LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ĢENĒTISKO ALGORITMU IZMANTOŠANA DAUDZAĢENTU SISTĒMĀS

BAKALAURA DARBS

Autors: **Kristaps Kols**
Studenta apliecības Nr.: KK08343
Darba vadītājs: profesors, Dr. sc. comp. Guntis Arnicāns

RĪGA 2012

ANOTĀCIJA

Bakalaura darbs „Ģenētisko algoritmu izmantošana daudzaģentu sistēmās” tika izstrādāts ar mērķi izpētīt, kādus ieguvums dod ģenētisko algoritmu pielietošana daudzaģentu sistēmu darbībā. Šī darba izstrādes laikā autors iepazinās un izpētīja, kas ir daudzaģentu sistēmas un kādas ir metodoloģijas to realizācijai, kāda ir ģenētisko algoritmu darbība un kādas varētu būt datora resursu plānošanas iespējas.

Darba rezultātā valodā Python tika izstrādāta daudzaģentu sistēma, kas simulē dažādu uzdevumu veikšanu, kas noslogo datora resursus. Sistēma sastāv no aģentiem, kuri izpilda uzdevumus noslogojot dažādus datora resursus. Tika izstrādāta arī daudzaģentu sistēmas versija, kas izmanto ģenētisko algoritmu pieeju aģentu attīstīšanai. Šo abu sistēmu salīdzinājumi grafiku veidā un secinājumi aprakstīti darbā.

ABSTRACT

This Bachelor thesis „The use of genetic algorithms in multi-agent systems” was developed to explore what benefits gives the use of genetic algorithms in multi-agent systems. During the work out of the thesis the author explored and examined what are a multi-agent systems and what are the methodologies for their implementation, what is the operation of the genetic algorithms and what and what capabilities might be there for computer resource planning.

Work has resulted multi-agent system developed in the programming language Python that simulates a variety of tasks, which load the computer resources. The version of system was developed using genetic algorithm approach in order to modify the agents. The comparison of these two systems as graphs and conclusions is described in the work.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS 5
IEVADS 6
1. DAUDZAĢENTU SISTĒMAS 8
1.1. Daudzaģentu sistēmas apraksts 8
1.2. Aģents, DAS uzbūve un piemēri 9
2. ĢENĒTISKIE ALGORITMI 15
2.1. Ievads 15
2.1.1. GA Operatori 15
2.1.2. Vienkārša GA apraksts 16
2.2. Mantošana 20
2.3. Atlase 20
2.4. Krustošanās 22
2.5. Mutācija 24
3. IZSTRĀDĀTĀ DAUDZAĢENTU SISTĒMA 27
3.1. Vienkārša daudzaģentu sistēma 28
3.2. Daudzaģentu sistēma ar GA 28
3.3. Pirmais modulis, daudzaģentu sistēma un GA 29
3.3.1. Daudzaģentu programmatūras inicializācija 29
3.3.2. Ģenētiskā algoritma pielietošana 30
3.3.2.1. Atlase un derīguma funkcija 30
3.3.2.2. Krustošanās 31
3.3.2.3. Mutācija 31
3.4. Otrais modulis, aģentu klase 32
3.5. Trešais modulis, virtuālā datora klase 32
3.6. Ceturtais modulis, uzdevumu klase 33
3.7. Piektais modulis, resursu klase 33
4. DAUDZAĢENTU SISTĒMU SALĪDZINĀJUMS 34
REZULTĀTI 36
SECINĀJUMI 37
PATEICĪBAS 38
IZMANTOTĀ LITERATŪRA UN AVOTI 39
PIELIKUMI 41
1. PIELIKUMS. PIRMAIS MODULIS 41
2. PIELIKUMS. OTRAIS MODULIS 47
3. PIELIKUMS. TREŠAIS MODULIS 49
4. PIELIKUMS. CETURTAIS MODULIS 51
5. PIELIKUMS. PIEKTAIS MODULIS 52

APZĪMĒJUMU SARAKSTS

GA – ģenētiskais algoritms, pie evolucionārās skaitļošanas metodēm pieskaitāms optimizēšanas algoritms.

DAS – daudzaģentu sistēma, kas sastāv no vairākiem savstarpēji saistītiem viedajiem aģentiem.

Python – universāla, augsta līmeņa programmēšanas valoda, kuras dizaina filozofija uzsver koda lasāmību.

IEVADS

Ir daudzas problēmas ko nav iespējams risināt izmantojot monolītu vai viena aģenta sistēmu, kas uzdevumus veic secīgi. Daudz un dažādu paralēlu problēmu risināšanai, kā viens no iespējamajiem variantiem ir daudzaģentu sistēmu izmantošana, tomēr šo sistēmu ieguvumi vai zaudējumi vēl joprojām ir diezgan maz izpētīta sfēra un izstrādātās daudzaģentu sistēmas ir diezgan eksperimentālas.

Daudzaģentu sistēma sastāv no daudziem aģentiem, kas katrs veic noteiktus uzdevumus, kā un vai veikt šos uzdevumus, nosaka dažādi aģentu parametri, kas parasti ir iepriekš izvēlētas konstantes. Šādās sistēmas, parasti, papildus darbu veikšanai jauni aģenti tiek radīti kopējot citus aģentus ar iepriekš definētām parametru vērtībām, kas visdrīzāk ir empīriski noteiktas, taču nenozīmē, ka tās ir vislabākās.

Šī darba izstrādes laikā tiek pētīts tas, vai evolucionārās attīstības idejas var izmantot, lai laika gaitā mainītu aģentu parametru vērtības.

Darba autors vēlas pētīt to, vai ar ģenētisko algoritmu izmantošanu var uzlabot aģentu veiktspēju, un uzlabot šīs katra aģenta parametru konstantes, tādējādi programmatūras darbības laikā tiktu uzlabota pati daudzaģentu sistēma un tās ātrdarbību, to vai tas vispār ir iespējams un kādi ieguvumi vai zaudējumi rodas no šādas pieejas.

Autora izvirzītā hipotēze ir tāda, ka daudzaģentu sistēmu ātrdarbību var uzlabot izmantojot ģenētiskos algoritmus aģentu uzlabošanai.

Šī darba mērķis ir izstrādāt daudzaģentu sistēmu pielietojot ģenētiskos algoritmus, kas sistēmas darbības laikā attīsta aģentus.

Lai sasniegtu darba mērķi tika izvirzīti šādi uzdevumi:

- iepazīties ar literatūru un esošo situāciju daudzaģentu sistēmu izmantošanā, to paradigmām;
- iepazīties ar literatūru par ģenētisko algoritmu paradigmām un to realizācijas iespējām;
- izstrādāt valodā Python daudzaģentu sistēmu, kas simulē dažādu darbu veikšanu.

Autors darba izstrādes laikā veica gan teorētiskus pētijumus (lieteratūra), gan eksperimentālus (izstrādāta daudzaģentu sistēma).

Kā faktoloģiskie materiālu avoti tiek izmantotas grāmatas, zinātniskie raksti un interneta resursi.

Pirmajā nodaļā ir dots skaidrojums, kas ir daudzaģentu sistēmas, kādi ieguvumi ir no to pielietošanas.

Otrajā nodaļā aprakstīts, kas ir ģenētiskie algoritmi, sīkāk izpētīta mantošana, atlase, krustošanās un mutācija.

Trešajā nodaļā aprakstīta izstrādātā daudzaģentu sistēma.

Ceturtajā nodaļā apkopoti testēšanas rezultāti un salīdzinātas izstrādātās daudzaģentu sistēmas – viena parastā daudzaģentu sistēma un otra, kas papildināta ar ģenētisko algoritmu pieeju.

1. DAUDZAĢENTU SISTĒMAS

1.1. Daudzaģentu sistēmas apraksts

Šajā nodaļā sīkāk ir izpētītas un aprakstītas daudzaģentu sistēmu paradigmas, izstrādes metodoloģija un apskatīti to izmantošanas piemēri.

Mākslīgā intelekta pētijumos, uz aģentiem balstītas sistēmas risinājumi, tiek pasniegti kā jauna paradigma programmatūras sistēmu konceptualizācijai, izstrādāšanai un ieviešanai. Aģenti ir sarežģītas datorprogrammas, kas darbojas autonomi, palīdzot to lietotājiem atklātās un dalītās vidēs, atrisināt arvien vairāk sarežģītas problēmas. Tomēr, pieaug programmatūras, kurām nepieciešami vairāki aģenti, kas strādā kopā savstarpēji sadarbojoties. Daudzaģentu sistēmas (DAS), ir brīvi savienots tīkls, kas sastāv no daudziem programmatūras viedajiem aģentiem, kas savstarpēji mijiedarbojas vienā vidē, lai atrisinātu problēmas, kas pārsniedz individuāla aģenta spējas un zināšanas par katru no problēmām, kas jāatrisina [1].

Daudzaģentu sistēmas var tikt izmantotas, lai risinātu problēmas, ko ir grūti vai neiespējami risināt ar monolītu vai vienaģenta sistēmu.

Daudzaģentu sistēmām ir sekojošas priekšrocības salīdzinot ar vienaģenta vai centralizēto pieeju:

- DAS izmanto skaitļošanai pieejamos resursus un iespējas visā savstarpēji saistīto aģentu tīklā. Gadījumos, kur centralizētas sistēmas var tikt ierobežotas ar resursiem, darbības vājajām vietām vai kritiskām kļūdām, DAS ir decentralizēta sistēma un tādējādi necieš no „viena punkta kļūdas" problēmas, kas parasti ir centralizētām sistēmām.
- DAS ļauj veidot starpsavienojumus un starpoperēšanu ar esošajām sistēmām. Izveidojot aģentu aptinumu ap šādām sistēmām, tās var tikt ievietotas kopējā tīklā kā aģentu sabiedrības sastāvdaļas.
- DAS modelē problēmu attiecības, kas tiek risinātas daudz dabiskāka veidā, kā, piemēra, uzdevumu sadalīšana, komandas darba plānošana, atvērta vide, utt.
- DAS efektīvi iegūst, filtrē un globāli koordinē informāciju no avotiem, kas ir izplatīti visā vidē.

- DAS piedāvā risinājumus situācijās, kur ekspertīze ir telpiski un īslaicīgi izplatīta.
- DAS uzlabo kopēju sistēmas veiktspēju, jo īpaši skaitļošanas efektivitāti, uzticamību, robustumu, apkopi, atsaucību, elastību un atkal izmantošanu [1].

1.2. Aģents, DAS uzbūve un piemēri

Aģents – datorsistēma, kas ir spējīga uz neatkarīgu (autonomu) darbību, sekmējot tā lietotāja vai īpašnieka vēlmju izpildi (mēģina saprast, kas ir jādara, lai atrisinātu problēmu, nesaņemot norādes no lietotāja) [2].

Aģentus var iedalīt vairākos tipos:

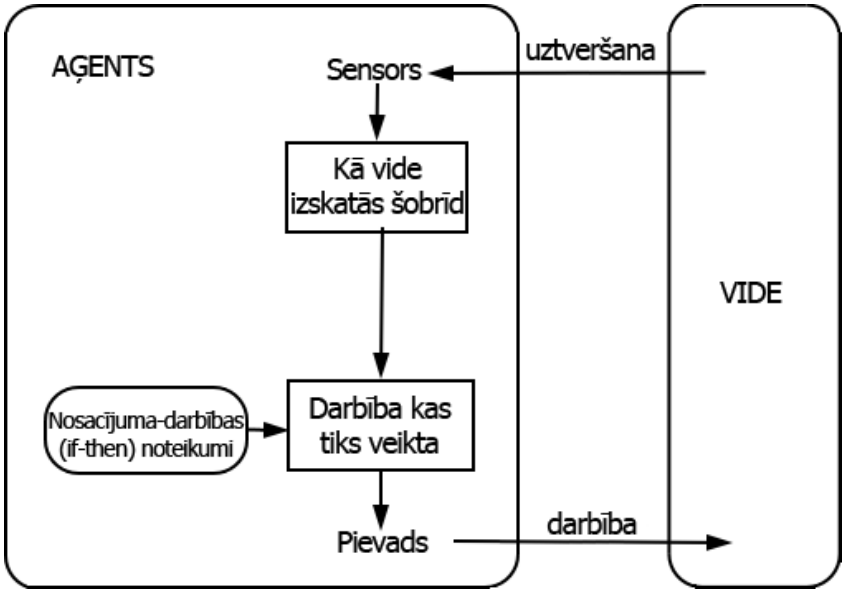
- Ļoti vienkāršs aģents – pasīvais aģents vai aģents bez mērķa (kā šķērslis vai jebkura vienkārša simulācija).
- Aktīvs aģents ar vienkāršiem mērķiem (kā putni, kas uzturas baros vai vilki-aitas medījuma-medītāja modelī).
- Ļoti sarežģīts aģents (piemēram, kognitīvs aģents, kam jāveic daudz sarežģīti aprēķini) [3].

Vidi kurā aģenti darobjas var iedalīt:

- Virtuāla vide
- Diskrēta vide
- Pastāvoša vide

Vidi kur atrodas aģenti var organizēt atbilstoši dažādiem parametriem, piemēram: pieejamība (atkarīgs, vai ir iespējams iegūt pilnīgu informāciju par vidi), determinētība (vai darbība, kas tiek veikta vidē, izraisa noteiktu efektu), dinamika (cik daudz vienības šobrīd ietekmē vidi), diskrētums (vai darbības, kas iespējamas vidē, ir ierobežota skaita), epizodiskums (vai aģenta veiktās darbības vienā laika periodā, ietekmē citos periodos notiekošo) un dimensionalitāte (vai teritorijas raksturojums ir svarīgs faktors videi un vai aģents ņem vērā pieejamos resursus lēmumu pieņemšanā) [4,5].

Kā redzams attēlā 1.1., vienkāršs aģents ar apkārtejo vidi mijiedarbojas divos virzienos – uztverot vidē notiekošo ar sensoru un veicot kādu atgriezentisku darbību uz vidi. Aģents pēc informācijas uztveršanas to apstrādā, un atkarībā no nosacījumiem izvēlas atbildes reakciju uz ārējo vidi.



1.1. att. **Vienkārša daudzaģentu sistēma [6]**

Īsumā vēlreiz varam definēt, kas ir daudzaģentu sistēma – tā ir sistēma, kas sastāv no inteliģentiem aģentiem, kas savstarpēji mijiedarbojas. Lielākoties aģenti darbojas izpildot lietotāja vēlmes, katram no tiem ir savs mērķis un motivācija problēmas izpildei. Lai aģenti veiksmīgi varētu sadarboties, tiem ir nepieciešamas tādas īpašības kā sadarbošanās, koordinēšanās un vienošanās, tā pat, kā to var novērot starp cilvēkiem.

Aģenti kā programminženierijas paradigma:

Programminženierijas speciālisti ir nonākuši pie daudz labākas izpratnes par iespējamo programmatūras sarežģītību. Šobrīd plaši tiek atzīts tas, ka mijiedarbība ir iespējams vissvarīgākā īpašība sarežģītā programmatūrā [2].

Kāda ir aģenta būtība?

- Pats svarīgākais pamatpunkts ir tāds, ka tie ir autonomi: spējīgi veikt patstāvīgas darbības.
- Tādējādi: aģents ir datorsistēma, kas spējīga autonomi veikt darbības kādas vides ietvaros, lai sasniegtu savus deleģētos mērķus.
- Par aģentu un tā vidi mēs varam domāt kā tuvu sapārotu pāri, kas viens ar otru mijiedarbojas: uztveršana – lēmums – darbība – uztveršana – lēmums.

Vienkārša aģenta piemērs

- Termostats
  - deleģētais mērķis ir saglabāt istabas temperatūru nemainīgu
  - darbības ko tas var veikt ir ieslēgts un izslēgts siltums
- UNIX iebelzt (*biff*) programma
  - deleģētais mērķis ir pārraudzīt ienākošos e-pastus un atzīmēt tos
  - darbības ko tā veic ir lietotāja interfeisa darbības (paziņojumi, utt.)

Šādi aprakstītie aģenti ir vienkārši un to lēmumu pieņemšana un turpmākās darbības ir triviālas.

Inteliģentu aģentu darbību var raksturot ar trim pamatīpašībām:

- reaktīvs;
- proaktīvs;
- sociāls.

Reaktivitāte

- Ja programmas izpildes vide garantēti ir nemainīga, programma var izpildīties „kā akla".
- Reālā pasaule nav gluži tāda, lielākā daļa vidu ir dinamiskas.
- Programmatūru ir grūti izstrādāt priekš dinamiskām jomām: programmai jāņem vērā neveiksmes iespējamību – rodas jautājums, vai tādai programmai vispār ir vērts izpildīties.
- Reaktīva sistēma ir tāda sistēma, kas uztur patstāvīgu sadarbību ar tās izpildes vidi, un reaģē uz izmaiņām, kas notiek tajā (laikā, kamēr tas vēl ir noderīgi).

Proaktivitāte

- Reaģēt uz vidi ir viegli (piemēram, stimuli -> atbilžu noteikumi).
- Bet mēs parasti gribam aģentus, kas *dara lietas priekš mums*.
- Tātad *mērķis, kas vērsts uz uzvedību*.
- Proaktivitāte = radīt un mēģināt sasniegt mērķus; nosaka ne tikai notikumi; uzņemties iniciatīvu.
- Atpazīt izdevības.

Sociālā spēja

- Reālā pasaule ir daudzaģentu vide: mēs nevaram izslēgt grūtības un neņemt citus vērā mēģinot sasniegt savus mērķus.
- Dažus mērķus sasniegt var tikai sadarbojoties ar citiem.
- Sociālā spēja aģentu kontekstā ir to spēja mijiedarboties ar citiem aģentiem (un iespējams cilvēkiem) caur sadarbošanos, koordinēšanos un vienošanos.
- Tātad mēs varam runāt vismaz par iespēju, ka aģenti komunicē savā starpā.

Sociālā spēja: Sadarbība

- Sadarbība ir strādāšana visiem kopā kā komandai, lai sasniegtu kopīgu mērķi.
- Bieži tas tiek pamatots ar to, ka aģents viens pats nemaz nevar sasniegt mērķi, vai ka sadarbība sniegs labākus rezultātus (piemēram, ātrāk tiks iegūts rezultāts).

Sociālā spēja: Koordinācija

- Koordinācija ir spēja pārvaldīt atkarības starp aktivitātēm.
- Piemēram, ja ir kāds resurss, ko var izmantot tikai viens, un to gribu izmantot gan es, gan kāds cits, tad mums vajag koordinēt mūsu darbības.

Sociālā spēja: Vienošanās

- Vienošanās ir spēja panākt saskaņu par kādu jautājumu, kas skar kopīgas intereses.
- Piemēram: jums mājās ir viens TV, un jūs gribat skatīties filmu, bet draugs grib skatīties futbolu.
- Iespējamais risinājums: skatīties futbolu šovakar un filmu rīt.
- Parasti ietver piedāvājumu un pret piedāvājumu, ar kompromisiem ko piedāvā dalībnieki.

Bieži tiek jautāts, kāda ir atšķirība starp objektorientēto pieeju un aģentu pieeju, vai tas nav tas pats, zemāk uzsvērtas atšķirības, kas parāda to, ka aģentu pieeja ir kaut kas cits.

- Aģenti ir autonomi: aģenti iemieso spēcīgāku priekšstatu par autonomiju nekā objekti, un tipiski, aģenti paši izlemj to vai veikt kādu darbību vai nē, ja to ir pieprasījis cits aģents.
- Aģenti ir gudri: tas izpaužas elastīgā (reaktīvā, proaktīvā, sociālā) uzvedības modelī – objektorientētās pieejas modelis neraksturo šādus uzvedības veidus.

- Aģenti ir aktīvi: tie nav pasīvi pakalpojumi sniedzēji, kur vienīgais mērķis ir izpildīt citu lūgums.

Objekti to dara par brīvu

- Aģenti to dara tāpēc, ka grib.
- Aģenti to dara naudas dēļ.

Vides īpašības

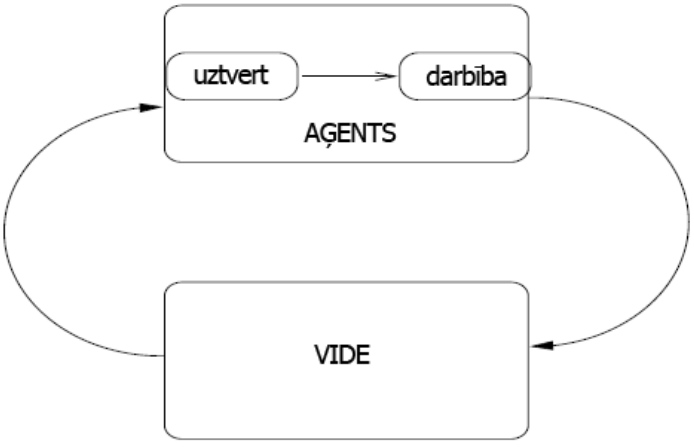
- Pieejamība pret nepieejamību. Pieejama vide ir tāda, kurā aģents var iegūt pilnīgu, precīzu un pašu jaunāko informāciju par vides stāvokli.
- Lielākā daļa sarežģītu vidu (tostarp, piemēram, ikdienas fiziskā pasaule un internets) ir nepieejamas.
- Jo pieejamāka ir vide, jo vieglāk ir izveidot aģentu, kas darbojas šādā vidē.

Pilnīgi reaktīvi aģenti

- Pastāv aģenti, kas savas darbības balsta bez ieskata pagātnē – tie balsta savas darbības tikai uz tagadnes situāciju, neņemot vērā, kas ir noticis pagātnē.
- Tādus aģentus mēs saucam par pilnīgi reaktīviem: *darbība* : *E -> Ac*
- Piemēram, termostats ir pilnīgi reaktīvs aģents.
- *darbība*(*e*) = izslēgt, ja *e* = temperatūra ir vajadzīgā
  ieslēgt, citādi.

Uztvere

Attēlā 1.2. redzama ļoti vienkāršota aģenta un vides mijiedarbība.



1.2. att. **Vienkārša aģenta mijiedarbība ar vidi [7]**

Uztveres funkcija ir aģenta spēja novērot savu apkārtējo vidi, savukārt *darbības* funcija attēlo aģenta lēmuma pieņemšanas procesu.

Uztveres funkcijas izeja ir priekšstats (*percept*):

Uztvert : E -> Per

Kas piemeklē vides stāvokļus priekšstatiem, un *darbība* tagad ir funkcija

Darbība : Per -> A*

Kas piemeklē priekšstata sekvences darbībām.

Aģenta kontroles cikls

1. Aģents sāk darboties sākuma stāvoklī i_0.

2. Atkārto mūžīgi:

Novēro vides stāvokli un veido priekštatu caur *uztveršanu*(...)

Atjauno iekšējo stāvokli caur *nākamo* funkciju(...)

Izvēlas darbību caur *darbību*(...)

Veic *darbību* [7].

2. ĢENĒTISKIE ALGORITMI

2.1. Ievads

Ģenētiskais algoritms (GA) ir pie evolucionārās skaitļošanas metodēm pieskaitāms optimizēšanas algoritms. Tas ir heiristiskas dabas algoritms un tiek pielietoti gan precīzu, gan tuvinātu risinājumu meklēšanai. Ģenētiskais algoritms ir uz evolucinārās bioloģijas idejas principiem balstīts algoritms.

Algoritma pamatā ir esošā risinājuma iteratīva uzlabošana, taču viena risinājuma vietā tiek izmantota risinājumu kopa, kuru, iedvesmojoties no bioloģijas, sauc par populāciju. Evolūcijas process ģenētiskajos algoritmos ir ļoti vienkāršota simulācija no mums zināmās bioloģiskās versijas. Tā sākas ar populāciju, kas sastāv no indivīdiem jeb kandidātiem, kas tiek uzģenerēti pēc nejaušības principa izmantojot kādu varbūtisku sadalījumu. Katras nākamās paaudzes radīšanai, tiek atlasīti indivīdi no pašreizējās paaudzes pamatojoties uz šī indivīda novērtējuma pēc derīguma funkcijas.

Risinājumu kopa iteratīvi tiek uzlabota, izpildot ar šo populāciju dabiskajai evolūcijai līdzīgas darbības: mantošanu, cīņu par izdzīvošanu, indivīdu pārošanu un mutēšanu [8].

2.1.1. GA Operatori

Visvienkāršākā ģenētisko algoritmu realizācija iekļauj trīs pamata operatorus: atlasi, krustošanos (viena punkta) un mutāciju.

Atlase – šis operators no kopējās populācijas atlasa indivīdus tālākai reprodukcijai. Jo indivīds ir derīgāks, jo biežāk pastāv iespēja, ka tas tiks izvēlēts reprodukcijai.

Krustošanās – šis operators nejauši izvēlas kādu vietu indivīda aprakstošajā virknē un apmaina vietām apakšvirknes pirms un pēc izvēlētā punkta starp abiem indivīdiem, lai radītu divus pēcnācējus. Piemēram, virknes 10000100 un 11111111 varētu tikt sakrustotas kopā pēc trešās pozīcijas katrā virknē, izveidojot divus pēcnācējus 10011111 un 11100100. Krustošanās operators aptuveni imitē bioloģisko krustošanos starp diviem vienas hromosomas (haploīda) organismiem.

Mutācija – šis operators pēc nejaušības principa apgriež otrādāk dažus bitus indivīda aprakstošajā virknē. Piemēram, virkne 00000100 varētu mutēt tās otrajā pozīcijā, izveidojot

virkni 01000100. Mutācija var notikt katrā virknes bita pozīcijā ar kādu varbūtību, parasti tā ir ļoti maza (piem., 0,001).

2.1.2. Vienkārša GA apraksts

Zinot skaidri definētu problēmu, kas ir jārisina un izmantojot bitu virknes reprezentāciju indivīdu aprakstīšanai, vienkāršs ģenētiskais algoritms strādā šādi:

1) Pēc nejaušības principa tiek uzģenerēta populāciju ar n l-bitu indivīdiem (iespējamie risinājumi problēmai).
2) Tiek aprēķināta derīguma vērtība (*fitness value*) f (x) katram indivīdam x no kopējās populācijas.
3) Tiek atkārtoti sekojošie soļi līdz tiek radīti n pēcnācēji jaunajai populācijai:
    a) Tiek atlasīti divi indivīdi no pašreizējās populācijas, varbūtība tikt izvēlētam pieaug esot derīgākam pēc derīguma funkcijas. Atlase tiek veikta „ar nomaiņu", kas nozīmē to, ka vieni un tie paši indivīdi var tikt atlasīti vairākas reizes, lai tiktu izmantoti kā vecāki nākamās paaudzes ģenerēšanai.
    b) Ar varbūtību p_c („krustošanās varbūtība" jeb „krustošanās iespējamība"), izvēlētais vecāku pāris tiek sakrustots nejauši izvēlētā punktā (izvēlēts ar vienmērīgu (*uniform*) varbūtību), lai izveidotu divus pēcnācējus. Ja nenotiek krustošanās, tiek izveidoti divi pēcnācēji, kas ir precīzas to vecāku kopijas. (Jāņem vērā, ka šeit krustošanās iespējamība tiek definēta kā varbūtība, ka abi vecāki krustosies vienā punktā. Pastāv arī „daudz punktu krustošanās" versijas GA, kur krustošanās iespējamību divu vecāku pārim nosaka punktu skaits, kuros notiks krustošanās.)
    c) Abiem pēcnācējiem tiek veikta mutācija kādos virknes punktos ar varbūtību p_m (mutācijas varbūtība jeb mutācijas iespējamība), un jaunie pēcnācēji tiek pievienoti jaunās paaudzes populācijai. Ja n ir nepāra skaitlis, viens jaunais indivīds var tikt izslēgts no populācijas pēc nejaušības principa.
4) Vecā populācija tiek aizvietota ar jauno populāciju.
5) Atgriežamies pie soļa 2).

Katru šā procesa iterāciju sauc par *paaudzi*. Tipiski GA iterācija tiek atkārtota no 50 līdz 500 paaudzēm vai pat vairāk. Visu paaudžu kopumu sauc par *skrējienu* (*run*). Skrējiena beigās parasti visā populācijā ir viens vai vairāki labi attīstīti indivīdi. Tā kā nejaušības princips spēlē lielu lomu katra skrējiena radīšanas laikā, divi skrējieni ar atšķirīgām nejauši ģenerēto skaitļu kopām, parasti izveidos dažādas uzvedības indivīdu skrējienus.

GA pētnieki dažreiz piedāvā apskatīt arī statistiku (kā piemēram, labākais derīgums, kas tika atrasts skrējienā un paaudze kurā indivīds ar labāko derīgumu tika atrasts) par dažādiem skrējieniem, kas ģenerēti vienas un tās pašas problēmas risināšanai [9].

GA savu darbu beidz tad, kad ir radīti indivīdi, kuru parametri atbilst pieņemama risinājuma nosacījumiem, vai kad ir izpildījies noteikts paaudžu skaits.

Vērts pieminēt, ka GA iniciēšanas fāzē sākotnējo populāciju var veidot gan no nejauši ģenerētiem indivīdiem, gan no mērķtiecīgi ģenerētiem indivīdiem ar iepriekš definētām parametru vērtībām. Pirmais gadījums labāk noder sākot risināt problēmu un vēloties panākt neatkarīgu GA risinājuma meklēšanu, otrais risinājums noder, mēģinot tālāk attīstit jau kādu salīdzinoši labu risinājumu.

Augstāk aprakstītā vienkāršā procedūra raksturo pamatus vairumam izstrādato GA lietojumu. Ir virkne detaļu, kas ir nozīmīgas un nav uzskaitītas, kā, piemēram, populācijas izmērs, un varbūtības krustošanai un mutācijai, un algoritma darbības veiksmīgums bieži lielā mērā ir atkarīgs no šīm detaļām. Protams, tiek izstrādātas arī daudz sarežģītākas GA versijas (piemēram, GA, kas indivīdu reprezantācijai izmanto citu risinājuma, nevis virknes vai GA kam krustošanās un mutācijas operātori krasi atšķiras no šeit aprakstītajām vienkāršajām realizācijām).

Kā detalizētāku piemēru vienkāršam GA, pieņemsim, ka l (virknes garums) ir 8, ka $f(x)$ ir vienāds ar vieninieku skaitu bitu virknē x (ļoti vienkārša derīguma funkcija, kas šeit tiek izmantota tikai ilustratīviem nolūkiem), ka n (populācijas izmērs) ir 4, ka $p_c = 0{,}7$ un ka $p_m = 0{,}001$. (Tā pat kā ar derīguma funkciju, šīs vērtības l un n tika izvēlētas vienkāršības pēc. Tipiskāk l un n vērtības ir robežās un 50 – 1000. Vērtības, kas dotas p_c un p_m, gan ir diezgan tipiskas šiem diviem GA parametriem)

Sākotnējā (nejauši ģenerētā) populācija varētu izskatīties šādi:

2.1. tabula

Nejauši ģenerētā populācija [9]

Indivīda apzīmējums	Indivīda virkne	Derīgums
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

GA atlases metode parasti ir derīguma samērīga atlase, kurā tas cik reizes kāds indivīds tiks pavairots ir vienāds ar tā derīgumu dalītu ar vidējo populācijas derīgumu.

(Tas ir līdzvērtīgi tam, ko biologi sauc par „dzīvotspējas izvēle".)

Vienkārša metode, kā īstenot derīguma samērīgu atlasi ir „ruletes-riteņa iztveršana" (Goldbergs 1989.g.), kas konceptuāli atbilst tam, ka katram indivīdam uz ruletes riteņa tiek piešķirta šķēle, kuras laukuma lielums ir proporcionāls indivīda derīgumam. Ruletes ritenis tiek iegriezts, un tas apstājas uz vienas ķīļveida šķēles, attiecīgais indivīds, kam pieder šī šķēle tiek atlasīts. Ar $n = 4$ piemēru, ruletes ritenis tiktu iegriezts četras reizes, piemēram, pirmajos divos griezienos kā vecāki tiktu izvēlēti indivīdi B un D, un atlikušajos divos griezienos kā vecāki tiktu izvēlēti indivīdi B un C. (Tas, ka A varētu netikt izvēlēts ir tikai laimes spēle. Ja ruletes ritenis tiktu griezts daudzas reizes, tad vidējais rezultāts būtu tuvs izvēlētajām derīguma vērtībām.)

Tiklīdz vecāku pāris ir izvēlēts, ar varbūtību p_c tie tiek sakrustoti, izveidojot divus pēcnācējus. Ja tie varbūtības dēļ netiek sakrustoti, tad pēcnācēji ir precīzas vecāku kopijas. Pieņemsim, ka augstāk minētajā piemērā (tabula 2.1.), vecāki B un D krustojas punktā pēc pirmā bita, tiek radīti pēcnācēji E = 10110100 un F = 01101110, un vecāki B un C nekrustojas, veidojot pēcnācējus kas ir precīzas B un C kopijas. Tālāk katrs pēcnācejs tiek pakļauts mutācijai katrā bita pozīcijā ar varbūtību p_m. Piemēram, pieņemsim, ka pēcnācējs E, tiek mutēts sestajā bita pozīcijā veidojot E' = 10110000, pēcnācēji F un C nemutē vispār, un pēcnācējs B mutē pirmajā bita pozīcijā veidojot B' = 01101110. Jaunā populācija būs sekojoša:

2.2. tabula

Populācijas nākamā paaudze [9]

Indivīda apzīmējums	Indivīda virkne	Derīgums
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Jāatzīmē, ka jaunajā populācijā, lai arī labākā derīguma (ar derīgumu 6) virkne ir zaudēta, vidējais derīgums ir pieaudzis no 12/4 līdz 14/4. Atkārtojot šo procedūru, galu galā tiks iegūta virkne, kura visa sastāvēs no vieniniekiem [9].

Realizējot GA, parasti tiek veidoti aģenti, kam katram ir sava loma. Atlases aģents ir tas, kas nosaka kādā virzienā jāattīstās visai sistēmai, tas nosaka kuri indivīdi un ar kādām īpašībām tiks atlasīti krustošanai nākamās paaudzes iegūšanai, kā arī to, kurus indivīdus ir jāizmet no kopējās sistēmas.

Ģenētisko izmaiņu aģents nosaka to kā tiks pāroti indivīdi, kādas ģenētiskās izmaiņas tiks veiktas. Izmaiņu aģents mutē un krusto indivīdu ģenētisko materiālu.

Svarīgs algoritma atribūts ir indivīdu reprezentācija. Vēsturiski pirmā pielietotā reprezentācija bija binārā virkne (kas visu apzīmē ar 0 un 1), kā bitu masīvs, taču tagad tiek izmantoti arī veselo skaitļu un reālo skaitļu masīvi. Pastāv arī kokveida reprezantācijas, taču tādas ir grūtāk realizējamas, kā arī sarežģītak realizet atlases, krustošanās un mutācijas algoritmus.

Ģenētiskie algoritmi kopā ar ģenētisko programmēšana (GP) ir vienas no galvenajām klasēm ģenētisko un evolucionārās skaitļošanas metodoloģijām [10].

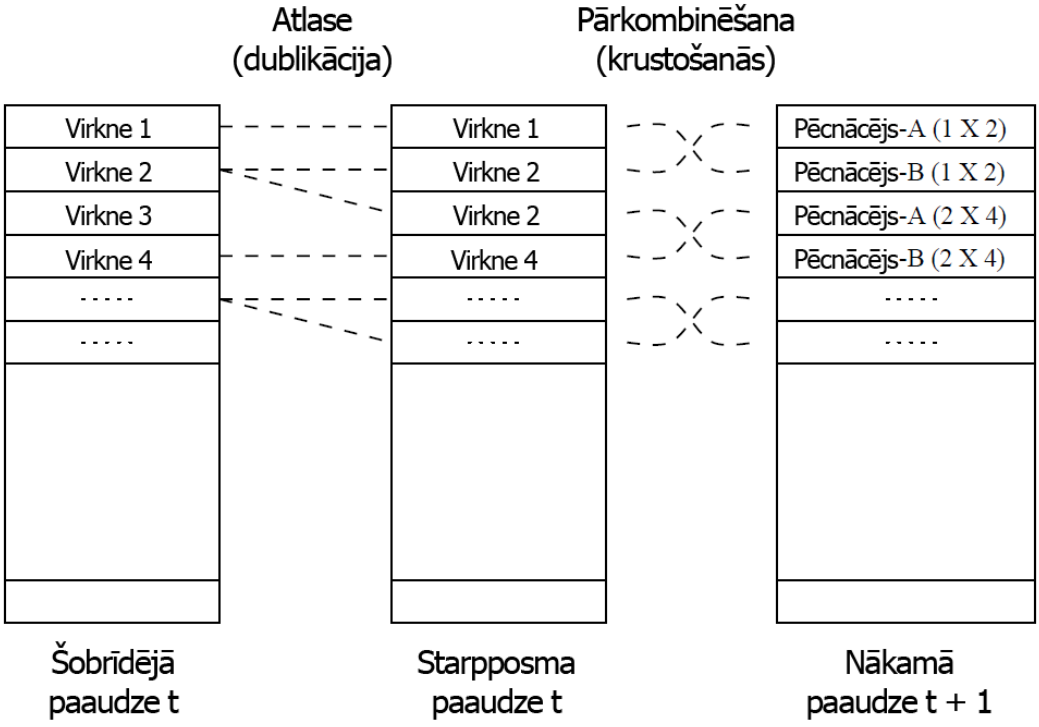
2.2. Mantošana

Iedzimtība ir vecāku pazīmju nodošana saviem pēcnācējiem (no vecākiem vai senčiem). Tas ir process kurā pēcnācēja šūnas vai organisms iegūst īpašības vai noslieci uz vecāku šūnām vai organisma īpašībām. Ar iedzimtību un dažādām variācijām, kas attīstās indivīdos, var attīstīties kādas noteiktas sugas izveidošanās. Ģenētisko algorimtu kontekstā, iedzimtība nodrošina to, kas krustošanās reuzltātā, vecāku parametri, pāriet bērniem [11].

2.3. Atlase

Viena no ģenētisko algoritmu stadijām ir atlase, šīs stadijas laikā no pašreizējās populācijas tiek atlasīti indivīdi, no kuriem tiek izveidota starpposma populācija. Tad pārkombinēšanas un krustošanas stadijās tiek izmantota starpposma populācija, lai radītu nākamās paaudzes populāciju [12].

Atlases stadijas laikā pašreizējā paaudze tiek dalīta priekš diviem stāvokļiem – atlases stadijas un pārkombinēšanas stadijas. Kā redzams attēlā 2.1, izvēlētie indivīdi tiek ielikti blakus esošajos slotos, lai pēc tam tos pa pāriem izmantoto nākamās paaudzes ģenerēšanai (pārkombinēšana un krustošana). Vēl labāka pieeja ir atlasītos indivīdus slots salikt sajauktā secībā, tādejādi panākot vēl lielāku nejaušības principu nākamās paaudzes ģenerēšanas laikā. Pēc krustošanas stadijas indivīdiem var tik izmantota mutācija.



2.1. att. **Ģēnētiskie algoritmi, atlase un krustošanās [12]**

Vispārīgi atlases procedūra varētu tikt īstenoti sekojoši:

1) Derīguma funkcija tiek piemērota katram populācijas indivīdam, iegūstot katra indivīda derīguma vērtību, kas pēc tam tiek normalizēta. Normalizācija nozīmē dalīt katra indivīda derīguma vērtību ar kopējo summu no visām populācijas derīguma vērtībām, kā rezultātā visu indivīdu derīguma vērtību kopējā summa ir vienāda ar 1.
2) Populācijas indivīdi tiek sakārtoti pēc derīguma vērtības dilstošā secība.
3) Tiek aprēķinātas uzkrātās normalizētās derīguma vērtības (viena indivīda uzkrātā derīguma vērtība ir summa, kas sastāv no paša indivīda derīguma vērtības, kā arī visu iepriekšējo indivīdu derīguma vērtībām). Pēdējā indivīda uzkrātajai derīguma vertībai jābūt vienādai ar 1 (citādi varam secināt, ka normalizācijas procesā kaut kas ir aprēķināts nepareizi).
4) Pēc nejaušības principa tiek izvēlēts skaitlis R, kura vērtība ir starp 0 un 1.
5) Tiek izvēlēts pirmais indivīds pēc kārtas, kura uzkrātā normalizētā derīguma vērtība ir lielāka kā R.

Ja šī procedūra tiek atkārtota, kamēr ir pietiekami daudz nepieciešamie indivīdi, tad šī atlases metode tiek saukta par derīguma samērīgu atlasi vai rulletes-riteņa atlasi. Ja tā vietā, lai atkārtotu procedūru vairākas reizes, mēs vienā reizē izvēlamies vairākas nejaušas vērtības, un tad ņemam pirmos labākos indivīdus virs katras vērtības, šo metodi sauc par stohastisko universiālo atlasi. Atkārtoti izvēloties labāko indivīdu no nejauši izvēlētas apakškopas sauc par turnīra atlasi. Metodi, kad tiek atlasīti labākie 50%, 30% vai citas proporcijas indivīdi, sauc par griešanas metodi, tomēr šī metodes trūkums ir tāds, ka var tikt zaudēti vājāki indivīdi, kuriem kādas noteiktas īpašības ir ļoti spēcīgas.

Pastāv arī citi atlases algoritmi, kas priekš atlases iekļauj tikai tos indivīdus, kuriem derīguma vērtība ir augstāka, kā dotā (patvaļīgi izvēlēta) konstante. Citi atlases algoritmi indivīdus atlasa no ierobežotas apakškopas, kur tikai zināma procentuāla daļa indivīdu tiek iekļauti, balstoties uz to derīguma vērtību.

Gadījumu, kad vienas paaudzes labākie indivīdi nākamajai paaudzei tiek nodoti nemainīti, sauc par elitismu vai elitāro atlasi. Tas dažreiz var būt veiksmīgs variants ko izmantot, lai konstruētu nākamās paaudzes jauno populāciju [13].

2.4. Krustošanās

Ģenētiskajos algoritmos krustošanās ir operātors, ko izmanto, lai katrs nākamās paaudzes indivīds vai indivīdi būtu atšķirīgi no iepriekšējās paaudzes indivīdiem, tomēr reizē saturētu daļu tā īpašību. Tas ir analoģisks reprodukcijai un bioloģiskajai krustošanai, uz kuras balstās ģenētiskais algoritms. Krustošānās ir process, kurā tiek izmantots vairāk kā viena vecāka risinājums, kā rezultātā tiek iegūts bērna risinājums no tiem. Ir vairākas metodes kā izvēlēties indivīdus krustošanai, zemāk tās ir aprakstītas

Dažas metodes kā atlasīt indivīdus krustošanai:

- Ruletes-riteņa atlase (zināma arī kā derīguma vērtības atlase)
- Boltzmanna atlase
- Turnīra atlase
- Ranga atlase
- Stabilā stāvokļa atlase

Eksistē daudzi krustošanās paņēmienu risinājumi, kas izmanto dažādas datu struktūras, lai saglabātu informāciju par indivīdu. Zemāk aprakstīti daži risinājumi.

Krustošanās metodes:

- Viena punkta krustošanās

  Abiem vecākiem tiek izvēlēts viens un tas pats krustošanās punkts. Visa informācija vecāku indivīdos, kas atrodas aiz izvēlētā punkta, abos jaunajos indivīdos tiek apmainīta ar otra indivīda informāciju. Iegūtie indivīdi ir bērni.

2.2. att. **Viena punkta krustošanās [14]**

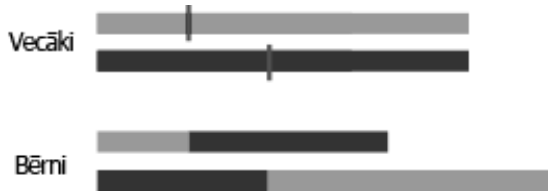
- Divu punktu krustošanās

  Divu punktu krustošanās gadījumā, tiek izvēlēti divi punkti uz vecāku indivīdiem. Viss kas atrodas starp šiem diviem punktiem, vecāku indivīdos tiek apmainīts vietām, izveidojot divus bērnu indivīdus:

2.3. att. **Divu punktu krustošanās [14]**

- „Sagriezt un savienot”

Vēl viena krustošanās pieeja „sagriezt un savienot”, izmaina bērnu indivīdu aprakstošās virknes garumu. Iemesls šādai atšķirībai ir tāds, ka katram vecāka indivīdam tiek izvēlēts cits krustošanās punkts.

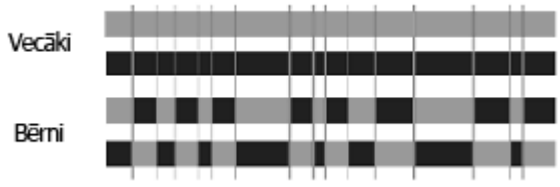


2.4. att. **„Sagriezt un savienot” krustošanās [14]**

- Vienmērīgā (*uniform*) krustošanās un pusvienmērīgā krustošanās

Vienmērīgā krustošanās izmanto konstanta lieluma sajaukšanas attiecību starp diviem vecākiem. Atšķirībā no viena un divu punktu krustošanās, vienmērīgā krustošanās ļauj vecāku indivīdiem sniegt ieguldījumu gēnu (parametru) līmenī, nevis tikai lielu segmentu.

Ja sajaukšanas attiecība ir 0,5, tad pēcnācējam aptuveni puse gēnu ir no viena vecāka un otra puse gēnu no otra vecāka, krustošanās punkti var tikt nejauši izvēlēti kā redzams attēlā 2.5.



2.5. att. **Vienmērīgā krustošanās [14]**

Vienmērīgā krustošanās novērtē katru vecāka virknes bitu apmaiņai ar varbūtību 0,5. Kaut arī vienmērīgā krustošanās ir slikta metode, empīriski pētījumi liecina, ka tā ir daudz izpētītāka pieeja krustošanai, kā tradicionālā ekspluatējošā pieeja, kas saglabā ilgāku slikto indivīdu kopu. Tas noved pie daudz pilnīgākas gēnu

sajaukšanas, saglabājot labās informācijas apmaiņu. Diemžēl nepastāv neviena pietiekama teorija, kas izskaidrotu atšķirības starp vienmērīgo krustošanos un tradicionālo pieeju [15].

Vienmērīgās krustošanās shēmā atsevišķi biti virknē tiek salīdzināti starp abiem vecākiem. Biti tiek apmainīti vietām ar fiksētu varbūtību, tipiski 0,5.

Pusvienmērīgās krustošanās shēmā, tieši puse no nevienādajiem bitiem tiek apmainīti vietām. Tādējādi vispirms tiek aprēķināts Hamminga attālums (atšķirīgo bitu skaits). Šis skaits tiek dalīts ar divi. Iegūtais skaitlis nosaka skaitu, cik daudz nevienādi biti abiem vecākiem tiks apmainīti vietām.

- Trīs vecāku krustošanās

Šajā pieejā bērns tiek iegūts no trim vecākiem, kas tiek nejauši izvēlēti. Katrs bits no pirmā vecāka tiek salīdzināts ar otrā vecāka attiecīgo bitu. Ja biti ir vienādi, tie tiek ņemti pēcnācējam, citādi pēcnācēja bits tiek ņemts no trešā vecāka.
Vecāks_1 110100010
Vecāks_2 011001001
Vecāks_3 110110101
Pēcnācējs 110100001 [16].

2.5. Mutācija

Ģenētiskajos algoritmos, mutācija ir ģenētisks operātors, ko izmanto, lai saglabātu ģenētisko daudzveidību no pašreizējās paaudzes populācijas uz nākamās paaudzes populāciju. Tas ir analogs bioloģiskajai mutācijai. Mutācija izmaina vienu vai vairāku indivīda parametru vērtības, no to sākotnējā stāvokļa. Mutācijas operātora ietekmē, rezultāts var pavisam mainīties, no tā kāds tiktu iegūts, ja mutācijas netiktu pielietotas. Līdz ar to ģenētiskais algoritms var nonākt pie labāka risinājuma izmantojot mutāciju. Mutācija notiek evolūcijas procesa laikā, ņemot vērā lietotāja definētu mutācijas izpildīšanās varbūtību. Šo varbūtību būtu ieteicams noteikt zemu. Ja šī varbūtība būs noteikta par augstu, ģenētiskais algoritms kļūs par primitīvu uz nejaušības principu balstītu algoritmu.

Klasisks mutācijas operātora piemērs ietver varbūtību, ka kāds patvaļīgi izvēlēts bits ģenētiskajā rindā, tiks mainīts no tā sākotnējā stāvokļa. Zināma metode mutācijas operātora

īstenošanai ietver nejauša mainīga uzģenerēšanu katram rindas bitam (ja indivīdu apraksta ar bitu rindu). Šī nejauši izvēlētā vērtība nosaka to vai konkrētais bits tiks mainīts vai nē. Šādu mutācijas procedūru, kas balstīta uz bioloģisko mutāciju, sauc par viena punkta mutāciju. Cita veida mutācijas ir apgriešanas un peldošā punkta mutācija. Kad indivīda parametru aprakstīšana ir ierobežota kā permutācijas problēmās, mutācijas ir mijmaiņas, apgriešanas un sajaukumi.

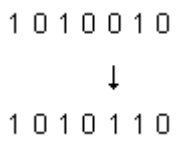
Mutācijas mērķis ģenētisko algoritmu izstrādāšanā, ir ieviest dažādības saglabāšanos indivīdos paaudžu maiņas gaitā. Mutāciju izmantošanai vajadzētu uzlabot ģenētiskā algoritma efektivitāti, samazinot iespējamību, ka populācijas indivīdi kļūst pārāk līdzīgi viens otram, tādējādi palēninot vai pat apstādinot evolūciju. Šis iemesls arī izskaidro faktu, kāpēc lielākā daļa ģenētisko algoritmu sistēmas izvairās izmantot tikai derīguma funkciju, lai atlasītu indivīdus nākamās paaudzes radīšanai, bet drīzāk izmanto nejaušu (vai daļēji nejaušu) atlasi, piemērojot kāda veida koeficientu pret tiem indivīdiem, kas ir virs vidējā derīguma [17].

Dažādiem indivīdu kodēšanas veidiem ir piemēroti dažādi mutāciju tipi:

- Bitu virknes mutācija

  Bitu virknes mutāciju nodrošina bitu apgriešana nejauši izvēlētā pozīcijā vai pozīcijās.

  Piemērs:



2.6. att. **Bitu virknes mutācija** [18]

  Varbūtība, ka indivīda viens bits mutēs ir 1/L, kur L binārā vektora garums. Tādējādi tiek sasniegts, ka mutācijas iespējamība ir 1 uz katru mutāciju un izvēlēto indivīdu.

- Bitu inversija

  Šis mutācijas operators izvēlētajam indivīdam invertē visus bitus (ja indivīda bits ir 1, tas tiek mainīts uz 0 un otrādāk).

- Robežas

Šis mutācijas operators aizstāj indivīda zemākās vai augstākās robežas vērtības ar nejauši izvēlētām vērtībām. To var izmantot veselo skaitļu un peldošā komata skaitļu indivīdu kodēšanas gadījumā.

- Nevienmērīgā

Varbūtība, ka daļa no mutācijām tieksies uz 0 nākamajās paaudzēs, tiek palielināta izmantojot nevienmērīgas mutācijas operatoru. Tas attur populāciju no stagnācijas sākumposma attīstībā. Tas uztur labāku rezultātu evolūcijas procesam nākamajās paaudzēs. To var izmantot veselo skaitļu un peldošā komata skaitļu indivīdu kodēšanas gadījumā.

- Vienmērīgā

Šis operators aizstāj izvēlētā indivīda parametra vērtību ar vienmērīgu nejaušas izvēles vērtību, kas tiek izvēlēta starp lietotāja noteikto augstākās un zemākās robežas vērtību izvēlētajam indivīdam. To var izmantot veselo skaitļu un peldošā komata skaitļu indivīdu kodēšanas gadījumā.

- Gausa

Šis operators pievieno Gausa izplatītās izlases vērtību izvēlētajam indivīdam. Ja šī vērtība neietilpst starp lietotāja noteikto augstākās un zemākās robežas vērtību izvēlētajam indivīdam, jauna vērtība tiek atgriezta. To var izmantot veselo skaitļu un peldošā komata skaitļu indivīdu kodēšanas gadījumā [18].

3. IZSTRĀDĀTĀ DAUDZAĢENTU SISTĒMA

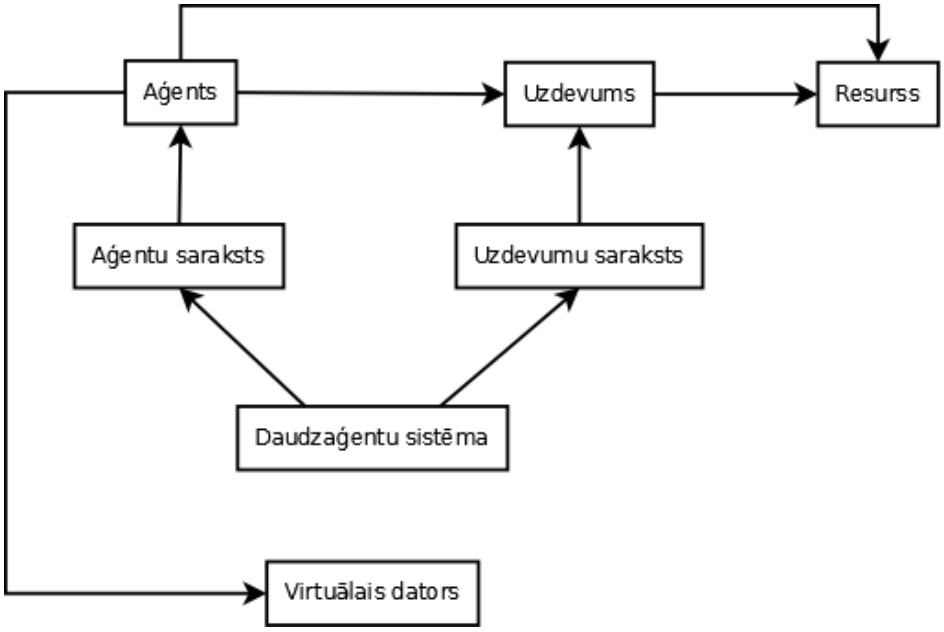
Izstrādātās daudzaģentu sistēmas pirmkods ir rakstīts programmēšanas valodā Python. Valoda Python tika izmantota, jo tā ir labi zināma darbam autoram, kā arī viena no valodām, kurā ir pieejama nepieciešamā funkcionalitāte, kas nepieciešama darba izstrādes laikā.

Programmas izstrādes laikā darba autors sadalīja uzdevumu piecos moduļos

- Pirmais modulis ir galvenais modulis, kurā tiek definētas visas galvenās konstantes daudzaģentu sistēmas darbībai, tiek uzģenerēts izpildāmo uzdevumu saraksts, kā arī aģentu saraksts. Tālāk izmantojot vienu un to pašu izpildāmo uzdevumu un aģentu sarakstu tiek izpildīta gan normāla daudzaģentu sistēma, gan noteiktu skaitu tiek ģenerētas jaunas aģentu paaudzes, kur katra jauno aģentu paaudze atkal izpilda uzdevumus no izpildāmo uzdevuma saraksta.
- Otrais modulis ir aģenta klases definēšanas modulis, šajā modulī ir definēti aģenta klases atribūti, kā arī pavediena (*thread*) izsaucošā funkcija un tās darbība.
- Trešais modulis ir virtuālā datora klase definēšanas modulis, šajā modulī ir definēti virtuālā datora klases atribūti, kā arī klases funkcijas resursu noslogošanai un atslogošanai.
- Ceturtais modulis ir uzdevuma klase definēšanas modulis, kur tiek definēti uzdevuma klases atribūti.
- Piektais modulis ir resursu klases definēšanas modulis, kur tiek definēti resursu klases atribūti, kā arī resursu salīdzināšanas funkcija.

3.1. Vienkārša daudzaģentu sistēma

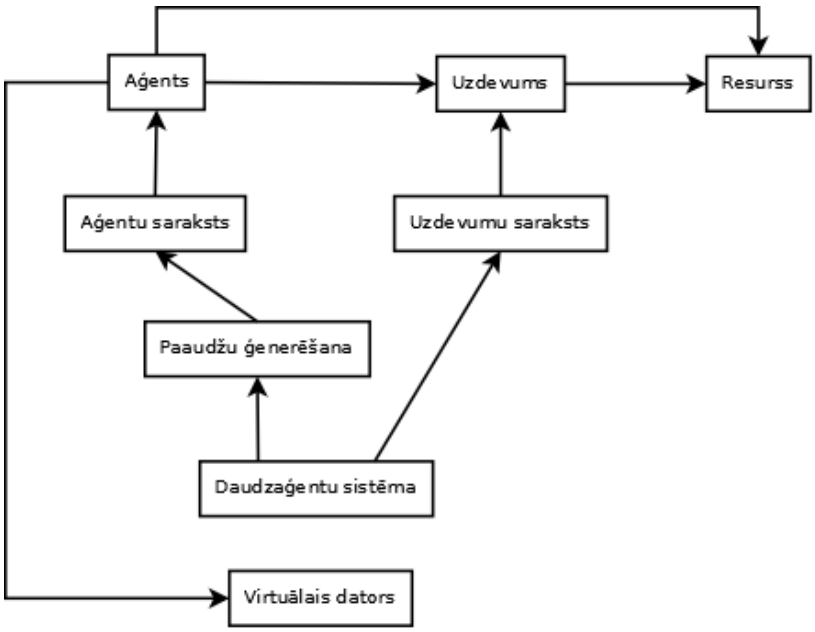
Attēlā 3.1. redzama izstrādātas daudzaģentu sistēmas klašu savstarpējā attiecība. Visa pamatā ir daudzaģentu sistēma, kurā tiek uzģenerēts aģentu saraksts, kas sastāv no aģentiem un uzdevumu saraksts, kas sastāv no uzdevumiem. Programmatūras izpildes laikā aģenti no uzdevumu saraksta ņem uzdevumus. Katrs uzdevums var noslogot noteiktus resursus, ko atspoguļo uzdevuma un resursu attiecība attēlā. Aģents savukārt atkarībā no tā vai ir brīvi pieejami resursi un no tā vai paša aģenta resursu parametri atļauj, sāk uzdevumu izpildi un noslogo virtuālā datora resursus. Kad visi uzdevumi no uzdevuma saraksta ir izpildīgi, aģenti pārstāj savu eksistenci jeb beidzas pavedienu darbība.



3.1. att. **Daudzaģentu sistēmas modulis**

3.2. Daudzaģentu sistēma ar GA

Attēlā 3.2. redzama izstrādātā daudzaģentu sistēma, kurā tiek izmantota ģenētisko algoritmu pieeja. Kā redzams pēc attēla, tad mainās tas, ka aģentu saraksts ir atkarīgs no paaudžu ģenerēšanas rezultāta, kamēr uzdevumi paliek tie paši. Šādas programmatūras arhitektūras mērķis pārbaudīt to vai GA izmantošana to pašu uzdevumu veikšanai daudzu paaudžu ģenerēšanas laikā, uzlabo uzdevumu izpildes ātrumu ar katru nākamo paaudzi.



3.2. att. **Daudzaģentu sistēmas modulis ar GA**

3.3. Pirmais modulis, daudzaģentu sistēma un GA

Pielikumā (1. pielikums. Pirmais modulis) redzams pirmā moduļa pirmkods.

3.3.1. Daudzaģentu programmatūras inicializācija

Pirmais modulis ir galvenais modulis kurā notiek visas programmatūras izpildes sākšana. Pirmajā modulī tiek definētas tādas konstantes, kā uzdevumu skaits, aģentu skaits, labako aģentu skaits (skaits aģentu, kas tiks ņemti uz nākamo populāciju nemainīti, kā arī skaits labako aģentu, kas tiks ņemti krustošanai), konstantes maksimālajam un minimālajam uzdevumu izpildes laikam, procesora noslodzei, atmiņai noslodzei, tīkla kartes noslodzei un cietā diska noslodzei. Pastāv arī konstantes, kas maina cik reizes tiks veikti cikli kuros notiks viena uzdevuma saraksta iztukšošana, maksimālais ģenētiskā algoritma paaudžu skaits vienā ciklā, kā arī konstante, kas nosaka vienas paaudzes ietvaros izmantojamo uzdevumu skaitu no uzdevuma saraksta.

Programmatūras darbības laikā tiek izveidots uzdevumu saraksts, kas sastāv no uzdevumiem, kurā visu datora resursu noslogošanas parametri tiek uzģenerēti pēc nejaušības principa no minimālajām un maksimālajām konstantēm, kā arī uzdevuma veikšanas ilgums tike uzģenerēts pēc nejaušības principa no minimālās un maksimālās uzdevuma veikšanas konstantes.

Tālāk notiek aģentu saraksta ģenerēšana, kas sastāv no aģentiem, kuri satur resursu noslodzes objektu, kura atribūti tiek aizpildīti pēc nejaušības principa lietojot iepriekš definētās maksimālās un minimālās konstantes.

Kad mums ir iegūti abi saraksti, varam sākt parastas daudzaģentu sistēmas programmas izpildi, izpildes laikā, katrs no sarakstā pievienotajiem aģentiem tiek palaists kā jauns pavediens, kas tālāk pats no kopējā uzdevuma saraksta mēģina ņemt un izpildīt uzdevumus, kamēr kopējais uzdevumu saraksts nav tukšs. Kad kopējais uzdevumu saraksts ir tukšs, visi aģenti beidz savu pavediena dzīves ciklu un mēs varam izmērīt laiku, cik ilgi izpildījās parasta daudzaģentu sistēma.

3.3.2. Ģenētiskā algoritma pielietošana

Tālāk programmatūrā seko tā paša uzģenerēto uzdevuma saraksta izmantošana, un pirmajā paaudzē to pašu uzģenerēto aģentu saraksta izmantošana, kas programmatūras daļā bez GA.

Programmatūra izpildās lietotāja noteikto maksimālo ciklu skaitu ilgi – tas ir nepieciešams, ja lietotājam ir vēlme iegūt vairāk dažādus variantus no sākotnējā uzdevumu saraksta un aģentu saraksta.

Katra cikla iekšienē, notiek cikls, kas izpildās tik daudz reizes, cik lietotāja noteiktais paaudžu skaits. Lietotājs var izvēlēties, vai katrai paaudzei pielietot pilnīgi visu uzdevumu sarakstu, vai katrai paaudzei ņemt mazāku, bet citu daļu no uzdevuma saraksta, kas neatkārtojas.

3.3.2.1. Atlase un derīguma funkcija

Atlases funkcijai tiek izmantots griešanas atlases un elitārās atlases paveids – daļa aģentu krustošanai tiek atlasīti paņemot noteiktu daļu labāko (griešanas atlase), bet daļa paņemot labāko daļu no visas populācijas (elitārā atlase).

Derīguma funkcija ir realizēta tā, ka katram aģentam ir atribūts, kurā tiek uzskaitīta katra uzdevuma izpildes sākšanas gaidīšanas ilgums un kopējais paveikto uzdevumu skaits. Aģenta gaidīšanas ilgums rodas sekojoši

1) Aģents virtuālajam datoram pieprasa aizņemt vietu priekš uzdevuma izpildei nepieciešamajiem resursiem un šajā brīdī notiek pārbaude vai aģenta noteiktās datora resursu sliekšņa konstantes ir mazākas vai vienādas ar virtuālajā datorām šobrīd pieejamajiem resursiem, ja nav, virtuālais dators atgriež vērtību 1, un

aģenta pavediens uz noteiktu laiku aizmieg, kā arī šis laiks tiek pieskaitīts pie aģenta kopējā uzdevumuma izpildes sākšanas gaidīšanas ilguma.

2) Aģents virtuālajam datoram pieprasa aizņemt vietu priekš uzdevuma izpildei nepieciešamajiem resursiem un šajā brīdī notiek pārbaude vai uzdevuma veikšanai nepieciešamās datora resursu vērtības ir mazākas vai vienādas ar šobrīd virtuālajā datorā pieejamajiem resursiem, ja nav, virtuālais dators atgriež vērtību 2, un aģenta pavediens uz noteiktu laiku aizmieg, kā arī šis laiks tiek pieskaitīts pie aģenta kopējā uzdevumuma izpildes sākšanas gaidīšanas ilguma.

Tātad šis gaidīšanas laiks tiek saskaitīts kopā un izdalīts ar kopējo paveikto uzdevumu skaitu, tādējadi iegūstot vidējo aģenta uzdevuma izpildes sākšanas gaidīšanas ilgumu, tā ir aģenta derīguma vērtība.

Visi aģenti tiek sakārtoti sarakstā pēc aģenta derīguma vērtības un puse labāko aģentu resursu objekti (skaits tiek noteikts ar programmā definētu konstanti), tiek atlasīti krustošanai un mutācijai kā vecāki, kā arī otrai pusei jauno aģentu, kas tiks saglabāti uz nākamo paaudzi kā pēcnācēji.

3.3.2.2. Krustošanās

Krustošanās ir realizēta sekojoši – no vecāku resursa klases objektiem, kas tiek atlasīti krustošanai, visi resursu objekti tiek sadalīti pa pāriem, un ar varbūtību 50%, tālāk notiek katra resursa objekta parametra apmaiņa ar otru resura objekta parametru, bieži vien šāda krustošanās tiek saukta par vienmērīgo krustošanos.. Tiek ģenerēti skaitļi robežas no 0 līdz 99, ja skaitlis ir mazāks par 50, tad resursu objektu parametra vērtības netiek apmainītas, savukārt, ja skaitlis ir lielāks vai vienāds ar 50, tad resursu objektu parametra vērtības tiek apmainītas starp abiem resursa objekta vecākiem. Gala rezultātā tiek iegūti divi resursa objekta pēcnācēji, kuriem kāda vai neviena resursu objekta parametru vērtība ir apmainīta vietām un visi resursa objektu pēcnācēji tike pievienoti jaunās paaudzes resursu objektu populācijas sarakstam

3.3.2.3. Mutācija

Mutācija notiek sekojoši – katram jaunā bērna resursa parametram, tiek ģenerēts nejaušs skaitlis robežas no 0 līdz 99, ja šis skaitlis ir mazaks par 5, tad tiek ģenerēts nejaušs skaitlis robežās no -3 līdz 3 un ja pieskaitot šo skaitli, jaunā resursa klases objekta parametra vērtība nav mazāka par 0 un lielāka par 100, tad šis skaitlis tiek pieskaitīts un ir notikusi mutācija. Šāda veida mutāciju var raksturot arī kā Gausa veida mutāciju.

3.4. Otrais modulis, aģentu klase

Pielikumā (2. pielikums. Otrais modulis) redzams otrā moduļa pirmkods.

Otrajā modulī tiek definēta aģenta klase, ar atribūtiem, kas satur aģenta identifikātoru, referenci uz virtuālo datoru, referenci uz uzdevumu sarakstu, aģenta resursa objektu, kas parāda aģenta individuālās vērtības, ar cik brīviem virtuāla datora resursiem, aģents vispār uzsāks darbu, kā arī aģenta izpildīto uzdevumu skaitu un katra uzdevuma izpildei patērēto gaidīšanas laiku.

Izveidojot aģenta objektu, aģents tiek izveidots un palaists kā jauns paveidiens, kas darbojas tik ilgi, kamēr uzdevuma saraksts nav tukšs. Šādi tiek izmantots paralēlās programmēšanas princips, kas nodrošina to, ka visi aģenti var vienlaicīgi izpildīt uzdevumus no uzdevumu saraksta, un tiem nav jāgaida rindā, kamēr cits aģents būs pabeidzis savu uzdevumu. Šāda pieeja nodrošina arī daudzkodola procesora visu kodolu izmantošanu.

Tātad aģents uzsākot savu dzīves ciklu, no uzdevuma saraksta mēģina paņemt uzdevumu, ja tas izdodas, tad aģents mēģina izpildīt savu uzdevumu, virtuālajam datoram prasot aizņemt nepieciešamos resursus, ja virtuālais dators resursus var aizņemt, tad aģents aizmieg tik ilgi, cik nepieciešams izpildīt uzdevumu, kad uzdevums ir izpildīts, aģents virtuālajam datoram pasaka, ka tas var atkal atbrīvot visus resursus – šādi tiek simulēta dažādu datora resursu noslodze dažados laika momentos. Ja aģents savu ierobežoto resursu dēļ vai virtuālā datora resursu dēļ nevar sākt uzdevuma izpildi, tad dators aizmieg uz lietotāja noteikto laiku, un šī uzdevuma izpildes sākšanai nepieciešamais laiks tiek palielināts par lietotāja noteikto laiku.

3.5. Trešais modulis, virtuālā datora klase

Pielikumā (3. pielikums. Trešais modulis) redzams trešā moduļa pirmkods.

Virtuāla datora klase sastāv no diviem atribūtiem – viens ir resursu objekts, kas apraksta to cik daudz virtuālā datora resursi ir aizņemti konkrētajā laika momentā, otrs savukārt ir pavedienu bloķēšanas atribūts, lai vairāki pavedieni vienlaicīgi nevarētu mainīt vienu un to pašu virtuālas datora klases atribūta vērtību.

Virtuālajai datora klasei arī ir divas funkcijas, kur pirmā saņemot divus resursa objektus, salīdzina tos savstarpēji, ja pirmā resursa objekta visi parametri ir mazāki vai vienādi par otra resursa objekta parametriem, notiek salīdzināšana vai otrā resursa parametri ir mazāki vai vienādi ar virtuāla datora resura parametriem, un ja ir, tad virtuālā datora resursa parametri tiek samazināti par attiecīgām otrā resursa objekta parametru vērtībām.

Otrā funkcija saņemot resursa objektu kā parametru, palielina visas virtuālā datora resursa objekta vērtības par saņemtā resursa objekta vērtībām.

3.6. Ceturtais modulis, uzdevumu klase

Pielikumā (4. pielikums. Ceturtais modulis) redzams ceturtā moduļa pirmkods.

Uzdevuma klasei ir trīs atribūti, kur pirmais ir uzdevuma identifikātors, otrais resursu klases objekts, kas satur uzdevuma noslodzes katram resursa parametram, trešais ir uzdevuma izpildes laiks

3.7. Piektais modulis, resursu klase

Pielikumā (5. pielikums. Piektais modulis) redzams piektā moduļa pirmkods.

Resursu klasei ir četri atribūti, kur pirmais apzīmē procesora noslodzi, otrais atmiņas noslodzi, trešais tīkla kartes noslodzi un ceturtais cietā diska noslodzi.

Piektā moduļa pirmkodā ir definēta arī resursu salīdzināšanas funkcija, kas atgriež patiesu vērtību tad un tikai tad, ja visi funkcijai padotā pirmā resursa objekta parametri ir mazāki vai vienādi ar funkcijai padotā otrā resursa objekta parametra vērtībām.

4. DAUDZAĢENTU SISTĒMU SALĪDZINĀJUMS

Izstrādātā programmatūra tika testētā daudz reizes ar dažādiem parametriem (maksimālie uzdevumi, maksimālais aģentu skaits, labāko aģentu konstante, dažādi minimālie un maksimālie parametri resursiem, utt.).

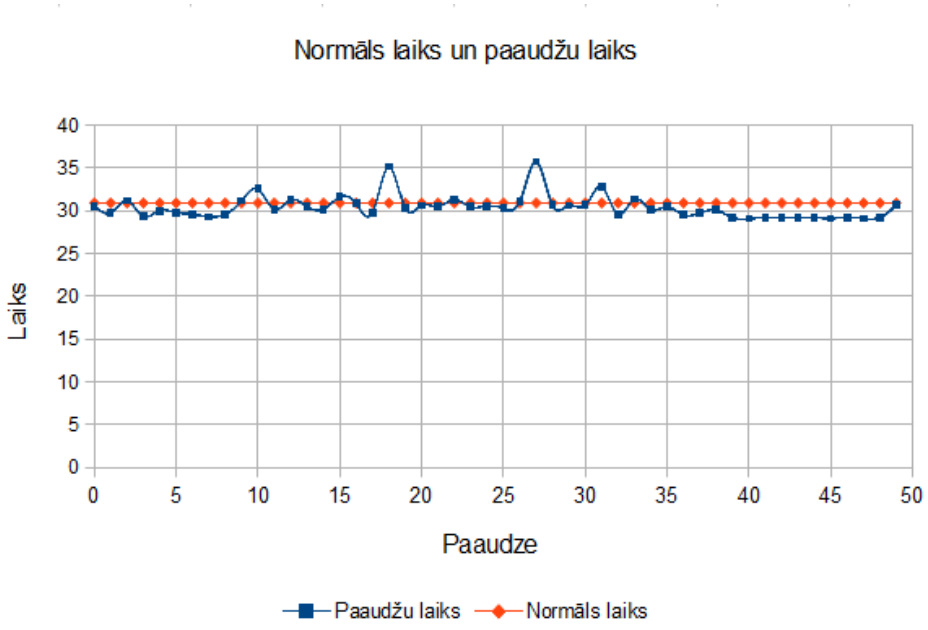
Testējot gan parasto daudzaģentu sistēmu, gan daudzaģentu sistēmu ar GA, tika izmantota viena un tā pati uzģenerētā uzdevumu un aģentu kopa, lai testēšana tiktu veikta uz vieniem un tiem pašiem ieejas datiem.

Novērotie secinājumi ir tādi, ka ar parametriem, kad kopējais veicamo uzdevumu skaits ir mazs (aptuveni 1000), aģenti ir optimāli (no literatūras) daudz (100) un labāko aģentu paņemšana ir puse (50), gan parastā daudzaģentu sistēma, gan daudzaģentu sistēma ar ģenētiskā algoritma pielietošanu izpildās līdzīgos laikos (atšķirības ir sekundes simtdaļu robežās). Jāpiebilst ka ar līdzīgu laiku, tiek domāts laiks, kas nepieciešams no visu aģentu pavedienu palaišanas, līdz visu pavedienu izpildes beigām, gan parastajā, gan GA ciklā – gadījumā, kur tiek izmantoti GA, un notiek iterācija pa paaudzēm, izpildes laikā netika skaitīts iekšā laiks ko prasa GA izpilde, t.i., atlases, krustošānās un mutācijas operātori.

Tātad šādos gadījumos redzams, ka daudzaģentu sistēma ar GA nav īpaši ātrāka, citreiz tā pat ir lēnāka, viss ir atkarīgs no tā, kādas vērtības tiek saģenerētas visās vietās kur tiek izmantots nejaušības princips jaunu vērtību iegūšanai.

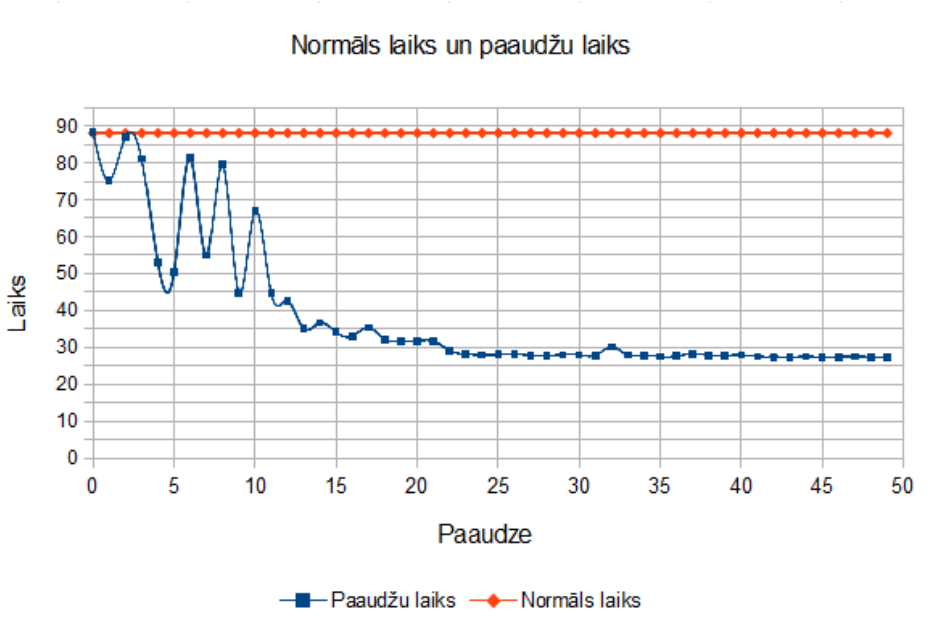
Tas kāpēc daudzaģentu sistēma ar mazu uzdevumu, pielietojot GA nav īpaši ātrāka, skaidrojams arī ar to, ka katrai paaudzes aģenti vidēji veikuši diezgan mazu uzdevumu skaitu, tāpēc, atšķirība starp labākajiem paaudzes aģentiem un sliktākajiem paaudzes aģentiem ir daudz mazāka, nekā tad, ja uzdevumu skaits ir ļoti liels.

Kā redzams attēlā 4.1, daudzaģentu sistēmā, kas izmanto GA, ir reizes, kad izpildes laiks dažreiz ir lielāks, kā parastajā daudzaģentu sistēmā – tas izskaidrojams ar to, ka citreiz var notikt ļoti neprognozējamas krustošanās vai mutācijas, un ka, piemēram, mutācijas atļaujošo konstanti vajadzētu samazināt.



4.1. att. **Daudzaģentu sistēmas izpildes laika grafiks ar maz uzdevumiem**

Savukārt, piemēram, ar parametriem, kad kopējais veicamo uzdevumu skaits ir liels (aptuveni 100000), aģenti ir optimāli daudz (100) un labāko aģentu paņemšana ir puse (50), kā redzams attēlā 4.2., var redzēt ļoti strauju kopējo uzdevumu izpildes laiku samazināšanos līdz ar katru nākamo paaudzi.



4.2. att. **Daudzaģentu sistēmas izpildes laika grafiks ar daudz uzdevumiem**

Attēlā 4.2. redzamais skaidrojams ar to, ka katras paaudzes 100 aģentiem, ir ļoti daudz veicamie uzdevumi, līdz ar to daudz precīzāk un izteiktāk veidojas atšķirība, starp labākajiem un sliktākajiem aģentiem, t.i, to derīguma vērtībām.

REZULTĀTI

Bakalaura darba mērķis bija izstrādāt daudzaģentu sistēmu, tās izstrādes laikā autors pētot grāmatas, zinātniskos rakstus un interneta resursus, iepazinās ar daudzaģentu sistēmu izmantošanu, paradigmām un izstrādes metodoloģiju, padziļināti iepazinās ar literatūru par ģenētisko algoritmu paradigmām un to realizācijas iespējām, kā arī papildus apguva programmēšanas valodu Python.

Darbā plaši aprakstīta gan daudzaģentu sistēmu būtība, sīki pētot to, kādas ir to īpašības, gan arī aprakstītas klasiskas ģenētisko algoritmu realizācijas, to operātori – atlase, krustošanās, mutācija.

Darba rezultātā, autors valodā Python ir izstrādājis daudzaģentu sistēmu, kas sastāv no pieciem moduļiem, pats izplānojis tās arhitektūru un realizāciju. Izstrādātā daudzaģentu sistēma izpildās gan kā parasta daudzaģentu sistēma, gan kā daudzaģentu sistēma, kuras aģentiem tiek pielietotas ģenētiskajos algoritmos aprakstītās darbības. Programmatūras izpildes rezultāts ļauj salīdzināt izpildes ātrdarbību šīm abām daudzaģentu sistēmām, izmantojot vienus un tos pašus ieejas datus (uzdevumus un aģentus).

Bakalaura darba trešajā nodaļā sīki izskaidrots un aprakstīts katrs izstrādātās daudzaģentu sistēmas programmatūras modulis, pa soļiem paskaidrotas un aprakstītas visas būtiskās funkcijas.

Savukārt baklaura darba ceturtajā nodaļā, sīkāk izklāsīti DAS testēšanas rezultāti, paskaidrots ar kādiem parametriem tādi iegūti, aprakstīta autora secinājumi par katru gadījumu. Ar grafiku palīdzību arī salīdzināti dažādi DAS izpildes rezultāti.

Ievadā izvirzītā autora hipotēze darba izstrādes laikā daļēji apstiprinās – noteiktās situācijās, un pie noteiktiem parametriem, daudzaģentu sistēmu ātrdarbību var uzlabot izmantojot ģenētiskos algoritmus aģentu uzlabošanai, to kādas ir šīs situācijas, var lasīt darba ceturtajā nodaļā.

SECINĀJUMI

Šī darba mērķis – izstrādāt daudzaģentu sistēmu pielietojot ģenētiskos algoritmus, kas sistēmas darbības laikā attīsta aģentus tika sasniegts.

Darba autors uzskata, ka visi darba sākumā izvirzītie uzdevumi tika precīzi izpildīti, kas nodrošināja sekmīgu programmēšanas valodā Python strādājošas daudzaģentu sistēmas izstrādi.

Darba izstrādes laikā viens no grūtākajiem momentiem bija iepazīšanās ar literatūru par daudzaģentu sistēmām, jo kā jau minēts iepriekš, daudzaģentu sistēmas vēl joprojām ir lielākoties eksperimentālas sistēmas, kas tiek izstrādātas zinātniskā lokā un plašāk nav pieejamas.

Izstrādātajai daudzaģentu sistēmai ir saskatāmi arī turpmāki attīstības virzieni, pie kuriem darba autors labprāt vēlētos strādāt – pirmkārt, iespējams, paplašināt virtuālo resursu klasi, tādējādi nodrošinot lielāku dažādību, katram aģenta un uzdevuma objektam, kas savukārt nodrošinātu plašākas iespējas novērot to, kādas ir atšķirības starp parasto daudzaģentu sistēmu un daudzaģentu sistēmu ar ģenētiskā algoritma pielotošanu.

Otrkārt, autors saskata plašas iespējas, gan krustošanās algoritma, gan mutācijas algoritma variāciju veidošanā un uzlabošanā, kas varētu sniegt labākus rezultātus GA daudzaģentu sistēmas ātrdarbībā.

Un treškārt, ir ļoti plašas iespējas eksperimentālai visu izstrādātās daudzaģentu sistēmas konstanšu variācijai un empīriskai labāku konstanšu meklēšanai.

Izstrādājot šo darbu, darba autors ir guvis būtiski pieredzi zinātniskās literatūras pētīšanā un analizēšanā.

PATEICĪBAS

Autors izsaka pateicību savam darba vadītājam Guntim Arnicānam par nozīmīgiem ieteikumiem darba rakstīšanas procesā. Autors izsaka pateicību Viesturam Kolam par ieteikumiem darba prakstiskās daļas rakstīšanas un izpētes procesā. Autors izsaka pateicību Lina Udrenei par morālu atbalstu darba rakstīšanas laikā.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. *Multi-Agent Systems* [tiešsaiste]. Carnegie Mellon University - [atsauce 26.05.2012.]. Pieejams: http://www.cs.cmu.edu/~softagents/multi.html.
2. **Michael Wooldridge.** *CHAPTER 1: INTRODUCTION Multiagent Systems* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://www.csc.liv.ac.uk/~mjw/pubs/imas/distrib/pdf-slides/lect01.pdf.
3. **Kubera, Yoann; Mathieu, Philippe; Picault, Sébastien**. Everything can be Agent!. ***In:*** *Proceedings of the ninth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2010)*, 2010, Toronto, Canada, p. 1547-1548
4. **Russell, Stuart J.; Norvig, Peter**. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Upper Saddle River, New Jersey: Prentice Hall, 2003, ISBN 0-13-790395-2
5. **Salamon, Tomas**. *Design of Agent-Based Models*. Repin: Bruckner, 2011
6. **Utkarshraj Atmaram**. *Simple reflex agent, based on Artificial Intelligence: A Modern Approach* [tiešsaiste]. [atsauce 26.05.2012]. Pieejams: http://en.wikipedia.org/wiki/File:IntelligentAgent-SimpleReflex.png
7. **Michael Wooldridge.** *CHAPTER 2: INTELLIGENT AGENTS An Introduction to Multiagent Systems* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://www.csc.liv.ac.uk/~mjw/pubs/imas/distrib/pdf-slides/lect02.pdf.
8. **Eiben, A. E., Smith, J. E.** *Introduction to Evolutionary Computing (1st ed.)*. Springer, 2003, 300 p.
9. **Mitchell Melanie**. *An Introduction to Genetic Algorithms (Fifth printing)*. A Bradford Book The MIT Press, Cambridge, Massachusetts, London, England, 1999, [7-9] lpp ISBN 0−262−13316−4 (HB), 0−262−63185−7 (PB)
10. **William H. Hsu**. *Genetic Algorithms*. Department of Computing and Information Sciences, Kansas State University
11. *Heredity* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://en.wikipedia.org/wiki/Heredity.
12. **Darrell Whitley**. *A Genetic Algorithm Tutorial*. Computer Science Department, Colorado State University
13. *Selection (genetic algorithm)* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://en.wikipedia.org/wiki/Selection_(genetic_algorithm).

14. *Crossover (genetic algorithm)* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).
15. **P. K. Chawdhry**. *Soft computing in engineering design and manufacturing*. London: Springer, 1998, p. 164. ISBN 3-540-76214-0
16. **S. N. Sivanandam, S. N. Deepa**. *Introduction to genetic algorithms*.
17. **Marek Obitko**. *XI. Crossover and Mutation* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php.
18. *Mutation (genetic algorithm)* [tiešsaiste]. [atsauce 26.05.2012.]. Pieejams: http://en.wikipedia.org/wiki/Mutation_(genetic_algorithm).

PIELIKUMI

1. PIELIKUMS. PIRMAIS MODULIS

```
rom pc import Pc
from resource import Resource
from agent import Agent
from task import Task

from random import uniform, randint as random
import time
from datetime import datetime
from time import time
import operator

MAX_TASKS = 100000
MAX_AGENTS = 100
BEST_AGENT_CONST = 50

MIN_TASKTIME = 0.001
MAX_TASKTIME = 0.01

MIN_CPU = 0
MAX_CPU = 30
MIN_RAM = 0
MAX_RAM = 30
MIN_LAN = 0
MAX_LAN = 30
MIN_HDD = 0
MAX_HDD = 30

MAX_CYCLES = 1
MAX_GENERATIONS = 50

TASK_CONSTANT = 1000
```

```
    PC = Pc()
    tasklist = []
    # use this later to get same tasks, original will be empty
    tasklistcopy = []

    for taskID in range(MAX_TASKS):
        resource = Resource(uniform(MIN_CPU, MAX_CPU),
uniform(MIN_RAM, MAX_RAM), uniform(MIN_LAN, MAX_LAN),
uniform(MIN_HDD, MAX_HDD))
        worktime = uniform(MIN_TASKTIME, MAX_TASKTIME)
        tasklist.append(Task(taskID, resource, worktime))
        tasklistcopy.append(Task(taskID, resource, worktime))

    agentlist = []

    # use this list later to generate new agents with same values
    stupiditylist = []

    for agentID in range(MAX_AGENTS):
        stupidity = Resource(uniform(MIN_CPU, MAX_CPU),
uniform(MIN_RAM, MAX_RAM), uniform(MIN_LAN, MAX_LAN),
uniform(MIN_HDD, MAX_HDD))
        stupiditylist.append(stupidity)
        agent = Agent(agentID, stupidity, tasklist, PC)
        agentlist.append(agent)

    start = time()
    for agent in agentlist:
        agent.start()

    for agent in agentlist:
        agent.join()

    print("######### NORMAL RUN ###########")
    print(time() - start)

    # make advanced stuff here
```

```
    for big_cycle in range(MAX_CYCLES):

        cycle_start = datetime.now()
        cycleTime = 0
        for generation in range(MAX_GENERATIONS):
            genTime = 0
            #generation_task_list                                    =
tasklistcopy[generation*TASK_CONSTANT:generation*TASK_CONSTANT+TASK_
CONSTANT]
            generation_task_list = tasklistcopy[:]
            if generation == 0:
                agentlist = []
                for agentID in range(MAX_AGENTS):
                    agent  =  Agent(agentID,  stupiditylist[agentID],
generation_task_list, PC)
                    agentlist.append(agent)
            else:
                # advanced stuff

                # selection
                agent_fitness_list = {}
                for agentID in range(MAX_AGENTS):
                    result_total = 0
                    agent_fitness = 0
                    # fitness function
                    for result in agentlist[agentID].results:
                        result_total += result
                    try:
                        # fitness value
                        agent_fitness      =      result_total      /
agentlist[agentID].taskNum
                    except:
                        pass

                    if agent_fitness > 0:
                        agent_fitness_list[agentID] = agent_fitness
```

```
                agent_fitness_list                                    =
sorted(agent_fitness_list.iteritems(), key=operator.itemgetter(1))
                best_agent_list = []
                for num in range(BEST_AGENT_CONST):

best_agent_list.append(agent_fitness_list[num][0])

                # crossover
                newAgentList = []
                for agentID in best_agent_list:

newAgentList.append(agentlist[agentID].stupidityResource)

                for       firstAgentID,       secondAgentID       in
zip(best_agent_list,best_agent_list[1:])[::2]:

                    stupidity                                         =
agentlist[firstAgentID].stupidityResource
                    stupidity2                                        =
agentlist[secondAgentID].stupidityResource

                    newStupidity = Resource(0, 0, 0, 0)
                    newStupidity2 = Resource(0, 0, 0, 0)

                    if random(0, 99)<50:
                        newStupidity.cpu = stupidity.cpu
                        newStupidity2.cpu = stupidity2.cpu
                    else:
                        newStupidity.cpu = stupidity2.cpu
                        newStupidity2.cpu = stupidity.cpu
                    if random(0, 99)<50:
                        newStupidity.ram = stupidity.ram
                        newStupidity2.ram = stupidity2.ram
                    else:
                        newStupidity.ram = stupidity2.ram
                        newStupidity2.ram = stupidity.ram
                    if random(0, 99)<50:
                        newStupidity.lan = stupidity.lan
```

```
                    newStupidity2.lan = stupidity2.lan
                else:
                    newStupidity.lan = stupidity2.lan
                    newStupidity2.lan = stupidity.lan
                if random(0, 99)<50:
                    newStupidity.hdd = stupidity.hdd
                    newStupidity2.hdd = stupidity2.hdd
                else:
                    newStupidity.hdd = stupidity2.hdd
                    newStupidity2.hdd = stupidity.hdd

                newAgentList.append(newStupidity)
                newAgentList.append(newStupidity2)


            # mutation
            for newStupidity in newAgentList:
                if random(0,99) < 5:
                    randomNum = random(-3,3)
                    if (newStupidity.cpu + randomNum) >= 0 and
(newStupidity.cpu + randomNum) <= 100:
                        newStupidity.cpu += randomNum
                if random(0,99) < 5:
                    randomNum = random(-3,3)
                    if (newStupidity.ram + randomNum) >= 0 and
(newStupidity.ram + randomNum) <= 100:
                        newStupidity.ram += randomNum
                if random(0,99) < 5:
                    randomNum = random(-3,3)
                    if (newStupidity.lan + randomNum) >= 0 and
(newStupidity.lan + randomNum) <= 100:
                        newStupidity.lan += randomNum
                if random(0,99) < 5:
                    randomNum = random(-3,3)
                    if (newStupidity.hdd + randomNum) >= 0 and
(newStupidity.hdd + randomNum) <= 100:
                        newStupidity.hdd += randomNum
```

```
                # make new list of agent stupidity

                agentlist = []
                for agentID in range(MAX_AGENTS):
                    agent  =  Agent(agentID,  newAgentList[agentID],
generation_task_list, PC)
                    agentlist.append(agent)

            # now run them and see results
            start = time()
            for agent in agentlist:
                agent.start()

            for agent in agentlist:
                agent.join()

            genTime = (time() - start)
            cycleTime += genTime
            print("generation        "+str(generation)+"        time
"+str(genTime))

        print("#########     CYCLE     "+str(big_cycle)+"     RESULTS
############")
        print(str(datetime.now()      -      cycle_start)+"      time
"+str(cycleTime))
```

2. PIELIKUMS. OTRAIS MODULIS

```
from resource import Resource
from threading import Thread
import time

sleeptime = 0.005

class Agent(Thread):
    def __init__(self, agentID, stupidityResource, stack, pc):
        Thread.__init__(self)

        self.agentID = agentID
        self.pc = pc
        self.stack = stack
        self.stupidityResource = stupidityResource
        self.results = []
        self.taskNum = 0

    def run(self):
        try:
            task = self.stack.pop()
        except IndexError:
            task = None
        while task is not None:
            self.do_task(task)
            try:
                task = self.stack.pop()
            except IndexError:
                task = None

    def do_task(self, task):
        self.taskNum += 1
        self.results.append(0)
        while                self.pc.take(self.stupidityResource,
task.resource) != 0:
            self.results[self.taskNum-1] += sleeptime
```

```
            time.sleep(sleeptime)

        # fake do it
        time.sleep(task.tasktime)

        self.pc.release(task.resource)
```

3. PIELIKUMS. TREŠAIS MODULIS

```
    from resource import Resource, smallerResource
    import threading

    class Pc:
        def __init__(self):
            self.resource = Resource(100, 100, 100, 100)
            self.lock = threading.Lock()

        def take(self, checkResource, resource):
            self.lock.acquire()
            if smallerResource(checkResource, self.resource):
                # ok, so it's smart enough to do it, now let's see
if we CAN do it
                if smallerResource(resource, self.resource):
                    self.resource.cpu -= resource.cpu
                    self.resource.ram -= resource.ram
                    self.resource.lan -= resource.lan
                    self.resource.hdd -= resource.hdd
                    self.lock.release()
                    return 0
                else:
                    self.lock.release()
                    return 2
            else:
                self.lock.release()
                return 1



        def release(self, resource):
            self.lock.acquire()

            self.resource.cpu += resource.cpu
            self.resource.ram += resource.ram
            self.resource.lan += resource.lan
```

```
self.resource.hdd += resource.hdd

self.lock.release()
```

4. PIELIKUMS. CETURTAIS MODULIS

```
class Task:
    def __init__(self, taskID, resource, tasktime):
        self.taskID = taskID
        self.resource = resource
        self.tasktime = tasktime
```

5. PIELIKUMS. PIEKTAIS MODULIS

```
def smallerResource(checkResource, resource):
    validity = True
    if checkResource.cpu > resource.cpu:
        validity = False
    if checkResource.ram > resource.ram:
        validity = False
    if checkResource.lan > resource.lan:
        validity = False
    if checkResource.hdd > resource.hdd:
        validity = False

    return validity

class Resource:
    def __init__(self, cpu, ram, lan, hdd):
        self.cpu = cpu
        self.ram = ram
        self.lan = lan
        self.hdd = hdd
```

DOKUMENTĀRĀ LAPA

Bakalaura darbs „Ģenētisko algoritmu izmantošana daudzaģentu sistēmās” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: (_____) Kristaps Kols

Rekomendēju darbu aizstāvēšanai

Vadītājs: profesors, Dr. sc. comp. Guntis Arnicāns (_____) 28.05.2012

Recenzents: asoc. prof., Dr. sc. comp. Jānis Zuters

Darbs iesniegts Datorikas fakultātē _____

Dekāna pilnvarotā persona: metodiķe _____(_____)

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____, vērtējums _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____(_____)