SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 495

OPTIMIZACIJA NEURONSKE MREŽE EVOLUCIJSKIM ALGORITMOM ZA IGRU ZMIJA

Luka Terzić

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 495

OPTIMIZACIJA NEURONSKE MREŽE EVOLUCIJSKIM ALGORITMOM ZA IGRU ZMIJA

Luka Terzić

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

ZAVRŠNI ZADATAK br. 495

Pristupnik:	**Luka Terzić (0036525175)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Optimizacija neuronske mreže evolucijskim algoritmom za igru zmija**

Opis zadatka:

Proučiti umjetne neuronske mreže te metode bazirane na evolucijskom računarstvu koje se mogu primijeniti za optimizaciju njihovih parametara. Istražiti primjenu i prilagodbu neuronskih mreža za igranje igre zmija. Odabrati i primijeniti prikladni evolucijski algoritam za optimizaciju parametara neuronske mreže. Razviti programski okvir koji omogućuje učenje igrača igre zmija korištenjem neuronskih mreža te grafički prikazuje ponašanje razvijenog igrača. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 10. lipnja 2022.

Zahvaljujem se mentoru doc. dr. sc. Marku Đuraseviću na svoj pomoći i podršci pruženoj tijekom studija i izrade završnog rada

Sadržaj

1. Uvod

Razvoj računalne grafike, nagli napredak mobilnih uređaja i pristup internetu omogućili su da danas mobilne uređaje možemo doživljavati i kao igrače konzole. Ovo sve možemo zahvaliti jednostavnoj igri imena Zmija koja se probila na tržište 1997. godine na dobro poznatom mobilnom uređaju Nokia 6110. Gotovo da ne postoji osoba koja nikada nije igrala igru Zmija ili barem čula da igra postoji.

Igra podrazumijeva svega nekoliko jednostavnih pravila:

1. Zmija se ne smije zabiti sama u sebe.
2. Zmija se ne smije zabiti u zid (okvir uređaja na kojem se igra).
3. Zmija mora pojesti što više hrane kako bi postigla veći rezultat.
4. Zmija smije napraviti potez u jednom od tri smjera (ravno, lijevo, desno).
5. Zmija poznaje točne koordinate hrane.
6. Zmija poznaje sve koordinate svoga tijela.
7. Koordinate hrane se generiraju nasumično.

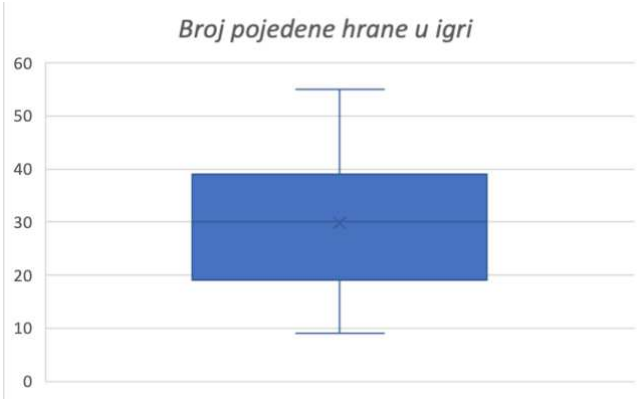
Razvoj područja umjetne inteligencije omogućio je da računalo nauči rješavati jednostavne probleme kao što je igranje igre Zmija bez ikakvog inicijalnog znanja. Upravo je rješavanje igre Zmija bez ikakvog inicijalnog znanja glavna tema koja se obrađuje u ovome radu.

Budući da zmija poznaje točne koordinate hrane i točne koordinate svoga tijela, rješenje koje nam može pasti na pamet jest da implementiramo jednostavan deterministički algoritam koji će se nastojati kretati u smjeru s najmanjom udaljenošću prema hrani poštujući sva pravila igre.

```
while alive {
    // moves sadrži poteze koji neće ubiti zmiju
    let moves = findAvailableMoves()

    // Ako nije vraćen niti jedan potez, zmija će sigurno umrijeti u sljedećem potezu.
    if moves.isEmpty {
        let randomMove = getRandomMove()
        executeMove(move: randomMove)
        alive = false
        break
    }

    // pronađi najkraći put do hrane i obavi taj potez
    let bestMove = findShortestDistanceFromFood(using: moves)
    executeMove(move: bestMove)
}
```

Sl. 1.1 Algoritam najkraćeg puta

Iz znatiželje rada i performansa ovoga algoritma, algoritam je implementiran kao dodatak na ovaj rad te je prikazan na slici 1.1.



Sl. 1.2 Rezultati algoritma najkraćeg puta

Performanse *algoritma najkraćeg puta* su same po sebi izrazito dobre te čak i ovako jednostavan algoritam može u većini slučajeva pobijediti vješte igrače igre. Na slici 1.2 možemo vidjeti *boxplot* koji prikazuje rezultate *algoritma najkraćeg puta* na 100 odigranih igara.

Algoritam *najkraćeg puta* pokazao je poprilično pohlepno ponašanje što je zapravo i bilo očekivano budući da se potez zmije formira isključivo prema najkraćoj udaljenosti od hrane te se isključuju samo oni potezi koji će ubiti zmiju direktno nakon obavljanja poteza. Također, *algoritam najkraćeg puta* koristi isključivo znanje koje je definirao programer.

Cilj ovoga rada je implementirati agenta koji će naučiti igrati igru bez ikakvog inicijalnog znanja. Želimo postići agenta koji će imati nepredvidivo ponašanje i koji će naučiti igrati igru u razumnom vremenu. Kako bismo implementirali agenta koristit ćemo *umjetne neuronske mreže*.

Nažalost, zbog vremenskog ograničenja nije moguće isprobati sve kombinacije težina kako bismo otkrili optimalnu kombinaciju težina u *umjetnoj neuronskoj mreži* za agenta koji igra igru. Zbog navedenog problema razvijeni su *metaheuristički algoritmi* koji ne pronalaze nužno optimalna rješenja već dovoljno dobra rješenja koja i dalje zadovoljavaju vremenska ograničenja [1]. Kao dio rada opisat ćemo *genetski algoritam* koji spada u klasu *metaheurističkih algoritama*. *Genetski algoritam* koristit ćemo za optimizaciju težina *umjetne neuronske mreže*.

Za potrebe rada razvijeno je i grafičko korisničko sučelje kojim se prikazuje rad algoritama.

2. Umjetne neuronske mreže

U području umjetne inteligencije često pokušavamo dati odgovor na pitanje „*Kada računalni program smatramo inteligentnim?*“. Jedan od odgovora na ovo pitanje mogao bi biti da je računalni program inteligentan kada bi uspio rješavati zadatke koje bi, kada bi ih radio čovjek, iziskivali inteligenciju. Već problem igranja igre Zmija iziskuje čovjekovu inteligenciju. Pri igranju igre moguće je primijeniti razne taktike koje će davati različite isplate.

Možemo se zapitati kako čovjek razvija taktiku za igranje igre? Poznato je kako se čovjekov mozak sastoji od velikog broja neurona koji rade paralelno te tako donose odluke. Budući da želimo da stroj oponaša čovjekovo ponašanje, mogli bismo pokušati simulirati prirodne neuronske mreže. Upravo na ovaj način nastale su *umjetne neuronske mreže*.

Umjetne neuronske mreže pripadaju konektivističkom pristupu umjetne inteligencije. Konektivistički pristup temelji se na izgradnji sustava arhitekture slične arhitekturi mozga koji, umjesto da ga se programira, uči samostalno na temelju iskustva [3].

Umjetna neuronska mreža je skup međusobno povezanih jednostavnih procesnih elemenata (neurona) čija se funkcionalnost temelji na biološkom neuronu i koji služe distribuiranoj paralelnoj obradi podataka [3].

2.1. Umjetni neuron

Najmanja jedinica u umjetnoj neuronskoj mreži jest neuron. Umjetni neuron sadrži *vektor težina*. Težine su *floating point* brojevi.

Vektor ulaza (također *floating point* brojevi) šalje se neuronu te se kombinira s vektorom težina koristeći skalarni umnožak. Rezultat skalarnog umnoška vektora ulaza i vektora težina prolazi kroz *prijenosnu funkciju* te iz prijenosne funkcije dobivamo *izlaz* neurona. Ovaj postupak prikazan je na slici 2.1.

Skalarni umnožak vektora ulaza i vektora težina može se predstaviti i na drugačiji način. Vrijednost svakog člana vektora ulaza x_i množi se s pripadnim članom vektora težina w_i te se akumulira u tijelu neurona [3].

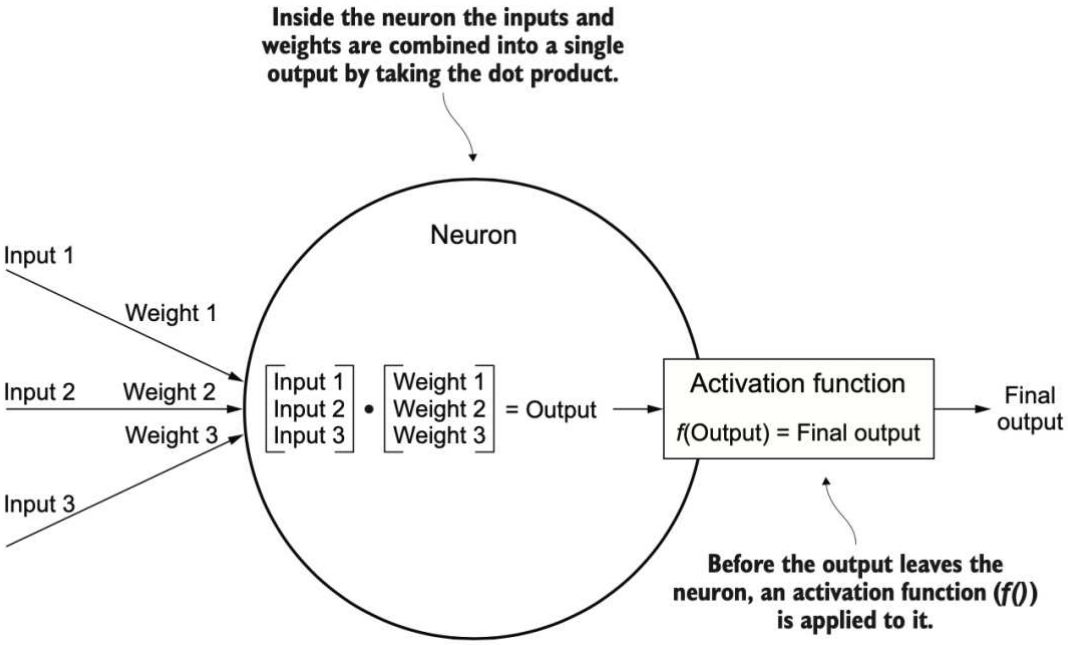
Time je definirana akumulirana vrijednost:

$$net = \left(\sum_{i=1}^{n} x_i \cdot w_i\right)$$

Ta se vrijednost propušta kroz prijenosnu funkciju f čime dobivamo izlaznu vrijednost neurona o [3].

Izlaz neurona možemo definirati:

$$o = f(net)$$



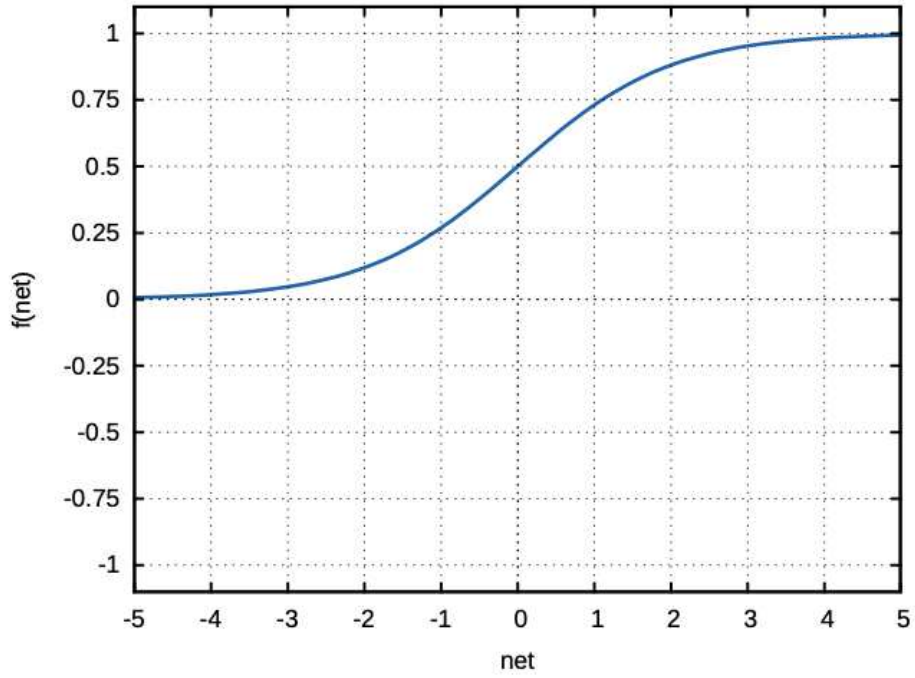
Sl. 2.1 Umjetni neuron [2]

2.2. Prijenosna funkcija

Prijenosne funkcije su gotovo uvijek nelinearne što omogućava neuronskoj mreži da daje rješenja i za nelinearne probleme. Kada ne bi postojale prijenosne funkcije, cijela neuronska mreža bi bila samo obična linearna transformacija [3].

Prijenosna funkcija odabrana za ovaj rad jest *sigmoidalna funkcija*. Graf *sigmoidalne funkcije* prikazan je na slici 2.2.

$$f(net) = sigm(net) = \frac{1}{1 + e^{-net}} = o$$



Sl. 2.2 Sigmoidalna funkcija [3]

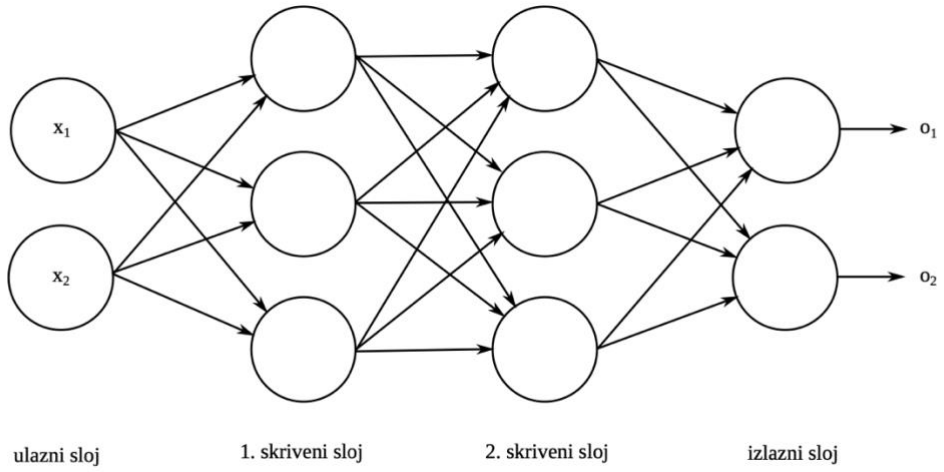
2.3. Umjetna neuronska mreža

Samo jedan umjetni neuron nam nije dovoljan kako bismo rješavali složenije probleme. Kao što znamo, niti čovjekov mozak se ne sastoji samo od jednog neurona već od mnoštva neurona koji su zajedno povezani u neuronsku mrežu.

Želimo napraviti složeniji sustav koji će biti izgrađen od više umjetnih neurona. Takav sustav nazivamo *umjetna neuronska mreža.* Arhitektura neuronske mreže govori nam kako su neuroni međusobno povezani i koliko ih ima.

Vrsta neuronske mreže koja je razvijena za potrebe rješavanja problema igranja igre Zmija naziva se *potpuno povezana unaprijedna slojevita umjetna neuronska mreža* te je primjer takve mreže prikazan na slici 2.3.

Pojam *unaprijedna* mreža govori nam da u mreži ne postoje ciklusi u obradi podataka. Pojam *slojevita* dodatno je ograničenje koje govori da su neuroni grupirani u jasno definirane slojeve te da neuroni grupirani u sloju i dobivaju pobudu (ulaze) isključivo od neurona u sloju $i-1$. Pojam *potpuno povezana* pojašnjava da svaki neuron iz sloja i na ulazu dobiva pobude (ulaze) od svih neurona iz sloja $i-1$ [3].



Sl. 2.3 Slojevita neuronska mreža arhitekture 2×3×3×2 [3]

Arhitekturu slojevitih mreža kraće zapisujemo navođenjem broja neurona u svakom od slojeva. Arhitekturu neuronske mreže prikazane na slici 2.3 označit ćemo s $2 \times 3 \times 3 \times 2$.

Rad umjetne neuronske mreže najlakše je opisati na primjeru neuronske mreže koja je odabrana za igranje igre Zmija. Navedena mreža sastoji se od tri sloja te ćemo arhitekturu te mreže označiti sa $6 \times 36 \times 3$.

Ulazni sloj čini šest neurona i kroz njih mreža dobiva ulazne podatke. Ulazni podatci koji su odabrani za ovaj rad:

1. Hoće li igra završiti ako zmija skrene lijevo (Ako hoće, dodjeljujemo vrijednost 1.0. Ako neće, dodjeljujemo vrijednost 0.0.).
2. Hoće li igra završiti ako zmija skrene desno (Ako hoće, dodjeljujemo vrijednost 1.0. Ako neće, dodjeljujemo vrijednost 0.0.).
3. Hoće li igra završiti ako se zmija nastavi gibati u istom smjeru (Ako hoće, dodjeljujemo vrijednost 1.0. Ako neće, dodjeljujemo vrijednost 0.0.).
4. Udaljenost zmije od hrane ako zmija skrene lijevo.
5. Udaljenost zmije od hrane ako zmija skrene desno.
6. Udaljenost zmije od hrane ako se zmija nastavi gibati u istom smjeru.

Implementacija ulaza umjetne neuronske mreže prikazana je na slici 2.4.

```
var input: [Double] = []

if dangers.left { input.append(1.0) } else { input.append(0.0) }
if dangers.right { input.append(1.0) } else { input.append(0.0) }
if dangers.straight { input.append(1.0) } else { input.append(0.0) }

input.append(distances.left)
input.append(distances.right)
input.append(distances.straight)
```

Sl. 2.4 Ulazi u neuronsku mrežu

Neuroni ulaznog sloja ne obavljaju nikakvu funkciju već je njihov izlaz direktno jednak podatku koji je dobiven izvana i koji mreža treba obraditi. Drugi sloj (odnosno jedini skriveni sloj) sastoji se od 36 neurona. Svaki od neurona u skrivenom sloju ima po šest težina (po jedna težina koja spaja svaki neuron ulaznog sloja). Te težine se ne dijele između neurona već svaki ima svojih vlastitih šest težina čime drugi sloj mreže (skriveni sloj) raspolaže sa $6 \cdot 36 = 216$ težina. Treći sloj (odnosno izlazni sloj) sastoji se od 3 neurona koji su spojeni na izlaze skrivenog sloja. U ovom sloju svaki od neurona ima po 36 težina (po jednu za svaki neuron skrivenog sloja). Ukupni broj težina u izlaznom sloju jest $36 \cdot 3 = 108$.

Izlazni sloj generira tri izlaza koje ćemo za potrebe igre interpretirati na način:

1. Izlaz koji predstavlja koliko je neuronska mreža sigurna da zmija treba skrenuti lijevo.
2. Izlaz koji predstavlja koliko je neuronska mreža sigurna da zmija treba skrenuti desno.
3. Izlaz koji predstavlja koliko je neuronska mreža sigurna da se zmija treba nastaviti gibati u istom smjeru.

Budući da se koristi sigmoidalna prijenosna funkcija, znamo da će vrijednost svakog izlaza biti između 0 i 1.

Akcija koju će naš algoritam napraviti jest ona koja je predstavljena izlazom s najvećom vrijednošću. Algoritam koji implementira ovakvo ponašanje prikazan je na slici 2.5.

```
func calculateNextMove(from input: [Double]) -> Constants.Game.Moves {
    let outputs = network.outputs(input: input)

    guard let maximum = outputs.max() else { return .straight }
    let indexOfLargestOutput = outputs.firstIndex(of: maximum)

    if indexOfLargestOutput == 0 { return .left }
    if indexOfLargestOutput == 1 { return .right }
    return .straight
}
```

Sl. 2.5 Funkcija koja računa sljedeću akciju koju će zmija učiniti

Navedena neuronska mreža raspolaže s $216 + 108 = 324$ težine, što predstavlja broj parametara koji algoritam učenja može podešavati kako bi neuronska mreža davala što bolje akcije za trenutno stanje igre te tako postigla bolji rezultat.

3. Evolucijsko računarstvo

S obzirom na to da su danas računala ekstremno brza, često se njihovom uporabom u jednoj sekundi mogu istražiti milijuni ili čak milijarde potencijalnih rješenja za razne probleme. To znači da pretraživanjem cjelokupnog prostora stanja možemo i za razne probleme doći čak i do optimalnog rješenja. Ovo, dakako, vrijedi samo ako smo jako sretni, pa rješavamo problem koji se može riješiti grubom silom, odnosno uporabom algoritama koji pretražuju cjelokupni prostor stanja. Nažalost, postoji čitav niz problema koji ne spadaju u ovu kategoriju [1].

U prethodnom poglavlju ustanovili smo kako se suočavamo s problemom podešavanja 324 parametra (težina) kako bi neuronska mreža davala što bolje akcije za trenutno stanje igre te tako postigla bolji rezultat. Pretraživanje cjelokupnog prostora stanja za problem ispitivanja optimalnih kombinacija težina neuronske mreže jednostavno nije moguć.

Zbog navedenog problema razvijeni su *metaheuristički algoritmi* koji ne pronalaze nužno optimalna rješenja već dovoljno dobra rješenja koja možemo dobiti u razumnom vremenu.

U metaheurističke algoritme spadaju algoritmi evolucijskog računanja. Evolucijske algoritme danas uobičajeno dijelimo na četiri potpodručja: *genetske algoritme*, *genetsko programiranje*, *evolucijske strategije* te *evolucijsko programiranje*. Zajednička im je ideja da rade s populacijom rješenja nad kojima se primjenjuju evolucijski operatori (selekcija, križanje, mutacija, zamjena) čime populacija iz generacije u generaciju postaje sve bolja i bolja [1].

Kao dio implementacije razvijen je genetski algoritam. Genetski algoritam koristi se za rješavanje problema optimizacije težina umjetne neuronske mreže arhitekture $6 \times 36 \times 3$ koji se koristi za određivanje poteza zmije. Rješenje pri tome ima definiranu *funkciju dobrote* (engl. *fitness-function*) koju će genetski algoritam pokušati maksimizirati.

3.1. Funkcija dobrote

Funkcija dobrote koja je odabrana u implementaciji je najbolji rezultat igre (engl. *high-score*) te će zbog toga algoritam preferirati ona rješenja koja ostvaruju bolje rezultate.

Na slici 3.1 prikazana je implementacija funkcije dobrote u kojoj je vidljivo kako funkcija dobrote pokreće simulaciju igre kojoj u konstruktor predajemo neuronsku mrežu s težinama

koju je generirao genetski algoritam. Simulacija odigra igru u kojoj je svaki potez zmije određen radom umjetne neuronske mreže. Na kraju igre simulacija vrati rezultat koji je postigla s predanom neuronskom mrežom te se taj rezultat koristi kao povratna vrijednost funkcije dobrote.

```
var fitness: Double {
    // Stvara se simulacija igre kojoj u konstruktor predajemo neuronsku mrežu s trenutnim težinama
    let simulation = SnakeGameSimulation(network: Network(weights: weights))

    // Simulacija igre se pokreće te se na kraju igre vraća rezultat koji je postignut
    return simulation.runSimulation()
}
```

Sl. 3.1 Funkcija dobrote

3.2. Kriterij zaustavljanja

Kriterij zaustavljanja predstavlja granicu koju algoritam evolucijskog računarstva mora dostići kako bismo rezultat optimizacije smatrali dovoljno dobrim te zaustavili rad algoritma.

Kriterij zaustavljanja koji smo postavili u implementaciji jest minimalan rezultat koji zmija mora dostići. Genetski algoritam će optimizirati težine umjetne neuronske mreže sve dok ne dostigne minimalan rezultat, a taj rezultat je u našem slučaju 40. Kada genetski algoritam pronađe rješenje (težine) koje su u simulaciji igre postigle kriterij zaustavljanja, algoritam staje, a težine se pohranjuju na uređaj.

4. Genetski algoritam

Postoje različite izvedbe genetskih algoritama. Dvije krajnosti koje možemo spomenuti su *eliminacijski genetski algoritam* i *generacijski genetski algoritam* [1]. Za ovaj rad odabran je generacijski genetski algoritam čija je implementacija prikazana na slici 4.1.

Generacijski genetski algoritam na početku svake iteracije fiksira populaciju roditelja. Iz te populacije biraju se roditelji koji križanjem i mutacijom stvaraju djecu koja se pohranjuju u pomoćnu privremenu populaciju. Postupak se radi sve do trenutka dok u populaciji djece nema jednak broj *jedinki* (odnosno sinonimi: *rješenje*, *kromosom*) koliko ih je i u roditeljskoj populaciji. Kada se to dogodi, roditeljska se populacija briše, a stvorena populacija djece promovira se u roditelje. Taj trenutak označava smjenu generacije nakon čega se postupak ciklički ponavlja [1]. Implementacija smjene generacija prikazana je na slici 4.2. Kromosom zmije predstavljen je poljem decimalnih brojeva koji su zapravo težine umjetne neuronske mreže. Veličina populacije u implementaciji postavljena je na 10 jedinki.

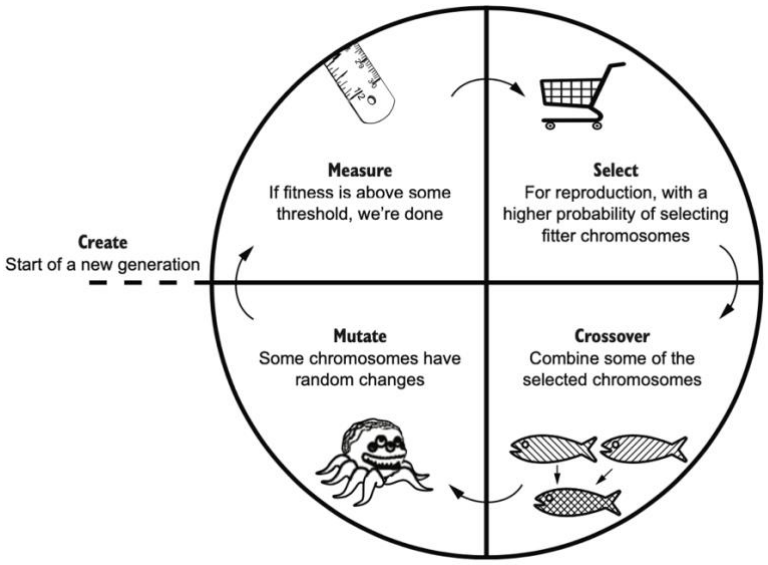
```
func run() -> ChromosomeType {
    // kreiraj nasumičnu jedinku
    var best: ChromosomeType = ChromosomeType.randomInstance()
    var bestFitness: Double = best.fitness

    // ponavljaj postupak
    while true {
        for (index, individual) in population.enumerated() {
            // izračunaj funkciju dobrote za svaku jedinku u populaciji
            fitnessCache[index] = individual.fitness
            // ako jedinka zadovoljava čvrsto ograničenje, završi algoritam
            if fitnessCache[index] >= threshold {
                return individual
            }
            // ako je jedinka postigla najveći rezultat do sada postavlja se kao najbolja jedinka
            if fitnessCache[index] > bestFitness {
                bestFitness = fitnessCache[index]
                best = ChromosomeType(from: individual)
            }
        }
        // ukupni rezultat populacije
        fitnessSum = fitnessCache.reduce(0, +)
        // stvaranje nove populacije za sljedeću generaciju
        reproduceAndReplace()
        // operator mutacije
        mutate()
    }
}
```

Sl. 4.1 Genetski algoritam

```swift
func reproduceAndReplace() {
    // stvori polje koje će sadržavati jedinke za iduću generaciju
    var newPopulation: [ChromosomeType] = [ChromosomeType]()

    // u novu populaciju se dodaju tri najbolje jedinke trenutne generacije
    newPopulation.append(contentsOf: findElite())

    // ponavljaj postupak dok ne popuniš cijelu populaciju
    while newPopulation.count < population.count {
        // odaberi dva roditelja koja ća se križati
        let parents = (parent1: pickTournament(), parent2: pickTournament())

        if Random.double() < crossoverChance {
            // križaj roditelje te generiraj dvoje potomaka
            let children = parents.parent1.crossover(other: parents.parent2)
            // potomke pohrani u novu populaciju
            newPopulation.append(children.child1)
            newPopulation.append(children.child2)
        } else {
            newPopulation.append(parents.parent1)
            newPopulation.append(parents.parent2)
        }
    }

    // u slučaju kada je nova populacija neparna, makni zadnju jedinku iz populacije
    if newPopulation.count % 2 != 0 { newPopulation.removeLast() }

    // zamijeni staru populaciju novom
    population = newPopulation
}
```

Sl. 4.2 Funkcija koja obavlja zamjenu populacija

Rad genetskog algoritma može se ilustrativno prikazati slikom 4.3:



Sl. 4.3 Rad genetskog algoritma [2]

4.1. Elitizam

Možemo primijetiti kako smo u funkciji *reproduceAndReplace* koja je prikazana na slici 4.2, u novu populaciju prvo nadodali tri najbolje jedinke trenutne generacije. Ovaj postupak nazivamo *elitizam*.

Elitizam je svojstvo algoritma da ne može izgubiti najbolje pronađeno rješenje. Budući da u novu populaciju odmah dodajemo najbolje jedinke trenutne populacije, osigurali smo da se najbolje jedinke neće izgubiti u procesu zamjene populacija što će značajno ubrzati rad algoritma.

Implementacija algoritma koji pronalazi najbolje jedinke u populaciji prikazan je na slici 4.4.

```
func findElite(numberOfElites: Int = 3) -> [ChromosomeType] {
    // kopiraj rezultate svih jedinki u novo polje
    var fitnessCopy = fitnessCache
    // stvori novo polje koje će sadržavati najbolje jedinke trenutne generacije
    var best: [ChromosomeType] = []

    // ponavljaj dok nisi našao traženi broj najboljih jedinki
    for _ in 0...numberOfElites {
        // pronađi najveći postignut rezultat
        let max = fitnessCopy.max()
        // pronađi prvi indeks najboljeg postignutog rezultata
        let index = fitnessCache.firstIndex(of: max!)!
        // nadodaj jedinku u polje najboljih jedinki
        best.append(population[index])
        // eliminiraj pronađenu jedinku iz kopiranog polja rezultata
        let indexOfCopy = fitnessCopy.firstIndex(of: max!)!
        fitnessCopy.remove(at: indexOfCopy)
    }
    return best
}
```

Sl. 4.4 Ostvarenje elitizma

4.2. Operatori

Na slici 4.3 koja ilustrativno prikazuje rad genetskog algoritma možemo primijetiti operatore *selekcije* (engl. *select*), *križanja* (engl. *crossover*) i *mutacije* (engl. *mutate*) koji su ključni za rad genetskog algoritma. Ako nismo pažljivo implementirali operatore možemo drastično degradirati performanse genetskog algoritma. Zbog toga opisujemo svaki operator i bitne značajke ispravne implementacije i odabira parametara.

4.2.1. Selekcija

Operator selekcije služi za odabir roditelja koji će operatorom križanja stvoriti potomke za novu populaciju. Možemo zaključiti kako je ovaj postupak vrlo važan budući da on određuje koja će se svojstva propagirati u sljedeću generaciju. Želimo križati jedinke koje postižu najbolje rezultate. Naivna implementacija selekcije bila bi da odaberemo dvije jedinke iz čitave populacije koje su postigle najbolje rezultate u igri. Ovaj pristup nije ispravan jer dolazimo u situaciju u kojoj lako možemo zaglaviti u lokalnom optimumu problema. Želimo postupak koji će uvesti stohastičnost u proces selekcije. Ne želimo birati samo najbolje jedinke iz čitave populacije već želimo birati jedinke koje su najbolje u slučajno generiranom uzorku populacije. Također, na ovaj način uvodimo raznolikost u generiranju potomaka za novu generaciju genetskog algoritma.

Upravo ovaj problem rješavaju *turnirske selekcije*. U sklopu rada implementirana je k-turnirska selekcija u kojoj je moguće kontroliranje selekcijskog pritiska jednostavnim podešavanjem parametra k. Implementacija ove selekcije prikazana je na slici 4.5:

```
func pickTournament(numberOfParticipants k: Int) -> ChromosomeType {
    // nasumično generiraj jedinku i postavi njen rezultat kao
        najbolji
    var best: ChromosomeType = ChromosomeType.randomInstance()
    var bestFitness: Double = best.fitness

    // ponavljaj k puta
    for _ in 0..<k {
        // iz populacije slučajnim postupkom uz jednoliku
            distribuciju vjerojatnosti odaberi jednu jedinku
        let test = Int(arc4random_uniform(UInt32(population.count)))
        // provjeri postiže li jedinka bolji rezultat od trenutne
            najbolje jedinke
        if fitnessCache[test] > bestFitness {
            // ako postiže bolji rezultat, postavi ju kao trenutnu
                najbolju jedinku
            bestFitness = fitnessCache[test]
            best = population[test]
        }
    }
    // vrati jedinku koja je pobijedila u turniru
    return best
}
```

Sl. 4.5 k-turnirska selekcija

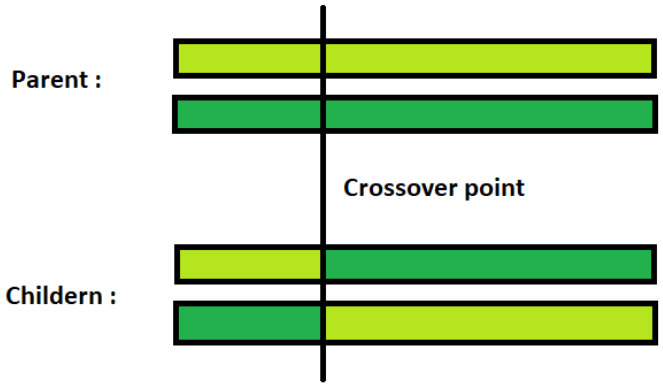
Opisani postupak iz populacije vadi uzorak od k jedinki i vraća onu koja je najpoželjnija. Najpoželjnija će jedinka biti ona koja od jedinki iz uzorka ima najveću dobrotu. Budući da trebamo dva roditelja, dobit ćemo ih tako da dva puta zovemo ovaj operator.

Što je parametar k veći, veći se naglasak daje na najbolja rješenja. Postavimo li da je k jednak veličini populacije, algoritam će kao roditelja uvijek birati najbolju jedinku populacije što će otežati algoritmu izlazak iz lošeg lokalnog optimuma. Ako k postavimo na jedan, roditelji će se birati potpuno slučajno bez utjecaja funkcije dobrote. Za potrebe rada k smo postavili na vrijednost tri (tro turnirski genetski algoritam), čime smo osigurali da postoji barem minimalna kompeticija između jedinki za roditeljstvo [1].

4.2.2. Križanje

Operator križanja predstavlja metodu razmjene svojstava (gena) između dva roditelja odabrana operatorom selekcije te mu je glavna uloga pretraživanje okoline roditelja. Metoda križanja koja je odabrana za ovaj rad jest *križanje s jednom točkom prijeloma* (eng. *one point crossover*).

Križanje s jednom točkom prijeloma vrlo je jednostavno te podrazumijeva postojanje dva roditelja. Budući da su roditelji kromosomi koji predstavljaju težine umjetne neuronske mreže oni su reprezentirani poljem *floating point* brojeva. Prvo nasumično odredimo indeks polja roditelja kojeg ćemo tretirati kao točku prijeloma (engl. *crossover point)*. Stvaramo prvo dijete tako da u prazno polje nadodajemo težine prvog roditelja sve do indeksa točke prijeloma. Od točke prijeloma i sve do kraja polja upisujemo težine drugog roditelja. Drugo dijete stvaramo na suprotan način. Do točke prijeloma upisujemo težine drugog roditelja, a od točke prijeloma upisujemo težine prvog roditelja. Ovaj postupak prikazan je na slici 4.6, a ostvarenje križanja s jednom točkom prijeloma prikazana je na slici 4.7.



Sl. 4.6 Križanje s jednom točkom prijeloma [4]

```
/// One point crossover
func crossover(other: SnakeChromosome)
-> (child1: SnakeChromosome, child2: SnakeChromosome) {

    // nasumično generiraj točku prijeloma
    let point = arc4random_uniform(UInt32(weights.count - 1))

    let child1 = SnakeChromosome(from: self)
    let child2 = SnakeChromosome(from: other)

    // stvori dva prazna polja u koja će se pohraniti težine
    child1.weights = []
    child2.weights = []

    // ponavljaj za svaki indeks polja težina roditelja
    for i in 0 ..< weights.count {
        // ako je indeks manji od točke prijeloma
        if i < Int(point) {
            // u prvo dijete upiši težinu prvog roditelja
            child1.weights.append(self.weights[i])
            // u drugo dijete upiši težinu drugog roditelja
            child2.weights.append(other.weights[i])
        } else {
            // u prvo dijete upiši težinu drugog roditelja
            child1.weights.append(other.weights[i])
            // u drugo dijete upiši težinu prvog roditelja
            child2.weights.append(self.weights[i])
        }
    }
    return (child1, child2)
}
```

Sl. 4.7 Implementacija križanja s jednom točkom prijeloma

Vjerojatnost križanja je također vrlo važna prilikom implementacije genetskog algoritma. Vjerojatnost križanja koja je odabrana u ovome radu jest 70% što znači da će se otprilike 70% nove populacije sastojati od jedinki koje su nastale iz procesa križanja.

4.2.3. Mutacija

Nakon što završi proces križanja, postoji vjerojatnost da će se na novim jedinkama dogoditi mutacija. Mutacija ima glavnu ulogu bježanja iz lokalnih ekstrema te to ostvaruje velikim skokovima u prostoru pretraživanja [1]. Možemo primijetiti kako operator križanja koristi isključivo genetski materijal oba roditelja, ali ne uvodi nikakav novi genetski materijal. Možemo zaključiti da ako koristimo samo operator križanja, bit će teško odvojiti se od starih rješenja koja su nas možda čak i dovela u situaciju da smo zaglavili u nekom od lokalnih

optimuma. Za uvođenje novog genetskog materijala u jedinku koristimo operator mutacije. Operator mutacije koji je u implementaciji pokazao najbolje performanse prvo prolazi kroz sve jedinke nove populacije te s vjerojatnošću od 10% odlučuje hoće li se jedinka mutirati. Ovakvim postupkom za sve jedinke nove populacije postoji vjerojatnost da će se mutirati. Vjerojatnost mutacije jedne jedinke od 10% znači da će se otprilike 10% nove populacije mutirati. Postupak je prikazan na slici 4.8.

```
func mutate() {
    // svaka jedinka u populaciji može biti mutirana
    for individual in population {
        if Random.double() < mutationChance {
            // mutiraj jedinku
            individual.mutate()
        }
    }
}
```

Sl. 4.8 Mutacija populacije

Mutacija jedinke prolazi kroz polje težina te s vjerojatnošću 50% mutira svaku težinu. Ako je vjerojatnost mutacije za trenutnu težinu zadovoljena, na trenutnu težinu dodaje se nasumična vrijednost iz *Gaussove distribucije*.

Za potrebe rada implementirana je funkcija koja vraća nasumičan broj iz Gaussove distribucije na intervalu $[-1, 1]$.

$$weight \mathrel{+}= gauss(-1, 1)$$

Ako je vrijednost težine nakon mutacije veća od 1, težina se podrezuje na vrijednost 1. Također, težina nakon mutacije ne može biti manja od -1 jer će se ograničiti na minimalnu vrijednost -1.

Ostvarenje mutacije težina jedinke prikazana je na slici 4.9.

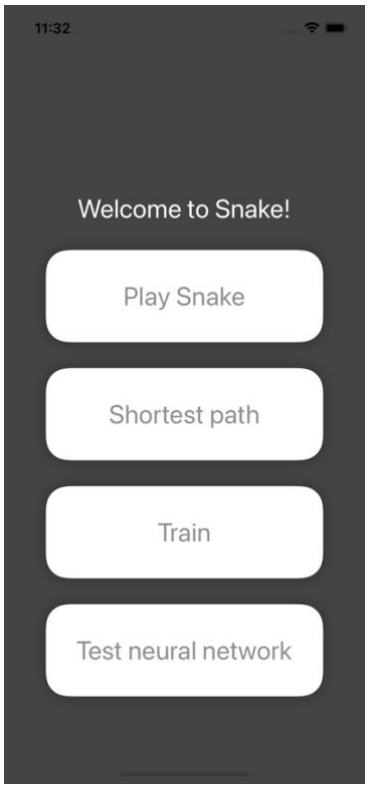
```
func mutate() {
    // iteriraj kroz svaku težinu
    weights = weights.map { weight in
        // izmjeni težinu s vjerojatnošću 50%
        if Random.double() > 0.5 {
            // na trenutnu težinu dodaj nasumično generiran broj iz Gassove distribucije
            var newWeight = weight + Random.randomNumberFromGaussianDistribution()
            // ako je težina veća od 1
            if newWeight > 1.0 {
                // podreži težinu na 1
                newWeight = 1.0
            }
            // ako je težina manja od -1
            else if newWeight < -1.0 {
                // podreži težinu na -1
                newWeight = -1.0
            }
            return newWeight
        } else {
            // nemoj izmjeniti težinu
            return weight
        }
    }
}
```

Sl. 4.9 Mutacija težina jedinke

5. Grafičko korisničko sučelje

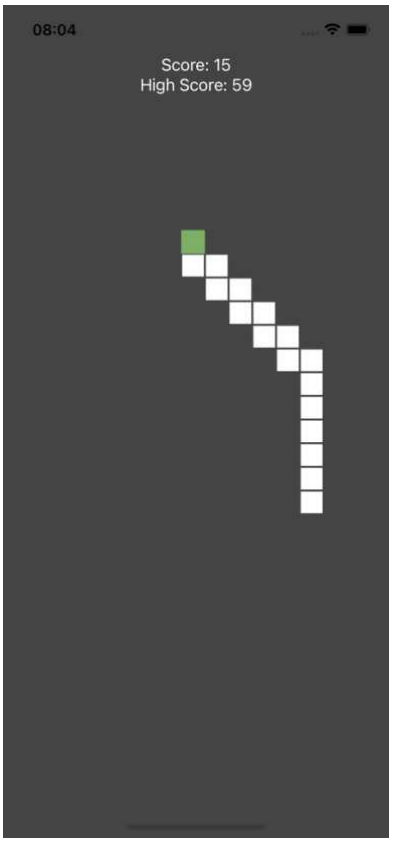
Kao dio implementacije izrađeno je grafičko korisničko sučelje koje omogućava korisniku da se okuša u igranju igre, pogleda ponašanje algoritma najkraćeg puta, pokrene učenje neuronske mreže genetskim algoritmom te da isproba rad naučene neuronske mreže.

Razvijena je podrška za uređaje s operacijskim sustavom iOS, iPadOS i MacOS.



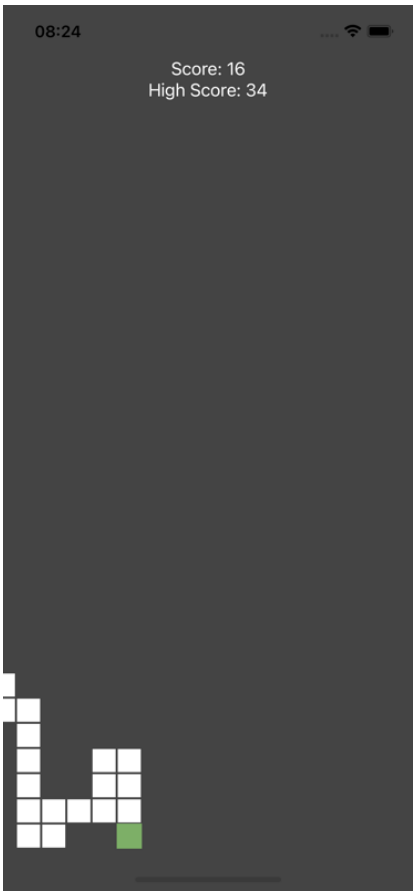
Sl. 5.1 Izbornik, iPhone 13 Pro

Ulaskom u aplikaciju korisniku se nudi izbornik prikazan na slici 5.1. Pritiskom na dugme *Play Snake* korisnik se može okušati u igranju igre te vidjeti može li pobijediti algoritam najkraćeg puta ili umjetnu neuronsku mrežu. Pritiskom na dugme *Shortest path* pokreće se agent koji koristi algoritam najkraćeg puta. Grafičko sučelje koje prikazuje rad algoritma najkraćeg puta prikazano je na slici 5.2.

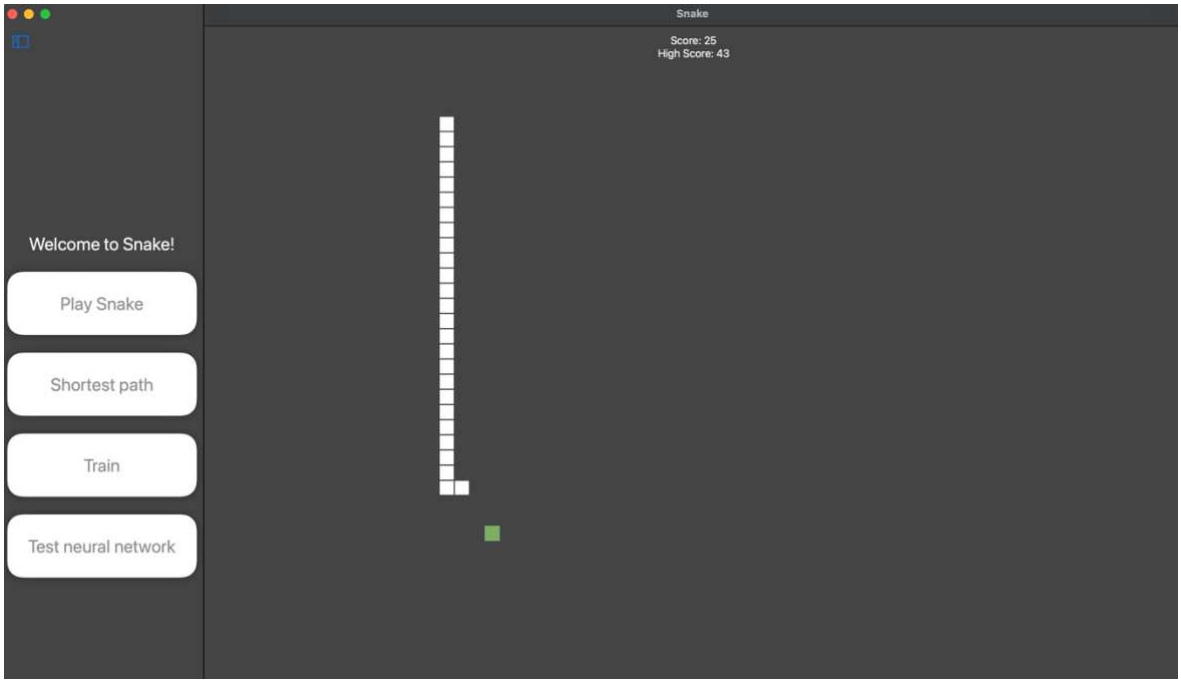


Sl. 5.2 Algoritam najkraćeg puta, iPhone 13 Pro

Pritiskom na dugme *train* pokreće se rad genetskog algoritma. Kada je dostignut kriterij zaustavljanja na uređaj se pohranjuju težine umjetne neuronske mreže. Pritiskom *Test neural network* može se vidjeti rad umjetne neuronske mreže koja je pohranjena nakon završetka rada genetskog algoritma. Grafičko sučelje koje prikazuje rad umjetne neuronske mreže prikazano je na slikama 5.3 i 5.4.



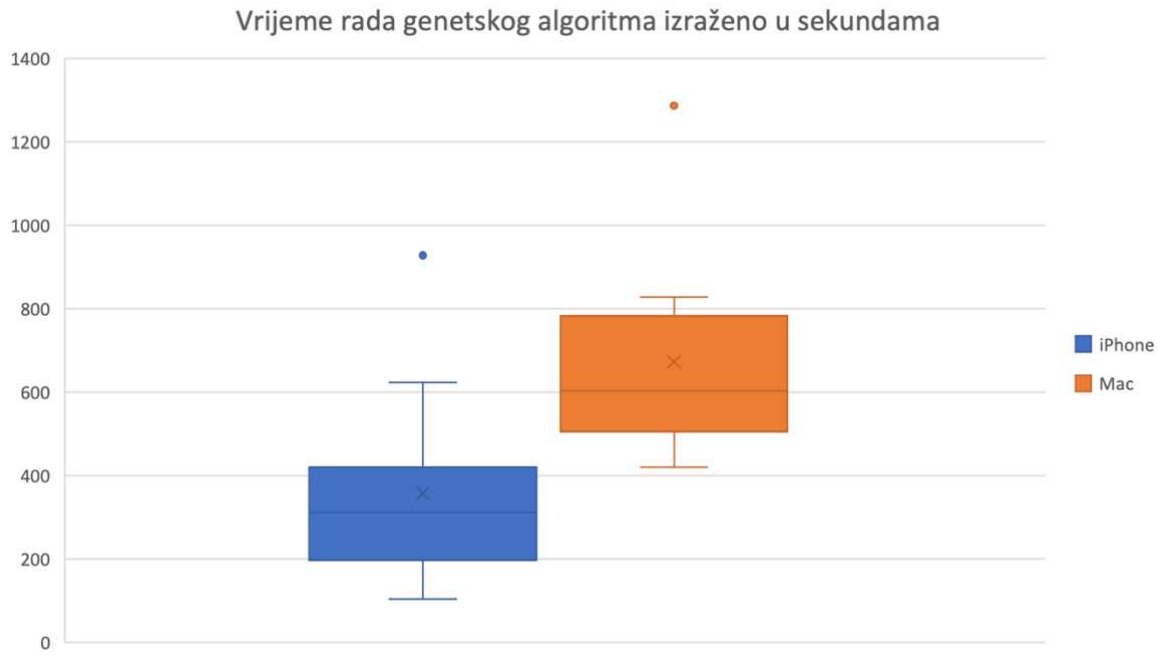
Sl. 5.3 Umjetna neuronska mreža, iPhone 13 Pro



Sl. 5.4 Umjetna neuronska mreža, MacBook Pro

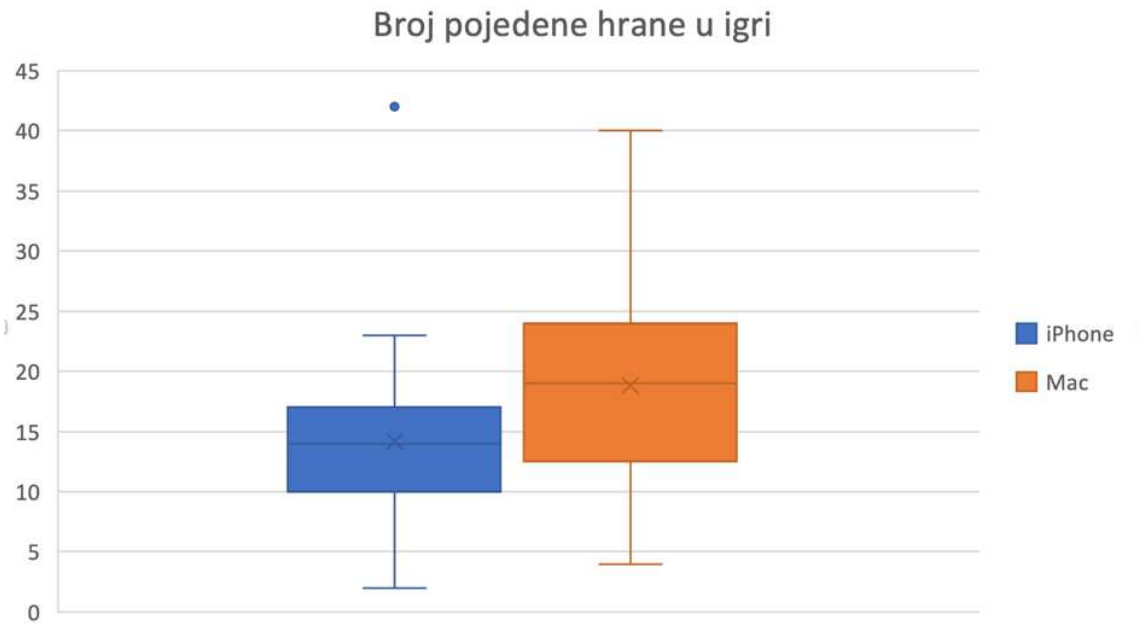
6. Rezultati implementacije

Budući da je razvijeno grafičko korisničko sučelje za sve Apple uređaje, zanimljivo je promatrati kako veličina zaslona utječe na brzinu optimizacije težina umjetne neuronske mreže. Na slici 6.1 vidljiv je *boxplot* koji prikazuje vrijeme potrebno za učenje umjetne neuronske mreže genetskim algoritmom na uzorku od 10 igara. Zanimljivo je za primijetiti kako genetski algoritam na uređaju iPhone značajno brže završi s radom u odnosu na Mac. Bitno je za naglasiti kako su apsolutne dimenzije hrane i tijela zmije fiksirane te se ne mijenjaju u ovisnosti o veličini zaslona. Iz toga možemo zaključiti kako će zmija na većim uređajima imati više prostora za kretanje nego na manjim uređajima.



Sl. 6.1 Vrijeme rada genetskog algoritma izraženo u sekundama

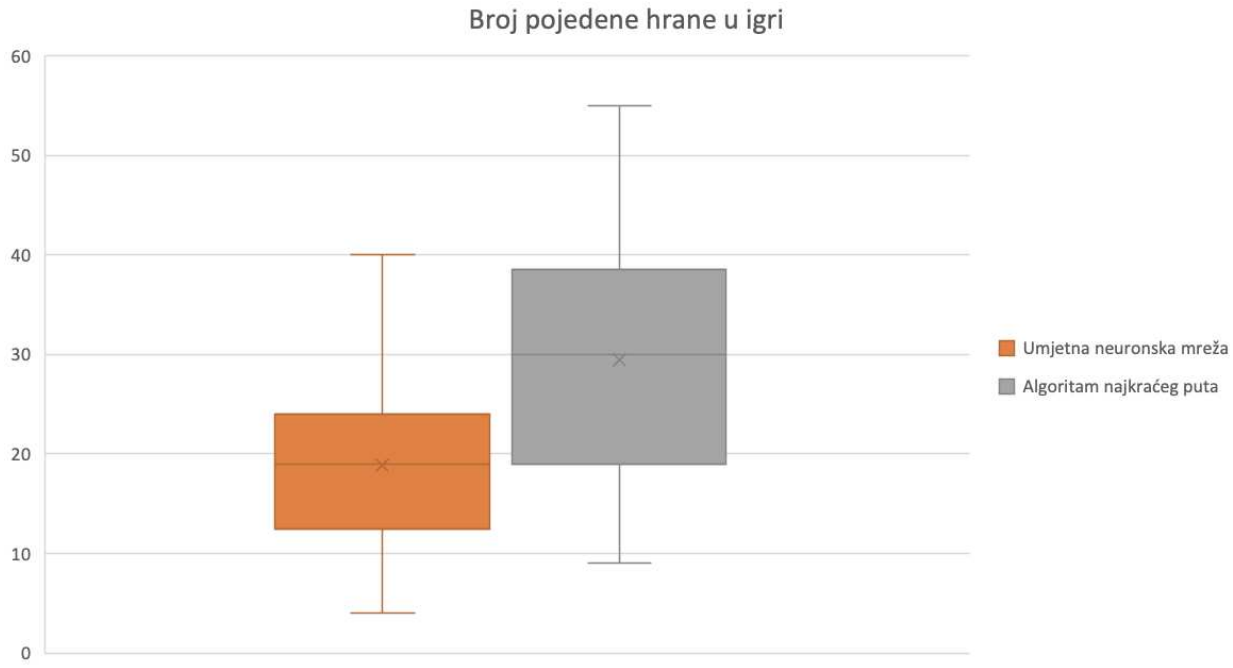
Ovakvo ponašanje možemo pripisati manjoj veličini zaslona uređaja iPhone. Budući da je zaslon manji, veća je vjerojatnost da će zmija pojesti više hrane jer je i veća vjerojatnost da će se hrana generirati bliže zmiji.



Sl. 6.2 Rezultati naučene neuronske mreže

Na slici 6.2 vidljiv je *boxplot* koji prikazuje rezultate umjetne neuronske mreže na uzorku od 100 igara. Zanimljivo je primijetiti kako naučena neuronska mreža na većem zaslonu ostvaruje znatno bolje rezultate od neuronske mreže na manjem zaslonu iako je potrebno više vremena za učenje. Ovo također možemo pripisati tome što je veći zaslon manje osjetljiv na faktor sreće zbog veće površine koju zmija mora proći za vrijeme učenja.

Prosječni rezultati koje postiže umjetna neuronska mreža na uzorku od 100 igara nisu niti približni vrijednosti kriterija zaustavljanja kojeg je uspjela ostvariti za vrijeme učenja. Razlog tomu leži u činjenici da funkcija dobrote evaluira samo jedan rezultat igre te se zbog toga može dogoditi slučajan sretan ishod u kojoj su koordinate hrane generirane na gotovo optimalnoj poziciji za trenutnu umjetnu neuronsku mrežu. Ovakvo ponašanje manje je uočljivo na uređaju Mac zbog već ustanovljenog manjeg utjecaja sreće.



Sl. 6.3 Usporedba rada umjetne neuronske mreže i algoritma najkraćeg puta

Na slici 6.3 vidljivo je kako na uzorku od 100 igara neuronska mreža ostvaruje lošije rezultate od algoritma najkraćeg puta. Postoji mogućnost da bi se podešavanjem parametara genetskog algoritma poboljšali rezultati umjetne neuronske mreže, međutim, to nije moguće garantirati. Nedostatak genetskih algoritama jest da nikada nije moguće garantirati da su odabrani optimalni parametri genetskih operatora i genetskog algoritma. Igra zmije je primjer kako nije uvijek potrebno posezati za algoritmima evolucijskog računarstva u problemima koji ih ne zahtijevaju već je bolje problem riješiti na jednostavniji način.

Zaključak

U sklopu rada implementiran je agent kojemu je cilj naučiti igrati igru Zmija bez ikakvog početnog znanja o igri. Za potrebe implementacije agenta razvijen je genetski algoritam koji služi za optimizaciju težina umjetne neuronske mreže. Nakon detaljnog opisa rada genetskog algoritma i motivacije za njegovu uporabu, opisani su operatori (*selekcija, križanje* i *mutacija*) koji su implementirani kao dio genetskog algoritma. Također, detaljno je objašnjena umjetna neuronska mreža i njen rad. Umjetna neuronska mreža donosi odluku o potezu koju će agent napraviti na temelju trenutne situacije u igri. Kako bi se prikazao rad umjetne neuronske mreže i genetskog algoritma razvijeno je grafičko korisničko sučelje koje je moguće pokrenuti na svim uređajima marke Apple.

Također, u radu je dana usporedba performansi rada genetskog algoritma na različitim veličinama uređaja. Kao dodatak na ovaj rad implementiran je i jednostavan deterministički algoritam najkraćeg puta koji postiže značajno bolje rezultate od umjetne neuronske mreže što nam može pokazati kako algoritmi evolucijskog računarstva nisu nužni za sve probleme.

Kako bismo unaprijedili učenje umjetne neuronske mreže genetskim algoritmom mogli bismo pokušati ostvariti druge vrste genetskih operatora koji bi mogli ostvariti bolje rezultate. Također, moguće je i pokušati podesiti trenutne parametre genetskih operatora i genetskog algoritma te vidjeti hoćemo li na ovaj način unaprijediti rezultate.

Literatura

[1] Čupić M. *Prirodom inspirirani optimizacijski algoritmi*. Prvo izdanje, lipanj 2016.

[2] Kopec D. *Classic Computer Science Problem in Swift*. First edition. New York: Manning Publications Co, 2018.

[3] Čupić M. *Umjetne neuronske mreže*. Prvo izdanje, svibanj 2016.

[4] Dutta A., Crossover in Genetic Algorithm, (2019, lipanj). Poveznica: https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/; pristupljeno 16. svibnja 2022.

Optimizacija neuronske mreže evolucijskim algoritmom za igru Zmija

Sažetak

Ovaj rad se bavi razvojem agenta koji će bez inicijalnog znanja naučiti igrati igru zmija. Daje se opis rada i implementacije genetskog algoritma i umjetne neuronske mreže. Kao dio implementacije genetskog algoritma ostvareni su i svi genetski operatori. Prikazane su performanse rada umjetne neuronske mreže te se uspoređuju njeni rezultati s jednostavnim algoritmom najkraćeg puta. Razvijeno je grafičko korisničko sučelje kojim se može prikazati rad algoritma te usporediti performanse rada genetskog algoritma na različitim veličinama uređaja.

Ključne riječi: igra Zmija, genetski algoritam, genetski operatori, umjetna neuronska mreža, algoritam najkraćeg puta, grafičko korisničko sučelje

Neural network optimization using an evolutionary algorithm for the game Snake

Abstract

This thesis deals with the development of an agent which will learn how to play the game Snake without any initial knowledge of the game. The thesis also provides a description of genetic algorithms and artificial neural networks. All of the genetic operators have been developed as a part of the implementation. The performance of the artificial neural network is evaluated against a simple shortest path algorithm. A graphical user interface is developed which could be used for comparing the performance of the genetic algorithm on different screen sizes.

Keywords: Snake game, genetic algorithm, genetic operators, artificial neural networks, shortest path algorithm, graphical user interface