KAUNO TECHNOLOGIJOS UNIVERSITETAS
ELEKTROS IR ELEKTRONIKOS FAKULTETAS

ROBERTAS KATKUS

MOBILAUS AUTONOMINIO ROBOTO TRAJEKTORIJOS PLANAVIMAS PAVOJINGOJE, DALINAI ŽINOMOJE APLINKOJE, SIEKIANT IŠVENGTI SUSIDŪRIMO SU KLIŪTIMIS

Baigiamasis magistro projektas

Vadovas
lekt. Gintautas Narvydas

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS
ELEKTROS IR ELEKTRONIKOS FAKULTETAS
ELEKTROS ENERGETIKOS SISTEMŲ KATEDRA

MOBILAUS AUTONOMINIO ROBOTO TRAJEKTORIJOS PLANAVIMAS PAVOJINGOJE, DALINAI ŽINOMOJE APLINKOJE, SIEKIANT IŠVENGTI SUSIDŪRIMO SU KLIŪTIMIS

Baigiamasis magistro projektas

VALDYMO TECHNOLOGIJŲ (621H66001) 2 metų studijų programa

Vadovas
lekt. Gintautas Narvydas
2015 m.

Recenzentas
_____
2015 m.

Projektą atliko
Robertas Katkus
2015 m.

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Elektros ir elektronikos fakultetas

(Fakultetas)

Robertas Katkus

(Studento vardas, pavardė)

Valdymo technologijos, 621H66001

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Mobilaus autonominio roboto trajektorijos planavimas pavojingoje, dalinai žinomoje aplinkoje, siekiant išvengti susidūrimo su kliūtimis“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

2015 m. gegužės m. 21 d.

Kaunas

Patvirtinu, kad mano, **Roberto Katkaus,** baigiamasis projektas tema „Mobilaus autonominio roboto trajektorijos planavimas pavojingoje, dalinai žinomoje aplinkoje, siekiant išvengti susidūrimo su kliūtimis“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

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

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

Katkus Robertas. Mobilaus autonominio roboto trajektorijos planavimas pavojingoje, dalinai žinomoje aplinkoje, siekiant išvengti susidūrimo su kliūtimis. Valdymo sistemų magistro baigiamasis projektas / vadovas lekt. Gintautas Narvydas. Kauno technologijos universitetas, Elektros ir elektronikos fakultetas, Elektros Energetikos Sistemų katedra.

Kaunas, 2015 m. 72 psl.

SANTRAUKA

Mobilaus roboto judėjimui aplinkoje reikia žinoti kelią, kuriuo jis turi judėti. Dažniausiai reikalinga žinoti trumpiausią kelią. Tam yra sukurta nemažai įvairių kelio paieškos algoritmų, bei kuriami nauji ir modifikuojami jau esantys sukurti.

Šiame darbe išnagrinėsime dažniausiai naudojamus trumpiausio kelio paieškos algoritmų, juos palyginsime. Po atliktos algoritmų analizės pasirinksime 2 algoritmus, juos realizuosime MATLAB programiniu paketu, palyginsime jais surastų kelių paieškos rezultatus. Pasirinkę vieną iš šių dviejų algoritmų, atliksime važiavimus su realiu *E-Puck* mobiliu autonominiu robotu sumodeliuotoje aplinkoje, surasdami trumpiausią kelią taip, kad robotas juo važiuodamas išvengtų susidūrimo su kliūtimis.

Raktiniai žodžiai – A* algoritmas, D*Lite algoritmas, trumpiausio kelio paieška, kelio perplanavimas.

Katkus Robertas. Path planning of an autonomous mobile robot to avoid the risk of collision with obstacles in a dangerous, partially known environment. Final project of master of control systems / supervisor lect. Gintautas Narvydas. Kaunas University of Technology, Faculty of Electrical and Electronics Engineering, Department of Electrical Power Systems.
Kaunas, 2015 m. 72 pgs.

SUMMARY

Mobile robot movement in the environment need to know the way he has to move. Most often necessary to know the shortest way. There are a number of very different path search algorithms, as well as the new ones and modified already in build.

In this paper we will the most commonly used shortest path search algorithms to compare them. After analysis of algorithms, we will choose 2 algorithms to realize them with Matlab software to compare the results of path planning. We will choose one of these algorithms to drive with real autonomous mobile robot E-Puck in simulated environment, by finding the shortest path in a way that robot will avoid collision with obstacles while driving it.

Keywords – A* algorithm, D*Lite algorithm, shortest path planning, path replanning.

TURINYS

Įvadas

Pastaruoju metu yra nemažai eksperimentuojama su mobiliais autonominiais robotais, siekiant juos pritaikyti kasdienėje žmogaus veikloje, pramonėje. Yra sukurta savaeigių robotų, tačiau dažno jų judėjimas yra valdomas operatoriaus. Mobilus robotas turi būti toks, kuris aplinkoje orientuotųsi, judėtų be žmogaus įsikišimo, vien savo įrangos – jutiklių, vykdiklių – pagalba arba žmogaus įsikišimas būtų minimalus – jis turi nurodyti tikslo, paskirties vietą robotui. Tokius robotus galima pritaikyti dirbti sandėliuose, prekybos centruose, gatvėse - to pavyzdys būtų „Google Car“ projektas, taip pat patalpose, kur tam tikrais momentais negali būti žmogaus (pvz., įvykus dujų nuotėkiui, sprogimui ir pan.). Jie galėtų atlikti stebėtojų vaidmenį, surinkti duomenis (vaizdinę medžiagą), arba nuvažiavę į nurodytą vietą atlikti tam tikrą veiksmą. Judėjimas robotą supančioje aplinkoje/patalpoje gali būti įvairus – robotas gali važiuoti neturėdamas paskirties vietos, pvz., ieškodamas kokio nors objekto, arba turi nuvažiuoti į konkrečią nurodytą vietą. Šiame darbe nagrinėsime pastarąjį būdą, kai judama iš vienos paskirties vietos į kitą - judama iš taško A į tašką B.

Kiekvienam įrenginiui, kurio paskirtis yra nusigauti iš taško A į tašką B, yra svarbi navigacija erdvėje, kurioje jis juda. Optimalios trajektorijos planavimas apima ne tik fizinių kliūčių išvengimą, bet ir nesaugių terpių ar nepageidaujamų sąlygų apėjimą. Kad tai padarytų, robotas turi žinoti savo padėtį erdvėje naudojamos atskaitos sistemos atžvilgiu.

Atliekant užduotį svarbu žinoti, kokiu tikslumu robotas turi vykdyti navigaciją erdvėje. Žinoma, šie reikalavimai labiausiai priklauso nuo paties roboto pritaikymo ar konkrečios atliekamos užduoties, tačiau pirminis įvertinimas gali būti toks: norėdamas nustatyti savo padėtį erdvėje, robotas turėtų judėti bent jau savo paties matmenų tikslumu. Pagal mastelį ir atskaitos sistemą, navigacija gali būti klasifikuojama:

1. Globali navigacija - tai gebėjimas nustatyti padėtį absoliutinėje (žemėlapio) atskaitos sistemoje, ir pasiekti pageidautiną tikslą joje.
2. Lokali navigacija - tai gebėjimas nustatyti savo padėtį supančių objektų atžvilgiu, ir su jais sąveikauti.
3. Personalinė navigacija - tai žinojimas kur ir kokioje padėtyje viena kitos atžvilgiu yra dalys, sudarančios patį robotą.

Šių skirtingų mastelių tikslai taip pat yra skirtingi. Globali navigacija skirta judėjimui tarp galinių (kraštinių) vietovių, kai tuo tarpu lokali navigacija koncentruojasi į užduočių atlikimą tose vietovėse. Personalinė navigacija naudojama roboto ir su juo kontaktuojančių objektų stebėsenai.

Judėjimui iš taško A į tašką B reikia žinoti tikslas koordinates, bei trajektoriją/kelią, kuriuo robotas turės judėti. Kelio planavimui yra sukurta daug įvairių kelio paieškos algoritmų. Algoritmai pradėti kurti XX-o amžiaus 6-e dešimtmetyje, šiandien dažniausiai yra tobulinami jau esantys sukurti algoritmai, siekiant juos optimizuoti, taip pat kombinuojama po keletą algoritmų į vieną, siekiant panaudoti kiekvieno jų geriausias savybes.

Vieni kelio paieškos algoritmų remiasi paieškomis grafuose, kiti paremti tinklelio (*„grid search“*) logika. Grafai dažniau naudojami maršrutų sudarymo uždaviniuose, pvz., marštuto parinkimui mieste navigaciniame prietaise. Tinklelio (*„grid search“*) logika naudojama uždaviniuose, kuriuose nėra persidengiančių paviršių ir pan. Robotikoje sprendžiant įvairius uždavinius dažniausiai reikia rasti trumpiausią kelią tarp taškų A ir B (algoritmuose jie dažnai vadinami mazgais). Vieni algoritmai kelią randa greitai, kiti lėčiau, tačiau gali rasti kelis trumpiausio kelio variantus. Kiekvienam sprendžiamam uždaviniui yra svarbu pasirinkti jam tinkamą kelio paieškos algoritmą. Dažnai reikia įvertinti ne vien tai, kaip greitai ar tiksliai jis suranda kelią, tačiau, jei roboto kelyje yra aptinkama iš anksto nenumatytų kliūčių, kaip greitai yra perplanuojamas naujas kelias.

Šiame darbe nagrinėjamus kelio paieškos algoritmus galima suskirstyti į tris grupes:

1. Pilno perrinkimo algoritmai. Tai algoritmai, kelio paieškai ištiriantys visą žemėlapį ar grafą. Tai vieni pirmųjų sukurtų algoritmų.
2. Euristiniai algoritmai. Tai algoritmai, paremti paieškų metodais, vedančiai arčiausiai tikslo, dažniausiai jie ištiria tik dalį žemėlapio.
3. Gamtos dėsniais paremti algoritmai. Tai algoritmai, sukurti remiantis dėsniais, vykstančiais gyvojoje gamtoje, tokie kaip genetiniai algoritmai, kolonijiniai algoritmai.

Šio darbo tikslas – išnagrinėti po keletą kiekvienos grupės algoritmų, išanalizuoti jų veikimo principus, juos palyginti, vėliau, pasirinkus porą tinkamiausių algoritmų, juos realizuoti MATLAB programiniu paketu, pritaikyti ir išbandyti kelio planavimą su realiu autonominiu mobiliu robotu E-PUCK.

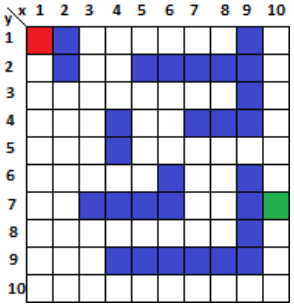
1. PROBLEMOS ANALIZĖ

1.1. Kelio paieškos metodai

Šiame darbe nagrinėjamuose kelio paieškos algoritmuose tiek grafo, tiek tinklelio tipo žemėlapiuose dažniausiai vadovaujamasi keliais paieškos principais, kitaip metodais. Jie apibūdina, kokia tvarka parenkami mazgai, kaip ir kokia kryptimi judama tikslo mazgo link. Šie paieškų tipai yra vadinami paieškos į plotį [4,25], paieškos į gylį [5, 6,25], geriausios pirmos paieškos metodais. Šiuos metodus galima laikyti ir analizuoti kaip atskirus paieškos algoritmus, tačiau didelė dalis paieškos algoritmų vadovaujasi vienu ar keliais iš šių trijų metodų. Šiuos metodus aprašysime 1.2, 1.3, 1.4 skyriuose.

1.2. Paieškos į plotį metodas

Turime aplinkos su kliūtimis tinklelio tipo žemėlapį, pateiktą 1 paveiksle. Jame mėlyni langeliai reiškia kliūtis (sienas), o baltais langeliais galima judėti iš vieno į kitą 8 kryptimis: aukštyn, žemyn, kairėn, dešinėn, bei įstrižai. Raudonas langelis – startinis (pradinis) mazgas, žalias – tikslo (galutinis) mazgas. Reikia rasti kelią iš pradinio į galutinį mazgą.

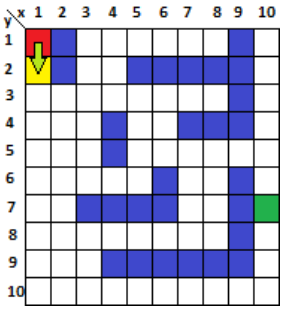


1 pav. Tinklelio tipo žemėlapis su kliūtimis

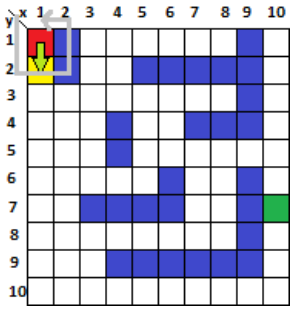
Aprašysime kaip veikia paieškos į plotį metodas. Nurodomas pradinis kelio taškas, šiuo atveju tai yra taškas, kurio koordinatės y = 1, x = 1 (kadangi žemėlapis iš esmės yra stačiakampė matrica, patogumo dėlei y koordinatė bus vietoje x pagal matricų adresavimą – pirma eilutės, po to stulpelio numeris). Atliktų iteracijų skaitliukas nustatomas lygus 0. Kitas žingsnis – sudaromas mazgų sąrašas, kuriuos norima/galima išplėsti, t.y., mazgų sąrašas, iš kurių galima patekti į naujus, dar netirtus, langelius (vėliau matysime, kad tokie ir panašūs sąrašai yra sudaromi dažnam kelio paieškos algoritmui). Paieškos pradžioje sąraše yra tik pradinis mazgas. Mazgų sąrašas atrodo taip:

y1x1

Toliau vyksta tikrinimas , ar šis mazgas turi bent vieną kaimyną, kuris nėra kliūtis. Iš žemėlapio matome, kad vienintelis toks mazgas yra y2x1. Mazgai, į kuriuos galima patekti, pažymimi geltonai (žr. 2 pav.):



2 pav. Mazgų žymėjimas paieškos žemėlapyje



3 pav. Kaimyninių mazgų tikrinimo tvarka

Prieš pradedant paiešką būtina nustatyti, kokia tvarka tikrinami kaimyniniai mazgai. Vienas iš būdų yra toks: imama kryptis prieš laikrodžio rodyklę, pradedant nuo viršutinio kairiojo mazgo (žr. 3 pav.):

Jei šis mazgas nėra tikslo mazgas, tuomet jis yra įtraukiamas į mazgų sąrašą, o iš sąrašo yra išmetamas mazgas y1x1. Išmetimas vyksta FIFO (First IN, First OUT) principu, t.y., pirmesnis atsidūręs sąraše, pirmesnis yra ir pašalinamas. Išmestą mazgą reikia įtraukti į išmestų, kitaip sakant jau aplankytų, mazgų sąrašą tam, kad jie nebūtų tikrinami daugiau kaip vieną kartą (išmesti mazgai žymimi raudonai).

Dabar turime sąrašus:

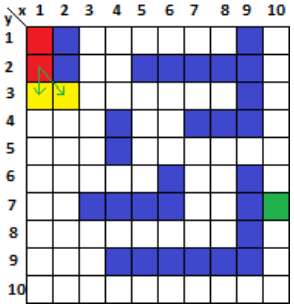
1) Tiriamųjų mazgų sąrašas

~~y1x1~~ y2x1

2) Išmestųjų (aplankytų) mazgų sąrašas

y1x1

Atlikta viena iteracija. Kitas žingsnis – iš eilės tikrinami likę sąraše mazgai, ar jie turi laisvų kaimynų. Matome, kad mazgas y2x1 turi du kaimynus – y3x1 bei y3x2 (žr. 4 pav.):



4 pav. Paieškos į plotį antroji kaimyninių mazgų tikrinimo iteracija

Nei vienas iš šių mazgų nėra tikslo mazgas, tai išmetimo/papildymo procedūra pakartojama – į sąrašą papildomai įtraukiami nauji mazgai y3x1 ir y3x2, o išbraukiamas y2x1, pastarasis papildomas sąraše jau aplankytų mazgų. Po šios iteracijos sąrašai atrodo taip:

1) Tiriamųjų mazgų sąrašas

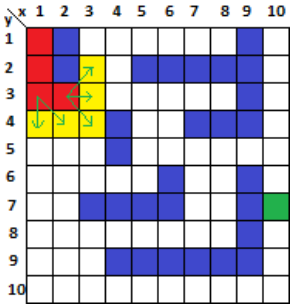
~~y2x1~~ y3x1 y3x2

2) Išmestųjų mazgų sąrašas

y1x1| y2x1 (y1x1)

Mazgai sąraše mazgai atskirti simboliu |, taip atskiriant iteracijas, per kurias jie papuolė į sąrašą. Šalia išmesto mazgo skliausteliuose įrašomas jo pirmtakas (jei toks buvo).

Kitoje, jau trečioje, iteracijoje, tikriname paeiliui sąraše esančius mazgų laisvus kaimynus. Pastaba – kaimyninis mazgas gali būti priskirtas tik vienam pirminiam mazgui, pirmenybė teikiama tam, kuris sąraše yra pirmasis. Radus kaimynus, jie eilės tvarka surašomi į sąrašą, o pirminiai mazgai iš jo eilės tvarka išmetami. Dar viena pastaba – per vieną iteraciją tikrinami ir išmetami iš sąrašo praeitoje iteracijoje papildomai įtraukti kaimyniniai mazgai. Mazgas gali būti išmetamas, net jei ir neturi nei vieno laisvo kaimyno. Matome, kad mazgas y3x1 turi 2 laisvus kaimynus , išdėstytus tokia eilės tvarka: y4x1, y4x2. Mazgas y3x2 turi 3 kaimynus: y4x3, y3x3, y2x3 (žr. 5 pav):



5 pav. Paieškos į plotį trečioji kaimyninių mazgų tikrinimo iteracija

Atnaujiname sąrašus:

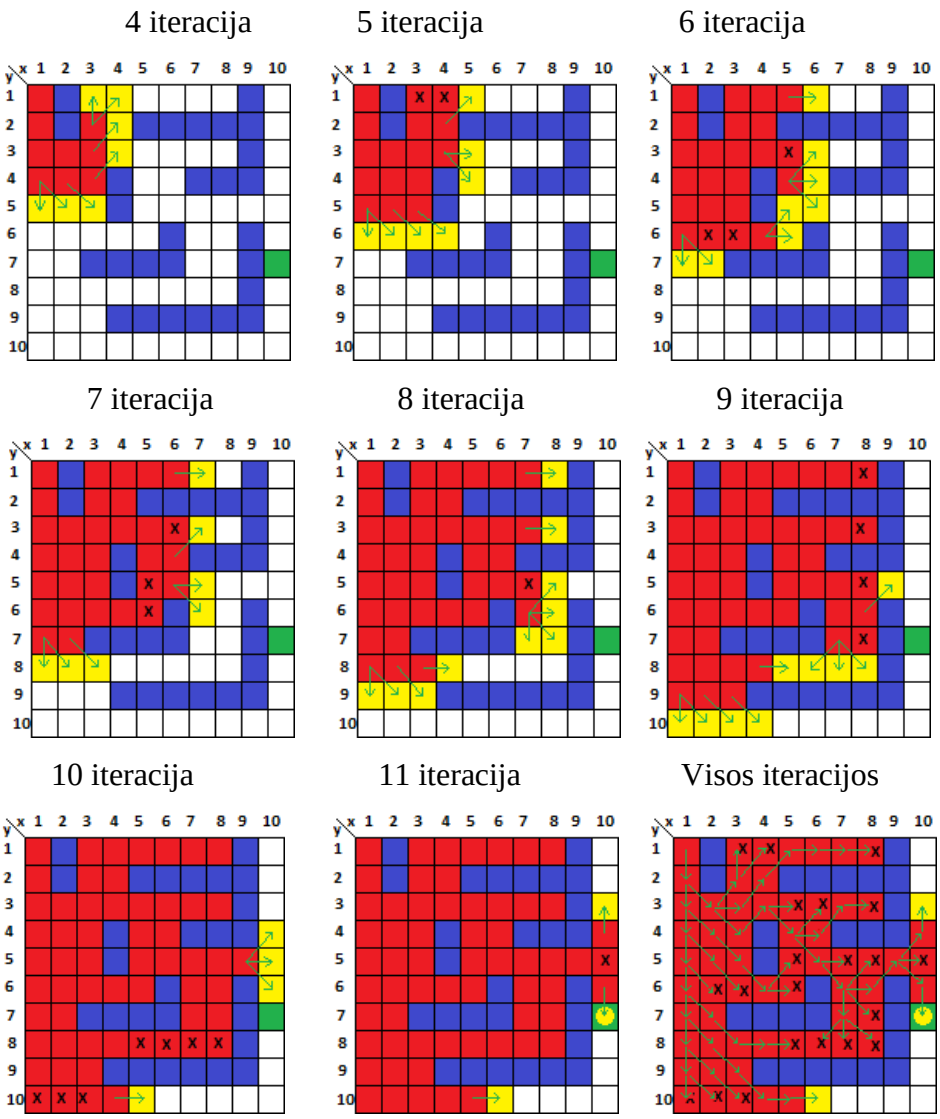
1) Tiriamų mazgų sąrašas

~~y3x1 y3x2~~ y4x1 y4x2 y4x3 y3x3 y2x3

2) Išmestų mazgų sąrašas

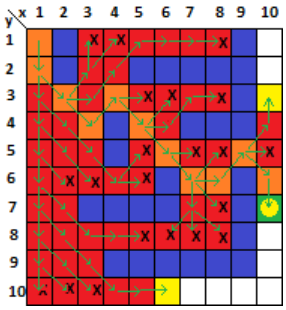
y1x1| y2x1| y3x1(y2x1) y3x2(y2x1)

Toliau veiksmų neaprašinėsime, tik pavaizduosime grafiškai (X pažymėti mazgai, neturintys laisvų kaimynų, tačiau iš sąrašo vis tiek išmetami). Tęsiame tol, kol nei vienam mazgui nebegalima rasti laisvų kaimynų, arba kol vienas iš kaimynų sutampa su tikslo tašku (žr. 6 pav.):



6 pav. Paieškos į plotį kaimyninių mazgų tikrinimo iteracijos

11-oje iteracijoje matome, kad mazgo y5x10 kaimynas y6x10 yra tikslo taškas. Paieška yra baigiama, lieka surasti t.y. apskaičiuoti kelią iš pradinio į galutinį tašką. Tam reikia išmestųjų mazgų sąrašo ir kiekvienoje iteracijoje surasti, koks buvo tikrinamo mazgo pirminis mazgas (pirmtakas). Juos rasti nesunku, kadangi kiekvienam mazgui pirminis gali būti tik vienas. Žemėlapyje tai galima atvaizduoti paprastai (žr. 7 pav. oranžinės spalvos langeliai):



7 pav. Paieškos į plotį algoritmo pirminiai mazgai

Vizualiai matyti, kad rastasis kelias nėra optimalus, tačiau bet kokiu atveju, jei kelią rasti galima, paieškos į plotį algoritmas jį suranda.

1.3. Paieškos į gylį metodas

Paieškos į gylį metodas nuo paieškos į plotį skiriasi tuo, kad visada kaimynų yra ieškoma paskutiniam mazgui tiriamų mazgų sąraše. Vadovaujamasi principu LIFO (Last IN, First OUT). Kaimyninių mazgų ieškojimo aplink mazgą kryptis bei papildymo į sąrašą eilės tvarka yra tokia pati (arba gali būti), kaip ir paieškos į plotį algoritme. Jei mazgas nebeturi laisvų kaimynų, grįžtama prie priešpaskutinio kaimyno, kuris dar nebuvo tirtas. Ištirti mazgai taipogi yra išmetami iš tiriamų mazgų sąrašo. Tarkime, turime tą patį žemėlapį su tais pačiais pradžios ir tikslo mazgais, kaip ir paieškos į plotį atveju. Pradedame pirmą iteraciją. Į sąrašą įtraukiamas mazgas y1x1. Šiuo atveju pradinis mazgas y1x1 turi tik vieną laisvą kaimyną y2x1. Jis įtraukiamas į sąrašą, iš sąrašo išbraukiamas mazgas y1x1 ir įtraukiamas į išmestų mazgų sąrašą. Mazgas y2x1 turi 2 kaimynus: y3x1 ir y3x2. Šie mazgai įtraukiami į sąrašą, iš jo išbraukiamas mazgas y2x1 ir įtraukiamas į išmestųjų mazgų sąrašą. Turime tokius mazgų sąrašus:

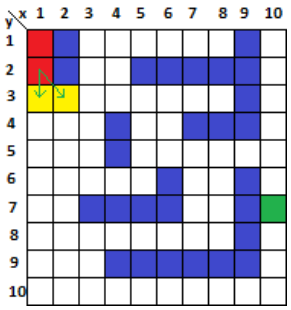
1) Tiriamųjų mazgų sąrašas

y3x1 y3x2

2) Išmestųjų mazgų sąrašas

y1x1 | y2x1 (y1x1)

Nueitas kelias pateiktas 8 paveiksle:



8 pav. Paieškos į gylį antroji kaimyninių mazgų tikrinimo iteracija

Kitoje, 3-oje, iteracijoje, paieškos algoritmas skiriasi nuo paieškos į plotį algoritmą, kadangi kaimynų yra ieškoma tik paskutiniam į tiriamų mazgų sąrašą įtrauktam mazgui. Šiuo atveju tai yra mazgas y3x2. Jo kaimynai (eilės tvarka): y4x1, y4x2 , y4x3, y3x3 ir y2x3. Šie mazgai eilės tvarka papildo tiriamų mazgų sąrašą, mazgas y3x2 išmetamas iš jo ir įtraukiamas į išmestųjų mazgų sąrašą. Turime tokius sąrašus:

1) Tiriamųjų mazgų sąrašas

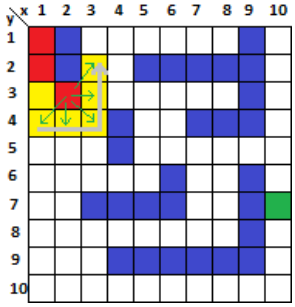
y3x1 ~~y3x2~~ y4x1 y4x2 y4x3 y3x3 y2x3

2) Išmestųjų mazgų sąrašas

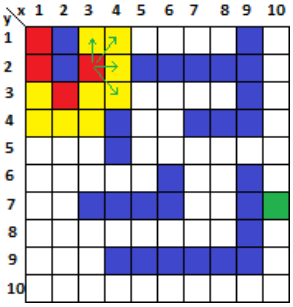
y1x1 y2x1(y1x1) y3x2 (y2x1)

Mazgų iš sąrašų išmetimas/papildomas parodytas 9 pav.

Paskutinis tiriamųjų mazgų sąraše esantis mazgas yra y2x3, taigi, kaimynų ieškoma tik jam. Jo kaimynai eilės tvarka (žr. 10 pav.): y3x4, y2x4, y1x4, y1x3. Mazgas y2x3 išmetamas iš sąrašo, juo papildomas išmestųjų mazgų sąrašas.



9 pav. Paieškos į gylį trečioji kaimyninių mazgų tikrinimo iteracija



10 pav. Paieškos į gylį ketvirtoji kaimyninių mazgų tikrinimo iteracija

Tiriamųjų mazgų sąrašas jau yra toks:

y3x1 y4x1 y4x2 y4x3 y3x3 ~~y2x3~~ y3x4 y2x4 y1x4 y1x3

Išmestųjų mazgų sąrašas:

y1x1 y2x1(y1x1) y3x2 (y2x1) y2x3 (y3x2)

Tirdami mazgą y1x3 matome, kad jis neturi nei vieno laisvo kaimyno. Jis irgi išmetamas iš tiriamų mazgų sąrašo. Sąrašai tampa tokie:

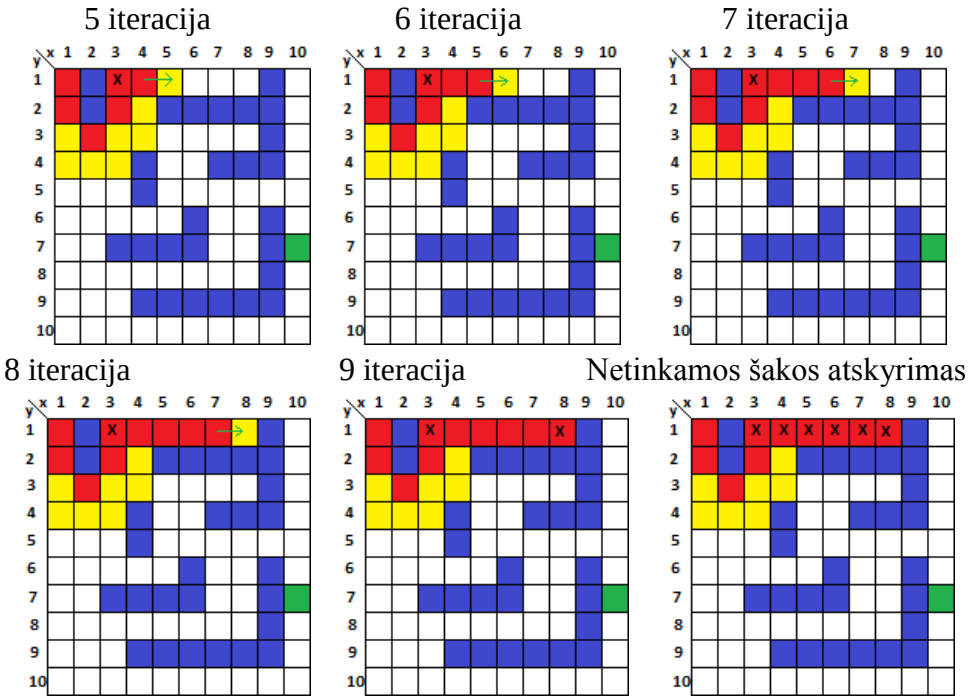
1) Tiriamųjų mazgų sąrašas

y3x1 y4x1 y4x2 y4x3 y3x3 y3x4 y2x4 y1x4 ~~y1x3~~

2) Išmestųjų mazgų sąrašas

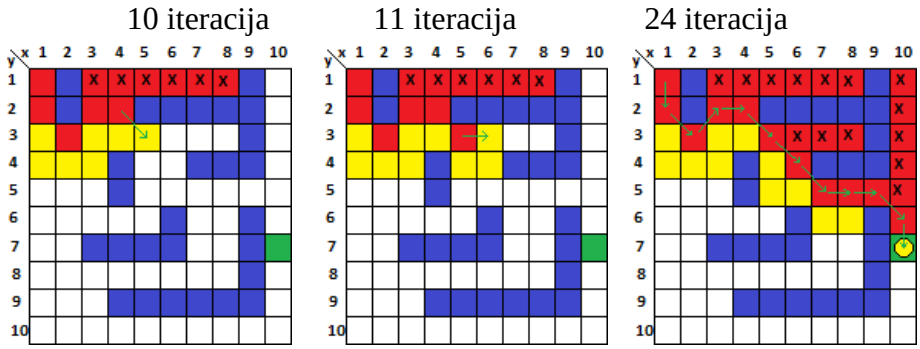
y1x1 y2x1(y1x1) y3x2 (y2x1) y2x3 (y3x2) y1x3 (y2x3)

Toliau tiriamas paskutinis esantis mazgas y1x4. Tolesnių veiksmų neaprašinėsime, tik pavaizduosime 11 paveiksle:



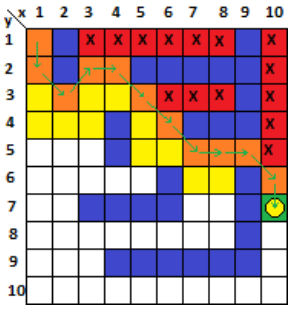
11 pav. Paieškos į gylį kaimyninių mazgų tikrinimo iteracijos

9-oje iteracijoje tiriamas mazgas y1x8 neturi nei vieno laisvo kaimyno, šioje šakoje kelio paieška baigiama. Sekančiam paieškos etapui imamas paskutinis esantis mazgas tiriamųjų sąraše, šiuo atveju y2x4 ir paieška tęsiama toliau (žr. 12 pav., 10-a iteracija):



12 pav. Paieškos į gylį kaimyninių mazgų tikrinimo iteracijos po netinkamų šakų atskyrimo

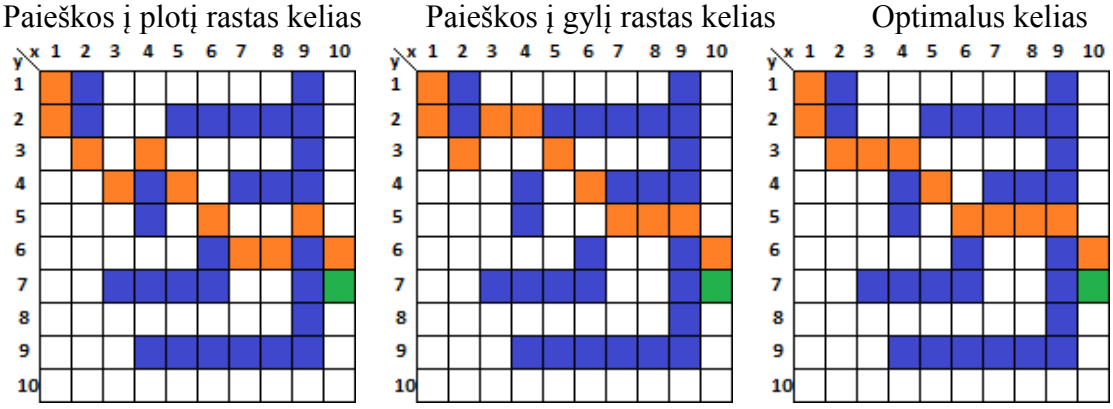
Matome, kad 24-os iteracijos metu pasiekiame tikslo mazgą. Kelias randamas tuo pačiu principu: keliaujama nuo tikslo mazgo iki pradinio, tikrinant mazgo pirmtakus. Rastas kelias, pažymėtas oranžine spalva, pavaizduotas 13 pav.



13 pav. Paieškos į gylį algoritmo pirminiai mazgai

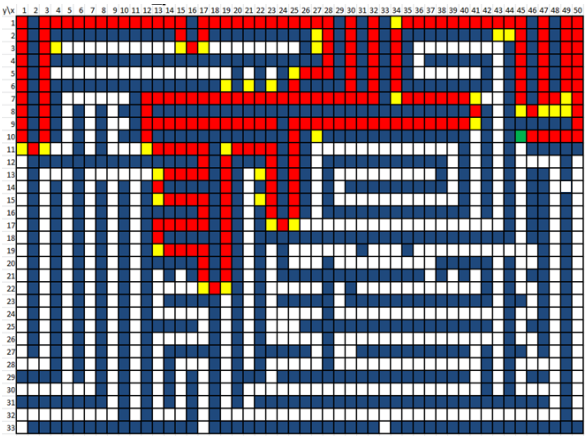
Matome skirtumą tarp paieškos į plotį ir paieškos į gylį algoritmų, pastarojo metu, nors ir per didesnį iteracijų skaičių, patikrinta žymiai mažiau mazgų, tuo pačiu buvo sudaryti ir mažesni mazgų sąrašai, dėl to paieškos į gylį algoritmui reikia mažiau atminties.

Abiem algoritmais-metodais surastas kelias nėra optimalus, tai tiesiog yra pirmasis surastas kelias. Šiais algoritmais surastų kelių palyginimas su optimaliu keliu pateiktas 14 pav.

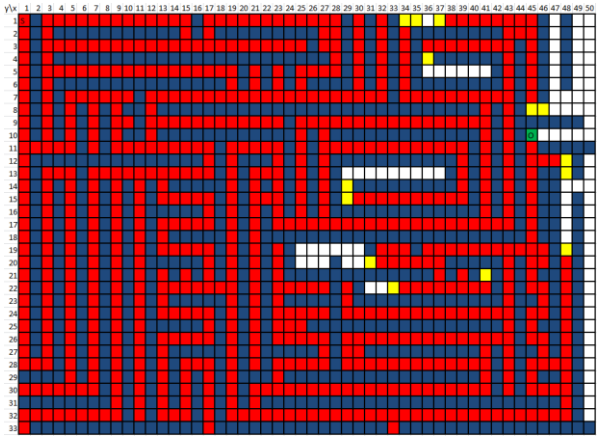


14 pav. Paieškų į plotį ir į gylį surastų kelių palyginimas su optimaliu keliu

Palyginimui paieškos į plotį ir paieškos į gylį metodais tiriamų mazgų skaičius didesniame žemėlapyje (raudonai pažymėti taškai, kurie turėjo būti ištirti) pateikiamas 15 ir 16 paveiksluose:



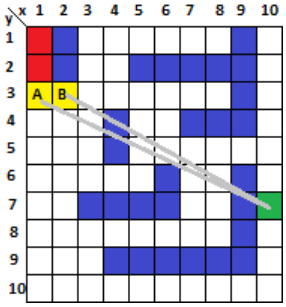
15 pav. Paieškos į gylį algoritmo tiriamų mazgų skaičius



16 pav. Paieškos į plotį algoritmo tiriamų mazgų skaičius

1.4. Geriausios pirmosios paieškos metodas

Paieškos algoritmuose galima rasti vartojamą sąvoką „geriausia pirmoji paieška“ [26]. Tai paieškos algoritmas, kuris tyrinėja grafą išplėsdamas mazgą (parinkdamas jam kaimyną) ta kryptimi, kuri atrodo teisingiausia, t.y., veda arčiausiai tikslo. Tokio kaimyninio mazgo parinkimas paprastai nustatomas taisyklėmis, taipogi iš tam tikro mazgų sąrašo (analogiškai kaip ir paieškos į plotį bei gylį algoritmuose), pagal sudėliotus prioritetus. Taisyklė dažniausiai būna tam tikra euristinė atstumo funkcija f(x,y) nuo tiriamojo iki tikslo mazgo. Grafiškai tai galima pavaizduoti taip: galimi du mazgo y2x1 išplėtimai – mazgas A (y3x1), mazgas B (y3x2). Tarkime, duotoji funkcija f(x,y) yra euklidinio atstumo iki tikslo mazgo funkcija. Atstumas nuo mazgo A iki tikslo mazgo: $f(x,y)=\sqrt{(10-1)^2+(7-3)^2}=\sqrt{81+16}=9.848$, nuo taško B iki tikslo mazgo: $f(x,y)=\sqrt{(10-2)^2+(7-3)^2}=\sqrt{64+16}=9.486$. Matome, kad atstumas iki tikslo iš taško B yra trumpesnis. Šio tipo paieškos algoritmas parinktų tolimesnę kelio kryptį nuo B mazgo, kadangi jis veda arčiau tikslo nei A mazgas. Tai pavaizduota 17 paveiksle:



17 pav. Geriausios pirmosios paieškos veikimo principas

Kartais šis paieškos algoritmas (tuo pačiu ir metodas) dar vadinimas „gobšiu“ (angl. *greedy*) geriausios pirmos paieškos algoritmu, nes iš kelių galimų pasirinkti kelio variantų ar mazgų išplėtimo krypties pirmiausia pasirenkamas tą (tas), kuris prognozuos didžiausią naudą, t.y., didžiausią priartėjimą prie tikslo.

Tokio algoritmo pavyzdys yra A* trumpiausio kelio paieškos algoritmas.

Sekančiuose skyriuose aptarsime dažniausiai naudojamus kelios paieškos algoritmus. Daugumos jų aprašymuose bus minima grafo, grafo mazgo, briaunų sąvokos. Taip yra todėl, kadangi šie algoritmai pirmiausia ieško kelio grafuose, tačiau sėkmingai gali būti pritaikyti ir tinklelio tipo žemėlapiams, juolab, kad iš tinklelio tipo žemėlapio galima sudaryti grafą arba medį. Taip pat kaip minėjome įvade, daugumą šių algoritmų pagal paieškos būdą, žemėlapio ištyrimo dalį, galima suskirstyti į kelias atskiras grupes:

- Algoritmai, ieškantis kelio ištirdami visus mazgus (pilno perrinkimo algoritmai)
- Euristiniai algoritmai
- Gamtos ir pan. dėsniais paremti algoritmai

Algoritmus panagrinėsime atskiromis grupėmis, vėliau juos palyginsime. 1.5 – 1.7 skyriuose aptarsime pilno perrinkimo, 1.8 – 1.13 skyriuose euristinius, 1.14 – 1.16 – gamtos ir pan. dėsniais paremtus algoritmus.

1.5. Dijkstra kelio paieškos algoritmas

Dijkstra algoritmas, 1956 m. sugalvotas ir 1959 m. paskelbtas informatiko Edsger Dijkstra, yra grafų paieškos algoritmas, sprendžiantis trumpiausio kelio paieškos problemą grafui su neneigiama kelio kaina, sukurdamas trumpiausio kelio medį. Jis dažnai naudojamas kaip geriausio kelio paieškos algoritmas kituose grafų algoritmuose [7].

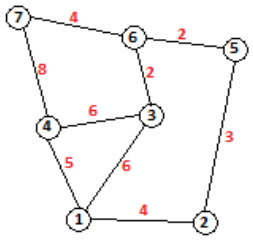
Algoritmas veikia tokiu principu:

Mazgas, iš kurio pradedama kelio paieška, yra pradinis mazgas. Įvesime 2 parametrus: mazgo atstumas nuo pradinio taško, ir būsenos parametras, pažymintis, ar mazgas jau aplankytas. Toliau Dijkstra algoritmas vykdomas tokia seka:

1. Kiekvienam mazgui priskiriama pradinė atstumo vertė – pradiniam mazgui 0, likusiems begalybė.
2. Visi mazgai pažymimi kaip neaplankyti. Pradinis mazgas pažymimas einamuoju. Sukuriama neaplankytų mazgų aibė, į kurią įtraukiami visi mazgai.

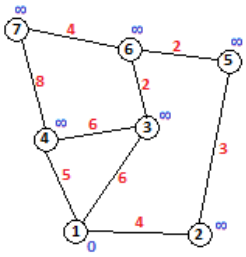
3. Iš einamojo mazgo patikrinami visi galimi mazgo kaimynai ir suskaičiuojami pirminiai atstumai iki jų. Gauti atstumai palyginami su jau priskirtomis kaimyniniams mazgams vertėmis, ir yra įrašoma mažesnioji vertė.
4. Kai apskaičiuojami atstumai iki visų kaimyninių mazgų, einamasis mazgas pažymimas kaip aplankytas ir yra pašalinamas iš neaplankytųjų mazgų aibės. Antrą kartą jis nebebus tikrinamas.
5. Jei tikslo mazgas pažymimas kaip aplankytas, arba mažiausia kelio iki mazgo vertė tarp neaplankytųjų mazgų aibėje esančių mazgų yra begalybė (šiuo atveju reiškia, kad kelio nėra), algoritmas baigiamas.
6. Iš neaplankytųjų mazgų aibės pasirenkamas mazgas, kurio vertė mažiausia, jis pažymimas einamuoju, grįžtama į 3 punktą.

Kad būtų aiškiau, panagrinėkime šį algoritmą detaliau. Turime grafą su mazgais, svoriais (svoris šiuo atveju – neneigiamas atstumas tarp mazgų, pažymėtas raudonai) ir jungtimis tarp mazgų, kaip pateikta 18 paveiksle:



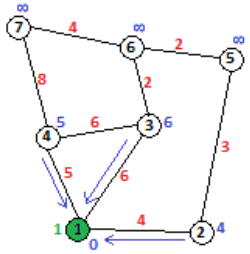
18 pav. Kelio paieškos Dijkstra algoritmu grafas

Tikslas – rasti trumpiausią kelią iš mazgo Nr.1 į mazgą Nr.7. Pagal algoritmą visi mazgai laikomi neaplankytais ir įtraukiami į neaplankytųjų mazgų aibę, visiems mazgams priskiriama kelio vertė ∞ (tai reiškia, kad kelio ilgis iki mazgo nuo pradinio nėra žinomas), išskyrus pradinį – jam priskiriama reikšmė 0 (žr. 19 pav.).



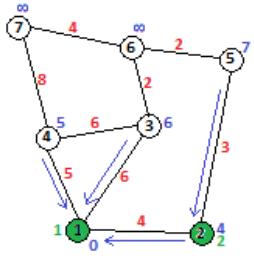
19 pav. Kelio paieškos Dijkstra algoritmu pradinis žingsnis

Toliau tikrinami visi mazgai, kuriuos galima pasiekti iš mazgo Nr.1. Tai mazgai Nr.2, Nr.3, Nr.4. Šiems mazgams reikia apskaičiuoti kelio kainą iki jų – tai yra atstumas nuo to mazgo, iš kurio šiuo momentu į juos ateinama, + to mazgo kelio kainos vertė. Mūsų atveju mazgų kelio kaina bus atstumas nuo mazgo Nr.1 + šio mazgo vertė, kuri lygi nuliui. Gautos vertės priskiriamos mazgams, kadangi jos mažesnės nei esamos priskirtos (< ∞). Priskyrus pažymima, iš kurio mazgo ateinama (kuris mazgas yra pirminis, mėlynos rodyklės 19 pav.). Patikrinama, ar patikrinti visi mazgai, kurie siejasi su mazgu Nr.1. Šiuo atveju patikrinti visi su juo susiję mazgai, mazgas Nr.1 pažymimas kaip aplankytas (pažymėsime žaliai) ir išmetamas iš aplankytųjų mazgų sąrašo (žr. 20 pav.). Galima prie jo pažymėti, kelintas eilės tvarka jis išmestas iš šio sąrašo (žalios spalvos vienetukas šalia mazgo):



20 pav. Kelio paieškos Dijkstra algoritmu aplankytas pirmasis taškas

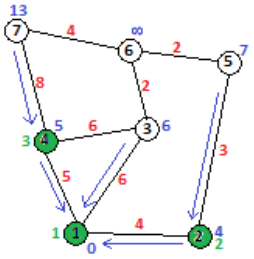
Kitame žingsnyje iš neaplankytų mazgų sąrašo imamas mazgas su einamuoju momentu mažiausia verte, tai yra mazgas Nr.2 (žr. 21 pav.). Tikriname, kuriuos mazgus iš jo galima pasiekti. Tai mazgas Nr.5. Apskaičiuojame kelio kainą iki šio mazgo, tai bus 4 (mazgo Nr.2 vertė) + 3 (atstumas tarp mazgų Nr.2 ir Nr.5) = 7. Priskiriame šią vertę mazgui Nr.5, pažymime, iš kurio mazgo į jį ateinama, mazgas 2 nurodomas kaip aplankytas (kadangi visi su juo susieti mazgai ištirti) ir išmetamas iš neaplankytų mazgų sąrašo:



21 pav. Kelio paieškos Dijkstra algoritmu aplankytas antrasis taškas

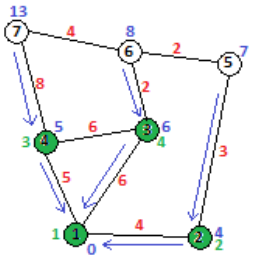
Sekantis neaplankytų mazgų sąraše mažiausią vertę turi mazgas Nr.4. Iš jo galima patekti į mazgus Nr.7 (tikslo mazgas) bei Nr.3 (žr. 22 pav.). Apskaičiuojame kelio kainas iki jų, gauname atitinkamai iki mazgo Nr.7 5+8=13, iki mazgo Nr.3 5+6=11. Mazgams Nr.7 ir Nr.3 priskiriame

apskaičiuotas vertes. Kadangi mazgas Nr.3. jau turi priskirtą vertę, nelygią ∞, tikriname, ar naujoji vertė mažesnė už esamą. Šiuo atveju 11 nėra mažiau už 6, taigi, mazgui Nr.3 paliekame priskirtą vertę 6 ir krypties, žyminčios, iš kurio mazgo į jį ateinama, nekeičiame. Mazgą Nr.4 pažymime kaip aplankytą ir išmetame iš jau aplankytų mazgų sąrašo:



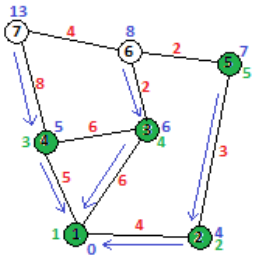
22 pav. Kelio paieškos Dijkstra algoritmu aplankytas trečiasis taškas

Matome, tikslo mazgas Nr.7 yra pasiektas, tačiau dar yra neaplankytų mazgų, todėl negalime teigti, kad radome trumpiausią kelią. Kitas mazgas su mažiausia verte yra mazgas Nr.3 (žr. 23 pav.). Iš jo galima pateikti į mazgą Nr.6. Apskaičiuojame kelio kainą: 6+2=8, šią vertę priskiriame mazgui Nr.6, o mazgą Nr.3 pažymime kaip aplankytą ir išmetame iš neaplankytų mazgų sąrašo:



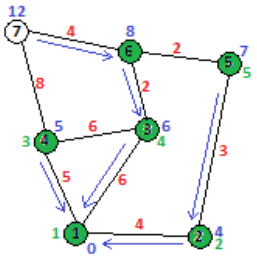
23 pav. Kelio paieškos Dijkstra algoritmu aplankytas ketvirtasis taškas

Dabar neaplankytų mazgų sąraše mažiausią vertę turi mazgas Nr.5 (žr. 24 pav.). Iš jo irgi galima patekti į mazgą Nr.6. Apskaičiuojame kelio kainą: 7+2=9. Tačiau mazgas Nr.6. jau turi priskirtą vertę, kuris yra mažesnė nei 9, taigi, jam naujos vertės nepriskiriame ir krypties į mazgo Nr.6 pirminį mazgą nekeičiame. Mazgą Nr.5. pažymime kaip aplankytą ir išmetame iš aplankytų mazgų sąrašo:



24 pav. Kelio paieškos Dijkstra algoritmu aplankytas penktasis taškas

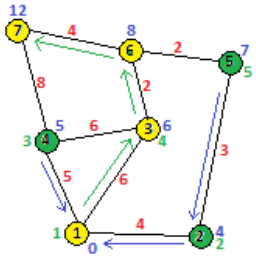
Sekantis mazgas su mažiausia verte yra Nr.6 (žr. 25 pav.). Iš jo galima patekti į mazgą Nr.7 (tikslo mazgą). Apskaičiuojame kelio kainą iki mazgo Nr.7: 8+4=12. Matome, kad gauta vertė yra mažesnė nei šiuo metu mazgui Nr.7. priskirta vertė, tai jam priskiriame vertę 12 bei pakeičiame rodyklę, kuria nurodomas pirminis mazgas. Nurodome kryptį į mazgą Nr.6, kadangi ateinant iš jo kelio kaina yra mažesnė nei ateinant iš mazgo Nr.4.:



25 pav. Kelio paieškos Dijkstra algoritmu aplankytas šeštasis taškas

Neaplankytų mazgų sąraše lieka vienintelis mazgas Nr.7 (žr. 26 pav.). Iš jo negalima patekti į jokį dar neaplankytą mazgą, taip pat jo vertė nėra lygi ∞, kas byloja, kad jis buvo pasiektas. Pažymime jį kaip aplankytą ir išmetame iš neaplankytų mazgų sąrašo. Kadangi šis sąrašas tapo tuščias, kelio paieška yra baigta. Lieka surasti trumpiausią kelią. Tai galima atlikti tokiais atgalinės paieškos būdais:

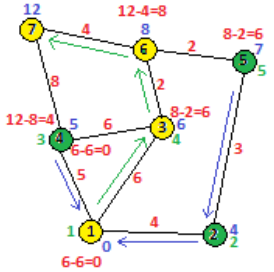
1) Sekame pažymėtomis rodyklėmis iki pradinio taško, šiuo atveju 7→6→3→1, seką invertuojame, ir gauname trumpiausio kelio seką 1→3→6→7, kurio kaina 6+2+4=12.



26 pav. Kelio paieškos Dijkstra algoritmu aplankytas septintasis taškas

2) Jei nėra galimybės žymėti krypčių į pirminius mazgus, skaičiuojama taip: tikrinami mazgai, iš kurių galima ateiti į mazgą Nr.7, ir skaičiuojama, kokia būtų jų vertė, iš tikslo mazgo atėmus kelio iki jų svorį (žr. 27 pav.). Gauname : 7→4 yra 12-8=4, 7→6 yra 12-4=8. Keliaujame į tą mazgą, kuriam priskirta vertė sutampa ką tik apskaičiuota, šiuo atveju tai mazgas Nr.6. Tikriname, iš kurių mazgų galima į jį ateiti, tai mazgai Nr.3 ir Nr.5. Apskaičiuojame atgalinę vertę iki jų: 6→5 yra 8-2=6, 6→3 yra 8-2=6. Sutampa vertė tik Nr.3 mazgui, tad keliaujame į jį. Į mazgą Nr.3 galima pateikti iš mazgų Nr.4 ir

Nr.1. Apskaičiuojame vertes: 3→4 yra 6-6=0, 3→1 yra 6-6=0. Vertė sutampa tik mazgui Nr.1, pasirenkama kryptis į jį. Kadangi mazgas Nr.1. yra pradinis taškas, kelias laikomas surastu. Seka invertuojama ir gaunamas toks pat kelias kaip ir pirmuoju būdu 1→3→6→7, kurio kaina 6+2+4=12:



27 pav. Trumpiausio kelio apskaičiavimas Dijkstra algoritmu

Dijkstra algoritmo paieškos algoritmą, kai neaplankytų mazgų prioritetų sąrašas yra FIFO sąrašas, galima vadinti paieškos į plotį algoritmu.

Algoritmo minusas – būtina ištirti visus grafo (žemėlapio) mazgus. Esant dideliam žemėlapiui, tai užima daug laiko.

1.6. Belmano-Fordo kelio paieškos algoritmas

Tai algoritmas, apskaičiuojantis trumpiausius kelius nuo duotojo mazgo kryptiniame grafe su svoriais, t.y., grafe, kuriame nurodytos kryptys tarp mazgų. Jis lėtesnis už Dijkstra algoritmą, tačiau lankstesnis, kadangi gali būti pritaikomas grafams su neigiamais atstumų tarp mazgų svoriais. Algoritmą nepriklausomai vienas nuo kito paskelbė: 1958 m. Richard Bellman, 1956 m. Lester Ford, Jr. Taip pat 1957 m. šį algoritmą nepriklausomai paskelbė Edward F. Moore, dėl to kartais šis algoritmas yra vadinimas Bellman-Ford-Moore algoritmu [15, 16].

Kaip ir Dijksta algoritmas, šis yra pagrįstas skaičiavimo principu, pagal kurį teisingas atstumas yra gaunamas jį palaipsniui pakeičiant tikslesnėmis vertėmis, kol randamas optimalus sprendimas. Abiejuose algoritmuose apytikslis atstumas iki kiekvieno mazgo yra didesnis už tikrąjį atstumą, ir yra pakeičiamas naujai rasto kelio ilgiu. Palyginimui: Dijkstra algoritmas pasirenka mažiausio svorio mazgą, kuris dar nebuvo apdorotas, ir atlieka šį atstumo perskaičiavimo procesą visiems iš mazgo išeinantiems keliams (iki kito mazgo); Bellman-Ford algoritmas paprasčiausiai apskaičiuoja visus išėjimus iš visų mazgų, tai atlieka V-1 kartų, kur V – mazgų skaičius grafe. Kiekvieno šių pakartojimų metu mazgų kiekis, kuriems teisingai suskaičiuojamas atstumas, didėja, kol galiausiai visiems mazgams yra teisingai suskaičiuojami atstumų įverčiai.

Bellman-Ford algoritmas kelio suradimui gali būti vykdomas daugiausia V*E kartų, kur V – mazgų skaičius grafe, E – briaunų tarp mazgų skaičius

Trumpiausio kelio paieškoje neigiami atstumai tarp mazgų yra problema, kuri daugumai kelio paieškos algoritmų neleidžia rasti trumpiausio kelio, kadangi bet koks žingsnis tarp mazgų, turinčių neigiamą atstumo tarp jų įvertį, neteisingai apskaičiuoja kelio ilgį, t.y., realiai kelias tampa ilgesnis, tačiau matematiškai jis sutrumpėja. Štai čia ir gali būti pritaikomas Bellman-Ford algoritmas, kuris gali tokius ciklus atskirti.

Bellman – Ford algoritmo trūkumai:

- Nėra efektyvus
- Kelio paieškos metu atsiradus pakitimams grafe, reaguoja į tai lėtai
- Skaičiavimo iki begalybės tikimybė – gali susidaryti amžinasis ciklas

1.7. Floyd-Warshall algoritmas

Šis algoritmas, kaip ir Bellman-Ford algoritmas, skirtas trumpiausio kelio paieškai kryptingame grafe tiek su teigiamais, tiek su neigiamais briaunų svoriais (tačiau be neigiamų ciklų). Vienas algoritmo įvykdymas randa trumpiausius kelius tarp visų grafo mazgų porų, tačiau nenurodo paties trumpiausio kelio tarp dviejų nurodytų mazgų.

Algoritmą 1962 m. paskelbė Roberts Floyd. Taip pat jį nepriklausomai 1959 m. paskelbė Bernard Roy, bei tais pačiais 1962 m. Stephen Warshall. Todėl jis dar vadinamas Roy-Warshall, Roy-Floyd, arba tiesiog WFI algoritmu [27].

Algoritmas palygina visus įmanomus kelius grafe tarp kiekvienos mazgų poros. Jei turimas grafas su mazgais, sužymėtais nuo 1 iki N, trumpiausiu keliu laikomas kelias tarp pradinio *i* bei galutinio *j* mazgų, išreikštas funkcija *shortestPath(i, j, k),* kelio paieškai panaudojant tik mazgus iš tam tikros aibės {1,2,...k}, kaip tarpinius kelio taškus. Turint šią funkciją, tikslas yra surasti trumpiausią kelią tarp kiekvieno *i* bei *j* naudojant tik mazgus nuo 1 iki k+1.

Kiekvienai šių mazgų porai, trumpiausias kelias gali būti:

1) kelias, einantis per mazgus iš aibės {1,..,k}
2) kelias, einantis iš i-ojo mazgo į k+1, tada iš k+1 į j-ąjį mazgą.

Geriausias (trumpiausias) kelias iš *i* į *j* , naudojantis tik mazgus iš aibės {1,...,k}, kaip jau rašyta, apibrėžiamas funkcija *shortestPath(i, j, k).* Jei w(i,j) yra briaunos (kelio) tarp i ir j mazgų svoris, tai tuomet trumpiausią kelią gali apibrėžti tokia rekursine formule:

1) bazinis modelis $shortestPath(i, j, 0) = w(i, j)$

2) rekursinis modelis:

$$shortestPath(i, j, k+1) = \min(shortestPath(i, j, k), shortestPath(i, k+1, k) + shortestPath(k+1, j, k))$$

Ši 2-ame punkte paminėta formulė yra Floyd-Warshall‘o algoritmo pagrindas. Pagal ją algoritmas pirmiausia apskaičiuoja visus trumpiausius kelius visoms (i,j) poroms, kai k=1, tada kai k=2, k=3 ir t.t. Procesas tęsiamas, kol k=N.

Pats algoritmas pateikia tik trumpiausio kelio ilgius tarp visų galimų mazgų porų. Norint „atsekti“ kelią, algoritmą reikia kiek modifikuoti. Tačiau šis procesas yra užimantis daug laiko ir atminties.

1.8. A* kelio paieškos algoritmas

A* (tariamas *A Star*) yra paieškos algoritmas, plačiai naudojamas kelio paieškai bei sankirtoms grafuose. Pasižymi dideliu tikslumu ir atlikimo sparta, dėl to jis yra dažnai taikomas įvairiuose kelio paieškos uždaviniuose. Pirmą kartą aprašytas 1968 m., Stanford instituto profesorių Peter Hart, Nils Nilsson ir Bertram Rapheal. Tai iš esmės Dijkstra algoritmo išplėtimas. Didesnę spartą algoritmas pasiekia naudodamas euristinius metodus [8].

A* algoritmas naudoja „geriausios pirmos paieškos“ būdą ir surandą mažiausios kainos kelią nuo pasirinkto pradinio mazgo iki vieno galutinio mazgo (jei yra keli keliai, pasirenka vieną iš jų). Keliaudamas grafu, A* algoritmas seka mažiausios tikėtinos kainos keliu arba mažiausiu likusiu atstumu iki tikslo, išlaikydamas surūšiuotą prioritetų eilę alternatyviems kelio segmentams, jei reikėtų tokius pasirinkti. Algoritme remiamasi „patirtis plius euristika“ mazgo x kainos funkcija (dažniausiai žymimo kaip f(x)) nusakyti eiliškumui, kuriuo turi būti aplankomi medžio (grafo) mazgai. Kainos funkcija susideda iš šių dviejų funkcijų:

1. Nueito kelio kainos funkcija, kuri reiškią nueitą atstumą nuo pradinio iki einamojo mazgo. Įprastai žymima g(x).
2. Likusio kelio kainos funkcija, kuri yra priimama kaip euristinis atstumo nuo einamojo iki tikslo mazgo įvertis. Įprastai žymima h(x).

h(x) funkcija f(x) funkcijoje privalo būti priimtinai euristinė, t.y., neturi pervertinti atstumo iki tikslo. Tad maršruto parinkimo algoritmuose, h(x) gali reikšti atstumą tiesia linija (Euklido atstumas) iki tikslo, kitaip tariant, trumpiausias įmanomas atstumas tarp dviejų mazgų.

Jei euristinis h(x) tenkina papildomą sąlygą h(x)<=d(x,y)+h(y) kiekvienai grafo briaunai (x,y), (d šiuo atveju reiškia briaunos ilgį), tai h(x) funkcija yra vadinama monotonine, arba nuoseklia. Dėl to A* algoritmas gali būti įgyvendintas efektyviau, nes nei vieno mazgo nereikia aplankyti daugiau kaip vieną kartą. Šiuo atveju A* algoritmas yra ekvivalentiškas Dijkstra algoritmui su sumažinta kaina: d‘(x,y)=d(x,y)+h(y)-h(x).

A* algoritmu pirmiausia ieškoma maršrutų, kurie atrodo labiausiai tikėtini pasiekti tikslui. Kas išskiria A* iš kitų „gobšių“ geriausios pirmos paieškos algoritmų, tai kad jis naudoja jau nueito kelio atstumą g(x), kuri yra nueito kelio atstumas nuo pradinio mazgo, o ne prieš tai buvusio mazgo.

Kelio paieška pradedama nuo pradinio mazgo, atsižvelgiama į turimų aplankyti mazgų eilės prioritetiškumą. Ši eilė dar vadinama atvira aibe, atviru sąrašu arba pakraščiu. Toliau naudosime terminą *atviras sąrašas*. Kuo mažesnė f(x) vertė kiekvienam iš šių mazgų, tuo didesnis mazgo aplankymo prioritetas. Kiekvieno žingsnio metu, mazgas su mažiausia f(x) verte yra išmetamas iš atvirojo sąrašo, tada atnaujinamos visų šio mazgo kaimynų g(x) ir h(x) vertės, apskaičiuojamos jų f(x) vertės, ir su šiomis vertėmis jie įtraukiami į atvirąjį sąrašą. Algoritmas tęsiamas tol, kol tikslo mazgo f(x) vertė yra mažiausia nei visų aibėje esančių mazgų f(x) vertė, arba kol sąrašas tampa tuščias. Tikslo mazgas gali būti aplankytas ne vieną kartą tuo atveju, jei yra kitų mazgų atvirame sąraše su mažesnėmis f(x) vertėmis, ir kelias per juos gali būti trumpesnis. Tikslo mazgo f(x) vertė ir yra trumpiausio kelio atstumas, tuo pačiu h(x) vertė tikslo mazgui yra lygi nuliui.

Norint rasti ne tik kelio ilgį, tačiau ir seką, kuriais mazgais jis pasiektas, reikia peržiūrėti algoritmą nuo galo iki pradžios taip, kad kiekvienas kelio mazgas parodytų savo pirmtaką. Pirmtakas randamas pagal paprastą formulę: g(x-1)= g(x)–d(x-1,x), čia x – tiriamas mazgas, x-1 – galimas tiriamo mazgo pirmtakas, d - atstumas tarp x ir x-1 mazgų pagal pasirinktą atstumo tipą (Euklido, Manheteno ar kt.). Taip tikrinama, kol pasiekiamas pradinis mazgas.

Jeigu euristika yra monotoninė (pagal anksčiau aprašytą formulę h(x)<=d(x,y)+h(y)), gali būti naudojamas uždarytų mazgų sąrašas, su tikslu tą patį mazgą aplankyti tik vieną kartą ir pačią kelio paiešką padaryti veiksmingesnę. Į šį sąrašą įtraukiami iš atvirojo sąrašo išmesti mazgai.

Bendri A* algoritmo principai paaiškinti žemiau:

1. Uždarytas sąrašas iš pradžių yra tuščias, į atvirąjį sąrašą įtraukiamas pradinis mazgas.
2. Visiems praeinamiems žemėlapio ar grafo mazgams apskaičiuojama h(x) vertė pagal parinktą atstumo funkciją. Pradinio mazgo g vertė priskiriama lygi 0, apskaičiuojama pradinio mazgo f(x) vertė = g(x) + h(x). Visiems likusiems praeinamiems mazgams g(x) vertė priskiriama lygi ∞.
3. Tikrinama, ar atvirasis sąrašas nėra tuščias.
4. Jei sąrašas nėra tuščias, iš jo išrenkamas mazgas su mažiausia f(x) verte. Jei sąrašas tuščias, peršokama prie 13 punkto.
5. Tikrinama, ar mazgas nesutampa su tikslo mazgu. Jei sutampa, ciklas baigiamas, peršokama į 14 punktą.
6. Jei išrinktasis mazgas nėra tikslo mazgas, atliekami sekantys veiksmai:

7. Surandami visi išrinktojo mazgo kaimynai – mazgai, į kuriuos iš jo galima pateikti visomis kryptimis. Kaimynai turi nepriklausyti uždarajam sąrašui.
8. Išrinktasis mazgas išmetamas iš atvirojo sąrašo ir įtraukiamas į uždarytąjį sąrašą. Uždarytame sąraše nurodoma, iš kurio mazgo į jį buvo ateita. Nuorodos neturi tik pradinis mazgas.
9. Visiems surastiems kaimynams atnaujinamos g vertės pagal principą:
   g (mazgo) = min (g (mazgo pirmtako) + atstumas tarp mazgo ir jo pirmtako pagal pasirinktą atstumo funkciją). Kitaip tariant, jei į mazgą galima patekti iš daugiau vieno mazgo, priskiriama mažiausia g vertė.
10. Visiems kaimynams apskaičiuojamos f(x) vertės = g (x) + h(x).
11. Visi kaimynai, jei jie nėra įtraukti į uždarąjį sąrašą, taip pat jei jie nepriklauso atvirajam sąrašui, įtraukiami į atvirąjį sąrašą su apskaičiuotomis f(x) vertėmis.
12. Grįžtama į 3-ą punktą.
13. Jei atvirasis sąrašas tampa tuščias, kelio surasti nepavyko.
14. Jei pasiektas tikslo mazgas, atliekami tokie veiksmai: tikslo mazgas išmetamas iš atvirojo sąrašo, įtraukiamas į uždarytąjį sąrašą su nuoroda, iš kurio mazgo į jį ateita. Taip pradedant juo ir sekant kiekvieno mazgo nuoroda į jo pirmtaką, randama kelio maršruto seka.

Pažymėtina: aukščiau esantis kodas daro prielaidą, kad euristinė funkcija yra monotoninė, kas yra dažnas atvejis daugelyje praktinių problemų, tokių kaip trumpiausio kelio paieška kelių tinkle. Tačiau, jei prielaida nepasitvirtina, mazgai uždarytame sąraše gali būti peržiūrėti iš naujo ir jiems padidinta kelio kaina. Kitaip tariant, pakoreguojamas uždarytų mazgų sąrašas, jei garantuojama, kad yra trumpiausias kelias, arba jei algoritmas yra adaptuotas taip, kad nauji mazgai yra pridedami į atvirą aibę tuo atveju, jei jų f(x) vertė yra mažesnė nei bet kurioje prieš tai buvusioje iteracijoje.

Algoritmo savybės:

- Jei kelią surasti įmanoma, A* algoritmas jį visada ras.
- Jei euristinė funkcija h(x) yra priimtina, turima omeny niekada nepervertina aktualios minimalios tikslo pasiekimo kainos, tai pats A* yra priimtinas (arba optimalus), jei nenaudojamas uždarytų mazgų sąrašas. Jei jis naudojamas, tai kad A* būtų optimalus, h(x) privalo irgi būti monotoninė. Tai reiškia, kad bet kuriai vienas su kitu besijungiančių mazgų porai x ir y, kur d(x,y) yra Euklidinis atstumas tarp jų, arba kito pasirinkto tipo atstumas, turi būti tenkinama sąlyga: h(x) <= d(x,y) + h(y)

Tai užtikrina, kad bet kuriam keliui X iš pradinio mazgo į mazgą x galioja:

L(X) +h(x) <= L(x) +d(x,y) +h(y)= L(Y) +h(y), čia L yra funkcija, žyminti kelio ilgį, o Y yra kelias X, padidintas įtraukiant mazgą y. Kitaip tariant, išplečiant kelią, įtraukiant kaimyninį mazgą, negalima sumažinti atstumo „nueitas kelias + likęs kelias“.

Pastebėjimas : nors pagal aprašymą A* skirtas statinei aplinkai, tačiau jį galima naudoti ir dinaminėje aplinkoje, tokiu atveju kiekvieną kartą atsiradus kliūčiai algoritmą teks pakartoti iš pradžių be jokios pirminės mazgų apdorojimo informacijos.

1.8.1. Ribojimo atpalaidavimas A* algoritme

Sąlyga, kad euristika h(x) turi būti monotoninė, leidžia surasti trumpiausią kelią. Tam A* algoritmas turi patikrinti visus vienodo ilgio kelius optimalaus kelio radimui. Tačiau yra uždavinių, kuriuose nebūtina rasti trumpiausią kelią. Tokiuose uždaviniuose galima pagreitinti kelio paiešką aukojant optimalumą naudojant tam tikro leistinumo kriterijaus „atpalaidavimą“ (angl. b*ounded relaxation*). Leistinumo kriterijus trumpiausiam keliui yra lygus 1. Dažniausiai šis atpalaidavimas susiejamas su garantija, kad rastas kelias nėra blogesnis daugiau kaip (1 + ε) kartų nei galimas trumpiausias. Ši atpalaidavimas yra vadinama ε - leistinumu.

Yra keletas ε-leistinumo algoritmo variantų:

- Svorinis A* [9]. Jei $h_{a(n)}$ yra priimtina euristinė funkcija, algoritmas naudoja $h_{w(n)} = \varepsilon h_{a(n)}$, $\varepsilon>1$ kaip euristinę funkciją, ir atlieka įprastinę A* paiešką, kuri tampa greitesne nei naudojant h_a, kai yra išplečiami keli mazgai. Tokiu būdu rastas kelias dažniausiai turi ε kartų didesnę kainą nei mažiausios kainos kelias grafe.
- Statiškas svorių suteikimas. Algoritmas naudoja tokią kainos funkciją f(n) = g(n) + (1 + ε)h(n).
- Dinaminis svorių suteikimas. Algoritmas naudoja kainos funkciją

f(n) = g(n) + (1 + ε w(n))h(n), kur:

$$w(n) = \begin{cases} 1 - \frac{d(n)}{N} & d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$$, čia *n* – paieškos gylis, N – tikimasis kelio ilgis.

- Atrenkamas dinaminis svorių suteikimas. Algoritmas naudoja mazgų atrinkimą geresniam euristinės klaidos įvertinimui.
- A^*_ε naudoja dvi euristines funkcijas. Viena jų yra FOCAL sąrašas, naudojantis parinkti mazgus-kandidatus, antra h_F yra naudojama žadančiam geriausią rezultatą mazgo parinkimą iš FOCAL sąrašo.

- A_ε parenka mazgus su funkcija A f(n) +B $h_F(n)$, čia A ir B – konstantos. Jei joks mazgas negali būti parinktas, tolesniems mazgų skaičiavimams algoritme naudojama funkcija C f(n) + D $h_F(n)$, čia C ir D – konstantos.
- AlphA* algoritmas bando propaguoti paieškos į gylį algoritmo panaudojimo galimybę teikiant pirmenybę paskiausiai išplėstiems mazgams. Naudojama tokia kainos funkcija: $f_\alpha(n) = (1 + w_\alpha(n))\, f(n)$, kur

  $$w_\alpha(n) = \begin{cases} \lambda & g(\pi(n)) \le g(\tilde{n}) \\ \Lambda & \text{otherwise} \end{cases}$$

  čia λ ir Λ yra konstantos, λ <= Λ, π(n) yra n mazgo pirmtakas, o n~ yra paskutinis išplėstas mazgas.

1.9. IDA* kelio paieškos algoritmas

Tai A* algoritmo atmaina (tariama IDA star, *Iterative deepening A**) , naudojanti iteracinį gilinimą, kad atminties sunaudojimas būtų mažesnis nei A* [13]. Tai informuota paieška, paremta neinformuotos iteracinio gilinimo paieškos į gylį algoritmo idėja. Priešingai nei standartinis iteracinis paieškos į gylį algoritmas, kuriame paieškos gylis naudojamas kaip riba kiekvienai iteracijai užbaigti, IDA* naudoja labiau informatyvią funkciją f(n)=g(n)+h(n), kur g(n) yra kelio kaina nuo pradinio iki n-ojo mazgo, o h(n) – euristinis kelio kainos įvertis nuo n-ojo mazgo iki tikslo pagal pasirinktą atstumo funkciją.

IDA* yra lėtesnis už A*, kadangi tuos pačius mazgus tikrina ne vieną kartą, nes neįsimena senesnių veiksmų, tačiau jo naudingumas pasireiškia ten, kur reikalingas griežtas atminties naudojimo ribojimas, o kelio paieškos greitis nėra svarbus. Detaliau šio algoritmo nenagrinėsim.

1.10. SMA* kelio paieškos algoritmas

Tai A* algoritmo atmaina (iš anglų kalbos Simplified Memory Bounded A*), supaprastintas apribotos atminties A* [14]. Pagrindinis jo pranašumas prieš A* yra apribotos atminties naudojimas, t.y., veikimui reikalinga mažiau atminties nei A* algoritmui. Visos kitos charakteristikos analogiškos A*.

Kaip ir A*, šis algoritmas remdamasis euristika išplečia tas trumpiausio kelio šakas, kurios laikomos labiausiai perspektyviomis. SMA* išsiskiria tuo, kad jis „nukerpa“ tuos mazgus, kurių išplėtimas pasirodė esąs mažiau perspektyvus nei tikėtasi. Šis metodas leidžia ištirti kelio šakas bei grįžti atgal ištirti kitas.

Mazgų išplėtimas bei „nukirpimas“ paremtas išsaugojant dvi f(x) funkcijos vertes kiekvienam mazgui. Mazgas x saugo f(x) vertę, kuri reiškia kelio kainą pasirenkant kelią per šį

mazgą. Kuo žemesnė vertė, tuo didesnis mazgo pasirinkimo prioritetas. Kaip ir A*, f(x) vertė yra lygi h(x) + g(x), tačiau vėliau yra atnaujinama norint atspindėti pokyčius, kai mazgas išplečiamas. Pilna išplėsto mazgo f(x) vertė yra ne mažesnė nei jo įpėdinio. Papildomai, mazgas išsaugo geriausio pamiršto įpėdinio f(x) vertę. Ši vertė atstatoma, jei pamirštasis įpėdinis pasirodo esantis labiausiai perspektyvus.

Paieška pradedama nuo pirmo mazgo, jos metu, kaip ir A* algoritme, sudaromas atviras mazgų sąrašas, kuriame jie surikiuoti leksikografiškai pagal f(x) vertę ir gylį. Kai renkamas mazgas išplėtimui, pasirenkamas geriausias pagal surūšiuotą sąrašą, o kai reikia mazgą išmesti („nukirpti“), pasirenkamas blogiausias.

Savybės:

- Remiasi euristika
- Yra užbaigtas, jei leidžiamas atminties kiekis yra pakankamas išsaugoti paviršutiniškam sprendimui
- Yra optimalus, jei leidžiamas atminties kiekis yra pakankamas išsaugoti optimalų paviršutinišką sprendimą, kitu atveju algoritmas pateikia geriausią į leidžiamą atmintį telpantį sprendimą
- Tol kol atminties ribojimai leidžia, išvengia pakartotinių būsenų
- Išnaudoja visą leidžiamą atminties kiekį
- Algoritmui skirtos atminties ribojimo didinimas tik pagreitintų skaičiavimus
- Kai yra pakankamai atminties išsaugoti visą paieškos medį, skaičiavimų greitis yra optimalus

Algoritmo vykdymas yra toks pats kaip ir A*, skirtumas toks, jei nebelieka vietos atmintyje, mazgai su didžiausia f(x) verte yra išmetami iš sąrašo. SMA* turi įsiminti geriausio išmesto mazgo palikuonio f(x) vertes. Tai reikalinga tam, jei paaiškėtų, kad išmetus šį mazgą, be jo rasti keliai yra prastesni nei keliaujant per šitą mazgą, ir būtų galima iš naujo apskaičiuoti kelią grąžinus šį mazgą atgal į sąrašą.

1.11. D* kelio paieškos algoritmas (D Star)

Yra skiriamos 3 šio algoritmo atmainos:

- Originalus D*, sukurtas Anthony Stentz, tai informuotas progresuojančios paieškos algoritmas [10].
- Fokusuotas D*, tai informuotas euristinis progresuojančioss paieškos algoritmas, sukurtas Anthony Stentz, apjungiantis A* ir Originalaus D* įdėjas [11].
- D*Lite tai euristinis progresuojančios paieškos algoritmas, sukurtas Sven Koenig ir Maxim Likhachev, paremtas LPA (Lifelong Planning A*) – euristiniu progresuojančios

paieškos algoritmu, kuris apjungia A* ir DINAMINIO SWSF-FP kelio paieškos algoritmus [12].

Visi trys algoritmai sprendžia tokias pat prielaidomis paremtomis kelio planavimo problemas, įskaitant laisvą planavimą, t.y., kai reikia rasti kelią pagal duotas koordinates nežinomoje teritorijoje. Daromos prielaidos apie dar neištirtą teritorijos dalį, pavyzdžiui, kad einamuoju momentu kelyje nėra jokių kliūčių, ir randamas trumpiausias kelias nuo einamųjų koordinačių iki tikslo koordinačių remiantis sudarytomis paklaidomis. Sudaromas teritorijos žemėlapis ir judama rastu keliu. Jei gaunama nauja informacija, pavyzdžiui, atsirado nenumatyta kliūtis, kuri prieš tai nebuvo žinoma, atnaujinama žemėlapio informacija, pažymimos kliūtys, ir, jei būtina, nuo einamojo taško yra surandamas naujas kelias iki tikslo. Procesas tęsiamas tol, kol pasiekiamas tikslas arba įsitikinama, kad tikslo pasiekti neįmanoma. Kadangi turima omenyje, jog kelio paieška vykdoma pilnai arba iš dalies nežinomoje teritorijoje, kurioje dažnai gali būti aptinkamos dar nematytos kliūtys, pakartotinis kelio radimas turi būti greitas. Progresuojančios paieškos (+ euristiniai) paieškos algoritmai, remdamiesi ankstesnių problemų sprendimų patirtimi, pagreitina einamuoju metu ieškomo kelio paiešką.

D* algoritmas ir jo variantai plačiai naudojami mobilių robotų, autonominių mašinų navigacijai, dažniausiai naudojamos algoritmo modifikacijos, nes originalus algoritmas yra gana sudėtingas. Šiuo metu daug sistemų naudoja D* Lite algoritmo variantą. Šiuo algoritmu paremtos navigacijos sistemos, sukurtos Carnegie Mellon universitete, buvo testuojamos marsaeigiuose „Opportunity“ ir „Spirit“, taip pat buvo naudojamos DARPA Urban Challenge turnyre, kur jį naudojanti komanda pasiekė pergalę.

Originalus D* algoritmas buvo pristatytas 1994, Anthony Stentz. Pavadinimas reiškia „Dinaminis A*“, tai reiškia, kad tai yra A* algoritmas su modifikacija, pagal kurią kelio kaina gali keistis algoritmo vykdymo metu.

Kaip Dijkstra ir A* algoritmuose, D* algoritme yra sudaromas mazgų sąrašas, vadinamas atidarytu sąrašu. Žemėlapio mazgai gali turėti vieną iš šių būsenų:

- NAUJAS. Šis mazgas dar nebuvo patekęs į atidarytą sąrašą.
- ATIDARYTAS. Šis mazgas einamuoju momentu yra atidarytame sąraše.
- UŽDARYTAS. Šio mazgo jau nebėra atidarytame sąraše.
- KYLANTIS. Šio mazgo kelio kaina yra padidėjusi nuo to karto, kai jis paskutinį buvo įtrauktas į atidarytą sąrašą.
- KRENTANTIS. Šio mazgo kelio kaina yra sumažėjusi nuo to karto, kai jis paskutinį kartą buvo įtrauktas į atidarytą sąrašą.

Algoritmas veikia taip: vienas po kito pasirenkami mazgai iš atidaryto sąrašo ir įvertinami. Mazgo įvertinimas sklinda visiems kaimyniniams mazgams, jie įtraukiami į atidarytą sąrašą. Šis

sklidimas vadinamas išplėtimu. Priešingai nei A*, kuris seka keliu nuo pradinio mazgo link galutinio, D* paiešką vykdo atgal nuo tikslo mazgo. Kiekvienas išplėstas mazgas turi atgalinę rodyklę, rodančią į kitą mazgą, vedantį į tikslą, ir tikslią kainą iki jo. Kai mazgas, kuris turi būti išplėstas, yra pradinis, algoritmas baigiamas, kelias surandamas sekant rodykles.

Kai numatytame kelyje atsiranda kliūtis, visi mazgai, kuriuos paveikia kliūtis, vėl įtraukiami į atidarytą sąrašą, šį kartą pažymimi kaip KYLANTYS. Prieš mazgo kainos padidinimą, algoritmas patikrina jo kaimynus ir įvertina, ar pagal juos galima kainą sumažinti. Jei ne, būsena KYLANTIS skleidžiama visiems mazgo palikuonims (mazgams, kurių rodyklės nukreiptos į jį). Šie mazgai įvertinami, jiems perduodama būsena KYLANTIS suformuoja bangą. Jei mazgo su būsena „KYLANTIS“ kaina gali būti sumažinta, atnaujinama jo rodyklė, kaimyniniams mazgams skleidžiama būsena KRENTANTIS, kuri irgi suformuoja bangą. Šios „kylančių“ ir „krentančių“ būsenų bangos yra D* algoritmo pagrindas. Algoritmas atsiradus kliūčiai dirba tik su tais mazgais, kurių kaina pasikeičia.

Kelio paieškoje D* algoritmu gali pasitaikyti aklaviečių. Šiuo atveju, taip paprastai aklavietės kaip pasitaikius kliūčiai apeiti negalima. Nei viename iš mazgų per jo kaimyninius mazgus negalima rasti kelio iki tikslo. Todėl ir toliau yra skleidžiamas jų kainos didėjimas. Tik išėjus iš aklavietės gali būti randami taškai, kurie gali vesti į tikslą tinkamu maršrutu. Tokiu būdu suformuojamos dvi KRENTANČIOS bangos, kurios plečiasi pažymėdamos nepasiekiamus taškus, su nauja informacija apie maršrutą.

1.12. Fokusuotas D* kelio paieškos algoritmas

Šis algoritmas remiasi euristika bangų skleidimui link einamojo mazgo. Šiuo būdu atnaujinamos tik reikšminės būsenos, taip kaip A* algoritmas apskaičiuoja tik kai kurių mazgų kainą.

1.13. D* Lite kelio paieškos algoritmas

Šis algoritmas nėra grindžiamas D* ar fokusuoto D* pagrindu, tačiau elgiasi taip pat. Yra lengviau suprantamas, įgyvendinimui reikalingas trumpesnis programinis kodas. Iš čia ir pavadinimas „D* Lite“ [21] [22]. Yra toks pat geras arba geresnis už fokusuotą D*. D* Lite yra pagrįstas LPA* (Lifelong Planning A*) [23][24], kurį Koenig ir Likhachev pristatė keliais metais anksčiau, 2002m, Artificial Intelligence 155 (2004) 93–146 Tai iš esmės naujesnis už A* ir D* kelio paieškos algoritmas. Kadangi jis paremtas LPA* algoritmu, pirmiausia panagrinėkime šį algoritmą.

LPA* kelio paieškos algoritmas taikomas tada, kada turime žinomą baigtinį grafą, kuriame atstumų tarp mazgų vertės kinta tam tikrą laiko kiekį. Tarkime, turime baigtinę aibę S, kuri

nurodo mazgų kiekį grafe, succ(s) $\subseteq$ S - tai mazgo s paveldėtojų/tęsėjų aibė (mazgai į kuriuos galima patekti iš mazgo s), mazgas s$\in$S. Pred(s) $\subseteq$ S - tai mazgo s pirmtakų aibė (mazgai iš kurių ateinama į mazgą s), mazgas s$\in$S. Dar apibrėžkime kelio kainos tarp mazgų funkciją c(s,s`), kuri tenkina sąlygą 0<=c(s,s`)<=∞, tai yra kelio kaina iš mazgo s į mazgą s`, kur s‘ $\in$ succ(s). LPA* algoritmas visada nustato trumpiausią kelią iš duoto pradinio mazgo $s_{pradinis}\in S$ iki mazgo $s_{tikslo}\in S$, žinant grafo topologiją bei briaunų, kitaip dar atstumų tarp mazgų, svorius (vertes). Dar yra apibrėžiama funkcija g*(s), čia s$\in$S (mazgas einamuoju metu), kuri vadinama „starto atstumu“ ir apibūdina trumpiausio kelio atstumą nuo pradinio iki s mazgo pagal tokią priklausomybę:

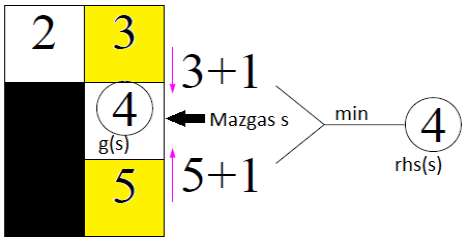
$$g^*(s)=\begin{cases} 0, jei \quad s=s_{start} \\ \min_{s`\in pred(s)}(g^*(s`)+c(s`,s)), kitu \quad atveju \end{cases}$$

Kaip matyti, funkcija yra rekursinė.

Dar vienas svarbus požymis LPA* algoritme - taip vadinamas „vietinis suderinamumas“. Jį apibrėžia jau aprašytos funkcijos g*(s) įvertinimas g(s) bei dar viena funkcija rhs(s), (*rhs – right hand side,* dešinės rankos pusės vertė) [28]. Jos vertė paremta g(s) pirmtakų vertėmis, ji visada tenkina šią sąlygą:

$$rhs(s)=\begin{cases} 0, \quad jei \quad s=s_{start} \\ \min_{s`\in pred(s)}\left(g(s`)+c(s`,s)\right) \quad kitu \quad atveju \end{cases}$$

Tai yra aplinkinių s mazgų vertėmis (per vieną žingsnį) pagrįsta reikšmė. Grafiškai ją galima pavaizduoti taip, kaip pavaizduota 28 paveiksle:



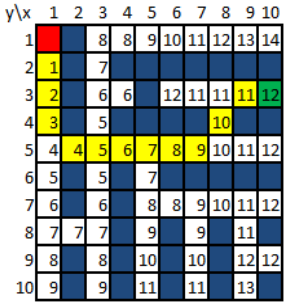
28 pav. Aplinkinių s mazgų vertėmis (per vieną žingsnį) pagrįsta reikšmė

Mazgas s yra laikomas vietiškai suderinamas tokiu atveju, jei g(s) = rhs(s), kitu atveju jis laikomas nesuderinamu. Jei visi mazgai yra vietiškai suderinami, tuomet visiems mazgams g(s)==g*(s).

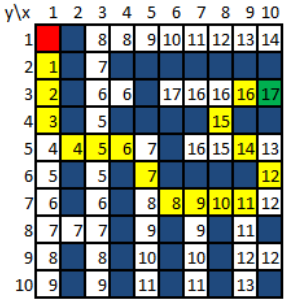
LPA* s mazgui yra sudaromas prioritetų sąrašas, kurie yra tik vietiškai nesuderinami mazgai. Šiems mazgams yra reikalinga pakeisti/atnaujinti jų g funkcijos vertes, kad jie taptų vietiškai suderinami. Eiliškumas prioritetų sąraše yra apibrėžiamas raktu k(s), tai yra dviejų komponenčių vektorius k(s)=[k_1(s);k_2(s)], čia k_1(s)=min (g(s), rhs(s) + h(s,s_{tikslo})), k_2(s)=min(g(s),rhs(s)).

$h(s,s_{tikslo})$ yra euristika, nurodanti trumpiausią atstumą nuo s iki tikslo mazgo (euklidinis atstumas). Mažesnio prioriteto mazgai yra pasirenkami pirmiausia.

Panagrinėsime algoritmą detaliau. Aiškesniam supratimui vaizdavimui pasirinksime ne grafą, o tinklelio tipo žemėlapį. Žemėlapyje esantys kvadratėliai bus pažymėti kaip praeinami arba nepraeinami. Iš vieno į kitą galima pakliūti 8 kryptimis (aukštyn, žemyn, kairėn, dešinėn bei kombinuotomis kryptimis įstrižai), be to, kelio kaina bet kuria kryptimi yra vienoda, pavyzdžiui, 1. Kaip jau minėta, LPA* visada nustato trumpiausią kelią tarp dviejų duotų mazgų, kai žinoma, kurie mazgai yra nepraeinami, kitaip tariant, kliūtys. Kuo LPA* skiriasi nuo tradicinių kelio paieškos metodų? Tuo, kad panaudoja jau turimą informaciją apie pirminę kelio paiešką, tai yra tinkamas metodas, kai žemėlapyje kelio paieškos metu atsiranda dar nematytų kliūčių. Imsime 29 paveiksle pavaizduotą žemėlapį. Langeliuose surašytos kelio kainos, pradedant pradiniu (raudonu), baigiant tikslo (žaliu). Geltonai pažymėtas trumpiausias kelias. Tokį kelią rastų A* algoritmas, paieškos į plotį algoritmas. Dabar pažiūrėkime, kas bus, jei kiek pakoreguosime žemėlapį, tarkim, y5x6 langelį padarykime nepraeinamu. 30 paveiksle matome, kad trumpiausias kelias pasikeičia:

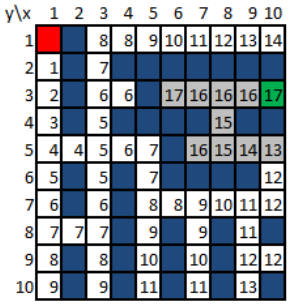


29 pav. Kelio paieška LPA* algoritmu



30 pav. Kelio paieška LPA* algoritmu po žemėlapio pakoregavimo

Pasikeičia ir trumpiausio kelio kaina, vietoj buvusios 12 dabar tampa 17. Reikia atkreipti dėmesį į tai, kad pasikeičia tik dalies langelių vertės (31 paveiksle pažymėta pilkai):



31 pav. Dalies langelių verčių pasikeitimas po žemėlapio pakoregavimo

LPA* algoritmas apjungia dviejų skirtingų kelio paieškos algoritmų – Dynamic SWSF-FP [28] ir A* idėjas greitesniam kelio perplanavimui atsiradus kliūtims. Pagal žemėlapį matyti, kad iki y5x4 mazgo kelias lieka tas pats, pasikeičia tik nuo šio mazgo. Tokie algoritmai, kaip A*, paieškos į plotį, perskaičiuotų kelią iš naujo nuo pradinio taško, o LPA* tik nuo taško, kuriame kelias pasikeičia. Tai sumažina mazgų, kurių kelio kainos vertes reikia perskaičiuoti, kiekį.

Dynamic SWSF-FP (*strict weakly superior function – fixed point*, fiksuoto taško nežymiai, bet tiksliai pranašesnė funkcija) algoritmo veikimo principo nenagrinėsime, kadangi pagrindiniai jo principai pateikti LPA* algoritme, kurį paaiškina jo pseudo kodas [21]:

```
procedure CalculateKey (s)
return [min (g(s), rhs(s)) + h(s, s_tikslo); min (g(d), rhs(s))];
procedure Initialize ()
U = ∅;
for all s ∈ S  rhs (s) = g(s) = ∞;
rhs (s_starto) = 0;
U.Insert (s_starto, CalculateKey(s_starto));
procedure Update Vertex (u)
if (u ≠ s_starto) rhs (u) = min_{s`∈Pred(u)} (g(s`) + c(s`,u));
if (u ∈ U) U.Remove (u)
if (g(u) ≠ rhs (u)) U.Insert (u, CalculateKey(u));
procedure ComputeShortestPath ()
while (U.Topkey () CalculateKey (s_tikslo) OR rhs (s_tikslo) ≠ g(s_tikslo))
        u = U.Pop ();
        if (g(u) > rhs (u))
            g (u) = rhs (u);
            for all s ∈ Succ (u) UpdateVertex (s);
        else
           g (u) = ∞;
        for all s ∈ Succ (u) ∪ {u} UpdateVertex (s);

procedure Main ()
Initialize ();
forever
        ComputeShortestPath ();
        Wait for changes in the edge costs;
        for all directed edges (u, v) with changed edge costs
                Update the edge cost c(u,v);
                UpdateVertex (v);
```

Procedūra **Main**() yra pagrindinis algoritmas, kurio metu pirmiausia įvykdoma funkcija **Initialize**(), kuri nustato pradines g vertes visiems mazgas į ∝, taip pat priskiria rhs vertes atsižvelgiant į ją aprašančią formulę. Toliau vykdoma trumpiausia kelio paieška **ComputeShortestPath**() tarp pradinio ir galutinio mazgų. Trumpai aptarsim dar porą naudojamų procedūrų, skirtų tvarkyti prioritetų sąrašą: **U**.**Topkey**() – grąžina mažiausią visų mazgų, esančių sąraše (pseudokode jis pavadintas **U**), prioritetą (jei sąrašas tuščias, grąžinama vertė [∝,∝]); **U**.**Pop** () – ištrina mažiausio prioriteto mazgą iš sąrašo ir grąžina mazgą; **U**.**Insert**(s,k) – įterpia mazgą s į sąrašą su k prioritetu; **Update**(s,k) – pakeičia mazgo s, esančio sąraše, prioritetą į k; **U**.**Remove**(s) – iš prioritetų sąrašo U išmetamas mazgas s.

Procedūra **CalculateKey**(s) apskaičiuoja s mazgui rakto vektoriaus k(s) komponentes. Procedūra **UpdateVertex**(u) priklausomai nuo poreikio, atnaujina mazgo u rhs vertes, patikrina vietinį suderinamumą, prideda arba išmeta iš prioritetų sąrašo U.

ComputeShortestPath() procedūra pasikartojančiai perskaičiuoja vietiškai nesuderinamų mazgų g(s) vertes pagal prioritetų sąrašą. Jei mazgo s g(s)>rhs(s), mazgas s laikomas vietiškai perderintu. Kai išplečiamas toks mazgas, laikoma, kad rhs(s)=g*(s), kas reiškia, kad k(s)=[f(s); g*(s)], čia f(s) = g*(s)+h(s, s_{tikslo}). Šio mazgo g(s) vertė pakeičiama rhs(s) verte, kas reiškia, kad mazgas tampa vietiškai suderinamu. Jo g(s) vertė nebesikeičia, kol procedūra **ComputeShortestPath**() nenutraukiama. Jei mazgo s g(s)<rhs(s), toks mazgas laikomas vietiškai nepriderintu (*underconsistent*). Kai toks mazgas išplečiamas, jo g(s) vertė nustatoma $\propto$. Tai mazgą paverčia vietiškai suderintu arba perderintu. Jei išplėstas mazgas buvo vietiškai perderintas ar nepriderintas, jo g(s) pasikeitimas gali įtakoti ir jo paveldėtojų vietinį suderinamumą.

LPA* algoritmas plečia mazgus tol, kol tikslo mazgas tampa vietiškai suderinamas, ir raktas k(s) mazgui, kuris turi būti išplėstas, yra ne mažesnis nei tikslo mazgo. Jei po visos paieškos g(s_{tikslo})=$\propto$, tokiu atveju nėra baigtinio kelio tarp pradinio ir tikslo mazgų. Kitu atveju, gali būti apskaičiuojamas trumpiausias kelias tokiu būdu: pradedama tikslo mazgu, tada ieškoma jo pirmtako, kuris minimizuotų g(s`)+c(s`,s), ir taip, kol bus pasiektas pradinis mazgas (ryšiai gali būti nutraukti pasirinktinai).

Kai trumpiausias kelias apskaičiuojamas, laukiama, ar neatsiras briaunų/atstumų tarp mazgų pasikeitimų. Jei taip nutinka, kviečiama procedūra **UpdateVertex**(), atnaujinamos rhs vertės bei raktai tų mazgų, kuriuos paveikė šie pasikeitimai. Tuo pačiu atnaujinamos jų prioritetų vertės , jei jie pasidarė vietiškai suderinami arba nesuderinami. Tada vėl skaičiuojamas trumpiausias kelias.

LPA* tuo efektyvesnis, kuo didesnis seno ir naujo kelio sutapimas bei kuo mažiau pakinta pradinis kelias. Taip pat, skirtingai nei A* algoritmas, LPA* nenaudoja uždaryto mazgų sąrašo.

LPA* pagrindu yra sukurtas D*Lite trumpiausio kelio paieškos algoritmas. Algoritmo esmė – pasikartojančiai nustatomas trumpiausias kelias tarp einamojo bei tikslo mazgo, kai keičiasi grafo briaunų/atstumų tarp mazgų vertės robotui judant tikslo taško link. D*Lite algoritmas nedaro jokių prielaidų dėl atstumų tarp mazgų verčių pasikeitimo, nei kurie tai mazgai, nei kaip toli jie nuo roboto ir pan.

Kūrėjai pristatė dvi D*Lite versijas, pirmoji paprastesnė, antroji modernesnė. Labai neišsiplečiant, kadangi tai yra LPA* pagrindu sukurtas algoritmas, panagrinėsim tik jų skirtumus, pasitelkę pseudo kodus. Pirmosios versijos pseudo kodas:

```
procedure CalculateKey (s)
return [min (g(s), rhs(s)) + h(s, s_starto); min (g(d), rhs(s))];
procedure Initialize ()
U = ∅;
for all s ∈ S  rhs (s) = g(s) = ∞;
rhs (s_tikslo) = 0;
U.Insert (s_tikslo, CalculateKey(s_tikslo));
procedure Update Vertex (u)
if (u ≠ s_tikslo) rhs (u) = min_{s`∈Succ(u)} (c(s`,u) + g(s`));
if (u ∈ U) U.Remove (u)
if (g(u) ≠ rhs (u)) U.Insert (u, CalculateKey(u));
procedure ComputeShortestPath ()
while (U.Topkey () CalculateKey (s_starto) OR rhs (s_starto) ≠ g(s_starto))
        u = U.Pop ();
        if (g(u) > rhs (u))
            g (u) = rhs (u);
            for all s ∈ Pred (u) UpdateVertex (s);
        else
           g (u) = ∞;
        for all s ∈ Pred (u) ∪ {u} UpdateVertex (s);
procedure Main ()
Initialize ();
ComputeShortestPath ();
while (s_starto ≠ s_tikslo)
   /* if (g(s_starto) = ∞, nėra jokio galimo kelio*/
   s_starto = arg min_{s`∈Succ (Sstarto)}(c(s_start, s`) + g(s`)); /*mazgas prieš kliūtį padaromas nauju pradiniu mazgu*/
   move to s_starto;
   Scan graph for changed edge costs;
   if any edge costs changed
        for all directed edges (u, v) with changed edge costs
            Update the edge cost c(u, v);
            UpdateVertex (u)
        for all s ∈ U
            U.Update (s, CalcKey (s));
        ComputeShortestPath ();
```

Pirmos 4-ios procedūros - **CalcKey**(), **Initialize**(), **UpdateVertex**(u), **ComputeShortestPath**() - pasikeičia nežymiai : reikia pastebėti, kad žodis *starto* susikeičia vietomis su *tikslo*, taip pat vietoje mazgų pasekėjų *(Succ)* atsiranda mazgai pirmtakai *(Pred)*. Taip yra todėl, kad D*Lite algoritme kelio paieška vykdoma iš tikslo mazgo į pradinį. Pasikeičia ir pagrindinė procedūra **Main**. Ji išplėsta, kad robotas galėtų pajudėti, tada perskaičiuojami mazgų raktai prioritetų sąraše esantiems mazgams. Tai būtina atlikti dėl to, kadangi robotui judant keičiasi euristika, kadangi ji apskaičiuojama atsižvelgiant į tai, kuriame mazge robotas yra. Joje pirmiausia atliekami inicializavimo veiksmai – priskiriamos rhs(s) ir g(s) vertės, sudaromas pradinis prioritetų sąrašas U. Pirmiausia vietiškai nesuderintas yra tik tikslo mazgas, tad jis įtraukiamas į prioritetų sąrašą su pagal prieš duotą formulę apskaičiuotu raktu. Toliau atliekama tokia veiksmų seka : jei robotas dar nepasiekė tikslo mazgo, atnaujinamas s_{starto} mazgas, kad atspindėtų einamąjį roboto mazgą, kitaip tariant, s_{starto} yra priskiriamas einamasis mazgas. Toliau tikrinama, ar nėra briaunų verčių tarp mazgų pasikeitimų. Jei pasikeitimų

randama, visoms su pasikeitimams susijusioms briaunoms atnaujinamos c(u,v) (atstumo tarp dviejų kaimyninių mazgų) vertės , tada atnaujinamos mazgų, kuriuos paveikė pasikeitimai, rhs vertės bei perskaičiuojami raktai, tuo pačiu atnaujinamas ir prioritetų sąrašas. Galiausiai, yra perskaičiuojami visų saraše esančių mazgų raktai, perskaičiuojamas trumpiausias kelias, ir algoritmas vėl kartojamas.

Patobulintos D*Lite versijos pseudo kodas:

```
procedure CalculateKey (s)
return [min (g(s), rhs(s)) + h(s, s_starto) + k_m; min (g(d), rhs(s))];
procedure Initialize ()
U = ∅;
k_m = 0;
for all s ∈ S  rhs (s) = g(s) = ∞;
rhs (s_tikslo) = 0;
U.Insert (s_tikslo, CalculateKey(s_tikslo));
procedure Update Vertex (u)
if (u ≠ s_tikslo) rhs (u) = min_{s`∈Succ(u)} (c(s`,u) + g(s`));
if (u ∈ U) U.Remove (u)
if (g(u) ≠ rhs (u)) U.Insert (u, CalculateKey(u));
procedure ComputeShortestPath ()
while (U.Topkey () CalculateKey (s_starto) OR rhs (s_starto) ≠ g(s_starto))
        k_old = U.TopKey ();
        u = U.Pop ();
        if (k_old < CalcKey (u))
            U.Insert (u, CalcKey (u));
        else if (g(u) > rhs (u))
            g (u) = rhs (u);
            for all s ∈ Pred (u) UpdateVertex (s);
        else
           g (u) = ∞;
        for all s ∈ Pred (u) ∪ {u} UpdateVertex (s);
procedure Main ()
s_last = s_starto;
Initialize ();
ComputeShortestPath ();
while (s_starto ≠ s_tikslo)
    /* if (g(s_starto) = ∞, nėra jokio galimo kelio*/
    s_starto = arg min_{s`∈Succ (Sstarto)}(c(s_start, s`) + g(s`)); /*mazgas prieš kliūtį padaromas nauju pradiniu mazgu*/
    move to s_starto;
    Scan graph for changed edge costs;
    if any edge costs changed
        k_m = k_m + h(s_last, s_starto);
        for all directed edges (u, v) with changed edge costs
            Update the edge cost c(u, v);
            UpdateVertex (u)
        for all s ∈ U
            U.Update (s, CalcKey (s));
        ComputeShortestPath ();
```

Pirmos versijos D*Lite algoritmo trūkumas toks, kad pasikartojantis prioritetų sąrašo mazgų perrikiavimas gali atimti daug laiko, kadangi dažnai jame būna didelis kiekis mazgų. Antroji versija naudoja paieškos metodą, paimtą iš D* algoritmo, kuriuo yra išvengiama pakartotinio sąraše esančių mazgų perrikiavimo. Skirtumai tarp pirmos ir antros D*Lite versijų šiame pseudo kode parodyti paryškintu šriftu. Dabar euristika h(s,s`) turi būti neneigiama bei abipusiai (tiek

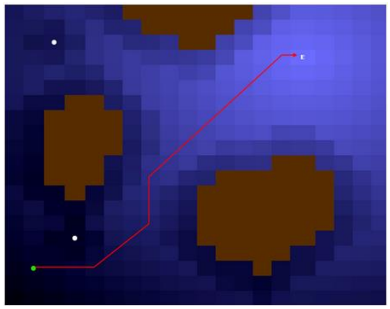
kryptimi į priekį, tiek atgal) suderinama, t.y., $h(s,s``) \leq h(s,s`)+h(s`,s``)$ visiems $s,s`,s`` \in S$. Tam pat turi būti tenkinama sąlyga, nepriklausomai kuris mazgas yra tikslo, kad $h(s,s`) \leq c^*(s,s`)$ visiems mazgams $s,s` \in S$, čia $c^*(s,s`)$ apibūdina trumpiausio kelio kainą tarp s ir s` mazgų, kai $s \in S$ ir $s` \in S$. Euristika su šiomis savybėmis atitinka pirmoje algoritmo versijoje turinčias atitikti savybes.

Antroji D*Lite versija naudoja raktus, kurie yra žemesnės raktų, naudojamų pirmojoje versijoje atitinkamiems mazgams, ribos. Jų inicializacija tokia pati, kaip ir pirmoje versijoje. Kai robotas pajuda iš s mazgo į kurį nors s` mazgą (pasekėją), kuriame nustatomas briaunų verčių pasikeitimas, pirmas raktų komponentas daugiausia gali sumažėti dydžiu h(s,s`) (antrasis komponentas nepriklauso nuo euristikos ir todėl išlieka nepakitęs). Toliau, norint išlaikyti žemesnes ribas, D*Lite algoritmas turi iš visų prioritetų sąraše esančių mazgų raktų pirmo komponento atimti vertę h(s,s`). Ir kol h(s,s`) yra vienoda visiems sąraše esantiems mazgams, jų eilės tvarka sąraše nesikeičia tol, kol nėra atliekama atimtis. Kai apskaičiuojamos naujos raktų vertės, jos yra h(s,s`) dydžiu mažesnės lyginant su atitinkamų mazgų raktais prioritetų sąraše. Todėl prie pirmų raktų komponenčių turi būti pridėtas dydis h(s,s`). Jei robotas pajuda dar kartą ir dar kartą nustato pasikeitimus, konstantos turi būti susumuojamos. Tai atliekama panaudojant kintamąjį $\mathbf{k_m}$, kitaip dar vadinamą rakto modifikatoriumi. Todėl visada, kai apskaičiuojamas naujas raktas, šis kintamasis privalo būti pridedamas prie pirmosios rakto komponentės. Tokiu būdu prioritetų sąraše mazgų prioritetas nesikeičia robotui pajudėjus, dėl to sąraše mazgų nereikia iš naujo perrikiuoti. Patys raktai, iš kitos pusės, yra žemosios ribos atitinkamų raktų pagal pirmąją versiją, kai pirmosios raktų komponentės yra padidinamos dydžiu k_m. Ši savybė yra išnaudojama modifikuojant **ComputeShortestPath**() procedūrą tokiu būdu: kai procedūra išmeta mazgą u su mažiausiu raktu k_{old} =**U.TopKey**() iš prioritetų sąrašo, panaudojama **CalcKey**() apskaičiuoti, kokį raktą jis galėjo turėti. Jei k_{old} < **CalcKey**(u), išmestasis mazgas grąžinamas atgal į sąrašą su **CalcKey**() apskaičiuotu raktu. Tokiu būdu patvirtinama, kad visų prioritetų sąraše esančių mazgų raktai yra žemesnės atitinkamų mazgų pagal pirmą D*Lite versiją ribos po pirmos raktų komponentės padidinimo dydžiu k_m. Jei k_{old} > **CalcKey**(u), tuomet laikoma, kad k_{old}=**CalcKey**(u), kadangi k_{old} buvo žemesnė **CalcKey**() apskaičiuotos vertės riba. Šiuo atveju, **ComputeShortestPath**() atlieka tokias pat operacijas mazgams u kaip ir pirmoje versijoje. Tai reiškia, kad antroji algoritmo versija turi daug panašumu su pirmąja. Galima teigti, kad antroji versija daugiausia gali išplėsti mazgą 2 kartus, t.y., ne daugiau kaip 1 kartą, kai jie yra visiškai nepriderinti, bei daugiausia 1 kartą, kai jie yra vietiškai perderinti, ir taip užbaigiamas mazgų verčių atnaujinimas. Toliau galima apskaičiuoti trumpiausią kelią pradedant pradiniu mazgu ir visada pasirenkant tokį mazgo s pasekėją s`, kuris minimizuotų c(s,s`) +g(s`) tol, kol bus pasiektas tikslo mazgas.

1.14. Potencialų laukų kelio paieškos algoritmas

Potencialų laukų algoritmo autonominio roboto navigacijai esmė yra traukiančių ir atstumiančių potencialų priskyrimas. Traukiantis potencialas priskiriamas roboto tikslui, o stumiantis potencialas kiekvienai iš kliūčių, esančių aplinkoje [2].

Potencialai – traukiantis ir atstumiantis – sukuria apie save lauką, kuris tolstant nuo to taško, silpsta, kol galiausiai pranyksta. Judančio objekto atžvilgiu veikia dvi jėgos – traukianti ir atstumianti, tokiu atveju, kiekvienoje pozicijoje galima rasti atstojamąją jėgą, kurios kryptimi objektas turėtų toliau judėti. 32 paveiksle vaizduojamas optimalaus kelio potencialų laukų metodu radimas: čia šviesesni laukeliai - traukiantis, o tamsesni - atstumiantis potencialas.



32 pav. Trajektorijos radimas potencialų laukų metodu.

Šio metodo privalumai yra tie, kad optimali trajektorija yra skaičiuojama ir įvertinama objektui judant. Šiuo metodu iškart įvertinamas visas kelias, ir trajektorija keičiama tik pasikeitus aplinkai. Taip lengviau susidorojama su dinamika, jeigu yra judančių kliūčių, ir sutaupoma laiko skaičiavimams, lyginant su kitais algoritmais [3].

Šis metodas jau buvo pritaikytas keletu skirtingų atvejų roboto navigacijai. Paprasčiausias iš atvejų – kai robotas juda žinomoje aplinkoje, kur tikslui ir kliūtims jau priskirti potencialai. Kai kliūtys nėra iš anksto žinomos, potencialai turi būti priskiriami robotui judant, ir aptinkant naujas kliūtis. Tokiu atveju, kai yra vienas traukiantis potencialas priskirtas tikslui, ir visi kiti atstumiantys potencialai - kliūtims, potencialų laukų metodas turi vieną svarbų apribojimą - kai kurioms kliūčių konfigūracijoms, gali būti neįmanoma sukurti tinkamos atstojamosios jėgos, kad kliūčių būtų išvengta.

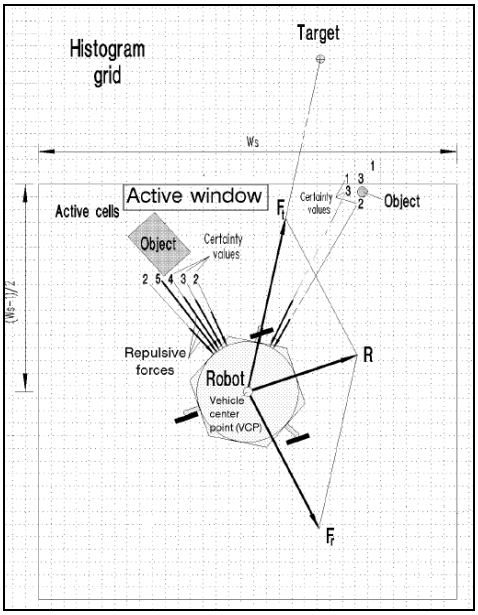
Kitas atvejis - kai naudojami keli traukiantys potencialai, bei laipsniškas atstumiančių potencialų įterpimas, robotui aptinkant naujas kliūtis, buvo įgyvendintas siekiant išvengti didelių kliūčių nežinomoje aplinkoje [2].

The virtual force field - virtualių jėgų laukas, tai potencialų laukų metodu paremtas metodas, specialiai pritaikytas realaus laiko kliūčių išvengimui greitaeigiams robotams. VJL metodas suteikia greitą, nenutrūkstamą, sklandų judėjimą tarp netikėtų kliūčių.

VJL metodas naudoja dviejų matmenų Dekarto sistemą (tinklelį) kliūtims atvaizduoti (žr. 33 pav.). Kiekvienas laukelis (i,j) šioje sistemoje turi užtikrintumo reikšmę (*certainty value*) $C_{i,j}$, kuri parodo algoritmo užtikrintumą dėl kliūties egzistavimo šiame laukelyje. Užtikrintumo reikšmės laukeliuose didėja, kai ultragarsinis ar kitoks jutiklis aptinka kliūtį tame laukelyje.

Tuo pačiu metu yra pritaikomas ir potencialų laukų principas. Kiekvienas laukelis, kuriame yra kliūtis, sukuria atstumiančią jėgą roboto link, kuri yra atvirkščiai proporcinga atstumui tarp roboto centro ir objekto. Taip pat, taikinys (tikslas) sukuria jėgą, kuri veikia nuo roboto, tikslo link - jėga yra priklausoma nuo tikslo koordinačių bei atstumo iki roboto.

Sudėjus visas veikiančias jėgas, gauname atstojamąją jėgą - vektorių, kurio dydis ir kryptis parodo robotui, kuria kryptimi ir kokiu greičiu važiuoti optimalia trajektorija [1] (žr. 33 pav).



33 pav. Virtualių jėgų metodu planuojama trajektorija [1].

1.15. Genetinis kelio paieškos algoritmas

Trumpiausio kelio paieškos problema gali būti sprendžiama pasitelkiant ir genetinius algoritmus.

Šiuo algoritmu pirmiausia yra atliekamas chromosomų užkodavimas. Chromosoma apibūdina galimą sprendimą - kandidatą. Ji susideda iš pradinio mazgo, tikslo mazgo bei mazgų kuriais robotas turi judėti. Visi šie mazgai vadinami chromosomos genais. Skirtingi kodavimo metodai

sukuria skirtingas chromosomas. Dažniausiai yra sukuriame dvejetainė eilutė, tačiau naudojamas ir dešimtainis kodas. Pastarasis yra labiau lankstus, reikalauja mažiau atminties.

Pradinė populiacija įprastai sukuriama atsitiktinai. Kai kurios chromosomos gali sudaryti nepraeinamus kelius, kai mazgais parenkamos kliūtys. Optimalus sprendimas randamas atliekant genetines operacijas. Genetinio algoritmo (nuo čia jį vadinsime GA) metodas dėl atsitiktinumo kriterijaus prailgina optimalaus kelio paieškos laiką.

Genetiniame algoritme trumpiausiu keliu laikoma tikslo funkcija. Sudarytos chromosomos tikslo funkcijos vertė gali būti išreiškiama taip [17]:

$$f = \begin{cases} \sum_{i=1}^{n-1} d(p_i, p_{i+1}), \; jei \quad kelias \quad praeinamas \\ \sum_{i=1}^{n-1} d(p_i, p_{i+1}) + bauda, \; jei \quad kelias \quad nepraeinamas \end{cases}$$

čia $d = (p_i, p_{i+1}) = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$. f yra tinkamumo funkcija, p_i yra i-atis chromosomos geras, n yra chromosomos ilgis, d yra atstumas taip dviejų mazgų (iš formulės matyti, kad tai Euklidinis atstumas), x_i, y_i – roboto einamosios koordinatės, x_{i+1}, y_{i+1} – roboto sekančios koordinatės. Roboto kryptis gali būti išreiškiama formule:

$$\alpha = arctg\left(\frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}\right)$$

Kitaip tariant, tikslo funkcijos vertė yra nusakoma kaip atstumų tarp kiekvieno mazgo , esančio kelyje, suma. Jei sugeneruotoje chromosomoje (kelyje) yra kliūtis, tikslo funkcijos vertei yra pridedama bauda, kuri turėtų būti didesnė nei maksimalus kelio ilgis žemėlapyje. Kelio paieškos algoritmas ieško chromosomos, kurioje bauda būtų eliminuota.

Pagrindinis genetinio algoritmo principas – išlikti turi geriausi pradinės chromosomos genai, kad juos galima būtų perkelti į sekančią kartą. Šiame etape yra svarbu atrinkti tokius genus. Išrinkimo procesą galima išskaidyti į tris dalis:

1. Apskaičiuojama tikslo funkcijos vertė visoms rastoms chromosomoms
2. Atitinkamai pagal tikslo funkcijos vertes, chromosomoms priskiriamos tinkamumo funkcijos vertės
3. Pagal priskirtą tinkamumo funkcijos vertę išrenkamos chromosomos ir paruošiamos poravimui, siekiant sukurti naujas chromosomas.

Toliau vykdomas chromosomų kryžminimas ir/arba mutacijos. Kryžminimo metu įprastai yra sukeičiama dalis vienos chromosomos genų su kitos chromosomos genais. Metodai gali būti įvairūs – sukeičiama pusė genų, kas antras, pirmi genai ir pan. Mutacijos metu chromosomoms kandidatėms po kryžminimo yra atliekamos atsitiktinės mutacijos. Tai iš esmės chromosomos geno pakeitimas, tolygiai atliekamas visiems populiacijos genams, su tam tikra mutacijos

tikimybe. Mutacijos operacija padidina populiacijos įvairovę, leidžia išvengti per ankstyvaus algoritmo konvergavimo.

Dažniausiai GA yra atliekama atsitiktinė genų mutacija. Toks būdas gali sukurti chromosomų su nepraeinamais keliais. Dėl to kelio paieška pailgėja, padidėja generacijų skaičius. Yra atlikta tyrimų, kurie leistų išspręsti tokias problemas. Vienas iš sprendimo būdų – patikrinti naujai sukurtą chromosomą, ar ja apibūdintas kelias yra praeinamas. Jei ne, chromosomai atliekama naujos mutacijos tol, kol kelias taps praeinamas [17]. Šio kelio paieškos algoritmo tikslas – pasirinkus tam tikrą mutacijos tipą, rasti kelią (chromosomą), kurios tinkamumo funkcijos vertė bus mažiausia.

1.16. Kolonijinis kelio paieškos algoritmas

Algoritmas sukurtas 1992 m., Marco Dorigo. Jis paremtas tikrų skruzdėlių elgesiu, kai jos ieško maisto. Nustatyta, kad jos savo kelyje palieka tam tikrą feromonų pėdsaką. Feromonas pritraukia kitas skruzdėles, kurios eidamas tuo pačiu keliu padidina paliekamo feromono koncentraciją. Tuo keliu, kuriuo keliaujant buvo rasta maisto, pradeda keliauti vis daugiau skruzdėlių, ir feromono koncentracija stiprėja. Kelius – aklavietes, kurie nenuveda prie maisto, renkasi vis mažesnis skruzdėlių skaičius, taip laikui bėgant feromono koncentracija mažėja, išgaruoja. Taipogi, kuo trumpesnis kelias iki maisto, tuo jame feromono koncentracija didesnė (lėčiau išgaruoja), taigi šį kelią renkasi vis daugiau skruzdėlių. Galiausiai jos pradeda keliauti trumpiausiu keliu. Jame feromono koncentracija, lyginant su aklavietėmis ar ilgesniais keliais, yra gerokai didesnė [18].

Buvo pasiūlyti 2 kolonijinio algoritmo variantai. Vienas iš jų yra tikslo siekimo, kitas – sienos sekimo algoritmas. Trumpai šiuos algoritmus galima aprašyti taip [19]:

Tikslo siekimo algoritmas (pseudokodas):

Inicializavimas: Užkrauti žemėlapį, inicializuoti kintamuosiuos
Iteracijos
    Einamajai iteracijai
    Kol yra judančių skruzdėlių
        Einamajai skruzdėlei
        Apskaičiuoti patekimo į kiekvieną kaimyninį tašką (žemėlapyje) tikimybę
        Išsaugoti esamą poziciją ėjimų istorijoje
        Kontroliuoti skruzdėlės atstumą nuo sienos
        Apsaugoti nuo grįžimo atgal
        Apsaugoti nuo 4 langelių (4 *square*) kilpų (grįžimo į tą patį langelį)
        Priskirti tikimybe pagrįstą naują poziciją
        Perkelti skruzdėlę
        Patikrinti, ar pasiektas tikslas (vykdomas tikslo kriterijus)

Pabaiga – einamajai skruzdėlei

Pabaiga – kol yra judančių skruzdėlių

Feromonų išgaravimas

Feromonų deponavimas (*depositing*)

Pabaiga – einamajai iteracijai

Sienos sekimo algoritmas (pseudokodas):

Inicializavimas: Užkrauti žemėlapį, inicializuoti kintamuosiuos

Iteracijos:

Einamajai iteracijai

Kol yra judančių skurzdėlių

Einamajai skruzdėlei

Priskiriama tikimybė pateikti į kiekvieną kaimyninį tašką

Išsaugoti esamą poziciją ėjimų istorijoje

Kontroliuoti mažiausią skruzdėlės atstumą nuo sienos

Kontroliuoti didžiausią skruzdėlės atstumą nuo sienos

Apsaugoti nuo grįžimo atgal

Apsaugoti nuo 4 langelių (4 *square*) kilpų (grįžimo į tą patį langelį)

Priskirti naują poziciją

Perkelti skruzdėlę

Pabaiga – einamajai skruzdėlei

Pabaiga – kol yra judančių skruzdėlių

Feromonų deponavimas

Pabaiga – einamajai iteracija

Paslėpti neištyrinėtus kelius

Tikimybių apskaičiavimui naudojamos tokios formulės:

Tikslo siekimo algoritmui :

$$tikimybė\_patekti\_į\_X\_tašką = \frac{Feromono\_kiekis\_X}{\sum kaimynų\_feromonai}$$

Sienos sekimo algoritmui:

$$tikimybė\_patekti\_į\_X\_tašką = \frac{\sum kaimynų\_feromonai}{Feromono\_kiekis\_X}$$

Šios tikimybės yra viena kitai atvirkščios. Taip pat reikia apskaičiuoti feromonų deponavimą, tai atliekama pagal funkcijas:

Abiems algoritmams, sėkmingam keliui (pasiekus tikslą):

$deponuoti\_feromonai = \left(\frac{1}{N}\right)^{\frac{P}{2}}$, čia N – kiek žingsnių atlikta nueitame kelyje, P – feromonų deponavimo konstanta.

Neigiamas feromonų kiekis gali būti deponuojamas pagal formulę:

$$deponuoti\_feromonai = \left(\frac{-0.001}{žemėlapio\_dydis}\right),$$

Dar reikia apskaičiuoti feromonų išgaravimą pagal formulę:

$R = E * P$, čia R = liekantis feromonų kiekis, E – feromonų garavimo konstanta, P – šiuo metu taške esantis feromonų kiekis.

1.17. Algoritmų palyginimas

Atlikę moksline literatūra paremtą kelio paieško algoritmų apžvalgą, algoritmų palyginimui galime teigti, kad:

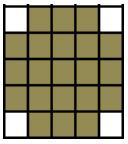
1) Paieškos į gylį, paieškos į plotį algoritmai tinkami tada, kai nėra žinoma tikslo taško vieta, paieška vykdoma „aklai“, ieškant bet kokio kelio iki tikslo. Tai nėra optimalaus/trumpiausio kelio radimas.
2) Bellman-Ford, Floyd-Warshall, Dijkstra algoritmai yra lėti, ypač kuo didesnis žemėlapis ar grafas, tuo algoritmas lėtesnis. Priežastis viena ir ta pati – būtina ištirti visą žemėlapį ar grafą.
3) Genetinis, kolonijinis algoritmai yra lėtesni už euristinius algoritmus, kadangi turi atsitiktinumo faktorių, nors ir gali rasti trumpiausius kelius.
4) Euristiniai algoritmai – A*, D* ir jų modifikacijos - iš aprašytųjų algoritmų yra greičiausi, kadangi kelio paieška vykdoma geriausios pirmos paieškos principu – einama keliu, kuris veda arčiausiai tikslo. Dažniausiai kelio paieškai užtenka ištirti tik dalį žemėlapio.

Tolimesniam tyrimui pasirinksime A* bei D*Lite algoritmus ir palyginsime jų rezultatus.

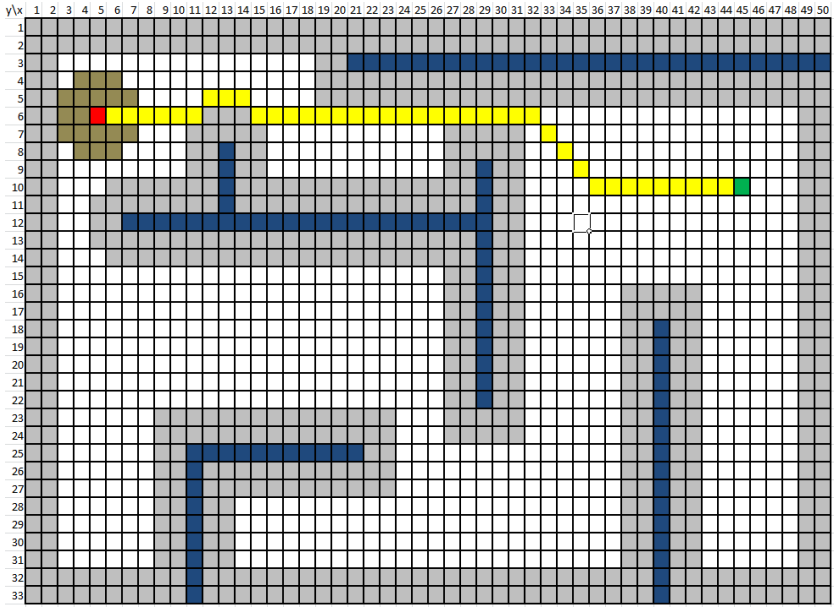
1.18. Žemėlapio sudarymas

Prieš pradėdami tyrimą algoritmų palyginimui, trumpai aptarsime roboto aplinkos žemėlapio sudarymo principus. Žemėlapyje darysime tinklelio tipo, nes tokiame žemėlapyje ieškoti trumpiausio kelio yra lengviau, be to, iš tokio žemėlapio, esant reikalui, galima nesunkiai sudaryti ir grafą ar medį. Be to, jei žemėlapis yra gana didelis, tarkim 800x600 taškų, tam reikėtų didelio grafo, o nagrinėti žemėlapį kaip stačiakampę matricą yra patogiau ir paprasčiau. Dar

vienas tokio žemėlapio pliusas – žinant, kad robotas, tiksliau jo išmatavimai, yra didesni nei vienas mazgas (žr. 34 pav.), atitinkamai tinklelio tipo žemėlapyje galima papildomai sužymėti mazgus, į kuriuos patekimas sąlygotų susidūrimą su kliūtimis (žr. 35 pav.).



34 pav. Roboto dydis (pilkos spalvos langeliai)



35 pav. Tinklelio tipo žemėlapis. Mėlynos spalvos langeliai – kliūtys, pilkos spalvos – pavojingos zonos – „virtualios“ kliūtys, į kurias robotas neturi patekti.

2. TYRIMŲ DALIS

2.1. A* ir D* Lite algoritmų rezultatų palyginimas

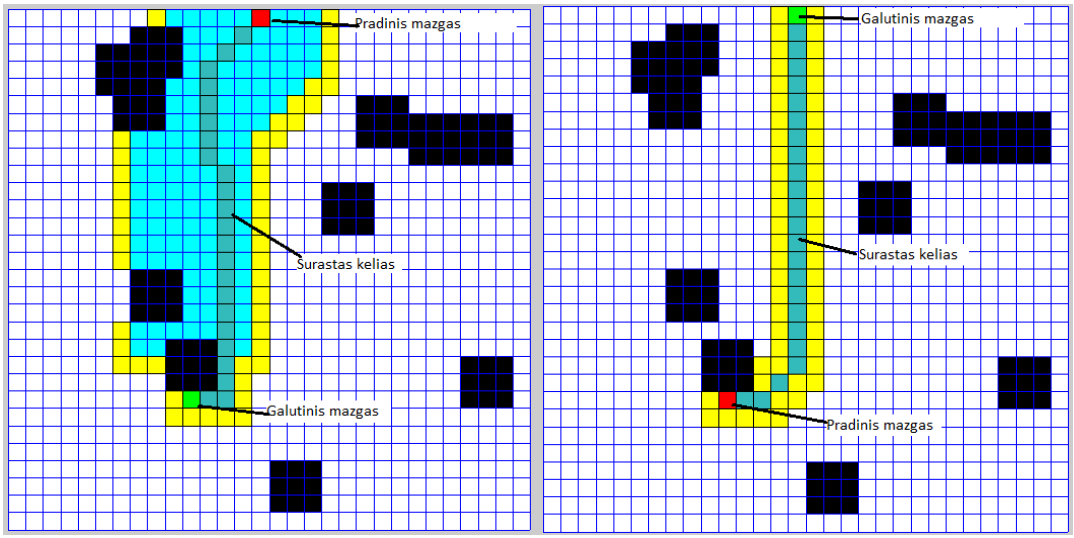
Tyrimams pasirinkome 2 algoritmus – A* bei D*Lite. Algoritmus suprogramuosime Matlab programinės įrangos paketo funkcijomis, atliksime jų palyginimą pagal įvairius parametrus, pagal gautus rezultatus tolimesniems darbams su realiu E-puck robotu pasirinksime vieną šių algoritmų.

Trumpai apibrėšime parametrus, naudojamus šiuose algoritmuose:

- Euristinis atstumas iki tikslo mazgo h (x): kiekvienam praeinamam žemėlapio taškui x apskaičiuojamas Manheteno atstumas iki tikslo mazgo. Parametras naudojamas abiejuose algoritmuose. Pastaba : D*Lite algoritme euristika skaičiuojama iki pradinio mazgo.

- Nueito kelio atstumas g (x) : Euklido kelio atstumas nuo pradinio iki tiriamojo mazgo. Parametras naudojamas abiejuose algoritmuose.
- f (x) atstumas f(x) = g(x) + h(x) : nueito kelio g atstumas iki x mazgo + šio mazgo euristika iki tikslo taško. Naudojamas A* algoritme.
- rhs (x) atstumas : mažiausias kelias iš mazgo x pirmtakų. Naudojamas D*lite algoritme.

Viena esminių skirtumų tarp A* ir D*Lite algoritmų : A* kelio paieška vykdoma iš pradinio mazgo į galutinį, o D*lite atvirkščiai – iš galutinio į pradinį. Abiejų algoritmų paieškos pradinio mazgo g vertė prilyginama 0, surandami mazgai, kuriuos galima pasieki. A* algoritmui sudaromas uždarytas bei atviras sąrašai, D*lite algoritmui : prioritetų sąrašas. Į A* atvirą sąrašą įtraukiami mazgai su apskaičiuota f verte, prieš tai apskaičiavus mazgų, kuriuos galima pasiekti iš einamojo mazgo, g vertes ir nurodant kiekvienam šių mazgų trumpiausio kelio pirmtakus mazgus. Iš atvirojo sąrašo išsirenkamas mazgas **s** su mažiausia **f** verte, kad irgi būtų tenkinama sąlyga, jog h(**s**) <= h(**s-1**), čia **s**-1 – paskutinis prieš tai pasirinktas mazgas. Jei sąlyga tenkinama, mazgas išmetamas iš atviro sąrašo ir įtraukiamas į uždarą sąrašą. Po to jis jau nebebus nagrinėjamas. Jei sąlyga netenkinama, yra plečiamas atviras sąrašas, į jį įtraukiami visi kaimyniniai mazgai mazgams iš šio sąrašo, šie mazgai atnaujinami, buvę mazgai išmetami ir įtraukiami į uždarytą sąrašą. Paieškos ciklas kartojamas, kol pasirinktas mazgas yra tikslo mazgas, arba kol atvirasis sąrašas taps tuščias (šiuo atveju kelio nėra). Netinkamai parenkant įtraukiamus į atvirą sąrašą mazgus, galima gauti tokį rezultatą (36 pav, a). Taip pat skirtingas kelias gali būti randamas sukeitus pradinį ir tikslo mazgus vietomis (36 pav, b).

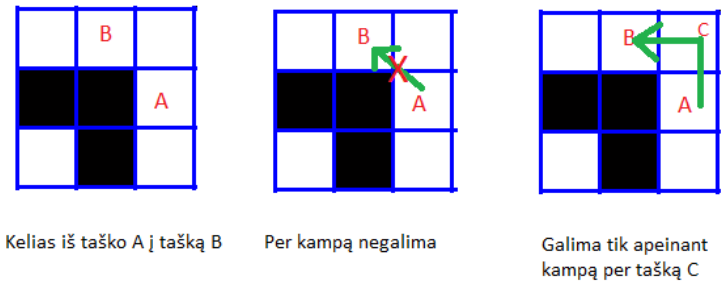


36 pav. a - blogai surastas kelias (kairėje), b- gerai surastas kelias (dešinėje)

D*Lite kelio paieškos atveju iš prioritetų sąrašo išsirenkamas mazgas , turintis mažiausią raktą, kuris apskaičiuojamas :

$k(s)=[k_1(s);k_2(s)]$, čia $k_1(s)=\min(g(s), rhs(s) + h(s,s_{tikslo})+k_m)$, $k_2(s)=\min(g(s),rhs(s))$, čia k_m – rakto modifikatorius, kurį aptarėme anksčiau. Išsirinkus mazgą, jis yra atnaujinamas, apskaičiuojama jo rhs(s) ir g(s) vertės, jei jos lygios, mazgas iš prioritetų sąrašo šalinamas, kitu atveju paliekamas. Atnaujinamos visų mazgų, kuriuos galima pasiekti iš s mazgo, g, rhs vertės, perskaičiuojami raktai. Algoritmas tęsiamas, kol išrenkamas mazgas yra tikslo mazgas arba prioritetų sąrašas tampa tuščias.

Pasirenkant kaimyninius mazgus, būti patikrinti, ar kelias neina tiesiogiai „per kampą“, kaip parodyta 37 pav.

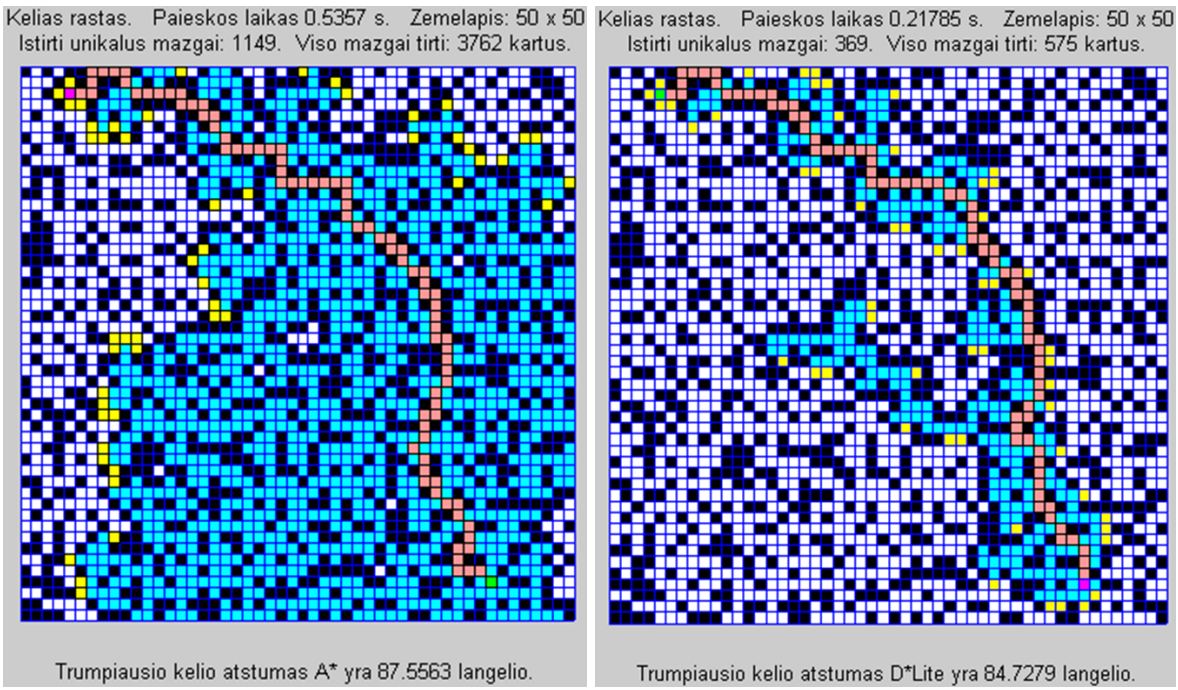


37 pav. Galimas ir negalimas kelio per kampą variantas

Abiejų algoritmų palyginimui naudosime atsitiktinai sugeneruotą 50x50 dydžio žemėlapį, kuriame kliūtys sudarys 36% - 900 iš 2500 langelių (38 pav). Kiekvienu algoritmu trumpiausio kelio paiešką atliksim po 10000 kartų – 5000 kartų į vieną pusę, tada 5000 kartų į pusę. Kiekvienu kartu žemėlapyje atsitiktine tvarka 10 kliūčių bus pakeistos į praeinamus langelius, o 10 praeinamų langelių bus pakeista kliūtimis. Algoritmų palygimui kelio paieškos laiko nevertinsime, kadangi priklausomai nuo kliūčių išsidėstymo ir skaičiavimo krypties jis gali būti neadekvatus. Lyginsime tokius parametrus:

- Viso ištirtų mazgų skaičius S1
- Viso atliktų atnaujinimų skaičius S2 (kai mazgui perskaičiuojama jo g vertė, jis įtraukiamas arba išmetamas iš mazgų sąrašo)
- Kiek kartų kuris algoritmas rado trumpesnį kelią S3
- Kiek kartų rasto kelio ilgis sutapo S4
- Vidutinis trumpiausio surasto kelio ilgio skirtumas abejiems algoritmams S5
- Vertinsim tik tuos atvejus, jei kelias bus surastas. Nerandant kelio, esant tam pačiam kliūčių išsidėstymui, dėl skirtingų algoritmų kelių paieškos krypčių (A* iš pradžios į pabaigą, D* Lite iš pabaigos į pradžią), rezultatai gali stipriai skirtis ir neduos tinkamos informacijos palyginimui.

38 pav. Testavimui sugeneruotas 50x50 langelių dydžio atsitiktinis žemėlapis su 36% kliūčių tankumu, pradinis taškas x=43, y=47, galutinis x=5 y=3.



39 pav. Pirminis kelios paieškos algoritmų palyginimas : kairėje – A*, dešinėje D* Lite

Po 10000 atliktų bandymų (kelias visais atvejais buvo surastas), gavome tokius rezultatus (2.1,2.2,2.3 lentelės):

2.1 lentelė: A* ir D*Lite algoritmų bandymų rezultatų palyginimas kelio suradimui, kelio paieška atlikta į abi puses (10000 bandymų):

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S1	1249	X	568	X
Vidutinis mazgų atnaujinimų skaičius S2	4014	X	901	X
Kiek kartų rastas trumpesnis kelias S3	54	0,54%	9712	97,12%
Kiek kartų rasto kelio ilgis sutapo S4	234	2,34 %	234	2,34%
Vidutinis trumpesnio rasto kelio skirtumas S5	1,22	X	4,37	X

2.2 Lentelė: A* ir D*Lite algoritmų bandymų rezultatų palyginimas kelio suradimui, kelio paieška atlikta iš [47,43] į [3,5] mazgą (5000 bandymų):

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S1	1153	X	415	X
Vidutinis mazgų atnaujinimų skaičius S2	3836	X	650	X
Kiek kartų rastas trumpesnis kelias S3	18	0,36%	4935	98,7%
Kiek kartų rasto kelio ilgis sutapo S4	47	0,94 %	47	0,94 %
Vidutinis trumpesnio rasto kelio skirtumas S5	1.89	X	4.37	X

2.3 lentelė: A* ir D*Lite algoritmų bandymų rezultatų palyginimas kelio suradimui, kelio paieška atlikta [3,5] į [47,43] mazgą (5000 bandymų):

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S1	1345	X	721	X
Vidutinis mazgų atnaujinimų skaičius S2	4191	X	1152	X
Kiek kartų rastas trumpesnis kelias S3	36	0,72%	4777	95,54%
Kiek kartų rasto kelio ilgis sutapo S4	187	3,74 %	187	3,74%
Vidutinis trumpesnio rasto kelio skirtumas S5	0.88	X	5.42	X

Kaip matyti iš šiose lentelėse pateiktų rezultatų, D*lite algoritmas visais parametrais yra pranašesnis už A* algoritmą. Papildomai atliksime dar 10000 bandymų – 5000 į vieną pusę, 5000 į kitą, tačiau žemėlapį kiekvieną kartą sugeneruosime atsitiktinį, kliūtys sudarys tuos pačius

36% žemėlapio ploto, abiem algoritmais surasime trumpiausią kelią, Tada kiekvieno surasto kelio viduryje įterpsime kliūtį, atliksime kelio perskaičiavimą nuo mazgo prieš , palyginsime, kiek kelio perskaičiavimui ištirta mazgų bei kiek kartų jie atnaujinti. Palyginsime tokius parametrus:

- Perskaičiuojam keliui rasti papildomai ištirtų mazgų skaičių S6
- Kiek kartų atnaujinti ištirti mazgai S7
- Kiek kartų kuris algoritmas rado trumpesnį kelią S8
- Kiek kartų rasto kelio ilgis sutapo S9
- Vidutinis perskaičiuoto kelio ilgio skirtumas abiems algoritmams S10

Po 10000 papildomų bandymų gauti tokie rezultatai (2.4, 2.5, 2.6 lentelės)

2.4 lentelė. Kelio perskaičiavimo A* ir D*Lite algoritmais įvertinimas, perplanuojant į abi puses (10000 bandymų)

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S6	660	X	156	X
Vidutinis mazgų atnaujinimų skaičius S7	2194	X	255	X
Kiek kartų rastas trumpesnis kelias S8	4484	44.84 %	5404	54.04 %
Kiek kartų rasto kelio ilgis sutapo S9	112	1,12 %	112	1,12 %
Vidutinis trumpesnio rasto kelio skirtumas S10	9.11	X	9.49	X

2.5 lentelė. Kelio perskaičiavimo A* ir D*Lite algoritmais įvertinimas, perplanavimas atliktas nuo kliūties į [3,5] mazgą (5000 bandymų)

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S6	656	X	164	X
Vidutinis mazgų atnaujinimų skaičius S7	2182	X	269	X
Kiek kartų rastas trumpesnis kelias S8	2177	43.54 %	2773	55.46 %
Kiek kartų rasto kelio ilgis sutapo S9	50	1,00 %	50	1,00 %
Vidutinis trumpesnio rasto kelio skirtumas S10	8.99	X	9.54	X

2.6 lentelė. Kelio perskaičiavimo A* ir D*Lite algoritmais įvertinimas, perplanavimas atliktas nuo kliūties į [47,43] mazgą (5000 bandymų)

Algoritmo pavadinimas/ Lyginamas parametras	A*		D*lite	
		% nuo bandymų skaičiaus		% nuo bandymų skaičiaus
Vidutinis ištirtų mazgų skaičius S6	665	X	148	X
Vidutinis mazgų atnaujinimų skaičius S7	2207	X	242	X
Kiek kartų rastas trumpesnis kelias S8	2305	46.10 %	2633	52.66 %
Kiek kartų rasto kelio ilgis sutapo S9	62	1,24 %	62	1,24 %
Vidutinis trumpesnio rasto kelio skirtumas S10	9,21	X	9,46	X

Kelio perplanavime taip pat geresnius rodiklius parodė D*Lite algoritmas. Galime pastebėti, kad kiekvieno atveju skirtingais algoritmais perplanuoto trumpesnio kelio vidurkis (S10 parametras) yra keliskart didesnis nei vidurkis, gautas tik kelio paieškos metu (S5 parametras). Taip yra todėl, kad kelio perplanavimui visada buvo imamas vidurinis rasto kelio mazgas. Skirtingais algoritmais rasto kelio trajektorija skiriasi, todėl skiriasi ir pasirinktas vidurinis mazgas, priklausomai nuo jo skiriasi ir perplanuoto kelio ilgis. Jei būti rastos sutampančios trajektorijos, tokiu atveju būtų didesnis ir sutampančio ilgio perplanuotų kelių kiekis.

2.2. Kelio paieškos A* ir D*Lite algoritmais palyginimas įvairiuose žemėlapiuose

Toliau pateiksime keletą detalesnių vaizdinių pavyzdžių, kaip surandamas ir /arba perplanuojamas kelias tokiame pat žemėlapyje abiem algoritmais. Žemėlapius parinksime įvairius, imituodami įvairias patalpas.

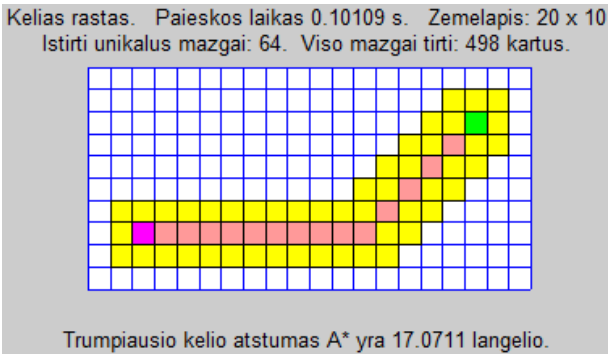
Pirmiausia palyginsime, kaip abu algoritmai suranda kelią žemėlapyje, kuriame nėra jokių kliūčių.

40, 41, 42, 43 paveiksluose pateiktas surastas A* bei D*Lite algoritmais bei kelio perplanavimas, kai žemėlapyje aptinkama nematyta kliūtis. Abiems algoritmams parinkti tie patys pradiniai ir galutiniai mazgai. Nesant kliūčių, matome, kad algoritmai suranda to paties ilgio, tačiau skirtingais langeliais einantį kelią – taip yra todėl, kad A* algoritmas kelio paiešką vykdo iš pradinio mazgo į galutinį, o D*Lite – iš galutinio į pradinį. Tai matyti 40 ir 41 paveiksluose. Šiuose žemėlapiuose, o taip pat ir sekančiuose, spalvos žymėjimui naudojamos taip:

- šviesi geltona spalva – mazgai, esantys atidarytame sąraše (A*) arba prioritetų sąraše (D*Lite)

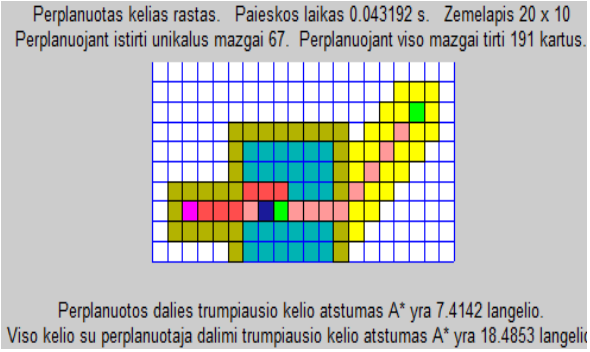
- šviesiai žydra spalva – ištirti mazgai, esantys uždarytame sąraše (A*) arba patekę ir išmesti iš prioritetų sąrašo (D*Lite)
- šviesiai rausva spalva – surastas kelias (abiem algoritmais)
- tamsesnė geltona spalva – mazgai, įtraukti į atidarytą sąrašą (A*) arba prioritetų sąrašą (D*Lite) kelio perplanavimo metu
- tamsesnė žydra spalva – ištirti mazgai, esantys uždarytame sąraše (A*) arba patekę ir išmesti iš prioritetų sąrašo (D*Lite) kelio perplanavimo metu
- tamsiai geltona spalva – likę prioritetų sąrašo mazgai (D*Lite), kai kelio perplanavimui dalis šio sąrašo mazgų paliekama
- tamsiai žydra spalva – likę ištirti mazgai (D*Lite), kai kelio perplanavimui dalis šių mazgų paliekama
- tamsesnė rausva spalva – surastas perplanuotas kelias (abiem algoritmais)
- tamsiai mėlyna spalva pažymėta iš anksto nežinoma kliūtis
- žalia spalva pažymėtas pradinis mazgas, taip pat mazgas, kurį pasiekus aptikta nematyta kliūtis, kitaip dar naujas pradinis mazgas
- avietine spalva pažymėtas galutinis mazgas
- juoda spalva – kliūtys
- balta spalva – pravažiuojami langeliai, kurių kelio paieškos metu neprireikė išnagrinėti.

42 ir 43 paveiksluose parodytas kelio perplanavimas abiem algoritmais. Kadangi keliai buvo surasti skirtingi – išsidėstymo atžvilgiu - , skiriasi ir perplanavimo rezultatas. Šiuo atveju kelio perplanavimui daugiau mazgų išnagrinėta A* algoritmu. Tuose langeliuose, kuriuose perplanuotas kelias sutampa su buvusiu keliu, jis nupiešiamas ant jo viršaus.



40 pav. Surastas kelias A* algoritmu

41 pav. Surastas kelias D*Lite algoritmu



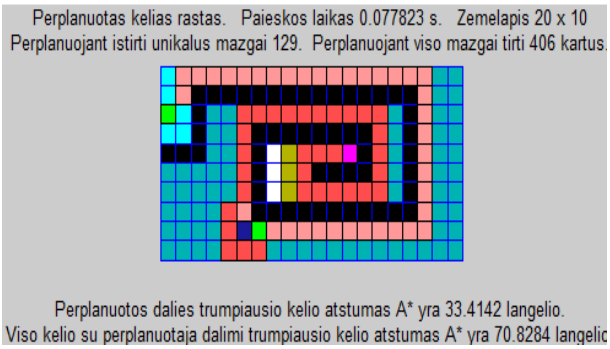
42 pav. Surastas perplanuotas kelias A* algoritmu

43 pav. Surastas perplanuotas kelias D*Lite algoritmu

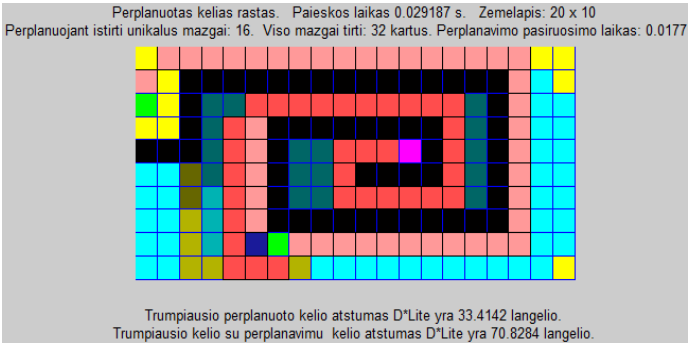
Realioje aplinkoje gali pasitaikyti spirale išdėstytos kliūtys. 44, 45 paveikluose pavaizduota, kaip tokiame žemėlapyje kelią suranda vienas ir kitas algoritmai. Matome, kad tokiame žemėlapyje abu algoritmai ištyrė praktiškai visą žemėlapį, ištirtų langelių (mazgų) skaičius beveik vienodas – 145 mazgai A* algoritmu ir 148 mazgai D*Lite algoritmu. Galima teigti, kad žemėlapyje esant ilgiems siauriems koridoriams, jei surastas kelias eina pro juos, tiek vienas, tiek kitas algoritmas vienodai tinkami kelio paieškai. Skirtumas tarp algoritmų matytis kelio perplanavime. Surastame kelyje įterpiama kliūtis, atliekamas kelio perplanavimas. 46, 47 paveiksluose parodytas kelio perplanavimas. Dalis perplanuoto kelio sutampa su pirminiu rastu keliu. Matome, kad D*Lite algoritmu kelio perplavimui reikia ištirti kelis kartus mažiau mazgų nei A* algoritmu – 16 mazgų D*Lite algoritmu ir 129 mazgai A* algoritmu. Taip yra todėl, kad D*Lite algoritmas kelio perplanavimui panaudoja kelio dalies nuo kliūties iki tikslo mazgo rezultatus – ištirtus mazgus bei mazgus prioritetų sąraše, tai realiai reikia ištirti tik mazgus, kurie reikalingi kliūties apėjimui. A* algoritmu kelio paieška, langelį pries kliūtį pažymėjus nauju pradiniu langeliu, vykdoma iš naujo, todėl mazgų reikia ištirti daugiau.

44 pav. Surastas kelias A* algoritmu, kai kliūtys sudaro spiralę

45 pav. Surastas kelias D*Lite algoritmu, kai kliūtys sudaro spiralę



46 pav. Surastas perplanuotas kelias A* algoritmu, kai kliūtys sudaro spiralę

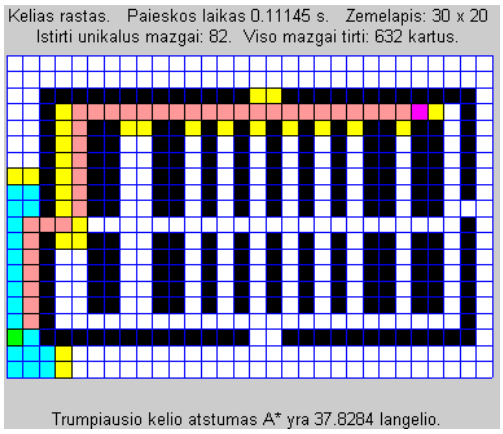


47 pav. Surastas perplanuotas kelias D*Lite algoritmu, kai kliūtys sudaro spiralę

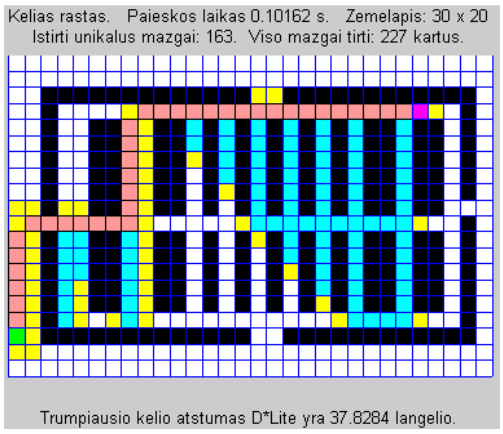
Realesni yra tokie žemėlapiai, kurie sudaryti pagal patalpoje esančių objektų – spintų, lentynų ir pan. – išsidėstymą. Palyginsime, kaip skiriasi ir kuom panašus kelio planavimas, perplanavimas tokiame žemėlapyje, kuriame daug atskirų kliūčių, ilgos sienos, daug skirtingų praėjimų.

48, 49 paveiksluose pateiktas surastas kelias žemėlapyje, vaizduojančiame nedidelį sandėlį. Matome, kad nors surasto kelio ilgis tiek vienu, tiek kitu algoritmu yra vienodas, dalis kelio eina skirtingais langeliais. Taip pat matyti, kad šį kartą A* algoritmui reikėjo išnagrinėti mažiau mazgų nei D*Lite algoritmui – 82 mazgai prieš 163 mazgus, skirtumas ~2 kartai. 50, 51 paveiksluose pateiktas surastas kelias tame pačiame žemėlapyje, tik pradiniai ir galutiniai mazgai sukeisti vietomis. Matome, kad dabar rezultatas gautas priešingas – A* algoritmui reikėjo išnagrinėti daugiau mazgų nei D*Lite algoritmui – 76 mazgai prieš 311 mazgų, skirtumas ~4 kartai. Vadinasi kelio paieškos rezultatas, ypač kai žemėlapyje yra daug įvairių kliūčių, išnagrinėtų mazgų skaičius priklauso nuo kelios paieškos krypties. Abiems atvejais išnagrinėtų mazgų atžvilgiu pranašesnis D*Lite algoritmas.

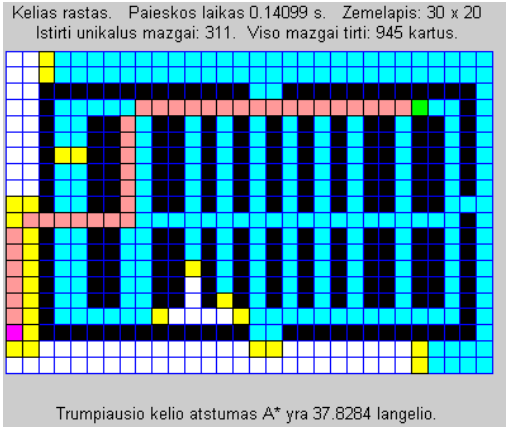
48 pav. surastas kelias A*algoritmu su 51 paveiksle D*Lite algoritmu surastu keliu, nes sutampa kelio paieškos kryptys, bei sutampa 49 pav. surastas kelias D*Lite algoritmu su 50 paveiksle A*Lite algoritmu surastu keliu, kadangi sutampa paieškos kryptys.



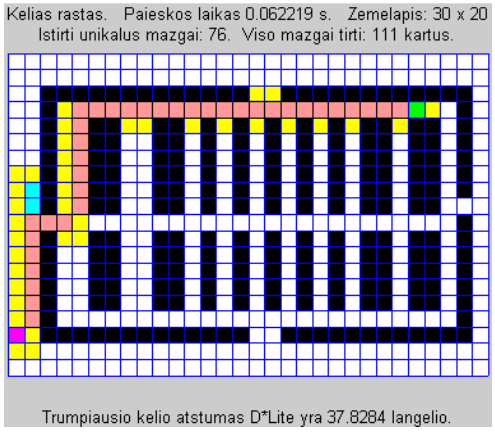
48 pav. Kelias patalpoje A* algoritmu



49 pav. Kelias patalpoje D*Lite algoritmu

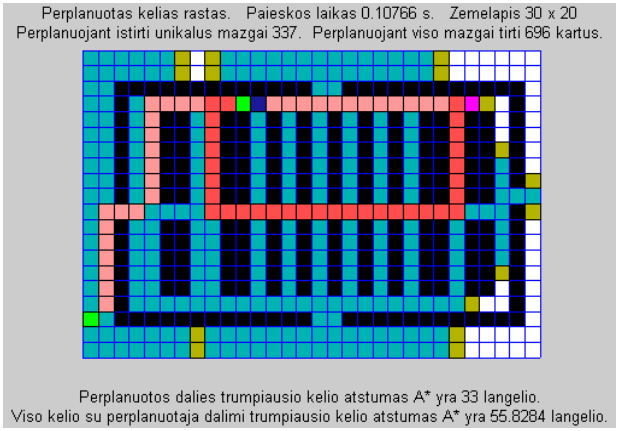


50 pav. Kelias patalpoje A* algoritmu

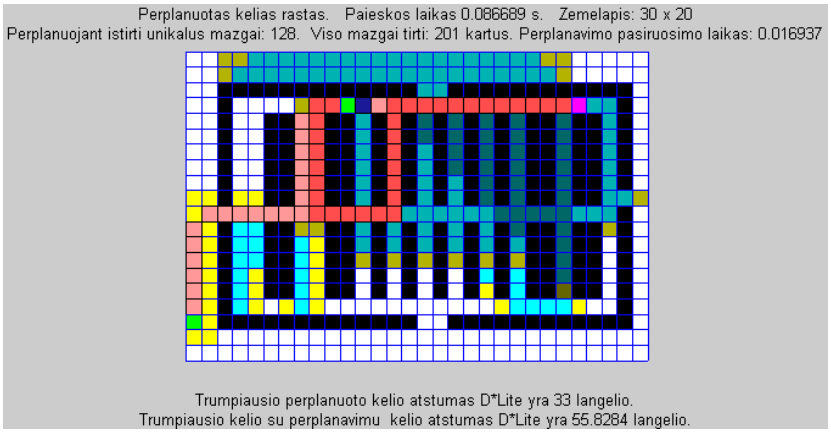


51 pav. Kelias patalpoje D*Lite algoritmu

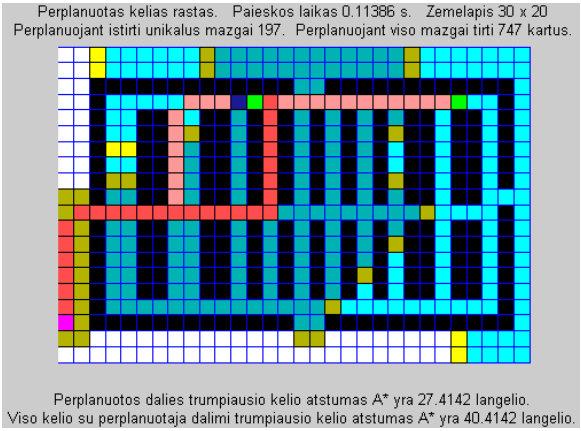
48, 49, 50 ir 51 paveiksluose pavaizduotiems keliams atlikime kelio perplanavimą, kliūtį padėdami tame pačiame langelyje. Perplanavimo rezultatai pateikti 52, 53, 54 ir 55 paveiksluose. Matome, kad nors abiem algoritmais perplanuotas kelias ilgis yra vienodas, o 54 ir 55 paveiksluose jis dar ir sutampa, visais atvejais pranešesnis D*Lite algoritmas.



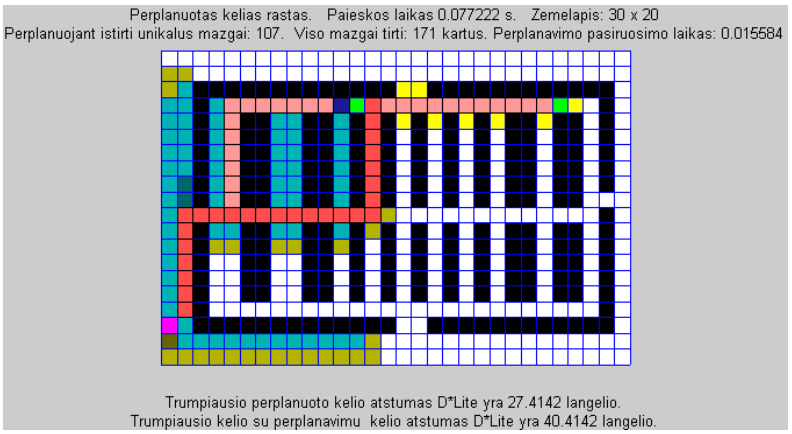
52 pav. Kelio perplanavimas patalpoje A* algoritmu



53 pav. Kelio perplanavimas patalpoje D*Lite algoritmu

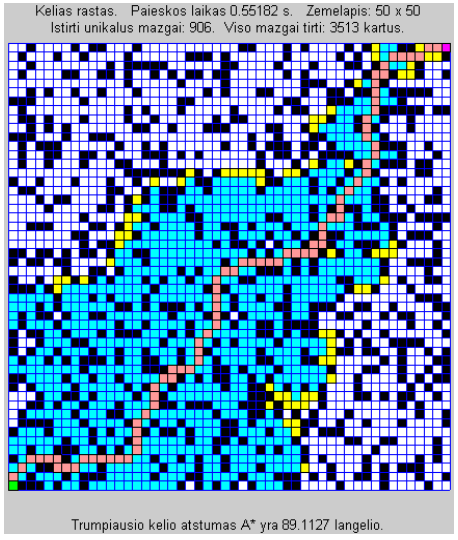


54 pav. Kelio perplanavimas patalpoje A* algoritmu priešinga kryptimi

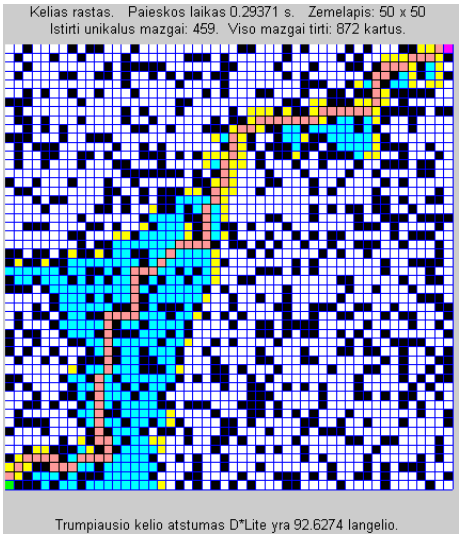


55 pav. Kelio perplanavimas patalpoje D*Lite algoritmu priešinga kryptimi

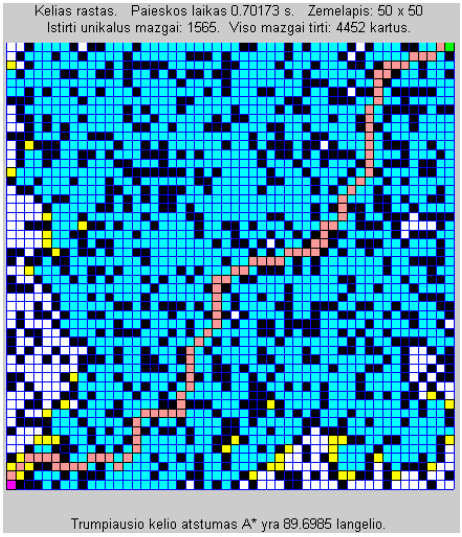
Pabaigai išnagrinėsime kelio paiešką žemėlapiuose, kuriuose kliūtys išdėstytos nesimetriškai, jos sąlyginai mažos, jų pakankamai daug, žemėlapyje nėra ilgų tiesių koridorių. Rezultatai parodyti 56, 57, 58, 59 paveiksluose. Žemėlapio dydis 50x50, 36% langelių yra kliūtys. Kelio paiešką atliksime pradinį ir galutinį mazgus pažymėję priešinguose žemėlapio kraštuose, paiešką atliksime į abi puses. Kelio perplanavimui kliūtį įterpsime kiekvieno surasto kelio viduryje. Perplanavimo rezultatai parodyti 60, 61, 62 ir 63 paveiksluose.



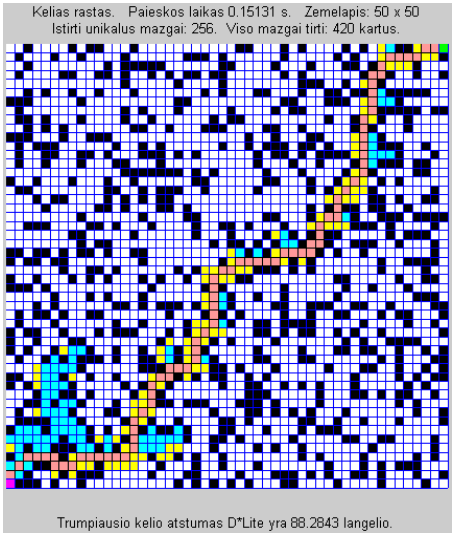
56 pav. Surastas kelias A* algoritmu



57 pav. Surastas kelias D*Lite algoritmu

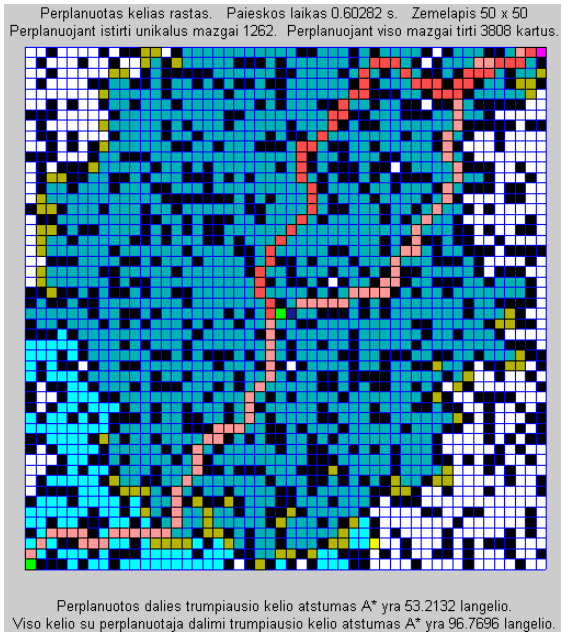


58 pav. Surastas kelias A* algoritmu priešinga kryptimi

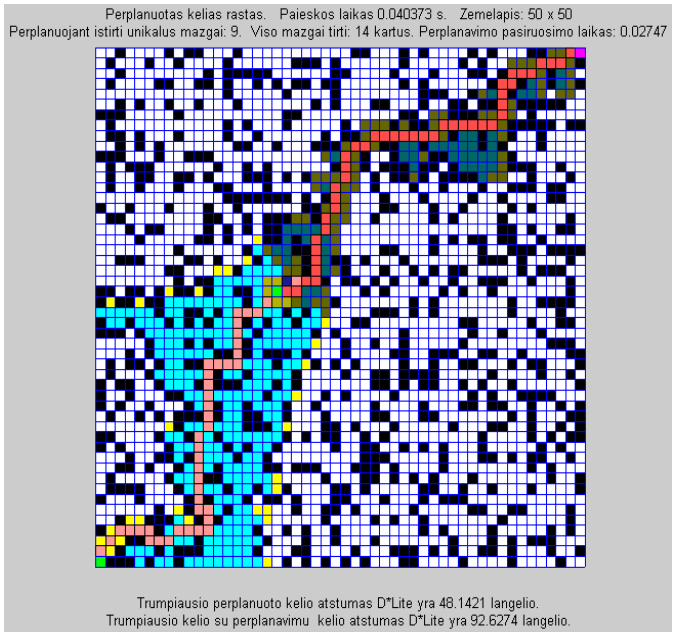


59 pav. Surastas kelias D*Lite algoritmu priešinga kryptimi

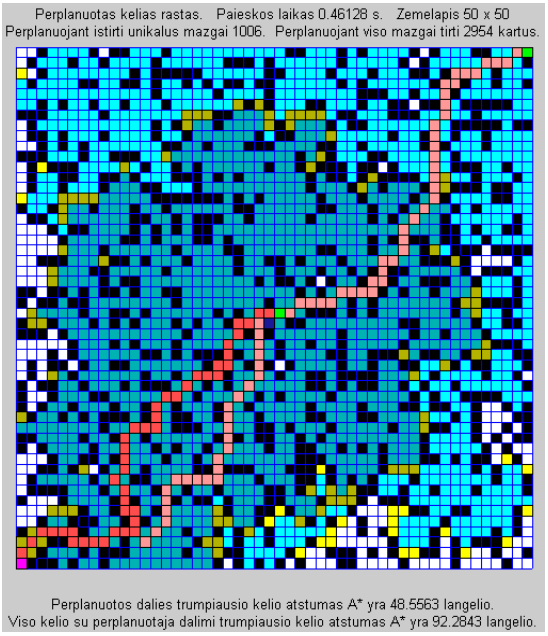
Perplanuoti keliai atsitiktiniame žemėlapyje



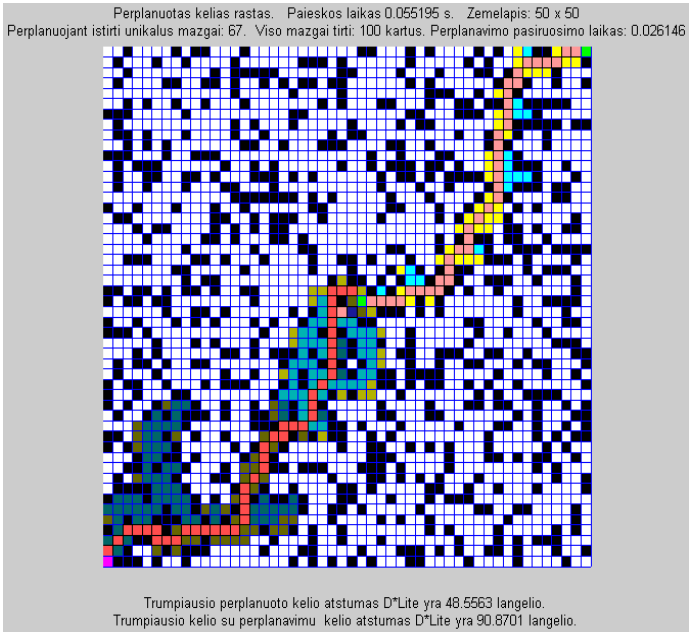
60 pav. Perplanuotas kelias A* algoritmu



61 pav. Perplanuotas kelias D*Lite algoritmu



62 pav. Perplanuotas kelias A* algoritmu priešinga kryptimi



63 pav. Perplanuotas kelias D*Lite algoritmu priešinga kryptimi

Atsižvelgiant į 40 – 63 paveiksluose bei 2.1 – 2.6 lentelėse pateiktus rezultatus, bandymams su realiu robotu pasirenkame D*Lite algoritmą.

2.3. Tyrimui naudojama techninė įranga

Praktiniam algoritmo pritaikymui, kelio parinkimo įvertinimui atliksime tyrimus su mobiliu autonominiu robotu E-puck (64 pav) [29].

64 pav. Mobilus autonominis robotas E-puck.

Trumpai apibūdinsime roboto navigaciją bei aptikimą tiriamame darbe, kitą naudojamą techninę įrangą. Roboto viršuje priklijuosime baltą skritulį su dviem raudonos spalvos skrituliais, roboto buvimo vietai ir judėjimo krypčiai nustatyti (65 pav). Robotas bus stebimas su virš jo įtaisyta skaitmenine kamera. Raudonųjų skritulių centrus rasime pasinaudodami Matlab

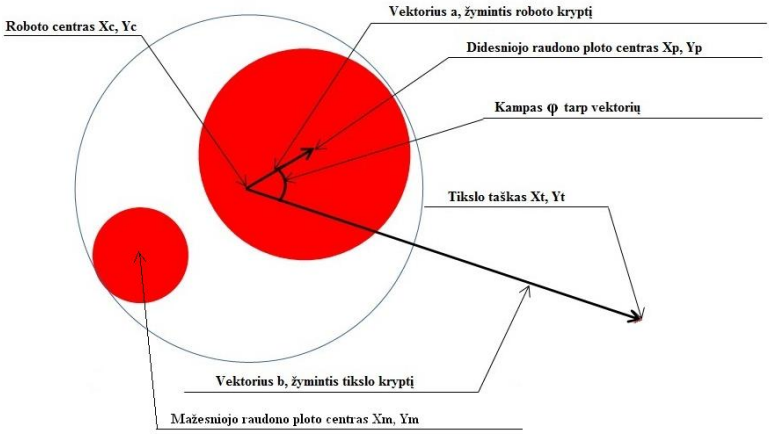
programų paketo vaizdų apdorojimo priemonėmis. Detaliai raudonos spalvos plotų išskyrimo metodų neaptarinėsime. Pagal šių skritulių centrus nustatysime roboto centrą. Roboto centro koordinates rasime pagal šią formulę:

$Xc = Xm + (Xp – Xm) * 0.634$

$Yc = Ym + (Yp – Ym) * 0.634$

Čia koeficientas 0.634 nurodo, kokia atstumo tarp centrų dalimi roboto centras nutolęs nuo mažesniojo raudono skritulio centro, nustatytas eksperimentiškai.

Centro koordinatės padės nustatyti roboto vietą žemėlepyje.



65 pav. Roboto padėties ir krypties nustatymo metodas

Vaizdo kamera taip pas bus naudojama nustatyti kliūtims, esančioms roboto aplinkoje, pagal kurias bus sudaromas žemėlapis kelio paieškai. Vaizdas nuskaitomas 640x480 RGB rezoliucija. Kamera pakabinsime 1.70 m aukštyje – iš tokio aukščio nuskaityto vaizdo dar galima pakankamai tiksliai išskirti raudonus skritulius roboto centro suradimui.

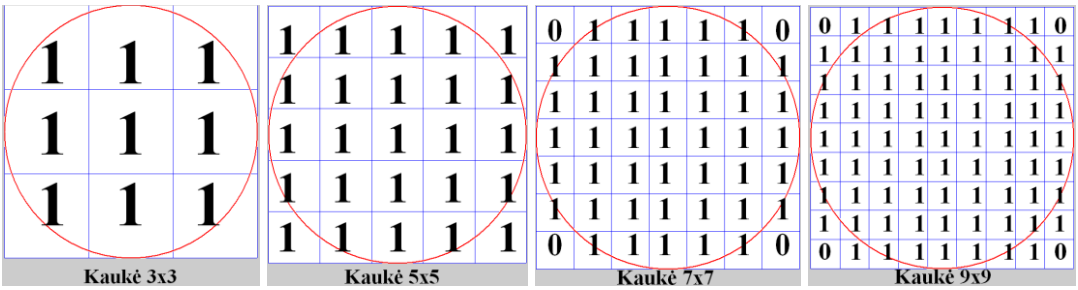
2.4. Kliūčių nustatymas, žemėlapio sudarymas bandymams su E-puck robotu

Aplinkos imatavimui tam tikrame plote sudėsime kliūtis robotui, tai bus įvairių dydžių juodo kartono figūros. Aplinkos pagrindo spalva turi būti kitokia nei kliūčių ir roboto žymeklio, kad būtų gali išskirti esančias kliūtis. Nuskaitę spalvotą vaizdą, ji konvertuosime į juodai baltą vaizdą tokiu principu, kad juoda spalva (reikšmė 0) reikš kliūtį, balta spalva (reikšmė 1) reikš pravažiuojamą langelį. Tuomet ant šio vaizdo „uždėsime“ norimo dydžio tinklelį, taip sukurdami roboto aplinkos žemėlapį. Jei į „tinklelio“ langelį pateks bent dalis kliūties, tokį langelį žymimime kaip kliūtį. Paprastumo dėlei, žinodami, kad kliūtys nėra sudarytos iš taškų (pikselių), pirmiausiai iš juodai balto vaizdo pašalinsime atskirus mažesnius nei 20 taškelių plotus.

Kalbant apie žemėlapio sudarymą realiose situacijose, juodai baltas vaizdas gali būti paruošiamas nebūtinai nuskaitant vaizdą su kamera, galima tai padaryti žinant patalpoje esančių objektų, sienų išsidėstymą, tuo pačiu principu kliūtis pažymint 0-iais, o pravažiuojamus langelius 1-ais.

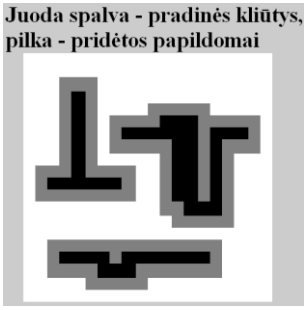
Palyginsime skirtumą tarp tinklelio, skirto žemėlapio sudarymui, dydžio. Paprasčiausia būtų žemėlapį padaryti tokį, kad visas robotas tilptų į vieną langelį. Tokiame žemėlapyje būtų žinoma, kad robotas , bent jau teoriškai, negali atsitrenkti į kliūtį, kadangi įstrižas važiavimas iš vieno langelio į kitą kertant kampą neleidžiamas, o kitu atveju (važiuojant aukštyn/žemyn arba kairėn/dešinėn), iš vieno langelio į kitą patenkama nekliudant jokios kliūties. Toks žemėlapis bus nedidelis (mažas eilučių, stulpelių aukštis), o tai reiškia, kad kelio paieškai, perplanavimui bus gaištama mažiau laiko. Tačiau taip sudarant žemėlapį, didelė jo pravažiuojama dalis bus pažymėta kaip kliūtys, tikėtina, kad bus uždaryti visi galimi pravažiavimai.

Siekiant išvengti nereikalingų kliūčių žymėjimo žemėlapyje, žemėlapio sudarymui reikia naudoti tankesnį tinklelį - su didesniu eilučių ir stulpelių skaičiumi. Juos reikia parinkti taip, kad robotas tilptų į nelyginį skaičių langelių, t.y., pagal tinklelio dydį roboto dydis būtų 3x3, 5x5, 7x7, 9x9 ir t.t. Nelyginis skaičius reikalingas tam, kad centrinis langelis būtų roboto centras. Sudarant tankesnio tinklelio žemėlapį, pažymėjus kliūtis iš paruošto juodai balto vaizdo, aplink kliūtis priklausomai nuo roboto dydžio, pravažiuojamą plotą reikia pažymėti kaip papildomas kliūtis – esamas kliūtis „apauginti“. Visa tai tam, kad važiuodamas surastu keliu robotas neatsitrenktų į realias kliūtis. Iš esmės, jei roboto dydis 3x3 langelio, nuo centrinio langelio likusieji nutolę per vieną langelį, tai ir aplink kiekvieną kliūtį visi langeliai 1 langelio atstumu pažymimi kaip kliūtys. Jei roboto dydis 5x5, atmetus centrinį langelį, likę langeliai nuo centrinio nutolę per 2 langelius, vadinasi, ir aplink kiekvieną kliūtį per 2 langelius esantys langeliai pažymimi kliūtimis. Jei roboto dydis 7x7, tuomet kliūtys apauginamos per 3 langelius, jei 9x9, per 4 langelius, ir t.t. Faktiškai šis kliūčių uždėjimas – tai juodai balto vaizdo vienos spalvos zonų padidinimas (angl „dilation“), naudojant skritulio (disko) pavidalo kaukę, kurioje papuolantys į skritulį taškai turi svorį 1, nepapuolantys – 0. Kaukių pavyzdžiai (66 pav):



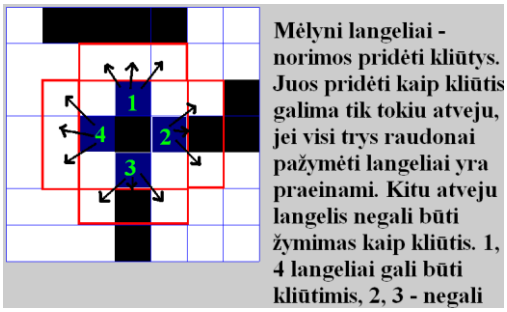
66 pav. Kaukių papildomoms kliūtims sudaryti pavyzdžiai.

Žemiau, 67 paveiksle, pateiktas pavyzdys, kaip atrodytų žemėlapis, apdorotas 3x3 dydžio kauke:



67 pav. Papildomų kliūčių pažymėjimas žemėlapyje naudojant 3x3 dydžio kaukę.

Kadangi šio darbo tikslas – parinkti roboto važiavimo trajektoriją taip, kad būtų išvengta susidūrimo su kliūtimis, o važiavimo greitis ir trajektorijos trumpumas nėra patys svarbiausi, nors ir svarbūs, kriterijai, jau apdorotame – su papildomos kliūtimis – žemėlapyje per 1 langelį aukštyn/žemyn ir kairėn/dešinėn nuo kliūčių esančius langelius irgi pažymėsime kliūtimis. Tokiu atveju planuojama roboto trajektorija bus patraukta toliau nuo sienų ir kliūčių. Istrižai papildomos kliūtys nežymimos, nes tokiu atveju atveju galima uždaryti esančius realius pravažiavimus. Taip pat dėl šių papildomų kliūčių pridėjimo reikia įvertinti, kad jos neuždarytų esamų pravažiavimų, jei tas pravažiavimas yra 1-o langelio pločio. Papildomos kliūties pridėjimui turi būti tenkinama žemiau pavaizduota sąlyga (68 pav):

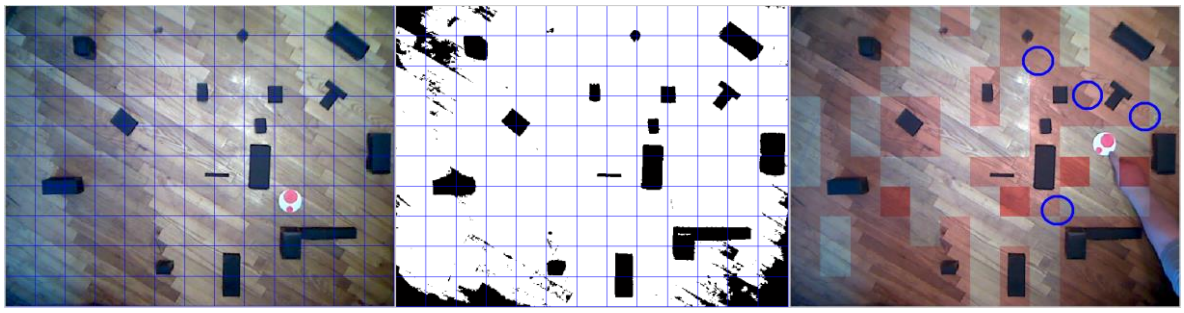


68 pav. Papildomų kliūčių norint patraukti planuojamą trajektoriją nuo sienos pridėjimo taisyklė

Pridedant kliūtis, tikrinant kiekvieną kliūties langelį, papildomai reikia įvertinti visas jau prieš pridėtas kliūtis, kitu atveju taip pat galima uždaryti galimą pravažiavimą. Tinkamas ir netinkamas šių kliūčių pridėjimas pavaizduotas žemiau, 69 pav, kairėje nevertinamos jau pridėtos kliūtys, dėšinėje - vertinamos. Pastaba – prie žemėlapio kraštuose esančių kliūčių papildomos kliūtys nepridedamos tuo atveju, jei tos irgi taptų kraštinėmis, kadangi jei ten nėra pravažiavimo, kelias šiais langeliais nebus suplanuotas.

69 pav. Teisingai (dešinėje) ir neteisingai (kairėje) pridėtos papildomos kliūtys

Toliau pateikiami realūs pavyzdžiai, kaip atrodo sudaryti žemėlapiai, vaizdą fiksuojant kameros pagalba, bei parenkant įvairaus dydžio tinklelius. Kliūčių išdėstymas visiems variantams vienodas.

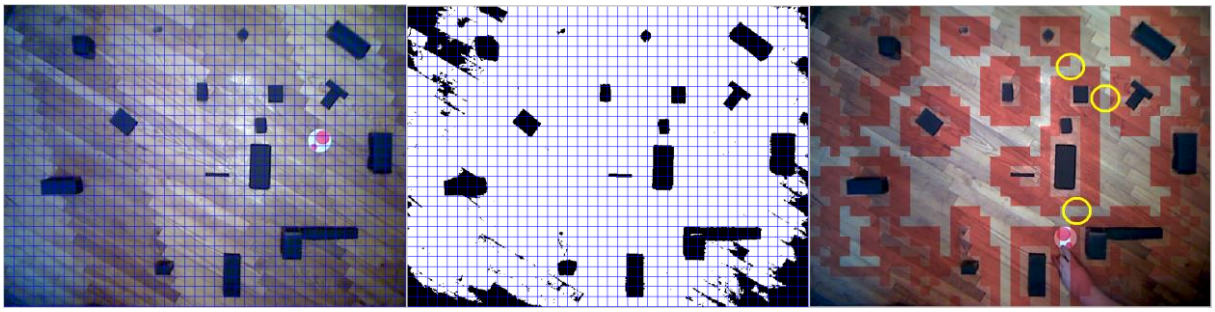


70 pav. a) pradinis vaizdas b) juodai baltas vaizdas c) pažymėtos kliūtys

70 pav. **a** pateiktas žemėlapis su kliūtimis, tinklelio dydis 10x13, roboto dydis 1 langelis. 70 **b** dalyje pateiktas juodai baltas vaizdas, pagal kurį pažymimos kliūtys žemėlapyje. **c** dalyje parodyta, kaip atrodo kliūtys žemėlapyje. Šviesiai rausva spalva pažymėtos kliūtys, sudarytos iš juodai balto žemėlapio, tamsiai rausva – pridėtos papildomos kliūtys. Mėlynais apskritimais pažymėtos zonos, kuriose vizualiai matyti, kad jose telpa visas robotas, tačiau dėl nedidelio tinklelio, net ir daliai kliūties papuolus į langelius, jie pažymimi kaip kliūtys. Taip pat matyti, kad jei į vieną langelį telpanti kliūtis papuola į tinklelio linijų susikirtimo tašką, kliūtimi bus pažymėtas nebe vienas, o mažiausiai 4 langeliai. Šiuo atveju toks žemėlapio sudarymas nėra tinkamas, kadangi robotas yra uždarytas.

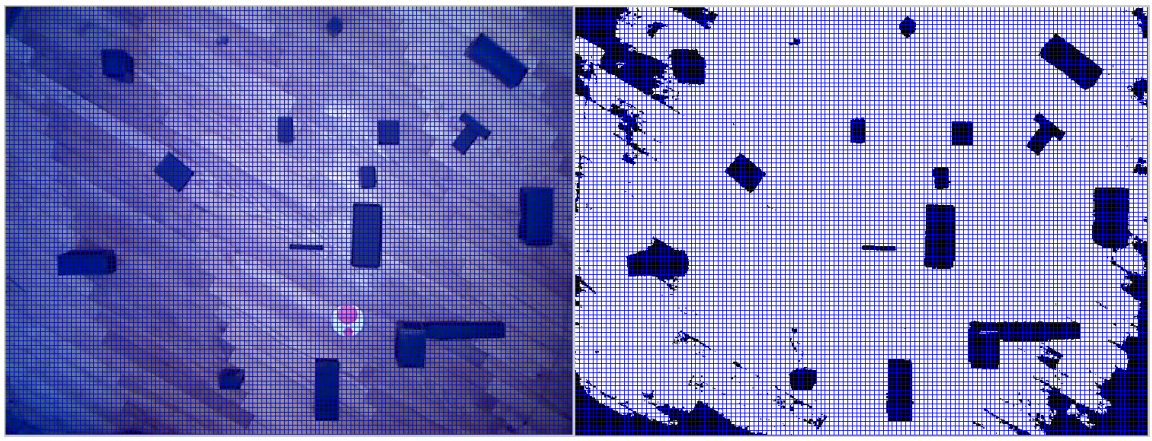
Toliau 71 pav. pateiktas žamėlapio sudarymas, kai kliūčių išsidėstymas toks pat, tik naudojamas 30x40 dydžio tinklelis, o roboto dydis yra 3x3 langelių. **c** dalyje matome, kad prie pradinių kliūčių yra mažiausiai 1 langelio atstumu yra pridėta papildomų kliūčių – įvertintos zonos, į kurias neturi papulti roboto centras, taip pat kur įmanoma papildomai pridėta kliūčių, siekiant patraukti galimos trajektorijos kelią toliau nuo kliūčių ir/ar sienų. Geltonais apskritimais pažymėtos zonos, kuriose, lygint su 70 pav. pavaizduotomis kliūtis, jau atsiranda 1 langelio

pločio pravažiuojamas tarpas, robotas nebėra uždarytas. Akivaizdu, kad šis žemėlapis teisingesnis nei sudarytas prieš tai.



71 pav. a) pradinis vaizdas b) juodai baltas vaizdas c) pažymėtos kliūtys

Toliau 72 pav. pateiktas žemėlapio sudarymas, kai roboto dydis 9x9 langelių, o tinklelio dydis 96x128 langelių. Matome, kad sudarytame juodai baltame vaizde į langelius papuolančios kliūties dalys užpildo didžiąją dalį arba visą langelį. 73 paveiksle parodyta, kaip atrodys kliūčių žemėlapis kartu su papildomai pridėtomis kliūtimis. Žaliais apskritimais pažymėtos tos zonos, kurios pirmame žemėlapyje , kuris sudarytas naudojant 10x13 dydžio tinklelį, buvo nepraeinamos. Matome, kad šios zonos smulkiame žemėlapyje tampa pravažiuojamomis.



72 pav. a) pradinis vaizdas b) juodai baltas vaizdas

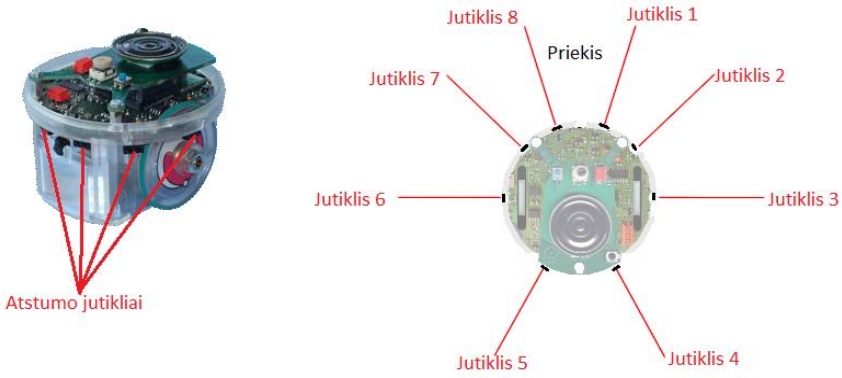
73 pav. 96x128 tinkleliu sudarytas kliūčių žemėlapis.

Išnagrinėję tris pateiktus patalpos žemėlapio sudarymo variantus, galime teigti, kad kuo smulkesnis žemėlapis, tuo tiksliau pažymimos kliūtys, mažiau laisvo ploto pažymima kaip kliūtys. Tačiau tuo pačiu pailgėja ir kelio paieškos laikas, kadangi reikia ištirti žymiai daugiau mazgų. Todėl kiekvienu konkrečiu uždaviniu reikia pasirinkti/nustatyti tokį zemėlapio dydį, kad būtų pakankamai tiksliai atskirtos kliūtys nuo pravažiuojamų plotų, ir kelio paieška būtų pakankamai greita.

Pagal vieno iš kelio paieškos algoritmų surastą kelią sudarysime roboto judėjimo maršrutą su nurodymais kiek laipsnių ir kuria kryptimi - prieš ar pagal laikrodžio rodyklę – robotui pasisukti, norint judėti iš pasiekto mazgo į sekantį trajektorijos mazgą. Judėjimo metu naudojantis roboto IR (Infrared – infraraudonųjų spindulių jutikliais) tikrinsime ar mazge, į kurį robotas turi važiuoti, nėra kliūčių. Radus kliūtį, ji bus pažymima žemėlapyje, o kelias perskaičiuojamas. Kliūties radimui reikia įvertinti, kokio dydžio yra robotas (langeliais), kadangi aptiktos žemėlapyje nepažymėtos kliūtys galės būti pridedamos tik aplink centrinį roboto langelį.

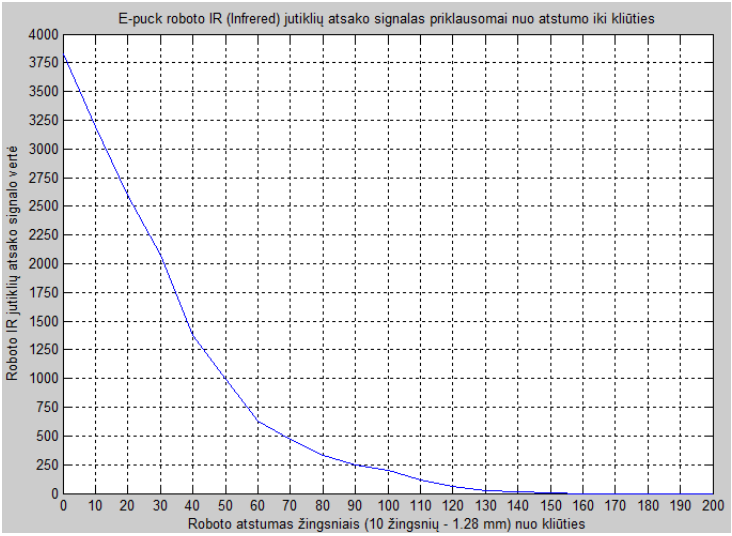
2.5. Bandymų su mobiliu robotu rezultatai

Atliksime eilę važiavimų su E-PUCK robotu, kiekvieno važiavimo metu suskaičiuosime, ar važiuodamas sumodeliuotoje aplinkoje robotas atsitrenkė į kliūtis, taip pat roboto važiavimo metu suplanuotame kelyje pastatysime kliūčių, kad būtų perplanuotas roboto važiavimo kelias. Kliūčių aptikimui pasinaudosime E-PUCK robote įtaisytais IR (*Infrared*) atstumo jutikliais, kurių jis turi 8-is, išdėstytus roboto perimetre, kaip parodyta 74 pav. [29] Jutikliais Nr.1, Nr.8 fiksuosime roboto judėjimo kelyje pasitaikančias kliūtis bei tikrinsime, ar robotas nesusidūrė su kliūtimi (siena), jutikliais Nr.3 ir Nr.6 tik fiksuosime, ar robotas važiuodamas suplanuota trajektorija šonais neprisilietė prie kliūčių (sienų).



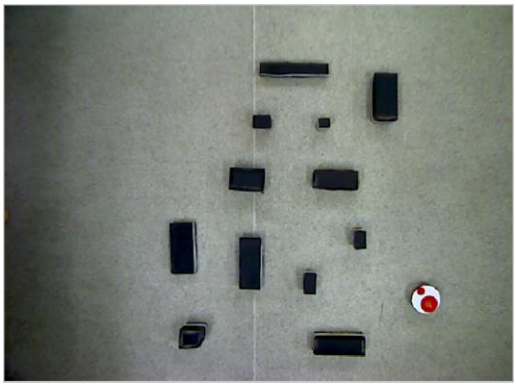
74 pav. kairėje : E-PUCK robotas, dešinėje – atstumo jutiklių išdėstymas.

Atstumo jutikliai kliūtį gali aptikti 4 cm atstumu. Įprastai, jei kliūtis yra toliau nei 4 cm nuo jutiklio, jo išėjimo vertė yra lygi 0. Kliūtį priartinus prie jutiklio, išėjimo vertė didėja kaip parodyta 75 pav. Ši priklausomybė gauta tokiu būdu: robotas pastatomas prie sienos, atsuktas į ją priekiu. Tada užfiksuojamos jutiklių Nr.1. ir Nr.8. vertės, robotas pajuda nuo sienos atgal per 10 žingsnių, vėl užfiksuojamo jutiklių vertės. Kartojama, kol robotas nuo sienos nutols 200 žingsnių. Roboto rato 1 pilnas apsisukimas yra 1000 žingsnių, tai yra 12,8 cm, 10 žingsnių = 1,28 mm. Laikysime, kad susidurta su kliūtimi, jei atstumas tarp jutiklio ir kliūties tampa mažesnis nei 1 mm, tam imsime, kad IR jutiklio signalo atsakas turi būti didesnis nei 3500. Realybėje šis priklausomybės grafikas gali būti ir kiek kitoks, kadangi jutiklio atsakas į atstumą iki kliūties priklauso nuo kliūties spalvos, nuo roboto aplinkos apšvietimo. Šie duomenys surinkti uždaroje patalpoje dienos šviesoje, tuo pačiu esant ir dirbtiniam patalpos apšvietimui.



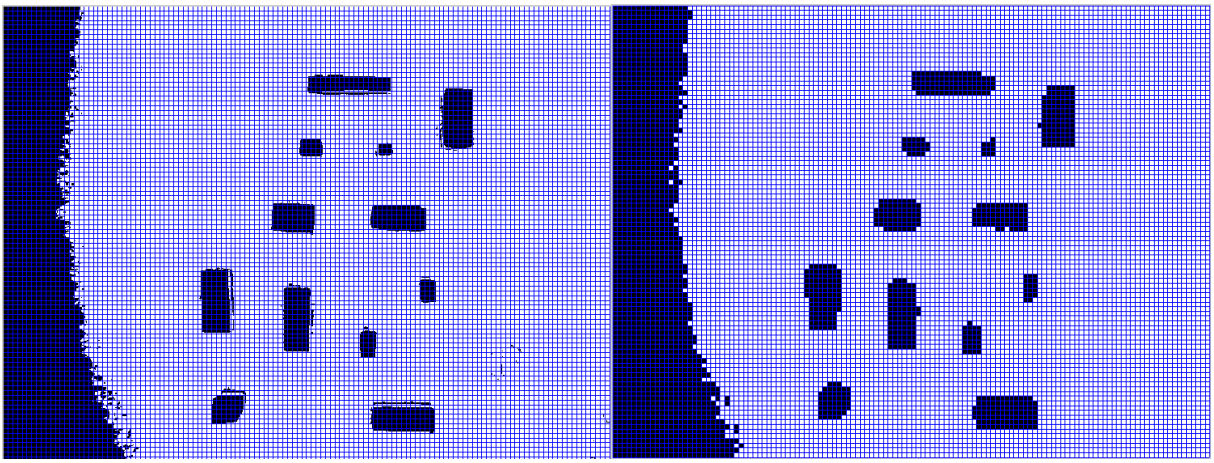
75 pav. Roboto E-Puck IR atstumo jutiklių atsako priklausomybė nuo atstumo iki kliūties

Atliksime eilę važiavimų roboto kelyje nesant kliūčių, kai žemėlapis sudaromas iš anksto, pagal esantį kliūčių išsidėstymą. Kliūtis išdėstysime taip, kad žemėlapis sudarant žemėlapį liktų 1 langelio pločio koridoriai, tuomet fiziškai tarpai tarp kliūčių bus kiek didesni nei roboto skersmuo. Kliūtys išdėstytos kaip parodyta 76 pav. (juodos spalvos stačiakampiai)



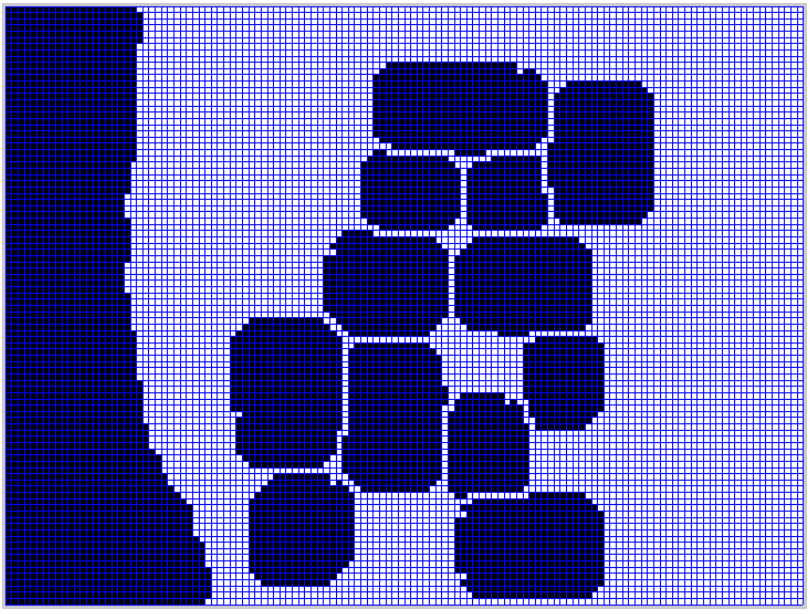
76 pav. Pradinis kliūčių išdėstymas, naudojamas tyrime

Žemiau 77 pav. pateiktas 96 x 128 dydžio žemėlapio sudarymas iš 76 pav. pateikto vaizdo.



77 pav. 96 x 128 dydžio žemėlapio sudarymas: kairėje – juodai baltas vaizdas su uždėtu tinkleliu, dešinėje – pagal tinklelį sudarytas žemėlapis.

96 x 128 dydžio žemėlapyje roboto dydis yra 9x9 langelių, tam aplink esančias pažymėtas kliūtis, kaip minėjome ankstesniuose skyriuose, pusės roboto dydžio atstumu (langeliais) esančias zonas taip pat pažymėsime kliūtimis, taip pat kur įmanoma 1-o langelio atstumu aplink jau papildomai pažymėtas kliūtis esančią zoną pažymėsime kliūtimis. Gausime 78 pav. pavaizduotą žemėlapį su kliūtimis. Žemėlapio kairėje pusėje esanti kliūčių zona atsirado dėl patalpos sienos šešėlio, tačiau kliūtys išdėstytos taip, kad ši zona tyrimui nekliudytų. Kaip matome, pravažiavimai yra tarp visų kliūčių, daugumos jų plotis 1-as langelis.



78 pav. 96 x 128 langelių dydžio žemėlapis su papildomai pažymėtomis kliūtimis.

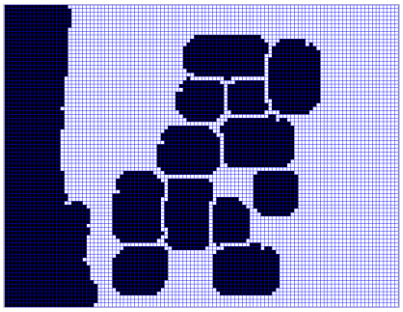
Važiavimus atliksime žemėlapį skaidydami į skirtingą eilučių ir stulpelių skaičių, atsižvelgiant į pasirinktą roboto dydį, kiekvienam žemėlapio dydžiui po 25 važiavimus. Kiekvienam važiavimui parinksime skirtingus tikslo taškus trajektorijų įvairovei (pradinis taškas visada bus roboto centras žemėlapio sudarymo metu), ir jutikliais Nr.1, Nr.3 Nr.6 bei Nr.8 fiksuosime, kiek kartu robotas susidūrė su viena ar kita siena. Bandymų rezultatai pateikti žemiau 2.7 lentelėje:

2.7 lentelė. Roboto susidūrimai su kliūtimis važiavimo surastomis trajektorijomis metu.

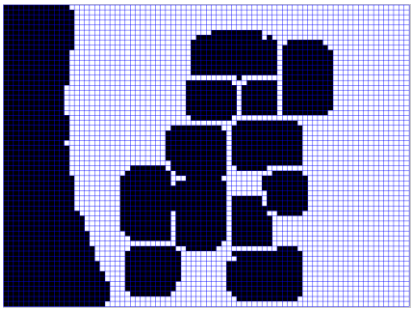
Žemėlapio dydis langeliais	Roboto dydis langeliais	Kiek atlikta važiavimų	Keliuose važiavimuose susidurta su kliūtimis	Kiek kartų iš viso susidurta su kliūtimis	Kiek % trajektorijų įveikta su susidūrimais
96 x 128	9 x 9	25	0	0	0
80 x 106	7 x 7	25	1	2	4%
60 x 80	5 x 5	25	2	2	16%
36 x 49	3 x 3	25	7	8	28%
13 x 17	1	25	12	25	48%

Kaip matyti, geriausi rezultatai gauti, kai roboto dydis 5x5 langelių arba didesnis. Susidūrimus su sienomis galėjo nulemti tai, jei sudarius kliūčių žemėlapį vienai ar kelioms kliūtims nebuvo galimybės papildomai pažymėti vieno langelio pločio zonos kaip kliūties. Kadangi uždėjome priartėjimo apribojimą iki 1 mm, dalyje pravažiavimų užfiksuotas susidūrimas su kliūtimis. Kai roboto dydis 3x3 arba tik 1 langelis, žemėlapio sudarymo metu pasitaikė atvejų, kai žemėlapio langelius skirianti linija praktiškai sutampa su kliūties kraštu. Esant mažam eilučių ir stulpelių skaičiui, taip pat ne apie visas kliūtis 1 langelio pločiu esančią zoną buvo galima pažymėti papildomomis kliūtimis, dėl to kai kur atstumai 1-o langelio pločio koridoriuose nuo pravažiuojamo langelio vidurio tapo lygūs roboto spinduliui (kai roboto dydis 3x3 langelio), arba langelio dydis tapo lygus roboto skersmeniui (kai roboto dydis 1-as langelis). Šiais atvejais taip pat dalyje važiavimų buvo užfiksuota susidūrimų su kliūtimis (arba priartėjimų prie kliūties arčiau nei per 1 mm). Taip pat susidūrimus galėjo lemti faktas, kad jei robotas yra ne kamera užfiksuoto vaizdo centre, raudoni žymeklio skrituliai nežymiai pasislenka, dėl to per kelis vaizdus taškus „pasislenka“ ir roboto centras.

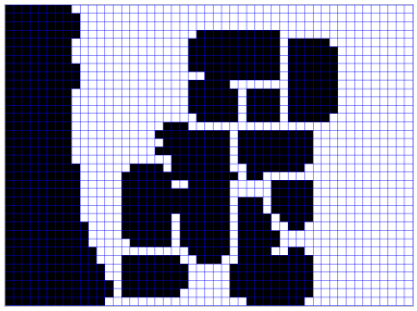
79, 80, 81 ir 82 paveiksluose žemėlapių sudarymas esant skirtingam eilučių ir stulpelių skaičiui, kai kliūčių išsidėstymas nekinta. Matome, kad 80 x 106 langelių dydžio žemėlapyje pravažiavimai dar išlieka tarp visų kliūčių (79 pav), žemėlapio dydžiui mažėjant kai kurie pravažiavimai traktuojami kaip kliūtys. Esant mažiausiam žemėlapio dydžiui, kai roboto dydis tik 1-as langelis, lieka tik 2 galimi keliai iš vienos žemėlapio pusės į kitą. Įvertinus 2.7 lentelėje esančius rezultatus, 2.4 skyriaus pradžioje iškelta teorija, kad jei langelis bus roboto dydžio, jos į kliūtis neatsitrenks, nepasitvirtino, priešingai, rezultatai tik patvirtina, kad žemėlapį reikia sudaryti kuo tankesnį, norint išvengti susidūrimų su kliūtimis.



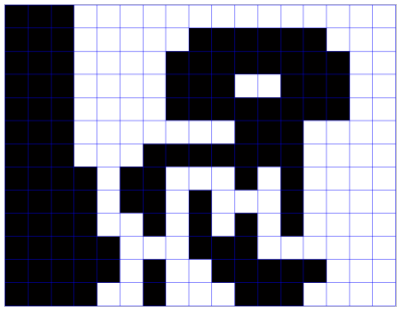
79 pav. 80 x 106 langelių dydžio žemėlapis su papildomai pažymėtomis kliūtimis.



80 pav. 60 x 80 langelių dydžio žemėlapis su papildomai pažymėtomis kliūtimis.

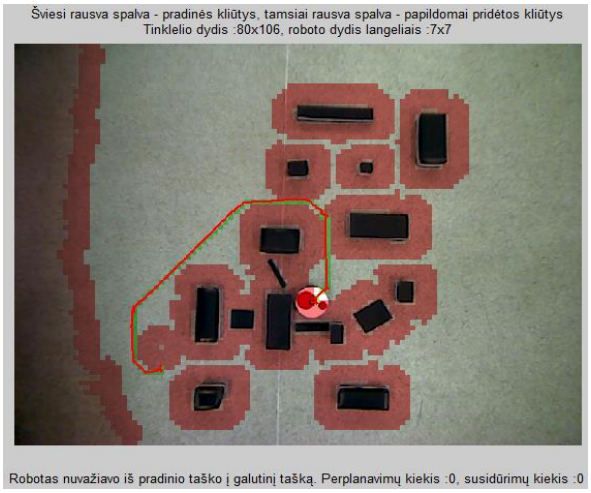


81 pav. 36 x 49 langelių dydžio žemėlapis su papildomai pažymėtomis kliūtimis.

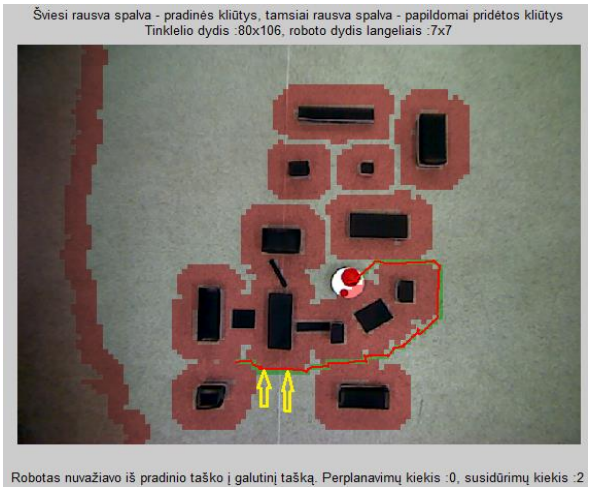


82 pav. 13 x 17 langelių dydžio žemėlapis su papildomai pažymėtomis kliūtimis.

Sėkmingo roboto pravažiavimo trajektorija parodytas 83 pav., pravažiavimas, kurio metu susidurta su kliūtimis – 84 pav. Paveikslėliuose žalia spalva pažymėtas surastas kelias, ryškiai raudona linija – roboto nuvažiuotas kelias, užfiksuotas pagal roboto centrą. Matyti, kad tiek vienu tiek kitu važiavimu robotas važiavimo trajektorija yra visai šalia kliūties, tačiau 84 pav. geltonomis rodyklėmis pažymėtose zonose ši trajektorija papuola į kliūčių (pažymėta rausva spalva) zoną, dėl to ir užfiksuoti susidūrimai su sienomis.

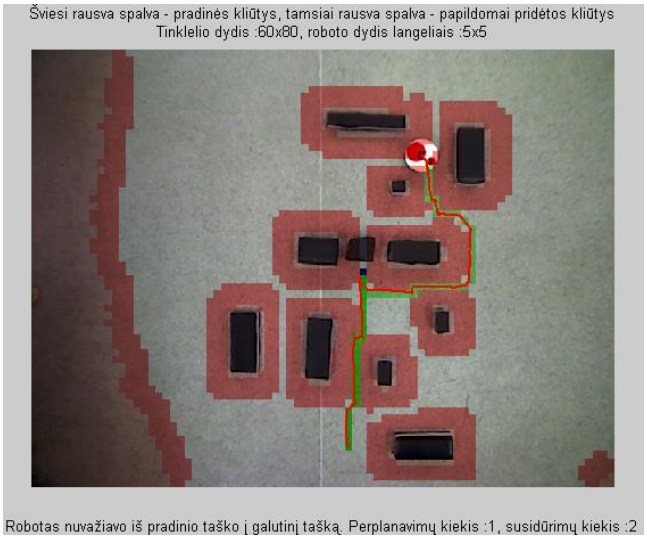


83 pav. Sėkmingas pravažiavimas trajektorija



84 pav. Pravažiavimas trajektorija susiduriant su kliūtimis

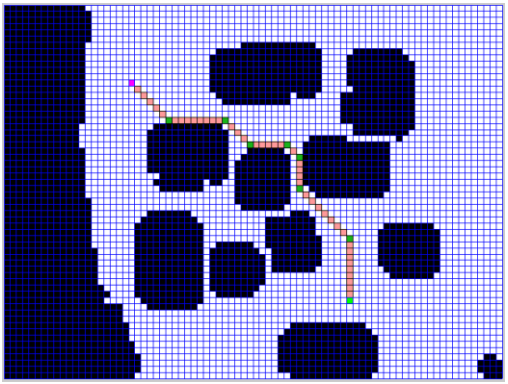
Kelio perplanavimui atliksime bandomąjį važiavimą, įsitikimui, kad D* Lite algoritmas veikia ir kelias yra perplanuojamas. Tam jutikliais Nr.1. ir Nr.8 tikrinsime, ar roboto priekyje neatsidaro iš anksto nenumatyta kliūtis. Laikysime, kad roboto priekyje yra kliūtis, jei šių jutiklių išėjimo signalo vertė bus didesnė už 2500, tai maždaug atitinka 20 roboto žingsnių (~ 2.56 mm). Nustatę kliūtį, ją pažymėsime žemėlapyje, atliksime kelio perplanavimą. Tokio važiavimo rezultatai pateikti 85 pav. Tamsiai žalia spalva pažymėtas kelias iki kliūties, mėlynas langelis – nustatyta kliūtis, šviesiai žalia spalva – perplanuotas kelias.



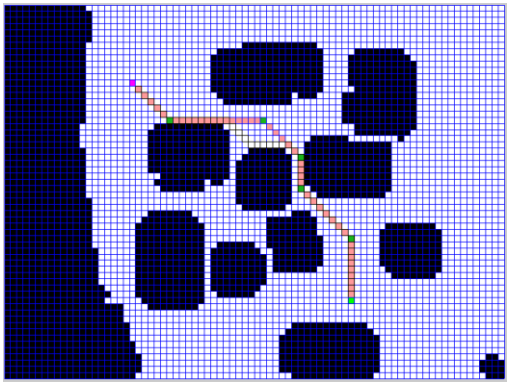
85 pav. Robot važiavimas trajektorija su kelio perplanavimu

Apibendrinant tiek A*, tiek D*Lite, tiek kitais algoritmais rastus trumpiausius kelius tinklelio tipo žemėlapiuose, galima pastebėti, kad dažnu atveju, jei kelias eina platesniu nei 1 langelio keliu, kelio trajektoriją galima būtų kiek modifikuoti, išlaikant tą patį kelią sudarančių langelių skaičių, tačiau mažinant posūkių skaičių tuo pačiu tiesinant kelio linijas. 86 pav. parodytame surastame kelyje yra 7 posūkiai (žali langeliai), o modifikuotame kelyje (87 pav) – tik 5 posūkiai. Tokiam trajektorijos optimizavimui galima sukurti atskirą algoritmą.

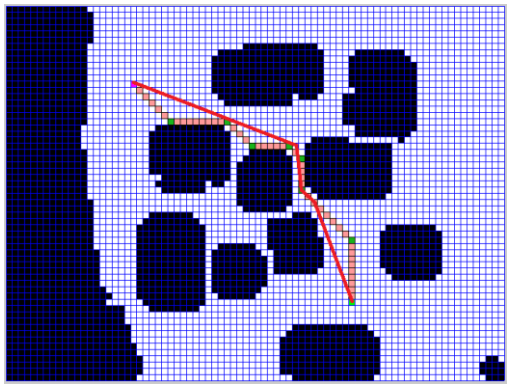
Taip pat galima pastebėti, kad mūsų tirtais algoritmais rastas trumpiausias kelias „fiziškai“ nėra pats trumpiausias kelias – tai yra trumpiausias kelias pagal esamą žemėlapį. Trumpiausias kelias būtų, jei robotas važiuotų ne iš vieno langelio į kitą, o kaip parodytą 88 pav, kiek įmanoma tiesesnėmis linijomis (ryški raudona linija paveiksle). Tačiau tam reikėtų naudoti dar kitus algoritmus, kurie surastą trumpiausią kelią ištiesintų bei parinktų atitinkamus kelio posūkius. Tai jau būtų atskiri trajektorijų optimizavimo uždaviniai.



86 pav. Pradinis surastas kelias su 7 posūkiais



87 pav. Modifikuotas to paties ilgio kelias su 5 posūkiais.



88 pav. Trumpiausias kelias tiesiomis linijomis

3. REZULTATAI IR IŠVADOS

Įvertinus tyrimo rezultatus, galima teigti, kad:

1. Trumpiausio kelio paieškai žemėlapyje (ar grafe) tinkamiausi yra euristinius, geriausios pirmos paieškos, progresuojančios paieškos metodus naudojantys algoritmai.
2. D*Lite algoritmas kelio perplanavimui tinkamesnis už tradicinį A* algoritmą dėl dalies jau turimos ankstesnio kelio paieškos informacijos panaudojimo ir mažesnio perskaičiuojamų mazgų skaičiaus, tačiau A* algoritmo paprastesnis realizavimas.
3. A* algoritmą galima tobulinti, parenkant atvirojo sąrašo sudarymo schemą ir mazgo išmetimo iš šio sąrašo schemą, kad jame mazgų skaičius būtų kuo mažesnis.
4. Sudarant roboto aplinkos žemėlapį, patartina parinkti tokį mastelį, kad bent per vieną žemėlapio langelį aplink kliūtis esančią zoną būtų galima papildomai pažymėti kliūtimis, siekiant išvengti susidūrimo su tikromis kliūtimis, kas gali būti itin aktualu pavojingose aplinkose.
5. Visais kelio paieškos algoritmais rastą trumpiausią kelią galima būtų optimizuoti taip, kad kelio linijos būtų kuo tiesesnės, kelyje būtų kuo mažiau posūkių.

4. LITERATŪRA

[1] Miguel A. Padilla Castaneda, Jesus Savage, Adalberto Hernandez and Fernando Arambula Cosío (2008). Local Autonomous Robot Navigation Using Potential Fields, Motion Planning, Xing-Jian Jing (Ed.), ISBN:978-953-7619-01-5, InTech.

[2] Y. Koren, J. Borenstein (1991) Potential field methods and their inherent limitations for mobile robot navigation. in: IEEE International Conference on Robotics and Automation, vol.2, Sacramento, CA, pp. 1398-1404

[3] Johan Hagelbäck (2009) A Multi-Agent Potential Field Based Approach for Real-Time Strategy Game Bots. PhD thesis, Printfabriken, Karlskrona, Sweden, 81p.

[4] Derek Partridge (1991) A New Guide to Artificial Intelligence, 546p.

[5] Stuart Russell, Peter Norvig (2009) Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA

[6] Steven M. LaValle, (2006) Planning algorithms. Cambridge University Press, 842 p.

[7] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.

[8] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics SSC4 **4** (2): 100–107. doi:10.1109/TSSC.1968.300136.

[9] Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. ISBN 0-201-05594-5

[10] Stentz, Anthony (1994), "Optimal and Efficient Path Planning for Partially-Known Environments", Proceedings of the International Conference on Robotics and Automation: 3310–3317.

[11] Stentz, Anthony (1995), "The Focussed D* Algorithm for Real-Time Replanning", In Proceedings of the International Joint Conference on Artificial Intelligence: 1652–1659.

[12] S. Koenig and M. Likhachev. Fast Replanning for Navigation in Unknown Terrain. Transactions on Robotics, 21, (3), 354–363, 2005.

[13] Korf, Richard (1985). "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search". Artificial Intelligence 27: 97–109.

[14] Russell, S. (1992). "Efficient memory-bounded search methods". In Neumann, B. Proceedings of the 10th European Conference on Artificial intelligence. Vienna, Austria: John Wiley & Sons, New York, NY. pp. 1–5. CiteSeerX: 10.1.1.105.7839

[15] Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR 0102435

[16] Moore, Edward F. (1959). "The shortest path through a maze". Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285–292. MR 0114710

[17] Adem Tuncer , Mehmet Yildirim (2012) „Dynamic path planning of mobile robots with improved genetic algorithm“. Computers and Electrical Engineering 38 (2012) 1564–1572

[18] Fatemeh Khosravi Purian, Fardad Farokhi, Reza Sabbaghi Nadooshan „Comparing the Performance of Genetic Algorithm and Ant Colony Optimization Algorithm for Mobile Robot Path Planning in the Dynamic Environments with Different Complexities“, Journal of Academic and Applied Studies Vol. 3(2) February 2013, pp. 29-44, ISSN1925-931X

[19] Velappa Ganapathy, Titus Tang Jia Jie, S. Parasuraman, „IMPROVED ANT COLONY OPTIMIZATION FOR ROBOT NAVIGATION“, Proceeding of the 7th International Symposium on Mechatronics and its Applications (ISMA10), (2010) Sharjah, UAE, April 20-22,

[20] Maram Alajlan, Anis Koub, Imen Chri, Hachemi Bennaceur, Adel Ammar, „Global Path Planning for Mobile Robots in Large-Scale Grid Environments using Genetic Algorithms“. International Conference on Individual and Collective Behaviors in Robotics, (2013).

[21] http://pub1.willowgarage.com/~konolige/cs225b/dlite_tro05.pdf

Žiūrėta 2014.03.15

[22] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5679403

Žiūrėta 2014.03.17

[23] http://www.sciencedirect.com/science/article/pii/S000437020300225X#

Žiūrėta 2014.03.19

[24] http://www.sciencedirect.com/science/article/pii/S0196677496900462#

Žiūrėta 2014.03.19

[25] http://dasl.mem.drexel.edu/Hing/BFSDFSTutorial.htm

Žiūrėta 2014.04.13

[26] http://en.wikipedia.org/wiki/Best-first_search

Žiūrėta 2014.03.15

[27] http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Žiūrėta 2014.04.10

[28] http://www.sciencedirect.com/science/article/pii/S0196677496900462#

Žiūrėta 2014.04.11

[29] www.e-puck.org

Žiūrėta 2014.05.25