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**:

Vadošais arhitekts, SAP SE

Krists Magons

**Darba konsultants**:

Programmēšanas skolotājs, Rīgas 64. vidusskola

Eduards Bukovskis

Rīga, 2025

**Ievads**

Pēdējo gadu laikā ir strauji attīstījušās datortehnoloģijas [1], kas līdzās sekmējušas arī mākslīgā intelekta (turpmāk tekstā – MI) attīstību, jo radās iespējas efektīvāk glabāt daudz datus mākoņos [2, pp. 27-28] [3], un jaudīgās videokartes [4] ļāva ātrāk veikt skaitļošanu un apstrādāt vairāk datus. Šis sniedza iespējas realizēt tādas tehnoloģijas kā TensorFlow, kas vienlaikus paralēli izmanto vairākus procesorus, kas būtiski paātrina matemātiskus aprēķinus [5]. Šādas skaitļošanas platformas brīvi pieejamas, pateicoties daudzkodolu videokaršu attīstībai. Pagaidām MI nav stingri definēts, jo atšķiras izpratnes par to. Vēsturiski ir nonākts pie četrām pamata definīcijām, kā izprast MI jēdzienu – domāšana vai rīkošanās cilvēciski, domāšana vai rīkošanās racionāli [2, p. 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 mēs realizēsim savu racionālo aģentu, lai saprastu MI limitācijas.

**Pētījuma mērķis**:

Izveidot mākslīgā intelekta modeli kā racionāli rīkojošu aģentu, lai izpētītu problēmjautājumu pamatus un gūtu praktisku priekšstatu par mākslīgā intelekta metožu, šajā tvērumā klasiskās meklēšanas uzdevumu limitācijām.

**Darba uzdevumi**:

1. Valodā C++ realizēt zema līmeņa datu struktūras un pamatalgoritmus (saistītais saraksts, prioritāšu rinda, steks, grafs), kā arī iepazīties ar teksta failu apstrādi;
2. Izmantojot realizētās datu struktūras un OOP (Objektorientētā programmēšana) paradigmas, realizēt aģenta un vides implementāciju un darbības simulāciju;
3. Realizēt aģenta stāvokļa pārejas funkciju, izmantojot A\* vai citus meklēšanas algoritmus;
4. Novērtēt aģenta veiktspēju atkarībā no vides iestatījumiem un noskaidrot aģenta potenciālās vajadzības pēc atmiņas daudzuma un izšķirtspējām;
5. Izvērtēt MI potenciālās iespējas nākotnē.

**Hipotēze**:

Jo sarežģītāka vide, jo aģentam, ar teorētiski iespējamu risinājumu, to realizēt kļūs arvien grūtāk līdz aģentam tas praktiski nemaz neizdosies ierobežoto resursu dēļ.

**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- tests).

**1. Ieskats mākslīgajā intelektā**

**1.1 Kas ir mākslīgais intelekts?**

Cilvēki jau sen ir mēģinājuši saprast, – kā cilvēks spēj uztvert, saprast, paredzēt, un ietekmēt pasauli, kas ir daudz plašāka par viņu. MI sfēra tiecas vēl tālāk – tā nav tikai mēģinājums izprast, bet arī atveidot inteliģenci [2, p. 1]. Vadoties pēc Stjuarta Rasela un Pītera Norviga mācību grāmatas “Artificial Intelligence: A Modern Approach”, MI nav vienotas definīcijas. Izmantojot astoņas dažādas MI definīcijas [2, p. 2], autori tās ir apvienojuši četrās galvenās kategorijās:

**Cilvēciska rīcība**: 1950. gadā Alans Tjūrings izgudroja 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. [2, pp. 2-3].

Tjūringa tests mūsdienās joprojām ir aktuāls, bet viedokļi par tā atbilstību MI definēšanā ir mainījušies. Tjūringa tests paredz, ka dators spēj simulēt cilvēcisku rīcību, bet ne gluži rīkoties cilvēciski [6]. 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 [7].

**Cilvēciska domāšana**: Šī MI pieeja cieši saistās ar kognitīvo zinātni, jo, lai konstruētu cilvēciski domājošu MI moduli, 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. [2, p. 3]

**Racionāla domāšana**: Racionāla domāšana saista loģikas jomu un izskata jautājumus par pareizu un sistematizētu domāšanu. Šī 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.” [2, p. 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. [2, pp. 4-5]

Šajā pētījumā MI pieeja būs putekļsūcēja modelis kā racionāli rīkojošs aģents. Tiks izstrādāts putekļsūcēja aģents, kurš pārvietosies pa vidi, kas sastāv no vairākām istabām, un iztīrīs visas istabas. Tas spēs uztvert, vai istabā, kurā tas atrodas, ir piesārņojums. Tad attiecīgi tas atkritumus sasūks vai pārvietosies uz nākamo istabu, ja piesārņojums netiks uztverts. Kā jau rīkojoties racionāli, putekļsūcēja mērķis ir visefektīvāk nokļūt no sākuma stāvokļa līdz rezultāta stāvoklim, proti, nonākt stāvoklī, kad visas istabas ir tīras.

**1.2 Izcelsme**

MI ir viena no jaunākajām zinātnes un inženierijas nozarēm [2, p. 1]. Runājot par MI tēmu, vārdam “nozare” ir jāliek īpašs uzsvars, jo nereti MI tiek pārprasts par atsevišķu, patstāvīgu zinātni. MI saista vairākas zinātnes, piemēram, matemātika, psiholoģija, datortehnika un pat filozofija [2, pp. 5-16]. Ievērojamie rezultāti MI jomā gūti vien pēdējo gadu laikā (par MI uzplaukumu lasīt nodaļā 1.3 Uzplaukums), 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 [2, pp. 16-17].

Ietekmīgs veikums 1956. gada vasarā bija Allena Nevela un Herberta Saimona izveidotā spriešanas programma “Loģikas teorētiķis”, kas spēja domāt neskaitliski [2, pp. 17-18]. Vēlāk viņi izveidoja programmu “Vispārējais problēmu risinātājs”, kas bija programma, spējīga atdarināt cilvēcisku domāšanu. Savukārt 1958. gads ir nozīmīgs ar Džona Makartija izveidoto teorētisko aprakstu programmai “Padomu ņēmējs”, kas hipotētiski spēja izmantot vispārējas pasaules zināšanas, lai meklētu problēmu risinājumus, un ko uzskatīja par pirmo pilnīgo MI sistēmu [2, pp. 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 [2, pp. 22-27], MI joma uz ilgāku laiku stagnēja, jo viss atdūrās pret datoru nespēju veikt grūtākus vai apjomīgākus uzdevumus [2, pp. 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 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 mainījusies no mehāniskas zināšanu iekodēšanas uz datora pašmācību pie dotiem datiem [2, pp. 27-28].

Lai apstrādātu datus un trenētu MI modeļus, ir nepieciešams veids, kā šos lielos daudzums datu efektīvi glabāt. 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 [8], mums ir pieeja jaudīgiem daudzkodolu procesoriem, 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 [4], jo, atšķirībā no procesoriem, tām ir daudz vairāk atsevišķu kodolu, lai veiktu 3D kalkulācijas. Piemēram, grafiskais procesors “GeForce RTX 3070” satur 5888 kodolu, kuri paralēli veic savas funkcijas [9]. Šī grafisko procesoru īpatnība, veikt vienlaikus paralēlus darbus, padara tās par ļoti efektīvu rīku paralēlajā datu apstrādē [10]. 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. Šo masīvu reizinājumu var veikt paralēli ar videokaršu palīdzību, kas būtiski samazina laiku masīvu sareizināšanai [4].

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 tiek iekļauts vispopulārāko vietņu sarakstā pasaulē [11]. 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 [12] [13]. 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 nāk ar savām limitācijām, jo šīs tehnoloģijas tiek trenētas, balstoties uz datiem, kuri var gadīties nepatiesi, nepilnīgi vai nepietiekami daudz. Rezultātā LVM tehnoloģiju sniegtās atbildes var būt maldīgas [13].

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 “Paul Bachmann” [11] pieraksta. *O(n)* apraksta funkcijas uzvedību, kad arguments tiecas uz noteiktu vērtību vai bezgalību, kur O – nozīmēja augšanas kārtību, bet n – argumenta lielumu [12].

Tostarp datorzinātnēs n definē, nevis, kā argumenta lielumu, bet kā argumentu skaitu vai ievaddatu skaitu [13]. 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. Šī algoritmu aprakstīšanas metode ļauj viegli un saprotami salīdzināt algoritmus, jo, pat nezinot neko par algoritmiem, varēsim secināt, ka algoritms ar sarežģītību *O(n)* būs ātrāks par algoritmu *O()*, jo n < .

Svarīgi ir piebilst, ka turpmāk aprakstot, salīdzinot vai pētot algoritmu sarežģītību, īsināsim to šādi:

*O(n + 1)* pārveidosim par *O(n)* tā, ka viens papildus solis algoritma izpildes laiku ietekmēs ļoti minimāli. Tātad īsināsim 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, un pareiza algoritma un datu struktūru izvēle var būtiski mainīt risinājumu ātrumu un precizitāti. Kā jau tika minēts 2.1 nodaļā, lielā O notācija ir ērts rīks, kas ļauj viegli salīdzināt algoritmus un novērtēt tā grūtību. Tas ir ļoti svarīgi šī projekta izpildījumā, jo ar lielo O notāciju tiks konkrēti aprakstīti pamatalgoritmu un datu struktūru ātrumi un prasības pēc atmiņas. Tas arī ļaus noskaidrot aģenta potenciālās vajadzības pēc atmiņas daudzuma un izšķirtspējām. Zinot šo informāciju, būs iespējams veikt pamatotus spriedumus par MI iespējām nākotnē.

Apskatot lielo O notāciju citos dzīves piemēros, kā 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 novērojamajā visumā [2, p. 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 ⋅ gadus [17].

**3.** **Meklēšanas algoritmi**

**3.1 A\* meklēšanas algoritms**

**3.1.1 Kas ir A\* meklēšanas algoritms?**

A\* ir informēts meklēšanas algoritms, kas strādā ar grafu, kurā katram ceļam ir zināma prioritāte [18]. Piemērs, kā A\* algoritms atrod īsāko ceļu (skatīt 1. attēlu):

Tā kā A\* algoritms zina attālumu no virsotnēm līdz galapunktam, lai sasniegtu mērķi ātrāk, tas izvēlēsies virsotnes, kuras ir tuvākas galapunktam.



(1. attēls: grafa attēlojums, kā ceļi starp Latvijas pilsētām)

A\* algoritma ātruma sarežģītība ir eksponenciāla O(), kur b ir sazarojuma faktors – cik sazarojumu ir katrai virsotnei, bet d ir īsākais ceļa garums līdz galapunktam. Tostarp atmiņas sarežģītība ir visu apskatīto virsotņu skaits, kas sliktākajā gadījumā būs O(|V|), kur V ir virsotņu skaits [18].

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

A\* 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. Šis 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 tikts izmantots A\* algoritms, piemēram, mašīnu sacīkstēs, lai nodrošinātu, ka citas mašīnas spēlē tiek līdzi [19].

**3.2 BFS meklēšanas algoritms**

**3.2.1 Kas ir BFS meklēšanas algoritms?**

BFS (angliski Breadth-First Search) meklēšanas algoritms ir viens no vienkāršākajiem grafa meklēšanas algoritmiem. BFS algoritmam nav nekādu zināšanu par mēra virsotņu iespējamo atrašanās vietu. Tas līdzvērtīgi un sistemātiski, sākot no sākuma virsotnes, izskata apkārtējās virsotnes soli pa solim līdz atrod mērķi vai izskata visu grafu, mērķi neatrodot. [20]

BFS algoritma laika sarežģītība ir O(|V| + |E|), kur |V| - virsotņu skaits, bet |E| - pāreju skaits. Šāda lineāra sarežģītība rodas, jo BFS algoritms katrai virsotnei izskata visas pārejas pēc kārtas. Atmiņas sarežģītība ir O(|V|), jo BFS algoritms izmanto rindu, lai glabātu izskatāmās virsotnes un masīvu ar jau izskatītajām virsotnēm, kas ir |V| izmērā. [20]

**3.2.2 Kur izmanto BFS meklēšanas algoritmu?**

BFS meklēšanas algoritms var būt noderīgs maršrutu sastādīšanā, kad nav svarīgi ceļa garumi un distance tiek mērīta soļos no starta. [20] BFS algoritmu var izmantot arī, piemēram, labirintu problēmu risināšanā, kur BFS algoritms sliktākajā gadījumā izies caur visu labirintu, lai atrastu izeju. [21]

**3.3** **Meklēšanas algoritmu nozīme šajā projektā**

A\* meklēšanas algoritms, kā viens no efektīvākajiem klasiskās meklēšanas algoritmiem [2, pp. 95-99], šajā pētījumā tiks izmantots, lai gūtu priekšstatu par racionāli rīkojoša aģenta limitācijām. BFS meklēšanas algoritms, kā primitīvāks meklēšanas algoritms, tiks izmantots, lai salīdzinātu un izpētītu, kā dažādi algoritmi var ietekmēt aģenta veiktspēju.

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.

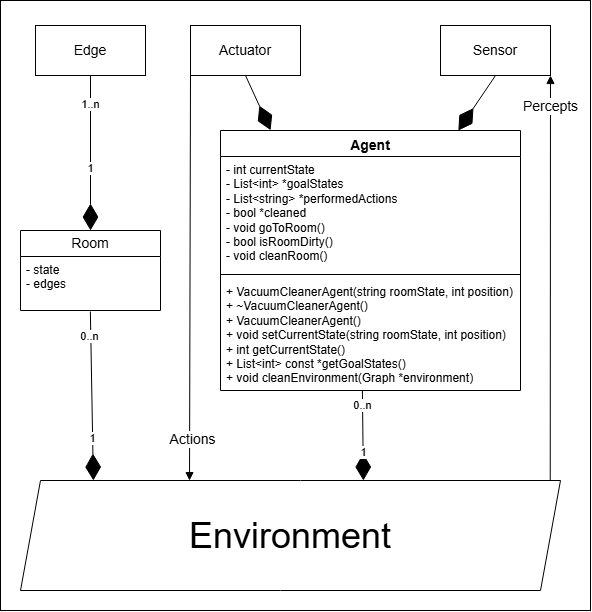
**4. Vides apraksts**

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 daļēji pārredzama, jo aģents ar sensoru palīdzību ir spējīgs uztvert tikai savu pozīciju jeb istabu, kurā tas atrodas dotajā brīdī, un apkārtējās istabas, uz kurām ir iespējams pārvietoties. 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 grafā reprezentē stāvokļu grafs, kur katra virsotne atbilst konkrētam stāvoklim un pāreja starp virsotnēm nozīmē stāvokļa maiņu. Grafa stāvokļu skaitu izsaka formula *⋅ n* (*n* – istabu skaits), jo 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 arī veikt tīrīšanu. Grafa pārejām pastāv *⋅ 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 , un ko ir jāreizina ar *n*, jo vienā no šīm *n* istabām atrodas aģents. Rezultātā grafa pāreju skaits ir:

*(n ⋅ ⋅ n) + (n ⋅ ⋅ (n – 1)) = n ⋅ (2n – 1)*

Vides relāciju grafiskais attēlojums (skatīt 2. attēlu):



(2. attēls: vides attēlojums relāciju modeļa diagrammā)

**5. Heiristika**

Klasiskā meklēšana nereti saistās ar bezgalīgi daudz iespējamo ceļu analīzi. Efektīva rezultāta priekšnosacījums bieži vien ir informācijas uzdošana par to, kuri ceļi 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 piemēru 3. attēlā) 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 ir 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.

***Attēls, kurā ir teksts, fonts, ekrānuzņēmums, tipogrāfija

Apraksts ģenerēts automātiski***

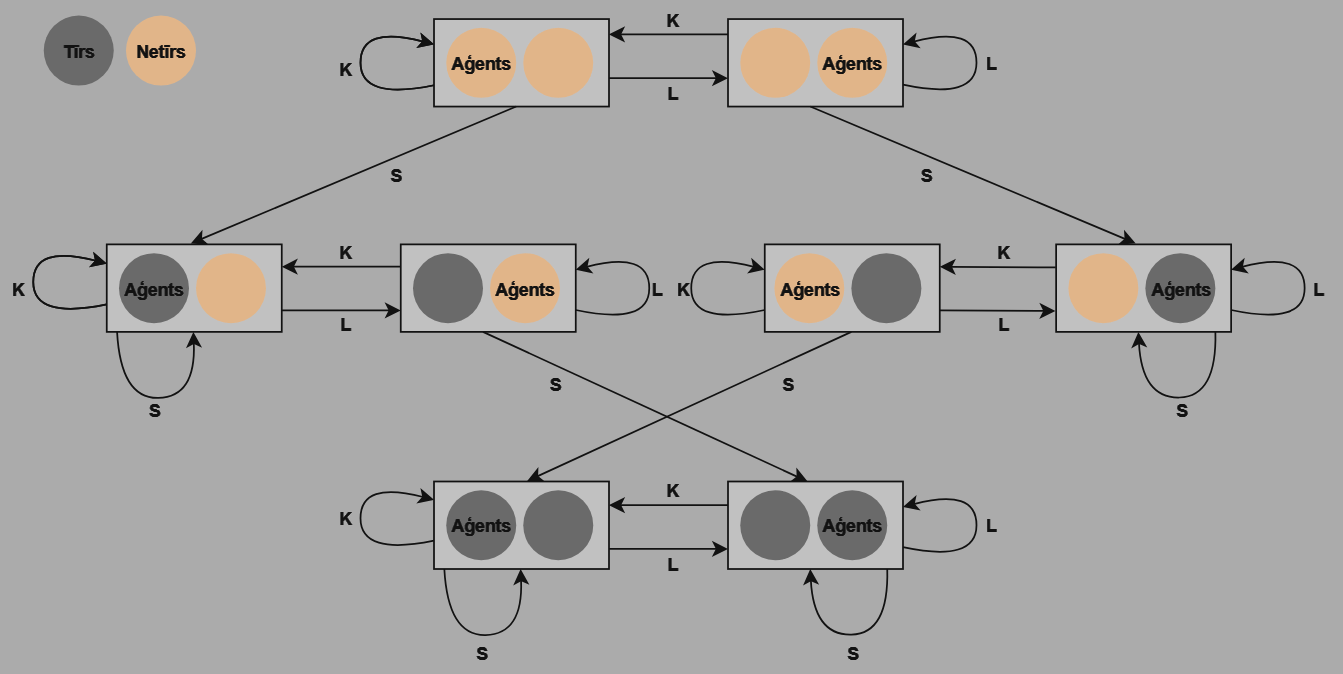
(3. attēls: 2 istabu faila reprezentācija)

Izmantojot šādu sakarību balstītu heiristiku, putekļsūcēja aģents veiks lieko grafa pāreju jeb pāreju uz jau tīrām istabām izslēgšanu, nodrošinot pareizo pāreju izvēli un efektīvu resursu lietojumu.

**6.** **Aģenta apraksts**

Šī projekta MI pieeja ir putekļsūcēja modelis, kā racionāli rīkojošs aģents. Matemātiski tas nozīmē, ka aģenta funkcija saņem ievaddatus, kurus aģents ar sensoru palīdzību nolasa, apstrādā un atgriež aktuatoru soļus. Aģenta funkcijas galvenais uzdevums ir maksimizēt 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 vismazākais aģenta darbību skaits.

Koda realizāciju skatīt GitHub ***ŠEIT***. Grafisks piemērs aģenta darbībai 2 piesārņotu istabu situācijā (skatīt 4. attēlu):



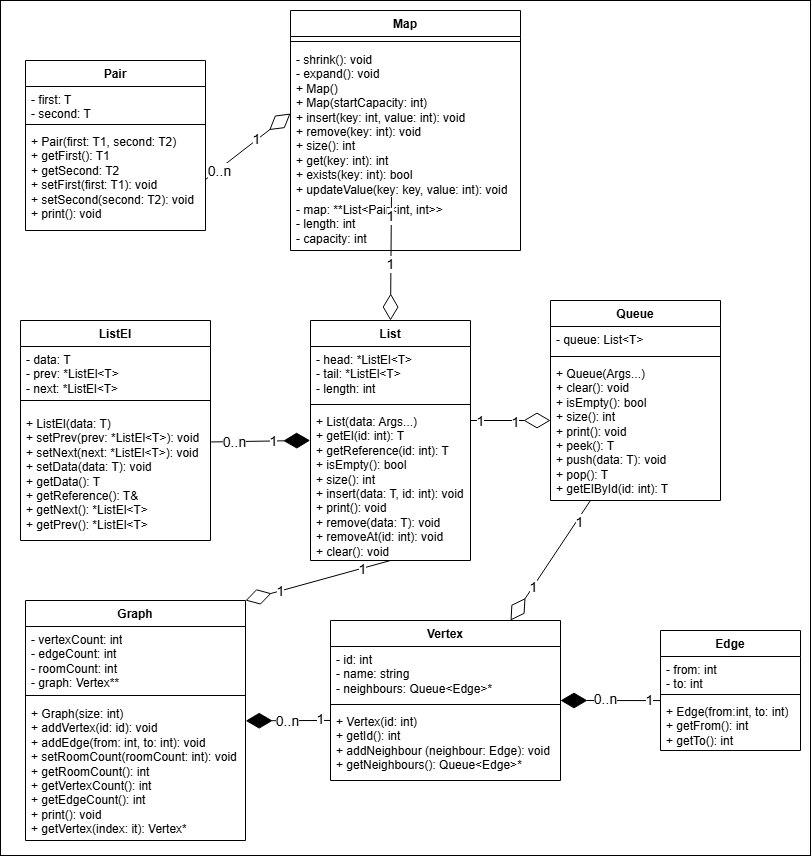
(4. attēls: aģenta darbības attēlojums 2 piesārņotu istabu situācijā, kur

K – iet pa kreisi, L – iet pa labi, S – sūkt)

**7. Datu struktūras**

Datu struktūru implementācija GitHub ***ŠEIT***

Datu struktūru grafiskais attēlojums (skatīt 5. attēlu):

****

(5. attēls: implementēto datu struktūru attēlojums UML klašu diagrammā)

**7.1 Saistītais saraksts**

Saistītais saraksts 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 saraksta 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 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 iegūšana.

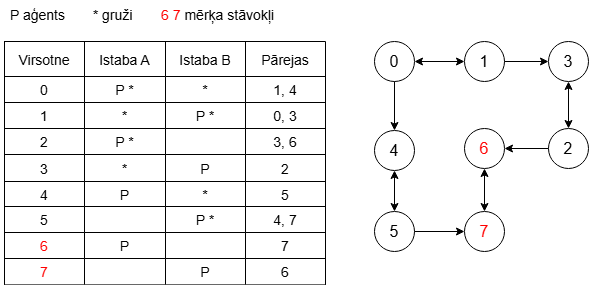
**7.2 Rinda**

Rinda ir lineāra datu kolekcija, kuras funkcionālais princips 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 ir nelineāra datu struktūra, kas sastāv no virsotnēm (skatīt 7.3.1) un pārejām (skatīt 7.3.2), 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. Pieeja grafa virsotnēm un pārejām var atšķirties, taču šajā tvērumā grafs glabā virsotņu objektus dinamiskā masīvā, kur katra virsotne glabā kaimiņvirsotnes, proti, pārejas. Piemērs grafa reprezentācijai 2 piesārņotu istabu situācijā (skatīt 6. attēlu):



(6. attēls: grafa attēlojums 2 piesārņotu istabu situācijā)

**7.3.1 Virsotne**

Virsotne 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 aģenta stāvokļiem.

**7.3.2 Pāreja**

Grafā izmanto pārejas, 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) būs sūkšana, kur stāvoklis B reprezentē aģentu tajā pašā istabā, bet tīrā.

**7.4 Mape**

Mape (zināma arī, kā vārdnīca) 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 mape ir izveidota, kā heša mape (angliski hash map), kas var saturēt tikai veselus skaitļus. Tā izmanto dinamisku masīvu no saistītajiem sarakstiem, kas savukārt sastāv no pāriem (atslēga – vērtība). Kad mapei 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 mapes saistītajā sarakstā atrodas viens elements. Lai šim ideālam pietuvotos, palielinoties vērtību skaitam, mapes masīvs arī tiek palielināts, lai izkliedētu vērtības.

Šajā projektā mapi izmanto BFS meklēšanas algoritmam, lai saglabātu virkni no 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 algoritms un informēts meklēšanas algoritms ar eksperimentāli noteiktu heiristiku. Neskatoties uz to, ka aģents var darboties dažādās vides variācijās, kā pamata stāvoklis sarežģītības izvērtēšanai 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, izejot caur visām grafa virsotnēm. Otrā gadījumā tiek izmantots A\* algoritma paveids, kas balstās uz pielāgotu heiristiku (vairāk skatīt 5. nodaļā), kas ļauj aģentam saprast, kura ir pareizā pāreja.

**8.2 BFS algoritma novērtēšana**

BFS meklēšanas algoritms, lai atrastu īsāko ceļu, sliktākajā gadījumā izies caur visu grafu, tāpēc laika sarežģītība būs O(|V| + |E|), kur |V| – grafa virsotņu skaits un |E| – pāreju skaits. Tā kā šī projekta ietvaros BFS tiek izmantots īsākā ceļa noteikšanai, telpas sarežģītība ir O(|V| + (2 ⋅ |E|) + n). Kur V atmiņa tiek izmantota masīvam, kurš glabā apmeklētās virsotnes. E atmiņa tiek izmantota gan rindai, lai secīgi izietu caur grafu, gan mapei, kas satur visas atsauces uz noietajiem ceļiem, tāpēc iegūst 2E. Bet 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() = O()

Telpas sarežģītība pie *n* istabām:

O() = O()

**8.3 A\* algoritma novērtēšana**

A\* meklēšanas algoritma sarežģītība ir tieši atkarīga no izmantotās heiristikas, kas šī projekta gadījumā ir nevajadzīgo zaru izslēgšana. Izpētot implementēto algoritmu (skatīt GitHub šeit) tika iegūta laika sarežģītība sliktākajā gadījumā O(), kur *n* ir stabu 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()), 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 O(*n + n + 3n*), kur pirmais n ir apmeklētās istabas, otrais n ir mērķa stāvokļu un 3n ir aģenta veiktās darbības. Tā kā aģents pārbaudīs, aizies un iztīrīs katru istabu, izņemot pirmo istabu uz kuru nav jāiet, taču šo soli vērā neņem, jo tas ir konstants viens solis.

Laika sarežģītība pie n istabām: O()

Telpas sarežģītība: O(*5n*)

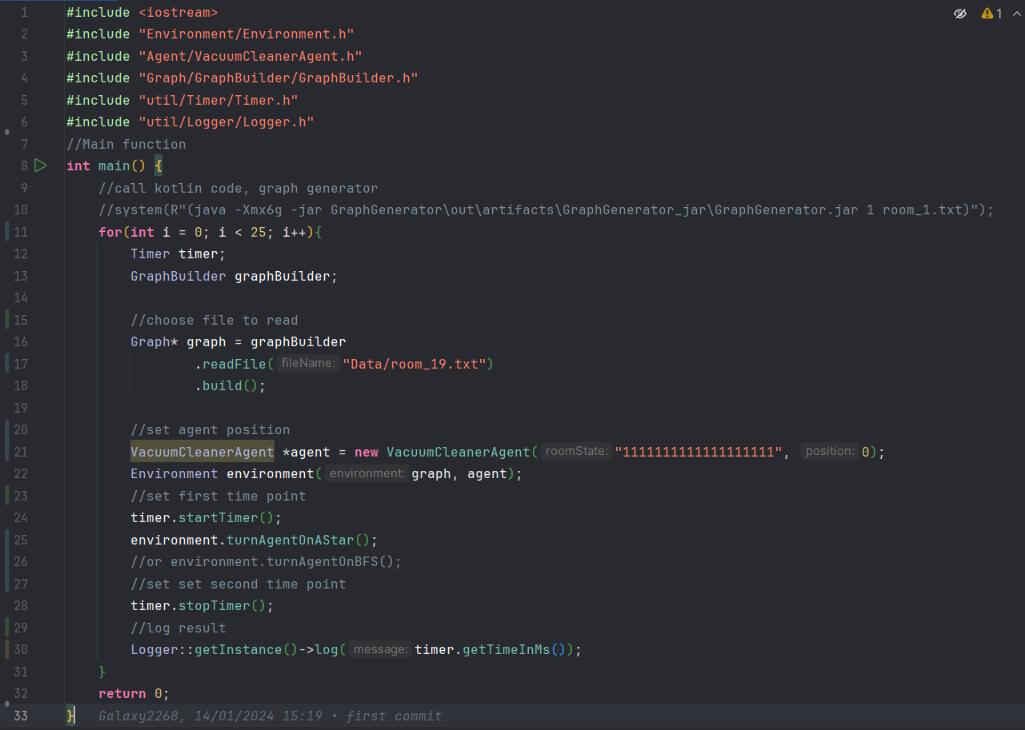
**8.1.3 BFS algoritma sarežģītība attiecībā pret A\* algoritma sarežģītību**

Pēc iegūtām teorētiskām laika un telpas sarežģītību formulām un to salīdzināšanas, tika secināts, ka aģents ar informētu stāvokļu pārejas algoritma palīdzību sākotnēji eksponenciālu komplicitātes problēmu pārvērš polinomiālā. Savukārt eksponenciālo telpas sarežģītību pārvērš lineārā.

**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 algoritms un A\* algoritms ar pašveidotu heiristiku. Atskaites punkts ir brīdis, kad algoritms uzsāk meklēšanu, bet mērīšanas beigas ir brīdis, kad algoritms pabeidz meklēšanu.

Aģenta darbība tika mērīta, izmantojot C++ bibliotēku “*chrono*”. Lai noteiktu algoritma aizņemto laiku, tiek noteikts starta un beigu punkts, kuru starpība ir patērētais laiks meklēšanai. Rezultāts tiek ierakstīts atsevišķā “*logs.txt*” failā, izmantojot logera instanci. Iestatītais kods rezultātu iegūšanai (skatīt 7. attēlu):



(7.attēls: kods 19 istabu palaišanai, kur var izvēlēties ieslēgt aģentu ar A\* vai BFS algoritmu)

Protams, ir svarīgi ņemt vērā izmantoto datoru, jo tas var ietekmēt programmas izpildi. Šajā projektā tiek izmantots dators ar šādām specifikācijām:

* Procesors: AMD Ryzen 5 Mobile 3550H (4 kodoli, frekvence 2,1 GHz)
* Videokarte: NVDIA GeForce GTX 1050 (GDDR 5, 3GB, 1400 MHz)
* Operatīvā atmiņa: 16GB DDR4 (palaišanas brīdī pieejamā atmiņa ir aptuveni 10GB)
* Operētājsistēma: Windows 10 Home (22H2 versija)
* Sistēmas tips: 64 bitu operētājsistēma, x64 bāzes procesors

Grafiki iegūtajiem rezultātiem:

***EXCEL GRAFIKI***

**9.1 T-testi**