|  |  |
| --- | --- |
| Flappy bird ai  Implementacija genetskog algoritma (https://github.com/zlatkovnik/flappy-ai) | Nikola Zlatkov 16593 Aleksa Antić 16472 Stefan Đurić 16574  Profesor: Leonid Stoimenov |

Contents

[Izveštaj 1](#_Toc62990492)

[Opis problema 1](#_Toc62990493)

[Pregled i kratak opis tehnika i algoritama VI koje se mogu koristiti za rešavanje navedenog problema 1](#_Toc62990494)

[Nešto o našoj odabranoj tehnici 2](#_Toc62990495)

[Formulacija problema na način kako to odgovara izabranoj tehnici veštačke inteligencije 3](#_Toc62990496)

[Kratak opis rešenja 4](#_Toc62990497)

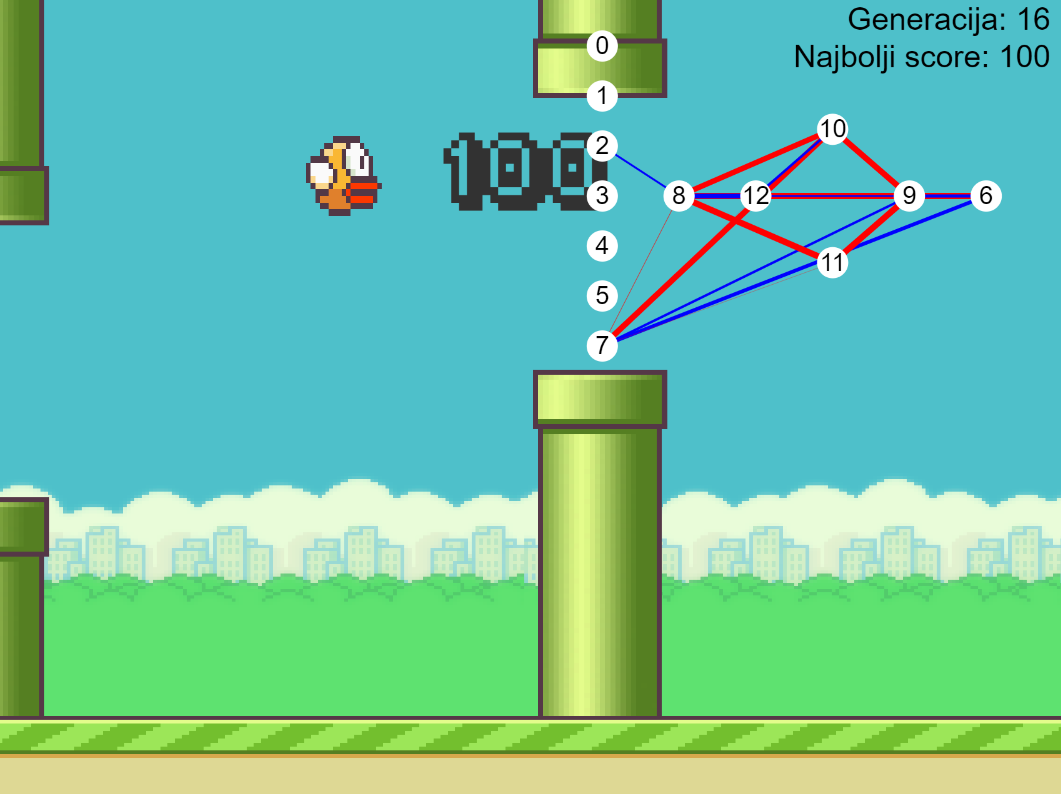
[Implementacija 5](#_Toc62990498)

[Klase: 5](#_Toc62990499)

[Razvoj 7](#_Toc62990500)

[Literatura 9](#_Toc62990501)

Link za github: <https://github.com/zlatkovnik/flappy-ai>



# Izveštaj

### Opis problema

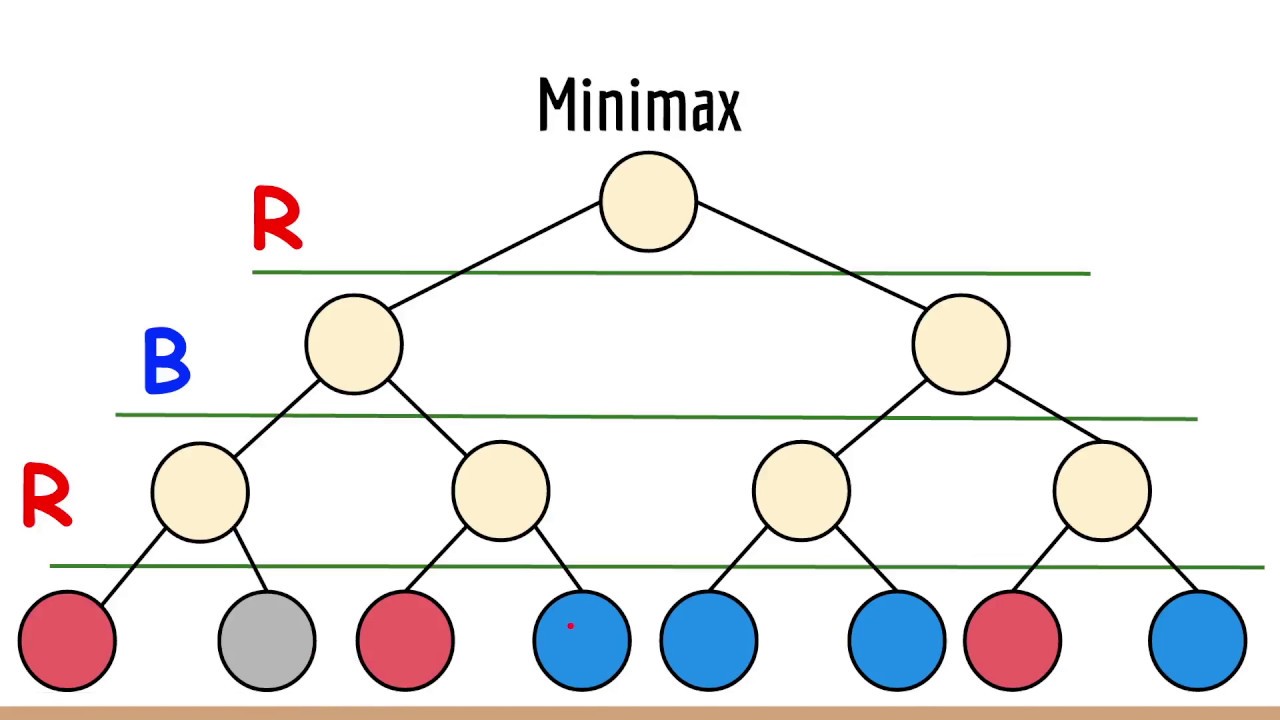
U svakoj igri bila ona društvena, kompjuterska ili sportska rađaju se određene strategije i taktike. Često je teško utvrditi koja je taktika dobra, a koja ne, i kroz istoriju su se taktike menjale ili poboljšavale, ali dosta sporo, a razlog je to što je teško uočiti šta su uzroci dobre ili loše taktike jer je ljudski faktor nepredvidiv.

Genetski algoritam može da razreši većinu ovih nedoumica jer možemo proći kroz mnogo više varijacija neke taktike ili više njih i to u kontrolisanim uslovima time zaključiti šta je bitno za svaku taktiku i koji je, ako ga ima, najbolji pristup nekoj igri. Naša ideja je da naučimo kako funkcioniše ovaj algoritam i vidimo njegov značaj. Zato smo za primer uzeli igru Flappybird koja je u osnovi jednostavna ali je bila popularna po svojoj težini toliko da su se javile glasine o kraju igre posle 100 poena koji su za većinu ostali nedostižni. Mi želimo da vidimo kako će genetski algoritam videti ovu igru i kakvo će razmišljanje razviti.

### Pregled i kratak opis tehnika i algoritama VI koje se mogu koristiti za rešavanje navedenog problema

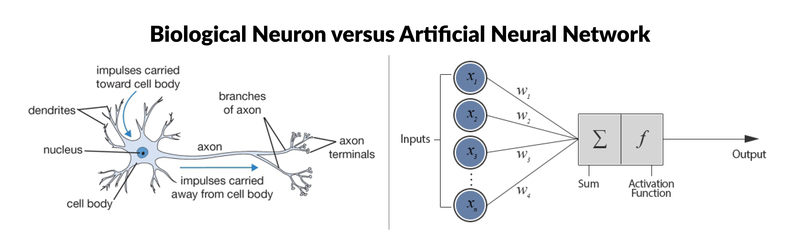
Za naš konkretan problem postoji mnogo različitih metoda za njegovo rešavanje. Moguće je da samo napišemo jedan if blok koji će relativno savršeno da radi, ali to nije zanimljivo ni edukativno, a pritom je plitko rešenje.

Sledeća metoda može biti Minimax algoritam sa pomoćnim funkcijama za odsecanje i transpozicionim bazama. Naime iz date pozicije u vremenu, odnosno frejmu i prostoru granaju se dva paralelna univerzuma, jedan u kome ptica skače, a drugi u kome ne preduzima ništa. Na taj način, pošto je broj mogućih poteza dva, može se doći do ogromnih dubina, i igra može biti savršeno odigrana. Ovo rešenje je odlično, ali nije dovoljno edukativno ni interesantno, jer agent ima idealan pogled u budućnost, što u realnom svetu nije izvodljivo, takođe da je igra komplikovanija ovaj algoritam bi zahtevao kompjutersku moć koju ne poseduje niko (savršeno odigrana partija šaha, sa dubinom koja teži beskonačnosti).



[slika minimax stabla odlučivanja, stablo je binarno jer ima samo dve mogućnosti u svakom momentu, to je da poleti ili da ne uradi ništa]

Mi želimo da simuliramo nešto što je primerenije za realan život i razmišljanje živog bića. Odnosno poenta je da agent vidi ono što bi videlo živo biće i na osnovu toga donese odluku o svom sledećem potezu, korišćenjem neuronske mreže. Na taj način naš agent donosi odluke prirodnije našem načinu delovanja.



[Kako se razlikuje biološki neuron u odnosu na mašinski]

### Nešto o našoj odabranoj tehnici

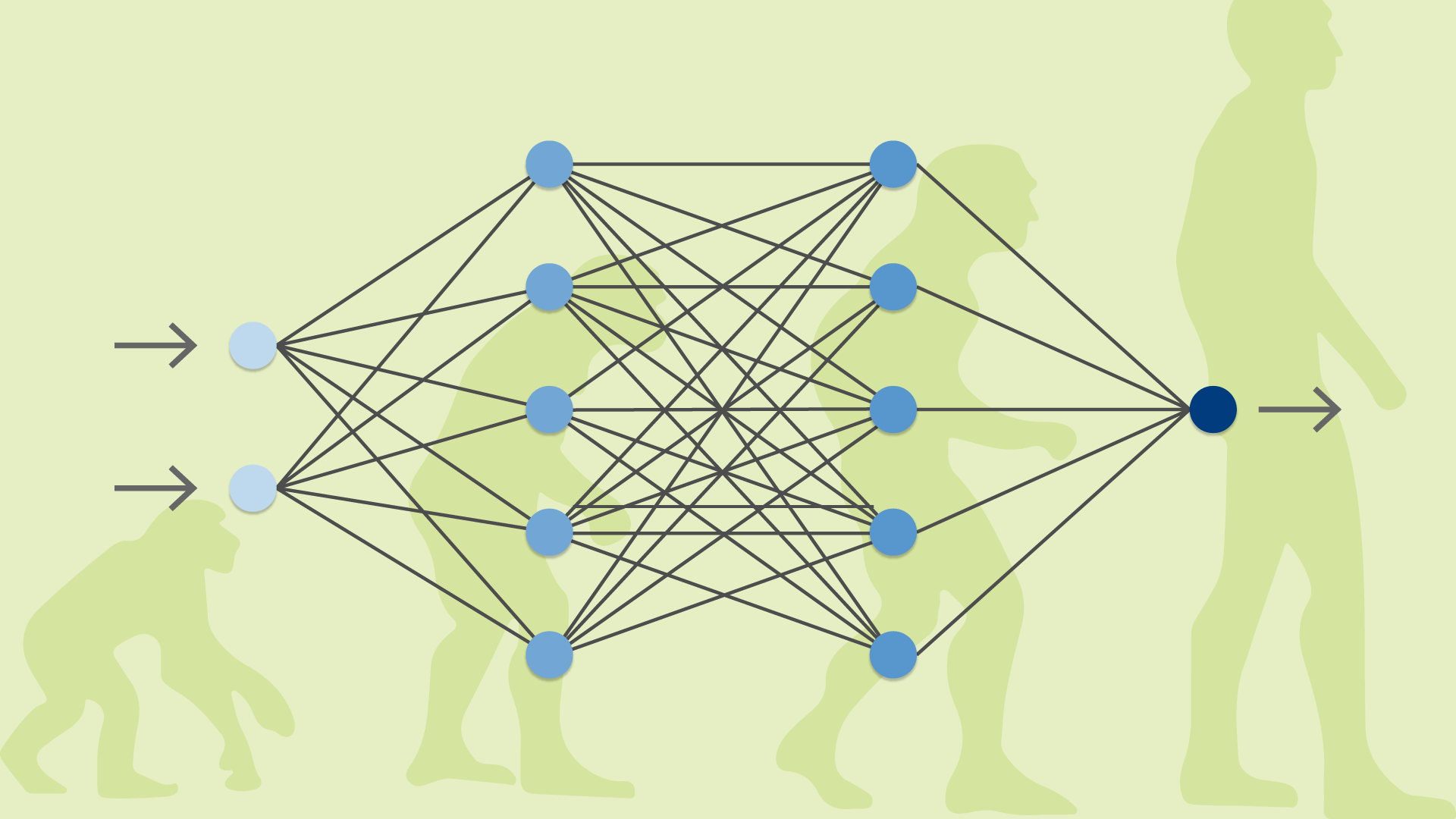
Kratkim pregledom tipova neuronskih mreža utvrdili smo da je najbolje koristiti Feedforward Neural Network tačnije NEAT algoritam tj. “Neuroevolution of augmenting topologies” koji je inspirisan evolucijom. NEAT je neuroevolutivna tehnika razvijena od strane Ken Stanley–a, u 2002. godini na Teksas univerzitetu u Astinu.

NEAT algoritam je osmišljen tako da evoluira iz generacije u generaciju, u početnom stanju mu prosleđujemo ulaze koji su parametri za koje smatramo da su potencijalno bitni da bi zaključio šta treba u kom trenutku da uradi. Ključna stvar je to što je programer ograničen u uticaju na odlučivanje neuronske mreže, odnosno kako i u kojoj meri će se koristiti ulazi za donošenje odluke. Ti ulazi su direktno ili indirektno vezani za izlaze koji predstavljaju odluku koju je algoritam doneo, ili u kom trenutku treba da je donese.

Struktura neurološke mreže se sastoji od čvorova, koji su podeljeni u slojeve i potpuno su povezani između dva susedna čvora. Gde su prvi sloj ulazni parametri mreže, koji sadrže vrednosti od -1 do 1, a zadnji sloj izlazni parametri mreže čije se vrednosti računaju na osnovu čvorova koji su povezani na njih, težine veze i aktivacione funkcije za računanje izlaza. Između prvog i zadnjeg sloja postoje tajni slojevi, koji se dinamički generišu na osnovu potrebe.

NEAT evoluira tako što menja težinu veza i strukturu mreže kako bi probao da nađe ravnotežu između performansi rešenja i njihove raznovrsnosti.

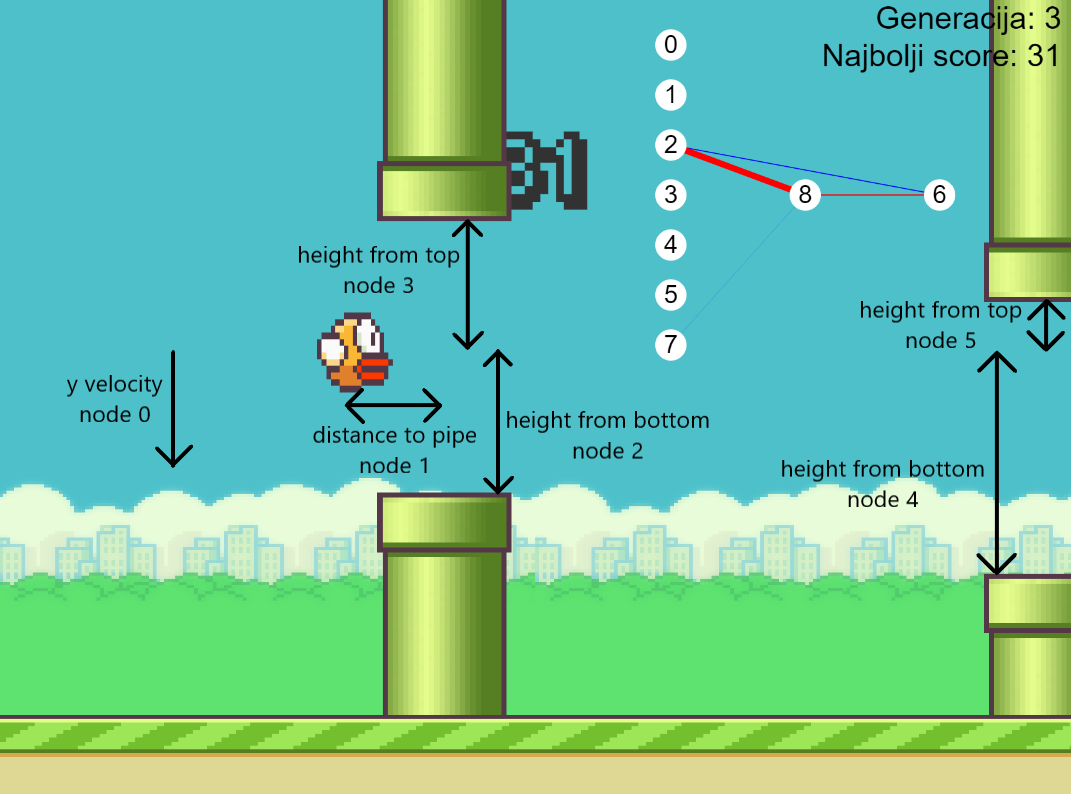
Zasniva se na tri ključne tehnike: praćenje gena kroz istoriju da bi se omogućilo ukrštanje genoma, primena evolucije vrsta jedinki (vrste su slični genomi) da bi se sačuvala inovacija, i razvoj genoma počevši od jednostavne strukture u kompleksnije, odnosno ulaza i izlaza.



### Formulacija problema na način kako to odgovara izabranoj tehnici veštačke inteligencije

Pošto koristimo tehniku koja odgovara tehnikama koju koriste živa bića, naš agent vidi samo ono što vidi i čovek koji igra igru. Konkretno naš agent “vidi” šest stvari:

* Brzina pada, odnosno brzina uzletanja
* Udaljenost od najbliže šipke
* Visina u odnosu na donju najbližu šipku
* Visina u odnosu na gornju najbližu šipku
* Visina u odnosu na donju dalju šipku
* Visina u odnosu na gornju dalju šipku



A jedina odluka koju može da donese je da poleti ili da ne preduzme ništa. Naš početni genom ima 6 ulazna čvora sa dodatnim “sklonost” čvorom i jedan izlazni. Težine veza između čvorova su nasumične na početku, jer je mreža jednostavna i vrlo brzo može doći do optimalnog rešenja.

Naš agent se nagrađuje što duže ostane u životu, i kad prođe kroz što više prepreka. Što ima bolje performanse, veća je šansa da njegova vrsta opstane. Na osnovu njihovih performansi i nasumičnih spoljašnjih uticaja mi ih ukrštamo i mutiramo. Kada se agent sudari sa cevkom ili zemljom on se smatra mrtvim, ali se njegova ocena čuva kako bi se kasnije utvrdilo da li je vredan da širi svoje gene ili da bude pregažen od strane boljih agenata i tako ostane u istoriji.

### Kratak opis rešenja

Kao tehnologiju za realizaciju ovog projekta koristili smo p5 JavaScript biblioteku za grafiku i osvežavanje frejmova. Na GitHub-u smo našli templejt za implementaciju NEAT algoritma koji smo modifikovali prema našim potrebama. (<https://github.com/GabrielTavernini/NeatJS>)

Za pokretanje je dovoljno da se pokrene server koji hostuje index.html, biblioteke su uključene u projekat.

Koristili smo JavaScript kao izborni jezik zbog jednostavnosti kreiranja grafike (da ne bismo gubili vreme oko stvari koje nisu vezane za veštačku inteligenciju), prenosivosti odnosno cross platform mogućnosti, brzini pisanja koda. Svesni smo da se gube performanse, jer nismo koristili c ili c++, ali performanse našeg rešenja su više nego zadovoljavajuće. Naime sa 3000 agenata po populaciji istovremeno ne opadaju performanse, sa 6000 već je vidljivo, ali zanemarljivo, a sa 10000 se gubi prilično na perfomansama, ali je prihvatljivo. Iz rezultata testiranja zaključili smo da je 1000 agenata po populaciji dovoljno da se brzo dođe do optimalnog rezulata.

# Implementacija

### Klase:

Node (Čvor)

Klasa sadrži identifikator, sumu inputa (kešira input), vrednost outputa (vrednost koju šaljemo dalje kroz mrežu), konekcije (između this i drugog čvora) i sloj u mreži.

Čvor sadrži metodu ***engage*** za aktivaciju koja prosleđuje izračunatu vrednost svim output čvorovima koji su povezani.

Za računanje output vrednosti koristi se ***sigmoid*** funkcija .

ConnectionGene (Konekcija | Gen)

Konekcija sadži izvorni i odredišni čvor koje povezuje, težinu veze, da li je omogućen, broj inovacije koji omogućava da se utvrdi da li su dva gena ista.

Metodu koja mutira težinu veze ***mutateWeight***. Postoji 10% šansa da težina bude kompletno nova vrednost, 90% da se sledeća vrednost minimalno promeni u odnosu na sadašnju.

Genome (Genom)

Genom sadrži sve čvorove u mreži, sve konekcije, broj input čvora, broj output čvora i broj slojeva.

Sadrži metodu ***connectNodes*** koja kompletno povezuje sve čvorove.

Metodu ***feedForward*** koja aktivira mrežu na osnovu inputa i računa vrednosti output čvorova, u našem slučaju jedan čvor koji sadrži vrednost od -1 do 1.

Metodu ***generateNetwork*** koja generiše mrežu za agenta.

Sadrži metodu ***addNode*** koja dodaje novi čvor u mrežu. Prvo se nasumično bira gen koji povezuje dva čvora (čvor a i čvor b). Ta konekcija se gasi, čvor a se povezuje sa novim čvorom sa težinom 1, a novi čvor se povezuje sa čvorom b sa težinom ugašenog čvora. Nakon toga se bias čvor povezuje sa novim čvorom sa početnom težinom 0. Ako je sloj novog čvora jedan sloju output čvora stare konekcije, kreira se novi sloj. Čvorovi sa većim ili jednakim nivoom idu u sledeći sloj, i povećava se ukupan broj slojeva.

Metoda ***addConnection*** za dodavanje konekcije bira dva nasumična čvora i proverava da li su u istom sloju ili su već povezani, ako jesu onda traži nova dva čvora. Nakon toga ih povezuje od nižeg sloja ga višem.

Sadrži metodu ***getInnovationNumber*** koja vraća jedinstveni identifikator kod uparivanja dva identifikatora. ([https://en.wikipedia.org/wiki/Pairing\_function#Cantor\_pairing\_function](https://en.wikipedia.org/wiki/Pairing_function%23Cantor_pairing_function))

Sadrži metodu ***mutate*** za mutiranje genoma. Postoji 80% šansa da se mutira težina svih gena, sa njihovom funkcijom za mutiranje, 5% da se doda novi gen koji povezuje dva nasumična čvora, i 1% šansa da se doda novi čvor u mrežu.

Metoda ***crossover*** za ukrštanje dva genoma. Objekat koji zove metodu je „glavniji“ roditelj. Traži se isti gen kod oba roditelja i ako je taj gen kod bilo kog roditelja isključen, postoji 75% da se isključi i kod deteta. Takođe svaki gen ima istu šansu da se nasledi od bilo kog roditelja. Ako drugi roditelj nema neki gen prvog roditelja onda se ubacuje nepromenjen gen prvog roditelja u gene deteta. Dete je sličnije po strukturi „glavnijeg“ roditelja zbog toga.

Agent (Agent)

Agent nasleđuje klasu Player, koja sadrži x i y koordinatu agenta, brzinu pada, gravitaciju, reference na šipke, referencu na zemlju, brzinu leta, da li je živ i njegov score. Player sadži metodu u kojoj ažurira svoje stanje svaki frejm, metodu za iscrtavanje i metodu za poletanje.

Agent sadrži fitness, odnosno ocenu, šta agent vidi, šta je agent zaključio, koliko dugo je živ, najbolji score do sada, koja je generacija, broj inputa i outputa, i mozak (neuronsku mrežu).

Sadrži metodu ***look*** za prikupljanje informacija o okruženju, odnosno brzinu pada, udaljenost od bliže šipke, visina u odnosu na donju bližu šipku, visina u odnosu na gornju bližu šipku, visina u odnosu na donju dalju šipku, visina u odnosu na gornju dalju šipku. Te vrednosti se mapiraju od 0 do 1.

Metodu ***think*** za korišćenje inputa koji se prosleđuje neuronskoj mreži kako bi se došlo do outputa mreže, odnosno vrednosti za zaključivanje da li treba ptica u datom momentu da poleti ili ne. Ako je vrednost outputa veća od 0.6, zaključujemo da verovatno treba da poleti.

Metodu ***calculateFitness*** za računanje fitness-a koja koristi zbir score-a na kvadrat i dužine života podeljen sa 20.

Sadrži metodu ***crossover*** za ukrštanje dva agenta odnosno njihovih mreža.

Population (Populacija)

Populacija sadrži sve agente u trenutnoj generaciji, najbolji score, broj generacije, vrste agenata.

Sadrži metodu ***updateAlive*** za ažuriranje svih agenata u svakom frejmu. Odnosno svaki živ agent prvo prikuplja informacije iz okoline ***look***, donosi zaključak ***think*** uz pomoću neuronske mreže i ažurira svoje stanje na osnovu odluke ***update***.

Metodu ***naturalSelection*** za prirodnu selekciju. Prvo razvrstava generacije po vrsti ***speciate***, računa se fitness svih igrača ***calculateFitness***, sortiraju se vrste po fitness-u najboljeg agenta u vrsti ***sortSpecies***. Nakon toga se ubija polovina vrste sa lošijim fitness-om ***cullSpecies***, pa se ubijaju svi agenti koji nisu napredovali 15 generacija ***killStaleSpecies***, nakon toga se ubijaju sve ispod prosečne vrste ***killBadSpecies***. Šampion svake vrste se dodaje bez promena, vrsti se odobrava broj dece srazmerno sa fitness-om vrste. Ako je prosečan fitness vrste veći od prosečnog fitness-a svih agenata ta vrsta će imati više agenata nego pre i obrnutno. Na osnovu broja dozvoljene dece generišu se novi agenti, koji su potomci dva roditelja. Na kraju se generiše mreža za svakog agenta.

Metoda ***speciate*** koja deli igrače po vrstama na osnovu sličnosti njihovih gena, odnosno težina gena. U slučaju da neki agent ne pripada nijednoj vrsti pravi se nova vrsta u kojoj je on šampion.

Species (Vrsta)

Vrsta sadrži sve agente koje njoj pripadaju, najbolji fitness u vrsti, šampiona vrste, i koliko je prošlo generacija bez napretka.

Sadrži metodu ***sameSpecies*** koja utvrđuje da li neki agent pripada toj vrsti. Ova metoda računa broj gena koji nisu isti između dve neuronske mreže i razliku njihovih prosečnih težina gena. Na osnovu ta dva parametra utvrđuje da li su dovoljno slični.

Metodu ***sortSpecies*** koja sortira agente u vrsti po fitness-u korišćenjem quicksort-a. Ako vrsta nema članova briše se. Proverava se da li je nastao bolji igrač u odnosu na prethodnu generaciju, ako nije povećava se broj generacija bez napretka.

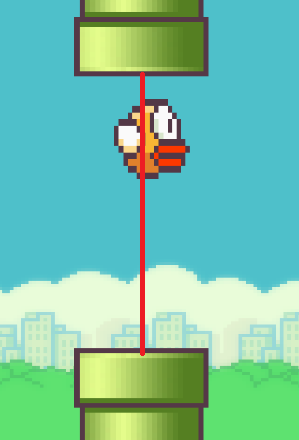
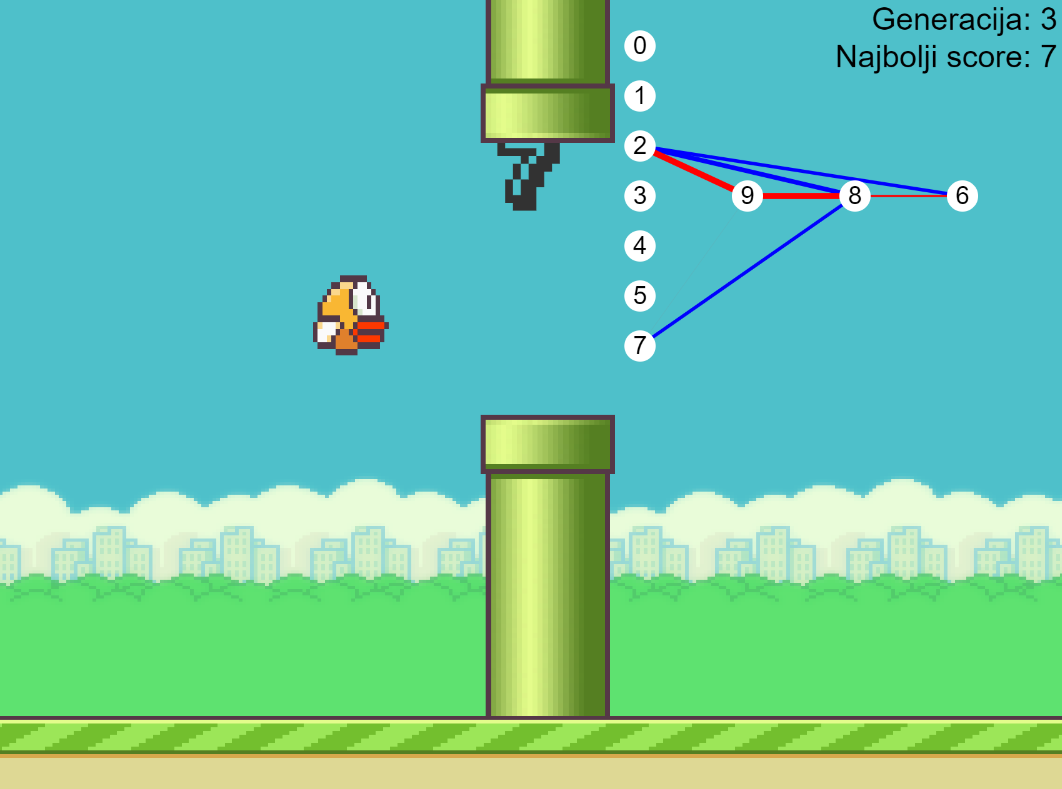
Sadrži metodu ***giveMeBaby*** koja pravi bebu na osnovu dva dobra roditelja u vrsti. Postoji 25% šansa da se ne vrši ukrštanje već da se samo klonira jedan roditelj. Inače se dete generiše na osnovu dva relativno nasumična roditelja. Kada se generiše dete, mutiraju se njegovi geni.

### Razvoj

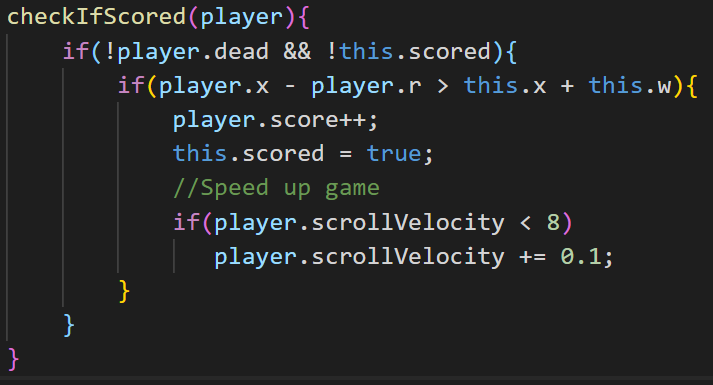
Početak je bio pravljenje funkcionalne replike Flappybird igre koju može da igra igrač, kasnije smo implementirali NEAT algoritam na templejtski način korišćenjem open source biblioteke NEAT.js kao smernica, odnosno tako da je modifikacija za različita rešenja jednostavna.

Klasu Player smo proširili atributima i metodama koje su potrebne algoritmu za evoluciju i donošenje odluka. Nakon toga imamo Klasu Populacija koja predstavlja sve agente u jednoj generaciji. Imali smo nedoumicu da li da svi agenti imaju svoje šipke ili da svi koriste iste šipke.

Prvobitno smo odlučili da svi agenti koriste iste šipke, tu je nastao problem da nije dolazilo do napretka iz generacije u generaciju, a neuronska mreža je izgledala zadovoljavajuće. Nakon sati debagiranja zaključili smo da šipka vidi samo jednog agenta i samo njemu povećava score, taj agent je uvek imao najbolju ocenu kakav god da mu je način razmišljanja bio. Zbog kompleksnosti popravljanja ovog problema prešli smo na metodu gde svaki agent ima svoju šipku jer pozicija šipke nije bitna u evoluciji.

Kada smo se odlučili da svaki agent ima svoju šipku naišli smo na sledeći problem da agenti pri iteracijama ne napreduje značajno čak i kada nam deluje da mu je neuronska mreža dobra. Razlog tome bio je način računanja score-a, zato što agent računa kao da je skroz prošao šipku i ako i dalje može da je zakači. Tu je problem bio njegov pogled na okruženje, zato što on score-ovanu šipku izbacuje iz pogleda i tada se desi da je pri izlazu dokači. Na slici je moment kada agent score-uje poen, u tom momentu on pogled prebacuje na sledeću šipku, nije svestan da može da udari u šipku ispod ili iznad koje se nalazi. Kako bi to rešili agent dobija poen za tu šipku nakon što je kompletno prođe.

Sada imamo sledeći problem, a to je da svaki agent generiše svoje šipke nasumično tako da svi agenti imaju različite visine rupe između šipki. To znači da ne možemo crtati na ekranu više od jednog agenta, svakako kada crtamo više agenata gubimo značajno na performansama tako da to nije neki problem, ali je problem da izaberemo najboljeg agenta u generaciji, jer ne znamo kada će da napravi grešku unapred. Tako da kada agent ugine prebacuje se kamera na sledećeg najboljeg na osnovu njegovih roditelja.

Nakon dosta probnih pokretanja videli smo razne evolucije. Nekad se dešavalo da agent preživi na sreću i onda on misli da je to što radi pametno, pa se to pamti kao loša navika u narednim generacijama. Za loše navike potrebno je dosta generacija da se isprave. Nekad se veoma brzo dođe do optimalnog rešenja. Koliko god da smo menjali ulaze za agente, na kraju svaka optimalna vrsta koristi najviše input koji je visina agenta od donje bliže šipke, što je i očekivano. U našem primeru taj čvor ima vrednost 2, što se vidi i sa slike. Mi kao igrači takođe najviše koristimo tu informaciju za donošenje odluke.

### Literatura

* NeatJS (<https://gabrieltavernini.github.io/NeatJS/>) – *Gabriel Tavernini*
* Cantor pairing function ([https://en.wikipedia.org/wiki/Pairing\_function#Cantor\_pairing\_function](https://en.wikipedia.org/wiki/Pairing_function%23Cantor_pairing_function))
* Sigmoid function (<https://en.wikipedia.org/wiki/Sigmoid_function>)
* p5js (<https://p5js.org/)>
* Sprites (<https://www.spriters-resource.com/mobile/flappybird/sheet/59894/>)