

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1779

**Primjena neuroevolucijskih
algoritama promjenjivih topologija
u igranju računalnih igara**

Tvrtko Zadro

Zagreb, veljača 2019.

Zagreb, 4. listopada 2018.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1779

Pristupnik: **Tvrtko Zadro (0036478569)**

Studij: Računarstvo

Profil: Računarska znanost

Zadatak: **Primjena neuroevolucijskih algoritama promjenjivih topologija u igranju računalnih igara**

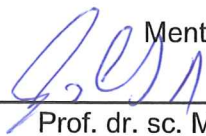
Opis zadatka:

Proučiti i opisati primjenu evolucijskih algoritama za optimiranje neuronskih mreža promjenjivih topologija. Primijeniti neuronske mreže za stvaranje strategije igranja neke jednostavne računalne igre. Načiniti programski sustav koji koristi evolucijske algoritme za optimiranje parametara i topologije neuronske mreže pogodne za igranje jednostavne računalne igre koristeći raspoloživo slobodno dostupno programsko okruženje u programskom jeziku Python. Razmotriti proširenje programskog sustava grafičkim sučeljem za vizualizaciju evolucije algoritma, odnosno za vizualizaciju jedinki u populaciji koje predstavljaju neuronske mreže. Uz rad priložiti izvorne tekstove programa i citirati korištenu literaturu.

Zadatak uručen pristupniku: 12. listopada 2018.

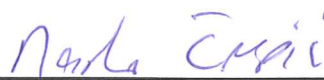
Rok za predaju rada: 8. veljače 2019.

Mentor:



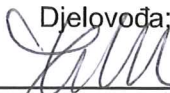
Prof. dr. sc. Marin Golub

Predsjednik odbora za
diplomski rad profila:



Doc. dr. sc. Marko Čupić

Djelovoda:



Izv. prof. dr. sc. Tomislav Hrkać

SADRŽAJ

1. Uvod	1
1.1. Slični radovi	2
1.2. Uloga videoigara u razvoju umjetne inteligencije	2
2. Neuroevolucija promjenjivih topologija	5
2.1. Zapis gena	5
2.2. Praćenje gena pomoću povijesnih oznaka	6
2.3. Očuvanje inovacija kroz specijaciju	7
2.4. Minimizacija dimenzionalnosti kroz inkrementalni rast iz minimalne strukture	9
3. Programsko ostvarenje	10
3.1. Korištene biblioteke	10
3.2. Struktura projekta	11
3.2.1. Klasa <i>Config</i>	12
3.2.2. Klasa <i>NEAT</i>	14
3.2.3. Klasa <i>Population</i>	15
3.2.4. Klasa <i>Species</i>	19
3.2.5. Klasa <i>Individual</i>	21
3.2.6. Klasa <i>Phenotype</i>	26
3.2.7. Klasa <i>Neuron</i>	27
3.2.8. Ostale metode	27
4. Opis problema	29
4.1. Problem konstrukcije XOR vrata	29
4.2. Radno okruženje <i>HalfCheetah</i>	30
4.3. Radno okruženje <i>LunarLander</i>	31

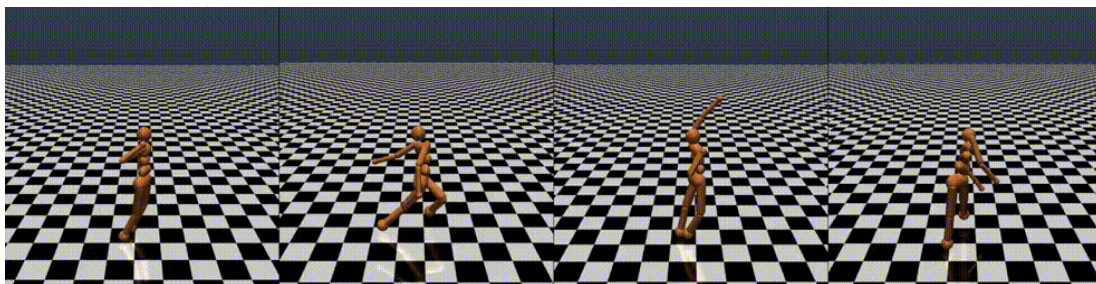
5. Ispitivanje	33
5.1. Problem konstrukcije XOR vrata	33
5.1.1. Pristranost u čvorovima	34
5.2. Optimiranje strategije hodanja u simulaciji <i>HalfCheetah</i>	35
5.2.1. <i>Turnirska selekcija</i>	36
5.3. Optimiranje strategije igranja igre <i>LunarLander</i>	37
5.3.1. Problem nasumičnog pokretanja	38
5.3.2. Ostali rezultati	39
5.3.3. Izlazak iz lokalnog optimuma	43
6. Zaključak	44
Literatura	46

1. Uvod

Proteklih nekoliko godina u svijetu umjetne inteligencije (eng. *Artificial Intelligence, AI*) vlada trend oživljavanja starih ideja. Strahoviti napredak tehnologije pokazao je da, ako postoji dovoljna komputacijska moć, davno predloženi koncepti mogu pokazati odlične rezultate u praksi. Tako su 2012. godine autori *AlexNet* mreže [1] pokazali kako dizajnirati, skalirati i trenirati konvolucijske neuronske mreže (eng. *Convolutional Neural Networks, CNNs*), koje su postavile nove standarde u području računalnog vida. Također, 2013. godine Mnih V. i ostali u svome su radu [2] kombinirali stari koncept Q-učenja (*Q-learning*) i CNN-a te s novonastalim dubokim Q-učenjem (eng. *deep Q-learning*) uspješno riješili *Atari* igre i time ponovno popularizirali područje potpornog učenja (eng. *Reinforcement Learning, RL*).

Bliže temi ovog rada, u 2017. godini *OpenAI*¹ izdao je članak [5] u kojem predlažu evolucijske strategije (eng. *Evolution Strategies, ES*) kao skalabilnu alternativu potpornom učenju. Usporedili su evolucijske strategije s trenutno popularnom metodom potpornog učenja, tzv. *Asynchronous Advantage Actor Critic (A3C)* [3]. Koristeći obje metode učili su čovjekolikog igrača da hoda (Slika 1.1). Ispitivanja su provodili u *MuJoCo (Multi-Joint dynamics with Contact)* okruženju za simulaciju kontrole pokreta, na čijoj je jednostavnijoj inačici isproban i algoritam iz ovog rada. Na grozdu od 80 računala i 1,440 CPU jezgri evolucijske strategije su igrača naučile da hoda u 10 minuta, dok je A3C-u na 32 jezgre trebalo otprilike 10 sati. Iako su se do tada evolucijske strategije smatrale lošijima u visoko dimenzijskim problemima, ovime su pokazali da su ipak upotrebljive zato što ih je lako paralelizirati. Algoritam koji dijeli istu karakteristiku s prethodnom metodom također je iz grane evolucijskog računanja (eng. *Evolutionary Computing, EC*) te je predmet ovog rada. Predložili su ga 2002. godine Stanley K.O. i Miikkulainen, R. pod nazivom neuroevolucija promjenjivih topologija (eng. *NeuroEvolution of Augmenting Topologies, NEAT*) [6].

¹*OpenAI* je neprofitna organizacija za umjetnu inteligenciju čiji je vlasnik Elon Musk.



Slika 1.1: Prikaz nekoliko pokušaja čovjekolikog igrača da hoda u *MuJoCo* okruženju

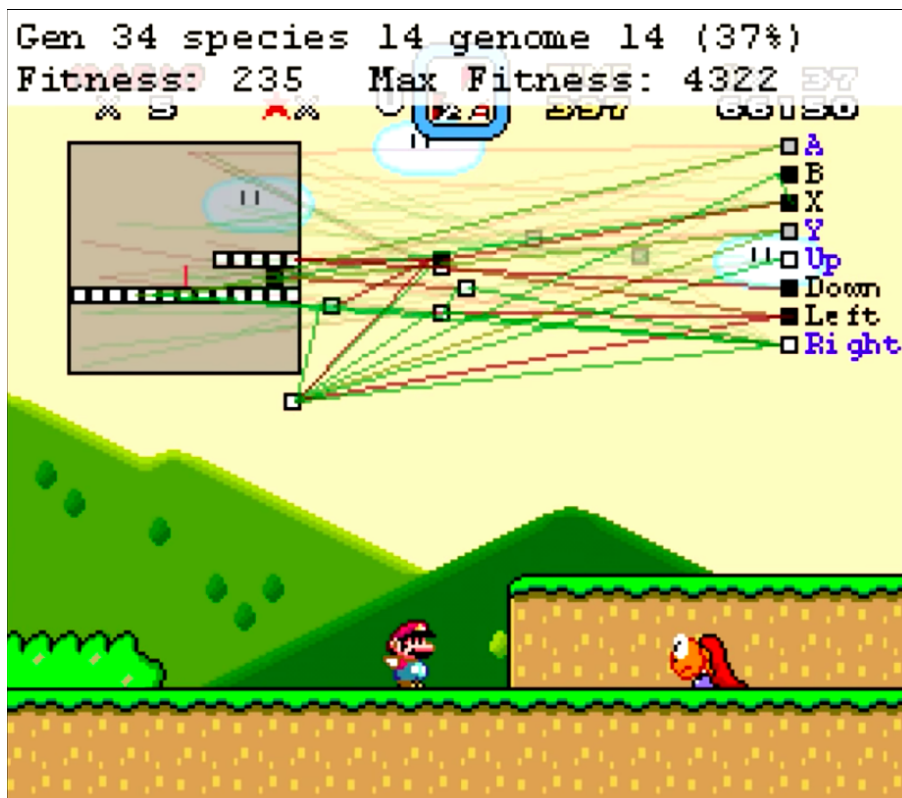
1.1. Slični radovi

Vjerojatno najpoznatiju implementaciju ovog algoritma napravio je autor pod nadimkom *SethBling* nazvanu *MarI/O*. Autor je 2015. godine u *YouTube* videu, koji je do sada prikupio više od 7,5 milijuna pregleda, predstavio agenta koji je naučio igrati videoigru *Super Mario World* uz pomoć algoritma NEAT. Na primjeru jednog pokušaja prelaska nivoa (Slika 1.2) mogu se vidjeti podaci o izvođenju algoritma (na vrhu), vizualizacija agentovog "mozga" (ispod tekstualnih podataka) te stanje igre (pozadina). Agent iz videa nakon 24 sata treniranja uspješno je prešao jedan nivo videoigre.

Nadalje, u radu [9] iz 2013. godine autori Zhen, J.S. i Watson, I. na još jednoj igri prikazuju uporabu algoritma NEAT i njegove varijante u stvarnom vremenu (eng. *real-time NeuroEvolution of Augmenting Topologies, rtNEAT*). Igra je bila poseban izazov jer su morali upravljati s više agenata u isto vrijeme, s obzirom da se radi o strategiji u stvarnom vremenu (eng. *Real Time Strategy, RTS*). Spomenuta igra je popularna računalna igra *StarCraft: Brood War*. Oba algoritma su koristeći mikromenadžment (eng. *micromanagement*) upravljali jedinicama u igri te su autori pokazali da ove dvije metode postižu bolje rezultate od skriptirane umjetne inteligencije ugrađene u igru. Bitno je primijetiti da je većina dosad spomenutih algoritama ostvarena i ispitana na videoigrama. Računalne igre i umjetna inteligencija dva su pojma koja se sve češće spominju zajedno.

1.2. Uloga videoigara u razvoju umjetne inteligencije

Brzorastuća industrija videoigara stvara sve složenije aproksimacije prirodnih okruženja. Realne simulacije postavljaju odličan okvir unutar kojeg se na djelotvoran i siguran način može razvijati umjetna inteligencija. Nove tehnike i metode mogu se ispitati i ocijeniti u kontroliranim uvjetima prije nego što ih se primijeni na složenije probleme u stvarnom svijetu. Novije računalne igre stvaraju sve veći prostor stanja i



Slika 1.2: Prikaz MarI/O agenta kako igra *Super Mario World*

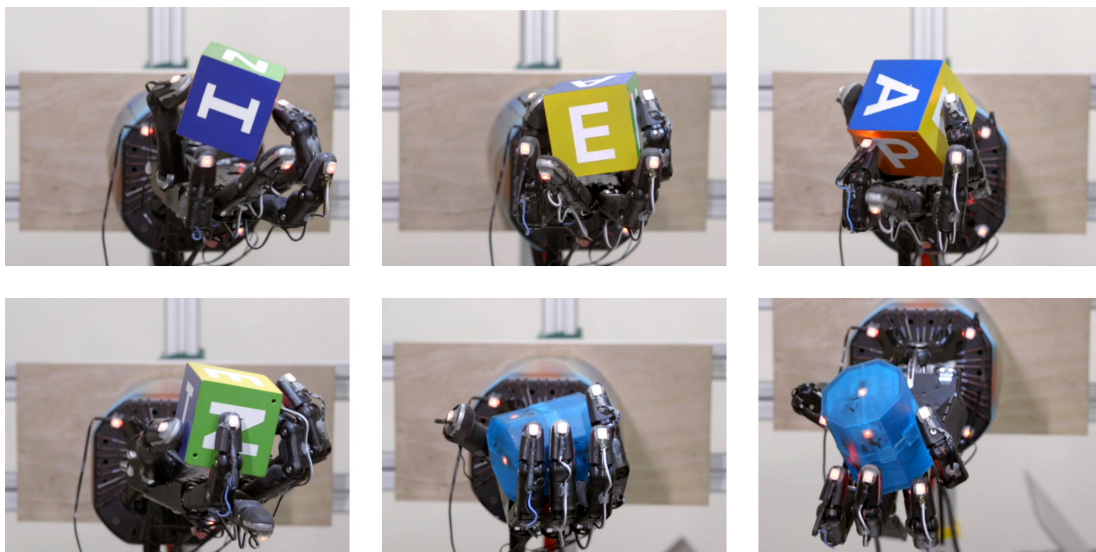
akcija, zbog čega predstavljaju veliki izazov u području razvoja umjetne inteligencije. Zbog postepenog rasta složenosti videoigara, odličan su izbor za razvoj umjetne inteligencije zato što je odabirom starijih ili novijih igara moguće lako upravljati težinom problema kojeg se pokušava riješiti. Dosad spomenute videoigre su bile relativno jednostavne, ali neke svjetski najpoznatije istraživačke skupine ispituju svoje algoritme na poznatim novim igrama s velikom kompetitivnom scenom.

Najpoznatiji takav projekt je *OpenAI Five* [4]. Njega kontinuirano razvija već spomenuti istraživački tim *OpenAI*. Radi se o sustavu koji je demonstrirao svoj kapacitet na poznatoj računalnoj igri *Dota 2*. To je jedna od najkompetitivnijih igara s velikim brojem igrača i *esport*² turnira. *OpenAI Five* je u osmom mjesecu 2017. godine postigao veliki uspjeh i pobijedio nekoliko najboljih *Dota 2 esport* igrača. Važno je napomenuti da je svrha toga bila razviti umjetnu inteligenciju šire primjene, tako da su istim algoritmom na kraju riješili i prethodno neriješen problem u robotici - kontrolu robotske ruke (Slika 1.3).

Osim toga, *DeepMind*³ nakon svog poznatog projekta *AlphaGo* najavio je da će ko-

²*Esport* je oblik natjecanja koji koristi videoigre. Procijenjeno je da će do 2019. godine 427 milijuna ljudi širom svijeta gledati neki oblik *esporta*.

³*DeepMind* je *Googleova* tvrtka za razvoj umjetne inteligencije.



Slika 1.3: Slike robotske ruke kojom upravlja *OpenAI Five*

ristiti RTS igru *StarCraft II* kao okruženje za razvoj umjetne inteligencije. Napravili su ekstenzivan okvir za igru u suradnji s vlasnikom igre, tvrtkom *Blizzard Entertainment*. Ova igra također je jedna od poznatijih na *esport* sceni. *DeepMind* je kratko nakon objave okruženja objavio i rad [7] u kojem prezentiraju izazov koji *StarCraft II* predstavlja za područje potpornog učenja. Početkom 2019. godine u video prijenosu prikazali su rezultate koje su postigli te je njihov agent pobijedio dva poznata profesionalna igrača te igre. Ova dva rada pokazala su kako zajednica prepoznaje videoigre kao odličan okvir za razvoj umjetne inteligencije.

U ostatku rada dan je pregled karakteristika i implementacije spomenute vrste neuroevolucijskog algoritma (eng. *NeuroEvolution, NE*), kratak opis problema koje se pokušava riješiti u radu te analiza njegove djelotvornosti u treniranju agenata za navedene probleme.

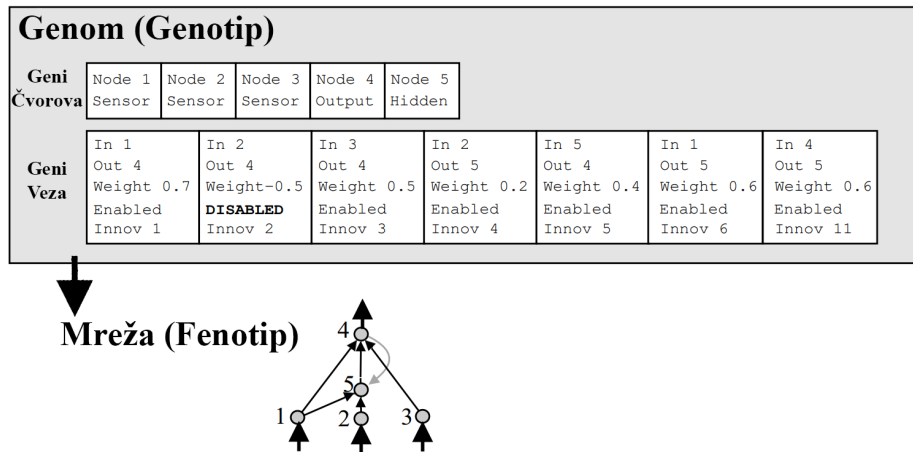
2. Neuroevolucija promjenjivih topologija

Neuroevolucija promjenjivih topologija (eng. *NeuroEvolution of Augmenting Topologies, NEAT*) je genetski algoritam (eng. *Genetic algorithm, GA*) za generaciju umjetnih neuronskih mreža (eng. *Artificial Neural Network, ANN*). Algoritam mijenja i težine i strukture mreža, pokušavajući tako naći ravnotežu između dobrote razvijenih rješenja i njihove raznolikosti. Bazira se na primjeni tri ključne tehnike: praćenje gena povijesnim oznakama (eng. *history markers*) kako bi se omogućilo križanje topologija, izvršavanje specijacije kako bi se očuvale inovacije te inkrementalan razvoj topologija mreže kako bi se poticala što jednostavnija rješenja.

2.1. Zapis gena

Postupak genetskog enkodiranja napravljen je tako da se odgovarajući geni lako usklade prilikom križanja. Genomi su linearna reprezentacija povezanosti unutar mreže (Slika 2.1). Svaki genom sadrži listu gena veza (eng. *connection genes*) od kojih svaki referencira dva gena čvora (eng. *node gene*) koji su povezani. Genom sadrži listu ulaznih, skrivenih i izlaznih čvorova koji se mogu spojiti. Svaki gen veze navodi ulazni čvor, izlazni čvor, težinu veze, indikator je li veza uključena te broj inovacije. Inovacijski broj omogućava lako prepoznavanje odgovarajućih gena, a indikator uključenosti veze definira je li veza aktivna ili nije. Ako nije, jedinka će i dalje nositi informacije o toj vezi, ali se ona neće očitovati u fenotipu.

Mutacije u algoritmu NEAT mogu promijeniti i težine veza i strukturu mreže. Mutacija težina odvija se kao i kod bilo kojeg neuroevolucijskog sustava, a strukturalna mutacija odvija se na dva načina (Slika 2.2). Svaka takva mutacija povećava veličinu genoma tako da dodaje gene. U mutaciji dodavanja veze (eng. *add connection mutation*) dodaje se novi gen s nasumično odabranom težinom između dva do tad nepovezana čvora. Nadalje, u mutaciji dodavanja čvora (eng. *add node mutation*) pos-



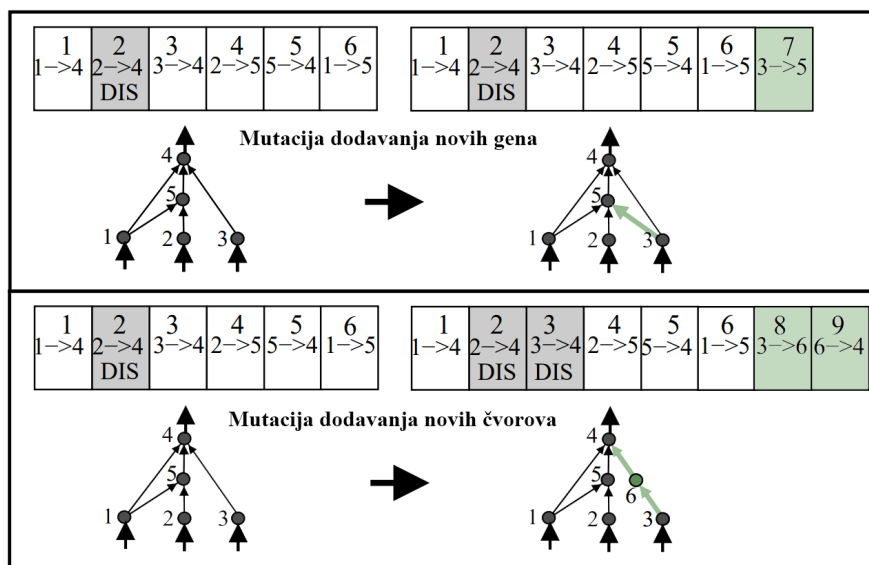
Slika 2.1: Primjer mapiranja genotipa u fenotip [2]

tojeća veza se prepolavlja te se postavlja novi čvor između dva koja su bila povezani tom vezom. Stara veza se isključuje i u genom se dodaju dvije nove veze. Vezi koja ulazi u novonastali čvor težina se postavlja na vrijednost 1, a vezi koja izlazi iz njega dodjeljuje se težina isključene veze. Kroz mutaciju genomi se postepeno povećavaju. To rezultira mrežama koje na istim mjestima imaju drugačije veze što može izazvati problem prilikom križanja. Osim toga, problem nastaje i zbog genoma različitih veličina. U nastavku je objašnjeno kako algoritam NEAT rješava ovaj problem te kako se ti genomi križaju.

2.2. Praćenje gena pomoću povijesnih oznaka

Sve što sustav treba raditi kako bi znao koji geni su sukladni jest pratiti povijesno ishodište svakog gena. Svaki put kada se pojavi novi gen (kroz strukturnu mutaciju), globalni inovacijski broj (eng. *global innovation number*) se povećava i dodjeljuje tom genu. Samim time, inovacijski broj predstavlja kronologiju svih gena u sustavu. Mogući problem je da iste strukturne inovacije dobiju različite inovacijske brojeve ako su se dogodile u istoj generaciji. U skladu s time, čuvanjem liste inovacija koje su se dogodile u trenutnoj generaciji moguće je osigurati da se svakoj identičnoj mutaciji dodijeli isti inovacijski broj.

Prilikom križanja, geni iz oba genoma s istim inovacijskim brojem se usklade. Geni koji se poklapaju (eng. *matching genes*) nasljeđuju se nasumično, dok se geni koji se ne poklapaju u sredini (eng. *disjoint genes*) i geni koji se ne poklapaju na kraju (eng. *excess genes*) nasljeđuju od roditelja s većom dobrotom. Ako im je dobrota jednaka, dijete nasljeđuje gene od roditelja koji ima jednostavniju strukturu. Dodavanjem novih



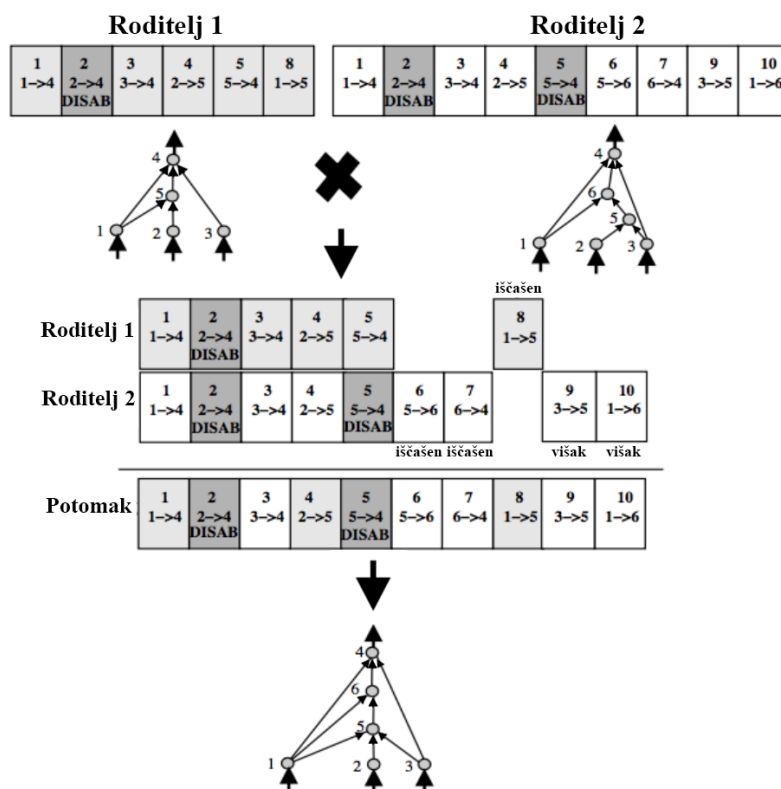
Slika 2.2: Dva tipa strukturalnih mutacija [2]

gena u populaciju i križanjem jedinki sustav formira populaciju raznolikih topologija. Naime, populacija sama po sebi ne može održavati topološke inovacije. Zbog toga što se manje strukture optimiziraju brže od velikih, dodavanje čvorova i veza obično inicijalno smanjuje dobrotu mreže. Stoga, izmijenjene strukture imaju male šanse preživjeti više od jedne generacije, iako inovacije koje uvode mogu biti presudne za rješavanje zadatka. Rješenje je da se inovacije zaštite tako da se populaciju podijeli u vrste, kao što je navedeno u nastavku.

2.3. Očuvanje inovacija kroz specijaciju

Dijeljenjem populacije u vrste u kojima su jedinke sa sličnim topologijama daje jedinkama više vremena da optimiziraju težine. Na ovaj način čuvaju se inovacije koje bi mogle dugoročno pomoći u dostizanju cilja. Način na koji se prepoznaju slične topologije opet je nešto što treba definirati i tu ponovno pomažu povijesne oznake. Broj neusklađenih gena (eng. *disjoint genes*) i viška gena (eng. *excess genes*) između dvije jedinke je mjera njihove sukkladnosti. Što je taj broj veći, dijele manje evolucijske povijesti, to jest manje su kompatibilni. Po izrazu iz rada [6], kompatibilna udaljenost (eng. *compatibility distance*) δ može se izračunati kao funkcija broja viška gena E i neusklađenih gena D te prosječne razlike u težinama usklađenih gena \bar{W} , uključujući i isključene veze:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W} \quad (2.1)$$



Slika 2.3: Usklađivanje genoma različitih topologija mreža koristeći inovacijske brojeve [2]

Koeficijenti c_1 , c_2 i c_3 su parametri kojima se definira važnost ova tri faktora, a broj N je broj gena u većem genomu i uloga mu je da normalizira izraz. Jednom kada se izračuna udaljenost δ , određuje se pripadnost vrsti pomoću praga kompatibilnosti (eng. *compatibility threshold*) δ_t . Ako je udaljenost δ između jedinke i predstavnika vrste manja od praga kompatibilnosti δ_t ta jedinka pripada toj vrsti. Kako bi se tako sortirale jedinke u vrste algoritam cijelo vrijeme treba održavati uređenu listu vrsti. Svaka postojeća vrsta predstavljena je nasumičnim genomom iz prethodne generacije te vrste. Kako se vrste ne bi preklapale, dani genom g u trenutnoj generaciji pridijeljen je vrsti prvog reprezentativnog genoma s kojim je kompatibilan. Ako g nije kompatibilan ni s jednom postojećom vrstom, stvara se nova vrsta kojoj se g postavlja kao reprezentativan uzorak.

Za reprodukcijski mehanizam koristi se eksplicitno dijeljenje dobrote (eng. *explicit fitness sharing*, Goldberg and Richardson, 1987), gdje organizmi u istoj vrsti moraju dijeliti dobrotu. Tako nijedna vrsta ne može postati prevelika, čak i ako su jedinke dobre. Dakle, teško je da će ijedna vrsta preuzeti cijelu populaciju, što je presudno da bi specijacija funkcionirala. Po izrazu iz rada [6], prilagođena dobrota f'_i za organizam i računa se sukladno njegovoj udaljenosti δ od svakog drugog organizma j u populaciji:

$$f'_i = \frac{f_i}{\sum_{j=1}^n sh(\delta(i, j))} \quad (2.2)$$

Funkcija dijeljenja sh je 0 kada je udaljenost $\delta(i, j)$ veća od praga δ_t , inače $sh(\delta(i, j))$ poprima vrijednost 1. Dakle, suma $\sum_{j=1}^n sh(\delta(i, j))$ približno računa broj organizama koji su iste vrste kao i jedinka i . Jednom kada se izračuna prilagođena dobrotu f'_i svih jedinki, uklanjaju se najlošiji organizmi unutar vrsta. Nakon toga, cijela populacija zamjenjuje se potomcima preostalih jedinki u svakoj vrsti. Razlog iz kojeg se provodi specijacija je da bi se očuvale topološke inovacije. Zadnja karakteristika algoritma je da se potraga za rješenjem sustava izvrši što djelotvornije. To se postiže minimiziranjem dimenzionalnosti prostora pretrage.

2.4. Minimizacija dimenzionalnosti kroz inkrementalni rast iz minimalne strukture

Idejna preteča algoritma NEAT, algoritam evolucije topologije i težina umjetne neuronske mreže (eng. *Topology and Weight Evolving Artificial Neural Networks, TWE-ANN*) [8], napunio bi inicijalnu populaciju nasumično generiranim topologijama kako bi potaknuo raznolikost kod završnih rješenja. Algoritam NEAT traži što jednostavnije rješenje i čini pretragu pristranom tako da je inicijalna populacija uniformna te nijedna početna mreža nema skrivenih čvorova. Nova struktura uvodi se postepeno tako što jedinke mutiraju i dobivaju nove čvorove i veze. Novopuširene jedinke preživjet će samo ako imaju visoku dobrotu te se zbog toga može reći da su sve strukturne promjene u algoritmu NEAT opravdane. Budući da je početna struktura populacije jako mala, dimenzionalnost prostora pretrage je smanjena i algoritam NEAT uvijek pretražuje po manje dimenzionalnim domenama od ostalih TWEANN algoritama i neuroevolucijskih sustava s fiksnom topologijom. Zbog toga NEAT ima bolja svojstva nad tim algoritmima, kao što je opisano u radu [6]. U sljedećem poglavlju dan je opis implementacije ostvarene u okviru ovog rada.

3. Programsko ostvarenje

Algoritam koji je ostvaren u okviru ovog rada podijeljen je u nekoliko datoteka i klasa kako bi kod bio pregledniji te lakši za daljnju nadogradnju. Program trenutno podržava izvođenje na nekoliko različitih igara i prikaz bitnih informacija na kraju izvođenja. Svi hiperparametri (eng. *hyperparameters*) odvojeni su u zasebnu datoteku kako bi se olakšalo njihovo podešavanje. Implementacija je napisana u programskom jeziku *Python*. *Python* je trenutno najpopularniji jezik za ostvarenje algoritama strojnog učenja zato što je fleksibilan i pristupačan. Iako je njegova mana to što je sporiji u izvođenju od nekih drugih programskih jezika, postoji veliki broj programskih biblioteka za *Python* čija je implementacija u programskom jeziku *C*, zbog čega je razlika u brzini izvođenja mnogo manja. Prednost je ta što generalno već postoji veliki broj programskih biblioteka koje mogu služiti kao okruženje za implementaciju.

3.1. Korištene biblioteke

Najvažnije programske biblioteke koje se koriste u projektu su tzv. *OpenAI Gym* i *PyGame Learning Environment (PLE)*. Ove dvije programske biblioteke su okruženja za ispitivanje algoritama potpornog učenja. Podržavaju upravljanje velikim brojem arkadnih i drugih videoigara uz pomoć jednostavnog programskog sučelja (eng. *Application Programming Interface, API*). Oba okruženja koriste se tako da se stvori inačica igre na kojoj se pokušava isprobati algoritam. Preko ove inačice pristupa se zapažanjima o trenutnom stanju igre koja su zadana u opisu okruženja. Ova zapažanja služe kao ulaz u neuronsku mrežu koja donosi odluke o sljedećem koraku. Nakon dohvaćanja zapažanja inačici se šalje odluka agenta. Svako okruženje ima definirane akcije koje agent može napraviti u svakom trenutku. Dakle, agent je crna kutija (eng. *black box*) i odabir bilo koje igre iz jednog od okruženja definira broj ulaza i izlaza iz te kutije, tj. neuronske mreže koja je opisana genomom. Nakon što se instanci igre dojadi akcija, okvir se brine da se promjene prikažu na ekranu te vraća nova zapažanja i nagradu koju je agent dobio za tu akciju. Akumuliranjem nagrada za svaki korak na

OpenAI gym environment
make(env_name) : environment
reset() : observation
render() : void
step(action) : observation, reward, done

Slika 3.1: Opis sučelja klase koja predstavlja okruženje u okviru paketa *gym*

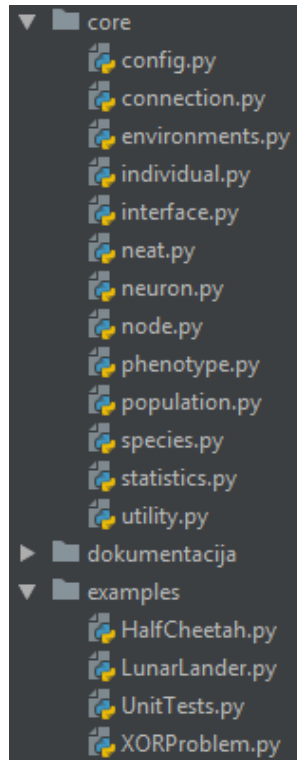
kraju se dobije vrijednost koja definira dobrotu jedinke, to jest agenta.

Na slici 3.1 opisano je sučelje klase koja predstavlja okruženje u okviru paketa *OpenAI gym*. Metoda *make* koristi se za instanciranje željenog okruženja. Metodi se predaje naziv okruženja koji se pokušava instancirati. Kao povratak iz funkcije dobiva se objekt okruženja na kojem se izvode ostale metode. Funkcija *reset* pokreće novu igru u okviru okruženja kojem pripada objekt. Povratak iz funkcije je početno zapažanje nove igre. Kako bi se igra nastavila igrati, potrebno je pozvati metodu *step* sa željenom akcijom. Ova metoda potom vraća i zapažanje novog stanja i nagradu koju je agent dobio za tu akciju. Osim toga, funkcija vraća zastavicu koja označava kraj trenutne igre. Kada agent završi igru ili izgubi pravo za igranjem, zastavica će biti postavljena. Na pozivatelju metode je da pazi da ne poziva *step* nakon što je ta zastavica pozitivna. Zadnja metoda, *render*, ne utječe na tijek izvođenja igre već samo prikazuje trenutno stanje igre na ekranu.

Programske biblioteke za vizualizaciju grafova i fenotipa su *Matplotlib* i *NetworkX*. *Matplotlib* je najpoznatija programska biblioteka za iscrtavanje grafova u *Pythonu* te olakšava vizualizaciju podataka na kraju izvođenja algoritma. *NetworkX* omogućava prikaz usmjerenih grafova i koristi se za vizualizaciju fenotipa, tj. neuronskih mreža. Programska biblioteka koju je također bitno napomenuti je *NumPy*. *NumPy* je moćan okvir za računanje u *Pythonu* koji je implementiran u programskom jeziku *C* te omogućava brzo i jednostavno baratanje visokodimenzionalnim nizovima. S ovime je zaključen opis vanjskih biblioteka koje su korištene u programskom ostvarenju. U nastavku poglavlja prikazana je struktura projekta koji je ostvaren u okviru ovog rada te su objašnjene neke ključne točke ove implementacije.

3.2. Struktura projekta

Kao što je navedeno, projekt se sastoji od nekoliko datoteka čija je struktura prikazana na slici 3.2. Sa slike je moguće razaznati dva glavna paketa. Prvi je *core* u kojem

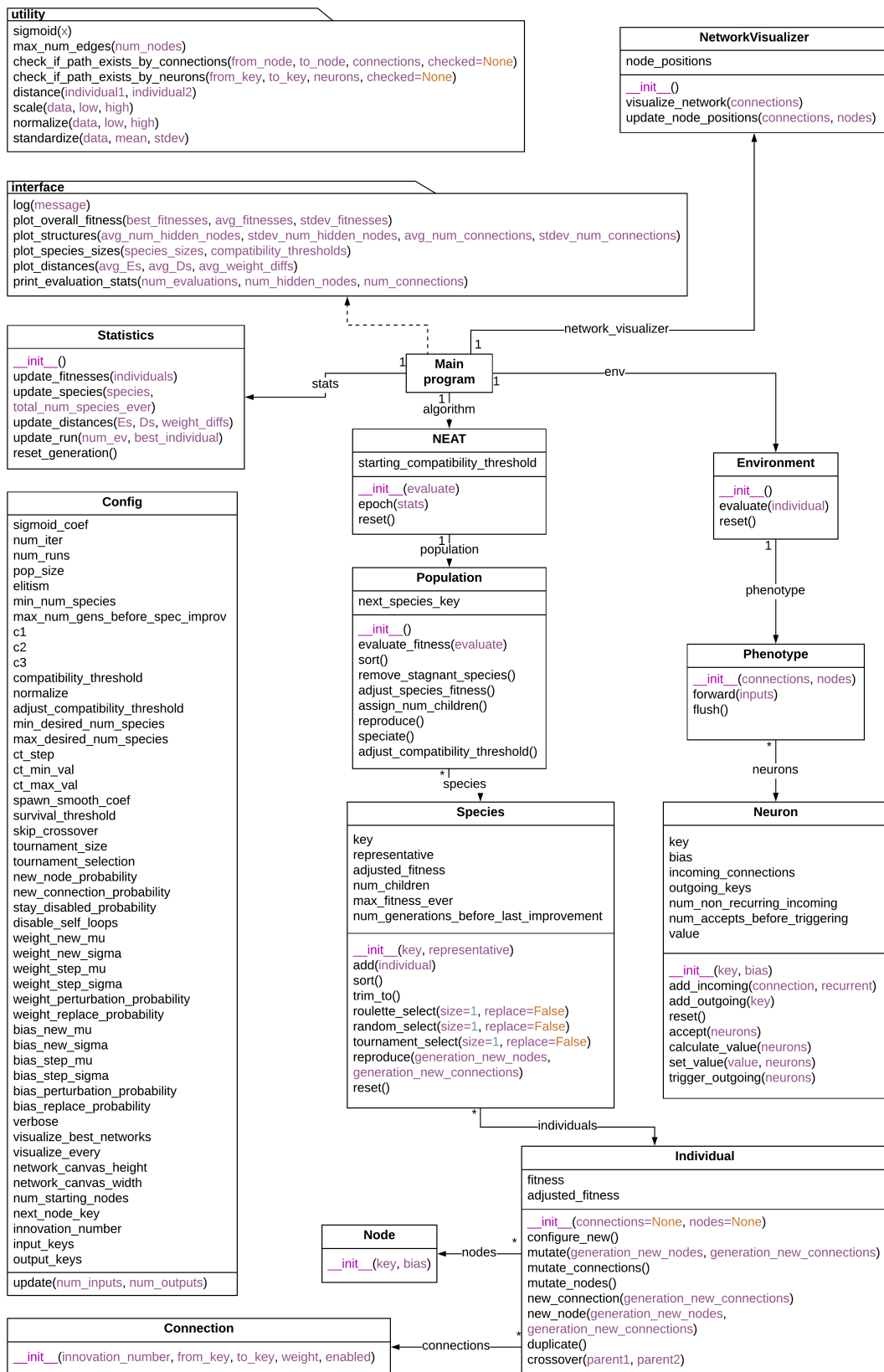


Slika 3.2: Struktura projekta u programskom okruženju *IntelliJ IDEA*

se nalazi implementacija algoritma koja nije direktno vezana za okruženje u kojem se koristi, a drugi je *examples* u kojemu su kratki programi koji pokreću algoritam na različitim problemima koji su podržani. Svaka od datoteka u paketu (Slika 3.2) sadrži definiciju jedne klase ili apstraktne klase i njenih implementacija. Slika 3.3 vizualni je prikaz strukture klasa u tzv. *Unified Modeling language* (UML) obliku. *Main program* s dijagrama (Slika 3.3) predstavlja program koji koristi paket *core*. Taj program brine se da stvori instance svih potrebnih klasa i da upravlja izvođenjem algoritma. U ostatku poglavlja prikazani su isječci koda iz paketa *core* bitni za rad programa.

3.2.1. Klasa *Config*

Klasa koja je na dijagramu (Slika 3.3) prikazana kao *Config* sadrži u sebi samo varijable koje su postavljene odmah prilikom instanciranja. Ove varijable upravljaju radom programa i većina ovih vrijednosti su hiperparametri algoritma. Ovako se na jednom mjestu nalaze sve vrijednosti kojima se može upravljati. Iz glavnog programa je onda moguće pisati preko (eng. *overwrite*) postojećih vrijednosti ako se želi mijenjati detalje izvođenja. Instanca se stvara prilikom pokretanja programa i budući da sadrži upute za rad svih aspekata algoritma, tom objektu imaju pristup sve klase unutar paketa.



Slika 3.3: UML dijagram paketa *core*

3.2.2. Klasa *NEAT*

Klasa *NEAT* najapstraktnija je razina izvedbe algoritma. Prilikom instanciranja automatski stvara populaciju veličine definirane u *config* objektu. U kodu 3.1 se vidi tok svake generacije. Prvi korak u svakoj generaciji je evaluiranje dobrote svake jedinke. Ovaj dio algoritma problematičan je zato što se ne može izbjeći evaluaciju svake jedinke u populaciji. Evaluacija u ovakvim problemima je jako skup proces. Opis metode za evaluaciju jedinke nalazi se u *Environment* klasi i na njega utječe način izvedbe okruženja, tj. videoigre koja se igra. Ono što bi pomoglo izvođenju algoritma je da se evaluacija paralelizira. Budući da je evaluacija svake jedinke neovisna, moguće je optimalno iskoristiti raspoložive resurse tako da se jedinke evaluiraju na odvojenim dretvama. Jedino što je potrebno na kraju je prikupiti izlazne vrijednosti metode, dobrote, i pridijeliti ih jedinkama. Ova implementacija ne podržava paralelizaciju zbog nepristupačnosti okvira u kojima su se igre ispitivale. S obzirom na to da je program na računalu s četiri CPU jezgre koristio samo 20% računalnih resursa, paraleliziranje algoritma uvelike bi ubrzalo izvođenje.

```
1 def epoch(self, stats):
2     # evaluate population and track stats
3     best_individual = self.population.evaluate_fitness(self.evaluate)
4     stats.update_fitnesses(self.population.species)
5     stats.update_structures(self.population.species)
6     log('\tBest fitness: {:.2f}, Average fitness: {:.2f}'.format(stats.best_fitnesses[-1], stats.avg_fitnesses
7     [-1]))
8     log('\tAverage num hidden nodes: {:.2f}, Average num connections: {:.2f}'.format(stats.avg_num_hidden_nodes
9     [-1], stats.avg_num_connections[-1]))
10
11 # sort individuals inside each species and then sort species by their max fitness
12 self.population.sort()
13 log('\tBest individual is in species {:d}'.format(self.population.species[0].key))
14
15 # remove species that showed no progress in recent generations
16 self.population.remove_stagnant_species()
17
18 # normalize adjusted fitness range and assign adjusted fitness to each species
19 self.population.adjust_species_fitness()
20
21 # assign children according to adjusted fitness and remove species with no children
22 self.population.assign_num_children()
23
24 # copy elites if enabled, remove worst in each species and breed new generation
25 children = self.population.reproduce()
26
27 # speciate new individuals and remove empty species
28 Es, Ds, weight_diffs = self.population.speciate(children)
29 stats.update_species(self.population.species, self.population.next_species_key)
30 stats.update_distances(Es, Ds, weight_diffs)
31 log('\tNum species is {:d}'.format(len(self.population.species)))
32 log('\tAverage distance is {:.2f}'.format(sum(weight_diffs) / len(weight_diffs)))
33
34 # change compatibility threshold if number of species becomes too big or too small
35 if config.adjust_compatibility_threshold:
36     self.population.adjust_compatibility_threshold()
37     log('\tCompatibility threshold is {:.2f}'.format(config.compatibility_threshold))
38
39 return best_individual
```

Izvorni tekst programa 3.1: Izvorni tekst metode *epoch* u klasi *NEAT*

Nadalje, prije ostatka epohe potrebno je sortirati postojeće vrste i jedinke u njima. Sortiranje se izvodi tako da se unutar svake vrste jedinke sortiraju po svojoj dobroti, a onda se vrste sortiraju po njihovoj najvećoj dobroti. Sortiranje olakšava ostatak epohe i iako je inače zahtjevan proces, u usporedbi s evaluacijom zanemariv je zato što kod ovakvih problema postupak evaluacije jedinki uzima puno više vremena, a broj jedinki nije toliko velik da bi sortiranje znatno usporilo rad algoritma. Nakon sortiranja iz populacije se brišu vrste koje dugo nisu vidjele napredak te se onda računaju prilagođene dobrote vrsti i broj jedinki koji će imati u sljedećoj generaciji. Nakon što se pridodjeli broj jedinki koji svaka vrsta može imati, slijedi reprodukcija te specijacija novih jedinki. Ove metode detaljnije su objašnjene u sljedećem potpoglavlju.

3.2.3. Klasa *Population*

Klasa koja predstavlja populaciju, osim klase vrste i jedinke, sadrži logiku algoritma. Klasa *NEAT* je na višoj razini apstrakcije kako bi se kasnije lakše dodavale varijacije algoritma. Osim metoda za evaluaciju populacije i sortiranje, sljedeća metoda koja je bitna za izvođenje algoritma je metoda koja iz populacije briše vrste koje dugo nisu pokazale napredak. Ova metoda opisana je u isječku koda 3.2. Ukratko, za svaku vrstu se gleda je li njena najbolja dobrota ove generacije bolja od njene dosadašnje najbolje dobrote. Ako je, brojač se resetira. Ako nije, gleda se koliko je generacije prošlo od njenog zadnjeg napretka te ako je prošlo više od broja generacija definiranog u konfiguracijskoj datoteci, ta vrsta se briše. Također, u konfiguracijskoj klasi definirano je i koliki je najmanji broj vrsta koji može ostati nakon brisanja stagnirajućih vrsta. Ako u bilo kojem trenutku preostali broj vrsta postane jednak definiranom, petlja se prekida. Zbog toga što su vrste sortirane silazno prolazak po vrstama zato se izvodi unazad. Tako uvijek se prvo brišu lošije vrste koje stagniraju. Ako se petlja zaustavi ranije zbog toga što je ostao minimalni dopušteni broj vrsta zbog unazadnog prolaska ostat će bolje vrste. Vrijedno je napomenuti da je u originalnoj implementaciji iz rada [6] također postojala opcija stagnacije na razini populacije, a ne samo vrsti. U tom slučaju omogućilo bi se razmnožavanje samo dvije najbolje vrste. Ovo je bilo u jednoj od iteracija implementacije, ali je maknuto jer bi takav pristup samo još više usporio algoritam u situacijama kada bi se nasumično postigao visoki rezultat za koji bi populaciji trebalo previše vremena da ga sustigne.

```
1 # assumes species are sorted
2 def remove_stagnant_species(self):
3     # iterate through all species from worst to best
4     for spec in reversed(self.species):
5         # stop removing stagnant species if number of species is less than or equal to minimum number of species
6         if len(self.species) <= config.min_num_species:
```

```

7         break
8
9         # if species improved reset counter, else increment it
10        best_fitness = spec.individuals[0].fitness
11        if best_fitness > spec.max_fitness_ever:
12            spec.num_gens_before_last_improv = 0
13            spec.max_fitness_ever = best_fitness
14        else:
15            spec.num_gens_before_last_improv += 1
16
17        # if species hasn't improved in a long time remove it from the population
18        if spec.num_gens_before_last_improv > config.max_num_gens_before_spec_improv:
19            log('\t\tRemoving stagnant species {:d} after {:d} generations without improvement'.format(spec.
20            key, spec.num_gens_before_last_improv))
                self.species.remove(spec)

```

Izvorni tekst programa 3.2: Izvorni tekst metode *remove_stagnant_species* u klasi *Population*

Nakon eliminacije stagnirajućih vrsta slijedi prvi korak pripreme populacije za re-produkciju. Kako bi se odredilo koliko će se potomaka pridijeliti svakoj vrsti prvo se mora izračunati prilagođena dobrota svake vrste. U originalnom radu [6] ovo je izvedeno tako da se računala prilagođena dobrota za svaku jedinu po jednakosti 2.2 te se onda prilagođena dobrota za vrstu gledala kao zbroj prilagođenih dobrota njenih jedinki. S obzirom da bi za računanje prilagođene dobrote jedinke trebalo gledati njenu udaljenost od svake jedinke u populaciji, postupak je pojednostavljen. Tako se u tom slučaju dobrota ne dijeli s brojem jedinki koji su unutar praga udaljenosti od iste, nego se dijeli s brojem jedinki unutar vrste kojoj jedinka pripada. Sukladno tome, zbroj prilagođenih dobrota svih jedinki unutar vrste zapravo bi dalo prosječnu dobrotu jedinki vrste. Zbog toga u metodi odmah računa prilagođena dobrota vrste, jer se ionako koristi samo u tom kontekstu. Također, zbog načina na koji se koristi ona u originalnom radu nikada nije smjela biti negativna. Kako bi se to ograničenje izbjeglo u implementaciju je dodana normalizacija. Stoga, u metodi se prvo računa najmanja i najveća dobrota u generaciji, kako bi se kasnije koristeći taj raspon mogla normalizirati prosječna dobrota vrste. Novodobivena vrijednost sprema se u objekt vrste kao prilagođena dobrota.

```

1 def adjust_species_fitness(self):
2     # find minimum and maximum fitness for normalization
3     min_fitness = math.inf
4     max_fitness = -math.inf
5     for spec in self.species:
6         for individual in spec.individuals:
7             if individual.fitness < min_fitness:
8                 min_fitness = individual.fitness
9
10            if individual.fitness > max_fitness:
11                max_fitness = individual.fitness
12
13        # define fitness range
14        fitness_range = max(abs(max_fitness - min_fitness), 1.0)
15
16        for spec in self.species:
17            # calculate species mean fitness
18            mean_fitness = sum([individual.fitness for individual in spec.individuals]) / len(spec.individuals)
19            # set adjusted fitness as normalized mean fitness

```

Izvorni tekst programa 3.3: Izvorni tekst metode *adjust_species_fitness* u klasi *Population*

Izračunate prilagođene dobrote vrsti bitne su za sljedeći korak algoritma koji se izvodi pomoću metode opisane u isječku koda 3.4. Metoda *assign_num_children* pridjeljuje svakoj vrsti broj potomaka za reprodukciju u sljedećem koraku. Svaka vrsta dobiva pravo na broj potomaka proporcionalan svojoj prilagođenoj dobroti. Bitno je da zbroj svih potomaka na kraju bude otprilike jednak veličini vrste definiranoj u postavkama. Hiperparametar za elitizam ne dopušta da se vrsti pridijeli manje od definiranog broja potomaka. Na ovaj način svaka vrsta sigurno će imati barem jednog potomka kako bi imala priliku razviti se, a na detekciji stagnacije je da ukloni vrste kojima treba previše vremena zbog zapinjanja u lokalnom optimumu.

```

1 def assign_num_children(self):
2     total_spawn = 0
3     adjusted_fitness_sum = sum([spec.adjusted_fitness for spec in self.species])
4
5     for spec in self.species:
6         adjusted_fitness = spec.adjusted_fitness
7         size = len(spec.individuals)
8
9         # calculate potential number of children proportionally to species adjusted fitness
10        potential_size = max(config.elitism, adjusted_fitness / adjusted_fitness_sum * config.pop_size)
11        # calculate difference between current size and potential number of children
12        size_delta = potential_size - size
13        # set number of children somewhere between current and potential size depending on smoothing coefficient
14        num_children = size + round(size_delta * config.spawn_smooth_coef)
15
16        spec.num_children = num_children
17        total_spawn += num_children
18
19        # calculate coefficient with which we will normalize all number of children
20        norm = config.pop_size / total_spawn
21
22        for spec in self.species:
23            # by normalizing we assure population size will always be as close to defined as possible
24            spec.num_children = max(config.elitism, round(spec.num_children * norm))
25
26        # if there is no elitism in species it is possible some have 0 children assigned, remove those species if so
27        if config.elitism == 0:
28            log('\t\tRemoving species {:d} because no children were assigned'.format(self.spec.key))
29            self.species = [spec for spec in self.species if spec.num_children > 0]

```

Izvorni tekst programa 3.4: Izvorni tekst metode *assign_num_children* u klasi *Population*

Reprodukcija se odvija tako da objekt populacije sakupi sve jedinke koje su stvorile pojedine vrste. Klasa populacije samo poziva metodu za reprodukciju iz klase vrste, zbog toga što se reprodukcija odvija unutar vrste. Ta metoda objašnjena je u sljedećem podpoglavlju. Nakon reprodukcije, nove jedinke treba odvojiti u vrste. Ovo je prikazano u isječku 3.5. Implementacija je jednostavna, iterira se po novim jedinkama i računa se njihova udaljenost do svakog predstavnika vrste. Jedinka se stavlja u vrstu prvog predstavnika s kojim je kompatibilna. Ako je udaljenost jedinke od svih predstavnika veća od praga definiranoj u postavkama, stvara se nova vrsta te se tu jedinku

postavlja kao predstavnika te nove vrste.

```
1 def speciate(self, children):
2     # assign every individual to a species
3     for individual in children:
4         placed = False
5
6         Es = []
7         Ds = []
8         weight_diffs = []
9
10        # first see if individual is compatible with any existing species
11        for spec in self.species:
12            dist_from_repr, E, D, weight_diff = utility.distance(individual, spec.representative)
13
14            Es.append(E)
15            Ds.append(D)
16            weight_diffs.append(weight_diff)
17
18            # if distance from species representative is smaller than threshold add individual to that species
19            if dist_from_repr <= config.compatibility_threshold:
20                spec.add(individual)
21                placed = True
22                break
23
24            # if individual is not placed to any existing species create new species and set it as representative
25            if not placed:
26                new_spec = Species(self.next_species_key, individual)
27                self.species.append(new_spec)
28                self.next_species_key += 1
29
30        # log empty species
31        for spec in self.species:
32            if len(spec.individuals) == 0:
33                log('\t\tSpecies {:d} is empty after speciation'.format(spec.key))
34
35        # delete any species that is empty after speciation
36        self.species = [spec for spec in self.species if len(spec.individuals) > 0]
37
38        return Es, Ds, weight_diffs
```

Izvorni tekst programa 3.5: Izvorni tekst metode *speciate* u klasi *Population*

Specijacija je dobra zato što omogućuje jedinkama da se bore za prevlast s jedinkama koje su im slične. U ovakvom algoritmu potrebno je imati neki način kojim se potiče razvoj novih struktura, budući da je to jedna od osnovnih premisa ovog algoritma. S obzirom na to da je prag udaljenosti hiperparametar koji treba unaprijed podesiti, jedan od problema koji se može dogoditi je da vrste budu ili prevelike ili premale. Ako su vrste premale jedinke će biti preslične i algoritma neće moći dobro pretraživati prostor rješenja. S druge strane, ako je broj vrsta premali ugušit će se strukturalne promjene koje inicijalno imaju loše težine, ali bi dugoročno mogle biti korisne. Stoga je dodana metoda koja prilagođava prag udaljenosti ovisno o trenutnom broju vrsti i željenom broju vrsti. Ovo smanjuje pritisak odabira dobrog praga, ali stvara novi hiperparametar. Ispitivanjem ponašanja algoritma pokazalo se da je taj hiperparametar robusniji i lakše ga je dobro procijeniti. Metoda je prikazana u isječku 3.6 i gleda je li broj vrsta manji od minimalnog ili veći od maksimalnog traženog. Ako je, prag će se ili smanjiti ili povećati. Ako je broj vrsta unutar željenog raspona ne

događa se ništa. Također, moguće je da se u prvom dijelu algoritma prag previše smanji ili poveća, dok algoritam još zbog mutacije ni ne dobije priliku stvoriti dovoljno različitih jedinki. Kako bi se izbjegla eksplozija broja vrsti, u slučaju da je pređe, novu vrijednost praga na kraju je potrebno zalijepiti na minimalnu ili maksimalnu moguću vrijednost. U sljedećem poglavlju na primjeru jednog od problema prikazan je utjecaj ove metode na brzinu algoritma.

```

1 def adjust_compatibility_threshold(self):
2     num_species = len(self.species)
3
4     # calculate compatibility threshold modifier depending on current number of species
5     delta = 0
6     if num_species > config.max_desired_num_species:
7         delta = config.ct_step
8     elif num_species < config.min_desired_num_species:
9         delta = -config.ct_step
10
11     # adjust current value with calculated modifier but make sure it doesn't go over defined boundaries
12     new_value = config.compatibility_threshold + delta
13     config.compatibility_threshold = np.clip(new_value, config.ct_min_val, config.ct_max_val)

```

Izvorni tekst programa 3.6: Izvorni tekst metode *adjust_compatibility_threshold* u klasi *Population*

3.2.4. Klasa *Species*

Ova klasa služi za praćenje jedinki unutar jedne vrste i za njihovo lako upravljanje. Osim osnovnih metoda kao što su dodavanje jedinki u vrstu, sortiranje, skraćivanje i nasumične selekcije, dodana je još jedna vrsta selekcije koja nije spomenuta u originalnom radu [6]. Selekcija koja se koristila u tom radu je selekcija analogna ruletu (eng. *roulette wheel selection*). Ona odabire nasumično gdje je vjerojatnost odabira jedinke proporcionalna s njenom dobrotom, ali je predstavljala probleme zbog toga što se s njom ne mogu koristiti negativne vrijednosti dobrote. U izvođenjima je nasumična selekcija pokazala bolje rezultate. Ali, selekcija o kojoj se tu priča nije ni nasumična. Selekcija s kojom su se u ispitivanjima uspoređeni rezultati je turnirska selekcija (Kod 3.7). Ova selekcija odabire k jedinki, gdje je k unaprijed definiran u postavkama kao hiperparametar i označava veličinu turnira. Nakon odabira k jedinki samo one se uspoređuju i pobjeđuje prva, tj. prvih n , gdje je n broj jedinki koje se žele zadržati u populaciji ili reproducirati. Zbog toga, veći k znači veći pritisak na jedinke, jer će slabije jedinke imati manju šansu za pobjedu što je veći turnir. Rezultati ispitivanja nalaze se u sljedećem poglavlju.

```

1 # returns single element, tuple or a list based on selection size
2 def tournament_select(self, size=1, replace=False):
3     assert size < config.tournament_size, 'Size must be smaller than tournament size for selection to have effect'
4
5     # randomly select individuals that will be inside the tournament and sort them from best to worst
6     tournament = self.random_select(config.tournament_size, replace)

```



```

7 tournament.sort(key=lambda x: -x.fitness)
8
9 # we return individuals from the top because they are the winners
10 if size == 1:
11     return tournament[0]
12 if size == 2:
13     return tournament[0], tournament[1]
14 else:
15     return tournament[:size]

```

Izvorni tekst programa 3.7: Izvorni tekst metode *tournament_select* u klasi *Species*

Osim selekcije, važna metoda unutar ove klase je metoda za reprodukciju (Kod 3.8). Ova metoda pretpostavlja da je vrsti već pridodijeljen broj potomaka. Metoda kao argument prima dvije varijable koje se mogu vidjeti u isječku koda. Te varijable su rječnici (eng. *dictionary*) koji pohranjuju generacijske inovacije. Ovo je potrebno zato što je vezama između istih čvorova unutar iste generacije potrebno dati isti inovacijski broj. Također, potrebno je prepoznati i nove čvorove između ista dva postojeća. S obzirom na to da se trenutno te promjene prate samo na razini jedne generacije, vrijedi razmotriti opciju u kojoj se ove strukturalne promjene prate na razini svih generacija. Tako bi nove veze između istih čvorova uvijek imale isti inovacijski broj, a ne samo ako su nastale u istoj generaciji.

Prije početka križanja jedinki, vrsta prvo prosljeđuje najbolje jedinke u sljedeću generaciju. Ova karakteristika zove se elitizam (eng. *elitism*). Najčešće se u sljedeću generaciju direktno prenosi samo jedna jedinka, ali njihov broj se može podesiti u postavkama. Nakon elitnih jedinki, brišu se najlošije jedinke koje neće imati pravo imati potomke. Postotak preživljavanja definiran je kao hiperparametar. Iako mali postotak preživljavanja može biti loš jer se tako isto guše moguće korisne strukturalne promjene, do sada je nasumična selekcija i mali postotak preživljavanja pokazao bolje rezultate od turnirske selekcije. Vrijedi isprobati turnirsku selekciju s velikim pritiskom kako bi se dobio sličan efekt ovome, ali na logičniji način. Ova metoda uzima u obzir da su jedinke sortirane, tako da nakon prorjeđivanja ostanu samo najbolje. Nakon toga započinje reprodukcija. Za svaku novu jedinku iznova se gleda kolika je vjerojatnost da se jedinka dobije kloniranjem jedne jedinke ili križanjem dvije. Ako se preskače križanje odabire se jedna jedinka i ona se klonira i mutira, inače se odabiru dvije i nakon križanja mutira se dobivena jedinka. Način selekcije takvih jedinki definiran je u postavkama i može biti obavljen nasumičnom ili turnirskom selekcijom. Kada se prikupi broj jedinki veličine dodijeljenog broja potomaka metoda završava i jedinke se prosljeđuju pozivatelju metode.

```

1 # assumes individuals are sorted
2 def reproduce(self, generation_new_nodes, generation_new_connections):
3     size = len(self.individuals)
4

```

```

5     children = []
6
7     # if elitism is enabled automatically forward best individuals to next generation
8     num_elites = min(config.elitism, size)
9     for i in range(num_elites):
10        child = self.individuals[i].duplicate()
11        children.append(child)
12        self.num_children -= 1
13
14    # remove bottom part of the individuals which are not allowed to reproduce
15    num_surviving = max(2, math.ceil(config.survival_threshold * size))
16    self.trim_to(num_surviving)
17
18    size = len(self.individuals)
19    # create children until assigned cap is reached
20    while self.num_children > 0:
21        # select two parents for crossover or duplicate a single individual
22        if random.random() < config.skip_crossover or size < 2:
23            parent = self.random_select()
24            child = parent.duplicate()
25        else:
26            if not config.tournament_selection or size < config.tournament_size:
27                parent1, parent2 = self.random_select(2)
28            else:
29                parent1, parent2 = self.tournament_select(2)
30
31            child = crossover(parent1, parent2)
32
33        # after we have a child, mutate it
34        child.mutate(generation_new_nodes, generation_new_connections)
35
36        children.append(child)
37        self.num_children -= 1
38
39    # reset species, remove all individuals and set a random one as a representative
40    self.reset()
41
42    return children

```

Izvorni tekst programa 3.8: Izvorni tekst metode *reproduce* u klasi *Species*

3.2.5. Klasa *Individual*

Klasa koja predstavlja jedinku ujedno sadrži u sebi i genom. Zbog toga, ako prilikom instanciranja nisu proslijeđene veze i čvorovi, jedinka mora stvoriti nove. Ovo je prikazano u isječku 3.9. Za svaki ulaz i izlaz stvara se novi čvor s nasumičnom pristranošću (eng. *bias*). Nakon toga između svih ulaznih i izlaznih čvorova stvaraju se veze s nasumičnom težinom. U originalnom radu [6] čvorovi nemaju težinu, nego su autori u ispitivanjima imali još jedan ulaz u mrežu koji je uvijek imao vrijednost 1. U ispitivanjima se pokazalo da je bolje imati pristranost ugrađenu u čvor koji onda mutira slično kao težina veze. Zbog toga, zapis čvora se također smatra dijelom genoma i čvorovi se prate jedinstvenim ključem isto kao i veze inovacijskim brojem. Naravno, prilikom križanja dijete roditelja preuzima karakteristike čvorova isto kao i težina.

```

1 def configure_new(self):
2     # create a node for every input and output defined by the problem
3     for key in config.input_keys + config.output_keys:
4         # pick random bias value with gaussian distribution
5         bias = random.gauss(config.bias_new_mu, config.bias_new_sigma)

```

```

6         node = Node(key, bias)
7         self.nodes[key] = node
8
9     next_innovation_number = 0
10    # fully connect inputs and outputs, i.e. create a connection between every input and output node
11    for input_key in config.input_keys:
12        for output_key in config.output_keys:
13            # pick random connection weight value with gaussian distribution
14            new_connection = Connection(next_innovation_number, input_key, output_key, random.gauss(config.
weight_new_mu, config.weight_new_sigma), True)
15            self.connections[next_innovation_number] = new_connection
16            next_innovation_number += 1

```

Izvorni tekst programa 3.9: Izvorni tekst metode *configure_new* u klasi *Individual*

Mutacija je bitna kod genetskih algoritama zato što osim križanja određuje kako se populacija kreće po prostoru rješenja. Kao što je vidljivo u isječku 3.10, prilikom mutiranja jedinke moguće su tri vrste mutacije. Zbog dodane pristranosti u čvorovima, u ovoj implementaciji postoji i četvrta vrsta mutacije, mutacija čvorova. U postavkama je definirano kolike su vjerojatnosti svake od mutacija.

```

1 def mutate(self, generation_new_nodes, generation_new_connections):
2     # new connection mutation
3     if random.random() < config.new_connection_probability:
4         self.new_connection(generation_new_connections)
5
6     # new node mutation
7     if random.random() < config.new_node_probability:
8         self.new_node(generation_new_nodes, generation_new_connections)
9
10    # connection weight and node bias mutations
11    self.mutate_connections()
12    self.mutate_nodes()

```

Izvorni tekst programa 3.10: Izvorni tekst metode *mutate* u klasi *Individual*

Mutacije veza i čvorova (Kod 3.11) su jednostavne, iterira se po čvorovima ili vezama i ovisno o vjerojatnostima zadanim u postavkama vrijednosti se perturbiraju ili ih se resetira na nove nasumične vrijednosti.

```

1 def mutate_connections(self):
2     # go through all connections and either adjust weight by a small amount or assign a new random one
3     for connection in self.connections.values():
4         r = random.random()
5         if r < config.weight_perturbation_probability:
6             connection.weight += random.gauss(config.weight_step_mu, config.weight_step_sigma)
7         elif r < config.weight_perturbation_probability + config.weight_replace_probability:
8             connection.weight = random.gauss(config.weight_new_mu, config.weight_new_sigma)
9
10 def mutate_nodes(self):
11    # go through all nodes and either adjust bias by a small amount or assign a new random one
12    for node in self.nodes.values():
13        r = random.random()
14        if r < config.bias_perturbation_probability:
15            node.bias += random.gauss(config.bias_step_mu, config.bias_step_sigma)
16        elif r < config.bias_perturbation_probability + config.bias_replace_probability:
17            node.bias = random.gauss(config.bias_new_mu, config.bias_new_sigma)

```

Izvorni tekst programa 3.11: Izvorni tekst metodi *mutate_connections* i *mutate_nodes* u klasi *Individual*

Stvaranje nove veze je potencijalno najzahtjevnija metoda (kod 3.12). Potrebno je paziti da se ne pokuša stvoriti veza kojom bi graf postao kružni. Zbog jednostavnosti poželjno je da neuronska mreža bude samo unaprijedna (eng. *feedforward*). Moguće je jedino u postavkama omogućiti stvaranje povratnih veza. Te veze bi započinjale i završavale u istom čvoru. Uz povratnu vezu čvor bi mogao uzeti u obzir svoje prethodne vrijednosti te bi se tako uvela memorija u neuronsku mrežu. Ovo je potrebno u problemima koji nemaju Markovljevo svojstvo (non-Markovian property). Kako bi se spriječila beskonačna petlja na početku metode prvo se provjerava je li uopće moguće dodati novu vezu. Ako je, pokušavaju se odabrati dva čvora takva da veza između njih već ne postoji. Ako se primijeti da bi novonastala veza stvorila kružnu vezu, usmjerenje veze se obrne. Jednom kada je to obavljeno, pomoću rječnika koji smo primili kao argument gleda se je li takva strukturalna promjena već nastala. Sukladno tome daje joj se ili prikladni inovacijski broj ili joj se pridodaje novi i strukturalna promjena se sprema za buduće mutacije. Nova veza ima nasumičnu težinu.

```

1 def new_connection(self, generation_new_connections):
2     node_keys = list(self.nodes.keys())
3     num_nodes = len(node_keys)
4     num_connections = len(self.connections.values())
5     num_inputs = len(config.input_keys)
6     num_outputs = len(config.output_keys)
7
8     # exit if it is not possible to create any new connections
9     if num_connections == utility.max_num_edges(num_nodes) - (utility.max_num_edges(num_inputs) + utility.
10        max_num_edges(num_outputs)):
11         return
12
13     while True:
14         # pick two random nodes
15         node1_key = random.choice(node_keys[num_inputs + num_outputs:])
16         node2_key = random.choice(node_keys)
17
18         # try again if chosen nodes are the same and self loops are disabled
19         if config.disable_self_loops and node1_key == node2_key:
20             continue
21
22         # try again if there is already an existing connection between chosen nodes
23         existing_connections = [c for c in self.connections.values() if c.from_key == node1_key and c.to_key ==
24            node2_key or c.from_key == node2_key and c.to_key == node1_key]
25         if existing_connections:
26             continue
27
28         # switch node positions if adding this link would make network recurrent
29         if node2_key in config.input_keys or utility.check_if_path_exists_by_connections(node2_key, node1_key,
30            self.connections) or (node2_key, node1_key) in generation_new_connections:
31             temp = node1_key
32             node1_key = node2_key
33             node2_key = temp
34
35         # assign new innovation number or assign existing if structural innovation has already occurred
36         key_pair = (node1_key, node2_key)
37         if key_pair in generation_new_connections:
38             innovation_number = generation_new_connections[key_pair]
39         else:
40             innovation_number = config.innovation_number
41             generation_new_connections[key_pair] = innovation_number
42             config.innovation_number += 1
43
44         # create new connection with random weight
45         new_connection = Connection(innovation_number, node1_key, node2_key, random.gauss(config.weight_new_mu,

```

```

43     config.weight_new_sigma), True)
44     self.connections[innovation_number] = new_connection
45     return

```

Izvorni tekst programa 3.12: Izvorni tekst metode *new_connection* u klasi *Individual*

Dodavanje novog čvora ima nešto jednostavniju implementaciju (Kod 3.13). Potrebno je samo iz liste veza izabrati jednu nasumično. U predanim generacijskim inovacijama gleda se je li čvor na ovom mjestu već stvoren. Ako je stvoren, uzima se postojeći ključ, a ako nije, pridjeljuje se novi i sprema se inovacija. Nakon toga, potrebno je stvoriti novi čvor s nasumičnom pristranošću i dvije nove veze. Staru vezu na tom mjestu se deaktivira. Jedna nova veza dobije težinu stare, dok se drugoj početna težina postavlja na 1.

```

1 def new_node(self, generation_new_nodes, generation_new_connections):
2     connections_values = list(self.connections.values())
3     # choose a random connection and disable it
4     connection = random.choice(connections_values)
5     connection.enabled = False
6
7     key_pair = (connection.from_key, connection.to_key)
8     # if a node has already been added between these two nodes assign existing innovation numbers and node key
9     if key_pair in generation_new_nodes:
10        new_node_key = generation_new_nodes[key_pair]
11
12        innovation_number1 = generation_new_connections[(connection.from_key, new_node_key)]
13
14        innovation_number2 = generation_new_connections[(new_node_key, connection.to_key)]
15    # otherwise create new and remember structural innovation
16    else:
17        new_node_key = config.next_node_key
18        generation_new_nodes[key_pair] = new_node_key
19        config.next_node_key += 1
20
21        innovation_number1 = config.innovation_number
22        generation_new_connections[(connection.from_key, new_node_key)] = innovation_number1
23        config.innovation_number += 1
24
25        innovation_number2 = config.innovation_number
26        generation_new_connections[(new_node_key, connection.to_key)] = innovation_number2
27        config.innovation_number += 1
28
29    # create a new node with random bias
30    bias = random.gauss(config.bias_new_mu, config.bias_new_sigma)
31    new_node = Node(new_node_key, bias)
32    self.nodes[new_node_key] = new_node
33
34    # create a new connection and set it's value to 1.0
35    new_connection1 = Connection(innovation_number1, connection.from_key, new_node_key, 1.0, True)
36    self.connections[innovation_number1] = new_connection1
37
38    # create a new connection and set it's value to the value of disabled connection
39    new_connection2 = Connection(innovation_number2, new_node_key, connection.to_key, connection.weight, True)
40    self.connections[innovation_number2] = new_connection2

```

Izvorni tekst programa 3.13: Izvorni tekst metode *new_node* u klasi *Individual*

Zadnja bitna metoda iz ove klase je metoda križanja (eng. *crossover*) (Kod 3.14). Ovo je statična metoda koja kao argumente prima roditelje. Budući da u slučajevima dislociranih gena dijete preuzima gen boljeg roditelja, na početku se odmah pokušava odrediti koji je to. U slučaju da su roditelji jednake dobrote, kao boljeg se gleda onog

s jednostavnijom strukturom. Način na koji se evaluira dobrota definirana je za svaki problem u sljedećem poglavlju rada. Metoda pretpostavlja da je već poznata dobrota za obje jedinke.

Nakon što se odredi bolji roditelj, za svaki inovacijski broj boljeg roditelja gleda se ima li drugi roditelj taj gen. U slučaju da ima, dijete nasumično dobiva gen jednog ili drugog roditelja. Ako drugi roditelj nema taj gen, dijete ga direktno nasljeđuje. Isti postupak ponavlja se za čvorove. Na kraju se s ovim genima i čvorovima stvara nova jedinica koja predstavlja dijete te se vraća pozivatelju metode.

```
1 def crossover(parent1, parent2):
2     # set fitter parent as parent1
3     if parent1.fitness > parent2.fitness:
4         fitter_parent, other_parent = (parent1, parent2)
5     elif parent1.fitness == parent2.fitness:
6         # if they have the same fitness, set the one with simpler structure as the fitter parent
7         if len(parent1.connections) < len(parent2.connections):
8             fitter_parent, other_parent = (parent1, parent2)
9         else:
10            fitter_parent, other_parent = (parent2, parent1)
11    else:
12        fitter_parent, other_parent = (parent2, parent1)
13
14    child_connections = {}
15    child_nodes = {}
16
17    # iterate through all connections from fitter parent
18    for innovation_number, connection in fitter_parent.connections.items():
19        # if other parent has the same gene assign a random parent's gene to the child
20        if innovation_number in other_parent.connections:
21            other_connection = other_parent.connections[innovation_number]
22
23            if random.random() < 0.5:
24                new_connection = copy.deepcopy(connection)
25            else:
26                new_connection = copy.deepcopy(other_connection)
27
28            if not connection.enabled or not other_connection.enabled:
29                new_connection.enabled = random.random() > config.stay_disabled_probability
30
31            child_connections[innovation_number] = new_connection
32    # otherwise just copy fitter parent's gene
33    else:
34        new_connection = copy.deepcopy(connection)
35
36        if not connection.enabled:
37            new_connection.enabled = random.random() > config.stay_disabled_probability
38
39        child_connections[innovation_number] = new_connection
40
41    # go through all nodes from the fitter parent
42    for key, node in fitter_parent.nodes.items():
43        # if other parent has the same node choose a random one
44        if key in other_parent.nodes:
45            other_node = other_parent.nodes[key]
46
47            if random.random() < 0.5:
48                new_node = copy.deepcopy(node)
49            else:
50                new_node = copy.deepcopy(other_node)
51
52            child_nodes[key] = new_node
53    # otherwise copy fitter parent's node
54    else:
55        new_node = copy.deepcopy(node)
56        child_nodes[key] = new_node
57
```

```

58 # create a new child with chosen connections and nodes
59 child = Individual(child_connections, child_nodes)
60 return child

```

Izvorni tekst programa 3.14: Izvorni tekst metode *crossover* u klasi *Individual*

3.2.6. Klasa *Phenotype*

Zadatak klasa *Phenotype* je da genom pretvori u neuronsku mrežu. Pomoću gena veza i čvorova bilokoje jedinice stvara neuronsku mrežu koju možemo unaprijedno pokrenuti. Prilikom inicijalizacije (Kod 3.15) definira se ono što predstavlja genom jedinice. Prvo se iterira kroz čvorove i stvaraju se neuroni s odgovarajućom pristranošću. Nakon toga prolazi se kroz veze i za svaku se obavještava odgovarajući izlazni i ulazni neuron. Bitno je odvojeno spremati ulazne i izlazne veze neurona kako bi se znalo s kojima računati vrijednost, a koje okinuti nakon izračuna.

```

1 def __init__(self, connections, nodes):
2     self.neurons = {}
3
4     # create a node for every neuron
5     for node in nodes:
6         self.neurons[node.key] = Neuron(node.key, node.bias)
7
8     # iterate through all connections
9     for connection in connections:
10        # skip disabled connections
11        if not connection.enabled:
12            continue
13
14        # save incoming key to outgoing neuron
15        self.neurons[connection.from_key].add_outgoing(connection.to_key)
16
17        # save connection to incoming neuron
18        recurrent = utility.check_if_path_exists_by_neurons(connection.to_key, connection.from_key, self.neurons)
19        self.neurons[connection.to_key].add_incoming(connection, recurrent)

```

Izvorni tekst programa 3.15: Izvorni tekst metode *__init__* u klasi *Phenotype*

Jednom kada je fenotip konstruiran na njemu se može izvesti unaprijedni prolaz. To se radi metodom opisanom u isječku koda 3.16. Metoda kao argument prima ulaz u mrežu. Pretpostavlja se da je to lista dimenzije broja ulaznih čvorova. Iz konfiguracijske datoteke se gledaju ključevi ulaznih čvorova i njima se postavljaju vrijednosti. Neuroni se okidaju i propagiraju aktivaciju kroz mrežu do izlaznih čvorova. Na kraju se mogu preuzeti vrijednosti izlaznih neurona i to se vraća kao izlaz funkcije. U sljedećem podpoglavlju objašnjena je propagacija neurona.

```

1 def forward(self, inputs):
2     # first reset all neurons
3     for neuron in self.neurons.values():
4         neuron.reset()
5
6     # set value for input neurons
7     for key, value in zip(config.input_keys, inputs):
8         self.neurons[key].set_value(value, self.neurons)

```

```

9
10 # neurons will automatically propagate and we can collect values from output nodes
11 output = [self.neurons[key].value for key in config.output_keys]
12 return output

```

Izvorni tekst programa 3.16: Izvorni tekst metode *forward* u klasi *Phenotype*

3.2.7. Klasa *Neuron*

U isječku koda 3.17 prikazane su tri bitne metode za propagiranje aktivacija. Metoda *trigger_outgoing* šalje svim neuronima u koje se taj neuron spaja da prihvate njegovu aktivaciju. Metoda *accept* prihvaća aktivaciju i s obzirom na broj ulaznih veza broji koliko aktivacija je još preostalo. Ako je neuron zaprimio dovoljno aktivacija od neurona koji ulaze od njega, onda izračunava svoju vrijednost i propagira aktivaciju daljnjim neuronima. Ovo se ponavlja dok se ne dođe do neurona koji nema izlazne veze. Vrijednost neurona računa se pomoću *calculate_value* metode. Ova metoda uzima vrijednosti svih neurona koji ulaze u trenutni i množi ih s težinama veza. Ove vrijednosti i pristranost zbrajaju se i provlače kroz aktivacijsku funkciju. U ovom slučaju to je sigmoidalna funkcija.

```

1 def calculate_value(self, neurons):
2     # start from bias
3     score = self.bias
4
5     # for every incoming connection add incoming nodes value adjusted by connection weight
6     for connection in self.incoming_connections:
7         from_neuron = neurons[connection.from_key]
8         score += connection.weight * from_neuron.value
9
10    # run sum through sigmoid activation
11    self.value = utility.sigmoid(score)
12
13 def trigger_outgoing(self, neurons):
14    # activate neurons for every outgoing connection
15    for key in self.outgoing_keys:
16        outgoing_neuron = neurons[key]
17        outgoing_neuron.accept(neurons)

```

Izvorni tekst programa 3.17: Metode bitne za propagiranje aktivacija neurona u klasi *Neuron*

3.2.8. Ostale metode

Preostale klase uglavnom samo pomažu izvođenju programa i vizualizaciji. Jedina preostala bitna metoda je metoda iz *utility* paketa za izračun udaljenosti između jedinki (Kod 3.18). Ova metoda prima dvije jedinke i računa broj dislociranih gena, gena koji se ne preklapaju na kraju i prosječne razlike u težinama. Ovo radi tako da iterira po uniji svih inovacijski brojeva ove dvije jedinke i uspoređuje gene. Ako obje jedinke imaju taj gen, računa se razlika u težini. U suprotnom, gleda se je li gen dislocirani

ili nepreklapajući s kraja. Na kraju se te vrijednosti množe s odgovarajućim koeficijentima definiranim i vraća se njihov zbroj. U sljedećem poglavlju ova implementacija ispitana je na problemima predstavljenim u prethodnom poglavlju.

```
1 def distance(individual1, individual2):
2     connections1 = individual1.connections
3     connections2 = individual2.connections
4
5     max_innovation_number1 = max(individual1.connections.keys())
6     max_innovation_number2 = max(individual2.connections.keys())
7     max_common_innovation_number = min(max_innovation_number1, max_innovation_number2)
8
9     all_innovation_numbers = set(connections1.keys()).union(set(connections2.keys()))
10
11     weight_diffs = []
12     E = 0
13     D = 0
14     max_num_nodes = max(len(connections1), len(connections2))
15     max_num_hidden = max_num_nodes - config.num_starting_nodes
16     N = max_num_hidden if config.normalize else 1.
17
18     # iterate through all innovation numbers in either of the parents
19     for innovation_number in all_innovation_numbers:
20         # if both parents have that gene calculate weight difference
21         if innovation_number in connections1 and innovation_number in connections2:
22             weight_diff = abs(connections1[innovation_number].weight - connections2[innovation_number].weight)
23             weight_diffs.append(weight_diff)
24         # else increment number of excess or disjoint genes depending on the location of the gene
25         else:
26             if innovation_number > max_common_innovation_number:
27                 E += 1
28             else:
29                 D += 1
30
31     # normalize excess and disjoint genes
32     adjusted_E = (config.c1 * E) / N
33     adjusted_D = (config.c2 * D) / N
34     # calculate average weight difference
35     avg_weight_diff = sum(weight_diffs) / len(weight_diffs)
36     adjusted_weight_diff = config.c3 * avg_weight_diff
37     # calculate distance
38     delta = adjusted_E + adjusted_D + adjusted_weight_diff
39     return delta, adjusted_D, adjusted_E, adjusted_weight_diff
```

Izvorni tekst programa 3.18: Izvorni tekst metode *distance* iz paketa *utility*

4. Opis problema

Programska izvedba neuroevolucije promjenjivih topologija ispitana je na jednom jednostavnom problemu, jednoj simulaciji kontrole pokreta i jednoj videoigri. Cilj je usporediti ostvarenje s izvedbom iz originalnog rada [6] i pokušati postići što bolji rezultat u videoigrama. U ovom poglavlju objašnjeni su problemi i pravila okruženja, dok su u kasnijem poglavlju prikazane postavke izvedbi i rezultati. Za svako okruženje potrebno je definirati nekoliko stvari. S obzirom na to da u okviru algoritma svako moguće rješenje predstavlja jednu neuronsku mrežu koja donosi odluke, treba definirati koliko će ulaza i izlaza u mrežu trebati biti u svakom trenutku problema. Ovo je određeno brojem parametara koji će definirati stanje igre i brojem akcija koje agent može napraviti u svakom trenutku. Osim toga, potreban je način na koji će se mreže ocijeniti, tako da okruženje treba imati neki način da ocijeni kvalitetu svakog pokušaja rješavanja igre. Sukladno tome, treća i zadnja stvar je uvjet rješavanja problema. Kako bi se algoritam mogao zaustaviti na vrijeme treba biti poznato koji je zahtjev koji se treba ispuniti da bi se problem smatrao riješenim. U nastavku su definirana sva tri problema.

4.1. Problem konstrukcije XOR vrata

Prvo okruženje u kojemu je ispitan algoritam je problem konstrukcije XOR logičkih vrata. XOR vrata primaju dva ulaza i daju jedan izlaz i svaki može biti samo 0 ili 1, tako da model mora moći točno odgovoriti u četiri slučaja. Sva četiri slučaja prikazana su u tablici 4.1. Stoga, cilj je naučiti agenta koji prima dva ulaza i vraća jedan izlaz. Problem se smatra riješenim kada agent za sve četiri kombinacije ulaza vrati točan izlaz. Kako bi se mogao ocijeniti agent koristi se sljedeća formula:

$$f_i = 4 - \sum_{j=0}^4 (y_j - h_i(\sigma_j))^2 \quad (4.1)$$

Dobrota jedinice i označena je s f_i . h_i predstavlja funkciju predviđanja izlaza

Ulaz 1	Ulaz 2	Izlaz
0	0	0
0	1	1
1	0	1
1	1	0

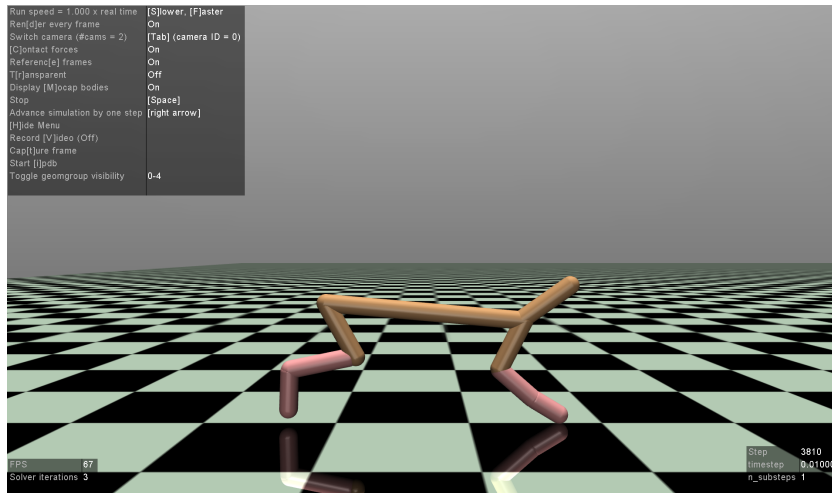
Tablica 4.1: Sva četiri slučaja ulaza i izlaza iz XOR vrata

agenta i za parametre σ_j , dok je y_j stvarni izlaz za primjer j . Dakle, radi se o zbroju kvadratnih udaljenosti. Ovaj zbroj se onda oduzima od 4 kako bi veći rezultat predstavljao bolje rješenje.

Bitno je napomenuti da je uvedena mala promjena u opis zadatka u odnosu na definiciju iz rada [6] s kojim je uspoređen. U navedenom radu dodan je treći ulaz koji služi za dodavanje pristranosti (eng. *bias*) u čvor. Vrijednost tog ulaza uvijek bi bila 1. Budući da je pristranost u ovoj izvedbi drugačije izvedena, taj ulaz je maknut. U kasnijem poglavlju analizirana je ta odluka i dana je usporedba u izvođenju.

4.2. Radno okruženje *HalfCheetah*

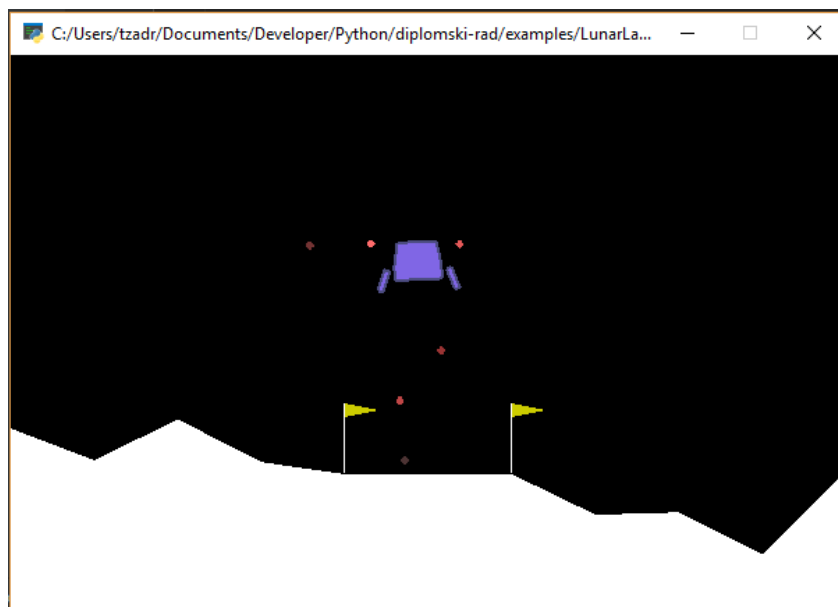
HalfCheetah naziv je jednog od okruženja iz okvira kojeg se koristi za igranje videoigara u programskom jeziku *Python*. Cilj je naučiti agenta koji konstitucijom nalichi gepardu da trči (Slika 4.1). Ovo okruženje jednostavnija je varijanta čovjekolikog agenta koji uči hodati, spomenutog u prethodnom poglavlju. Razlika između ta dva je u broju parametara kojima treba upravljati. Ova varijanta ima puno manje ulaza i izlaza što čini pretraživanje prostora rješenja jednostavnijim. Agent kojeg treba naučiti trčati primao bi točnije sedamnaest ulaza, a kao svoju akciju bi slao šest brojeva. Za razliku od ostala dva problema, ovo je jedini koji ima kontinuirani prostor akcija (eng. *continuous action space*), što znači da brojevi koje vraća mogu biti bilo koji brojevi u rasponu $[-1, 1]$. Svaki od tih brojeva predstavlja željeni okretni moment jednog od šest zglobova kojima agent upravlja. Agent šalje željenu akciju za svaki od 5,000 otkačaja koliko ima da pređe što veću udaljenost. Ta udaljenost na kraju gleda se kao njegov rezultat s kojim ga je moguće uspoređivati s ostalim rješenjima. Ovo okruženje nema uvjet završetka, stoga algoritam samo pokušava u zadani broj generacija postići što bolji rezultat.



Slika 4.1: Prikaz agenta u okruženju *HalfCheetah*

4.3. Radno okruženje *LunarLander*

LunarLander je računalna igra iz istog paketa i u njoj je cilj sletjeti na tlo bez oštećenja (Slika 4.2). Kao i prošlo okruženje, za svaki trenutak agentu se predaju neka zapažanja, a on ih prevodi u akcije. U ovom slučaju agent ima osam realnih brojeva koji opisuju stanje igre u svakom trenutku i može se odlučiti za jednu od četiri radnje. Neki od brojeva koji opisuju stanje su trenutna x pozicija, y pozicija, kut pod kojim je letjelica, dodiruju li lijeva ili desna noga tlo, itd. Od mogućih akcija, agent može upaliti donji, lijevi ili desni motor te ne napraviti ništa. U svakom trenutku ovisno o stanju i svojoj odluci dobiva određeni broj bodova. Za svaki otkucaj za koji agent pali motore gubi 0.3 boda, dok za svaki doticaj jedne od dvije noge s tlom dobiva 10 bodova. Na ovaj način agent se nagrađuje za što manje korištenje motora i što mekše slijetanje. Dobiva više bodova ako sleti bliže sredini ekrana gdje mu je označeno mjesto slijetanja. Sredina ekrana uvijek ima ravnu podlogu i nalazi se na koordinatama $(0, 0)$, dok se okolina može mijenjati. Na kraju izvođenja, agent dobije 100 bodova ako se uspješno zaustavio ili izgubi 100 ako se sudario. Osim što se okruženje može mijenjati, letjelica može započeti pod drugačijim kutem i početnom brzinom. Igra se smatra riješenom kada jedan agent u sto različitih pokretanja postigne u prosjeku 200 bodova po pokušaju. U sljedećem poglavlju provedena su ispitivanja za opisane probleme i analizirana je efikasnost algoritma u stvaranju strategije igranja za iste.



Slika 4.2: Prikaz agenta u okruženju *LunarLander*

5. Ispitivanje

U ovom poglavlju opisani su rezultati izvođenja za probleme predstavljene u drugom poglavlju.

5.1. Problem konstrukcije XOR vrata

U prethodnom poglavlju opisana je jednakost 4.1 kojom se računa dobrota svake jedinice za ovaj problem. U isječku koda 5.1 prikazano je ostvarenje te formule unutar metode *evaluate* kojom se računa dobrota svake jedinice. Metoda prima referencu na jedinku te konstruira fenotip koji je opisan jedinkinim genima. Fenotipu se za sva četiri slučaja problema predaju ulazi, nakon čega se uzima izlaz iz mreže. Pomoću izlaza iz fenotipa računamo pogrešku i time dobrotu jedinice. Ako mreža vrati točan rezultat za sva četiri slučaja spremamo informaciju da je okruženje riješeno. Na glavnom programu je da provjeri tu zastavicu ako želi zaustaviti izvođenje algoritma.

```
1 def evaluate(self, individual):
2     # track number of evaluations made
3     self.evaluations += 1
4
5     # create phenotype from individual's connection and node genes
6     phenotype = Phenotype(individual.connections.values(), individual.nodes.values())
7
8     correct_solutions = True
9     fitness = 4
10
11    # for every one of four different cases feed the network input and calculate error based on the output
12    for observation, solution in zip(self.observations, self.solutions):
13        phenotype.flush()
14        output = phenotype.forward(observation)
15        result = output[0]
16
17        correct_solutions = correct_solutions and round(result) == solution
18        fitness -= (solution - result)**2
19
20    # if solution made no errors the environment is considered solved
21    self.solved = self.solved or correct_solutions
22    return fitness
```

Izvorni tekst programa 5.1: Izvorni tekst metode *evaluate* za problem konstrukcije XOR vrata

Cilj ovog problema je usporediti rezultate dobivene implementacijom ostvarenom

u okviru ovog rada s rezultatima istog problema iz rada u kojem je algoritam originalno predložen [6]. Autorima rada trebalo je na sto pokretanja u prosjeku 32 generacije da nađu rješenje, a genom koji rješava problem u prosjeku ima 2.35 skrivena čvora. U ispitivanju predložene implementacije algoritma imitirani su hiperparametri iz originalnog rada. S tim postavkama ova implementacija u sto pokretanja pronalazi rješenje u prosječno 24 generacije, što je za četvrtinu manje vremena. Također, najbolja rješenja ovog ostvarenja imaju u prosjeku 1.08 skrivena čvora, što je čak duplo bolje. Važno je napomenuti da optimalno rješenje ovog problema ima jedan skriveni čvor, što znači da se ova implementacija jako približila optimalnom rješenju. U oba ispitivanja rješenje je pronađeno u svih sto pokretanja, gdje je najduže izvođenje originalnog rada bilo 90 generacija, a ovog skoro trećinu manje, 65 generacija. Više informacija o izvođenjima prikazano je u tablici 5.1. Ova tablica sadrži informacije o izvođenju i varijante predloženog algoritma koja ima zaseban ulaz za pristranost.

5.1.1. Pristranost u čvorovima

U prethodnom poglavlju spomenuto je da originalni rad izvodi pristranost tako da prilikom izvođenja ima još jedan ulaz u mrežu kojem je vrijednost uvijek 1. Na ovaj način veze iz ovog čvora u bilo koji drugi funkcioniraju isto kao pristranost u jednostavnim neuronima. Ova implementacija ne sadrži taj ulaz zato što je pristranost uvedena direktno u čvorove. U tablici 5.1 vidi se da varijanta ove izvedbe sa zasebnim ulazom za pristranost ostvaruje puno lošije rezultate, i od originalne implementacije i od predložene. Algoritam u sto izvođenja ne nalazi rješenje problema unutar sto generacija u čak šest slučajeva. U ostalih 94, ostvarenje pronalazi rješenje s prosječno 1.19 skrivena čvora, što nije puno lošije od predložene varijante, ali ga pronalazi u prosječno 44 generacije, što je skoro duplo duže. Pretpostavka zašto ova varijanta postiže lošije rezultate je to što, ako algoritam ima poseban ulaz za pristranost, pristranost u čvor će moći biti dodana tek kada se stvori nova veza između tog ulaza i čvora. Pristranost je bitan parametar neurona te se njegova vrijednost treba prilagođavati puno češće nego što se dodaju nove veze. Zbog toga je u predloženoj implementaciji odlučeno pristranost u čvorovima prilagođavati na sličan način i približnom učestalošću kao i težine veza. S ovom promjenom algoritam je u stanju postići puno bolje rezultate, kao što je vidljivo u tablici 5.1.

Problem konstrukcije XOR vrata		Originalna implementacija (poseban ulaz za pristranost)	Imitacija originalne implementacije (poseban ulaz za pristranost)	Predložena implementacija (pristranost ugrađena u čvorove)
Broj evaluacija	Prosječno	4,755 (32 generacije)	6,701 (44 generacije)	3,734 (24 generacije)
	Standardna devijacija	2,553	2,997	1,417
	Najgore pokretanje	13,459 (90 generacija)	-	9,891 (65 generacija)
Broj skrivenih čvorova	Prosječno	2.35	1.19	1.08
	Standardna devijacija	1.11	0.42	0.31
Broj aktivnih veza	Prosječno	7.48	6.64	5.12
Broj pokretanja u koliko je algoritam pronašao rješenje u 100 generacija		100/100	94/100	100/100

Tablica 5.1: Prikaz podataka o izvršavanju algoritma na problemu konstrukcije XOR vrata

5.2. Optimiranje strategije hodanja u simulaciji *Half-Cheetah*

Za ovaj problem metodu za evaluaciju jedinke trebalo je prilagoditi da odgovara načinu rada biblioteke koja se koristi za pokretanje simulacije. U izvornom kodu isječka 5.2 opisana je ta metoda. Glavna promjena je interakcija s objektom *env* koji predstavlja instancu igre kreirane uz pomoć programske biblioteke *gym* koja je opisana u prethodnom poglavlju. Metoda nakon konstrukcije fenotipa koristi objekt *env* kako bi započela novu igru te prikupljala zapažanja i slala akcije agenta. Kako je prethodno opisano, obzervacije predstavljaju ulaz u neuronsku mrežu, to jest fenotip, a izlaz iz te mreže predstavlja akciju koju agent poduzima. Svaki puta kada agent napravi korak, okruženje osim zapažanja vraća i nagradu i zastavicu koja označava je li igra gotova. Nagrade iz svakog koraka se akumuliraju kako bi se dobila dobrota agenta, a kada je zastavica za kraj igre gotova petlja se može zaustaviti i vratiti do tad akumuliranu količinu nagrada, to jest dobrotu jedinke.

```

1 def evaluate(self, individual, fixed_seed=True, render=False):
2     # create phenotype from individual's connection and node genes
3     phenotype = Phenotype(individual.connections.values(), individual.nodes.values())
4
5     fitness = 0
6
7     if fixed_seed:
8         # ensures every run starts with same observation
9         self.env.seed(self.seed)
10
11     # start new game
12     observation = self.env.reset()
13     while True:
14         if render:
15             # displays game state on screen
16             self.env.render()
17
18         # feed forward neural network with observation
19         output = phenotype.forward(observation)
20         # scale output to get action
21         action = utility.scale(output, self.env.action_space.low, self.env.action_space.high)
22         # take step with given action
23         observation, reward, done, info = self.env.step(action)
24
25         # accumulate rewards to get individual's fitness
26         fitness += reward

```



```
27
28     # stop if game is over
29     if done:
30         break
31
32     return fitness
```

Izvorni tekst programa 5.2: Izvorni tekst metode *evaluate* za problem optimiranja strategije hodanja u simulaciji *HalfCheetah*

Postavke hiperparametara za ovaj problem ne razlikuju se mnogo od prethodnog problema. Promijenjene su jedino tražena veličina populacije te vjerojatnost generiranja novih veza i čvorova, koje u ovom ispitivanju imaju duplo veću vrijednost. Veća populacija potrebna je zato što je domena ovog problema puno veća i potrebno ju je bolje pretraživati. Također, veća populacija ostavlja i više slobode kod dodatnih mutacija. Sve ostale postavke su nepromijenjene. Kako bi ispitivanje bilo olakšano svako pokretanje simulacije počinje iz istog početnog stanja. Na ovaj način osiguran je determinizam, to jest svako pokretanje simulacije s istim fenotipom dat će isti rezultat. U sljedećem podpoglavlju je opisano kako pokretanje s nasumičnim početnim stanjem utječe na rad algoritma.

Rezultati izvođenja algoritma na ovom problemu prikazani su u tablici 5.2. Kao što je vidljivo iz pet pokretanja algoritma, najbolji agent je u prosjeku postizao rezultat od 2, 146.28. Na internetu nisu dostupni dobro opisani rezultati drugih rješenja ovog problema, pa nije moguće ovaj rezultat saviti u perspektivu, ali po samoj motorici agenta moguće je zaključiti da agent u većini slučajeva upadne u lokalni optimum iz kojeg ne može izaći. Naime, prilikom trčanja agent ne miče noge efikasno kako bi iskoristio raspon koji može prijeći s njima. Kako je opisano u prethodnom poglavlju, agent upravlja robotom u obliku geparda, stoga ima noge s tri članka: natkoljenica, potkoljenica te stopalo. Agenti u pitanju trče tako da jako brzo kreću stopala, dok im natkoljenica i potkoljenica uglavnom miruju. Koliko god brzo radili, agenti tako neće moći nikada trčati onoliko brzo kolikom bi mogli da koriste i preostala dva dijela noge. Bitno je napomenuti da je poželjno da agent što brže trči jer se dobrota mjeri kao pređena udaljenost u određenom periodu, to jest u određenom broju poteza. U ostatku poglavlja analizirano je korištenje turnirske selekcije umjesto nasumičnog odabira roditelja.

5.2.1. Turnirska selekcija

U tablici 5.2 također je moguće vidjeti rezultate pokretanja i za drugi slučaj koji je predmet ovog potpoglavlja. U prethodnom poglavlju opisana je turnirska selekcija te kako se ona razlikuje od nasumičnog odabira. Ovo ostvarenje, kao i originalni al-

Postavke	Najbolja jedinka nakon 100 generacija	Pokretanje	Pokretanje	Pokretanje	Pokretanje	Pokretanje	Prosječno	Standardna devijacija
		1	2	3	4	5		
Visoka stopa smrtnosti (80%) & Nasumičan odabir roditelja	Rezultat	1,438	2,193	2,500	2,632	1,999	2,146	420
	Broj čvorova	5	2	4	1	10	4.40	3.14
Niska stopa smrtnosti (0%) & Turnirska selekcija (k = 5)	Broj veza	110	104	107	100	112	106.60	4.27
	Rezultat	1,505	2,939	1,333	1,866	2,452	2,019	599
	Broj čvorova	8	3	6	7	8	6.40	1.85
	Broj veza	72	91	84	72	67	77.20	8.89

Tablica 5.2: Prikaz podataka o izvršavanju algoritma na problemu optimiranja strategije hodanja u simulaciji *HalfCheetah*

goritam [6], općenito ima visoku stopu smrtnosti u populaciji, toliku da se čak 80% populacije nema pravo razmnožavati. Na ovaj način pokušava se osigurati da se razmnožavaju samo što bolje jedinke. Ovo je dosta agresivan način kako bi se postigao ovaj učinak, stoga je ovdje predloženo korištenje turnirske selekcije. Kao što je rečeno, turnirska selekcija odabire malu količinu jedinki koje će se međusobno nadmetati. Tako se postiže isti rezultat, nagrađuje se bolje jedinke, ali odabir jedinki za turnirsku selekciju je nasumičan, stoga je moguće da pobjede i neke jedinke koje inače ne bi imale pravo na križanje. Kako bi pritisak na populaciju i dalje ostao velik, veličina turnira povećana je na 5, što znači da su manje šanse da će lošija jedinka pobijediti u turniru. Na tablici 5.2 je vidljivo da obje varijante postižu približno jednak prosječan rezultat, turnirska selekcija s prosječnim rezultatom od 2,019.09 u odnosu na nasumičnu s rezultatom 2,146.28. Turnirska selekcija stvara jedinke s malo većim brojem čvorova, prosječno 6.4 u odnosu na nasumičnu s 4.4, ali rješenja imaju manji broj veza, prosječno 77.2 u odnosu na 106.6. Budući da obje varijante zapinju u istom lokalnom optimumu teško je odrediti koja varijanta je bolja.

5.3. Optimiranje strategije igranja igre *LunarLander*

Ovaj problem najbliži je situaciji u kojoj će se algoritam pronaći prilikom stvaranja strategije igranja u većini računalnih igara. Postavke odabrane za ovo ispitivanje iste su kao postavke korištene za ispitivanje optimiranja strategije hodanja u simulaciji *HalfCheetah*. Metoda koja evaluira ovaj problem gotovo je ista kao i za prethodni problem, jedina razlika je ta što objekt *env* mora biti inicijaliziran s drugim ključem igre, to jest s ključem igre *LunarLander*. Jedno pokretanje ovog algoritma u brzini kojoj bi čovjek igrao traje u prosjeku jedan cijeli dan prije nego algoritam pronađe rješenje. Bitno je uzeti u obzir da je ovo bez paralelizacije evaluiranja jedinki. U tablici 5.3 mogu se vidjeti rezultati pet pokretanja algoritma. Algoritam pronalazi rješenje, ali mu je standardna devijacija broja generacija prije pronalaska rješenja dosta velika. U

		Pokretanje 1	Pokretanje 2	Pokretanje 3	Pokretanje 4	Pokretanje 5	Prosječno	Standardna devijacija
Najbolja jedinka	Broj čvorova	8	6	9	5	5	6.60	1.62
	Broj aktivnih veza	47	48	57	48	42	48.40	4.84
Broj generacija prije pronalaska rješenja		88	68	103	80	42	76.20	20.54

Tablica 5.3: Prikaz podataka o izvršavanju algoritma na problemu optimiranje strategije igranja u videoigri *LunarLander*

ostatku poglavlja prikazani su grafovi s detaljnijim podacima o izvođenju. Svi grafovi u ostatku poglavlja odgovaraju istom pokretanju algoritma.

5.3.1. Problem nasumičnog pokretanja

Jedan bitan aspekt ovog okruženja je što agent mora u sto generacija postići dobar rezultat kako bi se problem smatrao riješenim. Budući da je to skup proces, teži se izbjeći potrebu da se u svakoj generaciji svaku jedinku ispituje sto puta. Pogotovo zato što je prilično sigurno da jedinke koje postignu loš rezultat kroz sto pokretanja neće moći to lako ispraviti. Zbog toga se u svakoj generaciji predloži samo najbolju jedinku za rješenje problema. Tom logikom, ako je u populaciji N jedinki, ukupan broj evaluacija u toj generaciji bi umjesto $N * 100$ trebao bi biti $N + 100$. Sto različitih pokretanja koja su potrebna za utvrditi je li jedinka zadovoljavajuća izvršavaju se na kraju generacije i osim što su uvjet za zaustavljanje algoritma nisu uzete kod daljnje evaluacije jedinke.

Mana ovakvog pristupa je što se najbolja jedinka u generaciji treba jako pažljivo izabrati, kako bi izbjegli situaciju u kojoj jedinka koja je možda bolja u ovom pokretanju nije bolja od drugoplasirane u ostalih 99. Stoga je važna odluka bila koja početna stanja koristiti prilikom evaluacije jedinke. U ovom slučaju postoje tri mogućnosti. Prva, najjednostavnija, bi bila da je početno stanje kod svakog pokretanja nasumično. To bi značilo da bi svake dvije jedinke u istoj generaciji prilikom evaluacije počinjale iz različitih početnih pozicija. Druga opcija je da je početno stanje u svim generacijama i za sve jedinke isto. Dakle, jedinka bi se u svakoj generaciji ispitivala na istom pokušaju, ali bi se jedinka koja je najbolja u svakoj generaciji i dalje morala pokrenuti za sto nasumičnih početnih stanja. Treća, zadnja opcija, je da je početno stanje za cijelu generaciju isto, ali da se mijenja iz generacije u generaciju. Tako sve jedinke u generaciji bile bi ispitane na istom okruženju, ali bi se to okruženje mijenjalo u svakoj generaciji kako bi se jedinke ispitivale na raznolikim slučajevima.

Prva opcija nije uspjela naći rješenje u sto generacija. Zbog toga što se svaka jedinka ispituje na nasumičnom početnom stanju na kraju generacije gotovo je nemo-

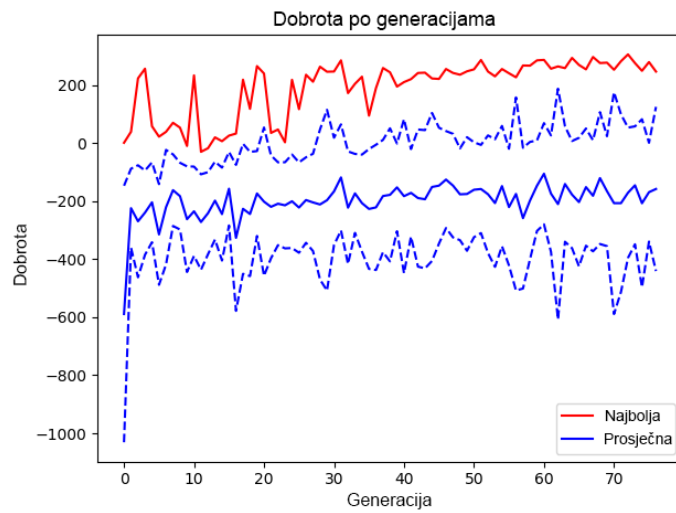
guće utvrditi koja jedinka je objektivno najbolja. U ovom načinu vrlo je vjerojatno da je jedinka koja je dobila najveći rezultat zapravo nasumično bila pokrenuta na vrlo lakom početnom stanju i bez puno znanja mogla je postići veći rezultat od neke koja je u prosjeku bolja od nje, ali je pokrenuta na vrlo teškom početnom stanju. Zbog ovoga populacija je vrlo slabo napredovala i čak u situacijama kada je postojalo potencijalno zadovoljavajuće rješenje u populaciji moguće je da je zaobiđeno zbog jedinke koja je nasumično postigla dobar rezultat.

Prilikom pokretanja druge opcije nije dobiven puno bolji rezultat. Algoritam također nije mogao naći rješenje unutar sto generacija. U ovom slučaju funkcija najbolje dobrote je za razliku od prve opcije monotono rasla, zato što je populacija kroz generacije zapravo učila postati ekspert za to određeno početno stanje. Štoviše, populacija je naučila odlično riješiti to specifično okruženje, ali zbog manjka varijacije u ispitivanju jedinke jednostavno nisu bile dovoljno robusne, tj. nisu se naučile prilagođavati različitim početnim situacijama.

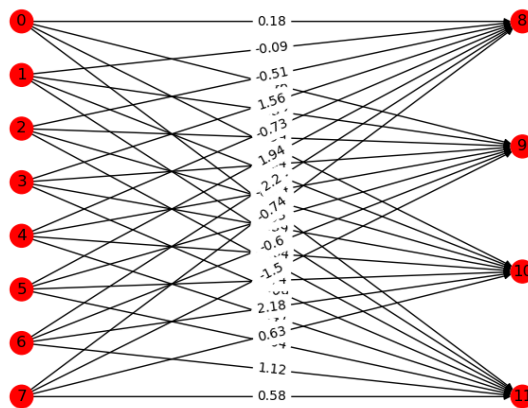
Zadnja opcija je kombinacija prve dvije. Ispitivanje svake generacije na istom početnom stanju omogućilo je puno veću sigurnost odabira najbolje jedinke u generaciji. Ovaj problem nije u potpunosti riješen, ali ako se ispituje sve jedinke na istom slučaju veća je vjerojatnost da će ona s najboljim rezultatom biti bolja i u ostalih 99. Osim toga, odabir novog početnog stanja za svaku generaciju uvelo je varijaciju u algoritam. Ispitivanje jedinki na različitim početnim stanjima stvara puno robusnije jedinke. Algoritam koji je pokrenut s ovom postavkom u jednom pokretanju pronašao je rješenje u malo 78 generacija. Funkcija najbolje dobrote prikazana je crvenom bojom na slici 5.1. Ona zbog ove opcije više nije monotona, ali u prosjeku se može vidjeti da funkcija ipak raste. Zanimljivo je primijetiti da u početku funkcija puno više varira, zbog toga što su neka početna stanja teža, a neka lakša. Kasnije, kada jedinke postaju robusnije, funkcija ima puno manju standardnu devijaciju. U sljedećem potpoglavlju prikazano je još nekoliko grafova s informacijama dobivenim iz istog pokretanja.

5.3.2. Ostali rezultati

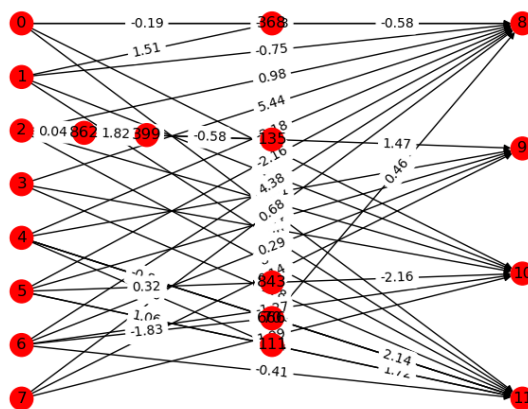
Na slikama su prikazane struktura najbolje jedinke u prvoj generaciji (Slika 5.2) i struktura jedinke koja je na kraju riješila problem (Slika 5.3). Po ovim slikama vidi se kako se kroz generacije struktura povećava. Na jednoj vezi najbolje jedinke (Slika 5.3) vide se tri čvora u koje se ne uključuje nijedna druga veza. Bilo bi idealno naći način da se ovakve redundancije izbjegnu. Rast broja čvorova i veza može se vidjeti na slici 5.4. Može se primijetiti da iako su im vjerojatnosti za stvaranje različite, ove dvije brojke



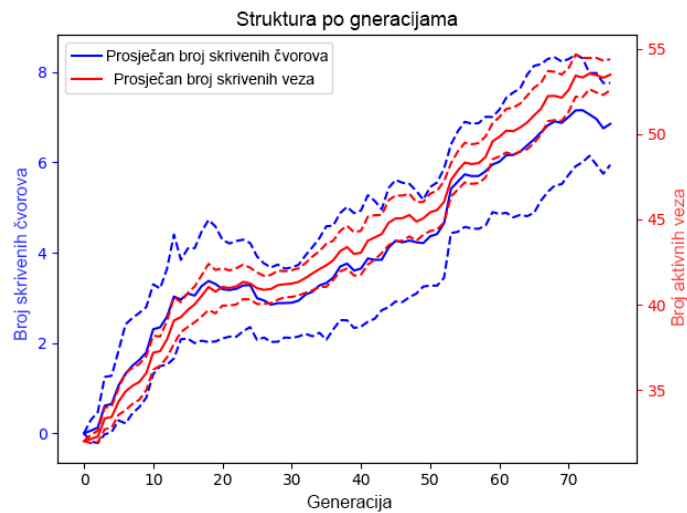
Slika 5.1: Funkcija najbolje i prosječne dobrote po generacijama



Slika 5.2: Prikaz neuronske mreže najbolje jedinice u prvoj generaciji



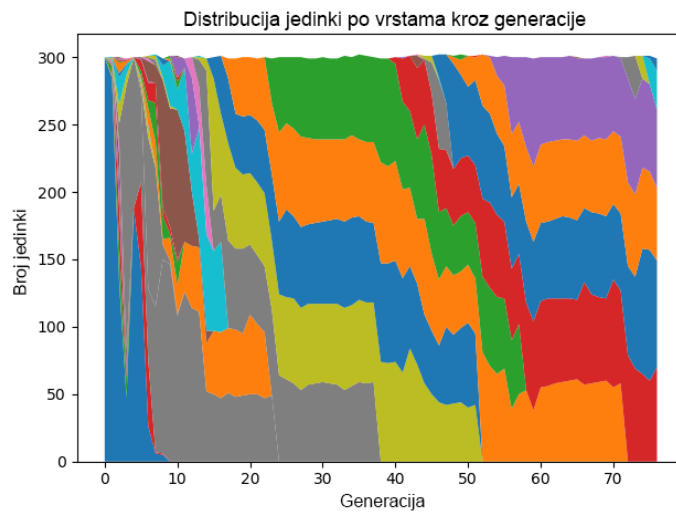
Slika 5.3: Prikaz neuronske mreže najbolje jedinice koja je riješila problem



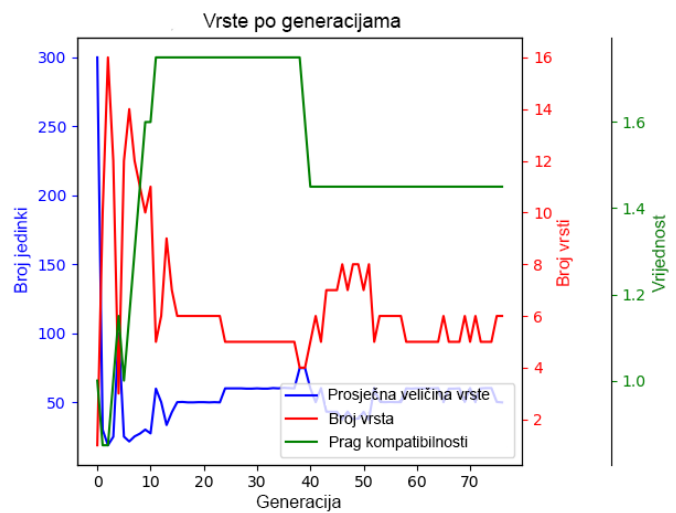
Slika 5.4: Prikaz rasta broja čvorova i veza kroz generacije

su korelirane zato što bez dovoljno čvorova nije moguće stvarati ni dodatne veze.

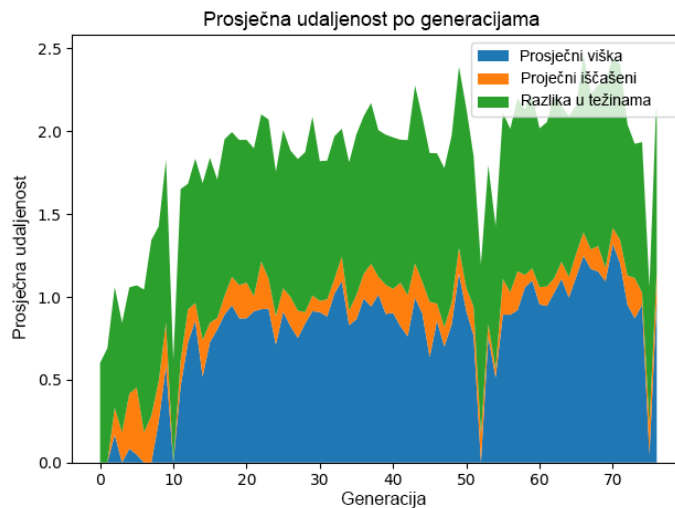
Kako bi postepeni napredak strukture jedinki bio bolje vidljiv, najbolje je pogledati udio vrsta kroz generacije prikazan na slici 5.5. Može se vidjeti kako veće vrste rastu te kroz dvadesetak generacija postepeno umiru, a kako one umiru postepeno se stvaraju nove vrste čime se ciklus nastavlja. Ako se uspoređi s grafom na slici 5.4 vidljivo je kako na početku, kada je struktura još jednostavna, postoji samo jedna vrsta. Nakon inicijalnih mutacija broj vrsta jako poraste, zbog čega se smanji prosječni broj jedinki po vrsti, a složenost strukture poraste. Razlog iz kojeg se ovo događa je kompenzacija praga kompatibilnosti kojeg se može vidjeti na slici 5.6. Prag kompatibilnosti definira koliko daleko jedinka može biti od predstavnika vrste da bi se smatrala dijelom te vrste. Zbog toga, ako je prag kompatibilnosti velik, vrste će obuhvaćati raznolikije jedinke. Stoga, ako je prag kompatibilnosti velik i vrste će biti veće te će broj vrsta biti manji. Problem eksplozije broja vrsta nastaje zato što na početku, kada algoritam još nema vremena stvoriti dovoljno raznolike jedinke, prag kompatibilnosti pokušava to kompenzirati tako da se smanjuje. Jednom kada počinju nastajati promjene prag kompatibilnosti je već mali te nastaje puno vrsta. Nakon što se prođe željeni prag broja vrsta prag se ponovo povećava te se i broj vrsta stabilizira. Na slici 5.7 se vidi kako prosječna udaljenost jedinki u početku brzo raste, ali se onda donekle stabilizira. Iako je potrebno bolje definirati način na koji se prag kompatibilnosti podešava, reguliranje broja vrsta na ovaj način pokazalo je bolje rezultate u problemu konstrukcije XOR vrata.



Slika 5.5: Vizualni prikaz udjela vrsta u populaciji kroz generacije



Slika 5.6: Korelacija broja vrsti i prosječne veličine vrste s pragom kompatibilnosti



Slika 5.7: Korelacija broja vrsti i prosječne veličine vrste s pragom kompatibilnosti

5.3.3. Izlazak iz lokalnog optimuma

Veliki izazov odabira postavki je znati koje omogućavaju konvergenciju globalnom, a ne samo lokalnom optimumu. U nekoliko pokretanja ovog problema agenti su zapinjali u istom lokalnom optimumu. U pravilima okruženja, agent u svakom trenutku u kojem nogom dodiruje tlo dobiva 10 bodova. Događalo se da su agenti naučili mirno sletjeti na tlo, ali umjesto da isključe sve motore i prime veliku nagradu kontinuirano su palili lijeve i desne motore kako bi se nastavili gibati dodirujući tlo i time primati malu količinu bodova. Ovime su zapeli u lokalnom optimumu jer su u tom trenutku to bili najbolji rezultati, ali im je trebalo dosta generacija da bi naučili da je nagrada veća ako se što prije zaustave na tlu.

6. Zaključak

U ovom radu prikazana je povezanost između videoigara i razvoja umjetne inteligencije. Opisane su karakteristike neuroevolucijskog algoritma promjenjivih topologija, algoritma neuroevolucije koji, koristeći genetske algoritme, optimizira težine i razvija strukturu umjetnih neuronskih mreža. Na ovaj način mogu se konstruirati strategije igranja igara te je algoritam ispitan na tri različita problema. Problemi su ispitani implementacijom algoritma koja je ostvarena u okviru ovog rada. Ostvarenje je rađeno po uzoru originalne implementacije iz rada u kojem je algoritam predložen [6] uz promjene koje su predložene u okviru ovog rada. Iako je kroz dinamičku optimizaciju praga kompatibilnosti i uvođenje pristranosti u genom pokazan napredak u odnosu na originalnu implementaciju [6], algoritam još nije konkurentan modernim metodama potpornog učenja. Neke ideje koje bi se isplatilo razmotriti kako bi se poboljšao rad algoritma su paralelizacija, čuvanje liste inovacija kroz više generacija te manju mutaciju veza i pristranosti što su veze i čvorovi stariji.

Budući da je velika prednost ovog algoritma lakoća implementacije paralelizacije uvelike bi se ubrzalo vrijeme izvođenja kada bi se evaluacije izvodile paralelno. Ovo je jednostavno dodati u predloženu implementaciju i sve što je potrebno je napraviti metodu koja bi se brinula za distribuiranje evaluacija po različitim dretvama i za prikupljanje rezultata evaluacija. Najveći problem je naći ili napraviti okruženje koje bi podržavalo lako paralelno izvođenje. Okruženja koja moraju stvarati prikaz na ekranu često nisu zamišljena da ih se nanovo stvara prilikom svake evaluacije.

Čuvanje lista inovacija tijekom cijelog izvođenja omogućilo bi ne samo da se prepoznaju iste mutacije unutar jedne generacije, nego da se iste strukturalne promjene prepoznaju u svakom trenutku. Ovo bi moglo pomoći jedinkama da se bolje usklade prilikom križanja te da udaljenost između jedinki bude konzistentnija.

Na kraju, manja mutacija starih veza i čvorova pomogla bi kod većih strukturalnih promjena da se algoritam fokusira samo na novonastale veze i čvorove. Naprimjer, veza koja je nastala u prvoj generaciji prošla je puno više iteracija od veze koja je nastala u pedesetoj generaciji. Zbog toga se može reći da su u tom trenutku veće

vjerojatnosti da je prva veza bolje prilagođena od nove. U tom slučaju nema potrebe da se obje veze jednako mutiraju, već je moguće napraviti sustav u kojem je korak perturbacije manji što je veza starija.

Zaključno s time, neuroevolucija promjenjivih topologija ovime je pokazala da može konstruirati dovoljno djelotvorna rješenja.

LITERATURA

- [1] Alex Krizhevsky, Ilya Sutskever, i Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, Svibanj 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL <http://doi.acm.org/10.1145/3065386>.
- [2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, i Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL <http://arxiv.org/abs/1312.5602>.
- [3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, i Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL <http://arxiv.org/abs/1602.01783>.
- [4] OpenAI. Openai five. <https://openai.com/five/>, 2017. Na dan: 3.2.2019.
- [5] Tim Salimans, Jonathan Ho, Xi Chen, i Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017. URL <https://arxiv.org/abs/1703.03864>.
- [6] Kenneth O. Stanley i Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL <http://nn.cs.utexas.edu/?stanley:ec02>.
- [7] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob

Repp, i Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017. URL <http://arxiv.org/abs/1708.04782>.

[8] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9): 1423–1447, Sep. 1999. ISSN 0018-9219. doi: 10.1109/5.784219. URL <https://ieeexplore.ieee.org/document/784219>.

[9] Jacky Shunjie Zhen i Ian Watson. Neuroevolution for micromanagement in the real-time strategy game starcraft: Brood war. U *Proceedings of the 26th Australasian Joint Conference on AI 2013: Advances in Artificial Intelligence - Volume 8272*, stranice 259–270, New York, NY, USA, 2013. Springer-Verlag New York, Inc. ISBN 978-3-319-03679-3. doi: 10.1007/978-3-319-03680-9_28. URL http://dx.doi.org/10.1007/978-3-319-03680-9_28.

Primjena neuroevolucijskih algoritama promjenjivih topologija u igranju računalnih igara

Sažetak

U radu je opisan razvoj umjetne inteligencije na primjerima računalnih igara. Proučeno je korištenje genetskih algoritama za razvijanje težina i topologije umjetnih neuronskih mreža. Naučene mreže koriste se za stvaranje strategije igranja jednostavnih videoigara. Ostvaren je algoritam u okviru rada koji je onda ispitan na jednostavnim problemima. Analizirana je djelotvornost sustava i razmotreni su sljedeći mogući koraci za unaprjeđenje algoritma.

Ključne riječi: neuroevolucija promjenjivih topologija, genetski algoritmi, neuroevolucija, umjetne neuronske mreže

Application of neuroevolution of augmenting topologies in playing computer games

Abstract

In this paper we describe the influence of computer games on the field of artificial intelligence. We cover application of genetic algorithms on optimizing weights and developing structure of artificial neural networks. Trained networks are used to define playing strategies for simple videogames. We implemented the algorithm within the scope of this paper which is then tested on simple problems. We then analyzed it's efficiency and proposed some further improvements.

Keywords: neuroevolution of augmenting topologies, genetic algorithms, neuroevolution, artificial neural networks