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
Edvards 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 sarežģītāku problēmsituāciju risināšanas. Teorētiskajā daļā tika izskatīta mākslīgā intelekta definīcija un attīstība, algoritmu sarežģītība, kā arī divi klasiskās meklēšanas algoritmi. Pamatojoties uz mākslīgā intelekta pamatdefinīcijām, tika izstrādāts racionāli rīkojošs aģents – putekļsūcēja modelis, kura uzdevums ir iztīrīt netīrās istabas vidē. Praktiskajā daļā tika izmantota zema līmeņa programmēšana, lai izvairītos no augsta līmeņa abstrakcijām, kā arī nodrošinātu pilnu izpratni pār izveidotā koda funkcionalitāti un sarežģītību. Izstrādājot nepieciešamās datu struktūras, tika izveidots racionāli rīkojošs aģents ar diviem meklēšanas algoritmiem, kā arī tam attiecīgā vide. Lai izpētītu aģenta veiktspēju, tika salīdzināti divi dažādi grafa meklēšanas algoritmi pēc to resursu patēriņa konkrētas problēmsituācijas risināšanā. Veicot mērīšanu, tika secināts, ka pareiza algoritma izvēle var būtiski uzlabot aģenta meklēšanu, tomēr galvenā pamatproblēma izrādījās vides realizēšana, ielasīšana operatīvajā atmiņā un uzturēšana. Kaut arī algoritmu laika sarežģītība ir būtisks faktors, atmiņa izrādījās izšķirošs ierobežojums praktiska risinājuma iespējamībā, lineāri pieaugot aģenta vides sarežģītībai.

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

Annotation

The scientific research paper "Theoretical solution of artificial intelligence in relation to practical feasibility" was developed by Kristiāns Magons and Daniels Muļukins, students at Riga 64. Secondary School of class 12 INZ.

The goal of this research paper is to explore the fundamentals of the artificial intelligence field and implement an artificial intelligence model to test its limitations in performing increasingly complex problem solving scenarios. The theoretical part focuses on the definition and evolution of artificial intelligence, algorithm complexity, as well as two classic search algorithms. Following the base definitions of artificial intelligence, a rational agent in the form of a vacuum cleaner model with the task of cleaning all dirty rooms in a simulated environment was designed. Low-level programming was utilized to avoid high-level abstractions and ensure clear understanding and control over the functionality and complexity of the implementation. The paper describes the implementation of data structures, a rational agent with two search algorithms and the operating environment. To test the agent's performance, two graph search algorithms were compared in terms of time and space complexity. Measurements revealed that choosing an appropriate algorithm can significantly enhance the agent's ability to perform search. However, the most important problem proved to be the implementation, reading and maintenance of the environment. While time complexity is a major factor, the memory aspect turned out to be the crucial limitation when practical problem solving is considered in the environment with a linearly increasing complexity.

Key words: artificial intelligence, classical search, graph theory, data structure, low-level programming.

Saturs

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 praktisku priekšstatu par MI modelēšanas limitācijām.

Pētījuma mērķis:

Izveidot MI modeli, kā racionāli rīkojošu aģentu, lai novērtētu tā veiktspēju, risinot arvien komplicētākas problēmsituācijas, 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 implementācijas analīze veiktspējas novērtēšanai un potenciālajai uzlabošanai;
3. Dinamiskā testēšana – simulācijas programma tiek vairākkārt palaista un mērīts laika un atmiņas patēriņš;
4. Empīrisko rezultātu statistiskā analīze (T-testi).

1. Ieskats mākslīgajā intelektā

1.1. 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: Racionālā 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 dotā sākuma stāvokļa līdz gala stāvoklim, proti, nonākt tādā vides stāvoklī, kurā visas istabas ir tīras.

1.2. 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 izveidoja 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).

1.3. 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 modelēšana 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)

Jau vairākus gadus ir pieejami jaudīgi daudzkodolu procesori, kuri var ātri un efektīvi veikt daudzus uzdevumus. Šo attīstību iespējams arī novērot, apskatot Mūra likumu (Moore law) (Rupp, 2022). 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 efektīvu 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 reprezentē kā matricas, bet apstrādes rezultāts ir šo matricu reizinājums. Matricu reizinājumu var efektīvi 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*.

2. Algoritmu sarežģītība

2.1. 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 $O(n)$, kas cēlās no vācu matemātiķa Paula Gustava Heinriha Bahmaņa (Paul Gustav Heinrich Bachmann) pieraksta (Bachmann, 1894). $O(n)$ apraksta funkcijas uzvedību, kad arguments tiecas uz noteiktu vērtību vai bezgalību, kur O – augšanas kārtībā un n – argumenta lielums. (Black, 2019)

Tostarp datorzinātnēs n definē, nevis, kā argumenta lielumu, bet, kā argumentu skaitu vai ievaddatu skaitu (Mohr, 2014). Piemēram, runājot par algoritmu ātrumu, $O(n)$ nozīmēs, ka ar n ievaddatu skaitu algoritms izpildīs n soļus – algoritms izpildās lineāri, bet algoritms ar sarežģītību $O(1)$ izpildīsies konstanti, veicot 1 soli. $O(n)$ algoritmu aprakstīšanas metode ļauj viegli un saprotami salīdzināt algoritmus, jo, pat nezinot neko par algoritmiem, var secināt, ka algoritms ar sarežģītību $O(n)$ būs ātrāks par algoritmu $O(n^2)$, jo $n < n^2$.

Piebilde: Šajā projektā aprakstot, salīdzinot vai pētot algoritmu sarežģītību, tā tiks saīsināta no $O(n+1)$ uz $O(n)$ tā, ka viens papildus solis algoritma izpildes laiku ietekmēs ļoti minimāli. Īsinot nelielus konstanta soļus vai atmiņas skaitu.

2.2. 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 10^{150} 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 $3{,}8 \cdot 10^9$ gadus. (Rivest u.c., 1978)

3. Meklēšanas algoritmi

3.1. 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):

Attēls 1: A algoritma darbības grafiskais attēlojums, kā ceļi starp Latvijas pilsētām*

A* meklēšanas algoritma sarežģītība ir eksponenciāla $O(b^d)$, kur b – sazarojumu faktors jeb, cik sazarojumu ir katrai virsotnei, bet d – īsākais ceļa garums līdz galapunktam. Atmiņas sarežģītība ir visu apskatīto virsotņu skaits, kas sliktākajā gadījumā būs $O(|V|)$, kur $|V|$ – 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)

Pseidokods implementētajam A* meklēšanas algoritmam (skatīt 2. attēlu)

```
A*(environment):
        if current room is dirty:
                for each connection in current connections:
                        if connection - roomCount >= currentState:
                                clean that room
                                break
        while current state is not goal state:
                for each neighbour in current neighbours:
                        if neighbour room is dirty:
                                go to that room
                                for each connection in current connections:
                                        if connection - roomCount >= currentState:
                                                clean that room
                                                break

isRoomDirty(connectionCount, roomCount, stateToCheck):
        if connectionCount >= roomCount:
                return true
        else:
                set that room as cleaned
                return false
```

Attēls 2: Pseidokods autoru implementētajam A meklēšanas algoritmam*

3.2. 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 $O(|V| + |E|)$, kur $|V|$ – virsotņu skaits un $|E|$ – pāreju skaits. Šāda sarežģītība rodas, jo BFS meklēšanas algoritms katrai virsotnei izskata visas pārejas pēc kārtas. Atmiņas sarežģītība ir $O(|V|)$, jo BFS algoritms izmanto masīvu ar jau izskatītajām virsotnēm, kas rezultātā ir $|V|$ 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)

Pseidokods implementētajam BFS meklēšanas algoritmam (skatīt 3. attēlu)

```
BFS(environment):
        create map to save path
        create array of visited vertices with all values as false
        create queue for vertices to visit
        insert currentState to path
        set currentState as visited in visited vertices array
        push currentState to queue
        while queue is not empty:
                current = queue.pop
                if current is goal:
                        return
                for each neighbour in all current neighbours:
                        if neigbour is not visited:
                                push neighbour to queue
                                set neighbour as visited in visited vertices array
                                insert neighbour with current to path
```

Attēls 3: Pseidokods autoru implementētajam BFS meklēšanas algoritmam

3.3. Meklēšanas algoritmu nozīme šajā projektā

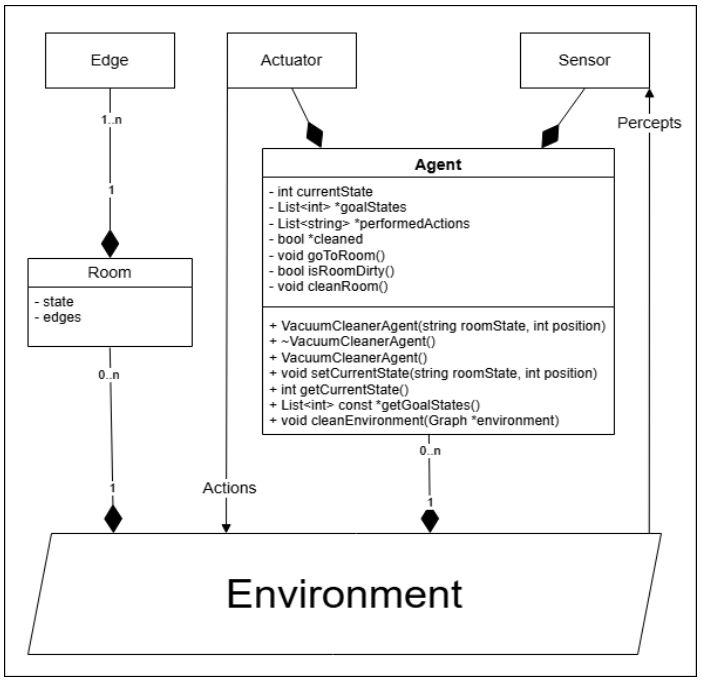
A* meklēšanas algoritms, kā viens no efektīvākajiem klasiskās meklēšanas algoritmiem (Russell, Norvig, 2009, 95-99), šajā pētījumā tiks izmantots, lai gūtu priekšstatu par racionāli rīkojoša aģenta veiktspējas limitācijām. BFS meklēšanas algoritms, kā primitīvāks meklēšanas algoritms, tiks izmantots, 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.

4. 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 4. attēlu):



Attēls 4: Vides relāciju grafiskais attēlojums

Grafa stāvokļu skaits:

$$2^n \cdot n$$

2^n, kur n – istabu skaits, reprezentē istabu stāvokļu skaitu, bet reizinājums ar n nozīmē, ka aģents var atrasties jebkurā no n istabām. Grafa pāreju skaits no tīras istabas ir $n-1$, jo ir iespējams tikai pāriet uz citām istabām, taču no netīras istabas būs n pāreju skaits, jo ir iespējams veikt arī tīrīšanu. Grafa pārejām pastāv $2^{n-1} \cdot n$ variācijas, aģentam atrodoties gan no tīras, gan netīras istabas. Tas ir tādēļ, ka, atrodoties tīrā vai netīrā istabā, pāri paliek $n-1$ istabu, kuru variāciju skaits ir 2^{n-1}, un ko jāreizina ar n, jo vienā no šīm n istabām atrodas aģents. Rezultātā tiek iegūts grafa pāreju skaits:

$$(n \cdot 2^{n-1} \cdot n) + (n \cdot 2^{n-1} \cdot (n-1)) = n \cdot 2^{n-1}(2n-1)$$

5. 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.

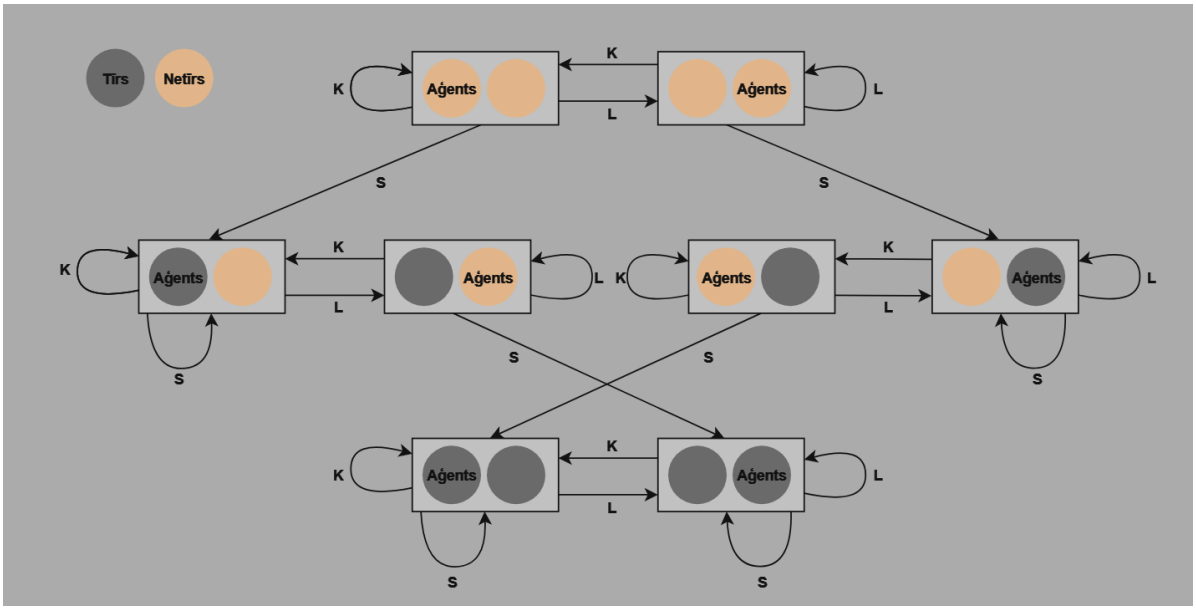
Heiristikas izveide vides un aģenta realizēšanā ir viena no svarīgākajām pamatproblēmām. Heiristika ir nepieciešama, lai piešķirtu aģentam racionālu risinājumu pareizu lēmumu izvēlē, proti, kuru zaru grafā izvēlēties, lai pēc iespējas ātrāk nonāktu mērķa stāvoklī. A* algoritma gadījumā heiristika balstās 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 $n - 1$ pārejas, kur n – istabu skaits. No netīras istabas putekļsūcēja aģents var gan aiziet uz citu istabu, gan iztīrīt esošo, tātad no šīs virsotnes būs n 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 n 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.

6. 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 5. attēlu):



Attēls 5: Putekļsūcēja aģenta darbības grafiskais attēlojums divām vidē piesārņotām istabām, kur K – iet pa kreisi, L – iet pa labi, S - sūkt

7. Datu struktūras

7.1. 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.

7.2. 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.

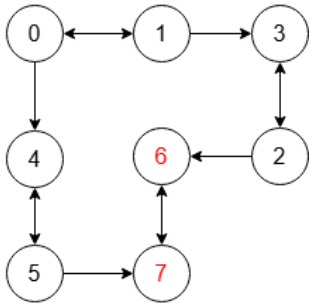
7.3. Grafs

Grafs (skatīt 3. pielikumu) ir nelineāra datu struktūra, kas sastāv no virsotnēm un pārejām, kur pāreja ir savienojums starp divām virsotnēm. Šajā projektā orientēts grafs reprezentē putekļsūcēja aģenta vidi, kur virsotne ir vides 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 (kaimiņu saraksta) no virsotņu pārejām izmantošana. Matricas pieeja konstantā laikā O(1) ļauj noskaidrot, vai eksistē ceļš starp divām virsotnēm, bet telpas ziņā izmantos $|V^2|$ atmiņas, kur $|V|$ – virsotņu skaits. Saistītā saraksta pieeja ļauj dinamiski pievienot un noņemt virsotnes, kā arī tā efektīvāk ļauj izmantot atmiņu $|V| + |E|$, kur $|V|$ – virsotņu skaits un $|E|$ – 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 n vai $n - 1$ pāreju, kur n – 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 6. attēlu):

P putekļsūcēja aģents * piesārņojums 6 7 mērķa stāvokļi

Virsotne	Istaba A	Istaba B	Pārejas
0	P *	*	1, 4
1	*	P *	0, 3
2	P *		3, 6
3	*	P	2
4	P	*	5
5		P *	4, 7
6	P		7
7		P	6



Attēls 6: Divu vidē piesārņotu istabu grafa reprezentācija

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 vides 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.

7.4. 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.

8. Implementācijas telpas un laika sarežģītības novērtēšana

8.1. 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 vides sākuma stāvoklis sarežģītība novērtēšanai tiek uzskatīts stāvoklis, kad visas istabas ir netīras. Vienā gadījumā aģenta stāvokļa pārejas 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 informēti prioritizēt iespējamās pārejas.

8.2. BFS meklēšanas algoritma novērtēšana

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

Laika sarežģītība pie n istabām:

$$O\left((2^n \cdot n) + \left(n \cdot 2^{n-1}(2n-1)\right)\right) = O(2^{n-1} \cdot 2n^2 + n) \equiv 2^n$$

Telpas sarežģītība pie n istabām:

$$O((2^n \cdot n) + (n \cdot 2^{n-1}(2n-1) + n) = O(2^n \cdot n + 2^n \cdot n^2 - 2^{n-1} \cdot n + n) \equiv 2^n$$

8.3. 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 $O(n^2)$, kur n – istabu skaits. Aģents, atrodoties katrā netīrajā istabā, veiks meklēšanu tīrīšanas pārejai, kas sliktākajā gadījumā tiks veikts n soļos. Tā kā aģents saglabā apmeklētās virsotnes, nākamās netīrās istabas meklēšana neietekmēs iepriekš iegūto algoritma sarežģītību $O(n^2)$, jo katra nākamā istaba būs netīra (pēc pieņēmuma, ka sākuma stāvoklis ir visas netīras istabas), bet jau apmeklētās istabas aģents neapskatīs. Savukārt atmiņas sarežģītība ir $O(n + n + 3n)$, kur pirmais n – apmeklētās istabas, otrais n – mērķa stāvokļi un $3n$ – 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 n istabām: $O(n^2)$

Telpas sarežģītība pie n istabām: $O(5n)$

9. 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 dotajā vides sākuma stāvoklī, bet mērīšanas beigu punktu tika noteikts brīdis, kad algoritms pabeidz meklēšanu, proti, atrod ceļu līdz beigu stāvoklim. Tika izmērīta arī grafa izveides 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., 6. un 7. pielikumu), kā arī izmantotā datora specifikācijas (skatīt 8. pielikumu).

10. Rezultāti

10.1. Implementētais kods un testu rezultāti

Autoru implementēto C++ kodu ar izveidotajām datu struktūrām un meklēšanas algoritmiem skatīt GitHub šeit (*github.com/Galaxy2268/VacuumCleanerAgent*). Koda atbilstība sākotnējām prasībām tika novērtēta, veicot funkcionālo testēšanu. Tika sagatavoti vides ievaddatu piemēri ar zināmiem sagaidāmajiem izvaddatiem, un programmatūra tika vairakkārt testēta, lai pārliecinātos par tās darbības atbilstību. Testēšana palīdzēja atklāt dažādas kļūdas, kā arī optimizēt implementāciju. Piemēram, kad tika konstatēts, ka BFS meklēšanas algoritms beidza darbu ar atmiņas kļūdas paziņojumu, meklējot risinājumu vidē ar 16 istabām, BFS meklēšanas algoritma masīvam tika pievienota norāde jeb pointeris. Šāda izmaiņa ļāva izmantot atmiņu efektīvāk un izpētīt BFS meklēšanas algoritma darbību pat pie vides ar 20 istabām.

10.2. Dinamiskā testēšana

Pēc aprakstītās algoritmu un grafa ielasīšanas veiktspējas novērtēšanas metodoloģijas (skatīt 9. nodaļu) iegūtie rezultāti par patērēto laiku (sekundēs) un atmiņu (megabaitos) tika apkopoti un analizēti (skatīt 1. un 2. tabulu) (skatīt 7., 8., 9., 10., 11., 12., 13. un 14. attēlu). Pilnā datu kopa ir pieejama pielikumā (skatīt 9. un 10. pielikumu).

Vidējie laika patēriņa mērījumi

Istabu skaits vidē	Grafa ielasīšana (s)	BFS meklēšanas algoritms (s)	A* meklēšanas algoritms (s)
2 istabas	0.000376	0.000011	0.000005
4 istabas	0.000540	0.000062	0.000009
6 istabas	0.001461	0.000646	0.000017
8 istabas	0.006381	0.003435	0.000023
10 istabas	0.036483	0.025856	0.000036
12 istabas	0.226042	0.116958	0.000052
14 istabas	1.253310	0.526984	0.000061
16 istabas	6.492430	2.732532	0.000077
18 istabas	33.342264	14.343724	0.000091
20 istabas	187.705400	511.590840	0.001904

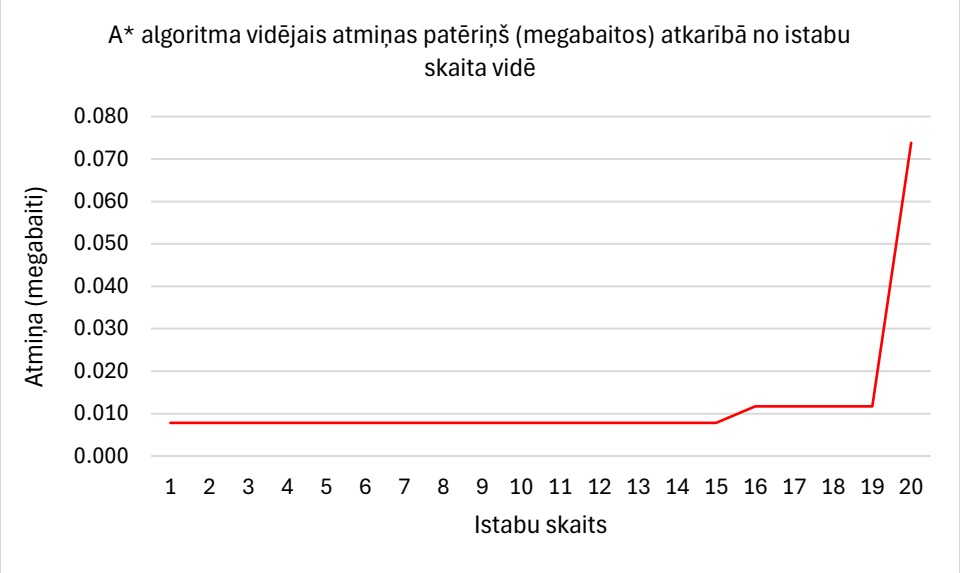
Tabula 1: Iegūtie vidējie rezultāti algoritmu un grafa ielasīšanas laika izpildījumam (sekundēs)

Vidējie atmiņas patēriņa mērījumi

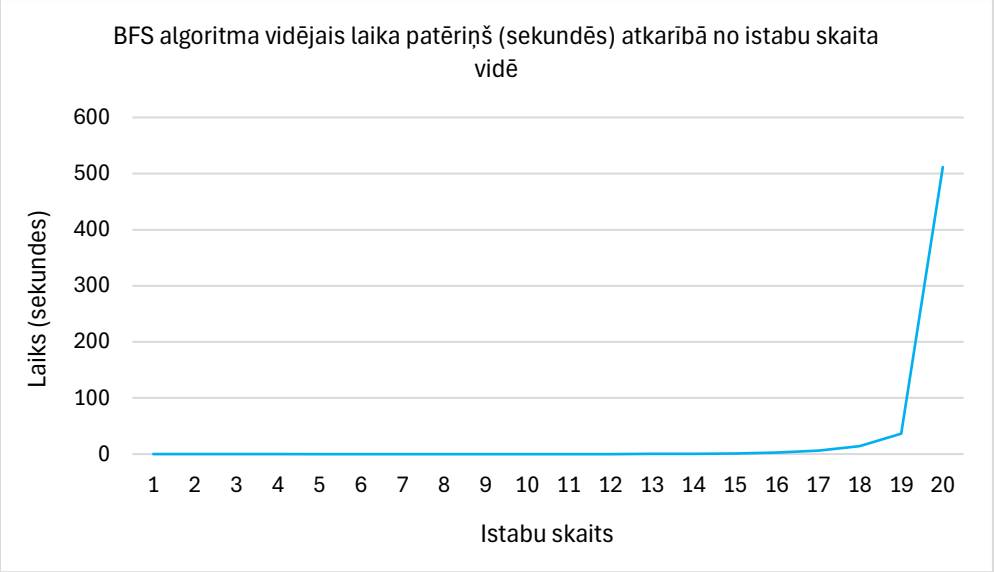
Istabu skaits vidē	Grafa ielasīšana (MB)	BFS meklēšanas algoritms (MB)	A* meklēšanas algoritms (MB)
2 istabas	4.191797	0.003906	0.007813
4 istabas	4.199997	0.003906	0.007813
6 istabas	4.290627	0.031250	0.007813
8 istabas	4.873439	0.085938	0.007813
10 istabas	7.945704	0.750000	0.007813
12 istabas	25.001950	3.474220	0.007813
14 istabas	115.255200	15.377730	0.007813
16 istabas	576.182000	70.111740	0.011719
18 istabas	2867.885000	315.648000	0.011719
20 istabas	10102.866000	1193.102100	0.073828

Tabula 2: Iegūtie vidējie rezultāti algoritmu un grafa ielasīšanas atmiņas izpildījumam (megabaitos)

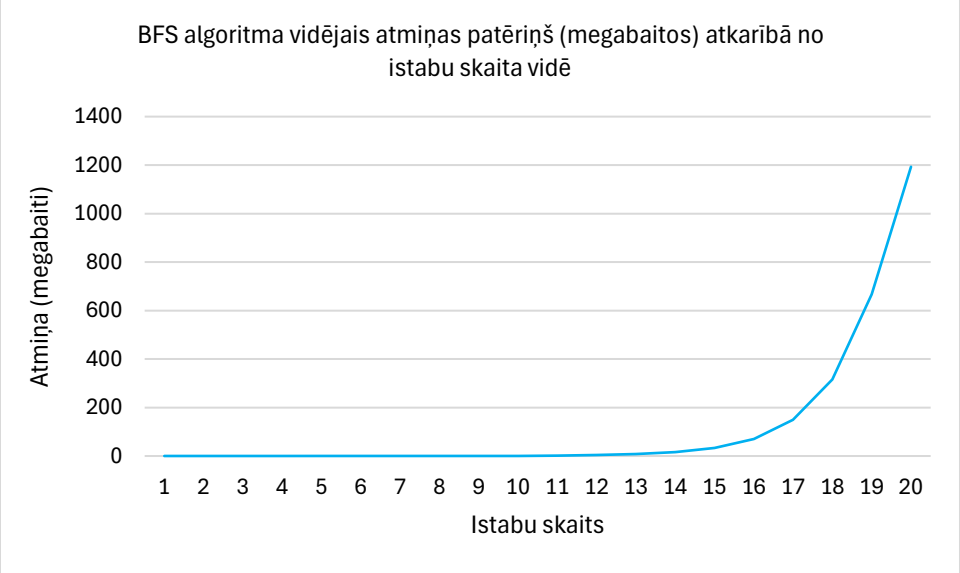
Attēls 7: A meklēšanas algoritma laika patēriņš (sekundēs) atkarībā no istabu skaita vidē*



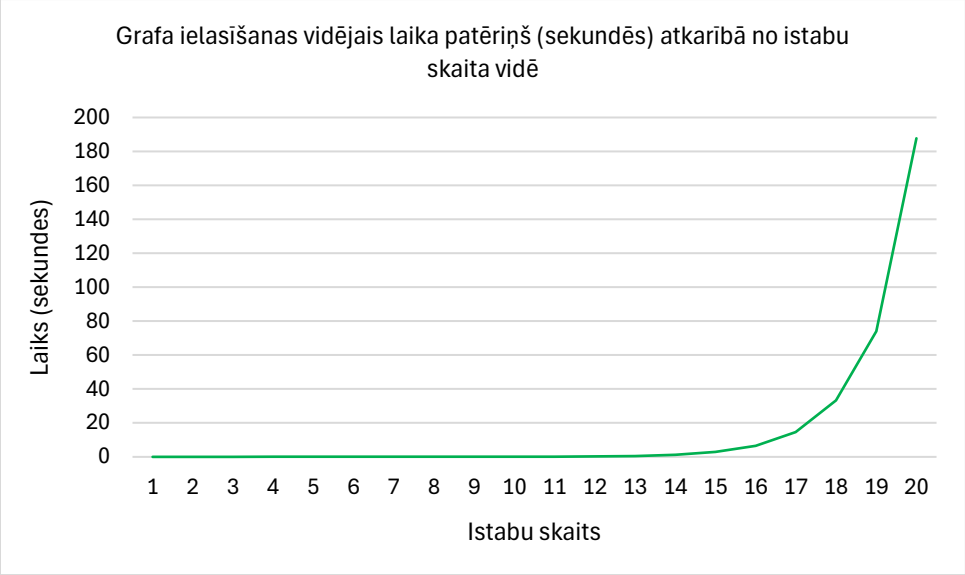
Attēls 8: A meklēšanas algoritma atmiņas patēriņš (megabaitos) atkarībā no istabu skaita vidē*



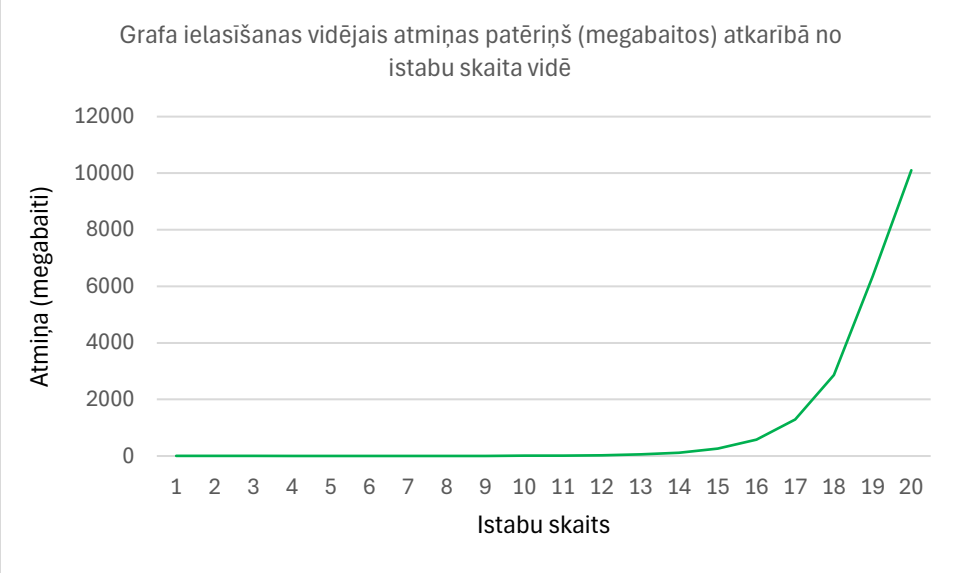
Attēls 9: BFS meklēšanas algoritma laika patēriņš (sekundēs) atkarībā no istabu skaita vidē



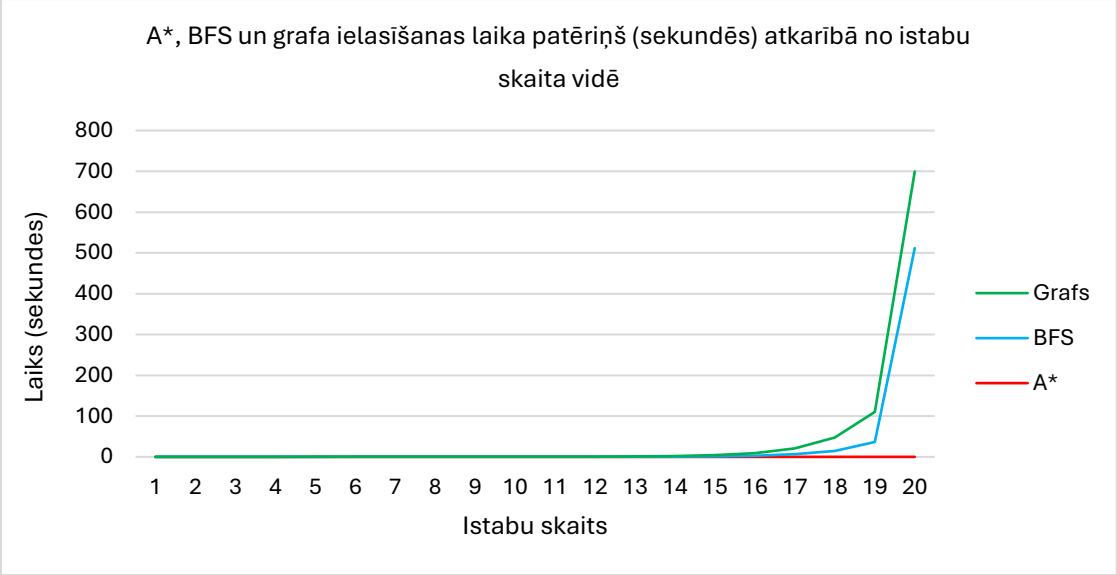
Attēls 10: BFS meklēšanas algoritma atmiņas patēriņš (megabaitos) atkarībā no istabu skaita vidē



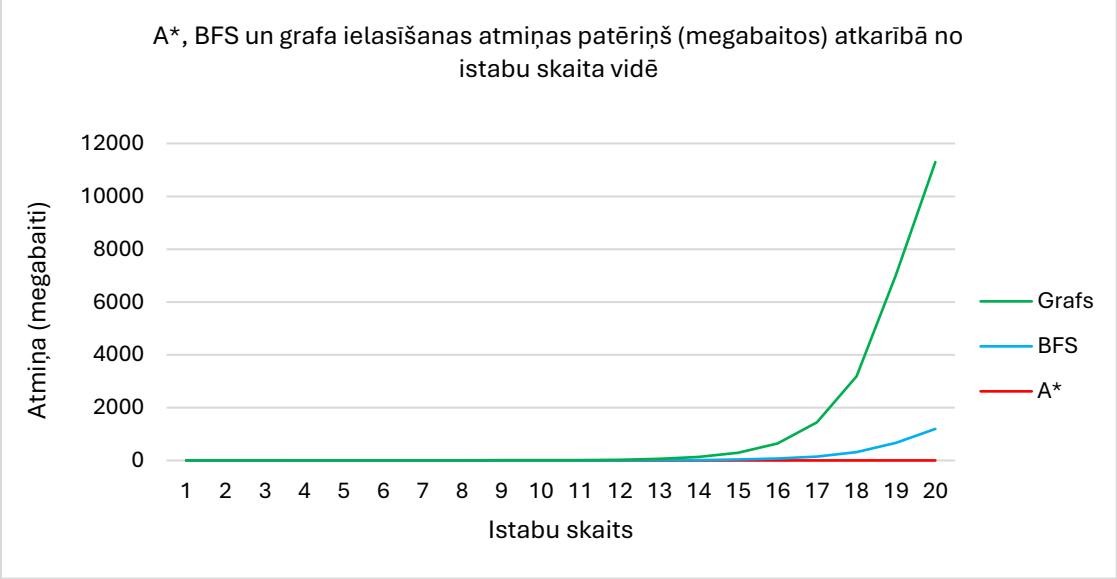
Attēls 11: Grafa ielasīšanas laika patēriņš (sekundēs) atkarībā no istabu skaita vidē



Attēls 12: Grafa ielasīšanas atmiņas patēriņš (megabaitos) atkarībā no istabu skaita vidē



Attēls 13: A un BFS algoritmu, un grafa ielasīšanas laika patēriņš (sekundēs) atkarībā no istabu skaita vidē*



Attēls 14: A un BFS algoritmu, un grafa ielasīšanas atmiņas patēriņš (megabaitos) atkarībā no istabu skaita vidē*

10.3. Statiskā novērtēšana

Pēc iegūtām teorētiskām laika un telpas sarežģītībām BFS un A* meklēšanas algoritmiem (skatīt 8. nodaļu) tika secināts, ka BFS meklēšanas algoritms vides atrisināšanu veic eksponenciāli gan telpas, gan laika izteiksmē. Savukārt A* meklēšanas algoritms, pateicoties heiristikai jeb zināšanām par konkrēto vidi, laika sarežģītību no eksponenciālās problēmas spēja to uzlabot polinomiālā jeb kvadrātiskā, un atmiņas sarežģītību no arī eksponenciālas uz lineāru.

Analizējot iegūtos praktisko mērījumu rezultātus, BFS meklēšanas algoritma grafiskā līkne gan pie laika, gan atmiņas patēriņa atkarībā no istabas skaita vidē (skatīt 9. un 10. attēlu) attīstās eksponenciāli, kas saskan ar izvirzītajām hipotētiskajām sarežģītībām. No otras puses A* meklēšanas algoritma grafiskā līkne pie laika patēriņa (skatīt 7. attēlu) līdz 19 istabu situācijai attīstās lineāri un tikai no 19. uz 20. istabu notiek relatīvi uzskatāms pieaugums, kas varētu liecināt par kvadrātisku laika sarežģītību. Lai iegūtu uzskatāmāku grafiku, būtu nepieciešams izpētīt A* meklēšanas algoritma darbību pie vairāk par 20 istabām, jo vides izveide ir eksponenciāla un to ievērot var tikai pie 20. istabas. Arī atmiņas patēriņš atkarībā no istabu skaita vidē (skatīt 8. attēlu) līdz 19 istabai ir lineārs, kas atbilst A* meklēšanas algoritma telpas sarežģītībai. Tomēr arī pie atmiņas patēriņa no 19 uz 20 istabu ir relatīvs pieaugums, kas arī liecina, ka būtu nepieciešami dati pie vairāk istabām, lai pārliecinātos par linearitāti.

10.4. Mērījumu kļūdas statistiskā analīze (T-testi)

Katra konfigurācija tika mērīta 25 reizes laika patēriņam un 10 reizes atmiņas patēriņam. Eksperimenta rezultātiem tika veikti absolūtās un relatīvās kļūdas aprēķini pēc Stjūdenta T testa formulas:

$$\triangle t = t \cdot \sqrt{\frac{\sum \triangle x_i^2}{n(n-1)}}$$

$\triangle t$ – absolūtā kļūda

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

$\sum \triangle x_i^2$ – gadījuma noviržu kvadrātu summa ($\triangle x$ – vidējā kvadrātiskā novirze, ko izmanto, lai noteiktu datu precizitāti jeb analizētu datu izkliedi attiecībā pret vidējo vērtību)

n – mērījumu skaits

Formula relatīvās kļūdas aprēķināšanai:

$$R = \frac{\triangle t}{x_{vid.}}$$

R – relatīvā kļūda

$x_{vid.}$ – mērījumu vidējā aritmētiskā vērtība

Aplūkojot absolūtās un relatīvās kļūdas gan grafa ielasīšanas, gan abu meklēšanas algoritmu laika patēriņa mērījumiem (skatīt 3. tabulu), pie mazāka istabu skaita ir raksturīga lielāka novirze no vidējās aritmētiskās vērtības. Par iemeslu šim varētu liecināt īsais laiks, kas nepieciešams gan grafa ielasīšanai, gan algoritmu meklēšanai. Piemēram, BFS algoritms 2 istabu vidē veic meklēšanu vidēji 0.000011 sekundēs, bet šāds laiks skaitliski ir pārāk niecīgs, lai programma atkārtoti izpildītos tik precīzi. Pie lielāka istabu skaita precizitāte uzlabojas, jo nepieciešamais laiks pieaug un ir relatīvi vieglāk pietuvoties šim apmēram. A* algoritmam ir lielākas novirzes, jo tā izpildījums ir salīdzinoši ātrs un nepieciešamais laiks pieaug minimāli.

Atšķirībā no laika, atmiņa tiek patērēta daudz precīzāk (skatīt 4. tabulu). Līdz 19 istabām vidē atmiņas patēriņš abiem meklēšanas algoritmiem ir konkrēts, ar nelieliem izņēmumiem BFS algoritmam. Pie grafa ielasīšanas relatīvā kļūda atmiņas patēriņam līdz 19 istabām saglabājas 0%, bet var novērot minimālas absolūtās kļūdas. Pie 20 istabām var novērot vienīgās būtiskās novirzes, jo 20 istabu vide ir apjoms, kas ievērojami izceļ vides eksponenciālo sarežģītību.

Salīdzinot laika un atmiņas faktorus, var secināt, ka nepieciešamā atmiņa ir precīza un nav tik atkarīgs faktors, kā laiks. Apskatot arī A* algoritma ātrumu salīdzinājumā ar BFS algoritmu (skatīt 5. tabulu), var praktiski novērot, cik reizes A* algoritms uzlabo tīrīšanu vidē jeb realizē eksponenciālu laika problēmu polinomiālā un eksponenciālu atmiņas problēmu lineārā.

Absolūtās un relatīvās kļūdas aprēķini laika patēriņa mērījumiem

	Grafa ielasīšana		BFS meklēšanas algoritms		A* meklēšanas algoritms	
Istabu skaits	Absolūtā kļūda	Relatīvā kļūda	Absolūtā kļūda	Relatīvā kļūda	Absolūtā kļūda	Relatīvā kļūda
1 Istaba	0.011994	4%	0.000860	19%	0.000572	17%
2 Istabas	0.028130	7%	0.001136	11%	0.000486	11%
3 Istabas	0.034984	8%	0.003819	16%	0.003554	27%
4 Istabas	0.065104	12%	0.006557	11%	0.001376	15%
5 Istabas	0.080004	9%	0.029072	13%	0.002364	18%
6 Istabas	0.076781	5%	0.075943	12%	0.001738	10%
7 Istabas	0.219438	8%	0.136353	11%	0.001928	10%
8 Istabas	0.275655	4%	0.158883	5%	0.002622	12%
9 Istabas	0.392345	3%	0.281238	4%	0.003483	12%
10 Istabas	1.202054	3%	0.932732	4%	0.005231	15%
11 Istabas	4.537020	5%	2.740798	5%	0.005653	12%
12 Istabas	5.255976	2%	3.000986	3%	0.009127	18%
13 Istabas	9.918063	2%	10.030947	3%	0.005988	9%
14 Istabas	17.672457	1%	6.574667	1%	0.002677	4%
15 Istabas	32.833134	1%	16.046228	1%	0.003068	5%
16 Istabas	76.266505	1%	35.407498	1%	0.004092	5%
17 Istabas	119.470937	1%	82.111928	1%	0.001911	2%
18 Istabas	165.825542	0%	266.444315	2%	0.002502	3%
19 Istabas	512.613316	1%	4288.496614	12%	0.187835	79%
20 Istabas	7636.049359	4%	46133.126810	9%	0.451893	24%

Tabula 3: Absolūtās un relatīvās kļūdas aprēķini laika patēriņa mērījumiem

Absolūtās un relatīvās kļūdas aprēķini atmiņas patēriņa mērījumiem

	Grafa ielasīšana		BFS meklēšanas algoritms		A* meklēšanas algoritms	
Istabu skaits	Absolūtā kļūda	Relatīvā kļūda	Absolūtā kļūda	Relatīvā kļūda	Absolūtā kļūda	Relatīvā kļūda
1 Istaba	0.003178	0%	0.000000	0%	0.000000	0%
2 Istabas	0.003028	0%	0.000000	0%	0.000000	0%
3 Istabas	0.003934	0%	0.000000	0%	0.000000	0%
4 Istabas	0.003849	0%	0.000000	0%	0.000000	0%
5 Istabas	0.002173	0%	0.000000	0%	0.000000	0%
6 Istabas	0.002958	0%	0.000000	0%	0.000000	0%
7 Istabas	0.003243	0%	0.000000	0%	0.000000	0%
8 Istabas	0.004143	0%	0.000000	0%	0.000000	0%
9 Istabas	0.002171	0%	0.000000	0%	0.000000	0%
10 Istabas	0.002409	0%	0.000000	0%	0.000000	0%
11 Istabas	0.002554	0%	0.000000	0%	0.000000	0%
12 Istabas	0.004147	0%	0.001419	0%	0.000000	0%
13 Istabas	0.002406	0%	0.001159	0%	0.000000	0%
14 Istabas	0.002674	0%	0.001327	0%	0.000000	0%
15 Istabas	0.002911	0%	0.001419	0%	0.000000	0%
16 Istabas	0.003044	0%	0.001419	0%	0.000000	0%
17 Istabas	0.003403	0%	0.000000	0%	0.000000	0%
18 Istabas	0.003713	0%	0.000000	0%	0.000000	0%
19 Istabas	0.005753	0%	0.000000	0%	0.000000	0%
20 Istabas	439.762346	4%	152.740188	13%	0.023974	32%

Tabula 4: Absolūtās un relatīvās kļūdas aprēķini atmiņas patēriņa mērījumiem

A* meklēšanas algoritma ātrdarbība attiecībā pret BFS meklēšanas algoritmu

Istabu skaits vidē	Laika patēriņš (x)	Atmiņas patēriņš (x)
2 istabas	2.3	0.5
4 istabas	6.5	0.5
6 istabas	38.8	4.0
8 istabas	151.0	11.0
10 istabas	722.7	96.0
12 istabas	2246.4	444.7
14 istabas	8572.2	1968.3
16 istabas	35368.0	5982.8
18 istabas	157450.3	26935.2
20 istabas	268627.8	16160.5

Tabula 5: A meklēšanas algoritma ātrdarbība un atmiņas izmantojums attiecībā pret BFS meklēšanas algoritmu*

Secinājumi

Izpētot literatūru, tika secināts, ka mākslīgais intelekts ietver plašu pētāmo problēmu un iespējamo risinājumu spektru. Kā jau minēts, jebkurā mākslīgā intelekta diskusijā, kritiski svarīga ir kopīga izpratne par definīciju. Šajā darbā ar mākslīgo intelektu tiek saprasta racionāla aģenta, proti, putekļu sūcēja modelēšana. Aģents darbojas vidē, kas šajā gadījumā ir deterministiska, statiska, pārredzama, pārtraukta un galīga. Aģenta uzdevums ir izmainīt sākotnējo vides stāvokli tā, ka visas vides istabas ir tīras. Tā lietderība tiek maksimizēta, samazinot patērēto enerģiju (sensoru un aktuatoru izmantošana). Praktiski šāds abstrakts racionāla aģenta modelis tika realizēts, izmantojot orientēta grafa datu struktūru, kā vides reprezentāciju, un klasiskās meklēšanas pamatalgoritmus, kā aģenta vides stāvokļu maiņas funkciju.

Salīdzinot aģenta meklēšanas algoritmus (BFS un A*), tika novērota nozīmīga atšķirība patērētā laika un atmiņas izteiksmē. Pēc iegūtām teorētiskām laika un telpas sarežģītību formulām un to salīdzināšanas praksē, tika secināts, ka aģents ar informētu stāvokļu pārejas algoritma palīdzību vispārīgi eksponenciālas komplicitātes problēmu spēj risināt polinomiālā laikā un eksponenciālo telpas sarežģītību realizē lineāri. Tātad pareiza meklēšanas algoritma izvēle var ievērojami uzlabot aģenta veiktspēju.

Pēc iegūtajiem rezultātiem izvirzīto hipotēzi par algoritmu sarežģītību un iespējamiem efektivitātes uzlabojumiem izdevās apstiprināt, kā arī tika novērotas un aprakstītas sākotnēji pieņemtās grafu meklēšanas algoritmu limitācijas praktiskā racionālu aģentu modelēšanā.

Veicot implementācijas sistemātisku testēšanu, tika secināts, ka sākotnēji realizētā BFS meklēšanas algoritma versijai sasniedzot pieejamās operatīvās atmiņas limitu pie 16 istabu vides, to bija iespējams uzlabot. BFS meklēšanas algoritma apmeklēto virsotņu masīvam pievienojot norādi jeb pointeri, bija iespējams uzlabot atmiņas lietojumu un izpētīt tā darbību pat 20 istabu situācijā. Izveidotais A* meklēšanas algoritms vairākkārt pārspēja BFS algoritma efektivitāti meklēšanā, tomēr maksa par algoritma ātrdarbības uzlabošanu, uzdodot heiristiku, saistās ar ievaddatu sakārtošanas nepieciešamību. Ja vides fails nebūtu mākslīgi izveidots, bet būtu nejauši sakārtots bez sakarībām, secinām, ka izveidoto A* algoritmu pielietot neizdotos. Tādā gadījumā heiristikas izveide būtu sarežģītāka problēma, nekā grafa iziešana ar primitīvo BFS meklēšanas algoritmu.

Lai gan algoritma izvēle var uzlabot aģenta darbību, galvenā pamatproblēma izrādījās vides faila izveidošana, kā arī paša grafa, kas reprezentē vidi, ielasīšana operatīvajā atmiņā un uzturēšana. Vien 20 istabu vides saturēšanai tika izmantota visa datoram pieejamā atmiņa, kas ir aptuveni 10GB operatīvās atmiņas. Lai izveidotu 20 istabu vides failu, bija nepieciešams izmantot citu datoru, un šī faila izveidei bija nepieciešami 24GB operatīvās atmiņas. Tāpēc ir svarīgi 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. Kaut arī A* meklēšanas algoritms ar pašizveidotu heiristiku atmiņu izmanto lineāri, vides telpas sarežģītība joprojām ir eksponenciāla.

Veicot rezultātu statistisko analīzi, vēlreiz pārliecinājāmies, ka vides eksponenciālā telpas sarežģītība ir galvenais ierobežojums, praktiskos klasiskās meklēšanas pielietojumos. Meklēšanas algoritmu un grafa ielasīšanas mērījumu precizitāte gan laika, gan telpas sarežģītības izteiksmē uzlabojās, pieaugot istabu skaitam vidē. Tomēr pie 20 istabu situācijas notiek straujš pieaugums patērētajiem resursiem un ir manāmas novirzes no vidējās vērtības. Tāpēc secinām, ka piedāvātais

racionālā aģenta modelis nav praktisks pie risinājuma meklēšanas vidē, kur istabu skaits ir lielāks par 19. Šī pētījuma uzlabošanai vērtīgi būtu uzlabot vides failu izveidošanu, lai iegūtu plašāku pētāmo datu kopu. Tostarp ieviest dažādību putekļsūcēja aģenta vidē vai 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 (*github.com/Galaxy2268/VacuumCleanerAgent*)

2. pielikums

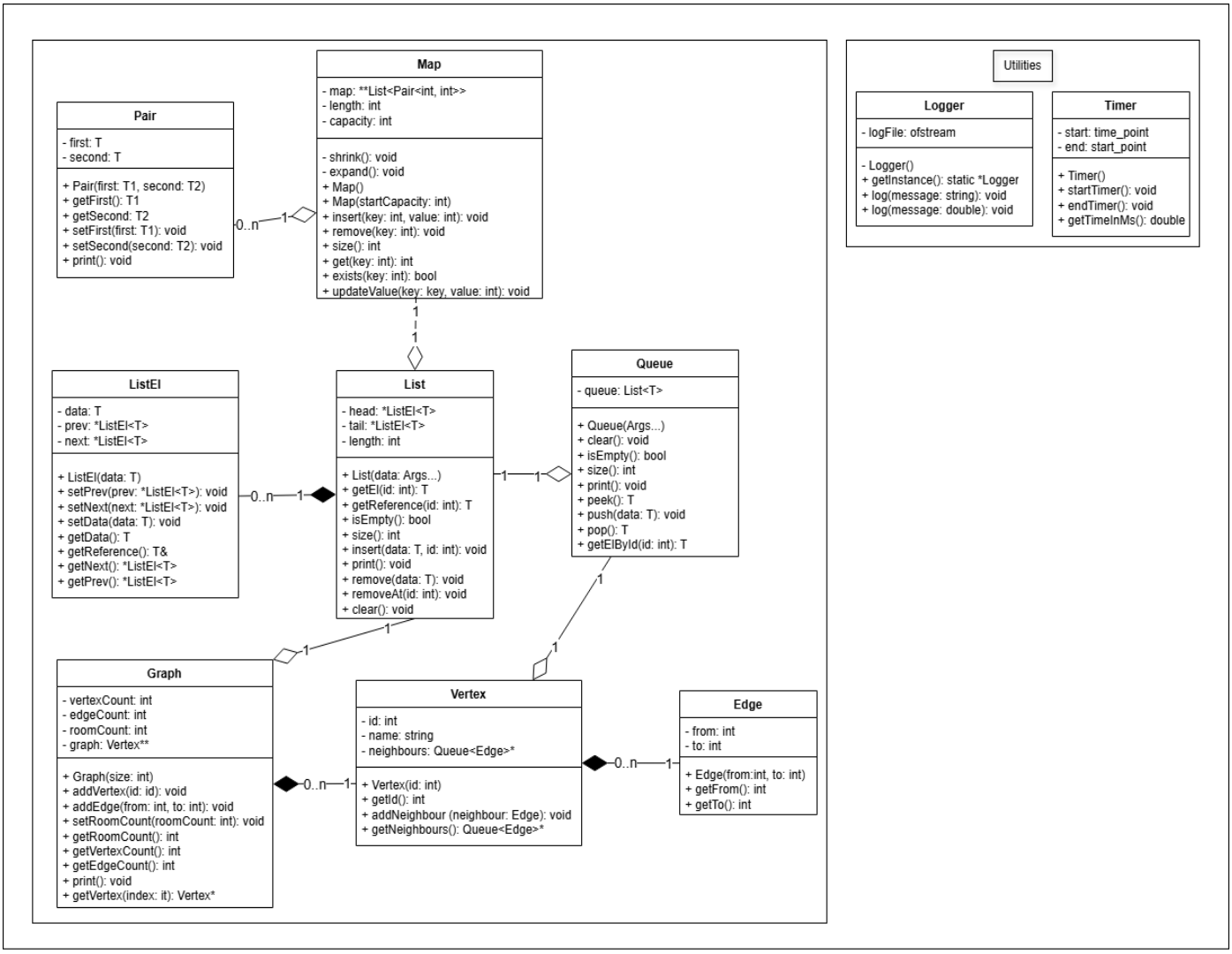
“**Divu vidē piesārņotu istabu faila reprezentācija**”

```
2 8 12

0 4
0 1
1 3
1 0
2 6
2 3
3 2
4 5
5 7
5 4
6 7
7 6
```

3. pielikums

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



"Kods 19 vidē piesārņotu istabu laika patēriņa mērīšanai"

```
#include <iostream>
#include "Environment/Environment.h"
#include "Agent/VacuumCleanerAgent.h"
#include "Graph/GraphBuilder/GraphBuilder.h"
#include "util/Timer/Timer.h"
#include "util/Logger/Logger.h"

int main() {
    //call kotlin code, graph generator
    //system(R"(java -Xmx6g -jar GraphGenerator\out\artifacts\GraphGenerator_jar\GraphGenerator.jar 1 room_1.txt)");
    for(int i = 0; i < 25; i++){
        Timer timer;
        GraphBuilder graphBuilder;

        //choose file to read
        Graph* graph = graphBuilder
                .readFile("Data/room_14.txt")
                .build();


        VacuumCleanerAgent *agent = new VacuumCleanerAgent("11111111111111", 0);
        Environment environment(graph, agent);
        timer.startTimer();
        environment.turnAgentOnAStar();
        //or environment.turnAgentOnBFS();
        timer.stopTimer();
        Logger::getInstance()->log(timer.getTimeInMs());
    }
    return 0;
}
```

"Kods 19 vidē piesārņotu istabu atmiņas patēriņa mērīšanai"

```
#include <iostream>
#include "Environment/Environment.h"
#include "Agent/VacuumCleanerAgent.h"
#include "Graph/GraphBuilder/GraphBuilder.h"
#include "util/Logger/Logger.h"
#include <windows.h>
#include <psapi.h>

int main() {
    //call kotlin code, graph generator
    //system(R"(java -Xmx6g -jar GraphGenerator\out\artifacts\GraphGenerator_jar\GraphGenerator.jar 1 room_1.txt)");
        GraphBuilder graphBuilder;

        //choose file to read
        Graph *graph = graphBuilder
                .readFile("Data/room_20.txt")
                .build();

        double graphSize;
        PROCESS_MEMORY_COUNTERS pmc;
        if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc))) {
            graphSize = static_cast<double>(pmc.WorkingSetSize) / (1024 * 1024);
        }
        VacuumCleanerAgent *agent = new VacuumCleanerAgent("1111111111111111111", 0);
        Environment environment(graph, agent);
        environment.turnAgentOnAStar();
        //or environment.turnAgentOnBFS();
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc))) {
        double graphAndAlgorithmSize = static_cast<double>(pmc.WorkingSetSize) / (1024 * 1024);
        Logger::getInstance()->log(graphAndAlgorithmSize - graphSize);
    }
    return 0;
}
```

6. pielikums

"Kods grafa ielasīšanas laika patēriņa mērīšanai"

```
#include <iostream>
#include "Environment/Environment.h"
#include "Agent/VacuumCleanerAgent.h"
#include "Graph/GraphBuilder/GraphBuilder.h"
#include "util/Timer/Timer.h"
#include "util/Logger/Logger.h"

int main() {
    //call kotlin code, graph generator
    //system(R"(java -Xmx6g -jar GraphGenerator\out\artifacts\GraphGenerator_jar\GraphGenerator.jar 1 room_1.txt)");
    Timer timer;
    timer.startTimer();
    GraphBuilder graphBuilder;

    //choose file to read
    Graph* graph = graphBuilder
        .readFile("Data/room_20.txt")
        .build();
    timer.stopTimer();
    Logger::getInstance()->log(timer.getTimeInMs());
}
```

"Kods grafa ielasīšanas atmiņas patēriņa mērīšanai"

```
#include <iostream>
#include "Environment/Environment.h"
#include "Agent/VacuumCleanerAgent.h"
#include "Graph/GraphBuilder/GraphBuilder.h"
#include "util/Timer/Timer.h"
#include "util/Logger/Logger.h"
#include <windows.h>
#include <psapi.h>

int main() {
    //call kotlin code, graph generator
    //system(R"(java -Xmx6g -jar GraphGenerator\out\artifacts\GraphGenerator_jar\GraphGenerator.jar 1 room_1.txt)");
        GraphBuilder graphBuilder;

        //choose file to read
        Graph *graph = graphBuilder
                .readFile("Data/room_20.txt")
                .build();
        PROCESS_MEMORY_COUNTERS pmc;

    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc))) {
        double graphSize = static_cast<double>(pmc.WorkingSetSize) / (1024 * 1024);
        Logger::getInstance()->log(graphSize);
    }
    return 0;
}
```

8. pielikums

“Pētījumā izmantotā datora specifikāciju apraksts”

```
Procesors:
        AMD Ryzen 5 Mobile 3550H (4 kodoli, 2,1 GHz)

Videokarte:
        NVIDIA GeForce GTX 1050 (GDDR 5, 3GB, 1440MHz)

Operatīvā atmiņa:
        16GB DDR4 (programmas aktivizēšanas brīdī ieejamā atmiņa ir aptuveni 10GB)

Operētājsistēma:
        Windows 10 Home (22H2 versija)

Sistēmas tips:
        64 bitu operētājsistēma, x64 bāzes procesors
```

9. pielikums

“**Iegūtie vidējie rezultāti algoritmu un grafa ielasīšanas laika izpildījumam (sekundēs)**”

Istabu skaits	Grafa ielasīšana	BFS meklēšanas algoritms	A* meklēšanas algoritms
1 istaba	0.000322	0.000004	0.000003
2 istabas	0.000376	0.000011	0.000005
3 istabas	0.000433	0.000023	0.000013
4 istabas	0.000540	0.000062	0.000009
5 istabas	0.000875	0.000220	0.000013
6 istabas	0.001461	0.000646	0.000017
7 istabas	0.002775	0.001284	0.000020
8 istabas	0.006381	0.003435	0.000023
9 istabas	0.015130	0.007391	0.000028
10 istabas	0.036483	0.025856	0.000036
11 istabas	0.094546	0.057328	0.000048
12 istabas	0.226042	0.116958	0.000052
13 istabas	0.553311	0.353322	0.000063
14 istabas	1.253310	0.526984	0.000061
15 istabas	2.855866	1.194329	0.000065
16 istabas	6.492430	2.732532	0.000077
17 istabas	14.688496	6.347452	0.000083
18 istabas	33.342264	14.343724	0.000091
19 istabas	73.928172	36.149916	0.000237
20 istabas	187.705400	511.590840	0.001904

10. pielikums

"Iegūtie vidējie rezultāti algoritmu un grafa ielasīšanas atmiņas izpildījumam (megabaitos)"

Istabu skaits	Grafa ielasīšana	BFS meklēšanas algoritms	A* meklēšanas algoritms
1 istaba	4.191407	0.00390625	0.0078125
2 istabas	4.191797	0.00390625	0.0078125
3 istabas	4.192967	0.00390625	0.0078125
4 istabas	4.199997	0.00390625	0.0078125
5 istabas	4.229688	0.0117188	0.0078125
6 istabas	4.290627	0.03125	0.0078125
7 istabas	4.435546	0.105469	0.0078125
8 istabas	4.873439	0.0859375	0.0078125
9 istabas	5.766404	0.304688	0.0078125
10 istabas	7.945704	0.75	0.0078125
11 istabas	13.09299	1.58594	0.0078125
12 istabas	25.00195	3.47422	0.0078125
13 istabas	52.49259	7.95	0.0078125
14 istabas	115.2552	15.37773	0.0078125
15 istabas	257.2452	32.90464	0.0078125
16 istabas	576.182	70.11174	0.0117188
17 istabas	1288.043	149.082	0.0117188
18 istabas	2867.885	315.648	0.0117188
19 istabas	6354.72	666.234	0.0117188
20 istabas	10102.866	1193.1021	0.07382808