Evolucijsko optimiranje neuronske mreže mobilnog robota

Dragičević, Nino

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture / Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:235:718120

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-27**

Repository / Repozitorij:

Repository of Faculty of Mechanical Engineering and Naval Architecture University of Zagreb

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Nino Dragičević

Zagreb, 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentori: Student:

Izv.prof. dr. sc. Petar Ćurković, dipl. ing. Nino Dragičević

Zagreb, 2022.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se mentoru izv. prof. dr. sc. Petru Ćurkoviću za dostupnost, strpljenje i razumijevanje tokom izrade završnog rada.

Naposlijetku, zahvaljujem se svojoj obitelji, posebice sestri, na podršci i pruženoj pomoći prilikom pisanja rada te svojoj djevojci i prijateljima za moralnu podršku tijekom studija.

Nino Dragičević

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za završne i diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 – 04 / 22 – 6 / 1**	
Ur.broj: **15 - 1703 - 22 -**	

ZAVRŠNI ZADATAK

Student: **Nino Dragičević** JMBAG: **003535220251**

Naslov rada na hrvatskom jeziku: **Evolucijsko optimiranje neuronske mreže mobilnog robota**

Naslov rada na engleskom jeziku: **Evolutionary optimization of a mobile robot neural network**

Opis zadatka:

Evolucijski algoritmi u kombinaciji s metodama strojnog učenja mogu se koristiti za razvoj kontrolera različitih tehničkih sustava. Pri tome predmet optimiranja evolucijskim algoritmom mogu biti težine sinapsi, ali i topologija mreže i vrsta aktivacijskih funkcija. Prednost ovakvog pristupa vidi se kod složenih upravljačkih problema za koje je teško unaprijed odabrati optimalne parametre umjetne neuronske mreže.

U ovom radu potrebno je napraviti sljedeće:

- upoznati se s područjem evolucijske robotike
- u fizičkom simulatoru implementirati 2D model mobilnog robota
- razviti evolucijski algoritam kojim se optimira rad neuronske mreže koja upravlja mobilnim robotom

Model robota treba se temeljiti na Braitenbergovu pristupu. Potrebno je osigurati da robot nauči slijediti liniju i pratiti izvor svjetla.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:	Datum predaje rada:	Predviđeni datumi obrane:
30. 11. 2021.	**1. rok:** 24. 2. 2022. **2. rok (izvanredni):** 6. 7. 2022. **3. rok:** 22. 9. 2022.	**1. rok:** 28. 2. – 4. 3. 2022. **2. rok (izvanredni):** 8. 7. 2022. **3. rok:** 26. 9. – 30. 9. 2022.

Zadatak zadao: Predsjednik Povjerenstva:

Doc. dr. sc. Petar Ćurković Prof. dr. sc. Branko Bauer

SADRŽAJ

SADRŽAJ II

POPIS SLIKA IV

POPIS TABLICA VI

SAŽETAK VIII

SUMMARY IX

1. UVOD 1

2. BRAITENBERGOVO VOZILO 2

3. EVOLUCIJSKI ALGORITMI 5

4. NEURONSKE MREŽE 8

5. NEUROEVOLUCIJA PROMJENJIVIH TOPOLOGIJA (engl. *NeuroEvolution of Augmenting Topologies*) 12

5.1. Topology and Weight Evolving Artificial Neural Networks (TWEANN) 13
5.1.1. Kodiranje TWEANN 13
5.1.2. Učestali problemi TWEANN 14
5.2. Kako funkcionira NEAT algoritam 16
5.2.1. Kodiranje u NEAT 16
5.2.2. Križanje u NEAT 17
5.2.3. Očuvanje inovacije kroz specijaciju 18
5.2.4. Minimizacija dimenzionalnosti mreže u NEAT 19

6. IMPLEMENTACIJA NEAT ALGORITMA 20

6.1. Problem slijeđenja ravne linije 21
6.2. Problem praćenja svjetlosti 24

7. REZULTATI 29

7.1. Rezultati učenja slijeđenja ravne linije 29
7.2. Rezultati učenja praćenja svjetlosti 34
7.2.1. Mnogokutna trajektorija 34
7.2.2. Trajektorija gore-dolje 36
7.2.3. Pravokutna trajektorija 38

8. ZAKLJUČAK 41

LITERATURA 42

PRILOG 1: PYTHON KOD ZA PRAĆENJE RAVNE LINIJE 44

PRILOG 2: *CONFIG* TEKSTUALNA DATOTEKA ZA 1. PROBLEM 49

PRILOG 3: PYTHON KOD ZA SLIJEĐENJE SVJETLA 51

PRILOG 4: *CONFIG* TEKSTUALNA DATOTEKA ZA 2. PROBLEM 57

POPIS SLIKA

Slika 1. Braitenbergova vozila ... 2
Slika 2. Ponašanja Braitenbergovih vozila ... 3
Slika 3. Fenotip i genotip ... 5
Slika 4. Selekcija ruletnim pravilom ... 6
Slika 5. Rekombinacija ... 6
Slika 6. Mutacija ... 7
Slika 7. Biološki neuron ... 8
Slika 8. Prikaz umjetnog neurona ... 9
Slika 9. Struktura tipične neuronske mreže ... 9
Slika 10. Aktivacijske funckije ... 10
Slika 11. *Competing conventions problem* ... 15
Slika 12. Mutacije u NEAT-u ... 16
Slika 13. Primjer križanja u NEAT-u ... 17
Slika 14. Primjer selekcije u NEAT-u ... 19
Slika 15. Okoliš za problem slijeđenja ravne linije ... 21
Slika 16. Prikaz jedinke (mobilnog robota) ... 22
Slika 17. Fitnes funkcija za problem slijeđenja ravne linije ... 23
Slika 18. Inicijalizacija populacije ... 23
Slika 19. Ponašanje populacije nulte generacije ... 24
Slika 20. Okoliš za problem praćenja snopa svijetlosti ... 24
Slika 21. Fitnes funkcija za problem praćenja snopa svijetlosti ... 25
Slika 22. Izgled prve trajektorije za problem praćenja svijetlosti ... 26
Slika 23. Izgled druge trajektorije za problem praćenja svjetlosti ... 26
Slika 24. Izgled treće trajektorije za problem praćenja svjetlosti ... 27
Slika 25. Inicijalizacija programa za problem praćenja svjetlosti ... 27
Slika 26. Praćenje svjetlosti kod trajektorije gore-dolje ... 28
Slika 27. Prikaz maksimalne i prosječne ocjene fitnesa populacije kroz generacije, problem slijeđenja ravne linije (5. pokretanje) ... 30
Slika 28. Prikaz specijacije, problem slijeđenja ravne linije (5. pokretanje) ... 31
Slika 29. Najčešća topologija mreže u problemu slijeđenja ravne linije ... 32
Slika 30. Topologija mreže 3. pokretanja, problem slijeđenja ravne linije ... 32
Slika 31. Topologija mreže 4. pokretanja, problem slijeđenja ravne linije ... 33

Slika 32. Mutacija mreže kod mnogokutne trajektorije (1. pokretanje) 35
Slika 33. Prikaz fitnesa kroz generacije, mnogokutna trajektorija (4. pokretanje) 35
Slika 34. Prikaz fitnesa kroz generacije, trajektorija gore-dolje (2. pokretanje) 37
Slika 35. Izgled mreže, trajektorija gore-dolje (2. pokretanje) 37
Slika 36. Mutacija mreže, trajektorija gore-dolje (1. pokretanje) 38
Slika 37. Graf fitnesa kroz generacije, pravokutna trajektorija, 3. pokretanje 39
Slika 38. Mreža najbolje jedinke, pravokutna trajektorija, 3. pokretanje 39
Slika 39. Najčešća mutacija mreže kod pravokutne trajektorije 40

POPIS TABLICA

Tablica 1. Rezultati učenja slijeđenja ravne linije 29
Tablica 2. Rezultati učenja praćenja snopa svijetlosti, mnogokutna trajektorija 34
Tablica 3. Rezultati učenja praćenja snopa svijetlosti, trajektorija gore – dolje 36
Tablica 4. Rezultati učenja praćenja snopa svijetlosti, pravokutna trajektorija 38

POPIS OZNAKA

Oznaka	Jedinica	Opis
b_j	-	Vrijednost *biasa* j-tog sloja neuronske mreže
c_1	-	1. koeficijent podešavanja genetske udaljenosti
c_2	-	2. koeficijent podešavanja genetske udaljenosti
c_3	-	3. koeficijent podešavanja genetske udaljenosti
D	-	Broj nesparenih gena
d_n	-	Tražena vrijednost izlaza neuronske mreže
E	-	Broj suvišnih gena
E_f	-	Funkcija cilja neuronske mreže
f_i	-	Ocjena fitnesa i-te jedinke
J	-	Broj slojeva neuronske mreže
K	-	Broj neurona izlaznog sloja neuronske mreže
N	-	Broj jedinka u populaciji
N_g	-	Broj gena veće jedinke u specijaciji
net_{Ok}	-	Vrijednost k-tog izlaza neuronske mreže
O_n	-	Dobivena vrijednost izlaza neuronske mreže
p_i	-	Vjerojatnost odabira i-te jedinke za roditelja iduće generacije
$\overline{W}$	-	Razlika težina uparenih gena
w_{ij}	-	Vrijednost težine pripadajuće veze
y_j	-	Vrijednost aktivacijskog broja pripadajućeg neurona
δ	-	Genetska udaljenost jedinke
δ_i	-	I-ta genetska udaljenost jedinki
δ_z	-	Zadani parametar genetske udaljenosti (*compatibility threshold)*
σ	-	Sigmoidna aktivacijska funkcija mreže

SAŽETAK

Razvoj umjetne inteligencije omogućava otkrivanje novih metoda za rješavanje dosadašnjih optimizacijskih problema. Primjena evolucijskih mehanizama u razvoju novih rješenja za razne probleme optimizacije oduvijek je bila zanimljiva tema kojom se bave i robotika i računarstvo.

Cilj rada bio je pobliže proučiti područje evolucijske robotike te implementirati evolucijski algoritam koji optimira rad neuronske mreže u virtualnom okruženju na problemima praćenja ravne linije i slijeđenja izvora svjetlosti.

Ovaj rad ispitao je kako se evolucijskim algoritmom evoluira neuronska mreža, koja služi za upravljanje Braitenbergovog modela mobilnog robota. Objašnjeni su glavni mehanizmi evolucije i princip rada neuronskih mreža. Izvršena je implementacija pomoću *Neuro Evolution of Augmenting Topologies* (NEAT) algoritma u programskom jeziku *python*, koji je korišten za rješavanje problema slijeđenja ravne linije i praćenja izvora svjetlosti.

Ključne riječi: evolucijska robotika, implementacija, mobilni robot, NEAT, *python*

SUMMARY

The development of artificial intelligence enables the discovery of new methods for solving existing problems. The application of evolutionary mechanisms in the development of new solutions to various problems has always been an interesting topic to both robotics and computer science.

The aim of this study was to examine the field of evolutionary robotics in more detail, and to implement an evolutionary algorithm that optimizes the operation of a neural network in a virtual environment on the problems of tracking a straight line, and following a light source.

This study investigated how the evolutionary algorithm evolves a neural network which is used to control Braitenberg's model of a mobile robot. The main mechanisms of evolution and the working principle of neural networks were explained. Implementation was performed using the *Neuro Evolution of Augmenting Topologies* (NEAT) algorithm in the *python* programming language which was used to solve the problems of following a straight line and tracking a light source.

Key words: evolutionary robotics, implementation, mobile robot, NEAT, *python*

1. UVOD

Evolucijski algoritmi u kombinaciji s metodama strojnog učenja mogu se koristiti za razvoj kontrolera različitih tehničkih sustava. Pri tome predmet optimiranja evolucijskim algoritmom mogu biti težine sinapsi, ali i topologija mreže i vrsta aktivacijskih funkcija. Prednost ovakvog pristupa vidi se kod složenih upravljačkih problema za koje je teško unaprijed odabrati optimalne parametre umjetne neuronske mreže. Razvojem tehnologije i povećanjem procesorske moći u računalima otvara se novi način za rješavanje složenih problema.

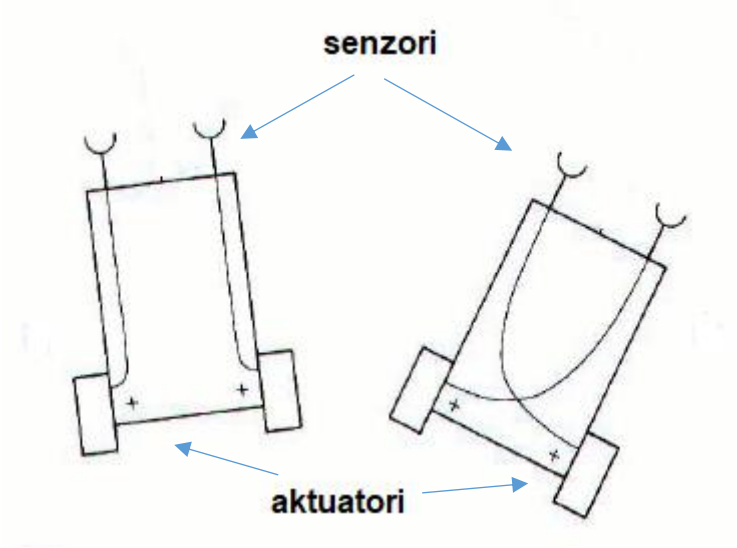
U ovom radu pobliže će se proučiti evolucija neuronske mreže koja kontrolira ponašanje mobilnog robota pomoću NEAT algoritma i procesi koji se događaju tokom same evolucije te će se ovakav pristup implementirati na sljedećim problemima:

1. Slijeđenje ravne linije
2. Praćenje izvora svjetlosti

Model mobilnog robota temeljit će se na Braitenbergovom pristupu.

2. BRAITENBERGOVO VOZILO

Cijeli koncept Braitenbergova vozila proizašao je od kibernetičara Valentina Braitenberga, koji je zamišljao ponašanje i reakcije najjednostavnijih vrsta vozila, sačinjenih od svega dva senzora i dva aktuatora (Slika 1). Ovi senzori mjere utjecaje okoline Braitenbergova vozila te proslijeđuju svoja očitanja aktuatorima (npr. motorima). Vanjski utjecaji koje senzori mjere mogu biti udaljenost od prepreke (npr. ultrazvučni senzor), količina svjetlosti (npr. fotodioda), tlak, temperatura, pa čak i nešto napredniji elektrokemijski senzori za lociranje plina [1]. Aktuatori također mogu biti raznih izvedbi, npr. motor koji pokreće auto, propeler koji pokreće brod ili zvučnik koji stvara zvuk. Senzori Braitenbergovih vozila su direktno povezani s aktuatorima, što znači da će za veće očitanje senzora aktuator imati veći izlaz pa će veza između aktuatora i senzora biti proporcionalna. Na direktnu vezu moguće je staviti i inverter (-) pa će ta veza biti obrnuto propocionalna, tj. za veće očitanje senzora aktuator će imati manji izlaz.



Slika 1. Braitenbergova vozila [2]

Ponašanja koja su Braitenbergova vozila razvila u njegovom eksperimentu su sljedeća [3]:

1. Strah
2. Agresija
3. Ljubav
4. Znatiželja

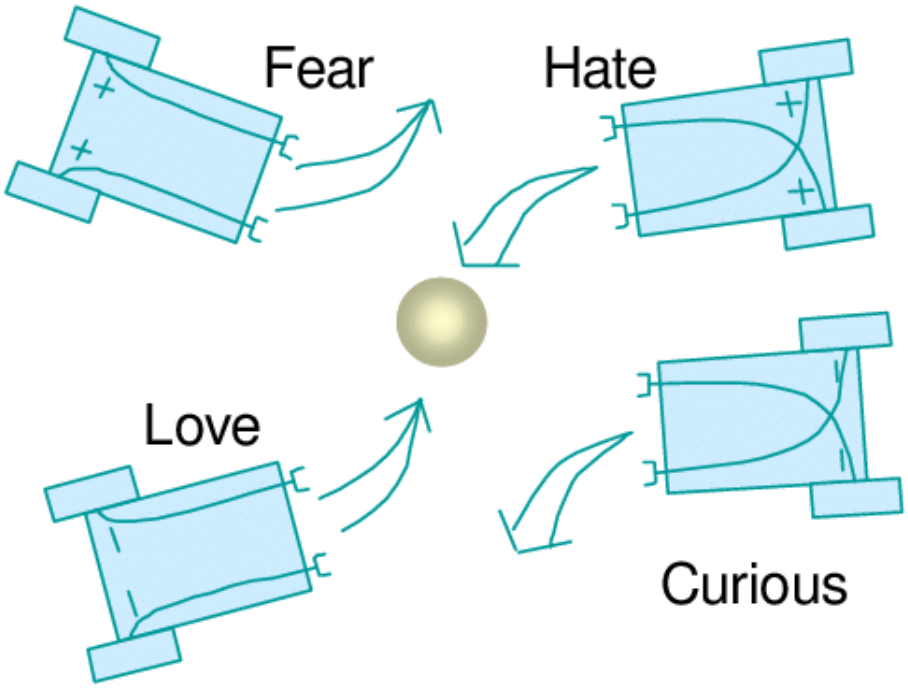
Sva ova ponašanja proizlaze iz različitih spojeva senzora s aktuatorima (Slika 2). Ako se uzme primjer vozila koje za senzore ima fotodiode, a za aktuatore motore koji pokreću kotače, vozilo s direktnom neinvertiranom vezom senzora i aktuatora će pokazati strah.

Što se vozilo više približava izvoru svjetla, kretat će se sve brže, no ako se dogodi da senzor s jedne strane očita više svjetla nego senzor s druge strane, vozilo će skrenuti u suprotnu stranu od izvora svjetlosti, kretajući se najbrže pokraj izvora svjetlosti, pokazujući strah.

Vozilo sa direktnom neinvertiranom vezom senzora i aktuatora, ali invertiranim stranama, pokazat će agresiju. Što se vozilo više približava izvoru svjetla, kretat će se sve brže te će razviti maksimalnu brzinu prolazeći kroz izvor svjetlosti. Ovakvo ponašanje izgleda kao agresija prema svjetlu.

Vozilo s direktnom invertiranom vezom senzora i aktuatora, pokazat će ljubav. Što se vozilo više približava izvoru svjetla, kretat će se sve sporije te će stati prolazeći kroz izvor svjetlosti. Ovakvo vozilo svojim ponašanjem pokazuje afinitet prema svjetlu.

Znatiželju će pokazati vozilo s direktnom invertiranom vezom senzora i aktuatora, ali invertiranim stranama. Što se vozilo više približava izvoru svjetla, kretat će se sve sporije, no ako se dogodi da senzor s jedne strane očita više svjetla nego drugi senzor, vozilo će skrenuti u suprotnu stranu od izvora svjetlosti, ubrzavajući kako se udaljuje od izvora. Takvo vozilo sporo će proći pokraj izvora svjetlosti, pokazujući znatiželju.



Slika 2. Ponašanja Braitenbergovih vozila [4]

Braitenbergova vozila su najjednostavnija vrsta autonomnih agensa. Oni su autonomni (tj. njihovo ponašanje u realnom svijetu odvija se bez ikakvog ljudskog upravljanja) i samodostatni

(tj. mogu dugo vremena održavati svoju razinu energije) sustavi koji razvijaju ponašanje u svojoj okolini. Kako bi agens uistinu bio samodostatan, potrebno ga je nagraditi za dobro ponašanje, tj. za ono ponašanje koje je nama povoljno. Takva vrsta učenja naziva se podržano učenje. Ponašanje takvih vozila razvija se pomoću raznih parametara, koji se najčešće određuju pomoću metode pokušaj-promašaj.

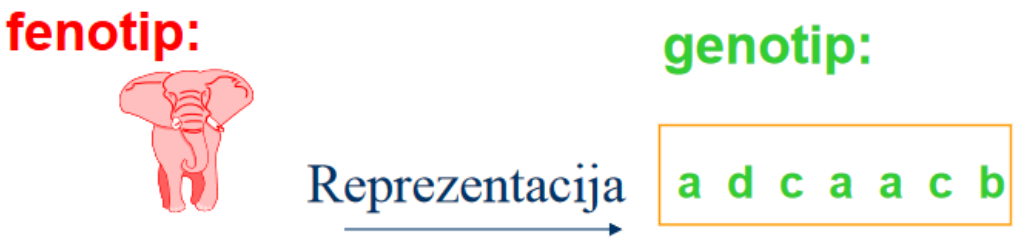
3. EVOLUCIJSKI ALGORITMI

Evolucijski algoritmi su heuristički optimizacijski algoritmi koji se temelje na kreiranju populacija i korištenju metoda biološke evolucije za dolazak do prihvatljivog rješenja. Koriste se za rješavanje raznih problema heurističke prirode, kao što su optimiranje konstrukcija raznih rešetkastih nosača [5], automatizacija dijelova konstruiranja zrakoplova te analiza satelitskih snimaka [6]. Nadahnuće za razvoj ovakvih vrsta algoritama dala je teorija prirodne evolucije Charlesa Darwina.

Postoje tri glavna mehanizma koja se provode u evoluciji:

1. Selekcija (engl. *selection*)
2. Rekombinacija (engl. *cross – over*)
3. Mutacija (engl. *mutation*)

Genetički algoritam (GA) je najopsežnija podvrsta evolucijskih algoritama te se u njemu odvijaju sva tri glavna mehanizma evolucije. Pri inicijalizaciji algoritma, jedinke (koje su potencijalna rješenja) se kreiraju u fenotipskom prostoru. Svakoj jedinki pridodaje se kromosom, koji pripada genotipskom prostoru, nad kojim će se kasnije vršiti selekcija, rekombinacija i mutacija. Obično su ovi kromosomi u GA kodirani kao binarne vrijednosti, tj. nizovi jedinica i nula, kako bi se nad njima lakše mogli vršiti mehanizmi evolucije. Svaka jedinka reprezentira vlastiti genotip u fenotipskom okruženju (Slika 3).



Slika 3. Fenotip i genotip [5]

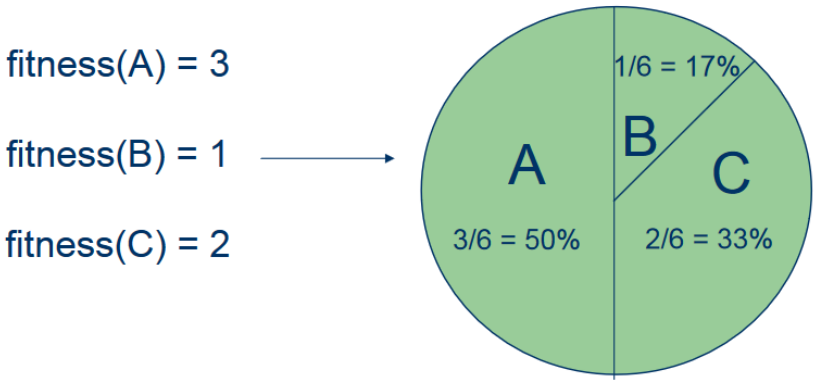
Nakon inicijalizacije jedinki, koja se najčešće događa nasumično, u fenotipskom prostoru odvija se borba za preživljavanje. Fitnes funkcija (ili funkcija dobrote) daje ocjenu svake pojedine jedinke na temelju njenog ponašanja u trenutnoj generaciji.

Jedinke koje su stekle najveću ocjenu fitnesa, tj. one jedinke čije je ponašanje najpovoljnije za rješavanje zadanog problema imaju priliku biti izabrane kao roditelji nove generacije. Ovaj proces naziva se selekcija. Selekcija se može odvijati ruletnim pravilom ili turnirskim pravilom.

Ruletno pravilo (Slika 4) je pravilo u kojem je ocjena fitnesa jedinke direktno proporcionalna šansi odabira te iste jedinke kao roditelja iduće generacije [7]. Ovo je stohastička metoda odabira roditelja, jer uvijek postoji mogućnost odabira jedinke s manjom ocjenom fitnesa.

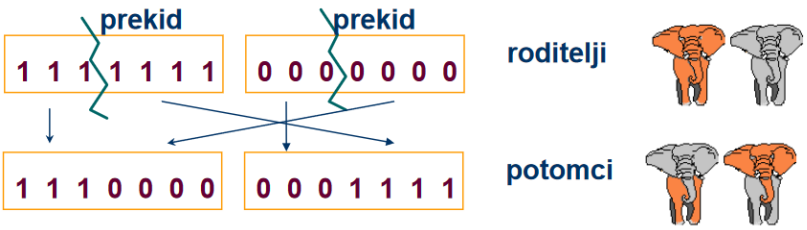
$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{1}$$

Turnirsko pravilo nasumično odabire dio generacije te uspoređuje ocjene fitnesa odabranih jedinki. Jedinke sa najvećom ocjenom fitnesa odabrane su kao roditelji iduće generacije. Broj nasumično odabrane generacije za trenutni turnir (tj. veličina turnira), kao i broj jedinki sa najvećom ocjenom fitnesa, koje se odabiru za roditelje iduće generacije, unaprijed su određeni. Ova metoda je deterministička, u svim slučajevima osim kad je veličina turnira $p = 1$, jer je za roditelja uvijek odabrana jedinka s većom ocjenom fitnesa.



Slika 4. Selekcija ruletnim pravilom [5]

Rekombinacijom (Slika 5) se spajaju informacije genotipa roditelja u potomke. Uloga rekombinacije je kombinirati obilježja iz različitih izvora. Ona ne daje nužno bolje rješenje od rješenja prethodne generacije, ali služi kao varijacijski operator. Većom varijacijom potencijalnih rješenja dobiva se veća mogućnost pronalaska onog optimalnog.



Slika 5. Rekombinacija [5]

Kao drugi varijacijski faktor služi mutacija (Slika 6). Ona uzrokuje malu, slučajnu promjenu u genotipu i stvara novi genotip. Ovaj proces je potpuno stohastičan te daje veću raznovrsnost populacije.

Slika 6. Mutacija [5]

Naposlijetku, vrši se izbor jedinki koje će biti u idućoj generaciji. Najčešće je to deterministički proces u kojem ili svi potomci zamijene roditelje, ili se roditelji i potomci rangiraju prema ocjeni fitnesa te najbolji prolaze u iduću generaciju. K tome, elitizam jedinke omogućava roditelju koji je imao najveću ocjenu fitnesa u prethodnoj generaciji da uvijek postane član iduće generacije.

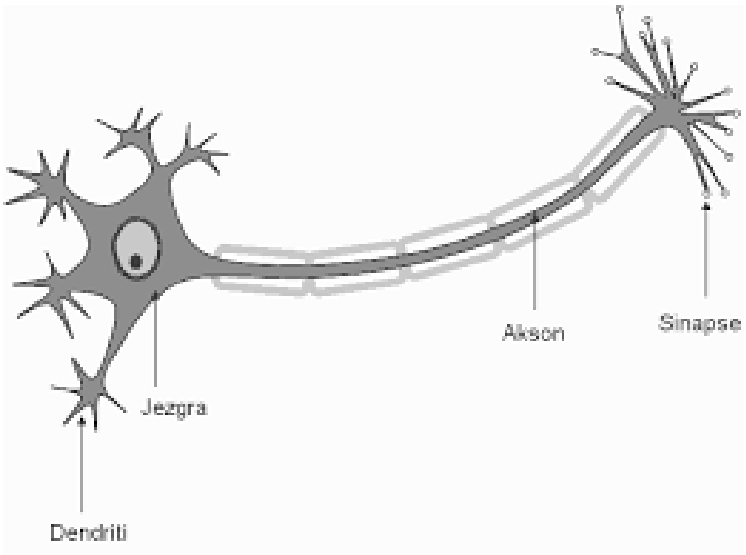
Tok generičkog genetičkog algoritma izgleda ovako [6]:

Ponavljaj zadani broj generacija:

1. Kreiraj nasumičnu inicijalnu populaciju jedinki koje su moguće rješenje.
2. Izračunaj fitnes svake jedinke u trenutnoj populaciji
3. Odaberi određeni broj jedinki s najvećom ocjenom fitnesa. Oni su roditelji sljedeće generacije.
4. Upari roditelje. Svaki par rekombinacijom generira nove jedinke koje ulaze u iduću generaciju, uz šansu nasumične mutacije. Ovaj proces se ponavlja dok iduća generacija nije popunjena. Sada iduća generacija postaje trenutna generacija.
5. Vrati se na 2. korak.

4. NEURONSKE MREŽE

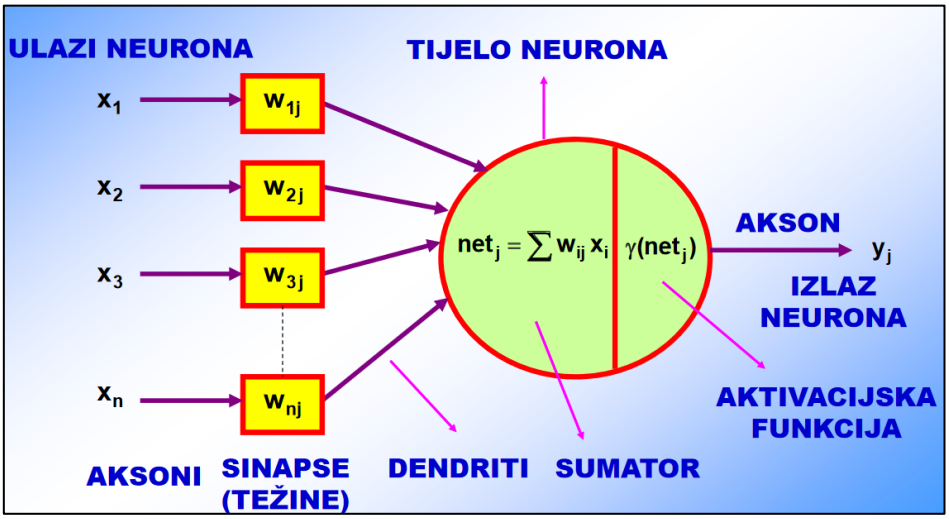
Umjetna neuronska mreža (engl. *Artificial Neural Network,* ANN) je skup umjetnih neurona međusobno povezanih sinapsama, koji međusobno komuniciraju te daju određeni izlaz. Struktura umjetnih (računalnih) neuronskih mreža napravljena je po uzoru na ljudski mozak te pokušava oponašati biološki neuron (Slika 7).



Slika 7. Biološki neuron [8]

Biološki neuroni dobivaju podražaje (električne signale) od ostalih neurona putem dendrita. Taj podražaj putuje biološkim neuronom kroz akson do sinapse. Sinapsa je mjesto gdje se signal prenosi s jednog neurona na drugi. Ako je podražaj dovoljno jak, signal prelazi na dendrite povezanih neurona pa se potom taj proces lančano ponavlja. Jedan realan biološki neuron može imati i preko deset tisuća sinapsi.

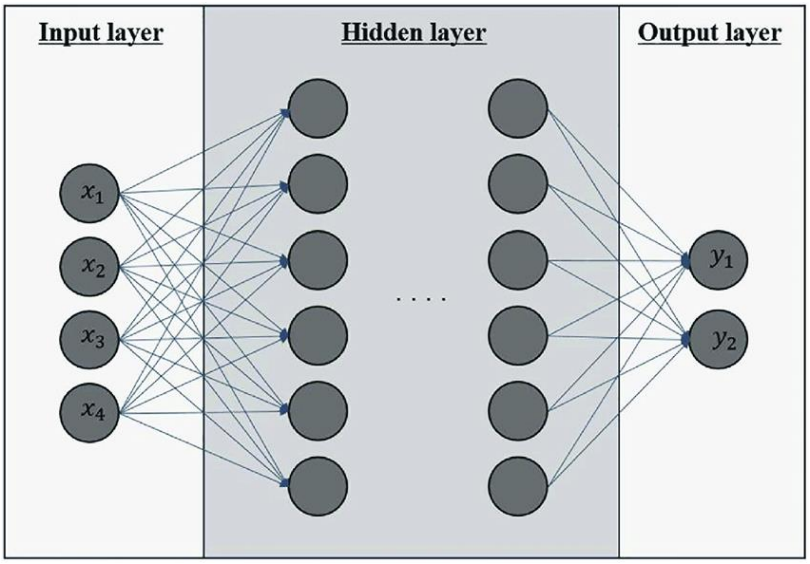
Model umjetnog neurona (Slika 8) prvi su formalizirali McCulloch i Pitts 1943. godine. Aktivnost pojedinog umjetnog neurona može se prikazati kao suma ulaza neurona, gdje su ti ulazi otežani na način da su pomnoženi određenim faktorima, koji se nazivaju težine neurona [9].



Slika 8. Prikaz umjetnog neurona [9]

Dakle, svaka ANN sastoji se od neurona povezanih sinapsama (težinama). Ovi neuroni (čvorovi) podijeljeni su u slojeve (Slika 9), ovisno o redoslijedu njihove aktivacije:

- Ulazni sloj
- Skriveni sloj
- Izlazni sloj



Slika 9. Struktura tipične neuronske mreže [10]

U ANN, svakom neuronu dodijeljen je aktivacijski broj. Aktivacijski broj je broj najčešće između nula i jedan, koji određuje granicu aktivacije neurona. Slično kao i u biološkom neuronu, ako ulaz u neuron zadovolji granicu aktivacije (tj. ako je podražaj dovoljno snažan),

taj će se neuron aktivirati i pomoću sinapse (tj. veze) proslijediti signal idućem povezanom neuronu u slijedećem sloju ANN.

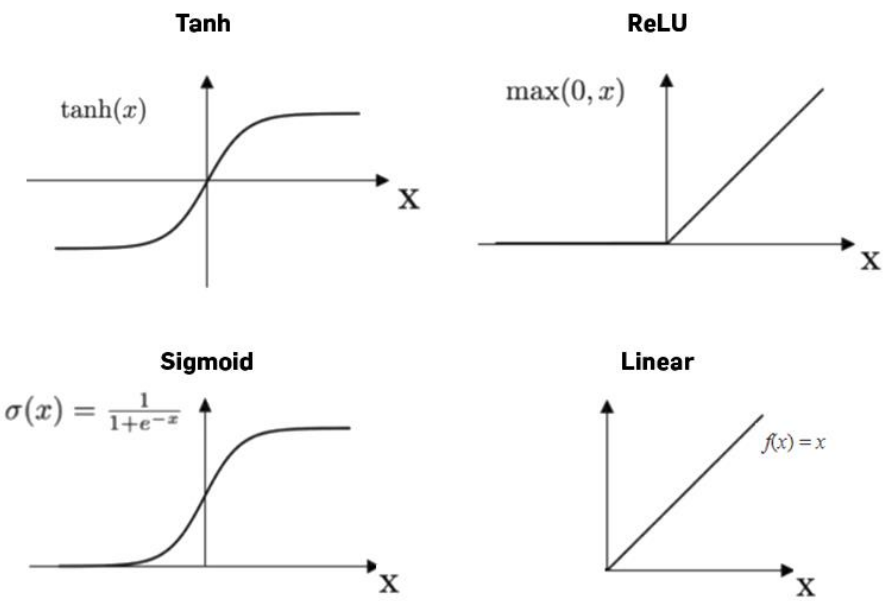
Svakoj vezi dodijeljena je težina s kojom se množi aktivacijski broj aktiviranog neurona prije dolaska do idućeg sloja. U izlaznom sloju, suma otežanih ulaza može se prikazati [8]:

$$net_{Ok} = \sum_{j=1}^{J} w_{kj} y_j \qquad , k = 1,2, \dots , K \tag{2}$$

K je broj neurona izlaznog sloja, J broj slojeva, dok su *w* pripadajuće težine, a *y* vrijednosti aktivacijskih brojeva.

Prije usporedbe aktivacijskog broja sa sumom potrebna je aktivacijska funkcija. Aktivacijska funkcija uzima sumu otežanih ulaza i vraća izlaz koji je broj od nula do jedan, kako bi se mogao usporediti s aktivacijskim brojem neurona. Neke od korištenih aktivacijskih funkcija su tanh, sigmoidna funkcija, linearna funkcija i ReLU (Slika 10). Izlaz iz ANN je tada (za npr. sigmoidnu funkciju):

$$net_{Ok} = \sigma\left(\sum_{j=1}^{J} w_{kj} y_j\right) \qquad , k = 1,2, \dots , K \tag{3}$$



Slika 10. Aktivacijske funckije [11]

Bitno je napomenuti i pojam *biasa*. *Bias* je dodatni ulaz svakog sloja ANN koji nije pod utjecajem prethodnog sloja. Analogan pojmu konstante u linearnoj funkciji, *bias* određuje da čak i ako mreža ne dobiva ulaz, ona će biti aktivirana njegovom zadanom vrijednošću [12].

Izraz za vrijednost neurona izlaznog sloja neuronske mreže je tada:

$$net_{Ok} = \sigma\left(\sum_{j=1}^{J} w_{kj} y_j + b_j\right) \quad , k = 1,2, \dots, K \tag{4}$$

Umjetne neuronske mreže široke su primjene, a najčešće se upotrebljavaju u:

- Obradi zvuka
- Obradi slike
- Prepoznavanju lica
- Predviđanju ekonomskog rasta
- Upravljanju tehničkim sustavima
- Simulacijama

Učenje neuronske mreže podrazumijeva proces mijenjanja vrijednosti težina sinapsi. Obično se učenje ANN dijeli na dvije faze:

1. Unaprijedna faza (engl. *Feed – forward*)
2. Povratna faza (engl. *Backpropagation*)

Inicijalno, sve težine neuronske mreže zadane su nasumično. U nultom koraku, za zadane ulazne podatke računaju se izlazi, tj. vrijednosti aktivacijskog broja neurona izlaznog sloja mreže, pomoću (4). Zatim se računa pogreška pomoću funkcije cilja E_f, tj. određuje se razlika između željene i trenutne vrijednosti u izlaznom sloju neurona. Jedna od uobičajenih metoda računanja odstupanja od željene vrijednosti je suma kvadrata pogreške:

$$E_f = \frac{1}{2}\sum_{n=1}^{N} (d_n - O_n)^2 \tag{5}$$

Funkcija cilja se zatim u povratnoj fazi učenja nastoji minimizirati. Nakon povratne faze mijenjaju se vrijednosti težina veza i *biasa*.

Neuronska mreža uči sve dok se za zadane ulazne podatke ne postignu željeni izlazni podaci u izlaznom sloju. Ovakav način učenja, s poznatim željenim vrijednostima izlaznih varijabli naziva se nadzirano učenje, za razliku od nenadziranog učenja, gdje željene vrijednosti izlaznih varijabli nisu eksplicitno poznate.

Navedeni, ukratko objašnjeni način učenja neuronske mreže samo je jedan od mogućih načina pronalaska traženih vrijednosti težina. Za pronalazak odgovarajućih vrijednosti težina, *biasa*, brojeva slojeva, vrijednosti čvorova (tj. neurona) ANN, njihovog rasporeda i povezanosti (tj. topologije) moguće je koristiti i mehanizme evolucije opisane u prethodnom poglavlju.

5. NEUROEVOLUCIJA PROMJENJIVIH TOPOLOGIJA (engl. *NeuroEvolution of Augmenting Topologies*)

Pojam neuroevolucije (NE) odnosi se na umjetnu evoluciju neuronskih mreža koristeći GA. Dok uobičajene neuronske mreže uče pomoću povratne faze (*backpropagation*), neuroevolucijski pristup traži povoljne težine i *bias*-e pomoću evolucijskih mehanizama, spomenutih u [5].

NE je pokazala dobre rezultate u složenijim zadacima podržanog učenja, učenja koje se zasniva na nagrađivanju željenog ponašanja i kažnjavanju onog neželjenog [13]. Ukratko, NE traži onu mrežu koja dobro izvršava zadani zadatak na temelju njenog ponašanja u prostoru. Kao takva, NE je obećavajući pristup u rješavanju problema podržanog učenja iz razloga što ne traži funkciju vrijednosti, nego nagrađuje određeno ponašanje [14].

U tradicionalnom NE pristupu, topologija mreže (dakle njen broj slojeva, veza i *bias*-a) odabrana je prije početka eksperimenta te je stalna tokom eksperimenta. Evolucija zatim pretražuje težine veza ove potpuno povezane strukture mreže te dopušta reprodukciju mrežama koje su pokazale najpovoljnije ponašanje. To podrazumijeva upotrebu evolucijskih mehanizama [5] nad vektorima težina mreža. Krajnji cilj NE s trajnom topologijom je optimizacija težina veza mreže koje određuju njene performanse.

Međutim, ne utječu samo težine veza na ponašanje neuronske mreže, već i njena topologija. Promjena strukture veze dokazano je učinkovita u nekim metodama nadziranog učenja [15], no u principu, potpuno povezana mreža sposobna je aproksimirati bilo koju kontinuiranu funkciju. Zašto gubiti vrijeme u pronalaženju idealne topologije?

Argument za evoluciju topologije i težina proizašao je iz rada [16], koji je ustvrdio da evolucija topologije neuronske mreže štedi vrijeme ljudima koji pokušavaju odlučiti o prikladnoj topologiji mreže za zadani problem. Iako svi NE sustavi stalne topologije koriste potpuno povezani skriveni sloj, odluka o broju neurona skrivenog sloja i dalje se utvrđuje metodom pokušaj-promašaj. U [16], evolucija topologije mreže, kao i težina veza, pokazala se uspješnom u rješavanju najtežeg referentnog problema obrnutog njihala do tada. Međutim, kasniji rezultati pokazali su da ESP metoda (engl. *Enforced Subpopulations*) [13] rješava isti problem pet puta brže, ponovnim pokretanjem s nasumičnim brojem neurona skrivenog sloja kada god bi zaglavila.

Cilj [14] bio je pokazati da i evolucija topologije, ako je osmišljena ispravno, može značajno poboljšati neuroevoluciju, baš kao i evolucija težina veza neuronske mreže. Taj je rad

predstavio novu metodu NE, nazvanu *NeuroEvolution of Augmenting Topologies*, ili ukratko, NEAT, čiji je cilj korištenje strukture neuronske mreže kao načina minimiziranja potrage za ispravnim težinama veza. Ako struktura evoluira na način da se njena topologija najprije minimizira pa zatim postupno povećava, to rezultira značajnim poboljšanjem brzine učenja. Takvo poboljšanje rezultat je minimizacije topologije kroz evoluciju, a ne na samom kraju.

Evolucija strukture postepeno kroz generacije sa sobom donosi i određene tehničke izazove implementacije ovog pristupa u računalu, a to su:

1. Postoji li genetski prikaz koji dozvoljava križanje različitih topologija na smisleni način?
2. Kako zaštititi inovaciju topologije, kojoj je potrebno nekoliko generacija optimizacije, da potpuno ne nestane iz populacije?
3. Kako minimizirati topologije kroz evoluciju, bez potrebe za fitnes funkcijom koja mjeri složenost topologije?

NEAT algoritam sadrži rješenje za svaki od ovih izazova, kako je opisano u nastavku.

5.1. Topology and Weight Evolving Artificial Neural Networks (TWEANN)

5.1.1. Kodiranje TWEANN

Umjetne neuronske mreže evoluirajućih topologija i težina (engl. *Topology and Weight Evolving Artificial Neural Networks*) mogu se kodirati direktno i indirektno. Direktno kodiranje, koje koristi većina TWEANN, u genotipu jasno prikazuje svaki neuron (čvor) i vezu koja će se pojaviti u fenotipu. S druge strane, indirektno kodiranje najčešće samo prikazuje pravila po kojima će se izgraditi fenotip.

Direktno kodiranje TWEANN svodi se na tradicionalni prikaz koji koriste ostali GA, a to je binarno kodiranje. Implementirano u metodi nazvanoj *Structure Genetic Algorithm* (sGA), svaki niz bitova prikazuje matricu veza mreže [17]. Iako najjednostavnije za razumjeti, postoji nekoliko ograničenja kod kodiranja TWEANN binarnim kodiranjem:

- Veličina matrice veza je kvadrat broja čvorova. Ako je broj čvorova velik, matrica postaje prevelika za praktičnu upotrebu.
- Veličina niza bitova mora biti jednaka za sve jedinke u populaciji. To znači da najveći broj čvorova mora biti unaprijed definiran.

- Korištenjem niza bitova za prikaz grafovske strukture kao što su neuronske mreže teško je osigurati da će križanje pokazati dobre rezultate.

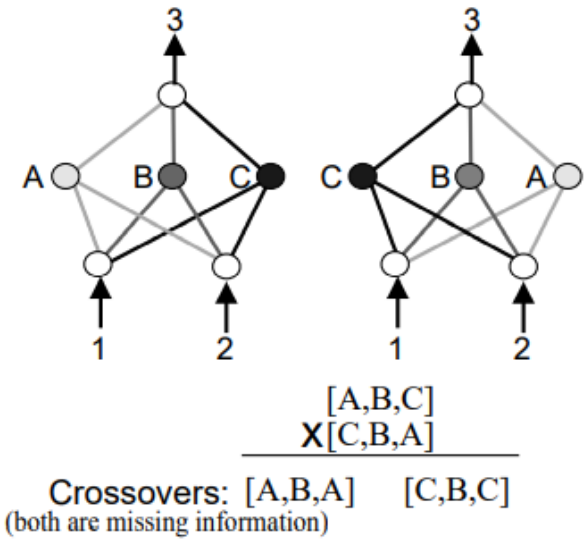
Kodiranje grafovima druga je vrsta kodiranja TWEANN. U *Parallel Distributed Genetic Programming* (PGDP) sustavu [18] koristi se dvostruka reprezentacija fenotipa za različite vrste križanja. Prva reprezentacija je struktura grafa, dok druga prikazuje genom koji sadrži veze ulaza i veze izlaza čvorova. Problem kod ovakvog kodiranja je činjenica da PGDP, kao i sGA, ima gornju granicu brojeva čvorova mreže, definiranu veličinom 2-D matrice koja prikazuje moguće strukture grafova.

Treća vrsta kodiranja, *GeNeralized Acquisition of Recurrent Links* (GNARL) [19], potpuno zaobilazi mehanizam križanja u TWEANN, jer križanje mreža različitih topologija može dovesti do nestajanja funkcionalnosti mreže. Iako GNARL koristi kodiranje grafovima, u srži se razlikuje od PGDP jer ne koristi križanje kao mehanizam evolucije. GNARL pokazuje da TWEANN ne treba mehanizam križanja da bude funkcionalan.

Vrsta indirektnog kodiranja je metoda *Cellular Encoding* (CE) [16]. U CE, genomi su programi napisani u specijaliziranom jeziku za transformacije grafova. Glavna prednost CE su kompaktne reprezentacije genoma. U CE, isti geni mogu se više puta koristiti tijekom izgradnje mreže. Zbog toga, geni u indirektnom kodiranju kao što je CE ne prikazuju direktno fenotip pa tako mogu nepredvidivo utjecati na rezultate evolucije. Iz tog razloga NEAT koristi direktno kodiranje.

5.1.2. Učestali problemi TWEANN

Jedan od glavnih problema TWEANN je problem permutacija, nazvan *Competing conventions problem*. U TWEANN rješenje može biti prikazano u više mogućih permutacija. Kada genomi koji prikazuju isto rješenje imaju različito kodirani genom, križanjem se najčešće ne dobije validno rješenje.



Slika 11. ***Competing conventions problem*** **[14]**

Slika 11 prikazuje jednostavnu mrežu s tri neurona u skrivenom sloju. Rješenja koja prikazuju mogu biti ista u 3! = 6 različitih permutacija. Križanjem mreža na slici 11 izgubit će se jedna trećina informacije koju su imala oba roditelja.

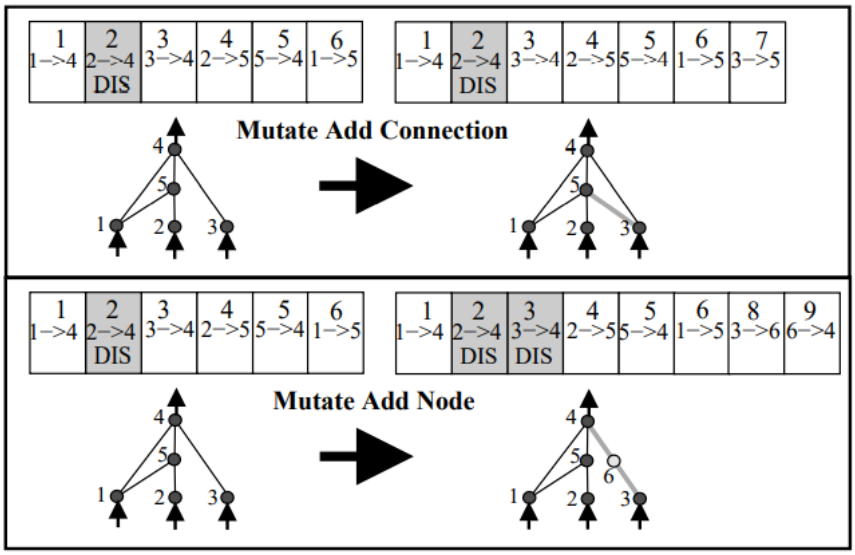
Drugi problem u TWEANN je zaštita inovacije kroz specijaciju. Inovacija se događa na način da se pomoću mutacije mreži mijenja struktura. Problem ovakve vrste mutacija je što se promjenom strukture mreže najčešće smanjuje njena ocjena fitnesa. Ovo smanjenje fitnesa uzrokovano je činjenicom da se inicijalno (u prvoj generaciji u kojoj je dodana nova veza/čvor) promjena strukture mreže nije imala vremena optimizirati. Zbog smanjenja fitnesa, inovacija kao takva ima male šanse za prolazak u iduću generaciju, stoga je inovacije potrebno zaštititi kako bi imale priliku optimirati svoju novonastalu strukturu.

Treći problem TWEANN je inicijalizacija populacija. Nasumičnom inicijalizacijom populacija osigurana je raznovrsnost topologija, ali postoji šansa za stvaranjem nefunkcionalne mreže. Takve mreže nemaju povezan svaki neuron ulaznog sloja sa svakim neuronom izlaznog sloja. Također, nasumičnom inicijalizacijom postoji mogućnost kreiranja velikog broja nepotrebnih veza ili čvorova. To se protivi ideji pronalaska minimalističkog rješenja. Budući da ocjena fitnesa nije povezana s brojem čvorova i veza u mreži, veće mreže mogle bi dominirati evolucijom dokle god je njihova ocjena fitnesa velika [14].

5.2. Kako funkcionira NEAT algoritam

5.2.1. Kodiranje u NEAT

Genetsko kodiranje u NEAT osmišljeno je na način da omogući lagano križanje. Genom je linearna reprezentacija povezanosti mreže (Slika 12). Svaki gen sadrži informaciju o povezanosti dva čvora mreže. Svakom genu također je dodijeljen i broj inovacije, kako bi se lakše uparili međusobno podudarni geni dviju jedinki prilikom križanja.



Slika 12. Mutacije u NEAT [14]

Mutacije u NEAT mogu promijeniti i težine veza i strukturu mreže. Težine veza mijenjaju se na isti način kao i u ostalim NE sustavima (određena šansa za promjenu težine).

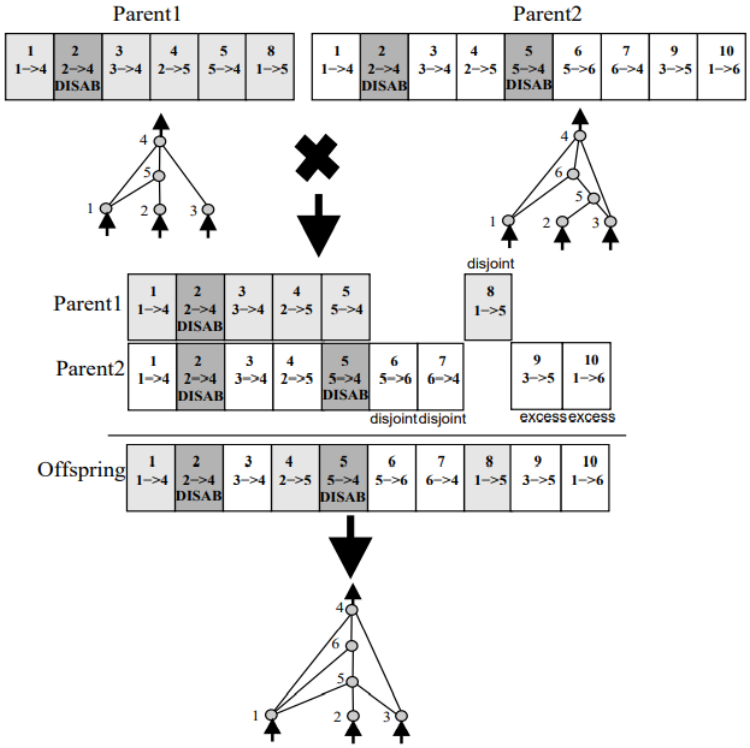
Struktura može mutirati na dva načina (Slika 12), a to su dodavanje nove veze ili dodavanje novog čvora. Pri dodavanju nove veze, spajaju se dva otprije nepovezana čvora. Ova nova veza nasumične težine (zajedno sa svojim inovacijskim brojem) dodaje se na kraj genoma. Pri dodavanju novog čvora, prijašnja veza je podijeljena u dva dijela. Tada se stara veza koja više nije aktivna u genomu označava kao neaktivna, a dvije nove veze dodaju se na kraj genoma. Bitno je da prilikom mutacije čvora, nova veza koja vodi do novog čvora dobiva težinu 1, a veza koja vodi iz novog čvora dobiva istu težinu kao i prošla (sada neaktivna) veza. Na taj način zaštićuje se inovacija, jer novonastala struktura neće inicijalno gubiti na ocjeni fitnesa zbog mutacije.

5.2.2. *Križanje u NEAT*

Kod križanja populacije različitih topologija, bitno je znati koji geni jednog roditelja podudaraju s genima drugog. Tu informaciju o porijeklu gena nosi globalni inovacijski broj (engl. *Global innovation number*). Globalni inovacijski broj dodijeljen je svakom novonastalom genu u sustavu.

Na primjer, na slici 12, mutacija veze nastala je prije mutacije čvora. Kao takva, mutacija veze (tj. gen koji ju prikazuje u genomu) dobit će sljedeći slobodni globalni inovacijski broj, a to je u ovom slučaju 7. Zatim se odvija mutacija čvora, koja u sustav dodaje dvije nove veze. Geni koji prikazuju te nove veze u genomu će redom dobiti sljedeće slobodne inovacijske brojeve, a to su 8 i 9, jer su oni nastali nakon mutacije veze. Kad dođe do križanja ovih genoma u budućnosti, njihov potomak naslijedit će njihove inovacijske brojeve. Na ovaj način kronološki je poznato porijeklo svakog gena kroz evoluciju.

Mogući problem ovog pristupa javlja se ako u istoj generaciji struktura mutira na isti način u više različitih jedinki. To bi značilo da se različiti inovacijski brojevi pridjeljuju za istu inovaciju. Međutim, ovaj problem jednostavno se rješava praćenjem inovacija koje su nastale u trenutnoj generaciji putem liste. Tako se istim inovacijama može pridjeliti isti inovacijski broj.



Slika 13. Primjer križanja u NEAT [14]

Križanje u NEAT je zatim vrlo jednostavno. Znajući inovacijski broj svakog gena, geni u genomima roditelja upareni su prema njihovim inovacijskim brojevima (Slika 13). Geni koji su ostali neupareni nazivaju se nespareni geni (engl. *disjoint*) ako se nalaze unutar duljine genoma, ili suvišni geni (engl. *excess*), ako se ne nalaze unutar duljine genoma. Zatim se križanje odvija na način da se upareni geni nasumično odabiru od jednog roditelja, dok se nespareni geni uvijek odabiru od roditelja s većom ocjenom fitnesa.

5.2.3. Očuvanje inovacije kroz specijaciju

Specijacija populacije omogućava jedinkama da se primarno natječu unutar svoje vrste. Ona omogućava očuvanje inovacije koja još nije imala vremena evoluirati kroz generacije te biti optimizirana kao takva.

Glavna ideja specijacije je podijeliti jedinke u grupe sličnih topologija mreža. U NEAT se jedinke dijele u vrste pomoću broja suvišnih (E) i nesparenih (D) gena, formulom [14]:

$$\delta = \frac{c_1 E}{N_g} + \frac{c_2 D}{N_g} + c_3 \overline{W} \qquad \textbf{(6)}$$

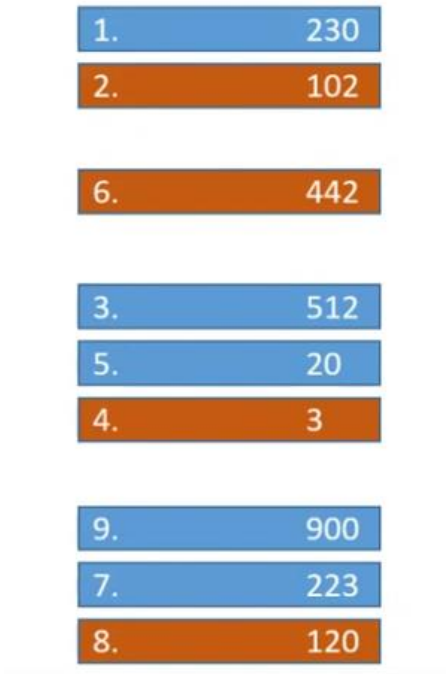
$\overline{W}$ je razlika u težinama uparenih gena, a N_g je broj gena u većem genomu, dok koeficijenti c_1, c_2 i c_3 omogućuju podešavanje važnosti ova tri faktora. δ je genetska udaljenost jedinki.

Specijacija se odvija na sljedeći način [20]:

1. Inicijalno, ne postoji nijedna vrsta. Za prvi genom populacije kreira se nova vrsta.
2. Pomoću formule (6), dobiva se genetska udaljenost jedinki δ.
3. δ_i se uspoređuje s δ_z, koji je zadani parametar genetske udaljenosti (engl. *compatibility threshold*).
4. Ako je $\delta_i < \delta_z$, jedinka se sortira u vrstu s jedinkom s kojom je uspoređivana
5. Ako je $\delta_i > \delta_z$, jedinka se uspoređuje s jedinkom iduće vrste
6. Ako jedinka više nema vrste s kojom se može usporediti, ona tvori novu vrstu

Navedeni proces ponavlja se dokle god sve jedinke u populaciji nisu sortirane u vrstu.

Zatim se selekcija odvija na razini vrste.



Slika 14. Primjer selekcije u NEAT-u [20]

Ovo znači da za roditelje iduće generacije mogu biti odabrane one jedinke koje imaju lošiju ocjenu fitnesa. Npr., na slici 14, jedinka 1 ima veći fitnes od jedinke 5, ali je jedinka 1 odabrana za roditelja iduće generacije. Na ovaj način osigurava se raznovrsnost i omogućava inovacija u populaciji.

5.2.4. Minimizacija dimenzionalnosti mreže u NEAT

U TWEANN, obično se populacija inicijalizira s nasumičnom topologijom, vezama i težinama veza. U NEAT, topologije mreža pokušavaju se minimalizirati. Inicijalna populacija tako uvijek sadrži minimalni broj veza, što znači da su svi ulazi u mrežu direktno povezani s izlazima, bez skrivenih čvorova. Nova topologija stvara se inkrementalno kako se događaju mutacije te se nove topologije održe samo ako je njihova ocjena fitnesa dovoljno dobra.

Na ovaj način su promjene u topologiji mreže u NEAT uvijek opravdane. Minimizacija dimenzionalnosti daje NEAT prednost brzine izvršavanja u odnosu na ostale NE algoritme [14].

6. IMPLEMENTACIJA NEAT ALGORITMA

NEAT algoritam iskorišten je za učenje mobilnog robota da slijedi ravnu liniju te prati izvor svjetlosti. Mobilni robot modeliran je prema Braitenbergovom pristupu, tj. nastojalo se izraditi što jednostavnije vozilo koje može izvršavati složene zadatke. Za implementaciju je odabran programski jezik *Python 3* te je kod napisan u virtualnom okolišu *PyCharm*.

Programski jezik *Python* pogodan je za ovakvu vrstu strojnog učenja, ponajviše zato što je objektno orijentiran. To je uvelike olakšalo izvedbu, jer se jedinke u populaciji mogu izraditi kao objekti klase.

Fenotipski okoliš jedinki izrađen je pomoću paketa *pygame,* preuzetog s GitHub-a [21]. Ovaj paket omogućava kreaciju modela vozila te njegovog okoliša. Model vozila u mogućnosti je kretati se naprijed – nazad po ekranu i skretati lijevo – desno rotacijom u mjestu.

NEAT algoritam implementirao se pomoću *neat-python* paketa [22]. *Neat-python* omogućava jednostavnu implementaciju određivanjem ulaza i izlaza inicijalne neuronske mreže koja upravlja mobilnim robotom. Ujedno je moguća i promjena većine evolucijskih parametara u *config* tekstualnom dokumentu. U nastavku će biti navedeni neki od bitnijih parametara koje je moguće mijenjati pomoću *neat-python*:

- *Fitness_criterion* – kriterij po kojem će se gledati uvjet prekida evolucije (može biti pri najmanjem, najvećem ili prosječnom fitnesu)
- *Fitness_threshold* – prekida evoluciju i daje rješenje kad određena jedinka (ili skup, u slučaju prosječnog fitnesa) prijeđe zadanu ocjenu fitnesa
- *Pop_size* – određuje broj jedinki koje će se inicijalizirati na početku svake generacije
- *Activation_default* – određuje koja će aktivacijska funkcija biti korištena
- *Conn_add_prob* – šansa mutacije dodavanjem nove veze
- *Conn_delete_prob* – šansa mutacije brisanjem nove veze
- *Node_add_prob* – šansa mutacije dodavanjem novog čvora
- *Node_delete_prob* – šansa mutacije brisanjem novog čvora
- *Weight_mutate_rate* – šansa mutacije težine veze
- *Num_hidden* – broj inicijalnih čvorova u skrivenom sloju mreže
- *Num_inputs* – broj inicijalnih ulaza mreže
- *Num_outputs* – broj inicijalnih izlaza mreže

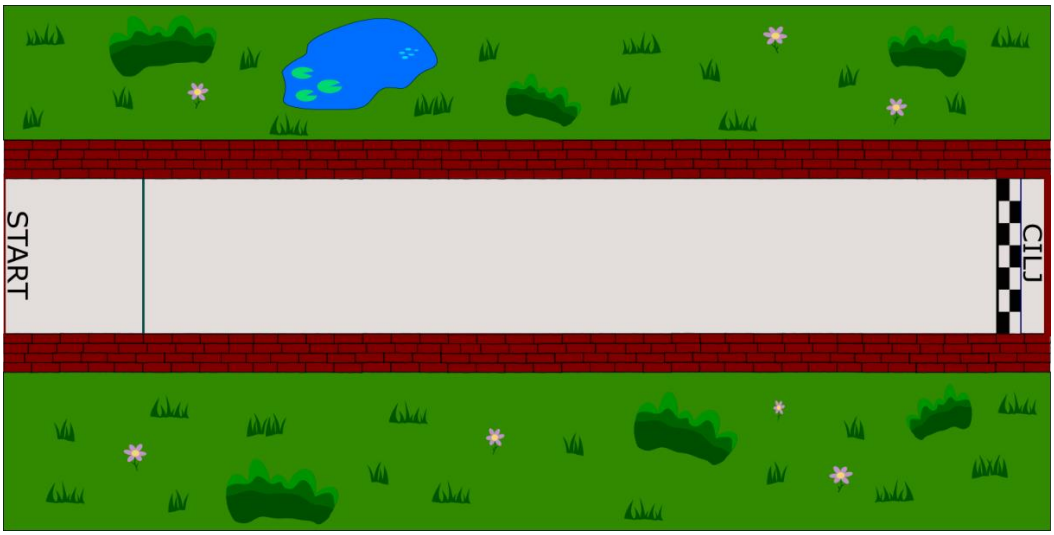
- *Compatibility_threshold* – parametar genetske udaljenosti (δ_z)
- *Max_stagnation* – vrste koje ne pokažu napredak u zadanom broju generacija biti će maknute iz populacije
- *Species_elitism* – broj vrsta koje će biti zaštićene od stagnacije, kako ne bi došlo do potpunog izumiranja populacije
- *Elitism* – broj jedinki najveće ocjene fitnesa u svakoj vrsti koje će se očuvati i preslikati u sljedeću generaciju
- *Survival_threshold* – postotak jedinki svake vrste kojima je dozvoljeno križanje
- *Min_species_size* – najmanja veličina vrste (po broju jedinki)

Obje implementacije problema adaptacija su koda [23] preuzetog s GitHub-a.

6.1. Problem slijeđenja ravne linije

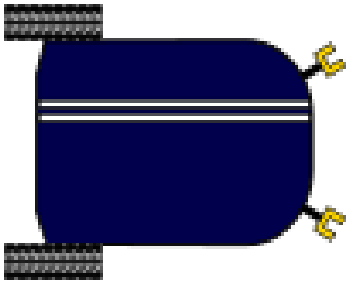
Problem slijeđenja ravne linije vrlo je jednostavan problem, koji nije ni potrebno učiti kompleksnim algoritmima kao što je NEAT. Ipak, zanimljivo je proučiti ponašanje mobilnog robota s minimalnim brojem senzora. Kako će vozilo koje jedva pojmi svijet oko sebe iz ničeg shvatiti kako se treba ponašati?

Najprije je potrebno osigurati okoliš u kojemu će se jedinke boriti za opstanak. U *pygame*-u, to je jednostavan računalni prozor koji se otvara prilikom pokretanja programa. Grafički prikaz fenotipskog okoliša izrađen je u programu za izradu vektorske grafike *Inkscape*. Okoliš (Slika 15) u kojem će se odvijati evolucija veličine je 1500x750 piksela.



Slika 15. Okoliš za problem slijeđenja ravne linije

Grafički prikaz jedinke (Slika 16), tj. mobilnog robota također je izrađen u *Inkscapeu*.



Slika 16. Prikaz jedinke (mobilnog robota)

Jedinka je veličine 80x100 piksela. Također, za programsko okruženje i sam prikaz vrlo je bitno da je pozadina transparentna.

Nakon izrade grafičkog prikaza, potrebno je osigurati kretanje jedinke u 2D prostoru. To se ostvarilo kreiranjem klase *Robot* i funkcija *update(self), drive(self)* i *rotate(self)*. Funkcija *drive(self)* omogućuje kretanje naprijed – nazad, a funkcija *rotate(self)* omogućuje skretanje lijevo – desno. *Update(self)* funkcija služi kako bi osvježila poziciju robota, za dobivanje kontinuiranosti.

Nakon toga robotu se dodaju senzori. U funkciji *radar(self, radar_angle)* dodaju se senzori pod kutevima definiranim u listi *radar_angle*. Broj senzora jednak je broju članova u listi. Ovi senzori mjere udaljenost od crvenog zida na slici, u pikselima. Za ovu implementaciju odabrana su dva senzora, pod kutevima +30 º i -30 º od x-osi. Ovakav je model vozila najsličniji Braitenbergovu vozilu.

U funkciji *collision(self)*, provjeravaju se jedinke koje su se sudarile sa zidom. Te jedinke izbrisane su iz trenutne generacije pri sudaru.

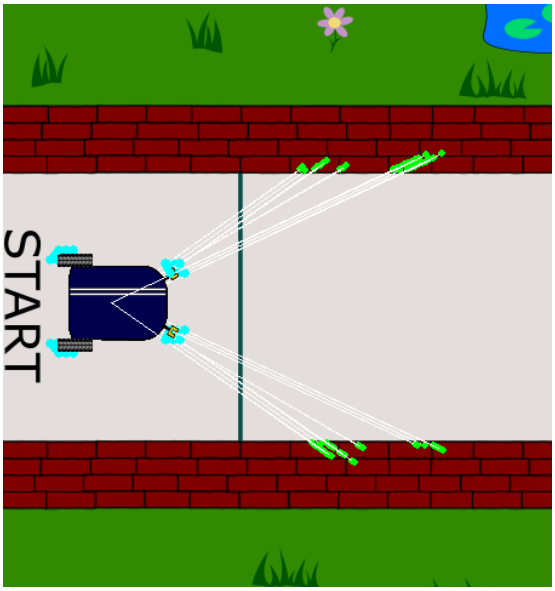
Zatim slijedi implementacija samog NEAT algoritma. Za ulaze neuronske mreže postavljena su očitanja dva senzora robota, a za izlaze iz mreže postavljeno je kretanje robota. Odabrana su samo 2 izlaza iz mreže, iako realnih izlaza ima 4 (lijevo, desno, naprijed, nazad). Glavni razlog je taj što je s manje izlaza mreža osjetljivija na ulaze. Uz postavku da je prvi izlaz kretanje naprijed-nazad, a drugi izlaz kretanje lijevo-desno, mreža će stalno morati balansirati drugi izlaz (između aktivacije za lijevo i aktivacije za desno) prema očitanjima senzora, dok će prvi izlaz morati biti stalno aktivan za kretanje unaprijed. Podjelom na 4 izlaza može se dogoditi da nasumično generirana mreža slučajno aktivira samo izlaz prema naprijed i dođe do rješenja bez obzira na vrijednosti senzora.

Nakon definicije ulaza i izlaza ostaje još osmisliti funkciju fitnesa. Cilj je razviti takvo ponašanje jedinki koje će od starta (gdje su inicijalizirane) doći do cilja. Jedinke koje nešto rade mogu se razlikovati od onih koje samo stoje na mjestu promatranjem prijeđenog puta. Dakle, osmišljena fitnes funkcija (Slika 17) gleda prijeđeni put po x-osi od točke inicijalizacije jedinke do njene pozicije u trenutku kolizije sa zidom (ili isteka vremena trajanja generacije).

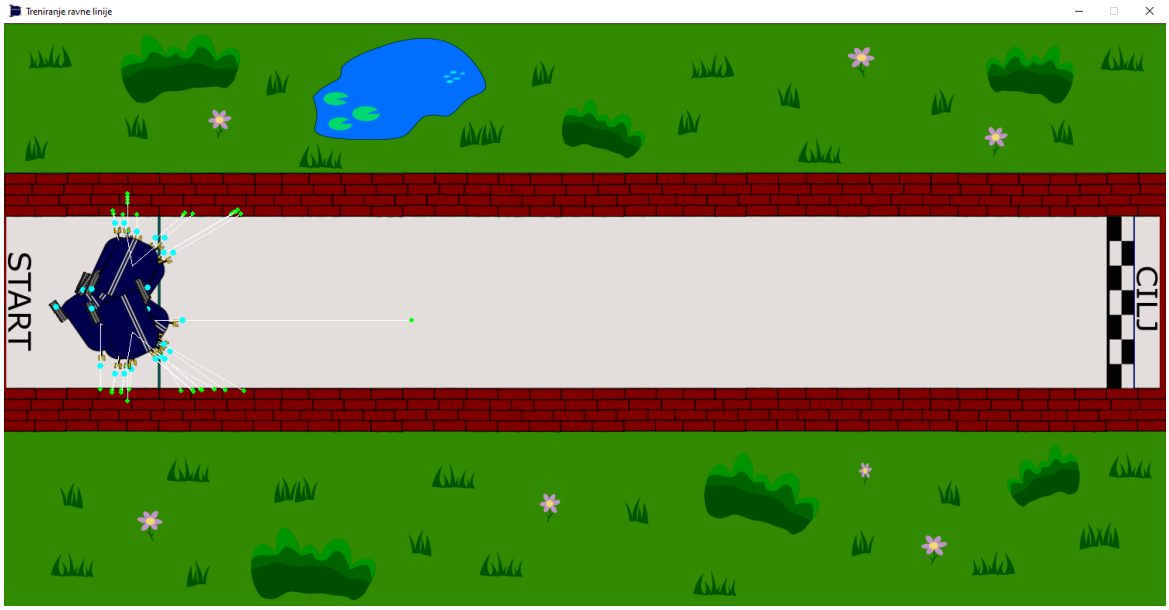
```
def distance_x(self):
    #funkcija koja vraća udaljenost jedinke od početne točke (x smjer)
    distance_travelled_x = int(self.rect.center[0] - x_spawn)
    return distance_travelled_x
```

Slika 17. Fitnes funkcija za problem slijeđenja ravne linije

Pri pokretanju programa inicijalizira se populacija na način da se kreira onoliko objekata klase *Robot()* koliko je vrijednost varijable *pop_size* u *config* tekstualnom dokumentu *neat-python* paketa. Fitnes svake kreirane jedinke mjeri se pri njenoj koliziji sa zidom, kada se ona miče iz fenotipskog prostora (tj. s ekrana). Njen genom ostaje te se nad njim vrše evolucijski mehanizmi, detaljnije opisani u 5. poglavlju. Na slici 18 prikazan je trenutak inicijalizacije populacije, kada su sve jedinke 'preklopljene' jedna na drugoj. Tirkizno plave točke su točke kolizije. Prelaskom jedne od tih plavih točaka preko crvenog zida jedinka će biti uklonjena sa zaslona. Zelene točke na krajevima bijelih crta krajnje su točke senzora koje mjere udaljenost jedinke od zida. Jedinke su inicijalizirane na poziciji x =100px i y = 355px. Na slici 19 prikazano je ponašanje te iste populacije u nultoj generaciji.



Slika 18. Inicijalizacija populacije



Slika 19. Ponašanje populacije nulte generacije

6.2. Problem praćenja svjetlosti

Problem praćenja svjetlosti složeniji je od problema slijeđenja ravne linije, jer zahtjeva stalno pozicioniranje robota u odnosu na poziciju snopa svjetlosti. Snop svjetlosti implementiran je kao pokretna meta koja se kreće po determinističkoj putanji.

Za ovaj problem napravljen je okoliš veličine 1200x750 piksela (Slika 20), također u *Inkscape*-u.

Slika 20. Okoliš za problem praćenja snopa svijetlosti

Grafički prikaz robota isti je kao i kod problema slijeđenja ravne linije (Slika 16).

Broj senzora u ovom problemu morao se povećati jer su se 2 senzora (u virtualnom okolišu) pokazala nedovoljnim za pouzdano rješavanje ovog zadatka. Tako se sada u funkciji *radar(self, radar_angle)* robotu pridodaje 5 senzora, pod kutevima -30 º, -15 º, 0 º, +15 º, i + 30 º od x-osi.

Senzori u ovoj implementaciji detektiraju samo snop svjetla, ali ne znaju poziciju smeđeg zida. Također, ne znaju ni udaljenost od snopa svjetla nego su ulazi u mrežu binarne vrijednosti. Kad određena linija senzora presječe snop svjetlosti, vrijednost tog ulaza u mrežu postaje 1, a u protivnom je 0. Kolizijom sa smeđim zidom jedinka se briše s ekrana, kao i u prethodnom problemu.

Izlazi iz mreže sada su podijeljeni kako bi se jedinka lakše pomicala u bilo kojem smjeru ekrana. Ovo je bitno jer se jedinki daje više slobode i omogućuju različita ponašanja, s obzirom na puno manje informacija koje dobiva senzorima nego u prethodnom primjeru. Dakle, sada su 4 izlaza iz mreže: naprijed, nazad, lijevo i desno.

Kako bi se nagradile jedinke koje najbolje prate pomični snop kroz vrijeme, potrebno je osmisliti fitnes funkciju, koja će zbog kontinuiranosti problema računati sumu. Stoga, fitnes funkcija će uzimati u obzir udaljenost jedinke od pomičnog snopa te nagrađivati one jedinke koje su mu najbliže.

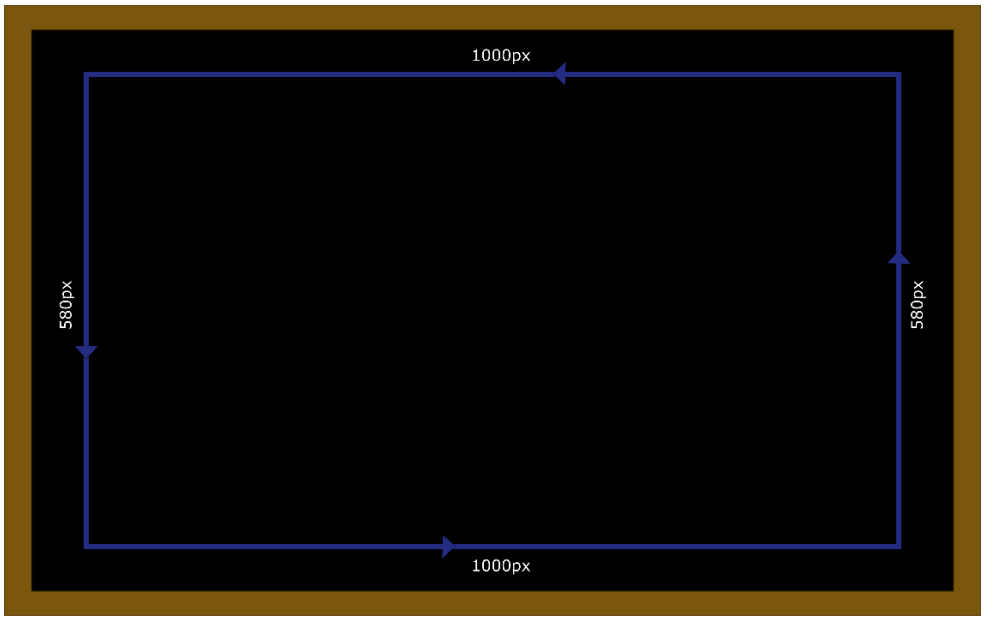
```
def measure_fitness(self,a):
    current_distance = int(math.sqrt(math.pow(self.rect.center[0] - a[0],2)
                                + math.pow(self.rect.center[1] - a[1], 2)))
    if self.is_light == 1:
        if current_distance < 250:
            self.fitness += 1
        if current_distance < 50:
            self.fitness += 10
    return self.fitness
```

Slika 21. Fitnes funkcija za problem praćenja snopa svijetlosti

Prema prikazu na slici 21, jedinke povećavaju fitnes što su bliže snopu svjetlosti. Približavanjem na svega 50 piksela od centra snopa svjetlosti, one dobivaju 10 'bodova' svakim prolaskom programa, a približavanjem na manje od 250 piksela, dobivaju po 1 bod. Ovakvom konstrukcijom fitnes funkcije pokušalo se promijeniti ponašanje jedinki na način da im se ponudi mogućnost rizika. Približavanjem bliže snopu svjetlosti jedinke dobivaju više 'bodova', ali ujedno riskiraju gubitak snopa svjetlosti iz vidokruga senzora ili zabijanje u zid (u trajektorijama gdje snop prolazi blizu zida). Bitno je

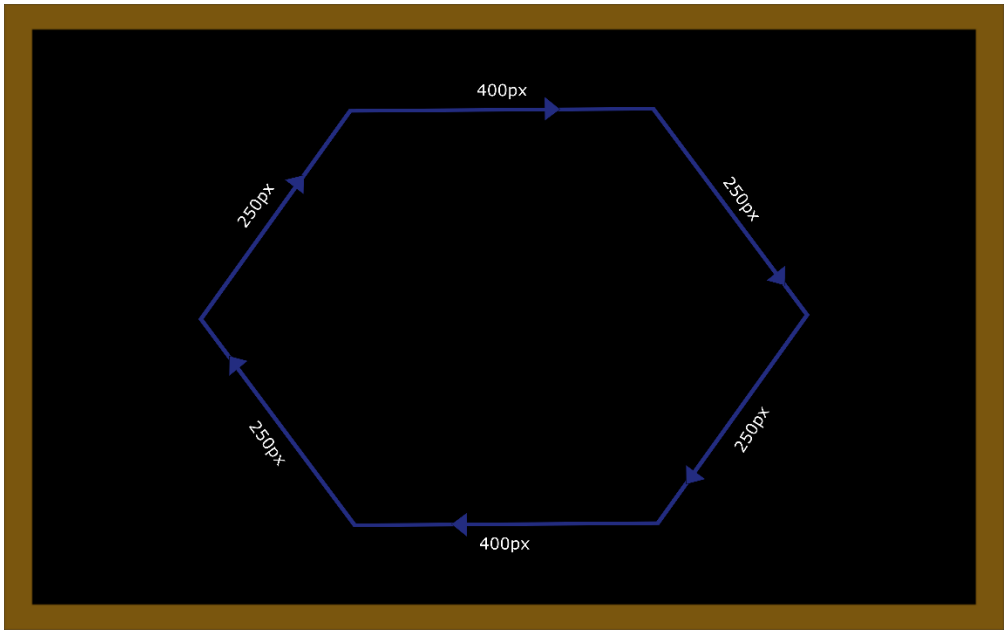
primjetiti da jedinke ne dobivaju nikakve 'bodove' ako im linija senzora ne siječe snop svjetlosti.

Za implementaciju problema korištena su 3 deterministička načina pomicanja svjetlosti: pravokutnik, gore-dolje i mnogokut, čije su trajektorije prikazane na slikama 22, 23 i 24. Na slici 25 prikazana je inicijalizacija programa, a na slici 26 ponašanje jedinka u okolišu.

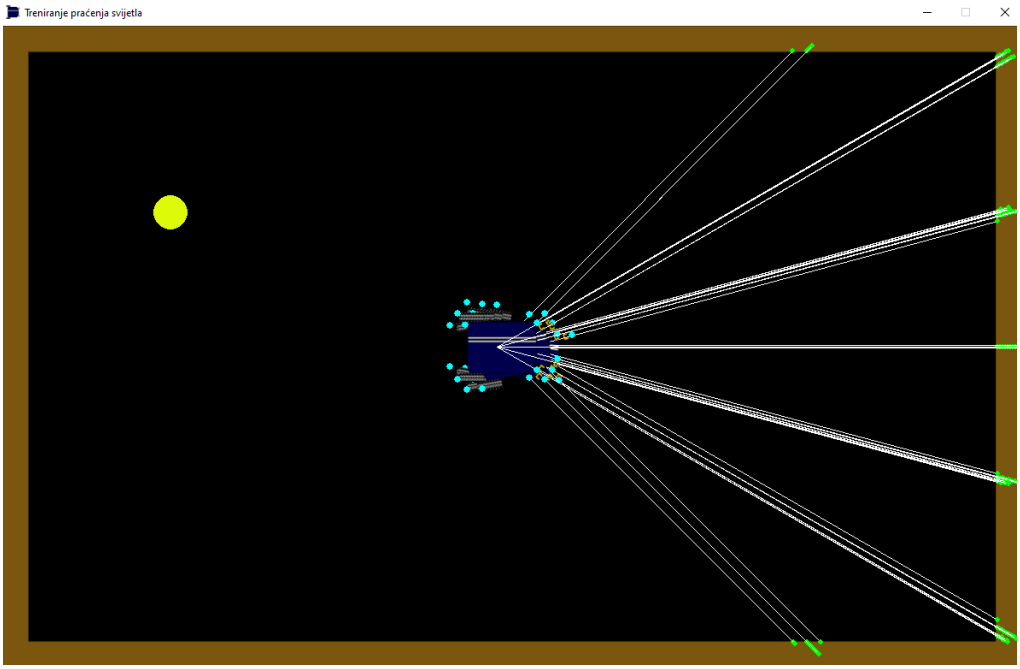


Slika 22. Izgled prve trajektorije za problem praćenja svijetlosti

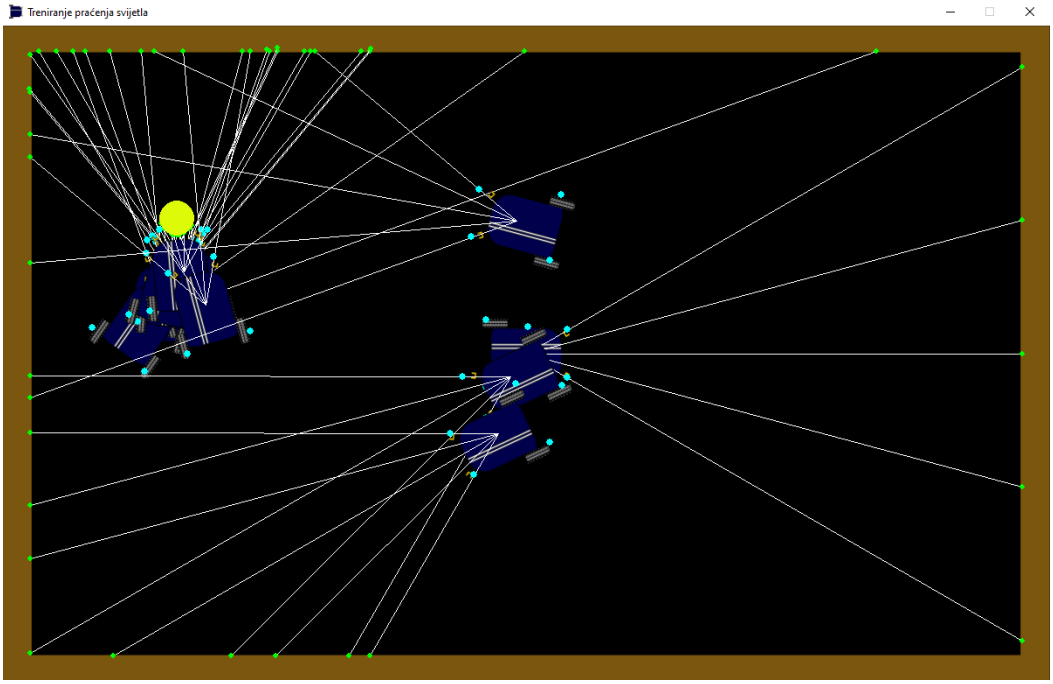
Slika 23. Izgled druge trajektorije za problem praćenja svjetlosti



Slika 24. Izgled treće trajektorije za problem praćenja svjetlosti



Slika 25. Inicijalizacija programa za problem praćenja svjetlosti



Slika 26. Praćenje svjetlosti kod trajektorije gore-dolje

7. REZULTATI

7.1. Rezultati učenja slijeđenja ravne linije

Kod problema učenja slijeđenja ravne linije, prethodno je poznat najveći fitnes kojeg jedinka može postići. To je maksimalni broj piksela koje jedinka može proći po ravnoj liniji x-osi, bez kolizije s crvenim zidom na kraju staze. Pomoću programa *Inkscape* jednostavno se može izmjeriti (aproksimativno) i izračunati maksimalna udaljenost (u pikselima) koju jedinka može proći:

$$x_{max} = x_{slike} - b_{zida} - \frac{b_{jednike}}{2} - x_{start} \approx 1350px \quad \textbf{(7)}$$

Dakle, maksimalan fitnes, tj. onaj koji se želi postići iznosi 1350.

Zbog relativne jednostavnosti problema, izvedeno je 10 pokretanja u 20 generacija, uz to da jedna generacija sadržava po 30 jedinki. Program se prekida kada jedinka prijeđe fitnes od 1350 ili kada se izmjeni 20 generacija. Tada se prikazuje najbolje pronađeno rješenje. Generacija se prekida kada prođe 15 sekundi (stvarnog vremena) ili kada više nema živih jedinki na ekranu.

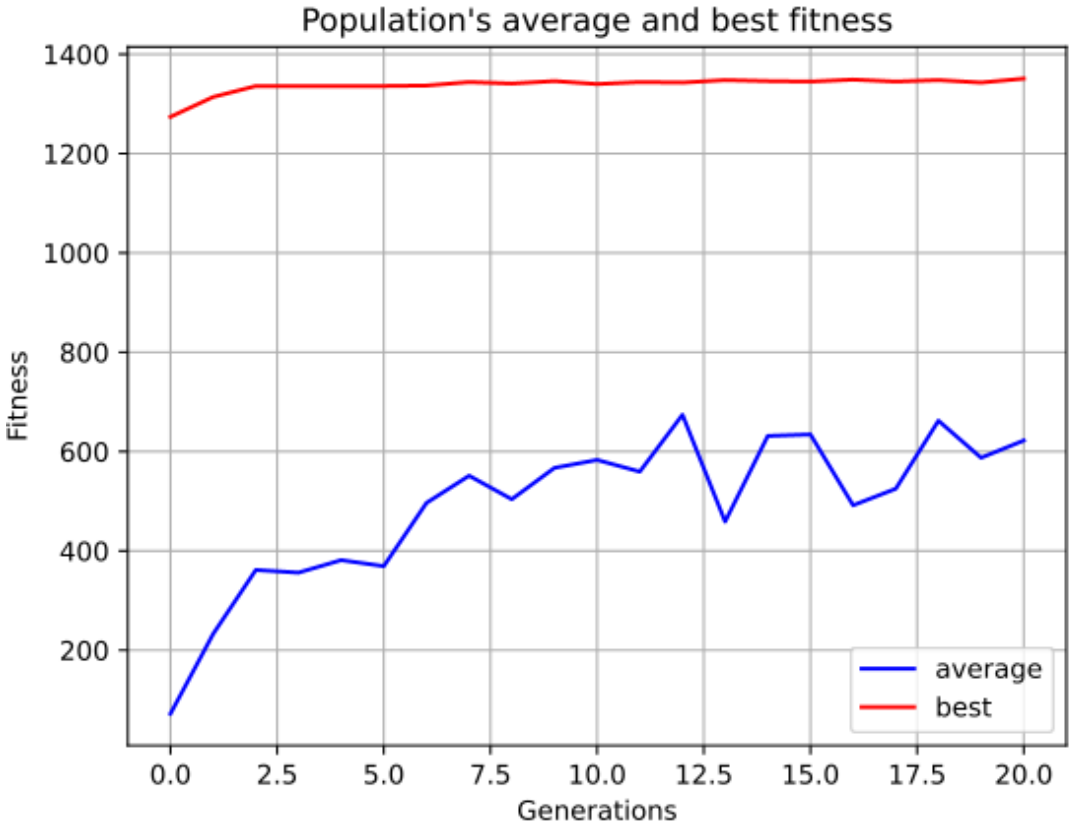
U *config* datoteci, parametri *species_elitism* i *elitism* postavljeni su na 1 jer nije teško pronaći rješenje problema pa nije potrebna raznovrsnost populacije.

Tablica 1. Rezultati učenja slijeđenja ravne linije

R. broj pokretanja	Broj generacija	Pronađeno rješenje?	Maks. fitnes	Broj stvo-renih vrsta	Značajne mutacije? (novi čvorovi i veze)
1.	7	DA	1350	4	1 veza deaktivirana
2.	17	DA	1352	4	1 veza izbrisana
3.	19	DA	1350	6	3 nova čvora
4.	20	NE	1336	4	1 čvor i 1 veza
5.	20	DA	1350	5	2 veze deak. i 2 čvora
6.	20	DA	1350	7	1 veza deaktivirana
7.	20	NE	1346	5	-
8.	16	DA	1351	5	1 veza deakt. i 1 čvor
9.	11	DA	1351	2	1 novi čvor
10.	19	DA	1352	8	1 veza deakt. i 1 čvor

U tablici 1 prikazani su rezultati 10 pokretanja programa. U 8 od 10 pokretanja algoritam je postigao traženo rješenje. U 2 pokretanja u kojima algoritam nije došao do rješenja, do tražene ocjene fitnesa nedostajalo je svega nekoliko piksela.

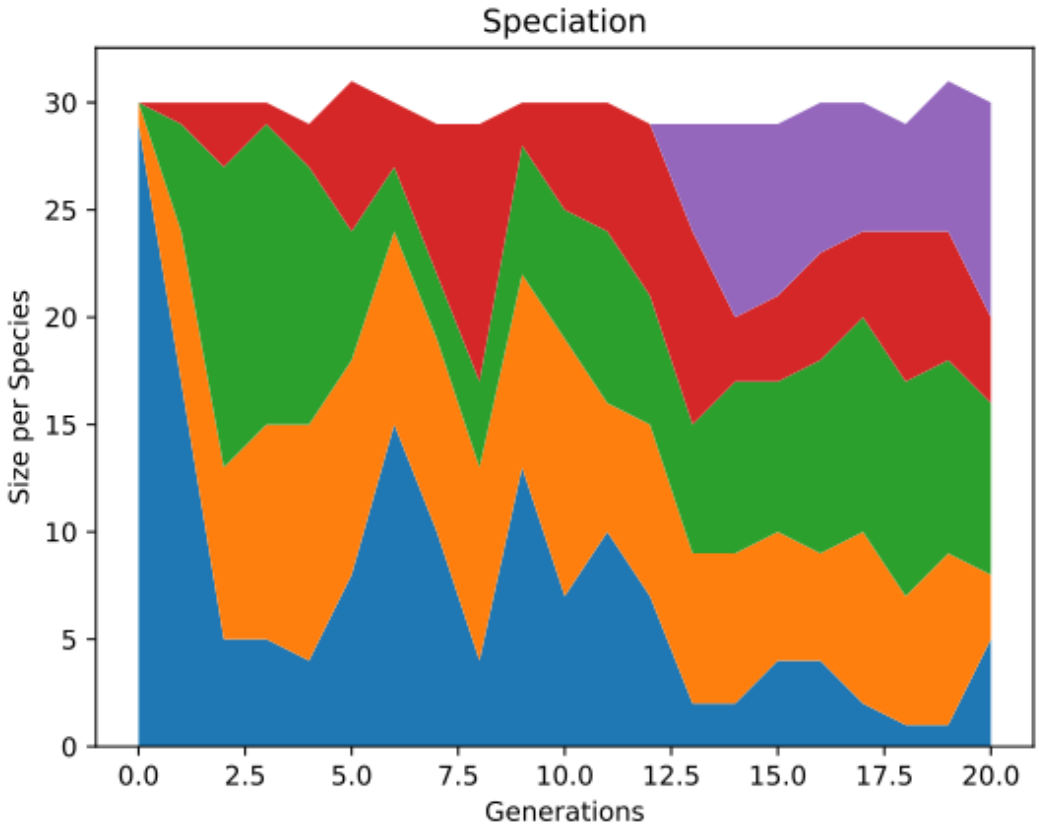
U nastavku, na slici 27 prikazana je krivulja fitnesa kroz generaciju 5. pokretanja programa. Grafovi su izrađeni pomoću *neat-python*-ovog modula *visualize*, koji pomoću paketa *graphviz* prikazuje rezultate. *Graphviz* je *python*-ov paket za upravljanje istoimenom aplikacijom, koja je moćan alat za vizualizaciju rješenja [24].



Slika 27. Prikaz maksimalne i prosječne ocjene fitnesa populacije kroz generacije, problem slijeđenja ravne linije (5. pokretanje)

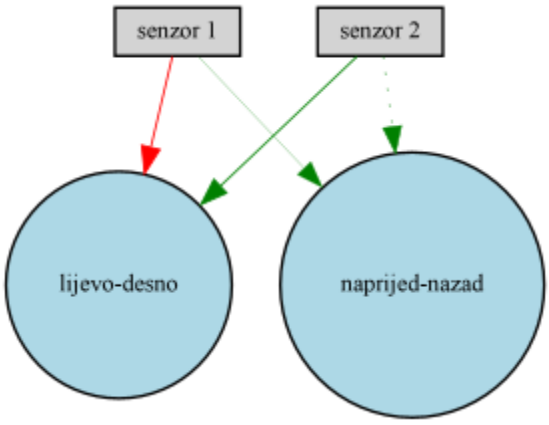
Na prikazu se vidi da su se već u 2. generaciji jedinke naučile kretati se do pozicije cilja. U ostalim generacijama njihova ocjena se polagano približava maksimalnom fitnesu.

Na slici 28 može se vidjeti podjela u različite vrste napravljena tokom generacija. Iako je problem u suštini jednostavan, algoritam je svejedno pronašao način da pomoću raznih mutacija stvori raznoliku populaciju.



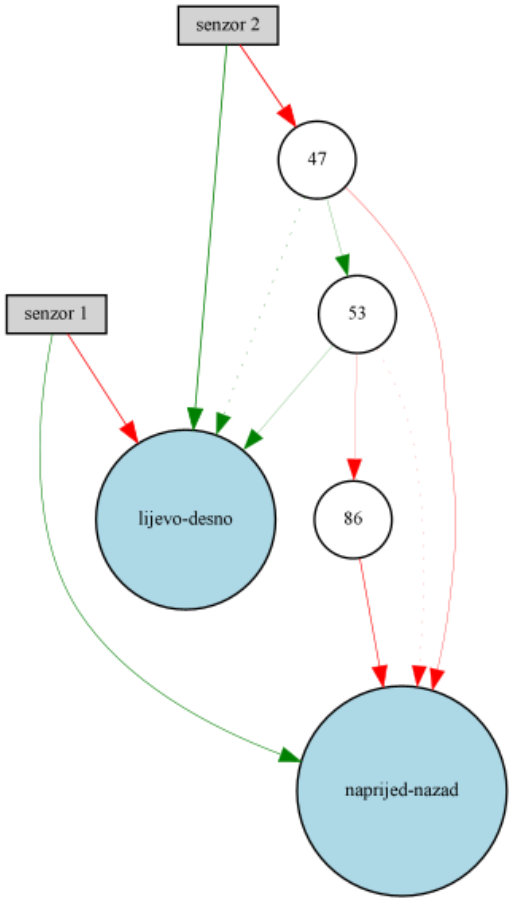
Slika 28. Prikaz specijacije, problem slijeđenja ravne linije (5. pokretanje)

Najčešćoj topologiji neuronske mreže stvorenoj u ovih 10 pokretanja (Slika 29) nedostaje veza (ili je izbrisana, ili deaktivirana) između jednog od dva senzora i izlaza naprijed-nazad. Ovo se događa jer mreža 'želi' da je izlaz naprijed-nazad cijelo vrijeme aktivan tako da se jedinka stalno kreće unaprijed. Očitanje drugog senzora želi direktno povezati samo s izlazom lijevo-desno, kojeg zatim aktivira da skrene u jednu stranu ako se približi zidu na određenu udaljenost. Na prikazu (Slika 29), točkasta crta znači da je veza neaktivna, zelena linija pozitivnu težinu, a crvena negativnu težinu (deblja linija znači veću apsolutnu vrijednost).



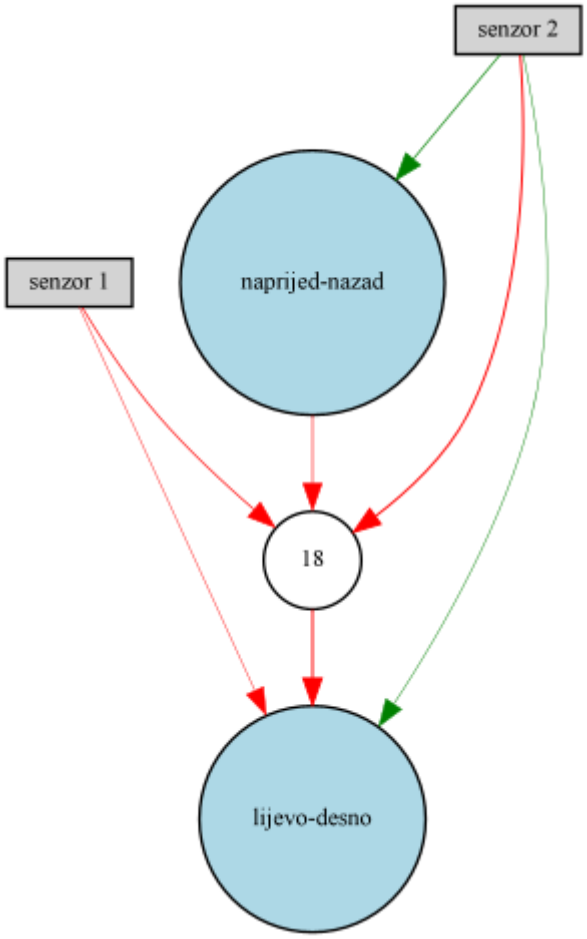
Slika 29. Najčešća topologija mreže u problemu slijeđenja ravne linije

Zanimljivo je prikazati i mrežu 3. pokretanja (Slika 30), gdje se mutacijom čak 3 nova čvora konačno došlo do rješenja u 19. generaciji. Evolucija ponekad nema najjednostavniji pristup rješavanju problema.



Slika 30. Topologija mreže 3. pokretanja, problem slijeđenja ravne linije

Naposlijetku, bit će prikazana topologija mreže 4. pokretanja, gdje rješenje zadanog zadatka nije uspješno pronađeno.



Slika 31. Topologija mreže 4. pokretanja, problem slijeđenja ravne linije

Iz slike 31 vidljivo je da kompliciranje topologije mreže nije poželjno kod rješavanja vrlo jednostavnih problema kao što je ovaj.

Promatranjem ponašanja jedinki u populaciji uočava se da većina jedinki pokušava uravnotežiti izlaz lijevo-desno i tako se kretati ravno. Međutim, dolaskom do kraja staze jedinke mijenjaju ponašanje tako da skreću u gornji ili donji kut staze. Razlog ovakve promjene ponašanja je crveni zid postavljen na kraju staze. Zbog toga što su senzori postavljeni na +30 º i -30 º, jedinka će uočavanjem zida senzorom i dalje pokušavati održati sve veću udaljenost od njega, tako da će skrenuti u onu stranu u koju bude zakrenuta u trenutku njegova uočavanja. Ovakvo ponašanje može se izbjeći postavljanjem 3. senzora u smjeru x-osi i podešavanjem njegovog ulaza u mrežu. Ipak, u ovom problemu nastojao se minimizirati model robota što je više moguće, dok god on daje zadovoljavajuće rezultate.

7.2. Rezultati učenja praćenja svjetlosti

Problem praćenja snopa svjetlosti teži je problem od slijeđenja ravne linije ne samo zato što zahtijeva stalno pozicioniranje robota u odnosu na poziciju snopa svjetlosti, nego i radi toga što unaprijed nije poznat maksimalni fitnes kojeg jedinka može postići. On se može vrlo grubo aproksimirati, ali bolje je pratiti samo ponašanje najbolje jedinke u generaciji i procjeniti ponaša li se ona kako bi trebala. Traženo ponašanje uočilo se stagnacijom maksimalnog fitnesa u populaciji.

Za ovaj problem izvedeno je 5 pokretanja za svaku trajektoriju, u 30 generacija, s time da jedna generacija sadrži po 30 jedinki. Program se prekida kada prođe svih 30 generacija. Generacija se prekida kada snop svjetlosti prođe do kraja svoje trajektorije ili kada više nema živih jedinki na ekranu.

7.2.1. Mnogokutna trajektorija

Najjednostavnija od 3 trajektorije je trajektorija u obliku šesterokuta. Ovaj način pomicanja svjetlosti predvidiv je jer nema naglih zakreta, pa je ujedno i šansa da jedinka izgubi snop iz vidokruga manja.

U *config* datoteci, parametar *species_elitism* postavljen je na 1, a *elitism* na 3. *Min_species_size* postavljen je na 1 kako bi se potaknula raznovsrnost populacije.

Rezultati u 5 pokretanja programa za mnogokutni oblik trajektorije snopa svjetlosti prikazani su u tablici 2.

Tablica 2. Rezultati učenja praćenja snopa svijetlosti, mnogokutna trajektorija

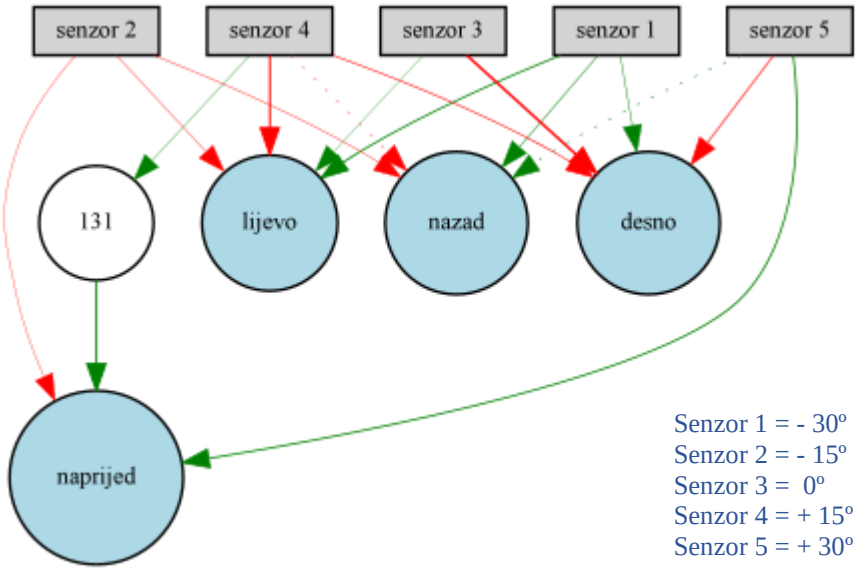
R. broj pokretanja	Postignuto traženo ponašanje?	Broj generacija do stagnacije fitnesa	Maks. fitnes	Broj stvorenih vrsta
1.	DA	4	126489	2
2.	DA	11	126489	2
3.	DA	7	126489	2
4.	DA	5	126489	1
5.	DA	7	126489	2

U tablici 2, sva pokretanja programa dostigla su maksimalni fitnes od 126489 pa zatim stagnirala do kraja izvršavanja. Konvergencijom u određeni broj više puta zaredom

vjerojatno je dostignut maksimum fitnesa koji jedinka može postići u ovom problemu. Jedinke su najčešće pratile snop s unutarnje strane mnogokuta, kako bi presjekle njegova skretanja na vrhovima mnogokuta.

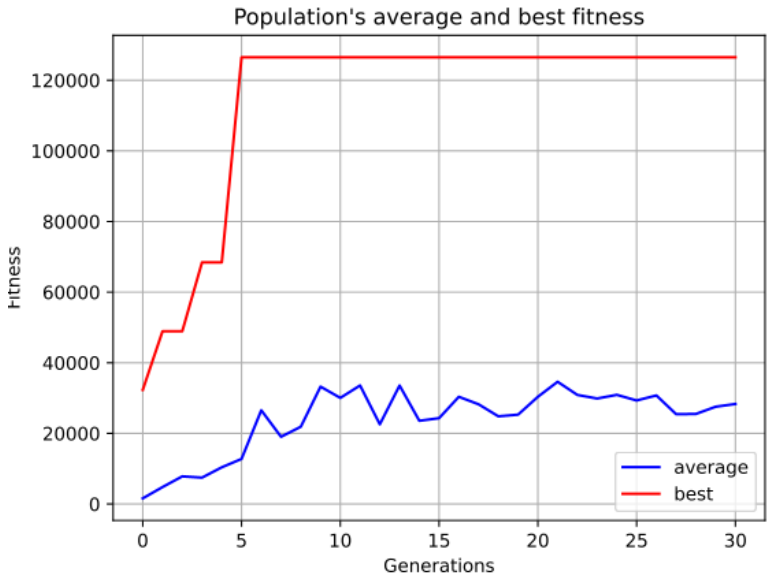
Specijacija je puno manje zastupljena nego u prošlom problemu, jer elitizam smanjuje raznovrsnost u populaciji. Manjak specijacije u ovom slučaju je neizbježan, jer je elitizam potreban kako bi se očuvalo najbolje rješenje, radi težine problema.

Mutacija dodavanja čvorova dogodila se u samo 2 od 5 pokretanja programa (Slika 32).



Slika 32. Mutacija mreže kod mnogokutne trajektorije (1. pokretanje)

Na grafu fitnesa kroz generacije 4. pokretanja programa jasno se vidi stagnacija u ocjeni fitnesa nakon 5. generacije (Slika 33).



Slika 33. Prikaz fitnesa kroz generacije, mnogokutna trajektorija (4. pokretanje)

7.2.2. *Trajektorija gore-dolje*

Trajektorija gore-dolje može predstavljati problem za praćenje zbog, naizgled stohastičkih, naglih promjena smjera. Za olakšanje problema trajektorija je odmaknuta od zida kako jedinke ne bi gubile fitnes zbog kolizije.

U *config* datoteci, parametar *species_elitism* postavljen je na 1, a *elitism* na 3. *Min_species_size* postavljen je na 1.

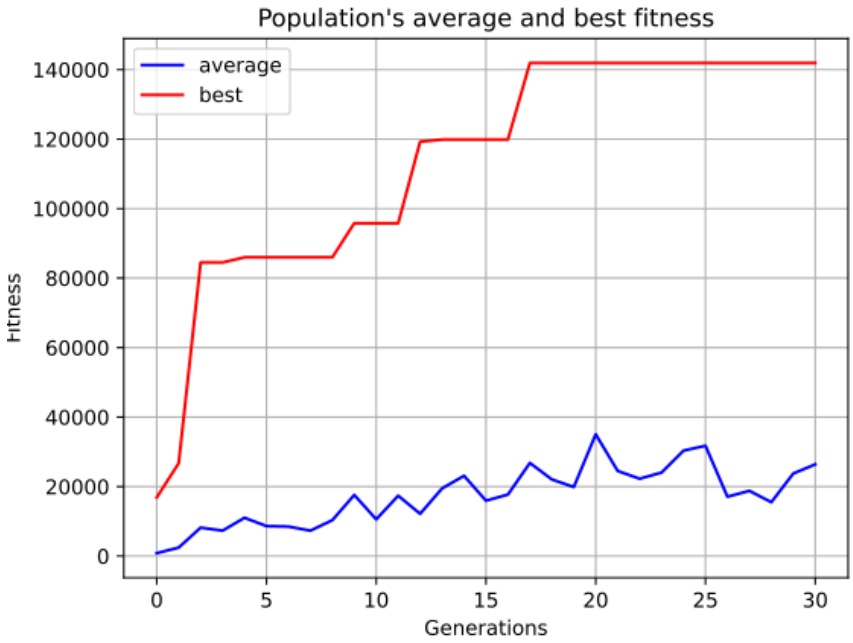
Rezultati u 5 pokretanja programa za oblik trajektorije gore-dolje (Slika 23) prikazani su u tablici 3.

Tablica 3. Rezultati učenja praćenja snopa svijetlosti, trajektorija gore – dolje

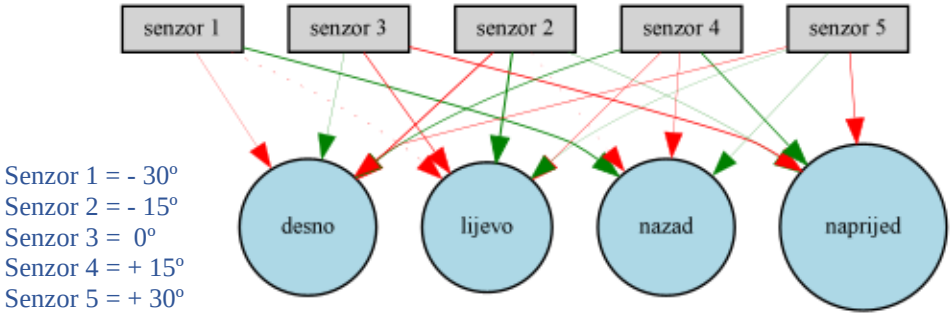
R. broj pokretanja	Postignuto traženo ponašanje?	Broj generacija do stagnacije fitnesa	Maks. fitnes	Broj stvorenih vrsta
1.	DA	5	102182	3
2.	DA	17	141897	1
3.	DA	3	97185	2
4.	DA	23	90232	2
5.	DA	28	98405	2

U svim pokretanjima programa jedinke su uspjele razviti ponašanje koje nalikuje na praćenje snopa svjetlosti. Međutim, u samo 1 od 5 pokušaja program je došao do optimalne putanje. Ponašanje se većinski sastojalo od toga da se jedinka zaokrene u lijevo ili desno za 180º kada snop dođe do kraja trajektorije. Pokušaj broj 2 razvio je optimalno ponašanje, u kojem se jedinka zakrene unazad kada snop promijeni smjer.

Na slici 34 prikazan je graf fitnesa kroz generacije 2. pokušaja, a na slici 35 izgled mreže.

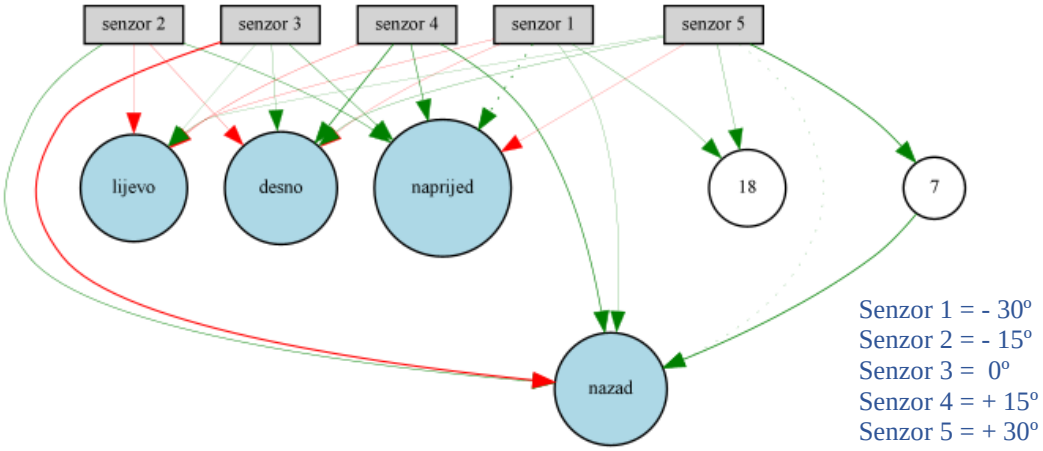


Slika 34. Prikaz fitnesa kroz generacije, trajektorija gore-dolje (2. pokretanje)



Slika 35. Izgled mreže, trajektorija gore-dolje (2. pokretanje)

Samo je 1 neuronska mreža mutirala s dva novododana čvora (Slika 36), dok su ostale zadržale izvornu topologiju.



Slika 36. Mutacija mreže, trajektorija gore-dolje (1. pokretanje)

7.2.3. Pravokutna trajektorija

Također zahtjevna za praćenje je i pravokutna trajektorija. Uz to što zahtijeva praćenje naglih skretanja snopa svjetlosti, koja bi mogla dovesti do njegovog gubitka iz vidokruga jedinke, ona prolazi i vrlo blizu zidu. Jedinke zato trebaju razviti takvo ponašanje koje će izbjegavati koliziju sa zidom dok se približavaju snopu svjetlosti što je više moguće.

U *config* datoteci, parametar *species_elitism* postavljen je na 1, a *elitism* na 3. *Min_species_size* postavljen je na 1.

Rezultati u 5 pokretanja programa za pravokutni oblik trajektorije (Slika 22) prikazani su u tablici 4.

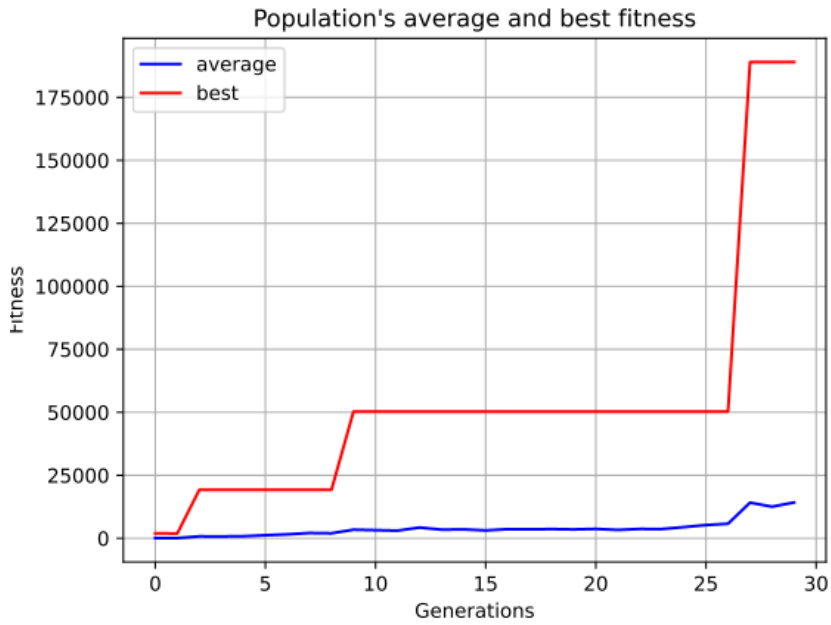
Tablica 4. Rezultati učenja praćenja snopa svijetlosti, pravokutna trajektorija

R. broj pokretanja	Postignuto traženo ponašanje?	Broj generacija do stagnacije fitnesa	Maks. fitnes	Broj stvorenih vrsta
1.	DA	26	76923	1
2.	DA	17	54758	1
3.	DA	26	189004	1
4.	DA	12	84488	3
5.	DA	26	114987	2

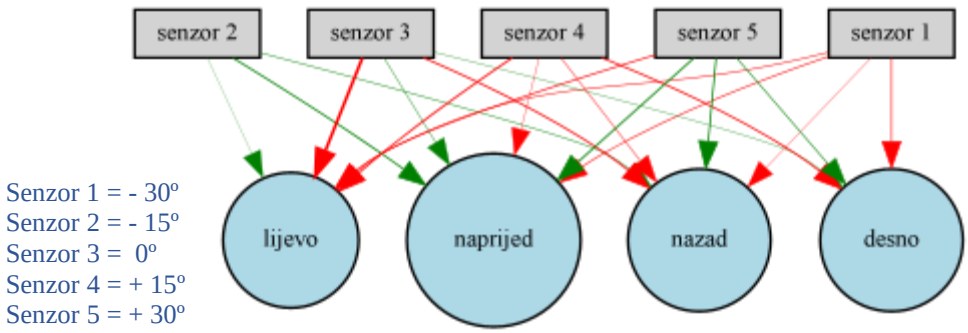
Iz različitih rezultata prikazanih u tablici 4 može se zaključiti da nije u svakom pokretanju dostignut maksimalni fitnes. Svejedno, jedinke su većinski razvile traženo ponašanje, samo je u nekim pokretanjima ono bolje izvršeno, a u nekim lošije.

U najuspješnijem 3. pokretanju iz tablice 4, jedinke su razvile vrlo zanimljivo ponašanje. Iz razloga što je snop svjetlosti sporiji od kretanja jedinke, one se ne mogu pomicati ravno za njim, jer će ga izgubiti iz vidokruga. Stoga, najuspješnije jedinke su se kretale za snopom svjetlosti po valovitoj putanji, konstantno se krećući lijevo-desno, kako bi izjednačile svoju brzinu s brzinom snopa.

Na slici 37 prikazan je graf fitnesa kroz generacije u 3. pokretanju, a na slici 38 mreža najbolje jedinke.

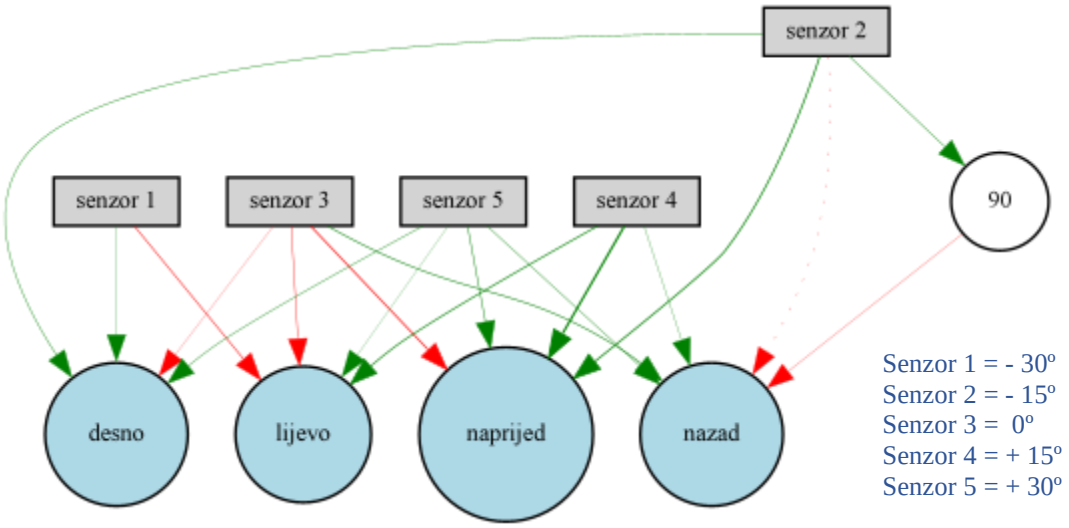


Slika 37. Graf fitnesa kroz generacije, pravokutna trajektorija, 3. pokretanje



Slika 38. Mreža najbolje jedinke, pravokutna trajektorija, 3. pokretanje

U problemu praćenja pravokutne trajektorije, najčešće su se javljale mutacije dodavanjem jednog čvora mreži (Slika 39). Ovakva mutacija javila se u dvije od pet završnih mreža.



Slika 39. Najčešća mutacija mreže kod pravokutne trajektorije

8. ZAKLJUČAK

Završni rad dotakao se osnovnih bioloških koncepata primjenjenih u današnjim računalima. Objašnjeni su glavni procesi koji se odvijaju u GA i način na koji neuronska mreža upravlja kontrolerom mobilnog robota. Također je pokazano da roboti vrlo jednostavnih struktura kroz evoluciju mogu razviti složena ponašanja, kao što su slijeđenje ravne linije i praćenje izvora svijetlosti raznih trajektorija. NEAT algoritam, koji je sam po sebi vrlo složen, uspješno je implementiran za rješavanje zadanog problema. Pojednostavljeno objašnjenje NEAT algoritma također je sadržano u ovome radu.

Ovaj rad prikazuje osnove onoga što NEAT algoritam može napraviti, dok se on uglavnom koristi za rješavanje puno složenijih optimizacijskih problema. Rad bi se mogao nadograditi implementacijom NEAT algoritma na probleme aktualne u današnje vrijeme, kao što su učenje mobilnog robota da pronađe optimalnu putanju kroz npr. skladište, ili optimizacija putanje sletanja svemirske letjelice u svemiru.

U konačnici, ovaj rad dokaz je da biološki koncepti i mehanizmi, ako se koriste ispravno, mogu biti moćno oruđe za rješavanje optimizacijskih problema.

LITERATURA

[1] Lilienthal A., Duckett T., Experimental Analysis of Gas-Sensitive Braitenberg Vehicles, W.-Schickard-Inst. for Comp. Science, University of Tubingen.

[2] Palensky B., Barnard E., A Brief Overview of Artificial Intelligence Focusing on Computational Models of Emotions, *5th IEEE International Conference, Industrial Informatics*, 2007.

[3] Braitenberg V., Vehicles: Experiments in Synthetic Psychology, *Duke University Press*, 1986.

[4] Browning B., Wyeth G., Neural Systems For Integrating Robot Behaviours, University of Queensland Computer Science and Electrical Engineering Department, 1998.

[5] Ćurković P., Predavanja iz kolegija umjetna inteligencija, FSB, 2017.

[6] Mitchell M., Complexity: A Guided Tour, *Oxford University Press*, 2009.

[7] https://en.wikipedia.org/wiki/Fitness_proportionate_selection, 13.9.2022.

[8] Bešić S., Identifikacija dinamike laboratorijske grijalice zraka primjenom umjetne neuronske mreže, završni rad, FSB, 2019.

[9] Majetić D., Predavanja iz kolegija Neuronske mreže, FSB, 2017.

[10] Pyo S., Lee J., Cha M., Jang H.,Predictability of machine learning techniques to forecast the trends of market index prices: Hypothesis testing for the Korean stock markets, *PLOS ONE Journal*, 2017.

[11] https://machine-learning.paperspace.com/wiki/activation-function, 14.9.2022.

[12] Medovka K., Primjena evolucijskog algoritma za učenje neuronske mreže, diplomski rad, FSB, 2022.

[13] Gomez F., Miikkulainen R., Solving non-Markovian control tasks with neuroevolution, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann*, 1999., 1356-1361

[14] Stanley K. O., Evolving Neural Networks through Augmenting Topologies, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, *MIT Press*, 2002.

[15] Chen D. et al., Constructive learning of recurrent neural networks, *Computational Learning Theory and Natural Learning Systems III., MIT Press*, 1993.

[16] Gruau F., Genetic synthesis of modular neural networks, *Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann*, 1993., 318–325

[17] Dasgupta D., McGregor D., Designing application-specific neural networks using the structured genetic algorithm, *Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, *IEEE Press*, 1992., 87–96

[18] Pujol J.C.F., Poli,R. Evolving the topology and the weights of neural networks using dual representation, *Special Issue on Evolutionary Learning of the Applied Intelligence Journal*, 1998., 73-84

[19] Angeline P.J., Saunders G.M., Pollack J.B., An evolutionary algorithm that constructs reccurent neural networks, *IEEE Transactions on Neural Networks*, 1993., 54-65

[20] https://www.youtube.com/watch?v=VMQOa4-rVxE, 8.9.2022.

[21] https://github.com/pygame/, 30.8.2022

[22] https://neat-python.readthedocs.io/en/latest/, 7.9.2022.

[23] https://github.com/codewmax/DriveAI, 6.9.2022.

[24] https://graphviz.org/, 12.9.2022.

PRILOG 1: PYTHON KOD ZA SLIJEĐENJE RAVNE LINIJE

```
import pygame
import os
import neat
import math
import sys
import time
import visualize

pygame.init()

screen_height = 750
screen_width = 1500
screen = pygame.display.set_mode((screen_width,screen_height))

robot =
pygame.image.load(os.path.join("slike_zavrsni","robot_zavrsni.png")).conver
t_alpha() #100x81
staza =
pygame.image.load(os.path.join("slike_zavrsni","staza_zavrsni.png")).conver
t()
x_spawn = 100
y_spawn = 355

class Robot(pygame.sprite.Sprite):

    def __init__(self):
        super().__init__()
        self.robot_image = robot  #surface (robot)
        self.image = self.robot_image
        self.rect = self.image.get_rect(center=(x_spawn,y_spawn)) #kvadrat
oko objekta robota
        self.position = self.rect.center #initial position
        self.velocity = pygame.math.Vector2(1,0) #vektor brzine
        self.angle = 0 #inicijalni kut
        self.rotation = 5 #inkrement rotacije
        self.state = 0 #naprijed ili nazad
        self.direction = 0 #lijevo ili desno
        self.alive = True #zabio se u zid ili nije
        self.radars = [] #lista pohrane podataka senzora

    def update(self):
        #update funkcija
        self.radars.clear()
        self.drive()
        self.rotate()
        for radar_angle in (-30,30):
            self.radar(radar_angle)
        self.collision()
        self.data()

    def drive(self):
        #funkcija pomoću koje se robot kreće naprijed-nazad
        if self.state == 1:
            self.position += self.velocity * 6
        if self.state == -1:
            self.position -= self.velocity * 6
        self.rect.center = self.position
```

```
    def rotate(self):
        #funkcija pomoću koje se robot kreće lijevo-desno
        if self.direction == 1:
            self.angle -= self.rotation

        if self.direction == -1:
            self.angle += self.rotation

        self.velocity = pygame.math.Vector2(1,0)
        self.velocity.rotate_ip(-self.angle)
        self.rect = self.image.get_rect(center=self.rect.center)
        self.image =
pygame.transform.rotozoom(self.robot_image,self.angle,1)

    def radar(self,radar_angle):
        #funkcija za dobivanje podataka senzora
        length = 0
        x = int(self.rect.center[0])
        y = int(self.rect.center[1])
        try:
            while not screen.get_at((x,y)) == pygame.Color(128,0,0,255) and
length < 350: #boja zida
                length += 1
                x = int(self.rect.center[0] +
math.cos(math.radians(self.angle+radar_angle))*length)
                y = int(self.rect.center[1] -
math.sin(math.radians(self.angle+radar_angle))*length)
        except IndexError: #napravljeno radi toga da izbriše jedinku koja
je izašla iz ekrana
            self.alive = False

        #crtanje
        pygame.draw.line(screen,(255,255,255,255),self.rect.center,(x,y),1)
        pygame.draw.circle(screen,(0,255,0,0),(x,y),3)

        #udaljenost do zida
        dist = int(math.sqrt(math.pow(self.rect.center[0] - x,2)
                             + math.pow(self.rect.center[1] - y, 2)))
        self.radars.append([radar_angle,dist])

    def data(self):
        #funkcija koja vraća očitanje udaljenosti senzora
        input = [0,0] #lista udaljenosti od zida
        for i, radar in enumerate(self.radars): #i = count, radar = član
liste
            input[i] = int(radar[1])
        return input

    def collision(self):
        #funkcija koja provjerava koliziju robota sa zidom
        length = 58
        back_length = 60
        collision_point_1 = [int(self.rect.center[0] +
math.cos(math.radians(self.angle+30))*length),
                int(self.rect.center[1] -
math.sin(math.radians(self.angle+30))*length)]
        collision_point_2 =[int(self.rect.center[0] +
math.cos(math.radians(self.angle-30))*length),
                int(self.rect.center[1] - math.sin(math.radians(self.angle-
30))*length)]
        collision_point_3 = [int(self.rect.center[0] +
```

```
math.cos(math.radians(self.angle+40))*(-back_length)),
                int(self.rect.center[1] -
math.sin(math.radians(self.angle+40))*(-back_length))]
        collision_point_4 = [int(self.rect.center[0] +
math.cos(math.radians(self.angle-40))*(-back_length)),
                int(self.rect.center[1] - math.sin(math.radians(self.angle-
40))*(-back_length))]
        collision_points =
[collision_point_1,collision_point_2,collision_point_3,collision_point_4]

        #brisanje jedinki pri koliziji
        for x in collision_points:
            try:
                if screen.get_at(x) == pygame.Color(128,0,0,255):
                    self.alive = False
            except IndexError: #napravljeno radi toga da izbriše jedinku
koja je izašla iz ekrana
                self.alive = False


        #crtanje točaka kolizije
        for x in collision_points:
            pygame.draw.circle(screen,(0,255,255,0),x,4)

    def distance_x(self):
        #funkcija koja vraća udaljenost jedinke od početne točke (x smjer)
        distance_travelled_x = int(self.rect.center[0] - x_spawn)
        return distance_travelled_x

def kill(index):
    #funkcija koja briše jedinku pri koliziji
    robots.pop(index)
    ge.pop(index)
    nets.pop(index)

def eval_genomes(genomes,config):
    global robots, ge, nets

    robots = []
    ge = []
    nets = []

    #pridodaje svakoj jedinki neuronsku mrežu
    for genome_id, genome in genomes:
        robots.append(pygame.sprite.GroupSingle(Robot()))
        ge.append(genome)
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        nets.append(net)
        genome.fitness = 0

    #mjerenje vremena, (uvjet prekida generacije)
    time_start = time.time()

    #limit na 30fps
    clock = pygame.time.Clock()

    #cosmetics
    pygame.display.set_caption('Treniranje ravne linije')
    icon =
pygame.image.load(os.path.join("slike_zavrsni",'robot_zavrsni_icon.png'))
    pygame.display.set_icon(icon)
```

```
    #gameloop
    run = True
    while run:
        time_current = time.time()
        time_passed = time_current - time_start
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
        screen.blit(staza,(0,0)) #crtanje pozadine na pygame prozor

        #prekid generacije
        if len(robots) == 0:
            break

        #uvjet brisanja jedinki iz generacije
        for i, robot in enumerate(robots):
            if not robot.sprite.alive or time_passed > 12:
                #ge[i].fitness = robot.sprite.measure_fitness()
                ge[i].fitness = robot.sprite.distance_x()
                #ge[i].fitness = robot.sprite.measure_fitness_2()
                kill(i)

        #aktivacijski uvjeti za izlaze mreže
        for i, robot in enumerate(robots):
            output = nets[i].activate(robot.sprite.data())
            if output[0] < 0.4:
                robot.sprite.direction = 1
            if output[0] > 0.6:
                robot.sprite.direction = -1
            if output[1] > 0.6:
                robot.sprite.state = 1
            if output[1] < 0.4:
                robot.sprite.state = -1

        #crtanje populacije
        for robot in robots:
            robot.draw(screen)
            robot.update()
        pygame.display.update()
        #clock.tick(30)

#setup NEAT

def run(config_path):
    global pop

    #NEAT config
    config = neat.config.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_path
    )

    #create population
    pop = neat.Population(config)

    #prikaz statistike NEAT librarija
```

```
    pop.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    pop.add_reporter(stats)

    winner = pop.run(eval_genomes, 21)

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

    node_names = {-1: 'senzor 1', -2: 'senzor 2', 0: 'lijevo-desno',1:
'naprijed-nazad'}
    #plotanje mean i max fitnessa pomoću visualize koda
    visualize.plot_stats(stats, view=True)
    visualize.plot_species(stats, view=True)
    visualize.draw_net(config, winner, view=True,node_names=node_names)


if __name__ == '__main__':
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir,'config_1.txt')
    run(config_path)
```

PRILOG 2: *CONFIG* TEKSTUALNA DATOTEKA ZA 1. PROBLEM

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 1350
pop_size              = 30
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 2
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0
```

```
# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 2.0

[DefaultStagnation]
species_fitness_func = mean
max_stagnation       = 20
species_elitism      = 1

[DefaultReproduction]
elitism            = 1
survival_threshold = 0.2
min_species_size   = 1
```

PRILOG 3: PYTHON KOD ZA PRAĆENJE IZVORA SVJETLOSTI

```
import pygame
import os
import neat
import math
import sys
import visualize

pygame.init()

screen_height = 750
screen_width = 1200
screen = pygame.display.set_mode((screen_width,screen_height))

robot =
pygame.image.load(os.path.join("slike_zavrsni","robot_zavrsni.png")).conver
t_alpha() #100x81
pozadina =
pygame.image.load(os.path.join("slike_zavrsni","pozadina_2.png")).convert()


class Robot(pygame.sprite.Sprite):

    def __init__(self):
        super().__init__()
        self.robot_image = robot  #surface (robot)
        self.image = self.robot_image
        self.rect = self.image.get_rect(center = (600,375)) #crtanje
kvadrata oko objekta robota
        self.position = self.rect.center #initial position
        self.velocity = pygame.math.Vector2(1,0)
        self.angle = 0
        self.rotation = 5
        self.state = 0
        self.direction = 0
        self.alive = True
        self.radars = []
        self.distance_sum = 0
        self.is_light = 0
        self.fitness = 0

    def update(self):
        self.radars.clear()
        self.drive()
        self.rotate()
        for radar_angle in (-30,-15,0,15,30):
            self.radar(radar_angle)
        self.collision()
        self.data()

    def drive(self):
        if self.state == 1:
            self.position += self.velocity * 6
        if self.state == -1:
            self.position -= self.velocity * 6
        self.rect.center = self.position

    def rotate(self):
        if self.direction == 1:
```

```
            self.angle -= self.rotation
        if self.direction == -1:
            self.angle += self.rotation
        self.velocity = pygame.math.Vector2(1,0)
        self.velocity.rotate_ip(-self.angle)
        self.rect = self.image.get_rect(center=self.rect.center)
        self.image =
pygame.transform.rotozoom(self.robot_image,self.angle,1)

    def radar(self,radar_angle):
        length = 0
        x = int(self.rect.center[0])
        y = int(self.rect.center[1])
        try:
            while not screen.get_at((x,y)) == pygame.Color(122,86,14,255)
and not screen.get_at((x,y)) == pygame.Color(221,251,9,255): #boja zida ili
svijetla
                length += 1
                x = int(self.rect.center[0] +
math.cos(math.radians(self.angle+radar_angle))*length)
                y = int(self.rect.center[1] -
math.sin(math.radians(self.angle+radar_angle))*length)
        except IndexError:
            self.alive = False
        try:
            if screen.get_at((x,y)) == pygame.Color(221,251,9,255):
                self.is_light = 1
            else:
                self.is_light = 0
        except IndexError:
            self.alive = False

        #crtanje
        pygame.draw.line(screen,(255,255,255,255),self.rect.center,(x,y),1)
        pygame.draw.circle(screen,(0,255,0,0),(x,y),3)

        self.radars.append([radar_angle, self.is_light])

    def data(self):
        input = [0, 0, 0, 0, 0]  # lista udaljenosti od svjetla
        for i, radar in enumerate(self.radars):  # i = count, radar = član
liste
            input[i] = int(radar[1])
        return input


    def collision(self):
        length = 55 #duljina robota
        back_length = 60
        collision_point_1 = [int(self.rect.center[0] +
math.cos(math.radians(self.angle+30))*length),
                int(self.rect.center[1] -
math.sin(math.radians(self.angle+30))*length)]
        collision_point_2 =[int(self.rect.center[0] +
math.cos(math.radians(self.angle-30))*length),
                int(self.rect.center[1] - math.sin(math.radians(self.angle-
30))*length)]
        collision_point_3 = [int(self.rect.center[0] +
math.cos(math.radians(self.angle+40))*(-back_length)),
                int(self.rect.center[1] -
math.sin(math.radians(self.angle+40))*(-back_length))]
```

```
        collision_point_4 = [int(self.rect.center[0] +
math.cos(math.radians(self.angle-40))*(-back_length)),
                int(self.rect.center[1] - math.sin(math.radians(self.angle-
40))*(-back_length))]
        collision_points =
[collision_point_1,collision_point_2,collision_point_3,collision_point_4]
        #collision
        for x in collision_points:
            try:
                if screen.get_at(x) == pygame.Color(122,86,14,255):
                    self.alive = False
            except IndexError:
                self.alive = False
        #crtanje
        for x in collision_points:
            pygame.draw.circle(screen,(0,255,255,0),x,4)

    def measure_fitness(self,a):
        current_distance = int(math.sqrt(math.pow(self.rect.center[0] -
a[0],2)
                                + math.pow(self.rect.center[1] - a[1], 2)))
        if self.is_light == 1: #bodove dobiva samo ako ima svjetlo u
vidokrugu
            if current_distance < 250:
                self.fitness += 1
            if current_distance < 50:
                self.fitness += 10
        return self.fitness


def kill(index):
    #funkcija koja briše jedinku pri koliziji
    robots.pop(index)
    ge.pop(index)
    nets.pop(index)

def reset(light):
    light = pygame.sprite.GroupSingle(Svijetlo()) #inicijalizacija objekta
pri novoj generaciji

class Svijetlo(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.light = pygame.draw.circle(screen,(221,251,9),(200,200),20)
        #(100,650) za light_square
        #(200,200) za light_line
        #(600,200) za light_mnogokut
        #
        self.pos = self.light.center
        #initial direction
        self.increment_direction = pygame.math.Vector2(0,1)
        #(1,0) za light_square
        #(0,1) za light_line
        #(1,0) za light_mnogokut
        #
        #initial increment
        self.increment_value = 6
        self.light_travelled = 0

    def increment_light_square(self):
        self.pos += self.increment_direction * self.increment_value
```

```
        #opisuje putanju svjetla
        if self.pos[0] > 1100:
            self.pos[0] = 1100
            self.increment_direction[0] = 0
            self.increment_direction[1] = -1
        if self.pos[1] < 100:
            self.pos[1] = 100
            self.increment_direction[0] = -1
            self.increment_direction[1] = 0
        if self.pos[0] < 100:
            self.pos[0] = 100
            self.increment_direction[0] = 0
            self.increment_direction[1] = 1
        if self.pos[1] > 680:
            self.pos[1] = 680
            self.increment_direction[0] = 1
            self.increment_direction[1] = 0
        self.light_travelled += self.increment_value

    def increment_light_line(self):
        self.pos += self.increment_direction * self.increment_value
        #opisuje putanju svjetla
        if self.pos[1] > 550:
            self.pos[1] = 550
            self.increment_direction[0] = 0
            self.increment_direction[1] = -1
        if self.pos[1] < 200:
            self.pos[1] = 200
            self.increment_direction[0] = 0
            self.increment_direction[1] = 1
        self.light_travelled += self.increment_value

    def increment_light_mnogokut(self):
        self.pos += self.increment_direction * self.increment_value
        #opisuje putanju svjetla
        if self.pos[0] > 800 and self.pos[1] == 200:
            self.pos = [800,200]
            self.increment_direction[0] = 1
            self.increment_direction[1] = 1
        if self.pos[0] > 975 and self.pos[1] > 375:
            self.pos = [975,375]
            self.increment_direction[0] = -1
            self.increment_direction[1] = 1
        if self.pos[0] < 800 and self.pos[1] > 550:
            self.pos = [800,550]
            self.increment_direction[0] = -1
            self.increment_direction[1] = 0
        if self.pos[0] < 400 and self.pos[1] == 550:
            self.pos = [400,550]
            self.increment_direction[0] = -1
            self.increment_direction[1] = -1
        if self.pos[0] < 225 and self.pos[1] < 375:
            self.pos = [225,375]
            self.increment_direction[0] = 1
            self.increment_direction[1] = -1
        if self.pos[0] > 400 and self.pos[1] < 200:
            self.pos = [400,200]
            self.increment_direction[0] = 1
            self.increment_direction[1] = 0
        self.light_travelled += self.increment_value
```

```
    def draw(self):
        self.light = pygame.draw.circle(screen, (221, 251, 9), self.pos,
20)

def eval_genomes(genomes,config):
    global robots, ge, nets

    robots = []
    ge = []
    nets = []
    light = pygame.sprite.GroupSingle(Svijetlo())
    clock = pygame.time.Clock()

    for genome_id, genome in genomes:
        robots.append(pygame.sprite.GroupSingle(Robot()))
        ge.append(genome)
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        nets.append(net)
        genome.fitness = 0

    #cosmetics
    pygame.display.set_caption('Treniranje praćenja svijetla')
    icon =
pygame.image.load(os.path.join("slike_zavrsni",'robot_zavrsni_icon.png'))
    pygame.display.set_icon(icon)
    #gameloop
    run = True
    while run:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
        screen.blit(pozadina,(0,0)) #crtanje pozadine na pygame prozor

        if len(robots) == 0:
            break

        #uvjet brisanja jedinki iz generacije
        for i, robot in enumerate(robots):
            ge[i].fitness += robot.sprite.measure_fitness(light.sprite.pos)
            if not robot.sprite.alive or light.sprite.light_travelled >
2100:
                #3150 za light_square
                #1400 za light_line
                #1535 za light_mnogokut
                kill(i)
                if light.sprite.light_travelled > 2100:
                    reset(light)

        #aktivacijski uvjeti za izlaze mreže
        for i, robot in enumerate(robots):
            output = nets[i].activate(robot.sprite.data())
            if output[0] > 0.4:
                robot.sprite.direction = 1
            if output[1] > 0.4:
                robot.sprite.direction = -1
            if output[2] > 0.4:
                robot.sprite.state = 1
            if output[3] > 0.4:
                robot.sprite.state = -1
```

```
        #odkomentirati željenu putanju
        light.sprite.increment_light_square()
        #light.sprite.increment_light_line()
        #light.sprite.increment_light_mnogokut()

        for robot in robots:
            robot.draw(screen)
            light.sprite.draw()
            robot.update()
        pygame.display.update()
        clock.tick(30)

def run(config_path):
    global pop

    #NEAT config
    config = neat.config.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_path
    )

    #create population
    pop = neat.Population(config)

    #prikaz statistike NEAT librarija
    pop.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    pop.add_reporter(stats)

    winner = pop.run(eval_genomes, 201)

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

    node_names = {-1: 'senzor 1', -2: 'senzor 2', -3: 'senzor 3', -4:
'senzor 4',-5: 'senzor 5', 0: 'lijevo', 1: 'desno',2:'naprijed',3:'nazad'}
    # plotanje mean i max fitnessa pomoću visualize koda
    visualize.plot_stats(stats, view=True)
    visualize.plot_species(stats, view=True)
    visualize.draw_net(config, winner, view=True, node_names=node_names)

if __name__ == '__main__':
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config_2.txt')
    run(config_path)
```

PRILOG 4: *CONFIG* TEKSTUALNA DATOTEKA ZA 2. PROBLEM

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 1000000
pop_size              = 30
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 5
num_outputs             = 4

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0
```

```
# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 2

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 10
species_elitism      = 1

[DefaultReproduction]
elitism            = 3
survival_threshold = 0.2
min_species_size = 1
```