VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Baigiamasis bakalauro darbas

Genetiniai algoritmai dirbtiniams neuronų tinklams

(Genetic Algorithms for Artificial Neural Networks)

Atliko: 4 kurso 1 grupės studentas
Laimonas Beniušis (parašas)

Darbo vadovas:
Linas Litvinas (parašas)

Recenzentas:
Rimantas Kybartas (parašas)

Vilnius – 2018

Turinys

Įvadas

Dirbtinis neuronų tinklas (angl. *Artificial Neural Network*) gali būti pritaikomas įvairiems uždaviniams spręsti, kaip simbolių atpažinimas, paveikslėlių kategorizavimas ar dirbtinio intelekto mokymas. Didžioji tokių uždavinių dalis yra struktūrų atpažinimas (angl. *pattern recognition*), ką žmonės geba atlikti natūraliai.

Populiarus „YouTube“ vaizdo įrašas pavadinimu „MarI/O - Machine Learning for Video Games“ [Set15], pritraukė didžiulį susidomėjimą genetiniais algoritmais ir dirbtiniais neuronų tinklais. Šiame vaizdo įraše dirbtinis neuronų tinklas genetinės evoliucijos būdu apmokomas greitai įveikti pirmąjį „Super Mario World“ žaidimo lygį.

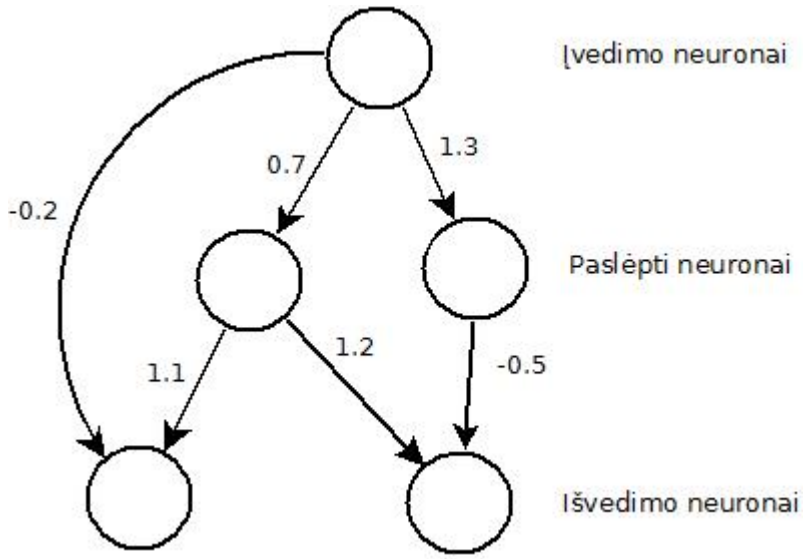
2016 metais, *Google DeepMind AlphaGo* programa įveikė pasaulio „Go“ žaidimo čempioną Lee Sedol, pasižymėdama išradingais ėjimais, kurie nustebino „Go“ ekspertus. Ši programos versija išmoko žaisti analizuodama tūkstančius mėgėjų ir profesionalų žaidimus. 2017 metais, buvo sukurta geresnė *AlphaGo* versija *AlphaGo Zero* [SSS+], kuri nesinaudoja esančiais žaidimais, tačiau išmoksta „Go“ taisykles vien tik žaisdama prieš save. Vėliau buvo sukurta bendresnė programa *AlphaZero*, kuri pasižymėjo šachmatų ir „Shogi“ žaidimuose, nugalėdama geriausius dabartinių žaidimų varikliukus (angl. *Game engine*), todėl pati idėja, kad programa pradeda neturėdama suvokimo kaip elgtis ir palaipsniui išvysto vis sudėtingesnes strategijas, yra verta dėmesio.

Šiame darbe yra aptariamas ir panaudojamas genetinis algoritmas (NEAT) dirbtiniams neuronų tinklams, bei sudėtingesnės jo variacijos. Sprendžiama dirbtinio neuronų tinklo pritaikymo kaip valdiklio (angl. *Controller*) problema. Pritaikomi įvairūs pajėgumo nustatymo būdai, tarp jų ir konkurencinga koevoliucija žaisti klasikiniam video žaidimui „Pong“

1. Dirbtiniai neuronų tinklai (DNT)

Dirbtinių neuronų tinklų yra įvairiausių formų, tačiau šiame darbe yra nagrinėjami tik tiesioginio sklidimo dirbtiniai neuronų tinklai. Tokio tipo tinklas matematiškai yra funkcija [Phi94, 8] ir tokio tinklo struktūrą gali apibūdinti orientuoto beciklio svorinio grafo analogija. Tinklą sudaro neuronai (grafo viršūnės) ir jungtys. DNT tinklo struktūra:

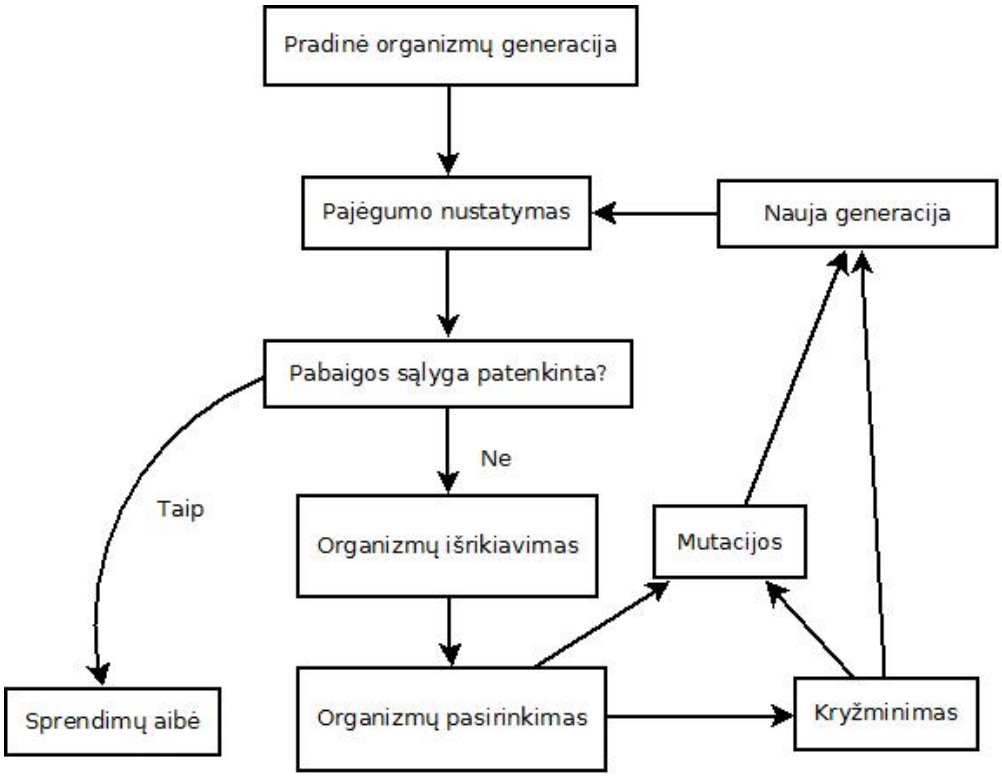
- Nekintantis kiekis įvedimo neuronų
- Nekintantis kiekis išvedimo neuronų
- Dinaminis arba nekintantis kiekis paslėptų neuronų
- Dinaminis arba nekintantis kiekis paslėptų neuronų sluoksnių
- Kiekvienas neuronas turi aktyvacijos funkciją ir (nebūtina) slenksčio reikšmę
- Dinaminis arba nekintantis kiekis jungčių, kurios apjungia neuronus
- Kiekviena jungtis turi svorio reikšmę



1 pav. Tiesioginio išvedimo dirbtinis neuronų tinklas

Rezultatas yra pasiekiamas sudauginant kiekvieno neurono įeinančius svorius ir perleidžiant juos per aktyvacijos funkciją. Kadangi DNT yra beciklio grafo struktūros per kiekvieną neuroną yra pereinama topologine tvarka. Tokiu būdu yra gaunama kompozicinė funkcija.

2. Genetiniai algoritmai



2 pav. Genetinio algoritmo veiksmų seka

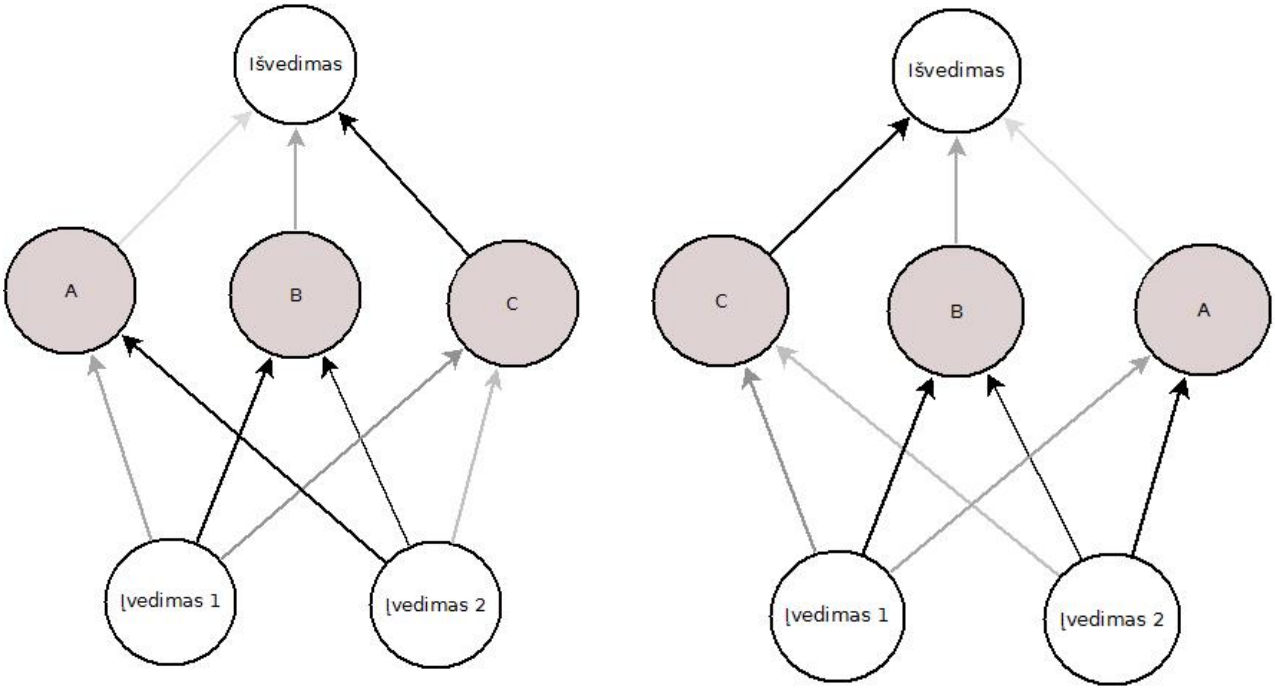
Genetiniai algoritmai yra evoliucinių algoritmų poklasis. Evoliuciniai algoritmai simuliuoja organizmų evoliuciją ir panašiai kaip gamtoje, išlieka stipriausi. Tai yra metaeuristinis optimizavimo metodas. Generuojant naują organizmų aibę yra naudojami įvairūs metodai kaip kryžminimas (angl. *crossover*) ar paprasčiausias klonavimas. Evoliucijoje dalyvaujantys organizmus galima vadinti genomais (genotipais). Genotipas yra genų sąrašas, o vizuali genotipo forma arba genų realizacija yra fenotipas [Phi94, 11].

3. Neuro-Evoliucija Augančioms Topologijoms (NEAT)

Augančių topologijų neuro-evoliucija (angl. *Neuro Evolution of Augmenting Topologies*) yra neuro-evoliucinis algoritmas, sukurtas Kenneth O. Stanley ir Risto Miikkulainen [KR02]. Šis algoritmas minimizuoja topologijas evoliucijos metu, o ne tik jos pabaigoje, nenaudojant specialios funkcijos, kuri matuoja DNT sudėtingumą. Taip pat DNT struktūros sudėtingėjimas koreliuoja su jo sprendinio korektiškumu, t. y. nėra nereikalingų neuronų ar jungčių.

3.1. Konkuruojantys sprendimai

Neuro-Evoliucijoje buvo vengiama kryžminimo (kas yra vienas iš pagrindinių GA principų) todėl, kad kryžminimas dažnai yra nenaudingas, nes DNT praranda funkcionalumą. Dėl šios priežasties kaikuriuose Neuro-Evoliucijos metoduose kryžminimo yra visiškai atsisakyta. Tai vadinama Evoliuciniu Programavimu [YL96].



[A, B, C] [C, B, A]

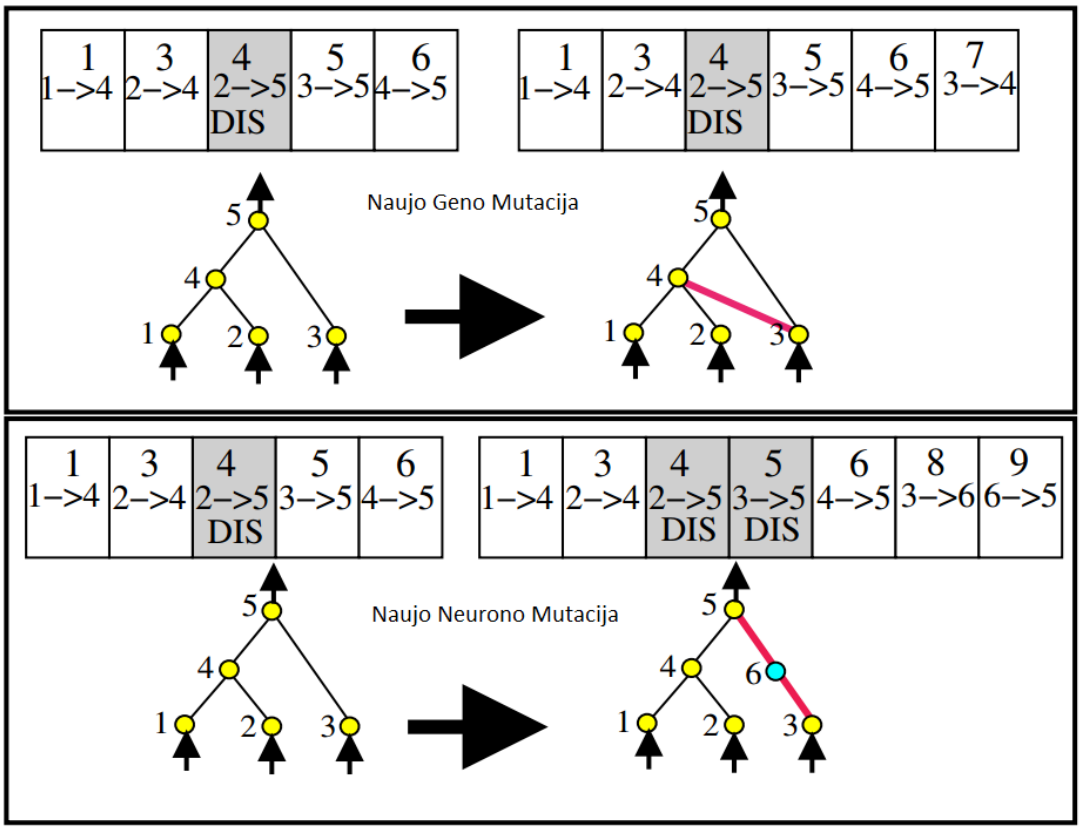
3 pav. Funkciškai vienodi tinklai. Jungčių ryškumas simbolizuoja reikšmę.

Galime nesunkiai įsitikinti, kad turime du funkcionaliai vienodus tinklus, tačiau jų paslėptų neuronų išsidėstymas daro juos skirtingais kryžminimo procese. Kryžminimo metu (B yra bendras požymis) [A, B, C]×[C, B, A] galimi vaikai yra [A, B, A] ir [C, B, C]; todėl prarandama informacija. Ši problema yra žinoma kaip konkuruojantys sprendimai (angl. *Competing Conventions*).

3.2. Genetinis Kodavimas

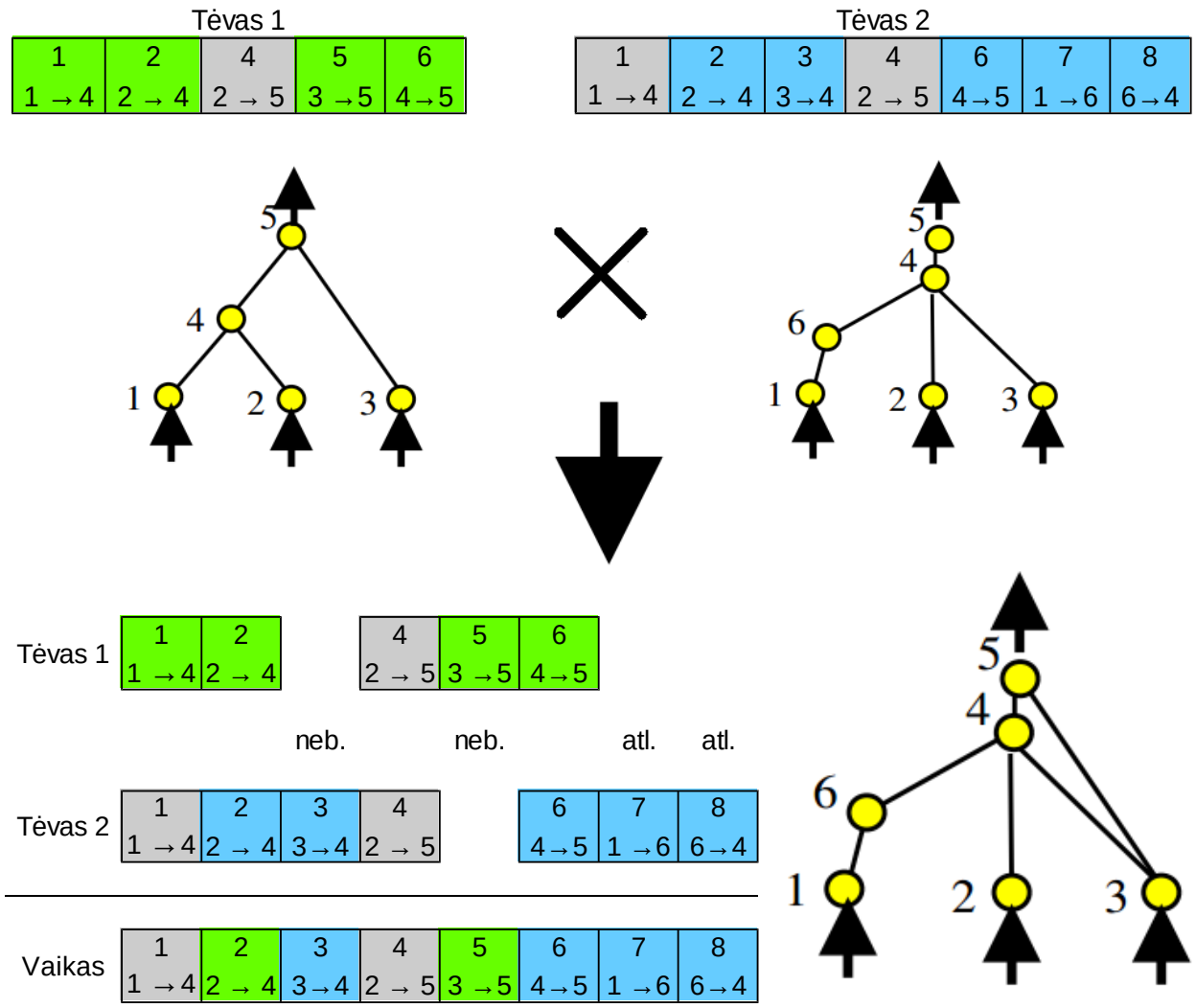
NEAT algoritme genas yra DNT lankas, kuris turi svorį, įėjimo ir išėjimo neuroną, gali būti išjungtas bei turi inovacijos žymę, kuri gali būti paprastas globalus skaitliukas [Ken04, 34]. Inovacijos žyme yra homologinis rodiklis tarp tėvų ir vaikų, t.y. paveldėti bruožai (šiuo atveju DNT struktūra). Ši žymė yra naudojama kryžminimo procese, nes tada galima atskirti sutampančius genus (svoriai gali skirtis) ir informacija yra išsaugojama. Naudojamas tiesioginis kodavimas, t. y. genai tiesiogiai atitinka fenotipą.

3.3. Mutacijos



4 pav. NEAT algoritme naudojamos mutacijos [Ken04, 36]. Pilki genai yra išjungti. Naujo neurono atveju senas genas išjungiamas, o jo vietoje atsiranda 2 nauji genai, sujungti naujo neurono. Mutacijų metu genai gali būti išjungiami. Kieviena naujo geno sukūrimo mutacija turi savo inovacijos žymę (viršuje). Vienodos genų mutacijos skirtinguose genomuose įgauna vienodą inovacijos žymę. Taip pat yra naudojamos standartinės Neuro-Evoliucijos mutacijos, kurios keičia lankų svorius.

3.4. Kryžminimas



5 pav. Skirtingų DNT kryžminimas pagal inovacijos žymes.

Nors ir tėvų fenotipai skiriasi, inovacijos žymės (geno viršuje) parodo sutampančius genus, kuriuos suporavus galima vykdyti kryžminimą [Ken04, 37]. Čia neb. (nebendri) genai yra tie, kurių inovacijos žymės neviršija mažesniojo tėvo didžiausiosios, atl. (atliekami) viršija mažesniojo tėvo didžiausiąją inovacijos žymę ir pilki genai yra išjungti

3.5. Konkuravimo išskirstymas naudojant rases

Dažniausiai, įterpiant naują atsitiktinį geną į genomą sumažėja jo pajėgumas, todėl globalioje populiacijoje modifikuotas genomas ilgai neišliks. Dėl šios priežasties yra įvedama lokali populiacija (angl. *species*), kurios genomai yra homologiškai panašūs ir konkuriuoja tarpusavyje, o ne globaliai. Tačiau kaip atskirti ar genomai yra iš tos pačios rasės? Inovacijos žymės ir vėl suteikia paprastą būdą šiai problemai spręsti. Genomų panašumo funkcija:

(1) $$d = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W$$

E - atliekamų (angl. *excess*) genų skaičius, D - nebendrų (angl. *disjoint*) genų skaičius, W - sutampančių genų svorių skirtumų vidurkis, N - didesniojo genomo genų skaičius, c_1, c_2, c_3 nustato šių daugiklių įtaką. „Kuo mažiau yra atliekamų ir nebendrų genų genomuose, tuo jie yra artimesni evoliucinės istorijos prasme“ [KR02, 112].

Kiekvienai rasei atstovauja atsitiktinis, arba pats geriausias (lyderis) genomas iš ankstesnės generacijos, kuris yra naudojamas naujos generacijos genomams sugrupuoti. Jeigu naujas genomas netinka visoms esamoms rasėms, yra priskiriamas naujai ir tampa jos atstovu. Skirstymas į rases apsaugo genomų topoligines mutacijas nuo ankstyvo eliminavimo ir suteikia šansą suoptimizuoti naujai įgytus genus. Mažiausiai pajėgūs rasės genomai yra pašalinami (pvz. mažesnio negu vidutinio pajėgumo). Generuojama nauja populiacija proporcingai kryžminant kiekvienos rasės genomus bei klonuojant geriausio pajėgumo genomus iš buvusios generacijos. Taip pat skirstymas į rases toleruoja dažnas mutacijas, kas spartina optimizavimą [Ken04, 41].

Genomo priskirimo rasei ciklas [Ken04, 39]:

1. Pasirinkti sekantį genomą *g* iš populiacijos *P*
2. Rasės ciklas:
    1. Jeigu visos rasės iš rasių sąrašo *S* buvo patikrintos, sukurti naują rasę ir įterpti į ją *g*
    2. Priešingu atveju:
        1. Pasirinkti netikrintą rasę *s* ir jeigu *g* turi pakankamai panšumų su *s* rasės atstovu, *g* priskirti *s* rasei
    3. Jeigu *g* nebuvo priskirtas rasei, tęsti rasės ciklą
3. Jeigu visi genomai buvo priskirti, baigti

3.5.1. Chaotiškos aplinkos išnaudojimas rasių išlyginimui

Originalus NEAT algoritmas pasitelkia pasverto pajėgumo metodą, norint išvengti vienos rasės dominavimo. Šis metodas tiesiog padalina genomo pajėgumą iš jo rasės dydžio, todėl didelės rasės būna labiau nubaudžiamos. Dažnai šis algoritmas yra pritaikomas problemoms, kurios yra chaotiškos, t. y. Mažas pokytis lemia kardinalų genomo elgesio pokytį. Tokiu atveju, daugumos naujų genomų pajėgumas krenta drastiškai. Ši savybė yra išnaudojama SethBling [Set15] NEAT algoritmo versijoje, kada programa mokosi greitai įveikti pirmąjį „Super Mario World“ žaidimo lygį. Šis žaidimas *side-scrolling platform* žanro, todėl maži pokyčiai pakeičia vieną veiksmą, o tas veiksmas turi didžiulę įtaką rezultatui. Dauguma naujų genomų prastai pasirodo, nes aplinka yra chaotiška, tačiau deterministinė. Tokia aplinka leidžia naudoti alternatyvų rasių išlyginimo metodą:

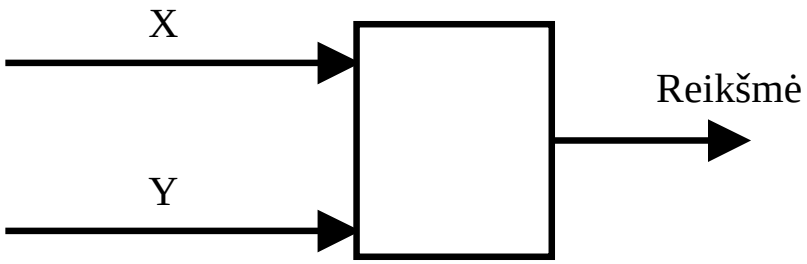
1. Surinki visus genomus į bendrą sąrašą
2. Išrūšiuoti juos pajėgumo didėjimo tvarka
3. Iš eilės priskirti genomams užimamą vietą (rangą), kuri yra genomo eilės numeris sąraše nuo 1
4. Apskaičiuoti kiekvienos rasės vidutinį rangą S_{avg}
5. Apskaičiuoti rasių vidutinio rango sumą S_{total}
6. Apskaičiuojamas rasei skiriamas naujų genomų skaičius $\lfloor \frac{S_{avg}}{S_{total}} \times \| P \| \rfloor - 1$

Tokia sistema veikia todėl, kad jeigu viena rasė dominuoja visą populiaciją, tai ji prigamins daugiau genomų. Kadangi nauji genomai dažniausiai yra nevykę, jie turės prastą rangą, kas lems rasės populiacijos pajėgumo vidurkio kritimą. T.y. kuo daugiau prastų genomų rasė prigamins, tuo mažiau resursų jai bus skiriama sekančios generacijos metu ir yra išvengiama vienos rasės dominavimo.

4. Kompoziciniai ornamentus konstruojantys tinklai (CPPN)

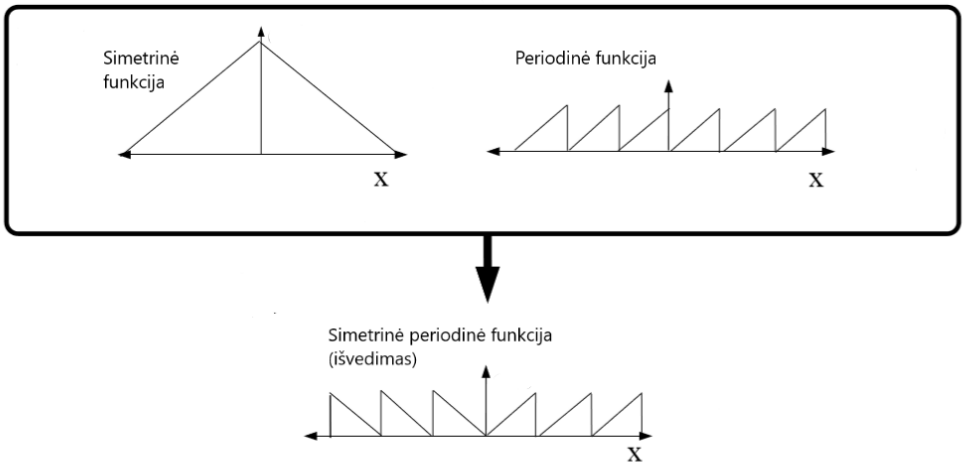
Tinkamos informacijos kodavimo abstrakcijos ieškojmas atvedė prie kompozicinių ornamentus konstruojančių tinklų (angl. *Compositional Patern Producing Networks*) išradimo. CPPN yra tiesioginio išvedimo dirbtinių neuronų tinklų klasė, kuri veikia kaip funkcija suteikianti erdvės taškui reikšmę. Tokiu būdu galima išsaugoti neribotos rezoliucijos paveikslėlį:

- Paduodamos koordinatės yra normalizuojamos [-1, 1], ar kitokiame intervale (centras yra 0)
- Galima įtraukti atstumo nuo centro reikšmę (lengviau atvaizduoti skritulius)
- Galima turėti kelias išvedimo reikšmes (ryškumui, spalvai, tamsumui)



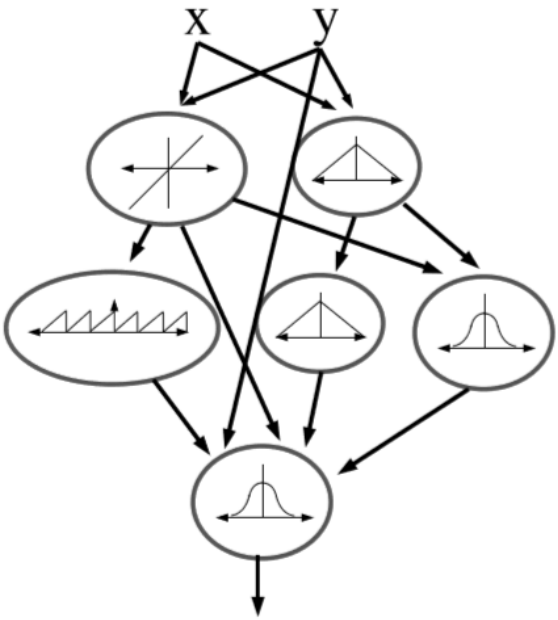
6. pav. Paprastas erdvės taško kodavimas.

Viena iš svarbiausių CPPN savybių yra aktyvacijos funkcijos. Skirtingos aktyvacijos funkcijos pagamina skirtingus dėsningumus. Kadangi tai yra tiesioginio išvedimo tinklas, skirtingos aktyvacijos funkcijų kompozicijos leidžia gauti dar sudėtingesnius ornamentus ar dėsningumus.



7. pav. Skirtingų funkcijų kompozicija.

Pagrindinė CPPN idėja yra aktyvacijos funkcijų eiliškumas. Tų pačių funkcijų skirtingas išsidėstymas kardinaliai keičia rezultatą [Ken07,11]. Natūraliai galima tokią kompoziciją įgyvendinti dirbtiniu neuronu tinklu. Taigi, CPPN yra daugiamatės erdvės informacijos abstrakcijos įrankis.



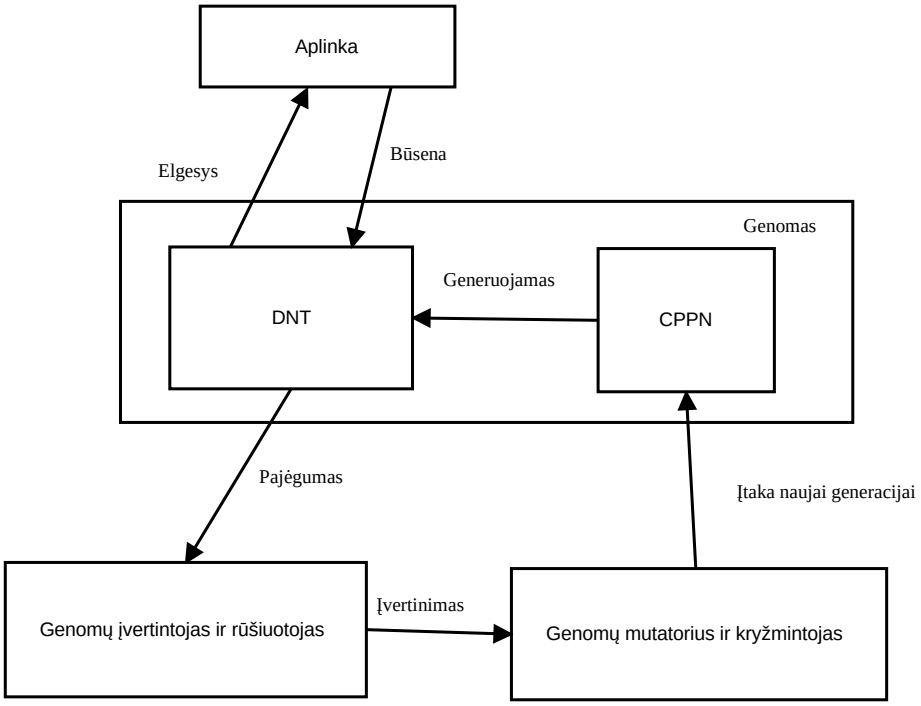
8. pav. CPPN eskizas šaltinis:[Ken07,12]

5. Hiperkubinė Neuro-Evoliucija augančioms topologijoms (HyperNEAT)

Ši NEAT algoritmo versija yra sukurta Kenneth O. Stanley, David D'Ambrosio ir Jason Gauci [KDJ09,NFAQ]. HyperNEAT (angl. *Hypercube-based NeuroEvolution of Augmenting Topologies*) algoritmo siekis yra išspręsti dvi DNT problemas:

- Efektyvi informacijos abstrakcija
- Erdvės suvokimas

Dauguma DNT metudų neatsižvelgia į užduoties erdvę. Pvz.: Įvedimo aibė būna masyvas, suformuotas iš matomos lentos. Pats DNT mato tik pačias reikšmes, tačiau neturi suvokimo kaip arti viena kitos jos yra ar kai panašios reikšmės yra šalia. Dažniausiai tokie dėsningumai yra atrandami iš naujo tinklui sudėtingėjant ir besimokant, jeigu iš viso yra atrandami.

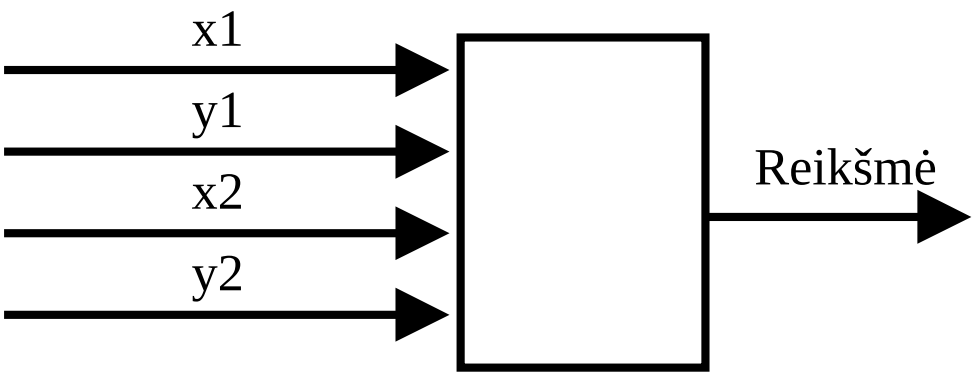


9. pav. HyperNEAT algoritmo ciklo modelis

HyperNEAT algoritmas pasitelkia jau minėtu informacijos abstrakcijos įrankiu erdvėje: CPPN. Pagrindinė esmė yra evoliucionuoti jungius CPPN naudojant NEAT algoritmą. NEAT gamina tokius CPPN, kurie išsaugo erdvines struktūras hipererdvėje. [CN, HyperNEAT 13].

5.1. Erdvės dėsningumų transformavimas į DNT lankus

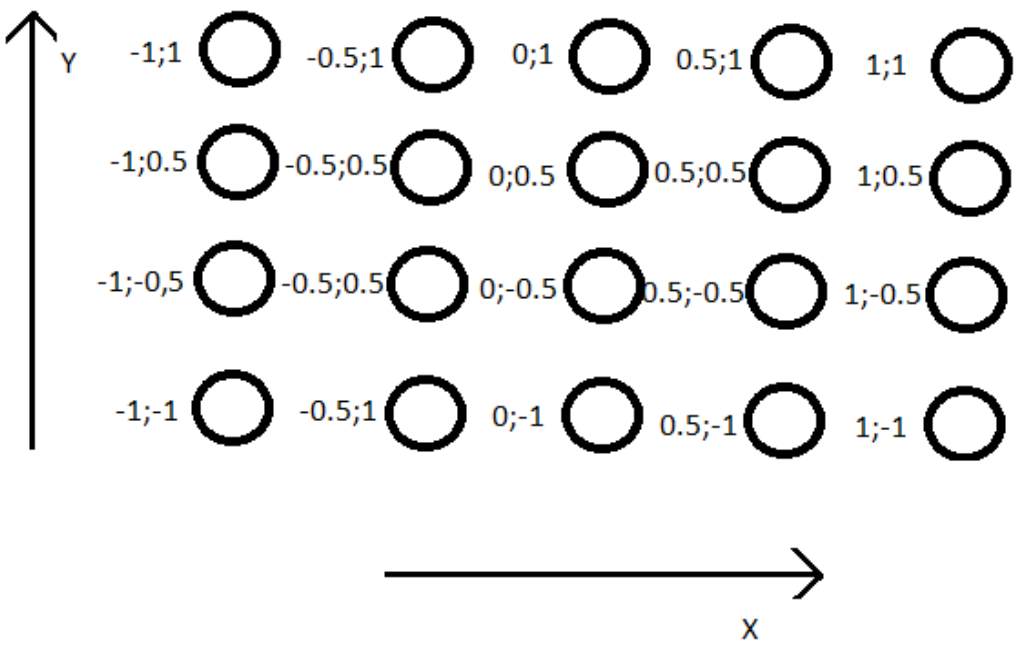
Kiekvienas taškas CPPN išsaugotoje struktūroje, kuri yra apribojama hiperkubo, yra interpretuojamas kaip jungtis žemėsnės dimensijos grafe. Žinoma, norint gauti CPPN kuris yra panašus į 8. pav. būtina naujo tipo mutacija. Taigi, mutacijų sąrašą papildo aktyvacijos funkcijos tipo mutacija. Taip pat reikia CPPN pritaikyti naujo DNT generavimui. Pavyzdžiui, paprastas sluoksniuotas dviejų dimensijų DNT gali būti generuojamas 4 įvedimų CPPN pagalba.



10. pav. CPPN pritaikytas 2 dimensijų DNT generavimui

Tokiu atveju, DNT įgyja stačiakampio formą (11. pav.):

- Sluoksnių kiekis tampa stačiakampio ilgiu bei generuojamos figūros tikslumu (kuo daugiau sluoksnių, tuo didesnis tikslumas)
- Įvedimo/išvedimo kiekis tampa stačiakampio pločiu.



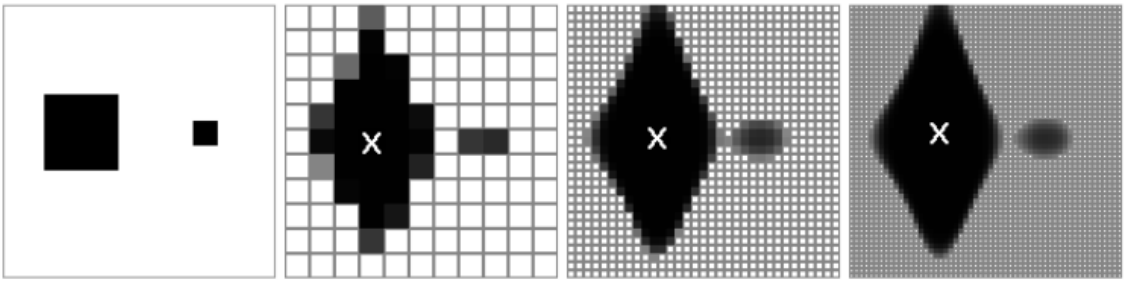
11. pav. 2 dimensijų stačiakampio metafora pritaikyta DNT su atitinkamomis koordinatėmis.

Taigi, dabar generuojamo DNT struktūra yra statinė, tik kinta jungčių reikšmės. Kiekvienas sluoksnis yra pilnai jungus su šalia esančiais. Kaip 9 pav. parodyta, kiekvienas neuronas turi lokalias koordinates, kurios ir yra naudojamos jungties svoriui nustatyti. Norint sumažinti nereikšmingų jungčių kiekį, siekiant pagreitinti sugeneruoto DNT įvertinimą, galima neįterpti jungčių, kurios turi mažą (moduliu) reikšmę. Siekiant išlaikyti tradicinį sluoksniuotą DNT modelį, jungtys yra įterpiamos tik tarp loginių sluoksnių, tačiau galima jungti bet kurio sluoksnio neuroną su bet kurio kito sluoksnio nesudarant ciklų.

5.2. HyperNEAT masiškumas

Galima pastebėti, kad šis algoritmas yra nesunkiai išplečiamas į didesnes dimensijas – reikia CPPN įvedimą adaptuoti iki neuronų koordinačių atitikimo. Jeigu įvedimo aibė būtų lenta, tada generuojamas DNT taptų stačiakampio gretasienio formos, jeigu įvedimo aibė yra trimatė erdvė, tada taptų 4 dimensijų hiperstačiakampiu arba hiperkubu (nuo pastarojo ir kilo pavadinimas) ir t.t..

Kadangi HyperNEAT algoritmo esmė yra vidinis CPPN, kuris ir išsaugo informaciją apie DNT jungtis, tą patį CPPN galima panaudoti su skirtingų kiekių įvedimo/išvedimo DNT (11. pav.). Taip pat galima tinklą apmokyti su didelio masto sluoksniais ir po to pritaikyti mažesnio masto DNT generavimui.



12. pav. To paties CPPN pritaikymas skirtingų dydžių DNT. Šaltinis:[KJ07]

6. Koevoliucija

Koevoliucija turi daug reikšmių biologijoje, tačiau klasikinis apibrėžimas yra skirtingų rasių priešiškas elgesys, verčiantis nuolat keistis ir adaptuotis.

Koevoliucijos modelis yra naudojamas kaip metaeuristinis optimizacijos metodas su algoritmais, kurių pagrindas yra populiacinė atranka [Sea15]. Optimizacijos prasme, koevoliucija apima situacijas, kuriose individo pajėgumas yra įvertinimas pagal kitų individų populiacijoje pajėgumą. Pagrindiniai koevoliucijos tipai galėtų būti skirstomi taip:

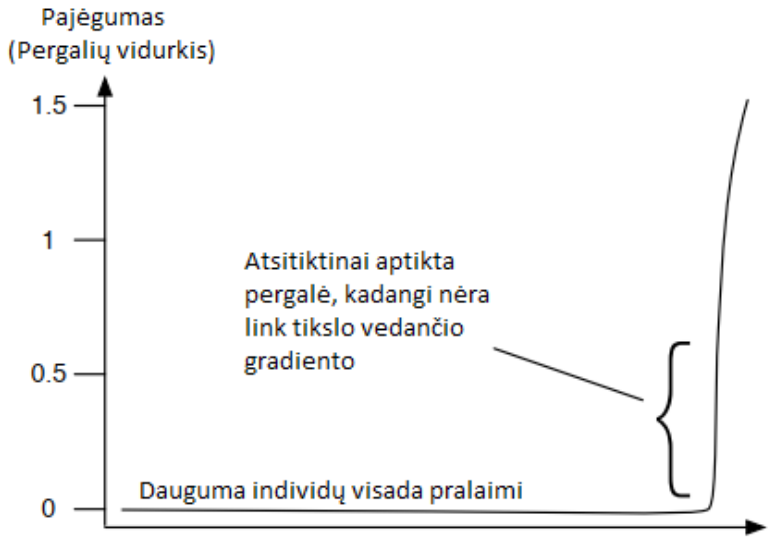
- Konkurencinga vienos populiacijos koevoliucija – Individai yra iš tos pačios populiacijos. Jų pajėgumas yra įvertinamas testais, kuriuose dalyvauja tos pačios populiacijos individų poros.
- Konkurencinga dviejų populiacijų koevoliucija – Populiacija susideda iš dviejų atskirų lokalių populiacijų (sub-populiacijų). Iš pirmos sub-populiacijos individo pajėgumas yra nustatomas pagal tai kiek individų iš antros sub-populiacijos jis sugebėjo nugalėti. Analogiškai nustatomas antros sub-populiacijos individų pajėgumas testuojant su pirmosios. Dažniausiai pirmoji sub-populiacija turi individus, kurie bando išspręsti duotą užduotį, o individai iš antros sub-populiacijos bando jiems sutrukdyti.
- Bendradarbiaujanti N-populiacijų koevoliucija – Sprendžiama užduotis yra padalinama į N mažesnių užduočių, kurioms spręsti yra paskiriama konkreti sub-populiacija. Iš esmės, bandoma evoliucionuoti komandą. Pajėgumas yra matuojamas surenkant individus iš kitų sub-populiacijų suformuojant N dydžio sprendinį ir įvertinant jo pajėgumą spręsti duotą užduotį. Dažniausiai naudojamas sumažinti paieškos erdvei sudėtinguose užduotyse suskaldant jas į paprastesnes užduotis.

Šiame darbe bus nagrinėjamos pirmosios dvi koevoliucijos rūšis.

6.1. Konkurencinga vienos populiacijos koevoliucija

Konkurencinga vienos populiacijos koevoliucija dažniausiai naudojama optimizuojant sprendimus, kurie pritaikyti tam tikro žaidimo įveikimui. Pvz.: Kumar Chellapila ir Devid Fogel [CF01] naudojo šią strategiją gerų šaškių žaidėjų paieškai. Vienas iš esminių šios strategijos privalumų yra geresnis (nuoseklus ir nestatus) mokymosi gradientas paieškos erdvėje.

Tarkime, problemos sritis yra robotų dvikova. Vienas iš sprendimų yra turėti vieną individą, kuris yra tokios dvikovos ekspertas. Tada individo pajėgumas bus nustatomas pagal tai, kiek dvikovų jis laimėjo prieš šį ekspertą. Tada kyla kita problema, kad didžioji dauguma atsitiktinių individų yra prasti: jie pralaimi visas dvikovas su ekspertu. Evoliucijos pradžioje, visa populiacija bus būtent iš tokių prastų individų ir programa negalės atskirti kuris iš individų yra geresnis už kitus. Tokiu atveju, sprendimo kokybės grafikas bus plokščias iki tol, kol atsitiktinai vienas iš individų sugebės nors kartą įveikti šį ekspertą dvikovoje (13. pav.), kas retai nutinka. Nuo to momento, evoliucija tampa lengvesnė.



13. pav. Prasta kokybės kreivė dėl gradiento trūkumo

Kitoks sprendimas būtų suporuoti dvikovas tarp pačių individų. Pradžioje, individas yra prastas, tačiau ir jo oponentas yra prastas, todėl kažkas galiausiai turi laimėti, todėl nors vienas ar keli individai turės nenulinį pajėgumą. Tada optimizavimo algoritmas turėtų atskirti kurie individai yra truputį geresni. Vėliau, kai individai pagerėja, pagerėja ir jų oponentai, todėl dvikova savaime tampa sudėtingesnė. Tokiu būdu yra sukuriama sistema kuri automatiškai reguliuoja ir adaptuoja mokymosi gradientą.

Individų pajėgumas įgyja kitą prasmę. Dabar sistema nebeturi galimybės apskaičiuoti absoliutaus individo pajėgumo. Optimizacijos procesas yra remiamas reliatyvaus kontekstinio pajėgumo įverčiu, Pvz.: Jeigu individas, iš 1-os generacijos įkeltas 101-ą, tai jo reliatyvus pajėgumas būtų prastas, nes per 100-ą generacijų individai pagerėjo. Šis pokytis sukelia kelias problemas. Ankščiau algoritmo progresą stebėti buvo paprastą, nes pajėgumas buvo matuojamas absoliučiai. Dėl to yra įmanoma situacija kada individai tobulėja, tačiau reliatyvus pajėgumas išlieka toks pat, nes jų oponentai irgi tobulėja. Taip pat toks pajėgumo įvertinimas gali sutrikdyti individų atranką bei kryžminimą, jeigu tai buvo atliekama pagal pajėgumo įverčius. Dėl šių priežasčių reikalingi 2 pajėgumo įvertinimo metodai: vidinis ir išorinis. Vidinis yra jau aptartas, reliatyvaus pajėgumo įvertis, o išorinis gali būti:

- Testas su rankiniu būdu sukurtais individais
- Testas su prieš tai esančios generacijos individais, darant prielaidą, kad vėlesnės generacijos individai yra geresni. Rezultatai gali būti keisti, jeigu sistema nėra stabili.
- Testas su išorine sistema tikrame pasaulyje

Abstraktus konkurencingos vienos populiacijos koevoliucijos algoritmas:

1. Sukurti pradinę populiaciją
2. Kartoti kol baigsis laikas arba rastas tenkinantis sprendimas
    1. Įvertinti vidinį pajėgumą
    2. Įvertinti išorinį pajėgumą
    3. Pagal išorinio pajėgumo įvertį rasti geriausią individą
    4. Sukurti naują populiaciją

6.2. Reliatyvaus vidinio pajėgumo įvertinimas

Yra daugybė skirtingų būdų įvertinti, kurie individai elgiasi geriau, poruojant juos su kitais individais. Pagrindinė problema yra testų kiekis. Daugybė žaidimų, turnyrų, dvikovų yra stochastiniai. Pvz.: jeigu norime surasti geresnį pokerio žaidėją, reikėtų sužaisti daugybę partijų. Net jeigu bandoma spręsti problema nėra stochastinė, kelią blokuoja ciklai tarp individų. T.y. jeigu A individas nugali B, o B nugali C, tačiau C nugali A, kuris iš šių individų yra geriausias? Dar didesnė problema yra jeigu B individas įveikia daugiau individų negu A, tačiau A įveikia B, kuris individas dabar geresnis? Dažniausiai yra ieškomas individas, kuris turi daugiausiai pergalių, arba kuris turi daugiausiai pergalių prieš gerus žaidėjus, arba turi didžiausią taškų vidurkį. Tokiais atvejais reikalingas ne vienas testas, norint rasti iš tiesų geriausius individus. Tai lemia kompiuterio resursų balansavimo dilemą: ar daugiau laiko tikrinti kuris individas geriausias, ar skirti resursus tolimesnei paieškai.

Porinis reliatyvaus pajėgumo įvertinimo algoritmas

1. sumaišyti(*P*)
2. *kol i =0; i < ||P||; i = i + 2*
    1. testuoti(*P[i] , P[i + 1]*)
    2. įvertinti pajėgumą pagal testo rezultatus

Čia $||P||$ – *P* populiacijos dydis, *P[i]* – populiacijos *P* individas *į* indekse.

Šio algoritmo privalumas yra toks, kad testų kiekis yra tik $\frac{||P||}{2}$, tačiau kadangi kiekvienas individas yra pratestuojamas tik su vienu kitu individu, rezultatai gali gautis triukšmingi. Alternatyva yra testuoti visus individus su visais.

Pilnas reliatyvaus pajėgumo įvertinimo algoritmas:

1. *kol i = 0; i < ||P||; i = i + 1*
    1. *kol j = i +1; j <||P||; j = j + 1*
        1. testuoti(*P[i], P[j])*
    2. Įvertinti pajėgumą pagal testų rezultatus (*P[i]*)

Šio algoritmo privalumas yra toks, kad kiekvienas individas yra testuojamas $||P||-1$ kartų, tačiau tai lemia, kad iš viso testų kiekis padidėja iki $\frac{||P||\times(||P||-1)}{2}$.

Reikėtų rasti kompromisą tarp pakankamai gerai įvertinto individo pajėgumo ir testų kiekio.

N-kartinis reliatyvaus pajėgumo įvertinimo algoritmas:

1. *n* = pasirinktas minimalus testų kiekis
2. *kol i = 0; i < ||P||; i = i + 1*
    1. *R = n dydžio atsitiktinų indeksų masyvas, kuriame nėra skaičiaus i*
    2. *kol j = 0; j <||R||; j = j + 1*
        1. *m = R[j]*
        2. testuoti(*P[i], P[m]*)
    3. Įvertinti pajėgumą pagal testų rezultatus (*P[i]*)

Štai toks algoritmas leidžia reguliuoti testų kiekį pagal *n* parametrą ir kiekvienas individas dalyvaus bent *n* kiekyje testų, tačiau dalis iš jų visgi dalyvaus daugiau negu *n* testuose. Toks algoritmas leidžia bendrą testų kiekį sumažinti iki $n \times \| P \|$, tačiau tai visgi gali būti per daug. Jeigu pajėgumas yra skaičiuojamas pagal pergalių kiekį, vienas iš algoritmų yra turnyrinė atranka. Tada pajėgumas yra tai, kaip aukštai individas pakyla turnyre.

Atkrentamų varžybų turnyro reliatyvaus pajėgumo įvertinimo algoritmas:

1. *R = sumaišyti(||P||)*
2. *kol i = 1; i < log_2(||P||); i = i +1*
    1. *Q = R*
    2. *R = {}*
    3. *kol j = 0; j < ||Q||; j = j + 2*
        1. *testuoti(Q[j], Q[j+1])*
        2. įterpti į aibę *R* nugalėtoją
3. Įvertinti pajėgumą visiems individams pagal testų rezultatus

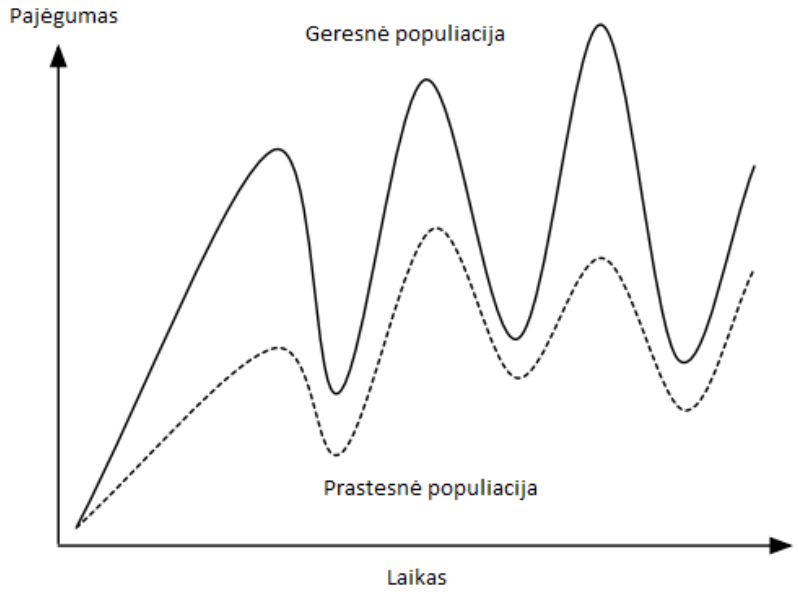
Štai toks algoritmas atlieka $\| P \| - 1$ testų, ir vidutiniškai kiekvienas individas dalyvauja dviejuose testuose. Taip pat šis algoritmas pasižymi tuo, kad geresni individai patenka į daugiau testų, tam tikra prasme, jau leidžia atskirti geresnius individus. Vienas didelis trūkumas tokio algoritmo, yra tai, kad jeigu sprendžiama užduotis yra stochastinė, geras individas lengvai gali pasimesti tarp prastų atrankoje anksti iškritęs. Taip pat toks pajėgumo įvertinimas riboja testų išlygiagretininą.

Visi šie reliatyvaus pajėgumo algoritmai pasižymi bendra savybe – individų atranka be absoliutaus pajėgumo įverčio. Visgi, mokymosi gradiento mažinimo savybė nėra visoms problemoms tinkantis sprendimas, nes gana lengva visai populiacijai užstrigti lokaliam ekstremume.

6.3. Konkurencinga dviejų populiacijų koevoliucija

Konkurencinga dviejų populiacijų koevoliucija tuo pačiu ieško gerų sprendinių bei atranda tokius testų atvejus, kurie nėra lengvai įveikiami. Pagrindinė populiacija gamina tokius individus, kurie bus naudojami uždavinio sprendimui, o kita, šalutinė populiacija, gamina pagrindinės populiacijos oponentus. Tokiu būdu, pagrindinės populiacijos individo pajėgumas bus nustatomas pagal, kiek oponentų iš šalutinės populiacijos jis įveikė. Analogiškai nustatomas šalutinės populiacijos individo pajėgumas. Jeigu sistema yra stabili, yra gaunami vis sudėtingesni sprendimai, sprendžiantys vis sudėtingėjančias užduotis. Taip pat visi anksčiau minėti reliatyvaus pajėgumo įvertinimo algoritmai (išskyrus turnyras) gali būti nesunkiai adaptuojami dviejų populiacijų koevoliucijai.

Dviejų populiacijų konkurencinga koevoliucija yra dažnai apibūdinama kaip abstrakti biologinių ginklavimosi varžybų versija: pirmoji populiacija išranda naują strategiją, priverčiama kitą populiaciją adaptuotis prie jos ir išrasti kažką, kas nugali pirmosios išradimą. Toks konkuravumas iš dalies primena šeimininko-parazito tipo santykius. Idealiu atveju, tokios varžybos lemią natūralią, lėtai ir pastoviai kylančią sunkumo kreivę, kuri leidžia išvengti problemos, pavaizduotos (13. pav.). Tačiau, tokie idealūs variantai pasitaiko retai. Dažnai gali būti taip, kad viena iš populiacijų gauna žymiai lengviau optimizuojamą užduotį, ir kita populiacija nespėja prisitaikyti. Galiausiai, dominuojanti populiacija įveikia visus atsiliekančios populiacijos individus, todėl atrankos algoritmas pradeda šlubuoti, kas lemia absoliutaus pajėgumo kritimą (14. pav.). Deja ši problema nėra lengvai išsprendžiama.



14. pav. Pajėgumo kreivės periodinis smukimas

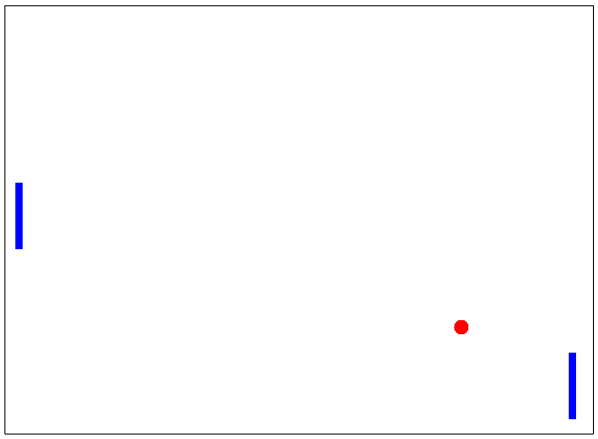
7. Neuro-evoliucija valdikliui

Užduoties tikslas – neuro-evoliucijos būdu suformuoti valdiklį (angl. *Controller*), kuris gebėtų išspręsti duotą užduotį. Informacijos apie ją valdikliui duodama mažai bei taisykles valdiklis turi suprasti/atrasti pats, iš jo pajėgumo įverčio. Gana platus NEAT ir HyperNEAT algoritmų pritaikymas evoliucionuojant roboto valdiklius. Šis mechanizmas yra atsakingas už roboto veiksmus, kurie skiriasi nuo jį supančios aplinkos būsenos, t.y. kiekvieną laiko vienetą robotas daro išmoktą veiksmą, atsižvelgiant į aplinką, tokiu būdu įgyvendindamas savo funkciją. Kadangi valdikliui evoliucionuoti yra žymiai pigiau ir greičiau naudoti robotų simuliacijas kompiuterizuotoje aplinkoje, kas yra labai panašu į video žaidimų veikėjo valdymą: Aplinka jau paruošta, belieka atrasti teisingus veiksmus, kurie lemtų norimą rezultatą.

NEAT algoritmas neišnaudoja geometrinės erdvės, todėl jeigu mokymui yra naudojami tik informacija ekrane, HyperNEAT algoritmas yra pranašesnis, tačiau analizuojant esamą informaciją ir paliekant tik tai kas gali būti aktualu (Pvz.: veikėjo normalizuotos koordinatės) NEAT algoritmas randa geresnius sprendimus. Tai buvo patikrinta su žaidimu „Tetris“ [GGS17].

7.1. Video žaidimas „Pong“

Klasikinis, vienas iš pirmųjų video žaidimų yra „Pong“, išleistas 1975 metais *Atari* video žaidimo platformai, kurio autorius yra Allan Alcorn. Tai yra vienas iš paprasčiausių video žaidimų. Tai yra dviejų dimensijų žaidimas, primenantis stalo tenisą. „Pong“ žaidimas yra skirtas dviem žaidėjams. Kiekvienas iš jų valdo po lentelę (raketę) priešingose arenos pusėse. Kaip ir stalo tenise, žaidimo pagrindinis objektas yra kamuoliukas, kuris lanksto nuo vienos pusės į kitą, kol vienas iš žaidėjų į jį nepataiko. Lentelės valdomos tik aukštyn arba žemyn.



15. pav. Paprastas „Pong“ žaidimas.

8. Valdiklis video žaidimui „Pong“

Kurti video žaidimų dirbtinį intelektą dažnai nėra paprasta užduotis. Tam reikia žinoti suprasti žaidimo ypatumus, bei mokėti juos išnaudoti. Dažniausiai, nors ir sukurtas intelektas yra stiprus, žmogaus išmonė jam neprilygsta. Visgi, *Google DeepMind* komanda sugebėjo įrodyti, kad ir seniausi žmonijos žaidimai kaip šachmatai ar „Go“ dar nėra iki galo suprasti, įveikdama stipriausius pasaulio žaidėjus bei žaidimų varikliukus, naudodama iki šiol nematytas strategijas.

Video žaidimas „Pong“ yra paprastas, todėl dirbtinis intelektas rankiniu būdų jam gali būti suprogramuotas nesunkiai, tačiau viena iš jo savybių yra tai, kad jis skirtas dviem žaidėjams, todėl yra tinkamas uždavinys koevoliucijos metodui.

8.1. DNT struktūra

DNT sudaro $2+B\times 2$ įvedimo neuronai ir 3 išvedimo neuronai, kur B yra kamuoliukų kiekis žaidime. Pirmi 2 neuronai atitinka žaidėjo, ir priešininko normalizuotas Y koordinates, likusieji atitinka kamuoliukų X ir Y normalizuotas koordinates, visi įvedimai yra apriboti [0..1] intervale.

Išvedimų yra 3, kurių jų reikšmės atitinka pasirinktą veiksmą (žingsnis viršun, žemyn, arba likti vietoje). Didžiausia neurono reikšmė ir bus tą laiko vienetą pasirinktas veiksmas.

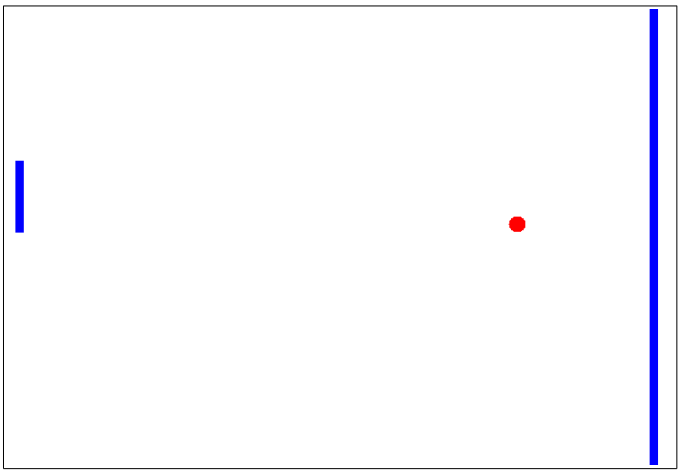
8.2. Skirtingos versijos

Originalioje „Pong“ žaidimo versijoje, kamuoliuko greitis priklauso nuo lentelės judėjimo greičio atsimušimo metu, bei atsimušimo taško. Skrydžio trajektorija taip pat priklauso nuo atsimušimo taško, t. y. kuo toliau nuo lentelės yra atmušamas kamuoliukas, tuo didesniu kampu jis skrieja. Taškai yra skiriami tik kada žaidėjas praleidžia kamuoliuką.

Pati primityviausiai „Pong“ versija yra kada kamuoliuko greitis nekinta ir trajektorija visada atsimuša 90° laipsnių kampu. Pasunkinta versija būtų kada kamuoliuko greitis didėja kas kiekvienu jo atmušimu (iki nustatyto limito). Galiausiai apjungiant kamuoliuko greičio augimą kartu su kampo keitimu nuo atmušimo taško gaunama gana sunki žaidimo versija. Taip pat, galima žaisti su daugiau negu vienu kamuoliuku. Kitoks taškų skaičiavimo būdas yra skaičiuoti ne oponento praleistus kamuoliukus, o savo atmuštus. Toks taškų skaičiavimo būdas suteikia tikslesnį pajėgumo matavimą, tačiau gali turėti neprognozuotų pasekmių, kurios bus paminėtos vėliau.

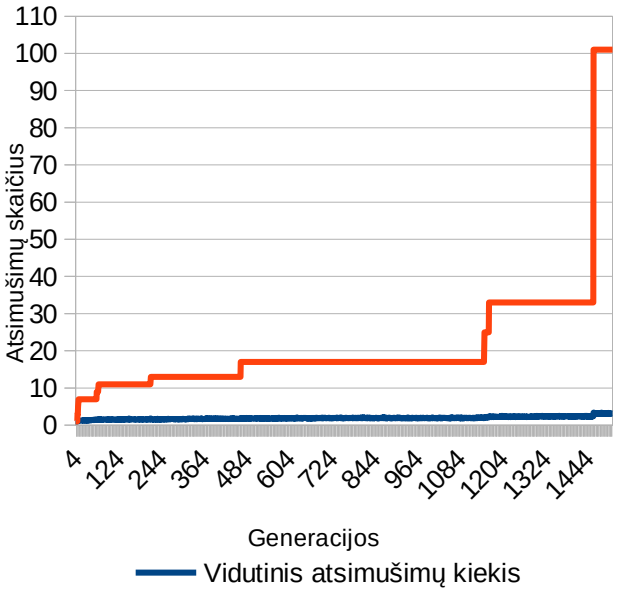
8.3. Bandymai ir rezultatai

Prieš taikant koevoliuciją, buvo prasmingą įsitikinti, kad veikia standartiniai metodai. Buvo naudojamas NEAT algoritmas su chaotiškos aplinkos rasių paskirstymo funkcija. Vienas iš būdų įvertinti pajėgumą, yra suporuoti genomą su žaidimo ekspertu, su kuriuo galėtų testuoti progresą ir mokytis. Galima naudoti lentelę, kuri seką kamuoliuko Y koordinatę, tačiau tai veikia tik jeigu yra vienas kamuoliukas ir atmušimo kampas visada bus toks pats. Šiuo atveju yra labai paprastas sprendimas: vietoj žaidimo eksperto, žaisti prieš sieną (16. pav.). Taip pat, žaidimas yra apribojamas iki 50 taškų vienam žaidėjui, todėl žaidimas baigiasi kada kamuoliukas yra atmušamas 101-ą kartą arba vienas iš žaidėjų jo neatmuša.

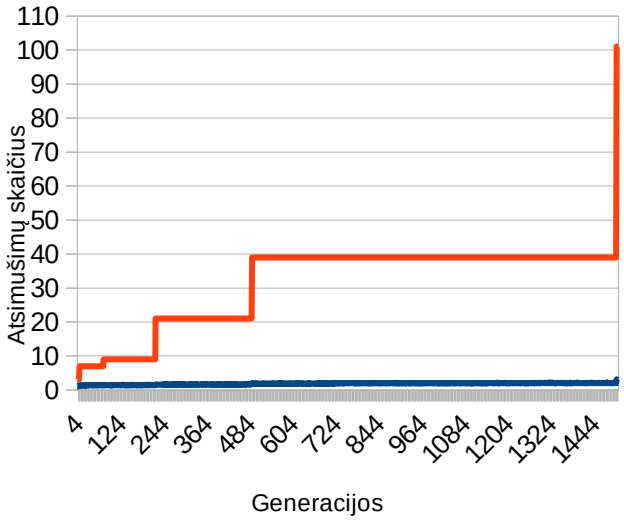


16. pav. „Pong“ žaidimas prieš sieną. Dešinysis žaidėjas negali pralaimėti, atmušimo kampas būna įvairus ir gali atremti bet kokį kamuoliukų skaičių.

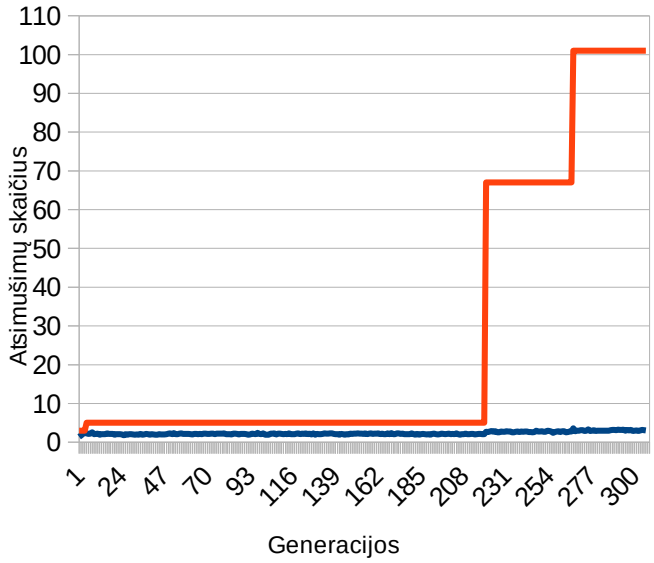
Kadangi algoritmas yra metaeuristinis, todėl rezultatai nėra garantuojami ir jo eiga skiriasi nuo mažų pokyčių parametruose. Toliau bus pateikiami įvairių žaidimo versijų bei algoritmų rezultatų grafikai, kurie buvo gana sėkmingi.



17. pav. NEAT algoritmo evoliucijos grafikas, žaidžiant prieš sieną. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis pastovus.



18. pav. NEAT algoritmo evoliucijos grafikas, žaidžiant prieš sieną. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis didėja.



19. pav. NEAT algoritmo evoliucijos grafikas, žaidžiant prieš sieną. Kamuoliuo trajektorija nekinta, greitis didėja.

Kaip matome, grafikas yra laiptuotas ir generacijų skaičius gana didelis, tačiau progresas, atsitiktinai vyksta. Atmušimo kiekio vidurkis rodo, kad iš tiesų evoliucija yra chaotiška. Taip pat, žaidimo versija turi didelę įtaką algoritmo progresui.

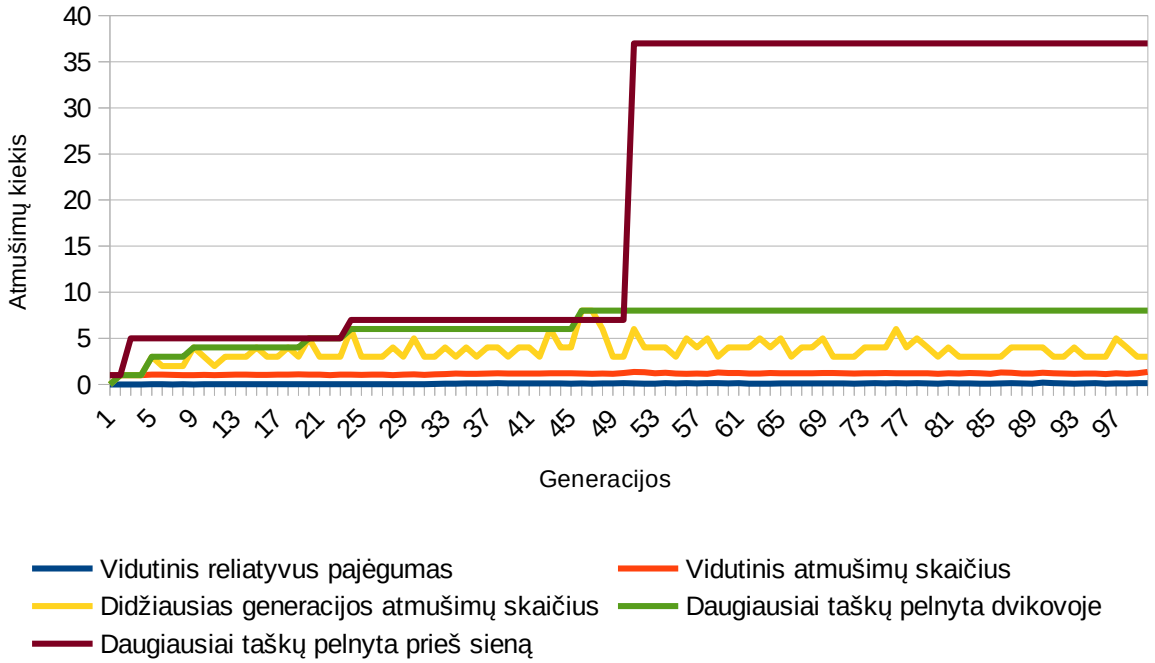
Problema pasirodė įveikiama standartiniams algoritmams, treniruojantis prieš žaidimo ekspertą, šiuo atveju – sieną. Tačiau dažnai nėra taip lengva sukurti oponentą, kuris butų geras mokytojas, todėl pritaikyta koevoliucija. Buvo nagrinėjama tik sunkiausia žaidimo versija, kurioje kamuoliuko kampas ir greitis kinta. Pirmiausiai buvo pritaikyta 1-os populiacijos koevoliucija. Buvo pasirinktas pilnas reliatyvaus pajėgumo įvertinimo algoritmas genomams poruoti, todėl kad testo simuliacija yra įvykdoma greitai. Genomai dalyvauja toje pačioje NEAT algoritmo populiacijoje. Pirmiausiai genomų pajėgumas buvo skaičiuojamas pergalių skaičiumi, tačiau toks būdas buvo nesėkmingas, nes toks įvertinimas nulėmė ne evoliuciją, o periodišką strategijų kaitą. T.y. jeigu kamuoliukas yra atmušamas į viršų, tada genomai turi išmokti jį priimti iš viršaus ir atmušti ten, iš kur dar nėra išmokta jo atmušti. Dar viena priežastis dėl ko taip galėjo būti yra tai, kad pagrindinis rezultato valdytojas yra kamuoliukas. Priešininkas neprivalo pajudėti, jeigu kamuoliukas skrenda tiesiai į jį, kas lemia pergalių skyrimą už neveiksnumą. Dėl šių priežasčių evoliucija negalėjo vykti nesinaudojant taškų sistema.

Buvo įvesta nauja pajėgumo matavimo sistema, kuri naudoja išnaudoja taškus (atmušimus). Genomo pajėgumas yra įvertinamas taip:

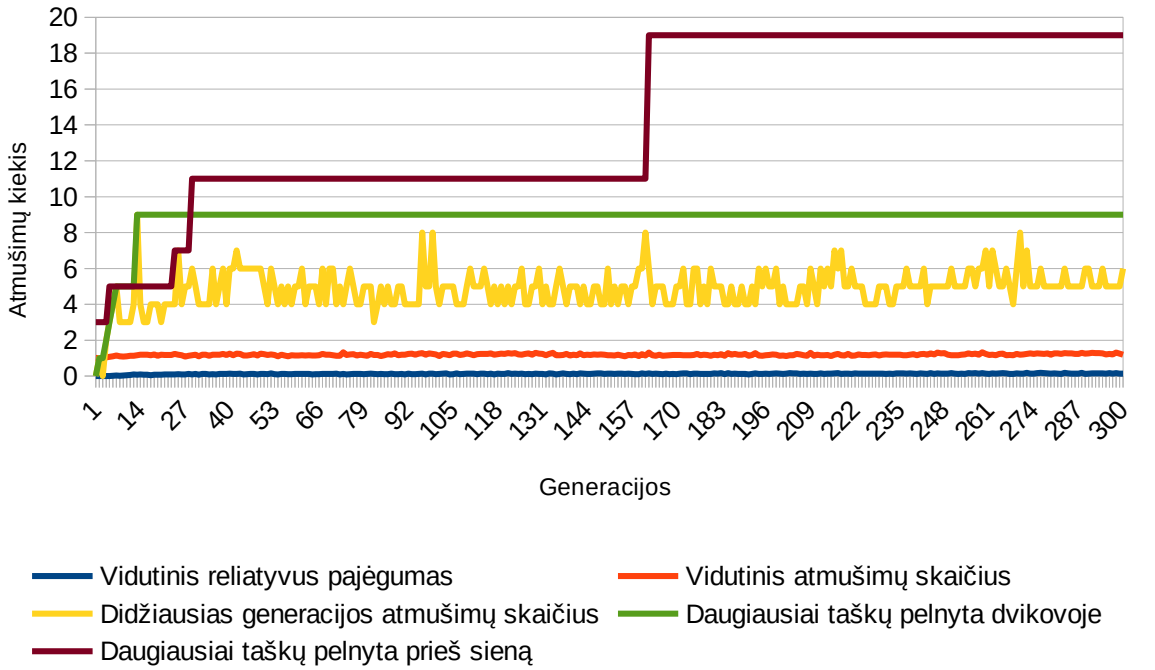
1. Susumuojami visų dvikovų taškai ir išsaugojami (Pvz.: Jeigu A laimėjo prieš B rezultatu 2:1, tai prie A taškų sumos prisideda 2, o prie B prisideda 1)
2. Genomo pajėgumas gaunamas susumuojant įveiktų oponentų taškus

Tokiu būdu yra stengiamasi atskirti pajėgius genomus pagal tai, kaip gerai jie uždirba taškus. Tokia įvertinimo sistema, netiesiogiai skatino ne konkuravimą, o bendradarbiavimą. Todėl pasitaikė situacija, kada genomų dvikovoje susidarė ciklas, kada jie vienas kitam perduoda lengvai atmušamą kamuoliuką ir taip lengvai renka taškus. Verta paminėti, kad šioje žaidimo versijoje kamuoliukas negreitėjo.

Toliau yra parodyti sėkmingų ir nesėkmingų vienos populiacijos koevoliucijų atvejų grafikai. Sėkmingu atveju yra laikoma evoliucija, kada atmušimų skaičius viršija 30.



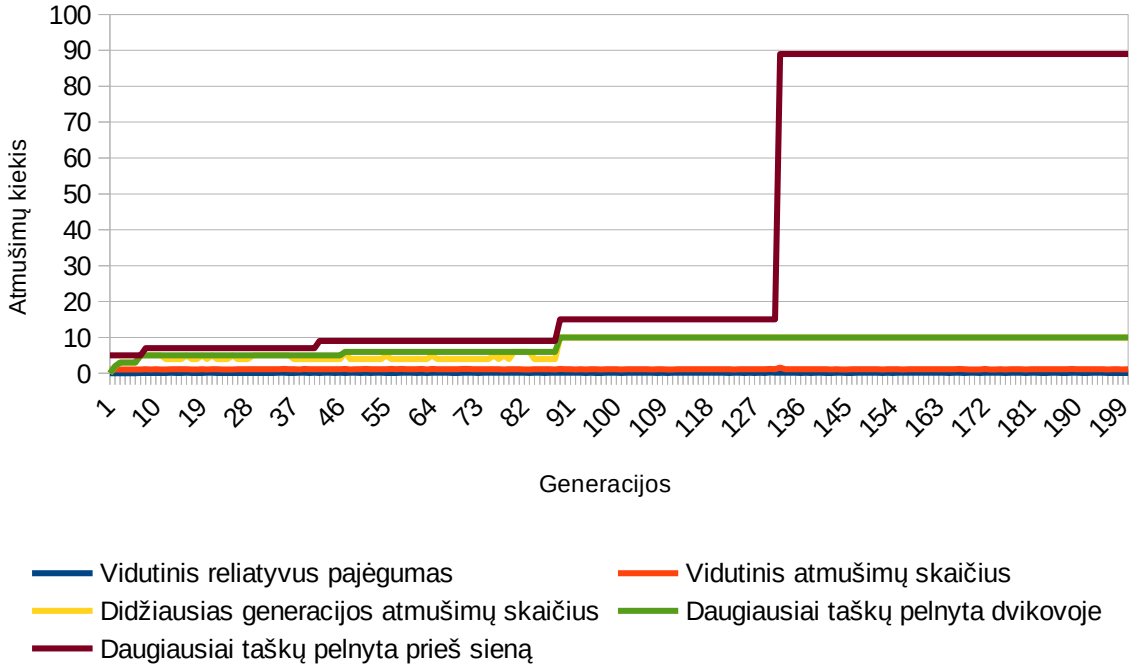
20. pav. Sėkmingos NEAT algoritmo evoliucijos grafikas, pritaikant koevoliuciją. Naudojama viena populiacija. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis didėja.



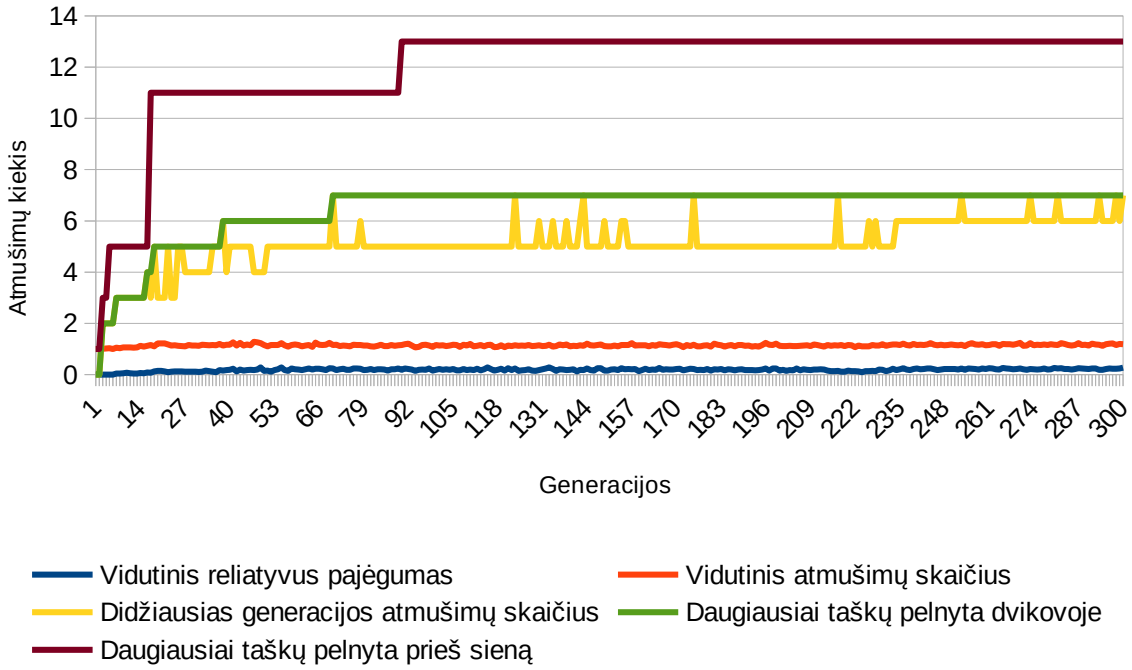
21. pav. Nesėkmingos NEAT algoritmo evoliucijos grafikas, pritaikant koevoliuciją. Naudojama viena populiacija. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis didėja.

Tai pat buvo įgyvendinta dviejų populiacijų koevoliucija. Tokį sprendimą paskatino tas faktas, kad žaidimo sąlygos nėra visiškai lygios, nes tas žaidėjas, į kurio pusę skrieja pirmasis kamuoliukas, evoliucijos pradžioje turi daugiau šansų pralaimėti. Taigi, buvo padalinama populiacija į dvi, atskirai NEAT algoritmo valdomas sub-populiacijas, kurių atstovai buvo poruojami pilnu reliatyvius pajėgumo įvertinimo algoritmu. Tokiu būdu buvo sumažinamas testų kiekis beveik 4 kartus.

$\lim_{p\to\infty} \frac{(p\times(p-1))}{((\frac{p}{2})\times(\frac{p}{2}))} = \lim_{p\to\infty} 4\frac{(p-1)}{p} = 4$, kur p – pilnos populiacijos dydis.



22. pav. Sėkmingos NEAT algoritmo evoliucijos grafikas, pritaikant koevoliuciją. Naudojamos dvi populiacijos. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis didėja.

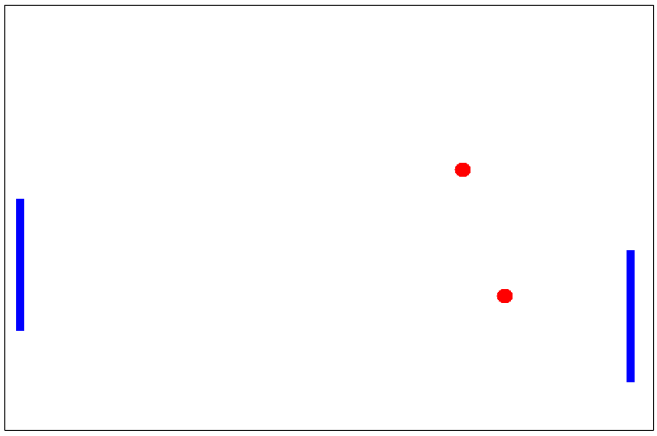


23. pav. Nesėkmingos NEAT algoritmo evoliucijos grafikas, pritaikant koevoliuciją. Naudojamos dvi populiacijos. Kamuoliuko trajektorija kinta nuo atmušimo taško, greitis didėja.

Kaip matome, abu koevoliucijos rezultatai gali būti ir sėkmingi ir nesėkmingi, tačiau 2-jų populiacijų koevoliucija visgi buvo sėkmingesnė ir progresas vyko pastoviai, įvertinus tai, kad didžiausias generacijos atmušimų skaičius yra nestabilus vienos populiacijos koevoliucijoje, tačiau pastoviai kyla dviejų populiacijų koevoliucijoje, kas užtikrina populiacijos tobulėjimą.

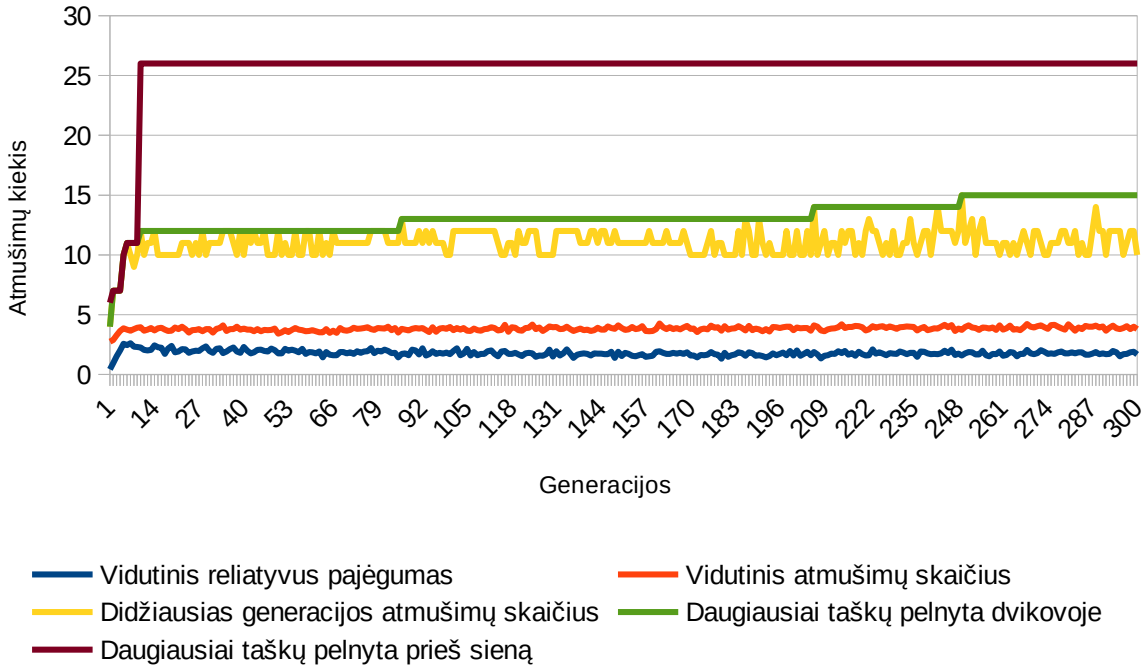
8.4. Dviejų kamuoliukų variacija

Viena iš anksčiau paminėtų žaidimo pasunkinimų yra padidintas kamuoliukų skaičius. Kadangi negalima tikėtis begalinio žaidimo, algoritmas stengiasi išdidinti taškų skaičių. Iš anksčiau atliktų bandymų, geriausiai pasirodė dviejų populiacijų koevoliucijos variantas, todėl jis ir buvo naudojamas bandant rasti geriausią dviejų kamuoliukų mušinėjimo strategiją. Suteikiant daugiau šansų, buvo dvigubai padidinamos raketės. Visi parametrai išliko tie patys, tačiau dabar DNT konfiguracija pakito: iš 4 tapo 6 įvedimo neuronai, nes padidėjo kamuoliukų skaičius.

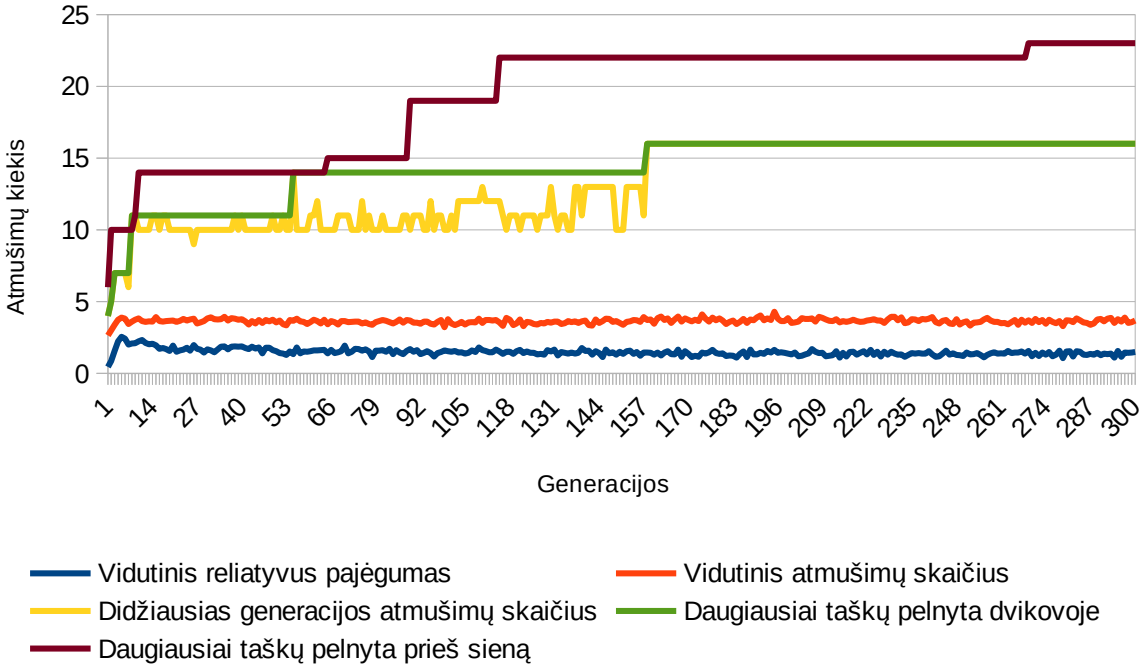


24. pav. „Pong“ žaidimas su 2 kamuoliukais

Algoritmui nepavyko išrasti novatoriškos strategijos, kuri leistų žaisti ilgai, ko ir buvo galima tikėtis. Tačiau pavyko rasti porą, kuri galėtų surinkti 15 bendrų taškų, bei atsitiktinai labai anksti evoliucijoje pavyko atrasti žaidėją, kuris surinktų 12 (26 bendrus) taškų žaisdamas prieš sieną. Žaidimas buvo su greitėjančiu ir trajektorijos kampą keičiančiu kamuoliuku.



25. pav. Geriausias atsitiktinės veiksmų sekos radimas, dviejų kamuoliukų „Pong“ žaidime.



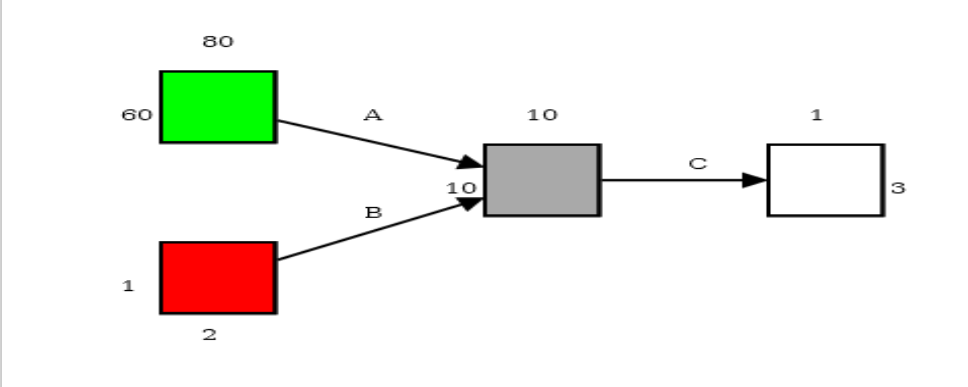
26. pav. Prastesnis, tačiau tolygesnis koevoliucijos grafikas, dviejų kamuoliukų „Pong“ žaidime

Dviejų kamuoliukų žaidime, daugiausiai taškų surinkdavo tie genomai, kurie sugebėdavo atrasti veiksmų eigą, kuri nukreipdavo kamuoliukus patogiai atmušamu kampu. Tokia strategija veikia tik tada, jeigu kamuoliukai visada pradeda judėti iš tos pačios pozicijos. Tokiu atveju, yra įsiminama veiksmų seka, tačiau nėra išmokstamos taisyklės.

HyperNEAT verta naudoti kada nėra akivaizdžių geometrinių priklausomybių ir naudinga pažinti geometrinę erdvę, tačiau tai reikalauja žymiai daugiau skaičiavimų.

8.5. HyperNEAT konfiguracija

Buvo pritaikytas HyperNEAT algoritmas, sunkiausiai žaidimo versijai (2 greitėjantys ir trajektorijos kampą keičiantys kamuoliukai). Vidinis CPPN buvo iš 4 įvedimo ir 3 išvedimo neuronų. Kiekvienas išvedimo neuronas, kuriuos pavadinkime A, B, ir C (27. pav.) buvo taikytas atskiram sluoksniui sujungti. Visi sluoksniai yra 2-jų dimensijų, nes tokia yra žaidimo erdvė. Sluoksnių matmenys ir kita informacija yra aprašyti žemiau.

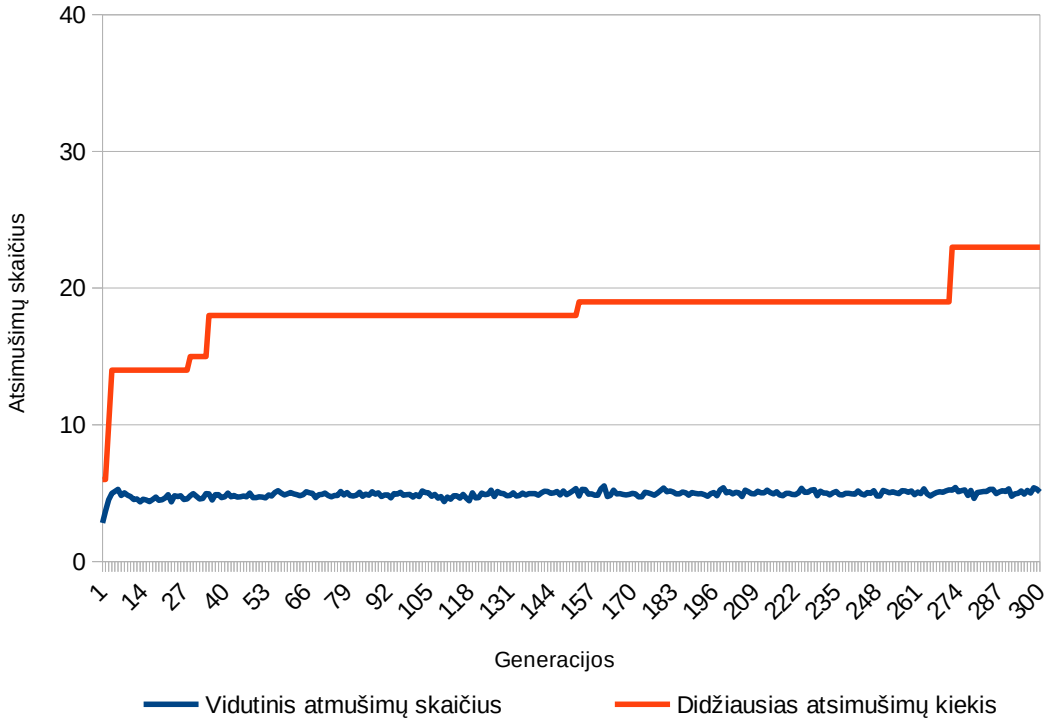


27. pav. HyperNEAT konfiguracijos eskizas.

Žalias sluoksnis naudojamas kamuoliukų pozicijoms nustatyti. Atitinkamo neurono reikšmė, pagal poziciją yra paverčiama 1. Žaidimo matmenys yra 800x600, koordinatės yra dalinamos iš 10 ir apvalinamos iki artimiausios reikšmės. Raudonas sluoksnis yra iš 2 neuronų, kurių pozicijos yra abscisių ašies galuose, naudojamas lentelių pozicijoms nustatyti. Neurono reikšmė priklauso nuo lentelės Y pozicijos žaidime. Pilkas sluoksnis yra apjungiamasis ir tarpinis sluoksnis. Baltas sluoksnis yra išvedimo sluoksnis, pagal kurį yra nusprendžiamas kitas žingsnis. Kiekviena koordinatė yra normalizuojama [0,1] intervale. Veiksmas, kaip ir NEAT algoritmo versijoje, yra nusprendžiamas pagal didžiausią išvedimo neurono reikšmę.

Tokiu būdu gaunamas tinklas, kuris gali turėti 5220 skirtingas jungtis $((80\times60)\times(10\times10))+((2\times1)\times(10\times10))+((10\times10)\times(1\times3))$. Tačiau ne visos jungtys yra naudojamos. Jeigu jungties absoliuti vertė neviršija 0.2, ji nėra įtraukiama. Taip bandoma sumažinti mažą įtaką darančių jungčių kiekį.

8.5.1. HyperNEAT rezultatai



28. pav. HyperNEAT algoritmo sunkiausioje žaidimo versijoje (2 greitėjantys ir kampą keičiantys kamuoliukai) grafikas.

HyperNEAT algoritmas atsižvelgia į erdvės geometriją, kas gali būti išnaudojama geresnių sprendimų darimui. „Pong“ žaidime erdvė yra paprasta (lentelės ir kamuoliukai) todėl lengva pritaikyti analitinį sprendimą jos interpretavimui, kas ir buvo padaryta NEAT algoritme. Kaip matome (28. pav.) , HyperNEAT algoritmas pasiekė panašius rezultatus kaip ir NEAT. Taip pat HyperNEAT rado ne veiksmų seką, kuri pasiektų didžiausią taškų skaičių, o išmoko geometrinius dėsningumus, kas leidžia geriau adaptuotis prie panašių (bet nevienodų) situacijų, ko ir buvo siekiama.

Vienas nedidelis privalumas yra tai, kad HyperNEAT kamuoliukams skirtas (žalias 27. pav.) sluoksnis yra pritaikytas neribotam kamuoliukų skaičiui, ko negalima pasakyti apie NEAT algoritmą.

9. Išvados

Aptarti neuro-evoliuciniai algoritmai yra galingi įrankiai įvairaus mąsto ir spektro problemoms spręsti. Konkretus jos algoritmas NEAT, kurio pagrindinės idėjos yra inovacijos žymės ir suskaidymas rasėmis, gali būti sėkmingai pritaikytas valdikliui evoliucionuoti. Pasitelkus CPPN yra įmanomas HyperNEAT algoritmas, kuris bando išspręsti informacijos abstrakcijos ir erdvės suvokimo problemą vystant DNT. HyperNEAT yra prasmės naudoti ten, kur yra išnaudojami geometriniai dėsniai.

Koevoliucija yra vertas dėmesio metodas, kadangi sistema turi išmokti aplinkos taisykles be jokių pradinių dirbtinių žinių, tačiau jos taikymo sfera yra ribota dėl dvikovos tipo pajėgumo testavimo būdo. Koevoliucijoje gaunasi natūraliai kylanti sudėtingumo kreivė, nes individai tobulėja kartu su savo priešininkais, tačiau tada atsakomybė krenta sistemai teisingai įvertinti koevoliucijoje dalyvaujančius individus. Taip pat toks metodas leidžia nekurti užduoties sferos eksperto (kas gali būti sudėtinga), tačiau norint stebėti visos populiacijos progresą dažnai neišvengiamas išorinis patikrinimas. Vienas koevoliucijos trūkumas yra didžiulis testų kiekis, lyginant su kitais pajėgumo įvertinimo metodais.

Kompiuterinis žaidimas „Pong“ buvo tinkamas projektas pritaikyti įvairius neuro-evoliucijos bei koevoliucijos metodus. Šis žaidimas yra lengvai patikrinamas ir sunkinamas, kas leido ištirti algoritmų galimybes bei trūkumus. Buvo bandoma įvertinti individus be informacijos apie žaidimą (šiuo atveju taškų kiekis) naudojant tik pergalių kiekį to pasiekti nepavyko.

Iš principo evoliuciniai algoritmai yra stiprus įrankis spręsti problemas, kurių analitiniai modeliai yra per sudėtingi tradiciniams metodams ir dažniausiai pilnas perrinkimas nėra realistiškai įmanomas. Taip pat yra aktualu, kad evoliuciniai algoritmai yra lengvai išlygiagretinami, kadangi gaminamų procesorių kompiuteriuose kiekis didėja, tačiau ne tranzistorių greitis (dažnis) [Kar18].

Summary

Artificial Intelligence has a massive application field. Combined with genetic evolutionary algorithms it can result in powerful problem solving tool. In this paper problem field is learning to efficiently play video games. Recently, *Google DeepMind* team proved the astonishing results of self-play in Go and Chess games which brought more attention to the co-evolution idea.

The main video game that is analyzed in this paper is Pong. The version of game Pong is varied by changing speed, trajectory based on the touching point or adding more balls in order to increase difficulty. The algorithms applied are NEAT, NEAT with co-evolution and HyperNEAT. Since these algorithms are meta-heuristic and utilize random mutations, the success of such methods is not definite. Nonetheless, achieved results indicate that the problem is indeed solvable with or without co-evolution, although fitness-less optimization was not achieved due the nature of the problem.

Literatūros sąrašas

[CF01] Chellapilla K., Fogel D. B. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation* (Volume: 5, Issue: 4) 2001

[GGS17] Gillespie E. Lauren, Gonzalez R. Gabriela, Schrum Jacob. Comparing Direct and Indirect Ecodings Using Both Raw and Hand-Designed Features in Tetris. Southwestern University, Texas, 2017

[Kar18] Karl Rupp. 42 Years of Microprocessor Trend Data (2018). https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[KR02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99, 127, 2002.

[Ken04] Kenneth O. Stanley. Efficient Evoliution of Neural Networks through Complexification. PhD thesis, The University of Texas, 2004

[Ken07] Kenneth. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines, 8(2):131–162, 2007.

[KJ07] Kenneth. O. Stanley, J. Gauci. Generating Large-Scale Neural Networks Through Discovering Geometric Regularities, GECCO ‘07 Proceedings of the 9th annual conference on Genetic and evolutionary computation 997-1004, 2007

[KDJ09] Kenneth. O. Stanley, D. D’Ambrosio, J. Gauci. A Hypercube-Based encoding for evolving Large-Scale neural networks. Artificial Life, 15(2):185–212, 2009

[MM96] Moriarty, D. E., and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32

[MPK+96] Meuleau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. P. (1999). Learning finite-state controllers for partially observable environments. *In Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence.*

[NFAQ] Kenneth O. Stanley, The NeuroEvolution of Augmenting Topologies (NEAT) Users Page, FAQ, https://www.cs.ucf.edu/~kstanley/neat.html

[Phi94] Phillipp Koehn (1994) Combining Genetic Algorithms and Neural Networks: The Encoding Problem. Master Thesis. University of Tennessee, Knoxville

[Sea15] Sean Luke. Essentials of Metaheuristics, Second Edition, George Mason University, 2015

[Set15] SethBling. MarI/O - Machine Learning for Video Games. (2015). https://youtu.be/qv6UVOQ0F44

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis. Mastering the Game of Go without Human Knowledge. DeepMind, London, 2017

[YL96] Yao, X. and Liu, Y. (1996). Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90.

Priedai

Java programos kodas yra GitHub tinklapyje, GPL v2

Pagrindiniai neuro-evoliucijos algoritmai: https://github.com/laim0nas100/NeurEvol

„Pong“ žaidimo pritaikymas su algoritmu: https://github.com/laim0nas100/PongNEAT

Pagalbinių klasių rinkinys naudotas rezultatų išrašymui, interfeisui bei programos paralelizavimui: https://github.com/laim0nas100/Commons