VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Kursinis projektas

Neuro-evoliucija pritaikyta valdikliui

(Neuro-evolution for evolving a controller)

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

Darbo vadovas:
Linas Litvinas (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.

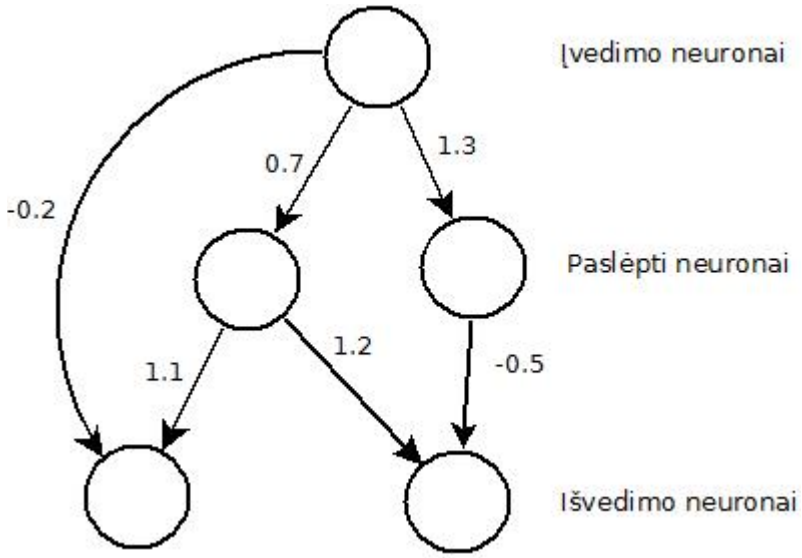
Populiarus „YouTube“ vaizdo įrašas pavadinimu „MarI/O - Machine Learning for Video Games“ [Set15], pritraukė didžiulį susidomėjimą genetiniais algoritmais ir dirbtiniais neuronų tinklais. Šiame vaizdo įraše dirbtinis neuronų tinklas genetinės evoliucijos būdu apmokomas greitai įveikti pirmąjį „Super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

Šiame darbe yra plačiau aptariamas ir panaudojamas kursinio darbo metu pristatytas genetinis algoritmas dirbtiniams neuronų tinklams, bei sudėtingesnės jo variacijos. Sprendžiama dirbtinio neuronų tinklo pritaikymo kaip valdiklio (angl. *Controller*) problema.

1 Dirbtiniai neuronų tinklai (DNT)

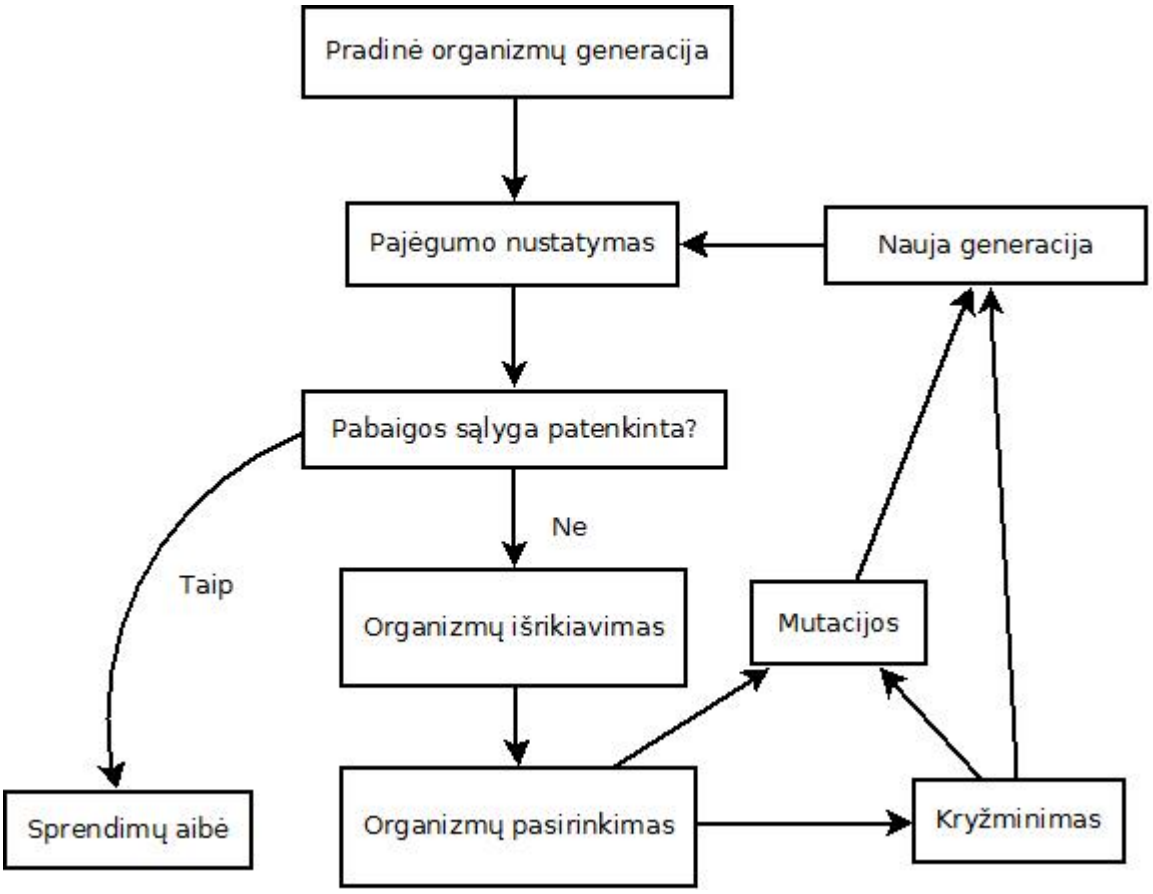
Tinklą sudaro neuronai (grafo viršūnės) ir jungtys. DNT tinklo struktūra:

- 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

2 Genetiniai algoritmai (GA)



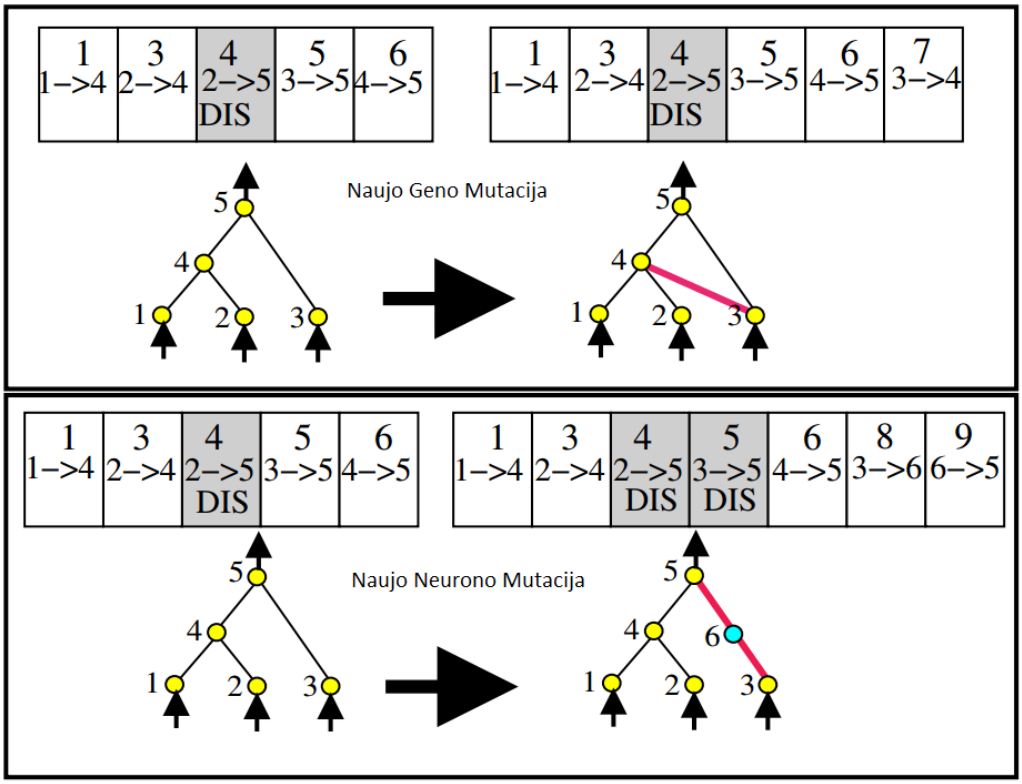
2 pav. Genetinio algoritmo veiksmų seka

Genetiniai algoritmai (toliau GA) 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][KW16, 3].

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.

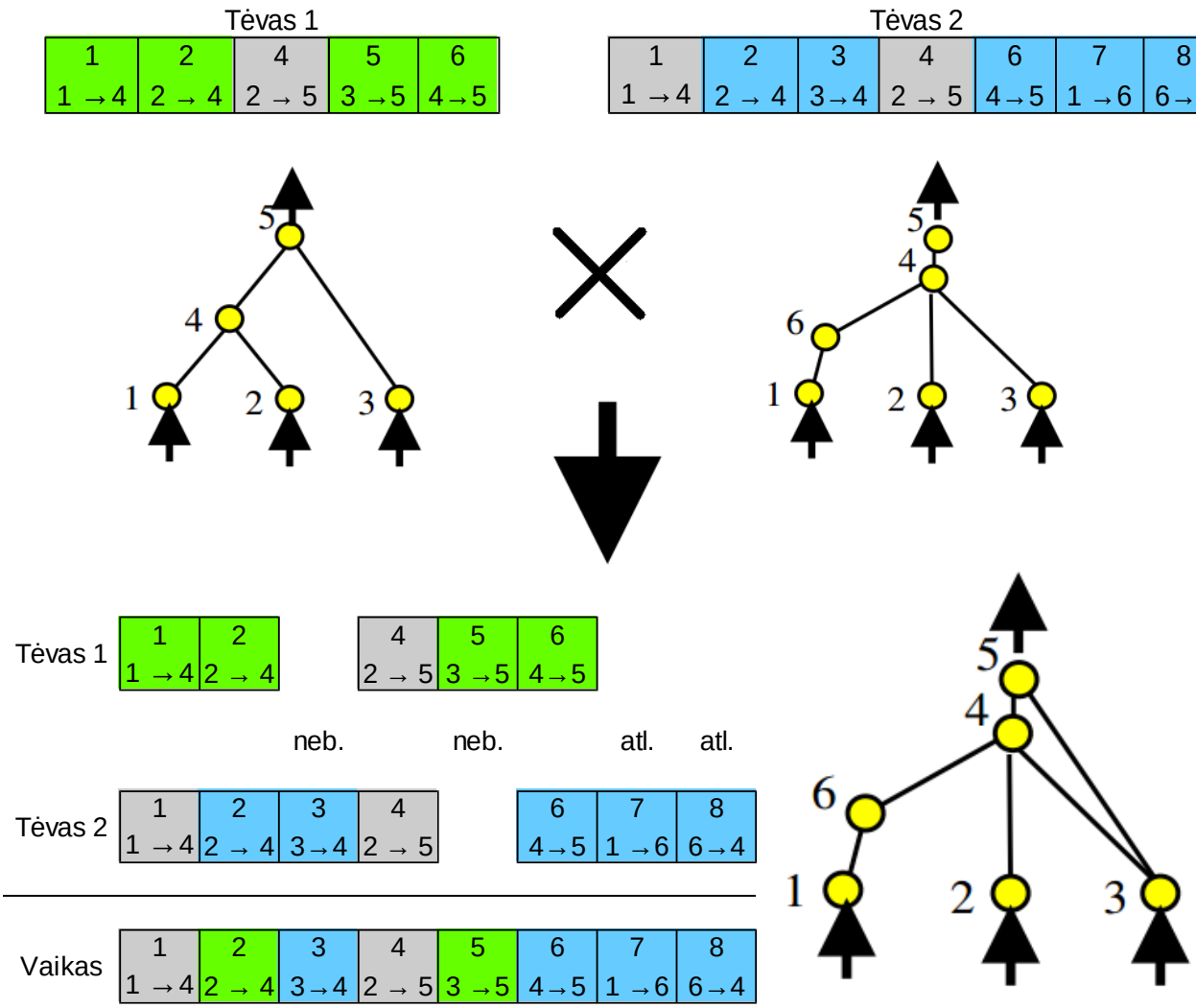
3.1 Mutacijos



3 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.

3.2 Kryžminimas



4 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]. Naudoti žymėjimai:

- neb. (nebendri) genai yra tie, kurių inovacijos žymės neviršija mažesniojo tėvo didžiausiosios.
- atl. (atliekami) genai yra tie, kurie viršija mažesniojo tėvo didžiausiąją inovacijos žymę.
- pilki genai yra išjungti

3.3 Konkuravimo paskirtymas rasėmis

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 suteikia paprastą būdą šiai problemai spręsti. Genomų panašumo funkcija:

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

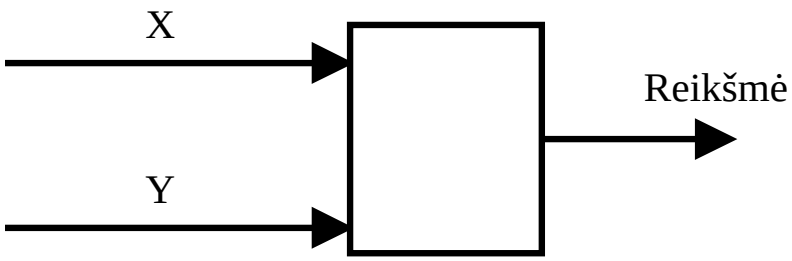
E - atliekamų (angl. *excess*) genų skaičius, D - nebendrų (angl. *disjoint*) genų skaičius, W - sutampančių genų svorių skirtumų vidurkis, N - didesniojo genomo genų skaičius, c_1, c_2, c_3 nustato šių daugiklių įtaką. „Kuo mažiau yra atliekamų ir nebendrų genų genomuose, tuo jie yra artimesni evoliucinės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

3.4 Informacijos abstrakcijos problema

Visa evoliucinių algoritmų idėja daugiau ar mažiau remiasi procesu, kuris vyko ir tebevyksta gamtoje. Stebinantis gamtos evoliucijos rezultatas paskatina algoritmus kurti modeliuojant jos procesus. Viena iš svaresnių problemų genetiniuose algoritmuose yra kodavimas. Kaip informacinėmis priemonėmis įgyvendinti tai, ką sugeba gyvuose organizmuose esantis DNR? T.y. užkoduojamos visos organizmo detalės palyginant mažame informacijos kiekyje.

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.

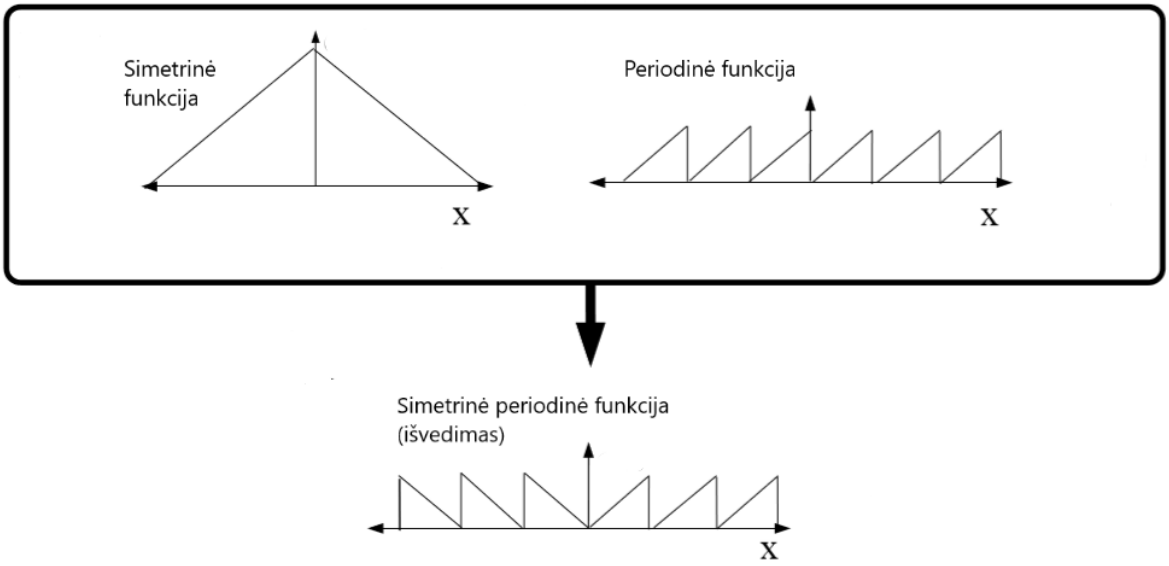


5. pav. Paprastas erdvės taško kodavimas

Tokiu būdu galima išsaugoti neribotos rezoliucijos paveikslėlį:

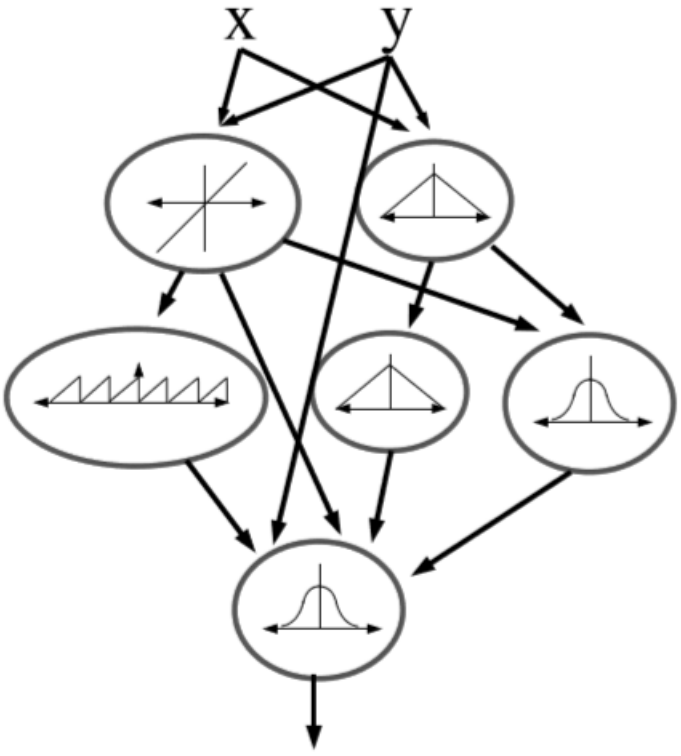
- Paduodamos koordinatės yra normalizuojamos [-1, 1], ar kitokiame intervale (centras yra 0)
- Galima įtraukti atstumo nuo centro reikšmę (lengviau atvaizduoti skritulius)
- Galima turėti kelias išvedimo reikšmes (ryškumui, spalvai, tamsumui)

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.



6. pav. Skirtingų aktyvacijos 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, dabar turime įrankį, kuris yra informacijos erdvėje abstrakcija. Tačiau kaip tai padeda su DNT?



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

5 Hiperkubinė Neuro-Evoliucija augančioms topologijoms (HyperNEAT)

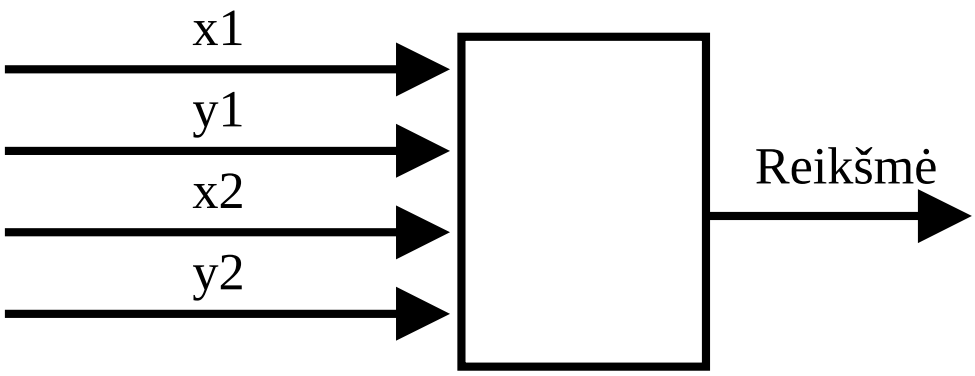
Ši NEAT algoritmo versija yra sukurta Kenneth O. Stanley, David D'Ambrosio ir Jason Gauci [KDJ09,NFAQ]. HyperNEAT (angl. Hypercube-based NeuroEvolution of Augmenting Topologies) algoritmo siekis yra išspręsti dvi DNT problemas:

- Efektyvi informacijos abstrakcija
- Erdvės suvokimas

Dauguma DNT 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.

5.1 Erdvės dėsningumų transformavimas į DNT lankus

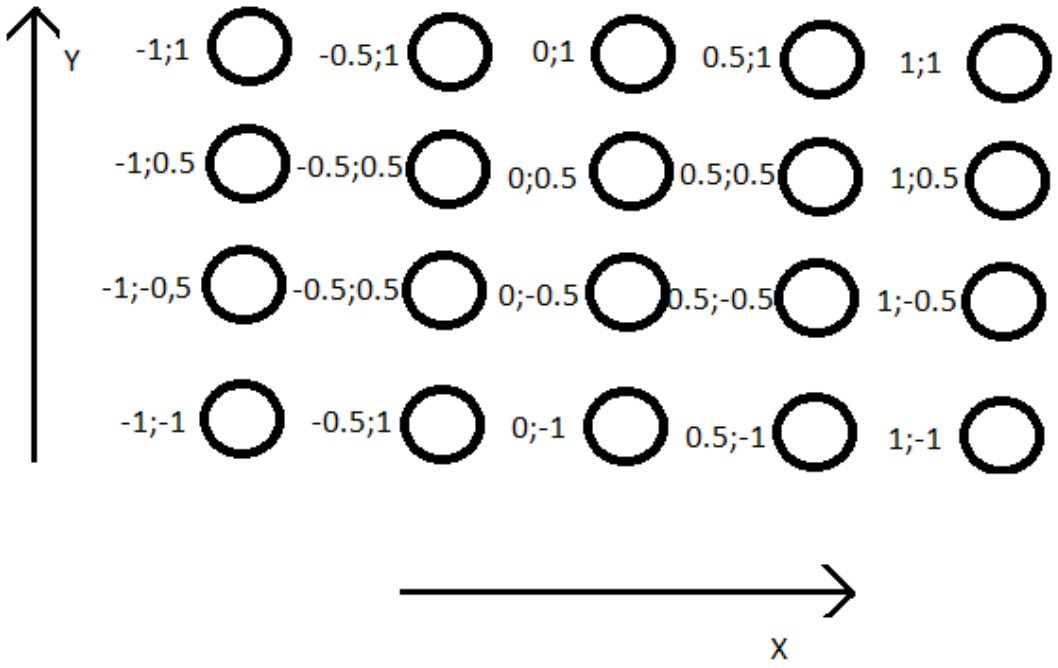
HyperNEAT algoritmas pasitelkia jau minėtu informacijos abstrakcijos įrankiu erdvėje: CPPN. Šiuo atveju, CPPN yra pagrindinis tinklas, kuris dalyvauja originaliam NEAT algoritme ir yra naudojamas kito tinklo generavimui. Žinoma, norint įgyvendinti kažką panašaus į 7 pav. būtina naujo tipo mutacija. Taigi, mutacijų sąrašą papildo aktyvacijos funkcijos tipo mutacija. Taip pat reikia CPPN pritaikyti naujo DNT generavimui. Pavyzdžiui, paprastas sluoksniuotas dviejų dimensijų DNT gali būti generuojamas tokio CPPN pagalba:



8. pav. CPPN pritaikytas 2 dimensijų DNT generavimui

Tokiu atveju, DNT įgyja stačiakampio formą:

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



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

Taigi, dabar generuojamo DNT neuronų struktūra yra statinė, tik kinta jungčių reikšmės. Kiekvienas sluoksnis yra pilnai jungus su šalia esančiais. Kaip pav. 10 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.

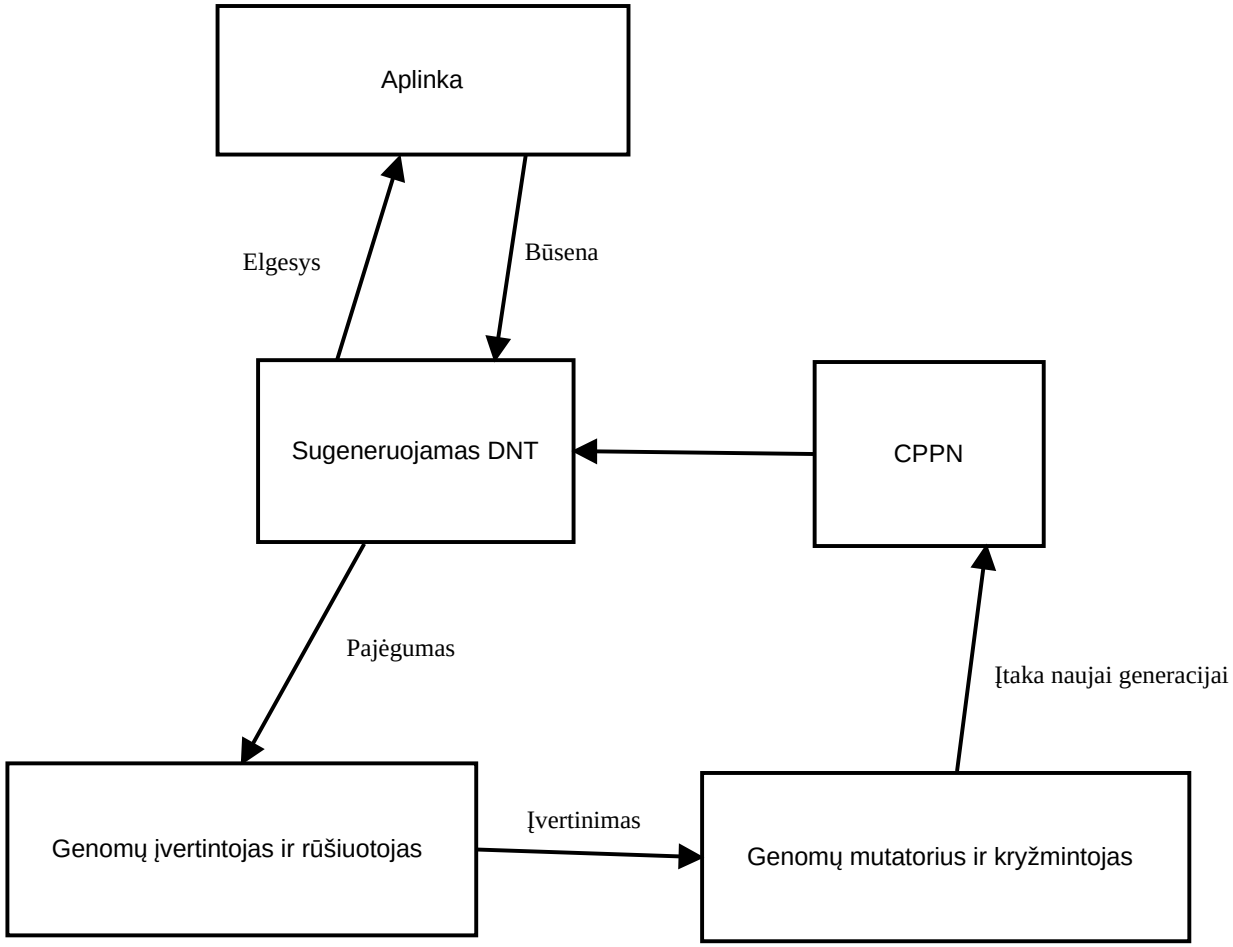
5.2 HyperNEAT masiškumas

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. Taip pat galima tinklą apmokyti su didelio masto sluoksniais ir po to pritaikyti mažesnio masto DNT generavimui, bei atvirkščiai.

Taip pat nebūtina apsiriboti stačiakampiais ir kvadratais, nes tai tik vienas iš galimų HyperNEAT modelių. Šį algoritmą galima adaptuoti apskritimo formai naudojant polines koordinates, jeigu to reikalautų problemos sritis pvz.: jūrų žvaigždės formos genomo elgesiui modeliuoti [KDJ09,11].

5.3 HyperNEAT algoritmo ciklas



10. pav. HyperNEAT algoritmo ciklo modelis

HyperNEAT algoritmo ciklas panašus kaip ir NEAT, tik yra vienas papildomas žingsnis. Evoliucijoje dalyvauja vidinis CPPN, tačiau pajėgumas įvertinimas pagal sugeneruotą išorinį DNT, kuris ir yra naudojamas duotai užduočiai spręsti.

6 NEAT ir HyperNEAT pritaikymas valdikliui

6.1 Užduoties formulavimas

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 įverčio.

6.2 Valdiklis video žaidimui „Tetris“

Valdikliui natūrali ir lengvai pritaikoma sritis – video žaidimai. Visame pasaulyje žinomas, daugiausiai kartų (daugiau negu 100 milijonų) parduotas video žaidimas – Tetris. Jeigu yra tokių, kurie nežino šio žaidimo:

- Žaidimas vyksta ribotos erdvės „šulinyje“
- Po vieną krenta įvairios kampuotos figūros, sudarytos iš 4 kvadratėlių
- Pasiekus dugną arba jau gulinčią figūrą, atsiranda kita figūra
- Reikia iš jų sudaryti horizontalias linijas
- Sudarius liniją (-as), ji (-os) išnyksta ir viskas kas buvo virš jos, nukrenta per išvalytų linijų skaičių
- Gaunami taškai už išvalytas linijas (kuo daugiau išvaloma per 1 ėjimą, tuo daugiau taškų)
- Gaunami taškai už padarytą ėjimą ir už paspartintą figūros nuleidimą
- Yra 4 galimi ėjimai, kurie valdo krentančią figūrą:
  - Figūros pajudinimas per vieną langelį į kairę/dešinę
  - Figūros pasukimas
  - Figūros paspartintas kritimas
- Žaidimas baigiasi kada, kada nauja figūra negali atsirasti, nes ją blokuoja

Taigi, iš tokių žaidimo taisyklių galima įgyvendinti štai tokį valdiklį:

- Įvedimas – visas „šulinio“ vaizdas.
- Išvedimas – ėjimo tipas.

6.2.1 Konkretus valdiklio įgyvendinimas

Tradiciniame video žaidime Tetris, figūros krito numatytu laiko intervalu (greitėjančiu), tačiau paprastumo dėlei figūra krenta kaip tik valdiklis padaro veiksmą. Naudotos konkrečios „Tetris“ žaidimo implementacijos „šulinio“ gylis yra 22 langeliai, o plotis 10. Taigi, valdiklio įvedimo neuronų yra 220. Kadangi galimi ėjimai yra 4, tiek ir yra išvedimo neuronų. Ėjimas nusprendžiamas pagal tai, kuris išvedimo neuronas turi aukščiausią reikšmę. Įvedimas formuojamas taip:

- 0, jeigu langelis tuščias
- 1, jeigu langelis nėra tuščias
- -1, jeigu langelis yra „skylė“

Langelis yra „skylė“, jeigu jis yra tuščias, tačiau yra netuščių nejudančių langelių virš jo. Pajėgumui matuojamas pagal žaidimo taškų skaičių:

- Padarytas ėjimas +1
- Paspartintas kritimas +10
- Išvalyta eilė +1000
- Išvalytos 2 eilės + 3000
- Išvalytos 3 eilės + 5000
- Išvalytos 4 eilės + 8000

Įvedama papildoma pabaigos sąlyga (laimėjimo sąlyga), kada taškų kiekis viršyja 500000. Taigi NEAT algoritmui įgyvendinti tiek ir tereikia žinoti.

6.2.2 Adaptavimas HyperNEAT

Kadangi įvedimą siekiama išsaugoti, natūralu naudoti 3 dimensijų HyperNEAT variaciją, kai sluoksniai yra 2 dimensijų. Sluoksnis šiuo atveju tampa stačiakampis kurio ilgis 10, o aukštis 22 (pagal žaidimo išmatavimus). Kuo daugiau sluoksnių, tuo tikslesnio galima tikėtis sprendimo, tačiau smarkiai padidėja DNT generavimui skirtas laikas. Galiausiai, kadangi išvedimas tapo nebe 4, o 220 neuronų, reikia naujo ėjimo interpretacijos metodo. Vienas iš galimų – suskirstyti išvedimo sluoksnį į 4 regionus, vieną regioną atitinka 55 neuronai. Kai turime 4 regionus (vienas regionas atitinka vieną ėjimo tipą), aukščiausio vidurkio regionas ir bus padarytas ėjimas.

6.3 Rezultatai

Pilnas žaidimas „Tetris“ buvo per sunkus NEAT ir HyperNEAT algoritmams, tačiau paprastesnės jo versijos pasirodė įveikiamos. Kadangi pajėgumas matuojamas pagal gautus taškus, pirmiausia abu algoritmai suoptimizavo figūrų greitą kritimą. Vėliau „šulinio“ užpildymą. Kaip matome, lengviausią pajėgumo padidinimą pasiekė lengvai. Eilučių išvalymas yra sudėtingesnis veiksmas, iš kelių tarpinių žingsnių. Šie žingsniai (sudaryti eilutę) atskirai nėra teigiamai įvertinami (nėra skiriama taškų), todėl ir sunkiau atsitiktinai tokį veiksmą atlikti. Kadangi neuro-evoliucijos pagrindas yra atsitiktinumai, tai ir rezultatai gali labai skirtis.

Pati paprasčiausia žaidimo versija – naudojant tik dėžės figūrą. Tada figūros pasukimas efekto neturi ir lieka tik 3 ėjimai. Tokiu atveju dauguma NEAT valdiklių pasiekia laimėjimo sąlyga neviršijant 100 generacijų. Sudėtingesnė žaidimo versija – naudojant tik linijos figūrą. Analogiškai kaip ir dėžės versija, tik pergalės sąlyga pasiekiama ne visada, dažnai viršija 100 generacijų ribą. Kitos žaidimo versijos (su kitomis figūromis) pasirodė neįveikiamos, per praktišką laiko kiekį.

HyperNEAT rezultatai buvo mažiau ištirti, nes dėl DNT generavimo, procesas vyksta labai lėtai. Dėžės versijoje progresas vyko labai greitai (per pirmas 8 generacijas buvo pasiekta daugiau negu 20000 taškų), bet realiu laiku užtruko gana ilgai. NEAT versija yra efektyvesnė vien tik dėl mažesnio skaičiavimų kiekio.

Taigi, jeigu normas rezultatas buvo evoliucionuoti valdiklį žaisti žaidimą „Tetris“, tuomet rezultatas nepasiektas, tačiau jeigu buvo norima rasti būdą/įrankį, kuris leistu kompiuteriui aptikti žaidimo taisykles vien tik iš elgesio įvertinimų, rezultatas buvo pasiektas. Nereikia papildomų pokyčių algoritmui pritaikyti žaisti skirtingas „Tetris“ versijas. Jeigu valdiklio logiką reikėtų įgyvendinti rankiniu būdu, ji būtų kitokia skirtingoms „Tetris“ versijoms.

Eksperimentų detalės:

- NEAT algoritmo taikymo metu populiacija buvo 200.
- HyperNEAT algoritmo taikymo metu populiacija buvo 100.
- Abiem atvejais norimas rasių kiekis buvo 5, bei kiti mutacijų parametrai išliko vienodi.

Algoritmų implementacijos pateikiamos priede.

7 Išvados

Neuro-Evoliucija (NE) yra galingas įrankis 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.

CPPN yra galingas informacijos abstrakcijos įrankis, leidžiantis išsaugoti struktūras erdvėje ir gali būti nesunkiai plečiamas į aukštesnes dimensijas. Pasitelkus CPPN yra įmanomas HyperNEAT algoritmas, kuris bando išspręsti informacijos abstrakcijos ir erdvės suvokimo problemą vystant DNT. Žinoma, HyperNEAT yra prasmės naudoti tik ten, kur yra geometriniai dėsniai. Taip pat didelių DNT generavimas kainuoja daug resursų, todėl matuojant realiu laiku NEAT gali būti greitesnis, tačiau matuojant generacijų skaičiumi, HyperNEAT dažniausiai bus greitesnis ar bent toks pat greitas kaip ir NEAT.

8 Priedai

8.1 NEAT algoritmo implementacija Java kalba

```
public class CrossoverList {
    public int excessIndexStart = -1;
    public int matchingGene = 0;
    public double matchingWeightSum = 0.0;
    public int disjointBias = 0;
    public double biasSum = 0.0;
    public ArrayDeque<Pair<? extends Gene>> geneList = new ArrayDeque<>();
    public ArrayList<Pair<? extends NeuronInfo>> biasList = new ArrayList<>();
    public static class Pair<T>{
        public T g1 = null;
        public T g2 = null;
        public Pair(T g1, T g2){
            this.g1 = g2;
            this.g2 = g2;
        }
        public Pair(){

        }
        public boolean full(){
            return this.g1 != null && this.g2 != null;
        }
        public T getRandom(){
            if(full()){
                if(F.RND.nextBoolean()){
                    return g1;
                }else{
                    return g2;
                }
            }else{
                if(g1==null){
                    return g2;
                }else{
                    return g1;
                }
            }
        }
    }
```

```
public CrossoverList(Genome net1, Genome net2){
   ArrayList<Gene> list1 = new ArrayList<>();
   ArrayList<Gene> list2 = new ArrayList<>();
   prepareList(net1,list1);
   prepareList(net2,list2);
   int i1 = 0;
   int i2 = 0;
   int combinedSize = list1.size() + list2.size();
   while( i1 + i2 < combinedSize){
      Pair<Gene> pair = new Pair<>();
      if(i1 >= list1.size()){
         if(excessIndexStart == -1){
            excessIndexStart = geneList.size();
         }
         pair.g2 = list2.get(i2);
         i2++;
      }
      else if(i2 >= list2.size()){
         if(excessIndexStart == -1){
            excessIndexStart = geneList.size();
         }
         pair.g1 = list1.get(i1);
         i1++;
      }
      else{
         Gene g1 = list1.get(i1);
         Gene g2 = list2.get(i2);
         int cmp = ArrayBasedCounter.compareCounterAscending.compare(g1.inn, g2.inn);
         if(cmp == 0){
            pair.g1 = g1;
            pair.g2 = g2;
            i1++;
            i2++;
            matchingGene++;
            this.matchingWeightSum += Math.abs(g1.w - g2.w);
         }
         else if (cmp < 0){
            pair.g1 = g1;
            i1++;
         }
         else{
            pair.g2 = g2;
            i2++;
         }
      }
```

```
            geneList.add(pair);
        }

        //BIAS pairing
        ArrayList<? extends NeuronInfo> bigger,smaller;
        if(net1.bias.size() >= net2.bias.size()){
            bigger = net1.bias;
            smaller = net2.bias;
        }else{
            bigger = net2.bias;
            smaller = net1.bias;
        }
        for(int i = 0; i < smaller.size(); i++){
            Pair<NeuronInfo> pair = new Pair<>(bigger.get(i),smaller.get(i));
            this.biasList.add(pair);
        }
        for(int i = smaller.size(); i < bigger.size(); i++){
            Pair<NeuronInfo> pair = new Pair<>();
            pair.g1 = bigger.get(i);
            this.biasList.add(pair);
        }
        this.disjointBias = bigger.size() - smaller.size();
    }


    private void prepareList(Genome net,ArrayList list){
        list.clear();
        for(Gene l:net.genes){
            list.add(l.clone());
        }
    }
}


public class Gene extends Synapse implements Comparable, Cloneable{
    public boolean en       = true;
    public int[]  inn      = null;

    public Gene(int in, int out){
        this.in = in;
        this.out = out;
    }
    public Gene(){};
    @Override
    public Object clone(){
```

```
        final Gene gene;
        gene = (Gene) super.clone();

        return gene;
    }

    @Override
    public int compareTo(Object o) {
        Gene other = (Gene)o;
        return ArrayBasedCounter.compareCounterAscending.compare(inn, other.inn);
    }

}


public class Genome implements Cloneable{
    public transient boolean needUpdate = false;
    public static Comparator<Genome> fitnessAscending = new Comparator<Genome>() {
        @Override
        public int compare(Genome o1, Genome o2) {
            return Double.compare(o1.fitness, o2.fitness);
        }
    };
    public static Comparator<Genome> fitnessDescending = new Comparator<Genome>() {
        @Override
        public int compare(Genome o1, Genome o2) {
            return Double.compare(o2.fitness, o1.fitness);
        }
    };
    public PriorityQueue<Gene> genes = new PriorityQueue<>();
    protected transient NeuralNetwork network;
    public ArrayList<NeuronInfo> bias = new ArrayList<>();
    public static Map<Integer,ActivationFunction> activationMap = F.getDefaultActivationMap();
    public float fitness;
    public transient int globalRank;
    public int input, output;
    public Genome(int input, int output){
        this.input = input;
        this.output = output;
        for(int i = 0; i < input + output; i++){
            bias.add(new NeuronInfo());
        }
    }

    public Genome(){};
```

```
@Override
public Object clone(){

   try {
      Genome genome = (Genome) super.clone();
      ArrayList<Gene> list = new ArrayList<>(this.genes);
      genome.genes = new PriorityQueue<>();
      genome.bias = new ArrayList<>();
      for(Gene g:list){
         genome.genes.add((Gene) g.clone());
      }
      for(NeuronInfo info:this.bias){
         genome.bias.add((NeuronInfo) info.clone());
      }
      return genome;
   } catch (CloneNotSupportedException ex) {
      throw new AssertionError();
   }
}

public NeuralNetwork getNetwork(){
   if(needUpdate || network == null){
      this.generateNetwork();
   }
   return network;
}
public NeuralNetwork generateNetwork(){
   if(bias.size()<input + output){
      for(int i = 0; i < input + output; i++){
         bias.add(new NeuronInfo());
      }
   }
   network = new NeuralNetwork(input,output,new ArrayList<>(genes),bias,activationMap,F::sigmoid);
   needUpdate = false;
   return network;
}
public Double[] evaluate(Double[] input){
   return getNetwork().evaluate(input);
}
public double[] evaluate(double[] input){
   Double[] input1 = new Double[input.length];
   for(int i = 0; i < input.length; i++){
      input1[i] = input[i];
   }
```

```
        Double[] output1 = evaluate(input1);
        double[] outputar = new double[output1.length];
        for(int i = 0; i < output1.length; i++){
            outputar[i] = output1[i];
        }
        return outputar;
    }

}


public class Pool implements Serializable{
    public double CROSSOVER = 0.7;
    public double SELECTION = 0.3;

    public double DELTA_DISJOINT = 1;
    public double DELTA_EXCESS = 1;
    public double DELTA_WEIGHTS = 0.4;

    public double similarity = 1;

    public transient double[] similarities;
    public double similarityChangeRate = 0.12;
    public int distinctSpecies = 5;

    public int maxStaleness = 20;
    public int populationSize = 200;
    public int generation = 0;

    public transient GenomeMutator mutator;
    public Genome allTimeBest;

    public LinkedList<Genome> bestGenomes;
    public ArrayBasedCounter innovation;
    public ArrayList<Species> species;


    private void addBest(Genome g){

        if(this.allTimeBest == null || this.allTimeBest.fitness <= g.fitness) {
            this.allTimeBest = g;
        }
        bestGenomes.addLast(g);
    }
    public Genome getCurrentBest(){
```

```
    return bestGenomes.getLast();
}



public double similarity(Genome net1, Genome net2){
    double e = Double.MIN_VALUE;
    CrossoverList cross = new CrossoverList(net1, net2);
    double N = Math.max(net1.genes.size(), net2.genes.size());
    double D = ((double)cross.excessIndexStart - cross.matchingGene) * this.DELTA_DISJOINT / N;
    double E = ((double)cross.geneList.size() - cross.excessIndexStart) * this.DELTA_EXCESS / N;
    double W = ((double)cross.matchingWeightSum / (cross.matchingGene+e)) * this.DELTA_WEIGHTS;
    double DB = (double)cross.disjointBias;
    double B = cross.biasSum / ((cross.biasList.size() - cross.disjointBias)+e);

    return D+E+W+DB+B;
}

public Genome crossover(List<Genome> list){
    Genome child;
    LinkedList<Genome> parents = F.pickRandomPreferLow(list, 2,list.size(),1);
    child = (Genome) parents.peekFirst().clone();
    child.bias.clear();
    child.genes.clear();
    CrossoverList cross = new CrossoverList(parents.peekFirst(), parents.peekLast());
    for(Pair<? extends NeuronInfo> pair:cross.biasList){
        child.bias.add((NeuronInfo) pair.getRandom().clone());
    }
    for(Pair<? extends Gene> pair:cross.geneList){
        child.genes.add(pair.getRandom());
    }
    return child;
}

public Genome breedChild(List<Genome> list){
    int size = list.size();
    Genome child;
    if(size > 1 && F.RND.nextDouble() < CROSSOVER){
        child = crossover(list);
    }else{
        child = (Genome) list.get(F.RND.nextInt(size)).clone();
    }
    this.mutator.mutate(child);
    return child;
}
```

```
public void assignToSpecies(Genome g){
   int toAssign = -1;
   Double bestSimilarity = Double.MAX_VALUE;
   double[] sim = new double[species.size()];
   int index = 0;
   if(sim.length>0){
      for(Species s:species){
         final int i = index++;
         sim[i] = similarity(s.getLeader(), g);
      }

      for(int i = 0; i < sim.length; i++){
         double d = sim[i];
         this.similarities[1] = Math.max(this.similarities[1], d);
         this.similarities[0] = Math.min(this.similarities[0], d);
         if(d < this.similarity && bestSimilarity > d){
            bestSimilarity = d;
            toAssign = i;
         }
      }
   }

   if(toAssign==-1){
      Species spe = new Species();
      spe.genomes.add(g);
      species.add(spe);
   }else{
      species.get(toAssign).genomes.add(g);
   }

}

public Genome rankGlobally() {
   ArrayList<Genome> global = new ArrayList<>();

   global.addAll(this.getPopulation());
   Collections.sort(global, Genome.fitnessAscending);

   int rank = 1;
   for(Genome g:global){
      g.globalRank = rank++;
   }
   return global.get(global.size()-1);
```

```
}

public double totalSpeciesAvgRank() {
   double total = 0;
   for (final Species s : species){
      total += s.avgRank;
   }
   return total;
}

public double calculateSimilarityThreshold(){
   if(species.size() > this.distinctSpecies){
      this.similarity *= (1 + this.similarityChangeRate);
   }
   else if(species.size() < this.distinctSpecies){
      this.similarity *= (1 - this.similarityChangeRate);
   }
   similarity = Math.max(similarity, similarities[0]);
   similarity = Math.min(similarity, similarities[1]);
   return similarity;
}

public void newGeneration(){
   final Species[] bestSpecies = new Species[1];
   bestSpecies[0] = species.get(0);
   this.generation++;
   this.similarities = new double[]{Double.MAX_VALUE,Double.MIN_VALUE};
   ConcurrentLinkedDeque<Species> survived = new ConcurrentLinkedDeque<>();
   //cull species
   int index = 0;
   for(final Species s:species){
      s.id = index++;

         if(s.genomes.isEmpty()){
            continue;
         }
         if(!s.genomes.isEmpty()){

            if(s.getLeader().fitness > s.bestFitness){
               s.staleness = 0;
               s.bestFitness = s.getLeader().fitness;
            }else{
               s.staleness++;
            }
            if(bestSpecies[0].bestFitness < s.bestFitness){
```

```
            bestSpecies[0] = s;
          }
          survived.add(s);
        }


}

species.clear();
species.addAll(survived);
survived.clear();

//romove stale
Genome best = null;
for(Species s:species){
   if(s.staleness < this.maxStaleness || s.bestFitness >= this.getCurrentBest().fitness || s.id == bestSpecies[0].id){
      survived.add(s);

      if(best == null || best.fitness < s.getLeader().fitness){
         best = s.getLeader();
      }
   }
}
this.addBest(best);
species.clear();
species.addAll(survived);
survived.clear();

rankGlobally();
for(Species s:species){
   s.calculateAverageRank();
}

ArrayList<Genome> leaders = new ArrayList<>();

//remove weak
double sum = this.totalSpeciesAvgRank();

for(Species s:species){

   double breed = Math.floor(s.avgRank / sum * this.populationSize);
   if(breed >= 1.0 || s.id == bestSpecies[0].id){
      survived.add(s);
      leaders.add(s.getLeader());
   }
```

```
    }

    species.clear();
    species.addAll(survived);
    survived.clear();


    Collections.sort(leaders,Genome.fitnessDescending);

    ConcurrentLinkedDeque<Genome> newGeneration = new ConcurrentLinkedDeque<>();
    //new generation
    for(Species s:species){
       double breed = Math.floor(s.avgRank / sum * this.populationSize) - 1;
       for(int i = 0; i < breed; i++){
          newGeneration.add(this.breedChild(s.genomes));
       }
    }

    int left = this.populationSize - leaders.size() - newGeneration.size();

    //crossover leaders
    for(int i = 0; i < left; i++){
       newGeneration.add(breedChild(leaders));
    }
    for(Genome g:newGeneration){
       this.assignToSpecies(g);
    }

    int specCount = species.size();

    this.calculateSimilarityThreshold();

    double ratio = (double)Math.max(specCount, distinctSpecies);
    ratio /= (double)Math.min(specCount, distinctSpecies);
    if(ratio > 1.5){//reassign
       species.clear();
       for(Genome g:leaders){
          assignToSpecies(g);
       }
       for(Genome g:newGeneration){
          assignToSpecies(g);
       }
    }
 }
 public void recreateSpecies(Collection<Genome> ancestors){
```

```
    species.clear();
    for(Genome ancestor:ancestors){
        Species s = new Species();
        s.genomes.add(ancestor);
        species.add(s);
    }
}
public void newGeneration(Genome ancestor){
    ArrayList<Genome> list = new ArrayList<>(1);
    list.add(ancestor);
    newGeneration(list);
}
public void newGeneration(Collection<Genome> ancestors){
    this.bestGenomes.clear();
    recreateSpecies(ancestors);
    equalCloning(ancestors);
}

public void equalCloning(Collection<Genome> ancestors){
    int size = ancestors.size();
    int sizePerSpecies = Math.max(this.populationSize / size,1);
    int left = this.populationSize - size;

    while(left > 0){
        for(Genome ancestor:ancestors){
            int create = Math.min(sizePerSpecies,left);
            for(int i = 0; i < create; i++){
                Genome g = (Genome) ancestor.clone();
                mutator.mutate(g);
                this.assignToSpecies(g);
            }
            left-=create;
            if(left == 0){
                break;
            }
        }
    }
}

public List<Genome> getPopulation(){
    ArrayList<Genome> pop = new ArrayList<>(this.populationSize);
    for(Species s:species){
        pop.addAll(s.genomes);
    }
    return pop;
```

```
    }

    public Pool(GenomeMaker maker, GenomeMutator mutator){
        this();
        this.mutator = mutator;
        Collection<Genome> gen = maker.initializeGeneration();
        this.populationSize = gen.size();
        for(Genome g:gen){
            mutator.mutate(g);
            assignToSpecies(g);
        }

    }


    public Pool(){
        species = new ArrayList<>();
        bestGenomes = new LinkedList<>();
        similarities = new double[]{Double.MAX_VALUE,Double.MIN_VALUE};
        this.innovation = new ArrayBasedCounter(1);
    }
    public void prepareToSerialize(){
        for(Species s:species){
            s.backup = (new ArrayList<>(s.cullSpecies(0, true)));
        }
    }
    public void restoreAfterSerialize(){
        for(Species s:species){
            s.genomes.addAll(s.backup);
        }
    }
    public void afterDeserialization(){
        recreateSpecies(this.getPopulation());
        equalCloning(this.getPopulation());
        for(Genome g:this.getPopulation()){
            g.needUpdate = true;
        }
    }
}

public class Species {
    public transient int id;
    public double bestFitness = 0.0;
    public transient double avgRank  = 0.0;
    public int staleness = 0;
```

```
    public transient ArrayList<Genome> backup = new ArrayList<>();
    public ArrayList<Genome> genomes = new ArrayList<>();
    public Genome getLeader(){
        return genomes.get(0);
    }

    public List<Genome> cullSpecies(double selection,boolean leave1){
        Collections.sort(genomes,Genome.fitnessDescending);
        int survivors = 1;
        if(!leave1){
            survivors = (int) Math.ceil(selection * genomes.size());
        }

        ArrayDeque<Genome> survived = new ArrayDeque<>(survivors);
        LinkedList<Genome> dead  = new LinkedList<>();
        dead.addAll(genomes);
        genomes.clear();

        for(int i = 0; i < survivors; i++){
            survived.add(dead.removeFirst());
        }
        genomes.addAll(survived);
        return dead;
    }
    public double calculateAverageRank() {
        double total = 0.0;
        for (final Genome genome : genomes)
            total += genome.globalRank;
        avgRank = total / genomes.size();
        return avgRank;
    }
    public int size(){
        return genomes.size();
    }
}
```

8.2 HyperNEAT adaptacija Java kalba

```
public class HyperGenome extends Genome implements Cloneable{

   public double maxLinkLength;
   private static double maxVal = 1d;
   private static double minVal = -1d;
   private transient NeuralNetwork generated;

   public transient HyperNEATSpace space;


   public HyperGenome(int... dimensions){
      super((dimensions.length -1) * 2,1);
      //{3,3,9} 3*3 and 9 layers

   }

   @Override
   public Object clone(){
      Object g = super.clone();
      HyperGenome genome = (HyperGenome)g;
      genome.space = this.space;
      return genome;
   }



   @Override
   public Double[] evaluate(Double[] input){
      if(needUpdate || generated == null){

         generateNetwork();
      }
      Double[] evaluate = generated.evaluate(input);
      return evaluate;
   }



   @Override
   public NeuralNetwork generateNetwork(){
      super.generateNetwork();

      ArrayList<Synapse> links = new ArrayList<>(space.getDimensionSum());
```

```
ArrayList<Pos> from;
ArrayList<Pos> to;
MinMax mm = new MinMax(minVal,maxVal);
for(int i = 0; i< space.layers.size()-1; i++){
   from = space.layers.get(i);
   to = space.layers.get(i+1);
   MinMax[] minmax = new MinMax[from.size()];

   for(int j=0; j<minmax.length; j++){
      minmax[j] = mm;
   }
   for(int iFrom = 0; iFrom < from.size(); iFrom++ ){
      Pos posFrom = from.get(iFrom);
      for(int iTo = 0; iTo < to.size(); iTo++){
         Pos posTo = to.get(iTo);
         Synapse syn = new Synapse();

         syn.in = space.idMap.get(posFrom);
         syn.out = space.idMap.get(posTo);
         int inputDim = posTo.dim() + posFrom.dim();
         Double[] inputs = new Double[inputDim];
         for(int t = 0; t < posFrom.dim(); t++){
            inputs[t] = posFrom.normalized(minmax)[t];
         }

         for(int t = posTo.dim(); t < inputDim; t++){

            inputs[t] = posTo.normalized(minmax)[t-posFrom.dim()];
         }
         Double[] evaluate = this.network.evaluate(inputs);
         syn.w = evaluate[0];
         double threshold = 0.2;
         if(Math.abs(syn.w)>threshold){
            links.add(syn);
         }

      }
   }
}
ArrayList<NeuronInfo> biasList = new ArrayList<>(space.getDimensionSum());
for(int i = 0; i < space.idMap.size(); i++){
   biasList.add(new NeuronInfo());
}

int layerSize = space.getLayerSize();
```

```
        this.generated = new NeuralNetwork(layerSize,layerSize, links, biasList);
        return this.network;

    }

}


public class HyperNEATSpace {
    public ArrayList<ArrayList<Pos>> layers;
    public HashMap<Pos,Integer> idMap;

    public int[] dimensions;

    public HyperNEATSpace(int... dimensions){
        this.dimensions = dimensions;
        this.initialPositions();
    }
    public int getLayerSize(){
        return layers.get(0).size();
    }

    public int getDimensionSum(){
        int dimSum = 1;
        for(int i = 0; i<dimensions.length; i++){
            dimSum*= dimensions[i];
        }
        return dimSum;
    }
    public final void initialPositions(){
        layers = new ArrayList<>();
        idMap = new HashMap<>();
        for(int i = 0; i < getLayers();i++){
            layers.add(new ArrayList<>());
        }

        for(int i = 0; i < getDimensionSum(); i++){
            Integer[] var = new Integer[dimensions.length];
            int t = i;
            for(int j = 0; j < dimensions.length; j++){
                var[j] = t % dimensions[j];
                t = t / dimensions[j];
            }
            Pos p = new Pos(var);
```

```
            int last = var[var.length-1];
            layers.get(last).add(p);
        }
        int i = 0;
        for(Pos P:layers.get(0)){
            idMap.put(P, i++);
        }
        for(Pos P:layers.get(getLayers()-1)){
            idMap.put(P, i++);
        }

        for(int j=1; j < layers.size()-1; j++){
            for(Pos p:layers.get(j) ){
                idMap.put(p, i++);
            }
        }
    }
    public Integer getLayers(){
        return dimensions[dimensions.length-1];
    }
}
```

8.3 Genomų mutacijos Java kalba

```
public interface GenomeMaker {
   public Collection<Genome> initializeGeneration();
}

public interface GenomeMutator {
   public void mutate(Genome genome);
}

public class DefaultNEATMutator implements GenomeMutator{
   public ArrayBasedCounter innovation;

   public double MUT_WEIGHT       = 1;
   public double MUT_WEIGHT_STEP   = 0.2;
   public double MUT_WEIGHT_RESET  = 0.1;

   public double MUT_ENABLE_TOGGLE = 0.1;

   public double MUT_BIAS         = 0.8;
   public double MUT_BIAS_STEP     = 0.2;
   public double MUT_BIAS_RESET    = 0.1;

   public double MUT_ENALBE       = 0.6;
   public double MUT_DISABLE      = 0.4;

   public double MUT_LINK         = 2;
   public double MUT_NODE         = 0.6;
   public double weightCap        = 20;

   public DefaultNEATMutator(){
      this.innovation = new ArrayBasedCounter(1);
   }

   private void mutateNode(Genome genome){
      Gene gene = F.pickRandom(genome.genes);
      gene.en = false;
      int neuronID = genome.bias.size();
      genome.bias.add(new NeuronInfo());

      Gene inputGene = new Gene(gene.in,neuronID);
      inputGene.inn = innovation.inc();
      inputGene.w = gene.w;
```

```
    Gene outputGene = new Gene(neuronID,gene.out);
    outputGene.inn = innovation.inc();
    outputGene.w = 1f;

    genome.genes.add(inputGene);
    genome.genes.add(outputGene);


}
private void mutateLink(Genome genome){
    Random RND = F.RND;
    NeuralNetwork network = genome.getNetwork();
    ArrayList<Integer> full = new ArrayList<>();
    ArrayList<Integer> candidatesInput = new ArrayList<>();
    HashSet<Integer> outputNodes = new HashSet<>();
    for(Neuron n:network.getOutputs()){
        outputNodes.add(n.ID);
    }
    for(Neuron n:network.neurons.values()){
        full.add(n.ID);
        if(!outputNodes.contains(n.ID)){
            candidatesInput.add(n.ID);
        }
    }
  int nextCandidate = RND.nextInt(candidatesInput.size());
    Neuron input = network.neurons.get(candidatesInput.get(nextCandidate));

    HashSet<Integer> parentSet = network.getParentSet(input.ID);
    parentSet.add(input.ID);
    for(Gene g:genome.genes){
        if(g.in == input.ID){
            parentSet.add(g.out);
        }
    }
    full.removeAll(parentSet);

    if(!full.isEmpty()){
        Neuron output = network.neurons.get(full.get(RND.nextInt(full.size())));
        Gene gene = new Gene(input.ID,output.ID);
        gene.inn = innovation.inc();
        genome.genes.add(gene);
    }
}

private void mutateEnableDisable(Genome genome,final boolean enable) {
```

```
    List<Gene> candidates = new ArrayList<>();
    for (Gene gene : genome.genes){
        if (gene.en != enable){
            candidates.add(gene);
        }
    }
    if (candidates.isEmpty()) return;

    final Gene gene = candidates.get(F.RND.nextInt(candidates.size()));
    gene.en = !gene.en;
}

@Override
public void mutate(Genome genome){

    Random RND = F.RND;
    double prob = this.MUT_LINK;
    while(prob > 0 || genome.genes.isEmpty()){
        if(RND.nextDouble() < prob){
            mutateLink(genome);
            genome.generateNetwork();
        }
        prob--;
    }
    if(MUT_NODE > RND.nextDouble()){
        mutateNode(genome);
    }
    if(MUT_WEIGHT > RND.nextDouble()){ //mutate weights
        for(Gene g:genome.genes){
            if(MUT_WEIGHT_RESET < RND.nextDouble()){
                g.w =  (2.0 * RND.nextDouble()- 1.0);
            }
            else{
                g.w +=  2.0 * RND.nextDouble()* MUT_WEIGHT_STEP - MUT_WEIGHT_STEP;
            }
            g.w =  Math.min(g.w,this.weightCap);
            g.w =  Math.max(g.w, -this.weightCap);
        }
    }
    if(MUT_ENALBE > RND.nextDouble()){
        this.mutateEnableDisable(genome, true);
    }
    if(MUT_DISABLE > RND.nextDouble()){
        this.mutateEnableDisable(genome, false);
    }
```

```
        if(MUT_BIAS > RND.nextDouble()){
            int index = RND.nextInt(genome.bias.size());
            NeuronInfo val = genome.bias.get(index);
            if(MUT_BIAS_RESET < RND.nextDouble()){
                val.bias = 2 * RND.nextDouble() - 1;
            }else{
                val.bias += 2 * RND.nextDouble() - 1;
            }
            val.bias %= this.weightCap;
        }
        genome.needUpdate = true;
    }
}


public class DefaultHyperNEATMutator implements GenomeMutator{
    public double MUTATE_ACTIVE_FUNCTION = 0.3;
    private GenomeMutator neatMutator = new DefaultNEATMutator();
    @Override
    public void mutate(Genome genome) {
        neatMutator.mutate(genome);
        if(F.RND.nextDouble() < MUTATE_ACTIVE_FUNCTION){
            int index = F.RND.nextInt(genome.bias.size());
            NeuronInfo get = genome.bias.get(index);
            get.afType = F.RND.nextInt(Genome.activationMap.size());
            genome.needUpdate = true;
        }
    }


}
```

8.4 Bendro DNT implementacija Java kalba

```
public interface ActivationFunction {
   public double activate(double d);
}

public class NeuralNetwork {
   public int inputs,outputs;
   public HashMap<Integer,Neuron> neurons = new HashMap<>();


   public NeuralNetwork(int inputs, int outputs, Collection<? extends Synapse> genes, Collection<NeuronInfo> biases){
      this(inputs, outputs, genes, biases, new HashMap<>(),F::sigmoid);
   }

   public NeuralNetwork(int inputs, int outputs, Collection<? extends Synapse> links, Collection<NeuronInfo> biases,
         Map<Integer,ActivationFunction> activationMap, ActivationFunction defaultActivation){
      this.inputs = inputs;
      this.outputs = outputs;
      int neuronID = 0;
      for(NeuronInfo val:biases){
         Neuron n = new Neuron(neuronID++);
         n.bias = val.bias;
         neurons.put(n.ID, n);
         if(activationMap.containsKey(val.afType)){
            n.af = activationMap.get(val.afType);
         }else{
            n.af = defaultActivation;
         }
      }
      for(Synapse g:links){
         Neuron get = neurons.get(g.out);
         if(get==null){
            throw new Error("NULL NEURON");
         }else{
            get.addLink(g);
         }

      }
   }

   public Double[] evaluate(Double[] inputs){
      Double[] output = new Double[this.outputs];
      for(Neuron n:neurons.values()){
         n.value = null;
```

```
    }
    ArrayList<Neuron> in = getInputs();
    for(int i = 0; i < this.inputs; i++){
        Neuron n = in.get(i);
        n.value = n.af.activate(inputs[i] + n.bias);
    }
    ArrayList<Neuron> out = getOutputs();
    for(int i = 0; i < this.outputs; i++){
        output[i] = out.get(i).resolve(neurons);
    }
    return output;
}

public final ArrayList<Neuron> getInputs(){
    ArrayList<Neuron> list = new ArrayList<>();
    for(int i = 0; i<inputs; i++){
        list.add(neurons.get(i));
    }
    return list;
}

public final ArrayList<Neuron> getOutputs(){
    ArrayList<Neuron> list = new ArrayList<>();
    for(int i = 0; i<outputs; i++){
        list.add(neurons.get(i+inputs));
    }
    return list;
}

public HashSet<Integer> getParentSet(int ID){
    HashSet<Integer> newSet = new HashSet<>();
    HashSet<Integer> visitNextIteration = new HashSet<>();
    HashSet<Integer> parentSet = new HashSet<>();
    visitNextIteration.add(ID);
    while(!visitNextIteration.isEmpty()){
        newSet.clear();
        for(Integer up:visitNextIteration){
            if(parentSet.contains(up)){
                continue;
            }
            parentSet.add(up);
            Neuron parent = neurons.get(up);
            newSet.addAll(parent.input.keySet());
        }
        visitNextIteration.clear();
```

```
            visitNextIteration.addAll(newSet);
        }
        return parentSet;
    }

}


public class Neuron {
    public int ID;
    public double bias;
    public Double value = null;
    public ActivationFunction af;
    public HashMap<Integer,Double> input = new HashMap<>();
    public Neuron(int i){
        ID = i;
    }

    public void addLink(Synapse g){
        if(g.out == ID){
            input.put(g.in,(double)g.w);
        }
    }

    public double resolve(HashMap<Integer,Neuron> neurons){
        if(value == null){
            value = 0d;
            double d = 0;
            for(Map.Entry<Integer, Double> entry:input.entrySet()){
                d += entry.getValue() * neurons.get(entry.getKey()).resolve(neurons);
            }
            value = af.activate(d + bias);
        }
        return value;
    }
}

public class NeuronInfo implements Cloneable{
    public Double bias;
    public Integer afType;
    public NeuronInfo(){
        bias = 0d;
        afType = null;
    }
    @Override
```

```
    public Object clone(){
        NeuronInfo b = null;
        try {
            b = (NeuronInfo) super.clone();
        } catch (CloneNotSupportedException ex) {
            throw new Error(ex);
        }
        return b;
    }
}

public class Synapse implements Cloneable{
    public int     in       = -1;
    public int     out      = -1;
    public double  w        = 0;
    public Synapse(){};

    @Override
    public Object clone(){
        Synapse cloned = null;
        try{
            cloned = (Synapse) super.clone();
        }catch(CloneNotSupportedException e){
            throw new Error(e);
        }
        return cloned;
    }

    public String toString(){
        return in + "["+w+"]"+out;
    }
}
```

8.5 Pagalbinės klasės Java kalba

```
public class F {

   public static <T> void merge(List<T> l1, List<T> l2, List<T> addTo, Comparator<T> cmp){
      Iterator<T> i1 = l1.iterator();
      Iterator<T> i2 = l2.iterator();
      Integer c = null;
      T o1 = null;
      T o2 = null;
      while(i1.hasNext() || i2.hasNext()){

         if(!i1.hasNext()){
            addTo.add(i2.next());
         }
         else if(!i2.hasNext()){
            addTo.add(i1.next());
         }
         else{
            if(c == null){
               o1 = i1.next();
               o2 = i2.next();
            }else{
               if(c > 0){//added o2
                  o2 = i2.next();
               }else{
                  o1 = i1.next();
               }
            }
            c = cmp.compare(o1, o2);
            if(c > 0){
               addTo.add(o2);
            }
            else{
               addTo.add(o1);
            }
         }
      }

   }
```

```
    public static <T> LinkedList<T> pickRandomPreferLow(Collection<T> col, int amount, int startingAmount, int
amountDecay){

        int limit = Math.min(amount, col.size());
        LinkedList<Integer> indexArray = new LinkedList<>();
        for(int i = 0; i < col.size(); i++){
            for(int indexAm = Math.max(1,startingAmount); indexAm>0; indexAm-- ){
                indexArray.add(i);
            }
            startingAmount-=amountDecay;

        }
        ArrayList<T> array = new ArrayList<>(col);
        LinkedList<T> result = new LinkedList<>();
        seededShuffle(indexArray,F.RND);
        for(int i = 0; i < limit; i++){
            result.add(array.get(indexArray.removeFirst()));
        }
        return result;

    }
    public static <T> LinkedList<T> pickRandom(Collection<T> col, int amount){

        int limit = Math.min(amount, col.size());
        LinkedList<Integer> indexArray = new LinkedList<>();
        for(int i = 0; i < col.size(); i++){
            indexArray.add(i);
        }
        ArrayList<T> array = new ArrayList<>(col);
        LinkedList<T> result = new LinkedList<>();
        seededShuffle(indexArray,F.RND);
        for(int i = 0; i < limit; i++){
            result.add(array.get(indexArray.removeFirst()));
        }
        return result;

    }
    public static <T> T pickRandom(Collection<T> col){
        return pickRandom(col,1).getFirst();
    }
    public static <T> T removeRandom(Collection<T> col){
        T pickRandom = pickRandom(col);
        col.remove(pickRandom);
        return pickRandom;
    }
```

```
public static Random RND = new SecureRandom(0);

public static void seededShuffle(List list, Random rnd){
   Integer size = list.size();
   LinkedList<Integer> indexArray = new LinkedList<>();
   for(int i = 0; i < size; i++){
      indexArray.add(i);
   }
   ArrayList newList = new ArrayList(size);
   for(int i = 0; i < size; i++){
      int nextIndex = rnd.nextInt(size-i);
      Integer remove = indexArray.remove((int)nextIndex);
      newList.add(list.get(remove));
   }
   list.clear();
   list.addAll(newList);

}

public static double sigmoid(final double x) {
   return 2.0 / (1.0 + Math.exp(-4.9 * x)) - 1.0;
}
public static int randomSign(){
   int sign = -1;
   if(RND.nextBoolean()){
      sign = 1;
   }
   return sign;
}

public static int StringNumCompare(String s1, String s2){
   int len1 = s1.length();
   int len2 = s2.length();

   if(len1 == len2){
      return s1.compareTo(s2);
   }
   return len1 - len2;
}
```

```
    public static Map<Integer, ActivationFunction> getDefaultActivationMap(){
        HashMap<Integer, ActivationFunction> map = new HashMap<>();
        int i = 0;
        map.put(i++, F::sigmoid);
        map.put(i++, Math::sin);
        map.put(i++, x -> Math.exp(-x*x / 2) / Math.sqrt(2 * Math.PI));//gaussian
        map.put(i++, x -> Math.abs(x));
        map.put(i++, x-> x%1);
        return map;
    }

}


public class Pos {
    public static class MinMax {
        public Number min,max;
        public MinMax(Number min, Number max){
            this.min = min;
            this.max = max;
        }
    }
    private Double[] vector;
    public Pos(Number...coordinates){
        vector = new Double[coordinates.length];
        for(int i = 0; i<coordinates.length; i++){
            vector[i] = coordinates[i].doubleValue();
        }
    }
    public Double[] normalized(MinMax[] minmax){
        Double[] res = new Double[vector.length];
        for(int i = 0; i < vector.length; i++){
            Double min = minmax[i].min.doubleValue();
            Double max = minmax[i].max.doubleValue();
            res[i] = (vector[i] - min)/(max - min);
        }
        return res;
    }

    public Double[] normalized(MinMax[] minmax,Number rangeStart, Number rangeEnd){
        Double[] res = this.normalized(minmax);
        for(int i = 0; i< res.length; i++){
            res[i] = res[i] * (rangeEnd.doubleValue() - rangeStart.doubleValue()) + rangeStart.doubleValue();
        }
        return res;
```

```
    }

    public Double[] get(){
        return vector;
    }
    public Integer dim(){
        return vector.length;
    }


    public String toString(){
        return Arrays.asList(vector).toString();
    }

    public Double euclidianDistance(Pos to){
        Double sum = 0d;

        int len = Math.min(to.vector.length, this.vector.length);
        for(int i = 0; i < len; i++){
            sum+= Math.sqrt(Math.pow(this.vector[i], 2) - Math.pow(to.vector[i], 2));
        }
        Pos higherDim = null;
        if(len < to.vector.length){
            higherDim = to;
        }
        else if(len < this.vector.length){
            higherDim = this;
        }
        if(higherDim!=null){
            for(int i = len; i < higherDim.vector.length; i++){
                sum+= Math.abs(higherDim.vector[i]);
            }
        }
        return sum;
    }


}
```

Literatūra

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

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

[KW16] Kearney, William T., Using Genetic Algorithms to Evolve Artifcial Neural Networks (2016).Honors Theses. Paper 818. htp://digitalcommons.colby.edu/honorstheses/818

[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

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