Rīgas 64. vidusskola

**Mākslīgā intelekta teorētiskais risinājums**

**attiecībā pret iespējamo realizāciju**

Zinātniski pētnieciskais darbs datorzinātnēs

**Darba autori**:

Rīgas 64. vidusskolas 12. klases skolēni

Kristiāns Magons, Daniels Muļukins

**Darba vadītājs**:

SAP SE vadošais arhitekts

Krists Magons

**Darba konsultants**:

Rīgas 64. vidusskolas programmēšanas skolotājs

Eduards Bukovskis

Rīga 2025

**Anotācija**

Zinātniski pētniecisko darbu “Mākslīgā intelekta teorētiskais risinājums attiecībā pret iespējamo realizāciju” izstrādāja Rīgas 64. vidusskolas 12 INZ klases skolēni Kristiāns Magons un Daniels Muļukins.

Zinātniskais darbs tika veikts ar mērķi izpētīt mākslīgā intelekta nozares pamatus un realizēt mākslīgā intelekta modeli, lai noskaidrotu tā limitācijas pie arvien sarežģītākām problēmsituācijām. Teorētiskajā daļā tika izskatīta mākslīgā intelekta definīcija un tā attīstība, algoritmu sarežģītības noteikšana, kā arī divi klasiskās meklēšanas algoritmi. Pamatojoties uz mākslīgā intelekta pamatdefinīcijām, tika izstrādāts racionāli rīkojošs aģents – putekļsūcēja modelis, kura uzdevums ir iztīrīt visas netīrās istabas vidē. Praktiskajā daļā tika izmantota zema līmeņa programmēšana, lai izvairītos no augsta līmeņa abstrakcijām, kā arī nodrošinātu izpratni un pilnu kontroli pār izveidotā koda funkcionalitāti un sarežģītību. Izstrādājot pamata datu struktūras (saistītais saraksts, grafs un asociatīvais masīvs), tika izveidots racionāli rīkojošs aģents ar diviem meklēšanas algoritmiem, kā arī tam attiecīgā vide. Lai izpētītu izveidotā aģenta limitācijas, tika salīdzināti divi dažādi meklēšanas algoritmi un to izpildījums laika un atmiņas izteiksmē. Veicot mērīšanu, tika secināts, ka pareiza algoritma izvēle var būtiski uzlabot aģenta meklēšanas efektivitāti, tomēr galvenā pamatproblēma izrādījās vides izveidošana, ielasīšana un uzturēšana. Kaut arī algoritmu laika sarežģītība ir būtisks faktors, realizētajā modelī atmiņas faktors izrādījās izšķirošs ierobežojums attiecībā pret praktisku risinājumu iespējamību, lineāri pieaugot aģenta vides sarežģītībai.

Atslēgas vārdi: mākslīgais intelekts, klasiskā meklēšana, grafu teorija, datu struktūras, zema līmeņa programmēšana.

**Saturs**

[Ievads 4](#_Toc185633745)

[1. Ieskats mākslīgajā intelektā 5](#_Toc185633746)

[1.1. Kas ir mākslīgais intelekts? 5](#_Toc185633747)

[1.2. Izcelsme 6](#_Toc185633748)

[1.3. Uzplaukums 6](#_Toc185633749)

[2. Algoritmu sarežģītība 7](#_Toc185633750)

[2.1. Kas ir lielā O notācija? 7](#_Toc185633751)

[2.2. Lielās O notācijas nozīme šajā pētījumā 8](#_Toc185633752)

[3. Meklēšanas algoritmi 8](#_Toc185633753)

[3.1. A\* meklēšanas algoritms 8](#_Toc185633754)

[3.2. BFS meklēšanas algoritms 9](#_Toc185633755)

[3.3. Meklēšanas algoritmu nozīme šajā projektā 9](#_Toc185633756)

[4. Vides apraksts 10](#_Toc185633757)

[5. Heiristika 11](#_Toc185633758)

[6. Aģenta apraksts 11](#_Toc185633759)

[7. Datu struktūras 12](#_Toc185633760)

[7.1. Saistītais saraksts 12](#_Toc185633761)

[7.2. Rinda 12](#_Toc185633762)

[7.3. Grafs 13](#_Toc185633763)

[7.4. Asociatīvais masīvs 14](#_Toc185633764)

[8. Implementācijas telpas un laika sarežģītības novērtēšana 14](#_Toc185633765)

[8.1. Teorētiskais veiktspējas novērtējums 14](#_Toc185633766)

[8.2. BFS meklēšanas algoritma novērtēšana 14](#_Toc185633767)

[8.3. A\* meklēšanas algoritma novērtēšana 15](#_Toc185633768)

[9. Veiktspējas mērīšana 15](#_Toc185633769)

[10. Rezultāti 15](#_Toc185633770)

[10.1. Implementētais kods 15](#_Toc185633771)

[10.2. Statiskā testēšana 15](#_Toc185633772)

[10.3. Dinamiskā testēšana 15](#_Toc185633773)

[10.4. Statistiskā analīze (T-testi) 16](#_Toc185633774)

[Secinājumi 17](#_Toc185633775)

[Izmantotās literatūras un informācijas avotu saraksts 18](#_Toc185633776)

[Pielikumi 20](#_Toc185633777)

# Ievads

Pēdējo gadu laikā ir strauji attīstījušās datortehnoloģijas (Moore, 1965), kas līdzās sekmējušas arī mākslīgā intelekta (turpmāk tekstā – MI) attīstību, jo radās iespējas efektīvāk glabāt daudz datus mākoņos (Russell, Norvig, 2009, 27-28; Erickson, 2024). Arī jaudīgās videokartes ļāva ātrāk veikt skaitļošanu un apstrādāt vairāk datus (Chellapilla u.c., 2006), sniedzot iespējas realizēt tādas tehnoloģijas kā TensorFlow, kas paralēli izmanto vairākus procesorus, būtiski paātrinot matemātiskus aprēķinus (TensorFlow, 2024). Šādas skaitļošanas platformas ir brīvi pieejamas, pateicoties daudzkodolu videokaršu attīstībai. Pagaidām MI nav stingri definēts, jo atšķiras izpratnes par to. Vēsturiski ir nonākts pie četrām pamatdefinīcijām, kā izprast MI jēdzienu – domāšana vai rīkošanās cilvēciski, domāšana vai rīkošanās racionāli. (Russell, Norvig, 2009, 2)

MI rīki cilvēkiem jau ir kļuvuši par būtisku palīgu ikdienā, taču prognozes par tā tālāko nākotni ir dažādas. Parasti nepilnības rodas, kad spriedumus mēģina veikt, nevienojoties par MI definīciju, jo, kā jau tika minēts, MI sfēra ir ļoti plaša ar dažādām definīcijām. Sākot no šaha spēles risinājumiem līdz robotam, kas domā kā cilvēks. Tāpēc darba autori izvirzīja par mērķi realizēt savu racionālu aģentu, lai gūtu priekšstatu par MI limitācijām.

**Pētījuma mērķis**:

Izveidot MI modeli, kā racionāli rīkojošu aģentu, lai novērtētu tā veiktspēju, risinot arvien komplicētākas problēmas, un atrastu potenciālus uzlabojumus ātrākam risinājumam.

**Darba uzdevumi**:

1. Valodā C++ realizēt zema līmeņa datu struktūras un pamatalgoritmus (saistītais saraksts, rinda, grafs), kā arī iepazīties ar teksta failu apstrādi;
2. Izmantojot realizētās datu struktūras un OOP (Objektorientētā programmēšana) paradigmas, realizēt aģenta un vides implementāciju, kā arī aģenta darbību simulāciju;
3. Realizēt aģenta stāvokļa pārejas funkciju, izmantojot A\* un BFS meklēšanas algoritmus;
4. Novērtēt aģenta veiktspēju un noskaidrot aģenta potenciālās vajadzības pēc atmiņas daudzuma un laika;
5. Izvērtēt realizētā MI modeļa veiktspējas limitus.

**Hipotēze**:

Pieaugot vides sarežģītībai, aģentam ar teorētiski iespējamu risinājumu un ierobežotiem resursiem to realizēt kļūs arvien grūtāk līdz tas problēmu vairs praktiski nevarēs atrisināt, ja vien netiks uzlabota risinājuma atrašanas efektivitāte.

**Darbā izmantotās metodes**:

1. Simulācijas programmas implementācija.
2. Statiskā analīze veiktspējas novērtēšanai un potenciālajai uzlabošanai.
3. Dinamiskā testēšana – simulācijas programma tiks vairākkārt palaista. Simulācijas izpildes laikā tiks mērīti iegūtie rezultāti.
4. Empīrisko rezultātu statistiskā analīze (T-testi).

# Ieskats mākslīgajā intelektā

## Kas ir mākslīgais intelekts?

Cilvēki jau sen ir mēģinājuši saprast, kā tie spēj uztvert, saprast, paredzēt un ietekmēt pasauli, kas ir daudz plašāka par viņiem. MI sfēra tiecas vēl tālāk – tā nav tikai mēģinājums izprast, bet arī atveidot inteliģenci (Russell, Norvig, 2009, 1). Vadoties pēc Stjuarta Rasela (Stuart Russell) un Pītera Norviga (Peter Norvig) mācību grāmatas *Artificial Intelligence: A Modern Approach*, MI nav vienotas definīcijas. Izmantojot astoņas dažādas MI definīcijas (Russell, Norvig, 2009, 2), autori tās ir apvienojuši četrās galvenajās:

**Cilvēciska rīcība**: 1950. gadā Alans Tjūrings (Alan Turing) piedāvāja testu, definējot MI, kas ir plaši zināms, kā “Tjūringa tests”. Šis tests iekļauj pratinātāju, kurš aizklāti veic interviju. Ja pratinātājs nevar noteikt, vai atbildes sniedz dators vai cilvēks, tad dators ir nokārtojis testu. (Russell, Norvig, 2009, 2-3)

Tjūringa tests mūsdienās joprojām ir aktuāls, bet viedokļi par tā attīstību MI definēšanā ir mainījušies. Tjūringa tests paredz, ka dators spēj simulēt cilvēcisku rīcību, bet ne gluži rīkoties cilvēciski (Goldstein, Kirk-Giannini, 2023). Proti, mērķis ir, lai MI nonāk līdz konkrētajai rīcībai to apdomājot tāpat, kā cilvēki, nevis imitējot cilvēciskumu. Pagaidām šis mērķis paliek īpaši grūts, jo tas, kā cilvēki domā un rīkojas, vadoties pēc savām domām, nav līdz galam izprasts. (Wells, 2023)

**Cilvēciska domāšana**: MI pieeja, kas cieši saistās ar kognitīvo zinātni, jo, lai konstruētu cilvēciski domājošu MI modeli, ir jāzina precīzas un pārbaudāmas teorijas par cilvēka prātu. Tiekot pie pietiekami precīzas teorijas par prātu, ir iespējams to realizēt, kā datora programmu. Ja programmas ievade–izvade saskan ar cilvēka rīcību, tad tiek iegūts reāls pierādījums, ka šis programmas mehānisms varētu darboties arī cilvēkā. (Russell, Norvig, 2009, 3)

**Racionāla domāšana**: Racionāla domāšana saista loģikas jomu un izskata jautājumus par pareizu un sistematizētu domāšanu. Racionāla domāšana, kā MI pieeja, balstās uz nenoliedzamiem spriešanas procesiem, kur pie pareiziem pieņēmumiem, veicot deducēšanu, tiek iegūti loģiski secinājumi. Piemērs: “Sokrāts ir cilvēks. Visi cilvēki ir mirstīgi. Tātad Sokrāts ir mirstīgs.” (Russell, Norvig, 2009, 4)

**Racionāla rīcība**: Aģents ir kaut kas, kas rīkojas. Protams, visas datora programmas kaut ko dara, bet datora aģentam ir jāspēj vairāk – rīkoties autonomi, uztvert apkārtējo vidi sev apkārt, pastāvēt ilgstošu laiku, adaptēties izmaiņām, veidot un sasniegt mērķus. Racionāls aģents ir tāds, kas rīkojas, lai sasniegtu labāko rezultātu vai labāko sagaidāmo rezultātu. (Russell, Norvig, 2009, 4-5)

Šajā pētījumā MI pieeja ir putekļsūcēja modelis kā racionāli rīkojošs aģents. Tiks izstrādāts putekļsūcēja aģents (projekta kodu skatīt 1. pielikumā), kas pārvietosies pa vidi, kas sastāv no vairākām tīrām vai netīrām istabām, un to iztīrīs. Tas spēs uztvert, vai istabā, kurā tas atrodas, ir vai nav piesārņojums. Attiecīgi tas atkritumus sasūks, ja istabā tiks uztverts piesārņojums, vai pārvietosies uz nākamo istabu, ja istabā netiks uztverts piesārņojums. Kā jau rīkojoties racionāli (pamatojoties uz racionālu rīcību), putekļsūcēja aģenta mērķis ir visefektīvāk nokļūt no sākuma stāvokļa līdz gala stāvoklim, proti, nonākt stāvoklī, kad visas istabas ir tīras.

## Izcelsme

Ievērojamie rezultāti MI jomā gūti vien pēdējo gadu laikā, taču tas ir bijis garš process vairāku desmitu gadu garumā. Mēģinājumi veiksmīgi izveidot MI modeli ir bijuši gan ar lielām cerībām, gan vilšanām.

Pirmie atzītie darbi MI jomā datējami 1943. gadā, kad, izmantojot primitīvu psiholoģiju, apgalvojumu loģiku un Tjūringa skaitļošanas teoriju, tika izgudrots pirmais modelis, kas reprezentēja mākslīgus neironus. 1950. gadā tika izveidots pirmais neironu tīkla dators. Lai gan tas simulēja vien 40 neironus, tas sastāvēja no 3000 vakuuma caurulēm un tehnoloģijām, kas tika aizgūtas no bumbvedēja B-24 automātiskā pilota mehānisma pārpalikumiem. (Russell, Norvig, 2009, 16-17)

Ietekmīgs veikums 1956. gada vasarā bija Allena Nevela (Allen Newell) un Herberta Saimona (Herbert Simon) izveidotā spriešanas programma “Loģikas teorētiķis”, kas spēja domāt neskaitliski (Russell, Norvig, 2009, 17-18). Vēlāk viņi izveidojas programmu “Vispārējais problēmu risinātājs”, kas bija programma, spējīga atdarināt cilvēcisku domāšanu. Savukārt 1958. gads ir nozīmīgs ar Džona Makartija (John McCarthy) izveidoto teorētisko aprakstu programmai “Padomu ņēmējs”, kas hipotētiski spēja izmantot vispārējās pasaules zināšanas, lai meklētu problēmu risinājumus, un ko uzskatīja par pirmo pilnīgo MI sistēmu. (Russell, Norvig, 2009, 18-20)

Kad tik salīdzinoši īsā laika posmā tika izveidotas mašīnas, kas spēja domāt, mācīties un veidot – cerības par tuvākajiem nākotnes plāniem bija spožas. Ārpus jaunu metožu mēģinājumiem (Russell, Norvig, 2009, 22-27), MI joma uz ilgāku laiku stagnēja, jo viss atdūrās pret datoru nespēju veikt grūtākus vai apjomīgākus uzdevumus (Russell, Norvig, 2009, 20-22).

## Uzplaukums

MI 60 gadu attīstības laikā akcents lielākoties ir bijis algoritmu izvēlei un atbilstībai, taču šī gadsimta laikā lielāka uzmanība tikta vērsta datiem, un to milzīgajai pieejamībai dažādos avotos, interneta tīmekļos. Ar plašo datu pieejamību, MI modulēšana atvieglojas, jo ar mazāk informāciju nepieciešami cilvēciski piemēri un sarežģīti algoritmi, turpretī ar vairāk informāciju var iztikt ar primitīvāku algoritmu. Šī iemesla dēļ pieeja MI veidošanai ir mainījusies no mehāniskas zināšanu iekodēšanas uz datora pašmācību pie dotiem datiem. (Russell, Norvig, 2009, 27-28)

Lai apstrādātu datus un trenētu MI modeļus, ir nepieciešams veids, kā šos lielos daudzumus datu glabāt efektīvi. Lai to realizētu, dati tiek glabāti mākoņos, un to apstrādei tiek izmantoti jaudīgi skaitļotāji vai jaunas pieejas, kā datus apstrādāt.

Jau vairākus gadus, izpildoties Mūra likumam (Moore law) (Rupp, 2022), ir pieejami jaudīgi daudzkodolu procesori, kuri var ātri un efektīvi veikt daudzus uzdevumus. No otras puses, milzīgu daudzumu datu apstrāde, pat tādiem procesoriem, aizņemtu pārāk daudz laika, lai apstrāde tiktu veikta efektīvi. Tādēļ cilvēki pievērsās grafiskajiem procesoriem jeb videokartēm (Chellapilla u.c., 2006), jo, atšķirībā no procesoriem, tām ir daudz vairāk atsevišķu kodolu, lai veiktu efektīvo paralēlo skaitļošanu. Piemēram, grafiskais procesors “GeForce RTX 3070” satur 5888 kodolus, kuri paralēli veic savas funkcijas (Nvidia, 2021). Šī grafisko procesoru īpatnība – veikt vienlaikus paralēlus darbus – padara tās par ļoti efektīvu rīku paralēlajā datu apstrādē (Merritt, 2023). Spilgtākais piemērs videokaršu pielietošanā, lai veidotu un apmācītu MI, ir CNN (konvolucionālie neironu tīkli). Tas ir neironu tīkls, kuru visbiežāk izmanto, lai uztvertu un apstrādātu bildes. Lai paātrinātu bilžu apstrādi, bildes un neironu tīklu pārveido, kā divus masīvus, bet apstrādes rezultāts ir šo masīvu reizinājums. Masīvu reizinājumu var veikt paralēli ar videokaršu palīdzību, kas būtiski ieekonomē laiku. (Chellapilla u.c., 2006)

Sakarā ar lielo uzplaukumu MI jomā, lielie valodu modeļi (turpmāk tekstā – LVM) pēdējos gados ir kļuvuši par visātrāk popularizētām tehnoloģijām pasaulē. Piemēram, viens no LVM pārstāvjiem *ChatGPT* ir iekļauts vispopulārāko vietņu sarakstā pasaulē (Semrush, 2024). LVM ir transformatoru modeļi neironu arhitektūrā, kuriem ir iedoti tik daudz dati, ka tie spēj modelēt vai atbildēt ar loģiskiem, cilvēciskiem tekstiem, kā arī saprast kontekstu (Radford u.c., 2019; Cloudflare). Šīs spējas bija iemesls LVM straujajai popularizācijai, jo LVM ir viegli pielietot un tie ir noderīgs rīks, lai ātri kaut ko uzzinātu, apkopotu garus tekstus vai pat izveidot jaunus tekstus. Protams, LVM piemīt arī savas limitācijas, jo šīs tehnoloģijas tiek trenētas, balstoties uz datiem, kuri var gadīties nepatiesi, nepilnīgi vai nepietiekami daudz. Rezultātā LVM tehnoloģiju sniegtās atbildes var būt maldīgas. (Cloudflare)

Šo tehnoloģiju attīstības rezultātā, it īpaši pēdējos gados, ir visai strauji attīstījies MI, kuru ikdienā izmantojam, piemēram, rakstot *ChatGPT*, veidojot bildes ar *MidJourney* vai tulkojot tekstu ar *DeepL*.

# Algoritmu sarežģītība

## Kas ir lielā O notācija?

Lielā O notācija datorzinātnēs ir veids, kā pierakstīt algoritma sarežģītību, proti, cik daudz algoritma soļu skaits vai izmantotā atmiņa pieaug, palielinoties ievaddatu skaitam. Lielo O notāciju pieraksta, kā , kas cēlās no vācu matemātiķa Paula Gustava Heinriha Bahmaņa (Paul Gustav Heinrich Bachmann) pieraksta (Bachmann, 1894). apraksta funkcijas uzvedību, kad arguments tiecas uz noteiktu vērtību vai bezgalību, kur *O* – augšanas kārtībā un – argumenta lielums. (Black, 2019)

Tostarp datorzinātnēs definē, nevis, kā argumenta lielumu, bet, kā argumentu skaitu vai ievaddatu skaitu (Mohr, 2014). Piemēram, runājot par algoritmu ātrumu, nozīmēs, ka ar ievaddatu skaitu algoritms izpildīs soļus – algoritms izpildās lineāri, bet algoritms ar sarežģītību izpildīsies konstanti, veicot 1 soli. algoritmu aprakstīšanas metode ļauj viegli un saprotami salīdzināt algoritmus, jo, pat nezinot neko par algoritmiem, var secināt, ka algoritms ar sarežģītību būs ātrāks par algoritmu , jo < .

Piebilde: Šajā projektā aprakstot, salīdzinot vai pētot algoritmu sarežģītību, tā tiks saīsināta no uz tā, ka viens papildus solis algoritma izpildes laiku ietekmēs ļoti minimāli. Īsinot nelielus konstanta soļus vai atmiņas skaitu.

## Lielās O notācijas nozīme šajā pētījumā

MI veidošana, apmācīšana un pētīšana ir saistīta ar lielām un sarežģītām datu kopām. Pareiza algoritma un datu struktūru izvēle var būtiski mainīt risinājumu ātrumu un precizitāti. Lielā O notācija ir ērts rīks, kas ļauj viegli salīdzināt algoritmus un novērtēt to sarežģītību. Šī projekta izpildījumā tas ir svarīgi, jo ar lielo O notāciju tiks konkrēti aprakstīti pamatalgoritmu un datu struktūru ātrumi un prasības pēc atmiņas daudzuma. Zinot šo informāciju, būs iespējams veikt pamatotus spriedumus par MI veiktspējas limitācijām.

Apskatot lielo O notāciju citos dzīves piemēros, kā, piemēram, šahā, vēl joprojām ar mūsdienu tehnoloģijām nav iespējams realizēt aģentu, kurš spētu izveidot un apstrādāt visu stāvokļu kopu šai spēlei. Tai ir vismaz stāvokļu, kas ir vairāk par atomu skaitu novērojamā visumā (Russell, Norvig, 2009, 47). Vēlviens spilgts piemērs būtu RSA šifra, ko daudzas bankas izmanto, kā šifrēšanas veidu, kurā tiek reizināti divi milzīgi pirmskaitļi. RSA atšifrēšana aizņems pārāk ilgu laiku. Piemēram, 200 ciparu lielu reizinājuma atšifrēšana aizņems aptuveni gadus. (Rivest u.c., 1978)

# Meklēšanas algoritmi

## A\* meklēšanas algoritms

**Kas ir A\* meklēšanas algoritms?**

A\* ir informēts meklēšanas algoritms, kas strādā ar grafu, kuram piemīt heiristika (Zeng, 2007). Tā kā A\* algoritmam ir zināmas prioritātes virsotnēm vai ceļiem, lai sasniegtu mērķi ātrāk, tas izvēlēsies virsotnes vai ceļus, kuras ir tuvākas galapunktam. Piemērs, kā A\* algoritms atrod īsāko ceļu (skatīt 1. attēlu):



(1. attēls: A\* algoritma darbības grafiskais attēlojums, kā ceļi starp Latvijas pilsētām)

Avots: Autoru veidots attēls (2024)

A\* meklēšanas algoritma sarežģītība ir eksponenciāla , kur – sazarojumu faktors jeb, cik sazarojumu ir katrai virsotnei, bet – īsākais ceļa garums līdz galapunktam. Atmiņas sarežģītība ir visu apskatīto virsotņu skaits, kas sliktākajā gadījumā būs , kur – virsotņu skaits. (Zeng, 2017)

**Kur izmanto A\* meklēšanas algoritmu?**

A\* meklēšanas algoritmu izmanto dažādās jomās, visbiežāk GPS navigācijā, kur programmai var tikt iedots sākumpunkts un galapunkts, līdz kuram jānonāk pēc iespējas ātrāk. Navigācijas programma, ņemot vērā ceļa apstākļus un sastrēgumus, atgriež lietotājam optimālāko maršrutu. A\* meklēšanas algoritms arī saistās, piemēram, ar piegāžu ķēžu pārvaldības jomu, kur efektīva maršruta plānošana ir svarīga, lai ietaupītu laiku, benzīnu un resursus. Arī spēļu veidošanā var tikt izmantots A\* meklēšanas algoritms, piemēram, mašīnu sacīkstēs, lai nodrošinātu, ka citas mašīnas spēlē tiek līdzi. (AlmaBetter Bytes, 2024)

## BFS meklēšanas algoritms

**Kas ir BFS meklēšanas algoritms?**

BFS (Breadth-First Search) meklēšanas algoritms ir viens no vienkāršākajiem grafa meklēšanas algoritmiem. BFS meklēšanas algoritmam nav nekādu zināšanu par mērķa virsotņu iespējamo atrašanās vietu. BFS meklēšanas algoritms līdzvērtīgi un sistemātiski, sākot no sākuma virsotnes, izskata apkārtējās virsotnes soli pa solim līdz atrod mērķi vai izskata visu grafu, mērķi neatrodot. (CelerData, 2024)

BFS meklēšanas algoritma laika sarežģītība ir , kur – virsotņu skaits un – pāreju skaits. Šāda lineāra sarežģītība rodas, jo BFS meklēšanas algoritms katrai virsotnei izskata visas pārejas pēc kārtas. Atmiņas sarežģītība ir , jo BFS algoritms izmanto masīvu ar jau izskatītajām virsotnēm, kas rezultātā ir izmērā. (CelerData, 2024)

**Kur izmanto BFS meklēšanas algoritmu?**

BFS meklēšanas algoritmu izmanto, piemēram, maršrutu sastādīšanā, kad nav svarīgi ceļa garumi un distance tiek mērīta soļos no starta (CelerData, 2024). BFS meklēšanas algoritmu var izmantot arī, piemēram, labirintu problēmu risināšanā, kur BFS algoritms sliktākajā gadījumā izies caur visu labirintu, lai atrastu izeju. (Soularidis, 2022)

## Meklēšanas algoritmu nozīme šajā projektā

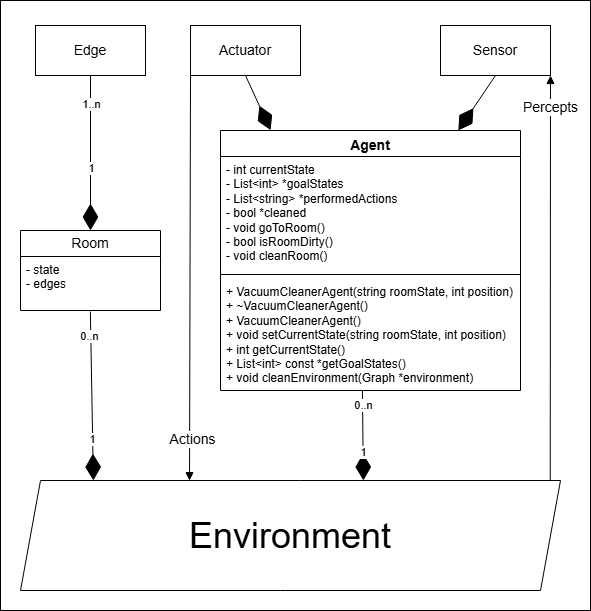
A\* meklēšanas algoritms, kā viens no efektīvākajiem klasiskās meklēšanas algoritmiem (Russell, Norvig, 2009, 95-99), šajā pētījumā tiks izmantots, lai gūtu priekšstatu par racionāli rīkojoša aģenta veiktspējas limitācijām. BFS meklēšanas algoritms, kā primitīvāks meklēšanas algoritms, tiks izmantots, lai salīdzinātu un izpētītu, kā dažādi algoritmi var ietekmēt aģenta veiktspēju un efektivitāti.

Lai realizētu meklēšanu, būs nepieciešama vide un racionāli rīkojošs aģents, kas šajā pētījumā būs putekļsūcēja modelis, kas atradīsies vidē ar tīrām vai netīrām istabām, kur istabas viena ar otru ir savienotas. Lai reprezentētu šo vidi, tiks izmantots stāvokļu grafs un A\* un BFS meklēšanas algoritmi, lai aģents varētu veikt meklēšanu un pārvietoties starp grafa virsotnēm.

# Vides apraksts

Putekļsūcēja aģenta vide sastāv no nemainīga skaita istabām un tajā nevar rasties jauns piesārņojums, ja aģents to vienreiz jau ir iztīrījis, tātad vide ir statiska. Tā ir arī pazīstama, jo aģents jau iepriekš zina, kā rīkoties vidē un kādas darbības tas var veikt. Šī projekta ietvaros vidē var atrasties tikai viens putekļsūcēja aģents. Vide ir pilnīgi pārredzama, jo aģents no jebkura sākumstāvokļa var atrast ceļu līdz beigu stāvoklim. Aģentam ir noteikti soļi, ko tas var veikt. Tas var pārbaudīt, vai istaba ir piesārņota, veikt sūkšanu vai pāriet uz citu istabu, padarot vidi par determinētu. Turklāt soļu skaits no sākuma stāvokļa (visas istabas ir piesārņotas) līdz gala stāvoklim (visas istabas ir tīras) ir ierobežots, kas nozīmē, ka vide ir diskrēta.

Vidi reprezentē stāvokļu grafs, kur katra virsotne atbilst konkrētam stāvoklim un pāreja starp virsotnēm nozīmē stāvokļu maiņu vides relāciju grafiskais attēlojums (skatīt 2. attēlu):



(2. attēls: vides relāciju grafiskais attēlojums)

Avots: Autoru veidota OOP klašu modeļa diagramma (2024)

Grafa stāvokļu skaits:

, kur – istabu skaits, reprezentē istabu stāvokļu skaitu, bet reizinājums ar nozīmē, ka aģents var atrasties jebkurā no istabām. Grafa pāreju skaits no tīras istabas ir , jo ir iespējams tikai pāriet uz citām istabām, taču no netīras istabas būs pāreju skaits, jo ir iespējams veikt arī tīrīšanu. Grafa pārejām pastāv variācijas, aģentam atrodoties gan no tīras, gan netīras istabas. Tas ir tādēļ, ka, atrodoties tīrā vai netīrā istabā, pāri paliek istabu, kuru variāciju skaits ir , un ko jāreizina ar , jo vienā no šīm istabām atrodas aģents. Rezultātā tiek iegūts grafa pāreju skaits:

# Heiristika

Klasiskā meklēšana nereti saistās ar bezgalīgi daudz iespējamo ceļu analīzi. Efektīva rezultāta priekšnosacījums bieži vien ir informācijas uzdošana par to, kuri ceļi ir apskatāmi pirmkārt, lai uzlabotu veiksmīga rezultāta atrašanas varbūtību. Šāda informēta algoritma izstrāde saistās ar heiristikas definīciju.

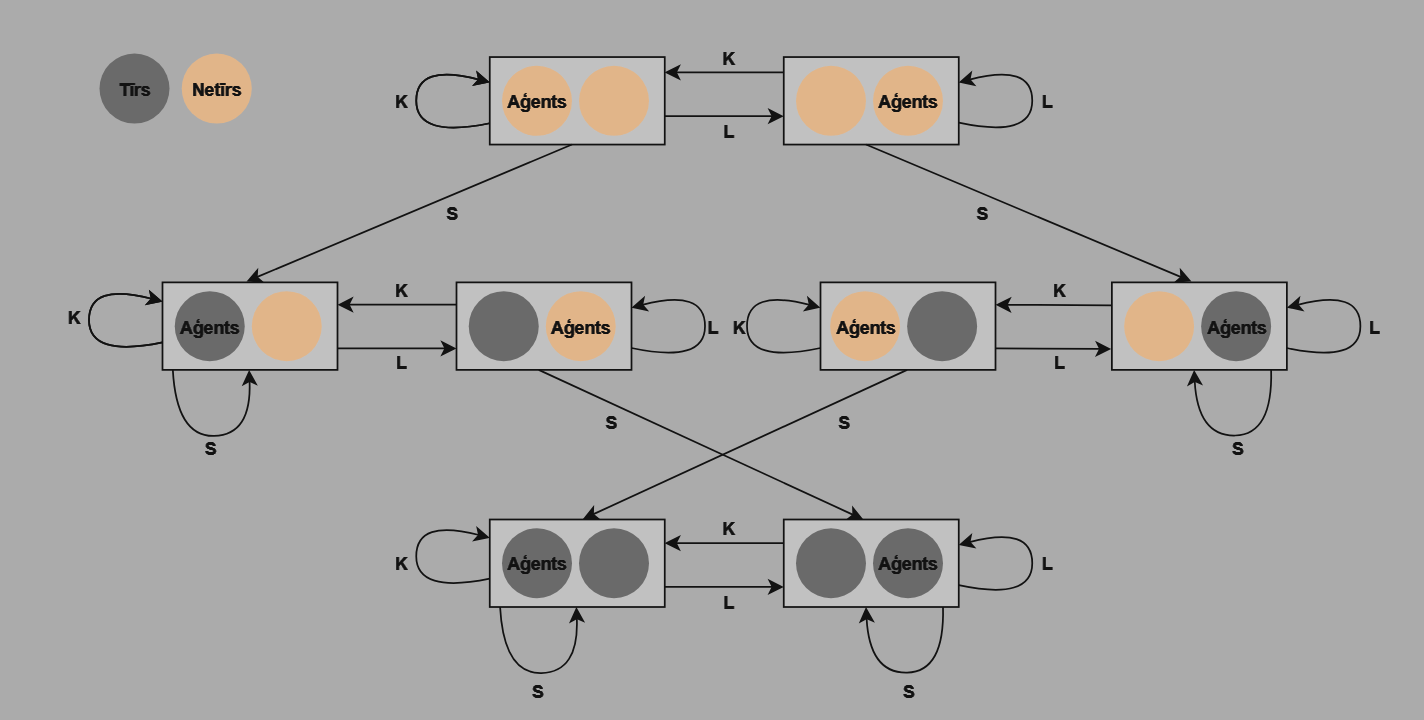
Heiristikas izveide vides un aģenta realizēšanā ir viena no svarīgākajām pamatproblēmām. Heiristika ir nepieciešama, lai piešķirtu aģentam racionālu risinājumu pareizu lēmumu izvēlē, proti, kuru zaru grafā izvēlēties, lai pēc iespējas ātrāk nonāktu mērķa stāvoklī. A\* algoritma gadījumā heiristika balstās uz dažāda prioritāšu stāvokļu pārejām grafā, savukārt šajā projektā katra pāreja grafā ir līdzvērtīga jeb 1 solis. Šīs būtiskās atšķirības dēļ, aģenta sensoru simulēšanai tika izvēlēts Uniform-Cost Search algoritma paveids jeb A\* algoritms, kura realizēšanai ir nepieciešams grafs ar pielāgotu heiristiku.

Visprimitīvākā pieeja šādas heiristikas izveidē būtu iziešana caur visām grafa virsotnēm, lai noskaidrotu attālumu jeb soļus līdz galapunktam. Šāda pieeja tomēr būtu pārāk sarežģīta, jo tā būtu mazāk efektīva par primitīvu meklēšanas algoritmu bez heiristikas. Piemēram, BFS meklēšanas algoritms sliktākajā gadījumā izies caur visu grafu, kas jau nozīmē, ka šāds algoritms sliktākajā gadījumā būs efektīvāks par algoritmu, kurš rēķinās heiristiku, vispirms izskatot visu grafu. Par alternatīvu tika veikta sakarību meklēšana ievaddatos, kuriem vienmēr pēc definīcijas piemīt noteiktas sakarības, lai heiristika nebalstītos uz liekām kalkulācijām, bet gan zināmām sakarībām.

Stāvokļu failā (skatīt 2. pielikumu) katra virsotne var reprezentēt tīru vai netīru istabu. No tīras istabas putekļsūcēja aģents var aiziet uz jebkuru citu istabu, tātad no šīs virsotnes būs pārejas, kur – istabu skaits. No netīras istabas putekļsūcēja aģents var gan aiziet uz citu istabu, gan iztīrīt esošo, tātad no šīs virsotnes būs pārejas. Šī ir būtiska sakarība, kuru aģents izmantos, lai noteiktu, vai istaba ir tīra vai netīra. Otrkārt, ir svarīgi identificēt, kura pāreja aģentam nozīmēs tīrīšanu, ja tas atrodas netīrā istabā. Saskaņā ar vides ievadfailu, pārejas no netīras istabas uz citām istabām vienmēr būs skaitliski tuvas pašreizējās virsotnes skaitliskajai reprezentācijai, savukārt tīrīšana vienmēr būs skaitliski lielāka. Šāda sakarība pastāv, jo faila sākuma stāvokļi, kuros visas istabas ir netīras, ir secīgi pirmie stāvokļi, bet gala stāvokļi, kuros visas istabas ir tīras, ir pēdējie stāvokļi. Attiecīgi istabas iztīrīšana aizvedīs tuvāk gala stāvoklim, kurš ir skaitliski lielāks. Apskatot vairākus failus, tika iegūta sakarība, ja *pašreizējā stāvokļa skaitliskā reprezentācija <= ar pārejas skaitlisko reprezentāciju - istabu skaitu*, tad šī pāreja nozīmē istabas iztīrīšanu. Jāpiezīmē, ka šī sakarība fundamentāli prasa loģiski sakārtotus vides realizācijas failus, kas vispārīgā gadījumā attiecīgi prasa priekšapstrādi.

# Aģenta apraksts

Šī projekta MI pieeja ir putekļsūcēja modelis kā racionāli rīkojošs aģents (koda realizāciju skatīt 1. pielikumā). Matemātiski tas nozīmē, ka aģenta funkcija saņem ievaddatus, kurus aģents ar sensoru palīdzību nolasa, apstrādā un atgriež aktuatoru soļus. Aģenta funkcijas galvenais uzdevums ir maksimalizēt lietderību, lai, veicot visu istabu tīrīšanu, netiktu veikti lieki soļi un patērēti lieki resursi. Proti, ceļš no aģenta stāvokļa līdz mērķa stāvoklim ir visīsākais jeb aģenta darbību skaits ir vismazākais. Grafisks attēlojums aģenta darbībai pie divām vidē piesārņotām istabām (skatīt 3. attēlu):



(3. attēls: Putekļsūcēja aģenta darbības grafiskais attēlojums divām vidē piesārņotām istabām, kur K – iet pa kreisi, L – iet pa labi, S - sūkt)

Avots: Autoru veidots attēls (2024)

# Datu struktūras

## Saistītais saraksts

Saistītais saraksts (skatīt 3. pielikumu) ir lineāra datu kolekcija, kas sastāv no saraksta elementiem, kuri satur datus un norādi uz nākamo elementu. Šī projekta ietvaros saraksta elementam ir arī norāde uz iepriekšējo elementu, lai uzlabotu saraksta dinamiskumu. Saistītajam sarakstam ir sākums un beigas, kā arī saraksta garums jeb elementu skaits. Atšķirībā no masīva datu struktūras, kurā dati tiek glabāti statiskā atmiņas apgabalā, un kuram lielumu ir jānosaka pirms kompilēšanas, saistītais saraksts elementus glabā dažādos atmiņas apgabalos. Tas ļauj viegli mainīt sarakstu, izmainot elementu norādes. No otras puses elementu atgūšana ir ilgāka, jo nav zināms, kur elementi atrodas, un ir nepieciešams iziet caur visām norādēm, lai atrastu vajadzīgo elementu. Tas nozīmē, ka saistītais saraksts ir efektīvs, kad ir nepieciešams dinamiskums, turpretī mazāk efektīvs par masīvu, ja ir nepieciešama datu atgūšana.

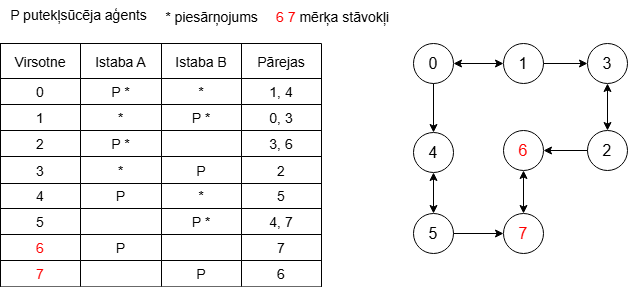
## Rinda

Rinda (skatīt 3. pielikumu) ir lineāra datu kolekcija, kuras funkcionālais koncepts ir *pirmais iekšā-pirmais ārā*. Tas nozīmē, ka pirmais tiek atgūts tas elements, kurš visilgāk jeb pirmais atrodas rindā. Šī projekta ietvaros rinda ir nepieciešama BFS meklēšanas algoritma implementēšanai, kur grafa izskatīšana notiek pakāpeniski. Proti, jaunatklātās virsotnes tiek pievienotas rindai un pēc kārtas tiek izskatītas. Kā tas tika veikts šajā projektā, rindas datu struktūru var implementēt, izmantojot saistīto sarakstu un ievērojot rindas konceptu.

## Grafs

Grafs (skatīt 3. pielikumu) ir nelineāra datu struktūra, kas sastāv no virsotnēm un pārejām, kur pāreja ir savienojums starp divām virsotnēm. Šajā projektā grafs reprezentē putekļsūcēja aģenta vidi, kur virsotne ir aģenta stāvoklis un pārejas nozīmē stāvokļu maiņu. Putekļsūcēja aģenta gadījumā grafa pārejas var būt arī vienpusējas, jo, iztīrot istabu, to vairs nevar atgriezt netīru.

Grafam piemīt virsotņu, pāreju un istabu skaits, kā arī virsotnes ar pārejām. Ir divas pamatpieejas grafa virsotņu un to pāreju glabāšanai – matricas vai saistītā saraksta no virsotņu pārejām izmantošana. Matricas pieeja konstantā laikā O(1) ļauj noskaidrot, vai eksistē ceļš starp divām virsotnēm, bet telpas ziņā izmantos atmiņas, kur – virsotņu skaits. Saistītā saraksta pieeja ļauj dinamiski pievienot un noņemt virsotnes, kā arī tā efektīvāk ļauj izmantot atmiņu , kur – virsotņu skaits un – pāreju skaits. Savukārt laika sarežģītība, izmantojot saistītos sarakstus, pieaug, jo, lai piekļūtu noteiktai pārejai, ir jāiet cauri saistītajam sarakstam. Tā kā ir zināms, ka katrai virsotnei būs vai pāreju, kur – istabu skaits, lai efektīvāk izmantotu atmiņu, tiek izmantota saistīto sarakstu pieeja, jo, izmantojot matricu, tiks patērēta nevajadzīgi daudz atmiņa. Grafisks attēlojums grafam divu vidē piesārņotu istabu situācijā (skatīt 4. attēlu):



(4. attēls: Divu vidē piesārņotu istabu grafa reprezentācija)

Avots: Autoru veidots attēls (2024)

**Virsotne**

Virsotne (skatīt 3. pielikumu) ir grafa pamatsastāvdaļa, kam piemīt savs indekss un rinda ar kaimiņvirsotnēm jeb pārejām uz citām virsotnēm. Lai grafam pievienotu jaunu pāreju, virsotnei *A* padod pāreju starp virsotni *A* un *B* un izsauc metodi, kas pievieno pāreju *(A, B)*. Katra virsotne reprezentē kaut kādu aģenta stāvokli, un no tām ir pārejas uz citām virsotnēm jeb vides stāvokļiem.

**Pāreja**

Grafā izmanto pārejas (skatīt 3. pielikumu), lai attēlotu ceļus jeb savienojumus starp divām virsotnēm. Pārejai piemīt virsotne *A* un virsotne *B*, ilustrējot savienojumu starp tām. Nereti pārejām piemīt arī izmaksa jeb attālums starp virsotnēm, taču šī projekta ietvaros šāda situācija netiek izskatīta, pieņemot visas pārejas par līdzvērtīgām. Putekļsūcēja aģenta vidē pāreja nozīmē stāvokļu maiņu. Piemēram, ja stāvoklis *A* reprezentē aģentu netīrā istabā, tad pāreja *(A, B)* nozīmēs, ka aģents veic sūkšanu, kur stāvoklis *B* reprezentē aģentu tajā pašā istabā, bet tīrā.

## Asociatīvais masīvs

Asociatīvais masīvs (skatīt 3. pielikumu) ir datu struktūra, kas efektīvi ļauj glabāt un iegūt vērtības ar tām saistītām, unikālām atslēgām. Tas nozīmē, ka vērtību iegūšana notiek, izmantojot tām saistīto atslēgu. Šajā projektā asociatīvais masīvs ir izveidots, kā heštabula jeb jaucējtabula, kas var saturēt tikai veselus skaitļus. Tā izmanto dinamisku masīvu no saistītajiem sarakstiem, kas savukārt sastāv no pāriem (atslēga-vērtība). Kad asociatīvajam masīvam pievieno vērtību ar atslēgu, tad atslēgai tiek aprēķināts heša kods un vērtība tiek ievietota masīvā pēc iegūtā heša koda. Ja divām vērtībām ir vienāds heša kods, notiek sadursme, kas nozīmē, ka attiecīgās šūnas saistītajā sarakstā būs divas vērtības. Labākajā gadījumā katrā asociatīvā masīva saistītajā sarakstā atrodas viens elements. Lai šim ideālam pietuvotos, palielinoties vērtību skaitam, asociatīvais masīvs arī tiek palielināts, lai izkliedētu vērtības.

Šajā projektā asociatīvo masīvu izmanto BFS meklēšanas algoritmam, lai saglabātu virkni ar jau apmeklētajām virsotnēm, lai algoritma beigās sastādītu visātrāko ceļu.

# Implementācijas telpas un laika sarežģītības novērtēšana

## Teorētiskais veiktspējas novērtējums

Šī projekta ietvaros tiek salīdzināta aģenta veiktspēja, izmantojot dažādas aģenta stāvokļa pārejas programmas. Proti, tiek apskatīts klasisks BFS meklēšanas algoritms un informēts meklēšanas algoritms ar eksperimentāli noteiktu heiristiku. Neskatoties uz to, ka aģents var darboties dažādās vides variācijās, kā pamata stāvoklis sarežģītība novērtēšanai tiek uzskatīts stāvoklis, kad visas istabas ir netīras. Vienā gadījumā aģenta stāvokļa programma tiek realizēta, izmantojot BFS meklēšanas algoritmu, jo tas neizmanto heiristiku un meklē ceļu, ejot caur katru grafa virsotni pēc kārtas. Otrā gadījumā tiek izmantots A\* meklēšanas algoritma paveids, kas balstās uz pielāgotu heiristiku, kas ļauj aģentam saprast, kura ir pareizā pāreja.

## BFS meklēšanas algoritma novērtēšana

BFS meklēšanas algoritms, lai atrastu īsāko ceļu, sliktākajā gadījumā izies caur visu grafu, tāpēc laika sarežģītība tam būs , kur – virsotņu skaits un – pāreju skaits. Tā kā šī projekta ietvaros BFS tiek izmantots īsākā ceļa noteikšanai, telpas sarežģītība ir . atmiņa tiek izmantota masīvam, kurš glabā apmeklētās virsotnes. atmiņa tiek izmantota mapei, kas satur visas atsauces uz noietajiem ceļiem. Savukārt ir gala stāvokļu skaits, tā kā aģents var atrasties jebkurā no iztīrītajām istabām. Tā kā ir zināms virsotņu un pāreju skaits pie istabām (skatīt 4. nodaļu), varam izteikt BFS meklēšanas algoritma teorētisko sarežģītību pie istabām.

Laika sarežģītība pie istabām:

Telpas sarežģītība pie istabām:

## A\* meklēšanas algoritma novērtēšana

A\* meklēšanas algoritma sarežģītība ir tieši atkarīga no izmantotās heiristikas, kas šī projekta gadījumā ir saistīta ar nevajadzīgo zaru izslēgšanu. Izpētot implementēto algoritmu (skatīt 1. pielikumā), tika iegūta laika sarežģītība sliktākajā gadījumā , kur – istabu skaits. Aģents, atrodoties katrā netīrajā istabā, veiks meklēšanu tīrīšanas pārejai, kas sliktākajā gadījumā tiks veikts soļos. Tā kā aģents saglabā apmeklētās virsotnes, nākamās netīrās istabas meklēšana neietekmēs iepriekš iegūto algoritma sarežģītību , jo katra nākamā istaba būs netīra (pēc pieņēmuma, ka pamata stāvoklis ir visas netīras istabas), bet jau apmeklētās istabas aģents neapskatīs. Savukārt atmiņas sarežģītība ir , kur pirmais – apmeklētās istabas, otrais – mērķa stāvokļi un – aģenta veiktās darbības. Rezultātā aģents pārbaudīs, izies un iztīrīs katru istabu, izņemot pirmo istabu, kuru nav jāiet, taču šo soli vērā neņem, jo tas ir konstants viens solis.

Laika sarežģītība pie istabām:

Telpas sarežģītība pie istabām:

# Veiktspējas mērīšana

Šī projekta mērķis ir izmērīt racionāla aģenta veiktspēju gan laika, gan atmiņas izmantošanas ziņā, izmantojot klasisko meklēšanu. Tika mērīti divi meklēšanas algoritmi – BFS meklēšanas algoritms un A\* meklēšanas algoritms ar pašveidotu heiristiku. Par atskaites punktu tika noteikts brīdis, kad algoritms uzsāka meklēšanu, bet mērīšanas beigu punktu tika noteikts brīdis, kad algoritms pabeidz meklēšanu. Tika izmērīta arī grafa laika un atmiņas sarežģītība, lai salīdzinātu vides izveidi attiecībā pret meklēšanu tajā.

Aģenta laika sarežģītība tika mērīta, izmantojot C++ bibliotēku *chrono*. Lai noteiktu algoritma aizņemto laiku, tika noteikts sākuma un beigu punkts, kuru starpība ir patērētais laiks mērīšanai. Savukārt, lai noteiktu aģenta izmantoto atmiņas daudzumu, tika izmantota C++ bibliotēkas *windows.h* un *psapi.h*. Arī atmiņas mērīšanā patērētā atmiņa ir algoritma beigu un starta starpība. Iegūtie rezultāti tika ierakstīti atsevišķā logs.txt failā, izmantojot logera instanci (skatīt 4. un 5. pielikumu). Grafa sarežģītība tika mērīta tāpat, kā algoritmi – sākuma punkts ir grafa izveides sākums, bet beigu punkts ir izveidots grafs (skatīt 6. pielikumu).

Eksperimenta vide fiksēta un aprakstīta pielikumā (skatīt 4., 5. un 6. pielikumu)

# Rezultāti

## Implementētais kods

## Statiskā testēšana

## Dinamiskā testēšana

## Statistiskā analīze (T-testi)

Katra konfigurācija tika mērīta 25 reizes laikam un 10 reizes atmiņai (iegūtos rezultātus skatīt 7., 8., 9. un 10. pielikumā). Eksperimenta rezultātiem tika veikti absolūtās un relatīvās kļūdas aprēķini pēc Stjūdenta T testa formulas:

– absolūtā kļūda

– Stjūdenta koeficients, kura vērtība ir atkarīga no mērījumu skaita un nepieciešamās ticamības varbūtības

– gadījuma noviržu kvadrātu summa ( – vidējā kvadrātiskā novirze, ko izmanto, lai noteiktu datu precizitāti jeb analizētu datu izkliedi attiecībā pret vidējo vērtību)

– mērījumu skaits

Formula relatīvās kļūdas aprēķināšanai:

– relatīvā kļūda

– mērījumu vidējā aritmētiskā vērtība

# Secinājumi

Salīdzinot aģenta meklēšanas algoritmus (BFS un A\*) tika novērota būtiska izmaiņa laika un patērētās atmiņas izteiksmē. Pēc iegūtām teorētiskām laika un telpas sarežģītību formulām un to salīdzināšanas praksē, tika secināts, ka aģents ar informētu stāvokļu pārejas algoritma palīdzību sākotnēji eksponenciālas komplicitātes problēmu spēj risināt polinomiālā (kvadrātiskā) laikā. Savukārt eksponenciālo telpas sarežģītību realizē lineāri. Tātad pareiza meklēšanas algoritma izvēle var ievērojami uzlabot aģenta veiktspēju, kā arī izvirzītā hipotēze par algoritmu sarežģītību un iespējamiem efektivitātes uzlabojumiem, definējot meklēšanas heiristiku, atbilda novērotajiem rezultātiem grafikā (skatīt 4. attēlu).

Lai gan algoritma izvēle var uzlabot aģenta darbību, galvenā pamatproblēma izrādījās vides faila izveidošana, kā arī paša grafa, kas reprezentē vidi, ielasīšana un uzturēšana. Vien 20 istabu saturēšanai tika izmantota visa datoram pieejamā atmiņa, kas ir aptuveni 10GB operatīvās atmiņas. Lai 20 istabu failu izveidotu, bija nepieciešams izmantot citu datoru un šī faila izveidei bija nepieciešami 24GB operatīvās atmiņas. Tāpēc svarīgi ir meklēt veidus, kā efektīvāk izveidot un glabāt aģenta vidi, piemēram, vides failu reprezentēt binārajā formā. Var secināt, ka šī projekta un attiecīgi klasiskās meklēšanas galvenā limitācija ir atmiņa, jo, kaut arī A\* algoritms ar pašizveidotu heiristiku atmiņu izmanto lineāri, vides telpas sarežģītība joprojām ir eksponenciāla.

Apstiprināt hipotēzi izdevās tikai, pētot BFS meklēšanas algoritmu, jo tas izmanto ļoti daudz atmiņu, sasniedzot pieejamās operatīvās atmiņas limitu. Tostarp izveidotais A\* algoritms vairākkārt pārspēja BFS algoritma efektivitāti meklēšanā, tāpēc, lai pārbaudītu hipotēzi A\* algoritmam, būtu nepieciešami vides faili ar vairāk istabām. Tomēr jāpiezīmē, ka maksa par algoritma ātrdarbības uzlabošanu un heiristikas uzdošanu saistās ar ievaddatu sakārtošanas nepieciešamību. Ja vides fails nebūtu mākslīgi izveidots, bet būtu nejauši sakārtots bez sakarībām, tad izveidoto A\* algoritmu pielietot neizdotos. Tādā gadījumā heiristikas izveide būtu sarežģītāka problēma, nekā grafa iziešana ar primitīvo BFS meklēšanas algoritmu.

Šī pētījuma uzlabošanai vērtīgi būtu uzlabot vides failu izveidošanu, lai iegūtu plašāku pētāmo datu kopu. Kā arī ieviest dažādību putekļsūcēja aģenta vidē un tā darbībā. Piemēram, piedot videi dinamiskumu un stohastiskumu vai ieviest vidē vairākus aģentus.

# Izmantotās literatūras un informācijas avotu saraksts

1. AlmaBetter Bytes (2024). *Real-world Applications of the A\* Algorithm with Examples* [tiešsaiste]. [skatīts 2024. gada 7. decembrī]. Pieejams: <https://www.almabetter.com/bytes/tutorials/artificial-intelligence/a-star-algorithm-in-ai>
2. Bachmann, P. G. H. (1894). Analytische Zahlentheorie. 401 lpp.
3. Black, P. E. (2019). *big-O notation* [tiešsaiste]. [skatīts 2024. gada 25. septembrī]. Pieejams: <https://xlinux.nist.gov/dads/HTML/bigOnotation.html>
4. CelerData (2024). *What is Breadth-First Search (BFS)?* [tiešsaiste]. [skatīts 2024. gada 2. decembrī]. Pieejams: <https://celerdata.com/glossary/breadth-first-search-bfs>
5. Chellapilla, K., Puri, S., Simard, P. (2006). *High Performance Convolutional Neural Networks for Documenting Processing*. Francija: Rennes universitāte
6. Cloudflare. *What is a large language model (LLM)?* [tiešsaiste]. [skatīts 2024. gada 10. oktobrī]. Pieejams: <https://www.cloudflare.com/en-gb/learning/ai/what-is-large-language-model/>
7. Erickson, J. (2024). *What is Cloud Computing? AI and Cloud Computing Explained* [tiešsaiste]. [skatīts 2024. gada 23. septembrī]. Pieejams: <https://www.oracle.com/artificial-intelligence/ai-cloud-computing/>
8. Goldstein, S., Kirk-Giannini, C. D. (2023). *Is the Turing test still relevant?* [tiešsaiste]. [skatīts 2024. gada 5. oktobrī]. Pieejams: <https://theconversation.com/ai-is-closer-than-ever-to-passing-the-turing-test-for-intelligence-what-happens-when-it-does-214721>
9. Merritt, R. (2023). *Why GPUs Are Great for AI* [tiešsaiste]. [skatīts 2024. gada 30. septembrī]. Pieejams: <https://blogs.nvidia.com/blog/why-gpus-are-great-for-ai/>
10. Mohr, A. (2014). *Quantum Computing in Complexity Theory and Theory of Computation*. Amerika: Dienvidu Ilinoisas universitāte Karbondeilā
11. Moore, G. E. (1965). *Cramming more components onto integrated circuits* [tiešsaiste]. IEEE Solid-State Circuits Society Newsletter [skatīts 2024. gada 23. septembrī]. Pieejams: <http://cva.stanford.edu/classes/cs99s/papers/moore-crammingmorecomponents.pdf>
12. Nvidia (2021). *GeForce RTX 30 Series* (2021). [tiešsaistē]. [skatīts 2024. gada 30. septembrī]. Pieejams: <https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/>
13. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I. (2019) Language Models are Unsupervised Multitask Learners [tiešsaistē]. [skatīts 2024. gada 10. oktobrī]. Pieejams: <https://www.bibsonomy.org/bibtex/b926ece39c03cdf5499f6540cf63babd>
14. Rivest, R. L., Shamir, A., Adleman, L. (1978). *A method for obtaining digital signatures and public-key cryptosystems* [tiešsaiste]. [skatīts 2024. gada 10. oktobrī]. Pieejams: <https://dl.acm.org/doi/10.1145/359340.359342>
15. Rupp, K. (2022). *Moore's law: The number of transistors per microprocessor* [tiešsaiste]. [skatīts 2024. gada 30. septembrī]. Pieejams: <https://ourworldindata.org/grapher/transistors-per-microprocessor>
16. Russell, S. J., Norvig, P (2009). *Artificial Intelligence: A Modern Approach (3rd ed.)*. Prentice Hall. 1132 lpp.
17. Semrush (2024). *Top websites in Worldwide (All industries)* [tiešsaistē]. [skatīts 2024. gada 10. oktobrī]. Pieejams: <https://www.semrush.com/trending-websites/global/all>
18. Soularidis, A. (2022). *Solve Maze Using Breadth-First Search (BFS) Algorithm in Python* [tiešsaiste]. [skatīts 2024. gada 2. decembrī]. Pieejams: <https://plainenglish.io/blog/solve-maze-using-breadth-first-search-bfs-algorithm-in-python-7931acbe8a93>
19. TensorFlow (2024). *Migrate single-worker multiple-GPU training* [tiešsaistē]. [skatīts 2024. gada 23. septembrī]. Pieejams: <https://www.tensorflow.org/guide/migrate/mirrored_strategy>
20. Wells, S. (2023). *Is the Turing Test Dead?* [tiešsaiste]. [skatīts 2024. gada 5. oktobrī]. Pieejams: <https://spectrum.ieee.org/turing-test>
21. Zeng, W., Church, R. (2007). *Finding shortest paths on real world networks: the case for A\** [tiešsaiste]. [skatīts 2024. gada 30. oktobrī]. Pieejams: <https://zenodo.org/records/979689>

# Pielikumi

1. pielikums

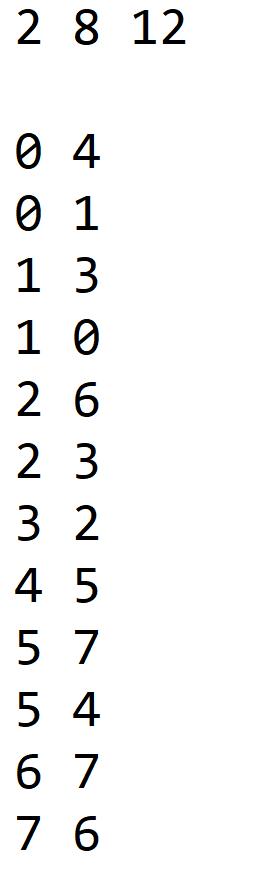
“**Pētījumā izveidotais kods vietnē GitHub**”

Skatīt izveidoto kodu [šeit](https://github.com/Galaxy2268/VacuumCleanerAgent) (*github.com/Galaxy2268/VacuumCleanerAgent*)

Avots: Autoru izveidotais kods (2024)

2. pielikums

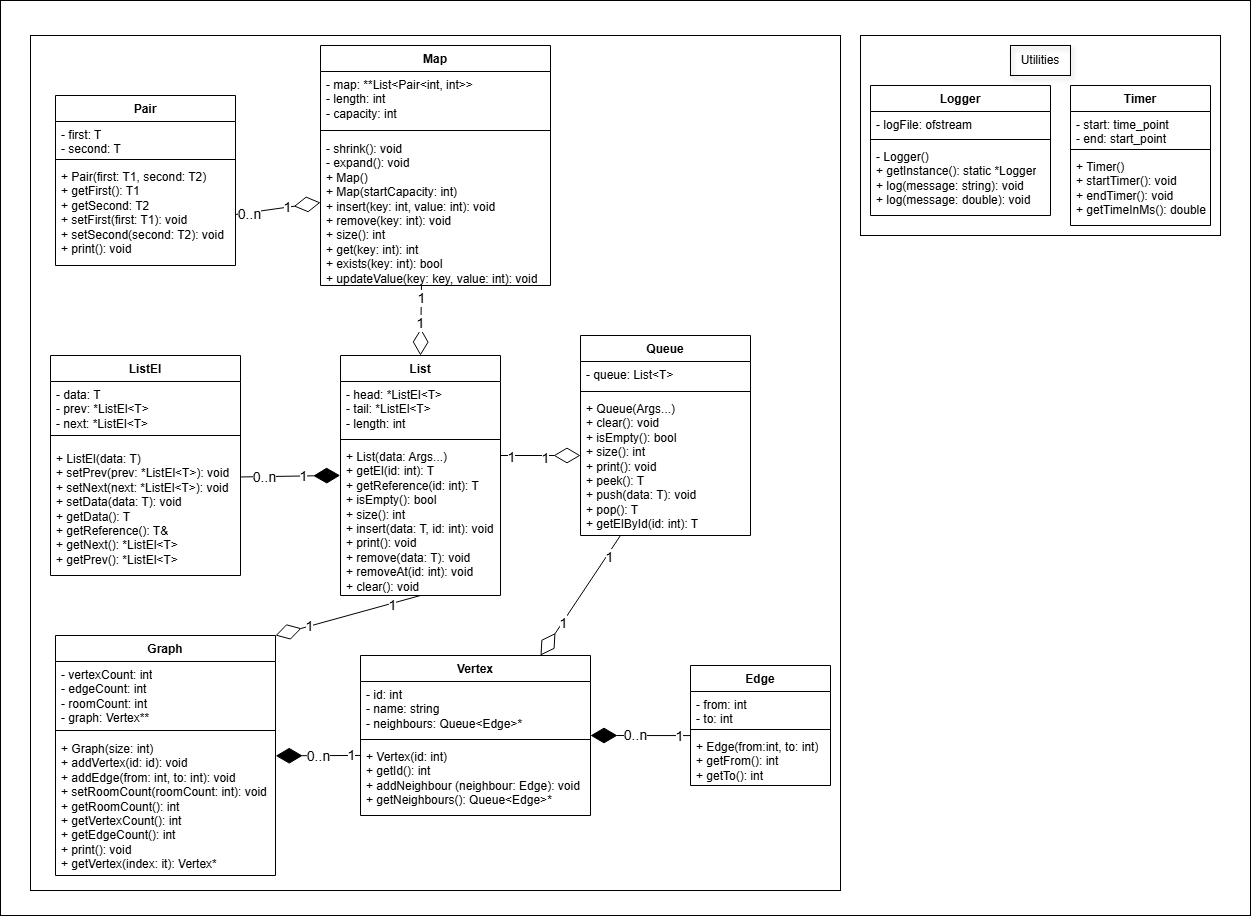
“**Divu vidē piesārņotu istabu faila reprezentācija**”



Avots: Autoru veidots attēls (2024)

3. pielikums

“**Implementēto datu struktūru relāciju grafiskais attēlojums**”



Avots: Autoru veidota relāciju modeļa diagramma (2024)

4. pielikums

“**Kods deviņpadsmit vidē piesārņotu istabu laika mērīšanai**”



Avots: Autoru veidots kods (2024)

5. pielikums

“**Kods deviņpadsmit vidē piesārņotu istabu atmiņas mērīšanai**”



Avots: Autoru veidots kods (2024)

6. pielikums

“**Kods grafa ielasīšanas mērīšanai**”

Avots: Autoru veidots kods (2024)

7. pielikums

“**A\* meklēšanas algoritma laika patēriņš (milisekundēs) atkarībā no istabu skaita**”

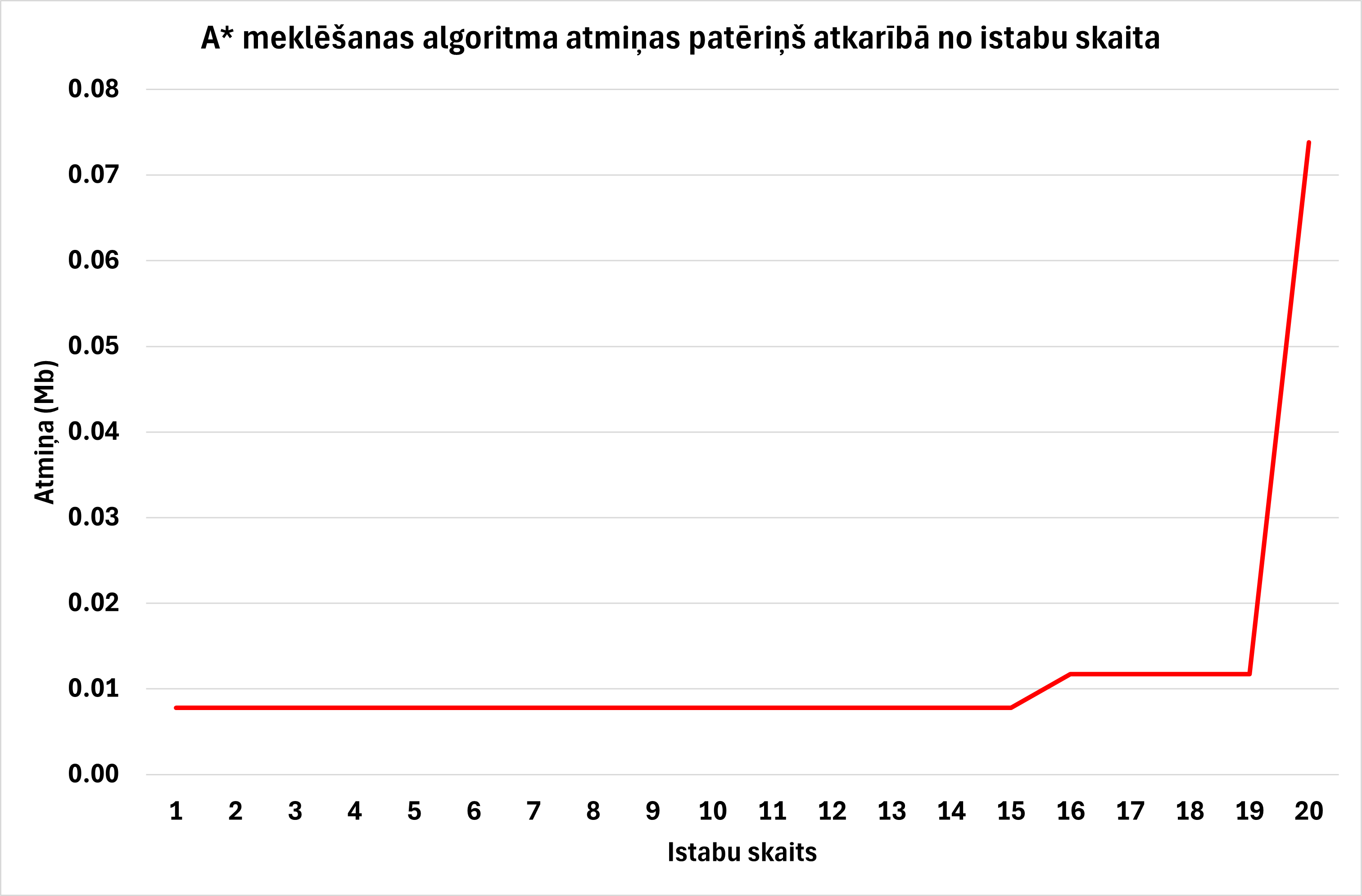
Attēls, kurā ir teksts, ekrānuzņēmums, rinda, fonts

Apraksts ģenerēts automātiski

Avots: autoru veidots grafiks (2024)

8. pielikums

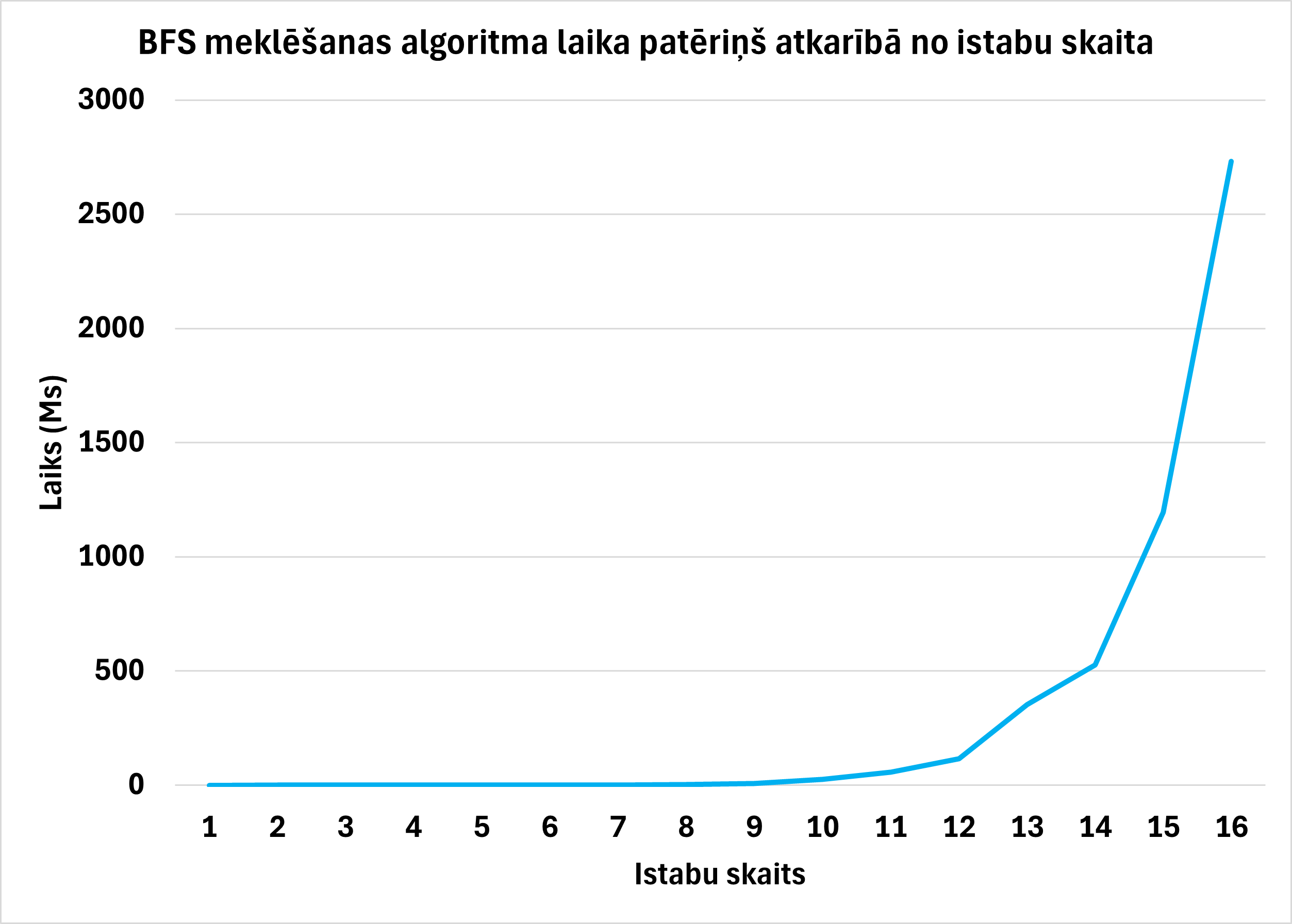
“**A\* meklēšanas algoritma atmiņas patēriņš (megabaitos) atkarībā no istabu skaita**”



Avots: autoru veidots grafiks (2024)

9. pielikums

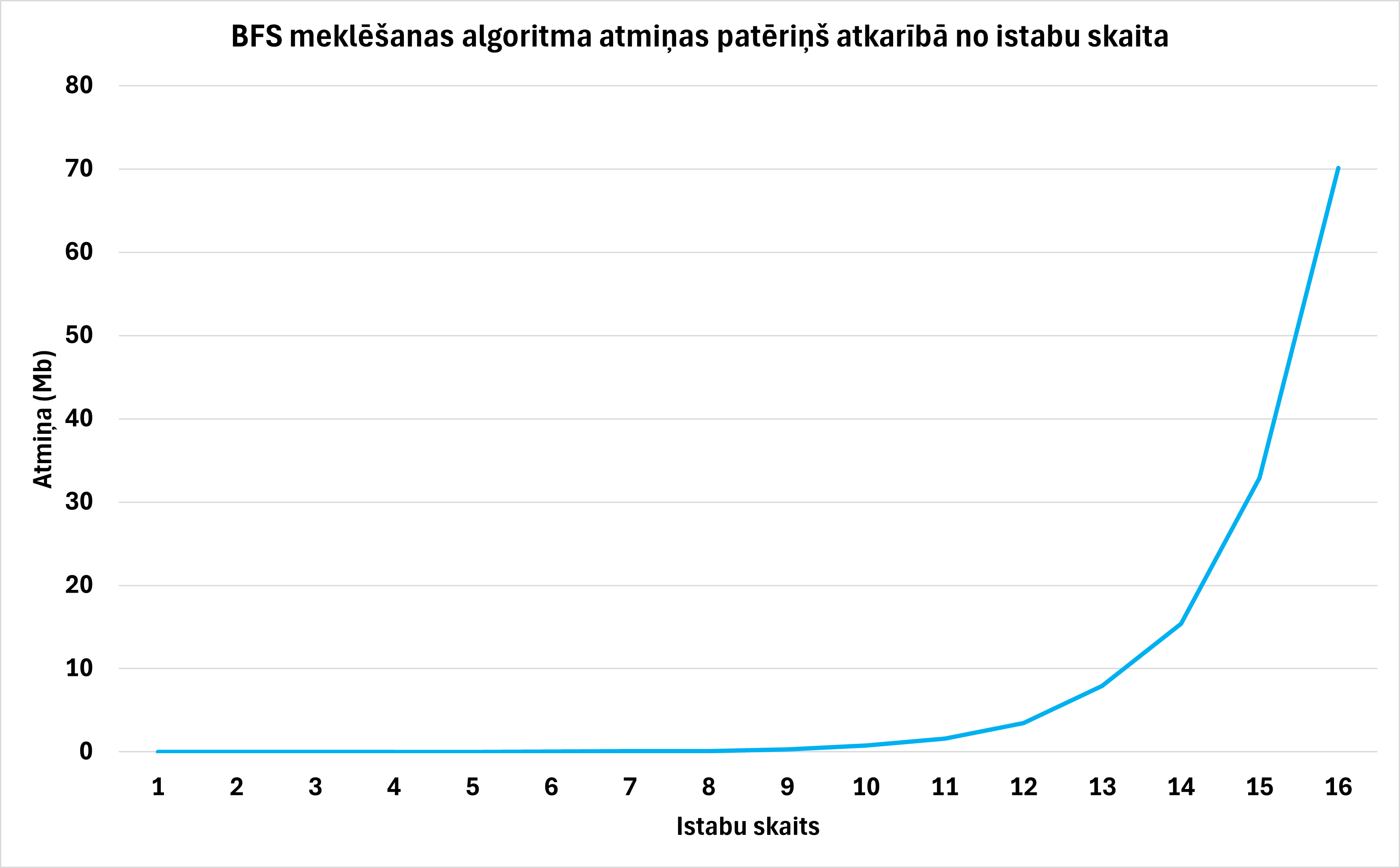
“**BFS meklēšanas algoritma laika patēriņš (milisekundēs) atkarībā no istabu skaita**”



Avots: autoru veidots grafiks (2024)

10. pielikums

“**BFS meklēšanas algoritma atmiņas patēriņš (megabaitos) atkarībā no istabu skaita**”



Avots: autoru veidots grafiks (2024)