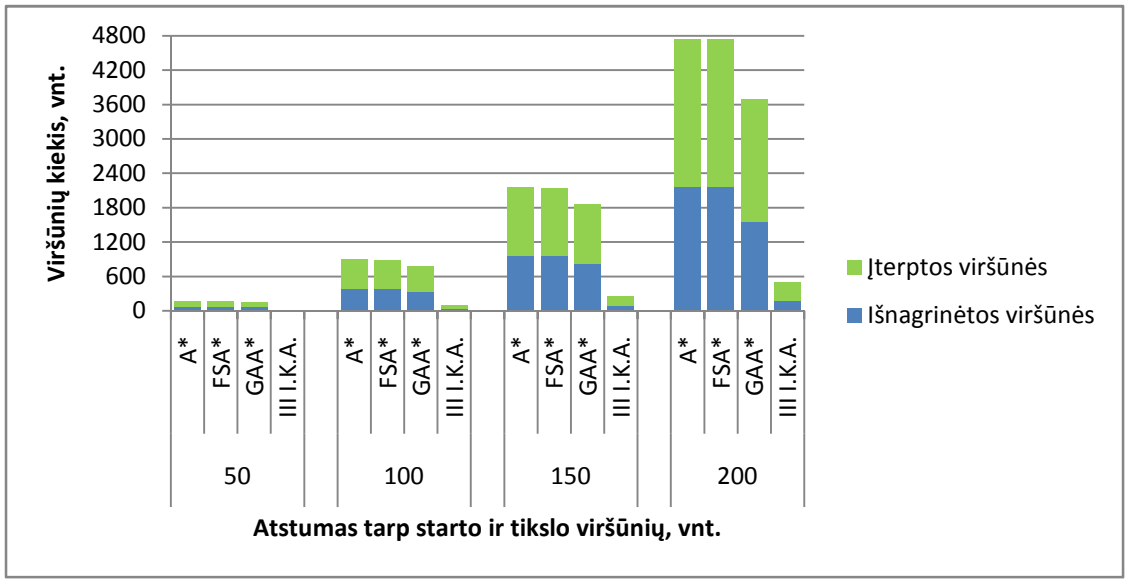
3.1 pav. Išnagrinėtų ir įterptų viršūnių kiekio priklausomybės nuo aplinkos dydžio grafikas



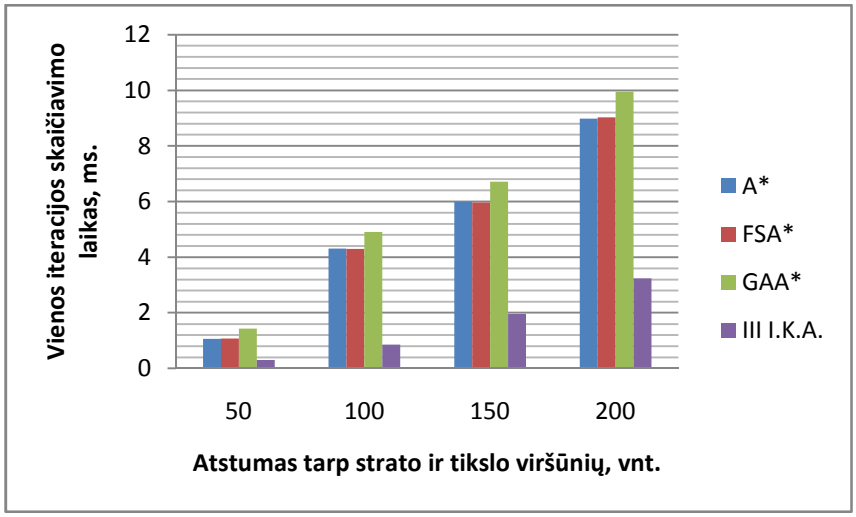
3.2 pav. Skaičiavimo laiko priklausomybės nuo aplinkos dydžio grafikas

3.4.2. Našumo priklausomybės nuo atstumo tarp starto ir tikslo viršūnių tyrimas

Kintant atstumui tarp starto ir tikslo viršūnių, išnagrinėtų ir įterptų viršūnių skaičiaus kitimas yra panašus visuose algoritmuose (žr. 3.3 pav.). Atstumui sumažėjus nuo 200 iki 50 A* ir FSA* išnagrinėtų ir įterptų viršūnių kiekis sumažėjo ~96,3% ir 97%, GAA* – ~95,8% ir ~96,6 %, o III I.K.A. – ~96,7% ir ~96,9%. Nors algoritmų kelio paieškos laikų tarpusavio santykis kinta nežymiai ir kintant atstumui nei vienas algoritmas neįgyja pranašumo, tačiau atstumas turi įtakos bendram algoritmų veikimo laiko tarpusavio santykiui. A* turi ~34% didesnį našumą negu GAA*, tačiau atstumui tarp starto ir tikslo kintant nuo 50 iki 200, našumas mažėja iki ~10%, nes didėjant atstumui didėja atnaujintų viršūnių įtaka kelio paieškai, dėl ko yra išnagrinėjama mažiau viršūnių. III I.K.A. laiko atžvilgiu yra nuo ~63% iki ~80% efektyvesnis už A* ir nuo ~66,7% iki ~82,4% – už GAA* (žr. 3.4 pav.).



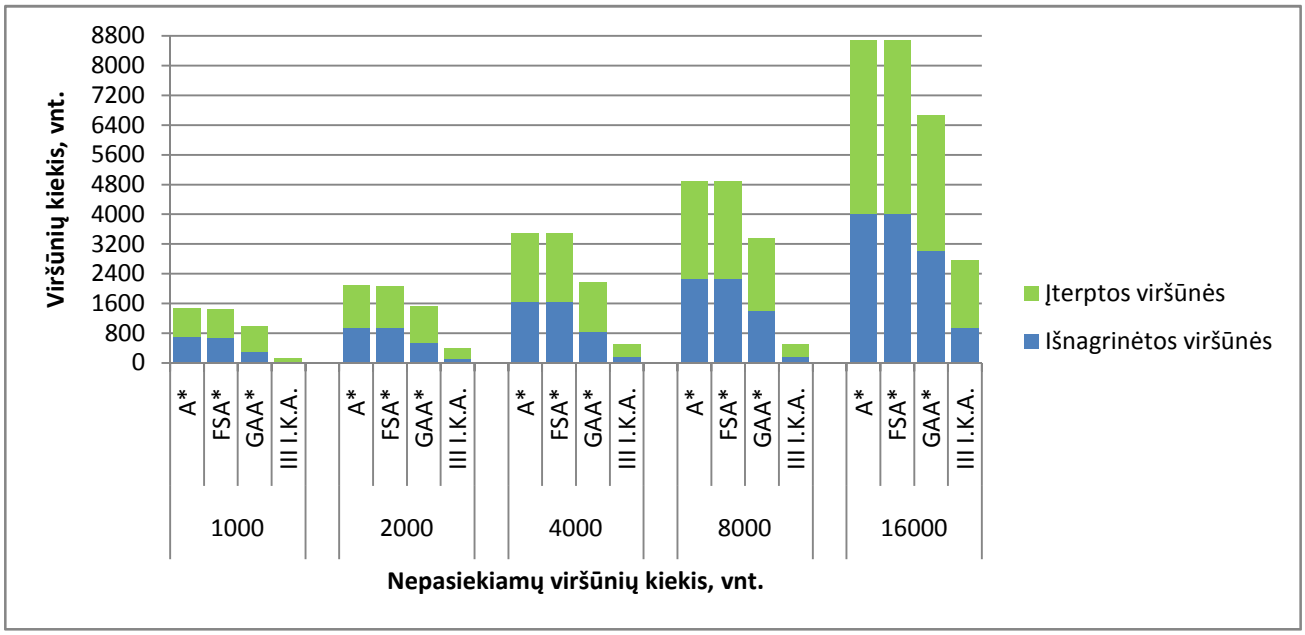
3.3 pav. Išnagrinėtų ir įterptų viršūnių kiekio priklausomybės nuo atstumo tarp starto ir tikslo viršūnių grafikas



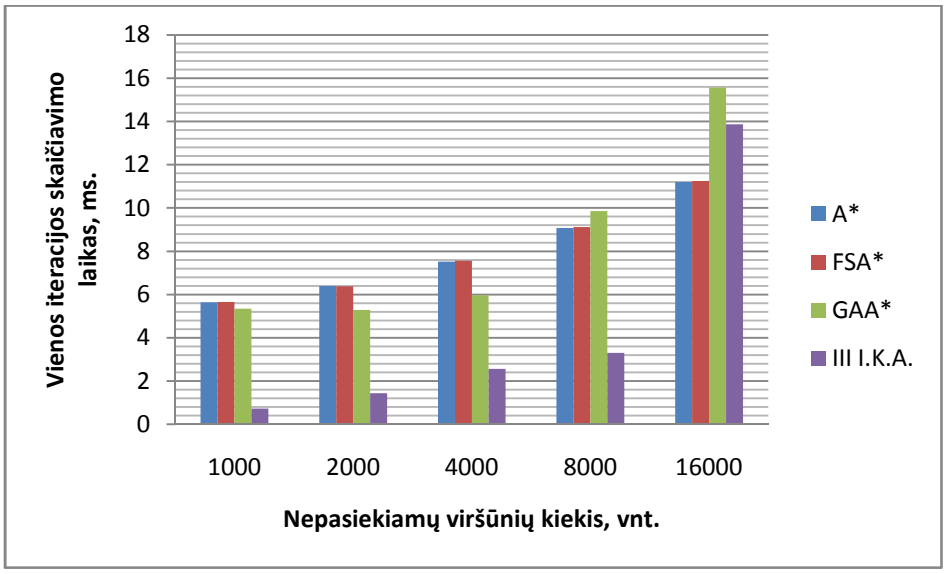
3.4 pav. Skaičiavimo laiko priklausomybės nuo atstumo tarp starto ir tikslo viršūnių grafikas

3.4.3. Našumo priklausomybės nuo nepasiekiamų viršūnių kiekio tyrimas

Iš 3.5 pav.**Error! Reference source not found.** ir 3.6 pav. matome, kad didėjant nepasiekiamų viršūnių kiekiui didėja visų algoritmų visos našumą nusakančios charakteristikos, nes kuo daugiau yra nepasiekiamų viršūnių, tuo daugiau jų turi būti apeinama ir tuo paieška tampa platesnė (užima didesnę aplinkos sritį). Nepasiekiamų viršūnių kiekiui pakitus nuo 16000 iki 1000 A* ir FSA* išnagrinėtų ir įterptų viršūnių kiekiai sumažėjo ~82,8% ir 83,5%, skaičiavimo laikas – ~50%, GAA* charakteristikos atitinkamai sumažėjo ~89,8%, 81,3% ir 65,7%, o III I.K.A. – 97,1%, 94,9% ir 94,7%.



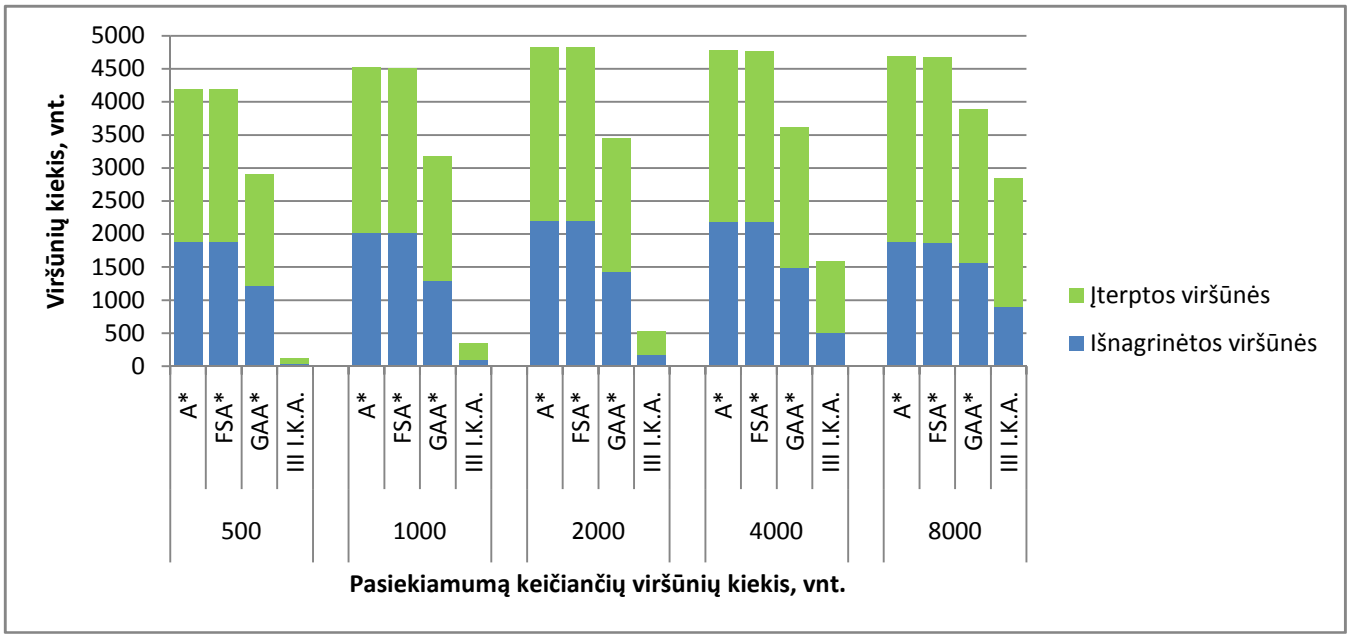
3.5 pav. Išnagrinėtų ir įterptų viršūnių kiekio priklausomybės nuo nepasiekiamų viršūnių kiekio grafikas



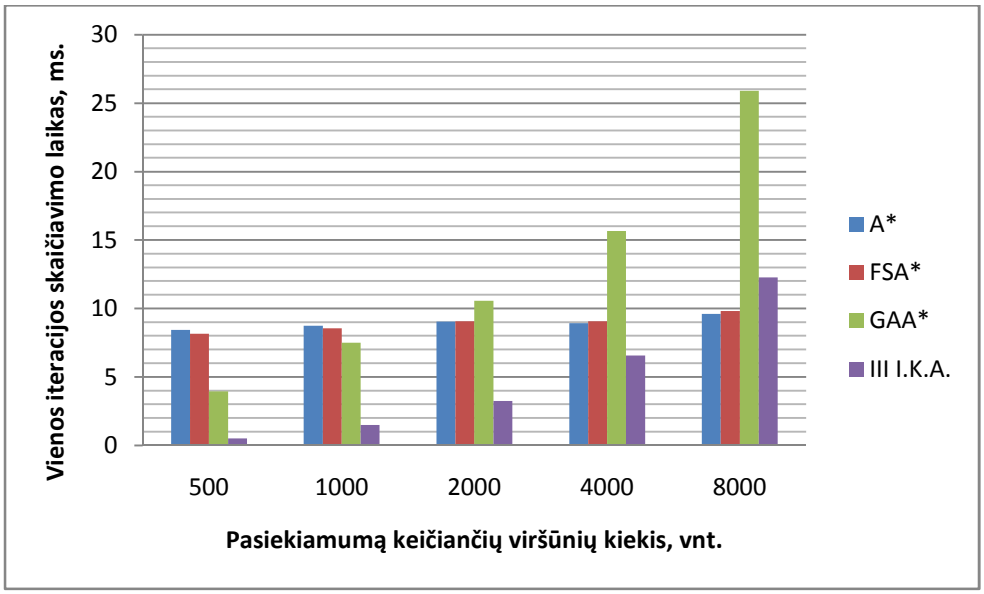
3.6 pav. Skaičiavimo laiko priklausomybės nuo nepasiekiamų viršūnių kiekio grafikas

3.4.4. Našumo priklausomybės nuo pasiekiamumą keičiančių viršūnių kiekio tyrimas

Pasiekiamumą keičiančių viršūnių kiekis neturi jokios įtakos A* algoritmui, kadangi jame nėra atnaujinamos viršūnės, kurios pakeitė pasiekiamumą. Taip pat neturi įtakos FSA* išnagrinėtų ir įterptų viršūnių kiekiui, tačiau turi įtakos nuo ~1% iki ~2% bendram skaičiavimo laikui. GAA* algoritmo išnagrinėtų ir įterptų viršūnių sumažėjimą (~23% ir ~ 27%) pasiekiamumą keičiančių viršūnių kiekiui sumažėjus nuo 8000 iki 500, įtakoja viršūnių, kurios tapo nepasiekiamomis kiekis (žr. 3.7 pav.) GAA* skaičiavimo laikas sumažėjo ~84,6%, nes mažėjant pasiekiamumą keičiančių viršūnių kiekiui, *pastovumo* procedūros, kuri atnaujina pasiekiamomis tapusių viršūnių *h* reikšmes, skaičiavimo laikas taip pat mažėja. Kadangi skirtingai negu GAA*, III I.K.A. tarp paieškų atnaujina *rhs* reikšmes viršūnių, kurios tapo tiek pasiekiamos, tiek nepasiekiamos, todėl pasiekiamumą keičiančių viršūnių kiekis III I.K.A. turi didesnę įtaką (našumo charakteristikos sumažėjo ~95,3%, ~95,9% ir ~95,7%). Kadangi A* ir FSA* skaičiavimo laikai kinta nežymiai, tai pasiekiamumą keičiančių viršūnių kiekiui esant 2000 jų veikimo laikas yra ~13,8%, o esant 8000 – net ~62,7% mažesnis negu GAA* ir ~21,2% mažesnis negu III I.K.A.



3.7 pav. Išnagrinėtų ir įterptų viršūnių kiekio priklausomybės nuo pasiekiamumą keičiančių viršūnių kiekio grafikas

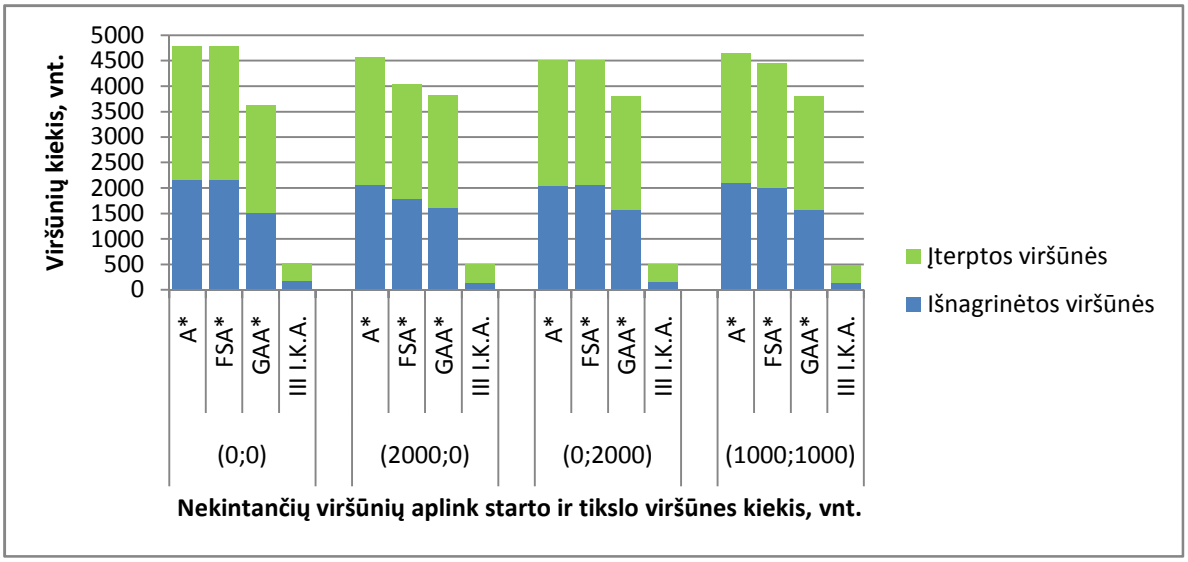


3.8 pav. Skaičiavimo laiko priklausomybės nuo pasiekiamumą keičiančių viršūnių kiekio grafikas

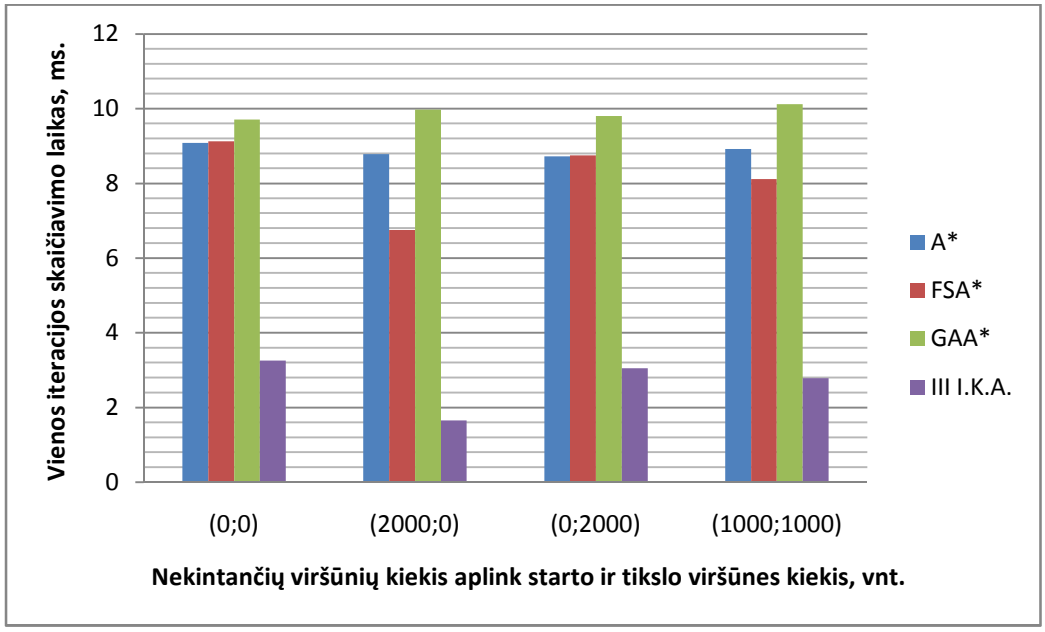
3.4.5. Našumo priklausomybės nuo nekintančių viršūnių kiekio aplink starto ir tikslo viršūnes tyrimas

Nekintančių viršūnių aplink starto ir tikslo viršūnes kiekis neturi jokios įtakos A* ir GAA* algoritmams, nes kiekviena nauja paieška yra atliekama esant tuščiam AVS sąrašui. 3.9 pav. matomi išnagrinėtų ir įterptų viršūnių nežymūs pokyčiai bei 3.10 pav. matomi skaičiavimo laikų nežymūs pokyčiai yra įtakojami viršūnių išsidėstymo aplinkoje. FSA* našumas prieš A* didėja su didėjančiu nekintančių viršūnių aplink starto viršūnę kiekiu. Kai kiekis yra 2000, FSA* išnagrinėtų viršūnių kiekis sumažėja ~13%, įterptų – ~9,9%, o skaičiavimo laikas – ~33%. Kai kiekis 1000 atitinkamai – ~3,8%, ~3,8% ir ~9%. Didėjantis nekintančių viršūnių aplink tikslo viršūnę kiekis FSA* našumui turi tik judėjimo link tikslo ir judančio taikinio situacijose, kai atstumas tarp starto ir tikslo viršūnių yra nedidelis. Tačiau našumo padidėjimas yra beveik nulinis. III I.K.A. našumas didėja, didėjant nekintančių viršūnių kiekiui, nes mažėja viršūnių, kurias gali paveikti pasiekiamumą keičiančios viršūnės, kiekis. III I.K.A. nekintančių viršūnių kiekis aplink starto viršūnę turi didesnę įtaką negu aplink tikslo viršūnę, nes tiesioginė paieška yra atliekama statinėje ir judančio taikinio situacijoje, o atbulinė paieška atliekama tik judėjimo link tikslo situacijoje. Aplink startą esant 2000 nekintančių

viršūnių algoritmo skaičiavimo laikas sumažėjo ~49,2%. Esant 2000 nekintančių viršūnių aplink tikslą – ~5,6%.



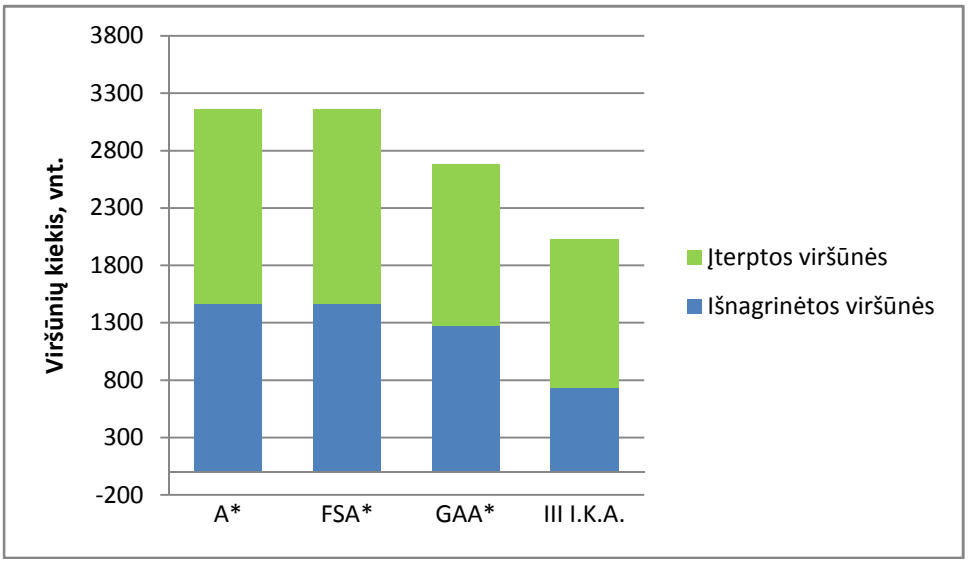
3.9 pav. Išnagrinėtų ir įterptų viršūnių kiekio priklausomybės nuo nekintančių viršūnių aplink starto ir tikslo viršūnes kiekio grafikas



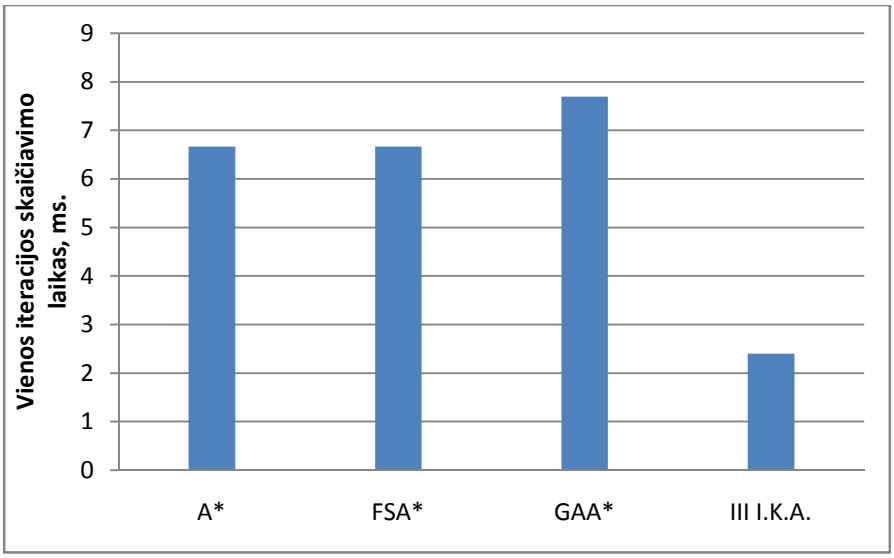
3.10 pav. Skaičiavimo laiko priklausomybės nuo nekintančių viršūnių aplink starto ir tikslo viršūnes kiekio grafikas

3.4.6. Našumo, kai tarp starto ir tikslo viršūnių kelias neegzistuoja tyrimas

Iš 3.11 pav. ir 3.12 pav. matome, kad visais atvejais, kai kelias tarp starto ir tikslo neegzistuoja, A* ir FSA* algoritmų veikimas yra identiškas (yra išnagrinėjamas ir įterpiamas toks pats skaičius viršūnių bei skaičiavimo laikai taip pat sutampa), kadangi kai kelias neegzistuoja, tai FSA* algoritme nėra vykdomi papildomi skaičiavimai. GAA* išnagrinėja ir įterpia ~12,3% ir ~16,7% mažiau viršūnių negu A* ir FSA*, tačiau dėl *pastovumo* procedūros bendras skaičiavimo laikas yra ~1,15 karto ilgesnis negu A* ir FSA*. III I.K.A. tiek išnagrinėtų ir įterptų viršūnių skaičiumi (nuo ~2% iki ~50%), tiek skaičiavimo laiku (nuo ~62,9% iki ~68,7%) yra našesnis už kitus tyrime naudojamus algoritmus.



3.11 pav. Išnagrinėtų ir įterptų viršūnių kiekių, kai tarp starto ir tikslo viršūnių kelias neegzistuoja, grafikas



3.12 pav. Skaičiavimo laiko, kai tarp starto ir tikslo viršūnių kelias neegzistuoja, grafikas

4. IŠVADOS

1. Nustatyta, kad ne visi algoritmai atitinka darbo pradžioje nusistatytus kriterijus – gaunamas optimalus (ne ilgesnis nei A*) kelio paieškos sprendimas aplinkose, kur perėjimų tarp viršūnių svoriai gali didėti ir mažėti. Tačiau kiekvienoje plečiamų algoritmų klasėje buvo bent po vieną algoritmą tenkinantį sąlygas.
2. Keičiant aplinką nusakančius parametrus keičiasi ir algoritmų našumas, tačiau tie patys parametrų pokyčiai skirtingiems algoritmams turi skirtingą poveikį.

2.1. A* našumas yra didesnis už GAA* ir trečios plečiamos klasės algoritmus (algoritmai, kuriuose atnaujinamos svorio tarp starto ir nagrinėjamos viršūnės reikšmės) aplinkose, kuriose yra didelis nepasiekiamų ir pasiekiamumą keičiančių viršūnių kiekis. Kai nepasiekiamų viršūnių kiekis viršija 8000, A* ~7,5% našumu lenkia GAA*, viršūnių kiekiui padidėjus iki 16000 našumas išauga iki ~27,8%, o trečios plečiamos klasės algoritmų atžvilgiu iki ~18,9%. Kai pasiekiamumą keičiančių viršūnių kiekis viršija 2000, A* ~14,2% našumu lenkia GAA*, viršūnių kiekiui padidėjus iki 8000 našumas išauga iki ~62,9%, o trečios plečiamos klasės algoritmų atžvilgiu iki ~21,7%. A* yra našesnis, nes tarp paieškų nėra atliekami jokie viršūnių atnaujinimai. Kai aplink starto viršūnę yra mažai pasiekiamumo nekeičiančių viršūnių, A* našumas yra didesnis už FSA*.

2.2. FSA* prieš A* turi našumo persvarą tik, kai naujai paieškai galima pakartotinai panaudoti sąlyginai didelę praeitoje paieškoje rasto kelio dalį. Kai aplink starto viršūnę yra 1000 pasiekiamumo nekeičiančių viršūnių, tai FSA* yra ~9% našesnis už A*, o viršūnių kiekiui esant 2000 – ~23,2%. Dėl beveik identiško veikimo principo kaip ir A* algoritmo, FSA* našumas prieš GAA* ir trečios plečiamos klasės algoritmus išryškėja tokiose pat aplinkose kaip ir A* atveju.

2.3. GAA* našumo didėjimą A* ir FSA* atžvilgiu lemia *pastovumo* procedūrą įtakojantys veiksniai: mažėjantis nepasiekiamų ir pasiekiamumą keičiančių viršūnių kiekis bei didėjantis atstumas tarp starto ir tikslo viršūnių. Kai atstumas tarp starto ir tikslo viršūnių yra 50, A* yra ~25,6% našesnis už GAA*, atstumui padidėjus iki 200 A* našumas siekia 9,7%. Esant mažam nepasiekiamų ir pasiekiamumą keičiančių viršūnių kiekiui *pastovumo* procedūros veikimo laikas taip pat trumpėja, o didėjant atstumui tarp starto ir tikslo viršūnių didėja kelio paieškos skaičiavimo laikas, taip *pastovumo* procedūra mažiau įtakoja bendrą veikimo laiką. Tiek GAA*, tiek trečios plečiamos klasės algoritmai yra daugiausiai priklausomi nuo nepasiekiamų ir pasiekiamumą keičiančių viršūnių kiekio, tačiau GAA* yra mažiau įtakojamas šių faktorių, todėl itin didelėm šių faktorių reikšmėm GAA* našumu gali pralenkti trečios plečiamos klasės algoritmus.

2.4. Du pagrindiniai trečios plečiamos klasės algoritmų našumą įtakojantys faktoriai yra nepasiekiamų ir pasiekiamumą keičiančių viršūnių kiekis. Algoritmas yra labiau priklausomas nuo šių faktorių negu GAA*, nes tarp iteracijų atnaujinant viršūnes yra atsižvelgiama į visas pasiekiamumą pakeitusias viršūnes, tuo tarpu GAA* atnaujinamos tik pasiekiamomis tapusios viršūnės. Nepasiekiamų viršūnių kiekiui kintant nuo 1000 iki 16000 trečios plečiamos klasės algoritmai yra vidutiniškai ~58,7% našesni už GAA* ir ~54% našesni už A* ir FSA*. Pasiekiamumą keičiančių viršūnių kiekiui kintant nuo 500 iki 8000 trečios plečiamos klasės algoritmai yra vidutiniškai ~69,3% našesni už GAA* ir ~47,8% našesni už A* ir FSA*.

5. LITERATŪRA

1. D* [žiūrėta 2012.01.10]. Prieiga per internetą: http://en.wikipedia.org/wiki/D*
2. Robin. What is heuristic search?. Aftificial intelligence. 2011 [žiūrėta 2012.02.10]. Prieiga per internetą: http://intelligence.worldofcomputing.net/ai-search/heuristic-search.html#.UXLe5cofh0Y
3. Sun, X; Yeoh, W; Koenig, S. Dynamic Fringe – Saving A*. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 891 – 898, 2009.
4. Hart, P; Nilsson, N; Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE, Transactions on Systems Science and Cybernetics. 1986, p. 100–107.
5. Sun, X; Koenig, S. The Fringe – Saving A* search algorithm – a feasibility study. In Proceedings of the International Joint Conference on Artificial Intelligence, 2007.
6. Koenig, S; Likhachev, M; Sun, X. Speeding up moving-target search. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, 2007.
7. Sun, X; Koenig, S; Yeoh, W. Generalized Adaptive A*. In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, 2008.
8. S, Koenig; M, Likhachev; D, Furcy. Lifelong Planning A*. Artificial Intelligence Journal, 155(1–2):93–146, 2004.
9. Stentz, A; Hebert, M. 1995. A complete navigation system for goal acquisition in unknown environments. Autonomous Robots 2(2):127–145.
10. Koenig, S; Likhachev, M. D* Lite. In Proceedings of AAAI, 2002.
11. Sun, X; Yeoh, W; Koenig, S. Generalized Fringe-Retrieving A*: Faster moving-target search on state lattices. In Proceedings of AAMAS, 2010.
12. Sun, X; Yeoh, W; Koenig, S. Moving Target D* Lite. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2010
13. Hernandez, C; Meseguer, P; Sun, X; Koenig, S. Path-Adaptive A* for incremental heuristic search in unknown terrain [short paper]. In Proceedings of the International Conference on Automated Planning and Scheduling, pages 358–361, 2009.
14. Likhachev, S; Gordon, G; Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In Advances in Neural Information Processing Systems. MIT Press.
15. Likhachev, M; Ferguson, D; Gordon, G; Stentz, A; Thrun, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In ICAPS, 262–271.
16. Hernandez, C; Sun, X; Koenig, S and Meseguer, P. Tree Adaptive A*. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 123-130, 2011.
17. Koenig, S. A comparison of fast search methods for real-time situated agents. In Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systemspages 864–871, 2004.
18. Koenig, S; Likhachev, M. Real-Time Adaptive A*. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, pages 281–288, 2006.
19. Koenig, S; Sun, X. 2009. Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents. Autonomous Agents and Multi-Agent Systems 18(3):313–341.
20. Bond, D; Widger, N; Ruml, W; Sun, X. Real-Time Search in Dynamic Worlds, Proceedings of the Symposium on Combinatorial Search (SoCS-10), 2010.