SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 848

Igranje igre *2048* korištenjem Kartezijevog genetskog programiranja

Paulo Erak

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 848

Pristupnik:	**Paulo Erak (0036532743)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Igranje igre 2048 korištenjem Kartezijevog genetskog programiranja**

Opis zadatka:

Proučiti pravila igre 2048 te izraditi jednostavnu programsku implementaciju igre koja omogućuje igranje od strane čovjeka ili automatski razvijenog agenta. Istražiti mogućnost primjene Kartezijevog genetskog programiranja za razvoj autonomnog igrača za navedenu igru. Razviti sustav za učenje temeljen na Kartezijevom genetskom programiranju, koji na temelju trenutnog stanja igre određuje najbolju akciju koju je moguće napraviti. Ocijeniti učinkovitost razvijenih agenata za igranje navedene igre te navesti moguća poboljšanja metode. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 9. lipnja 2023.

Zahvaljujem se mentoru doc. dr. sc. Marku Đuraseviću što je vrlo ugodnom radnom atmosferom, savjetima, preporukama literature i strpljenjem uvelike olakšao izradu ovog završnog rada.

Zahvaljujem se majci, ocu, bratu Jakovu i svoj svojoj obitelji na konstantnoj potpori i bodrenju u svemu što sam radio.

Zahvaljujem se svim svojim prijateljima koji su bili uz mene i koji su mi bili velika potpora kroz cijeli preddiplomski studij, a i prije njega.

Bez ovih ljudi ne bih bio tu gdje jesam i tko jesam. Uistinu veliko hvala.

SADRŽAJ

1. Uvod

Iako je većina igara napravljena za dvoje ili više igrača, razvojem tehnologije, a posebno računala, počinju se razvijati igre za samo jednog igrača. U njima je najčešći cilj dobiti najbolji/najveći rezultat koristeći pravila po kojima igra funkcionira na optimalan način. Tako se često igri pristupa kao slagalici koju u što manje poteza pokušavamo složiti što bolje, a svaki od naših poteza se boduje i utječe na sljedeće poteze.

Igra *2048* upravo je takva igra napravljena za jednog igrača. Cilj igre je dobiti što veći rezultat koji je sam po sebi zbroj novostvorenih brojeva na ploči ispred nas. Imamo vrlo jednostavne poteze: pomicanje svih dijelova ploče lijevo, desno, gore ili dolje. Prilikom pomicanja, elementi koji su isti i susjedni, a nalaze se jedan drugome na putanji, spajaju se i time otvaraju prostor na ploči. Nakon svakog poteza se na nasumično otvoreno mjesto stavlja novi element. Igra se čini poprilično jednostavna za shvatiti te ćemo u kasnijem poglavlju vidjeti još neke pojedinosti igre.

Unatoč svojoj jednostavnosti, igra može postati podosta kompleksna. Upravljanje praznim prostorom te općenita prostorna organizacija elemenata postaje zahtjevnija kako vrijeme prolazi. Igrač će se prije ili kasnije naći u situaciji kad mu je ploča ispunjena do kraja i više nije u stanju napraviti ijedan potez. U tom trenutku igra se završava.

Cilj ovog rada je stvoriti agenta ili populaciju agenata koristeći Kartezijevo genetsko programiranje koji bi na osnovu stanja ploče napravili što bolji potez u kontekstu cijele partije.

U sklopu ovog rada bavit ćemo se konceptima genetskog programiranja, Kartezijevog genetskog programiranja, razvojem agenata kroz generacije, implementacijom specifičnom za ovaj rad, njezinim rezultatima i diskusijom o rezultatima.

2. Igra 2048

U ovom poglavlju ulazimo dublje u pravila i funkcioniranje igre. Fokus će biti stavljen na neke pojedinosti koje su uzete u obzir prilikom implementacije ovog rada.

Odmah se stavlja naglasak na to kako ovaj rad nije napravljen nad originalnom verzijom igre *2048* već nad verzijom koju su razvili Krešo Orešković i Paulo Erak u sklopu projekta na Fakultetu elektrotehnike i računarstva (FER) pod mentorom Markom Đurasevićem.

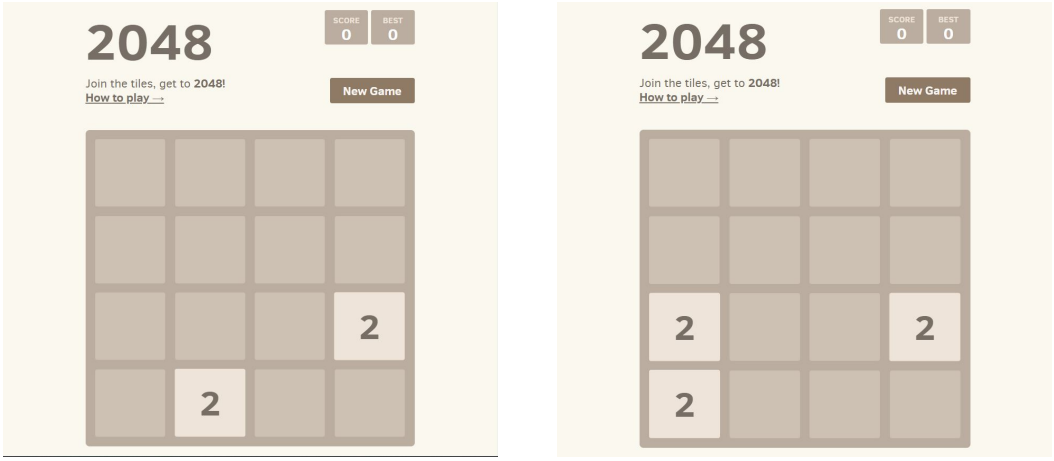
U samom uvodu pokrili smo neke od najvažnijih elemenata i pravila igre koja su sadržana u obje verzije: *Cilj igre je dobiti što veći rezultat koji je sam po sebi zbroj novostvorenih brojeva na ploči ispred nas. Imamo vrlo jednostavne poteze: pomicanje svih dijelova ploče lijevo, desno, gore ili dolje. Prilikom pomicanja, elementi koji su isti i susjedni, a nalaze se jedan drugome na putanji, spajaju se i time otvaraju prostor na ploči. Nakon svakog poteza se na nasumično otvoreno mjesto stavlja novi element.*

Uzevši to u obzir, verzije igre nisu iste i neka pravila se razlikuju. Ta pravila bit će posebno naglašena u potpoglavlju *Verzija igre korištena u radu*.

2.1. Originalna igra *2048*

Originalnu igru *2048* napravio je talijanski programer, Gabriele Cirulli, koji je svoj rad [2] postavio na internetsku uslugu za pohranu i upravljanje verzijama koda, GitHub.

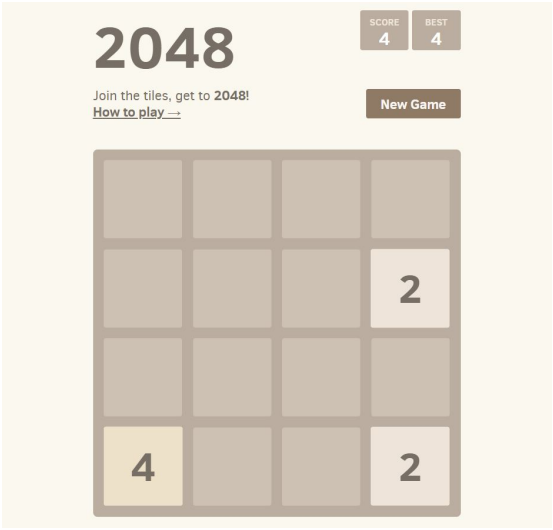
Igra se sastoji od nekoliko grafičkih elemenata: ploča (najčešća dimenzija je 4x4), kvadrati raznih boja koji sadrže brojeve (potencije broja 2) te se nalaze na ploči, brojači koji prikazuju trenutni i najbolji rezultat i gumb za ponovo pokretanje igre.



(a) Početno stanje **(b)** Potez u lijevo

Slika 2.1: Prikaz originalne igre *2048* [1]

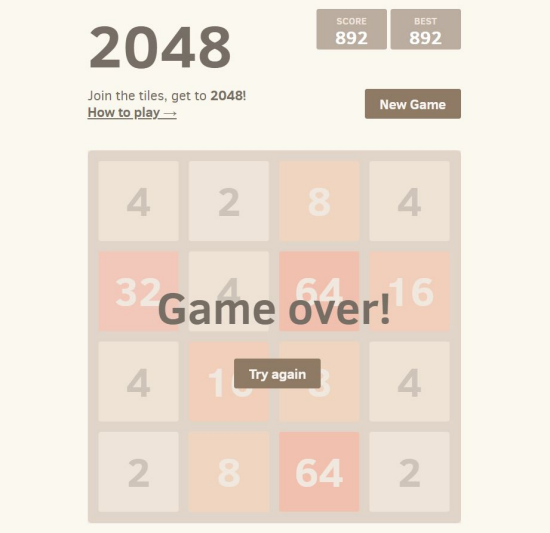
Potezima se svi postojeći elementi pomaknu u željenu stranu koliko i ako mogu, tj. ako im prazan prostor ili drugi elementi to dopuštaju. Primjer poteza može se vidjeti na slikama 2.1(a) i 2.1(b). Ako se dva ista kvadrata nađu jedan drugome na putanji spojit će se i proizvesti novi kvadrat s brojem jednakim zbroju brojeva u kvadratu. Spoj dva kvadrata s vrijednosti 2 prikazan je na slici 2.2.



Slika 2.2: Potez prema dolje, spajanje 2 i 2 [1]

Potezi se ostvaruju pritiskom i puštanjem tipki strelica koje signaliziraju u kojem smjeru želimo pomaknuti sve elemente (gore,dolje,lijevo,desno). Potez se neće izvesti ako se njegovim izvođenjem neće izvesti nikakva promjena ploče, tj. nijedan element se neće pomaknuti zbog otpora drugog elementa ili ruba ploče. Držanjem strelice se

postiže nekoliko rapidnih poteza u tom smjeru. Potezi prestaju otpuštanjem tipke ili ako se tim potezom neće pomaknuti niti jedan element. Kraj igre, Slika 2.3, nastaje ako imamo punu ploču i program prepozna da nijednim potezom nećemo postići promjenu rasporeda elemenata.



Slika 2.3: Kraj igre [1]

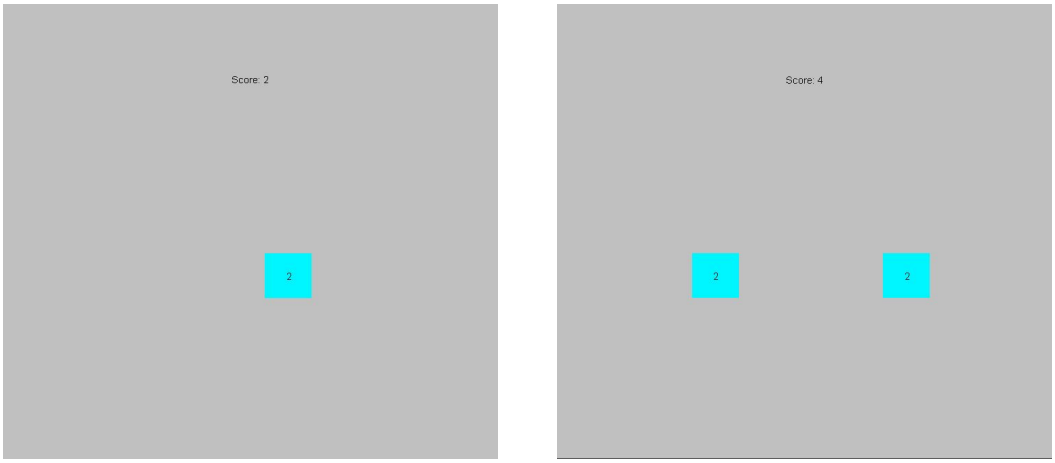
Ako se dogodi da u jednom redu ili stupcu imamo poredak poput: **[2 2 4 8]** i napravimo potez u desno, dobit ćemo red: **[- 4 4 8]**. Dakle spajanje novih elemenata u potezu u kojem su napravljeni nije moguće. Ako imamo red oblika: **[2 2 4 4]**, potezom u desno dobivamo: **[- - 4 8]**. Višestruko spajanje elemenata je moguće te je omogućeno pomicanje u nove praznine u istom potezu.

2.2. Verzija igre korištena u radu

Iako se ovaj rad referencira na originalnu igru, sam rad i njegova implementacija rađena je nad verzijom koja se po nekoliko elemenata razlikuje od originalne igre *2048*.

Korištena verzija sadrži sljedeće grafičke elemente: ploča dimenzija 4x4, kvadrati raznih boja koji sadrže brojeve (potencije broja 2) te se nalaze na ploči, brojač koji prikazuju trenutni rezultat. Elementi se mogu vidjeti na slikama 2.4 i 2.5.

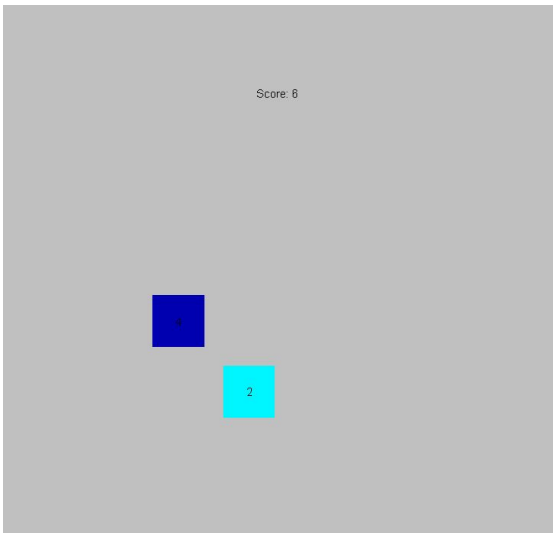
Prva bitna razlika u korištenoj verziji je ta da je rezultat zbroj brojeva na ploči, a ne zbroj brojeva u elementima koje je igrač dobio spajanjem. Iako je funkcionalnost donekle slična kao u početnoj verziji, vrijednost se više ne pripisuje igračevoj sposobnosti spajanja već se pripisuje količini i veličini brojeva na ploči.



(a) Početno stanje **(b)** Potez u lijevo

Slika 2.4: Prikaz korištene igre *2048*

Kretanja, ograničenja i spajanja elemenata na ploči slijede ista pravila kao i u originalnoj verziji igre.



Slika 2.5: Potez prema lijevo, spajanje 2 i 2

Potezi se ostvaruju pritiskom i **držanjem** tipki strelica koje signaliziraju u kojem smjeru želimo pomaknuti sve elemente. Ako se strelica ne drži do prestanka pomicanja svih elemenata postoji šansa da se igra počne nepredvidivo ponašati. Puštanje tipke strelice smatra se da je potez završio, a igra stvara novi element na nasumičnom slobodnom mjestu. Potezi će se izvesti čak iako se ne pomakne nijedan element. Kraj igre nastaje kad se pokuša dodati još jedan element na punu ploču.

Ako se dogodi da u jednom redu ili stupcu imamo poredak poput: **[2 2 4 8]** i

napravimo potez u desno, dobit ćemo red: **[- - - 16]**. Dakle spajanje novih elemenata u potezu u kojem su napravljeni je moguće. Ako imamo red oblika: **[2 2 4 4]**, potezom u desno dobivamo: **[- - 4 8]**. Višestruko spajanje elemenata je moguće te je omogućeno pomicanje u nove praznine u istom potezu.

Funkcioniranje verzije ovisi o grafičkom sučelju, a to stavlja veliko ograničenje na brzinu igranje igre. Potencijalna poboljšanja su smanjenje ovisnosti o grafičkom sučelju i promjena načina izvođenja poteza. To bi značilo da grafičko sučelje postane reprezentacija stanja koje je u pozadini, a ne ključni dio funkcioniranja igre, te da se potezi izvode na pritisak i puštanje tipki, a ne na držanje tipki.

3. Genetsko programiranje

U ovom poglavlju dajemo uvid u osnove genetskog programiranja, a kasnije poglavlje će dati detaljan uvid u posebnu vrstu genetskog programiranja koja je korištena u radu. To je razlog zašto u ovom poglavlju nećemo ulaziti u najmanje detalje već ćemo se samo dotaknuti ključnih pojmova i operacija vezanih za područje genetskog programiranja.

Genetsko programiranje je tehnika evolucijskog algoritma koji automatski rješava probleme bez potrebe za tim da korisnik unaprijed zna ili odredi strukturu rješenja [7].

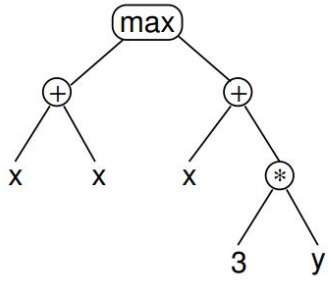
Ključni koraci genetskog programiranja mogu se svesti na sljedeće:

Algorithm 1 Genetsko programiranje [7]

Stvori nasumičnu početnu populaciju programa pomoću danih funkcija, varijabli i konstanti
repeat
    Pokreni svaki program i dodjeli mu dobrotu
    Odaberi jednog ili dvoje roditelja na osnovu odabranog načina selekcije
    Stvori novi ili nove programe primjenjujući genetske operacije s određenom vjerojatnošću
until Pronalazak zadovoljavajućeg rješenja ili ispunjenje nekog drugog uvjeta zaustavljanja (npr. dostignut maksimalan broj generacija)
return Najbolji program

U genetskom programiranju sve kreće od početne, često nasumične, populacije programa te se pokušava doći do što boljeg odgovora na početni zadatak kroz različite izmjene populacije tijekom generacija programa.

Sintaksna stabla su čest oblik programa u genetskom programiranju. Unutarnje čvorove stabla čine funkcije, a njihova djeca postat će varijable koje ulaze u zadanu funkciju. Terminalni čvorovi (listovi) su konstante ili varijable nad kojima će se provesti funkcije sadržane u unutarnjim čvorovima. Primjer programa u obliku sintaksnog stabla možemo vidjeti na slici 3.1.



Slika 3.1: Sintaksno stablo (program) [7]

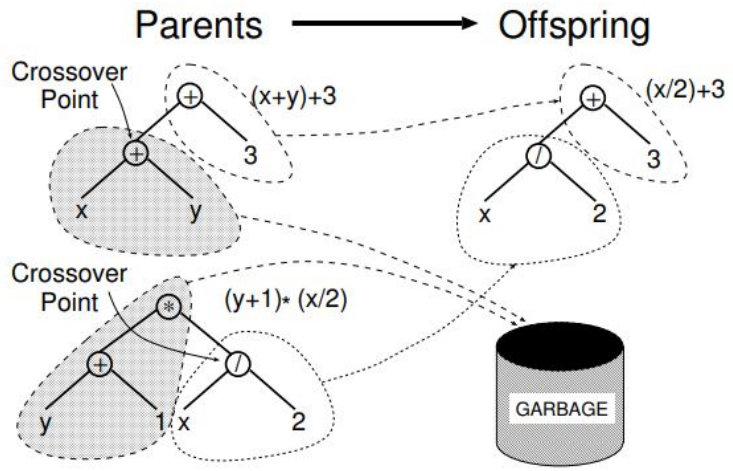
Promjene su ključan dio razvoja programa, ali kao i u prirodi, promjene su nasumične i ne možemo garantirati da svaka promjena dovodi k boljem rješenju. Nasumičnost ima i svoje jače strane te često dovodi do novih i neočekivanih rješenja koja zaobilaze uobičajene poteškoće na koje nailaze deterministične metode [7].

Prije uvođenja promjena vrši se selekcija. **Selekcija** je proces odabira roditelja uz pomoć kojih će se, kroz genetske operacije, uvesti novi programi u populaciju. U kasnijem poglavlju detaljnije ćemo opisati različite načine na koje se selekcija može izvesti. Bitno je da roditelji imaju veću šansu biti odabrani što su prilagođeniji našem zadatku. Osim toga, važno je dati mogućnost odabira manje prilagođenijih programa jer oni mogu sadržavati dijelove koji će dati bolje rezultate za generalni zadatak.

Promjene između generacija događaju se zbog dvije ključne genetske operacije:

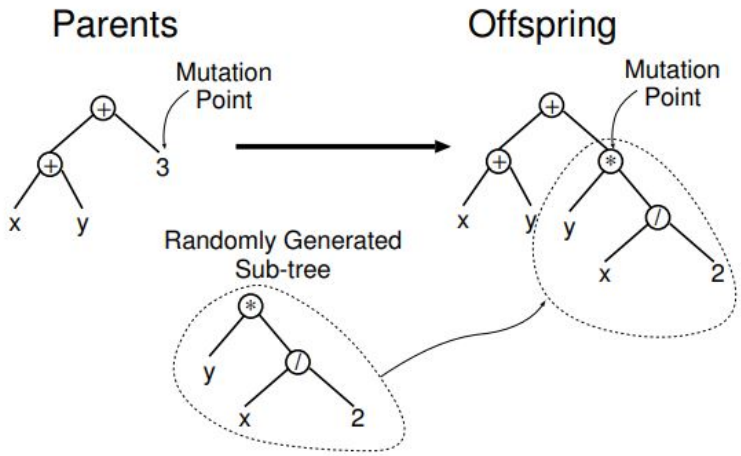
- **Križanje** (eng. *Crossover*): Stvaranje novog programa/djeteta kombiniranjem nasumičnih dijelova roditeljskih programa
- **Mutacija** (eng. *Mutation*): Stvaranje novog programa/djeteta nasumičnim mijenjanjem dijelova roditeljskog programa

Oblik stabla čini križanje i mutaciju jednostavnijim i intuitivnijim.



Slika 3.2: Križanje dviju kopija roditeljskih programa [7]

Križanje (Slika 3.2) možemo shvatiti kao miješanje i/ili zamjenu podstabala roditeljskih programa. Mutaciju (Slika 3.3) možemo shvatiti kao promjenu čvora gdje promjena može biti u obliku promjene funkcije/varijable na mjestu mutacije ili dodavanja u potpunosti novog podstabla na mjesto mutacije.



Slika 3.3: Mutiranje roditelja dodavanjem novog podstabla na mjesto mutacije [7]

Napredak pojedinih programa prati se kroz generacije preko funkcije **dobrote**. **Dobrota** (eng. *fitness*) je vrijednost koja ukazuje na to koliko je neki program dobar ili loš u rješavanju danog zadatka. Način računanja dobrote ovisni o tipu zadatka kojeg imamo te se prema tome može računati na različite načine.

Dobrota nas dovodi do pojma elitizma u kojem ona igra glavnu ulogu. **Elitizam** je strategija kojom uspoređujemo dobrote programa unutar generacije te najbolji ili

nekoliko najboljih programa trenutne generacije prebacujemo u sljedeću generaciju. Imamo veliku korist od ovakve strategije jer sprječava da najbolji program generacije bude lošiji od najboljih programa prošlih generacija. Sprječavamo nazadovanje kroz generacije.

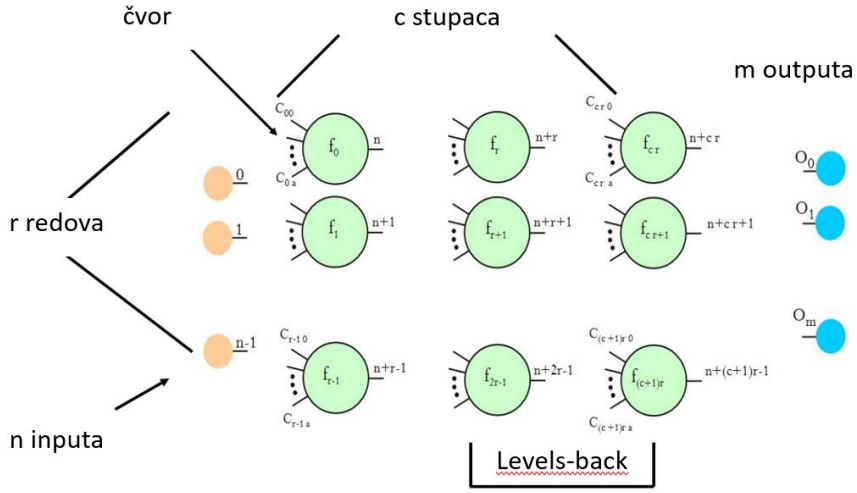
Na kraju, mora postojati i **uvjet zaustavljanja** koji će reći da nema potrebe za daljnjim generacijama. **Uvjet zaustavljanja** može biti određena vrijednost dobrote ili dostizanje određenog broja generacija, ali može biti i neki drugi specifičan uvjet.

4. Kartezijevo genetsko programiranje

Kartezijevo genetsko programiranje (nadalje poznato kao CGP) oblik je genetskog programiranja koji je korišten u radu. Ovo poglavlje proći će kroz detalje koje ga čine posebnim i koji su bili bitni pri implementaciji. Osnove genetskog programiranja prenose se, ali u nešto drukčijim oblicima koji će se obraditi u sljedećim poglavljima.

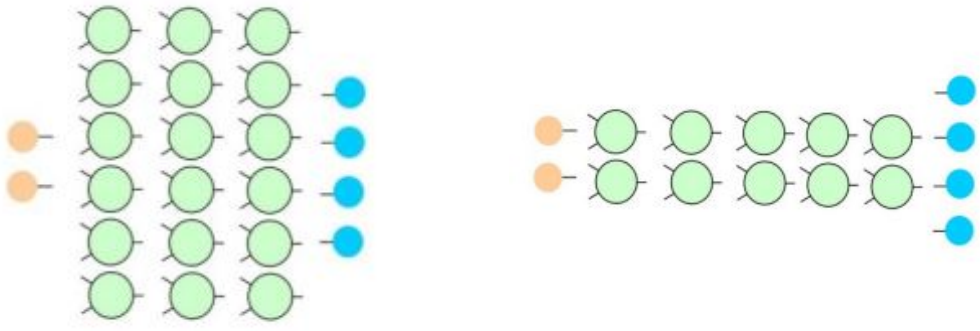
4.1. CGP Mreže

U Kartezijevom genetskom programiranju programi su prikazani kao dvodimenzionalne mreže čvorova. Mreža (Slika 4.1) je sastavljena od **c stupaca** i **r redova** koji dolaze uz **n ulaza** (eng. *inputs*) i **m izlaza** (eng. *outputs*).



Slika 4.1: CGP mreža [6]

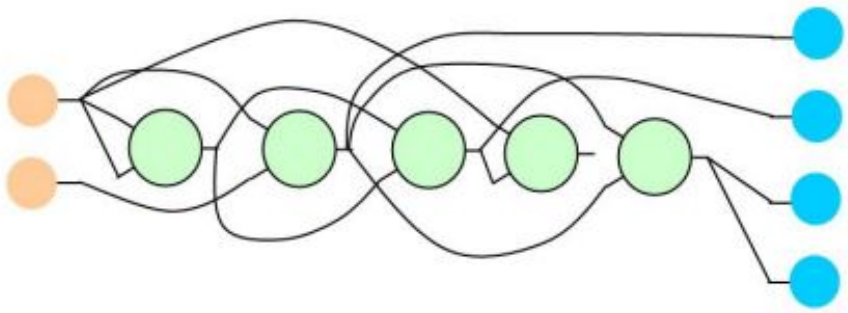
Ovisno o broju stupaca i redova koje zadamo možemo imati različite oblike mreža. Možemo imati široke i plitke mreže (Slika 4.2 (a)), možemo imati uske i duboke mreže (Slika 4.2 (b)) ili možemo imati jednostavnu "kvadratnu" mrežu s jednakim brojem stupaca i redova.



(a) Široka i plitka mreža [6] **(b)** Uska i duboka mreža [6]

Slika 4.2: Mogući oblici CGP mreža

Oblik i veličina mreže trebali bi biti prilagođeni zadatku s obzirom na to da premala ili prevelika mreža mogu biti izvor problema ili slabog treniranja programa. Jedna od preporučenih i zanimljivijih mreža je mreža s jednim redom i proizvoljnim brojem stupaca koja više nalikuje lancu. U ovom radu najviše se radilo s ovakvim oblikom mreže, a primjer oblika može se vidjeti na slici 4.3.



Slika 4.3: CGP mreža s jednim redom [6]

4.2. Čvorovi mreže

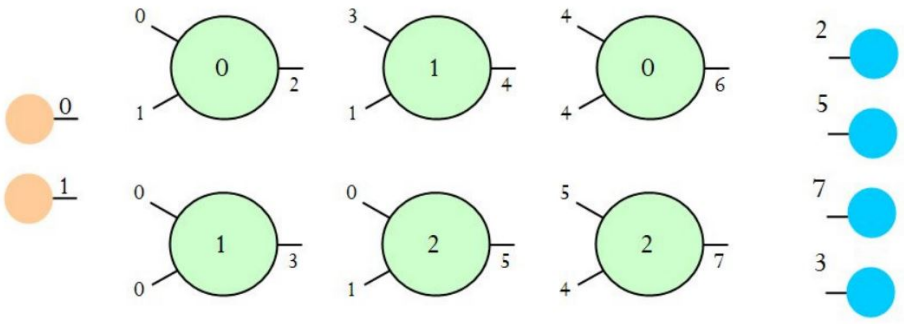
Čvorovi su sastavljeni od gena koji tvore genotip, a zapravo su brojevi koji sugeriraju od kuda će čvor dobiti podatke, koje operacije će primijeniti nad podacima i od kud će doći izlazni podaci [5].

Veze između čvorova upravo su geni veza i oni nam govore iz kojeg ćemo čvora ili ulaza dobiti podatke. Broj gena veza koje sadrži svaki čvor mreže jednak je najvećem broju parametara koje traži neka od funkcija koja se može pojaviti u mreži. Pošto su svi čvorovi i ulazi numerirani ti brojevi su nam adrese. Geni veza najčešće mogu sadržavati samo adrese čvorova i ulaza iz stupaca s lijeve strane čvora za koji se adresiraju ulazni parametri, ali postoje radovi i primjeri gdje se dopuštaju ciklusi (ciklički CGP) micanjem ove restrikcije [5]. Parametar ***Levels-back (l)*** govori nam iz kojih stupaca s lijeve strane trenutne pozicije možemo uzeti adresu. Primjerice za $l = 1$ adrese možemo uzimati samo iz lijevog neposrednog stupca. Za $l = n_c$ (l = broj stupaca) možemo dohvatiti bilo koju adresu iz mreže u bilo kojem stupcu.

Sve funkcije koje se mogu pojaviti u čvorovima zapisane su u posebnu tablicu uz svoju brojevnu reprezentaciju. Brojevna reprezentacija postaje adresa funkcije i ona se zapisuje kao funkcijski gen koji je ključni dio svakog čvora mreže. Nakon što čvor dobije potrebne parametre preko gena veza, dohvaća se funkcija pod zadanom adresom i ona se izvršava nad danim parametrima.

Izlazni geni nalaze se na krajnjoj desnoj strani mreže. Oni adresiraju čvorove ili ulaze iz kojih će se uzimati vrijednosti za konačni izlaz iz mreže. Za njih vrijede ista pravila kao i za gene veza. Na osnovu izlaza iz mreže poduzima se akcija jedinstvena za taj izlaz, čineći izlaz iz mreže odlukom mreže.

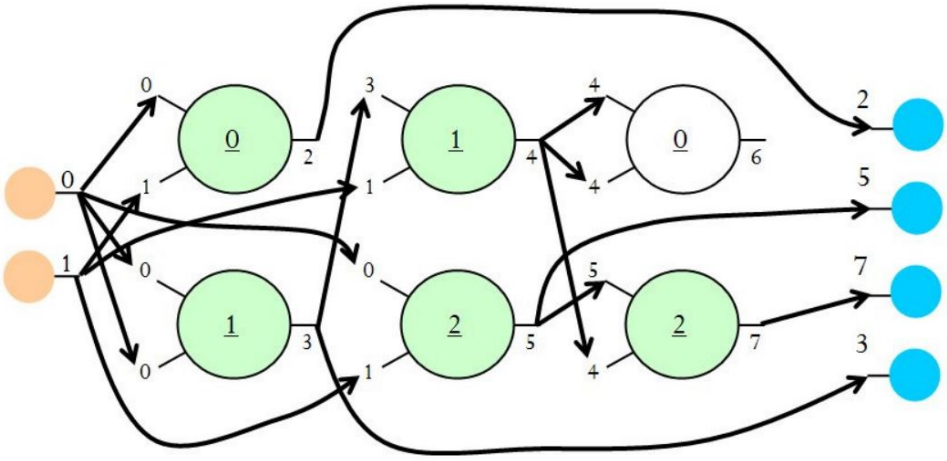
Primjer mreže čvorova sa zadanim genima i vezama možemo vidjeti na slikama 4.4. i 4.5. uz funkcije u tablici 4.1.



Slika 4.4: CGP mreža s genima [6]

0	1	2	3
+ Zbroji parametre	- Oduzmi parametre	* Pomnoži parametre	/ Podijeli parametre

Tablica 4.1: Tablica funkcija (Lookup table of functions)



Slika 4.5: CGP mreža s genima i vezama [6]

4.3. Genotip i fenotip

Pogledom na sliku 4.3. uviđamo kako je gen u trećem stupcu i prvom retku nekorišten, tj. on ne pridonosi niti jednom izlazu. On se naziva *nekodirajućim genom*. Dolazimo do pojmova **genotipa** i **fenotipa**. *"Dekodiranje genotipa rezultira programom koji se naziva fenotip [5]."* **Genotip** čine svi geni unutar mreže dok **fenotip** čine samo aktivni, *kodirajući*, geni koji utječu na izlaze iz mreže. U većim mrežama broj nekodirajućih gena može biti znatno veći od broja kodirajućih. Primjerice J. F. Miller i S. L. Smith pokazali su kako je u genotipu s 4000 čvorova postotak nekodirajućih čvorova oko 95% [4].

Dekodiranje genotipa naziv je za proces kojim ćemo proći kroz genotip i utvrditi koji geni su kodirajući, a koji ne. Princip procesa je krenuti od izlaznih gena i kretati se po čvorovima čije adrese se nalaze u genima veza dok ne dođemo do ulaza u mrežu. Pseudokodom u Algoritmu 2 prikazan je proces dekodiranja.

Algorithm 2 Dekodiranje genotipa

```
Lista čvorova
for svaki izlazni gen do
    if adresa = adresa čvora then
        Označi čvor kao kodirajući
    end if
end for
for svaki čvor u Lista čvorova krenuvši od zadnjeg do
    if čvor je kodirajući then
        for gen veze u čvoru do
            if adresa = adresa čvora then
                Označi čvor kao kodirajući
            end if
        end for
    end if
end for
return Lista čvorova
```

4.4. CGP kao niz brojeva

Tijekom opisa Kartezijevog genetskog programiranja susreli smo se već nekoliko puta s brojevima i brojevnom reprezentacijom, a svi brojevi zapravo su adrese. Upravo zbog tih brojeva postoji vrlo jednostavan zapis genotipa i to u obliku niza brojeva. U tom nizu moramo znati raspoznati gene - funkcijski geni, geni veza i izlazni geni. Iako u nizu nećemo imati nikakvu posebnu reprezentaciju znat ćemo točno koji je gen koji ako znamo koliko parametara svaki čvor treba primiti i koliko izlaza imamo. Uzmemo li mrežu na Slici 4.4 kao primjer, možemo konstruirati reprezentaciju mreže krenemo li od gornjeg lijevog čvora:

0 0 1

U čitanje niza ulazimo sa znanjem da je najveći broj parametara potreban nekoj od mogućih funkcija dva te da imamo dva izlaza iz mreže. Slijeva nadesno imamo: funkcijski gen s brojem *0* koji adresira funkciju zbrajanja u tablici funkcija (Tablica 4.1), prvi gen veze koji adresira od kud dobivamo prvi parametar što bi u ovom primjeru bio *0* - prvi ulaz/input, drugi gen veze koji adresira od kud dobivamo drugi parametar što

bi u ovom primjeru bio *1* - drugi ulaz/input. Nastavimo li dalje prolaziti po čvorovima dobivamo sljedeći niz:

0 0 1 1 0 0 1 3 1 2 0 1 0 4 4 2 5 4 2 5 7 3

Pogledamo li u nizu treći čvor vidimo kako njegov prvi gen veze sadrži broj *3*. Pošto imamo samo dva ulaza i njihove adrese su *0* i *1* to znači kako je adresiran jedan od čvorova; u ovom slučaju drugi čvor u prvom stupcu. Prema tome, adrese čvorova nastavljaju se na adrese ulaza jer oni sami postaju ulazi za neki sljedeći čvor ili izlaz. Adrese se povećavaju s lijeva nadesno i od vrha stupca prema dnu stupca.

Zadnji dio niza koji je ovdje posebno odvojen od ostatka niza čine izlazni geni. U ovom slučaju na izlaz iz mreže spojeni su prvi, drugi, četvrti i šesti čvor.

Ovakav oblik CGP mreže/programa izuzetno je koristan i pojednostavljuje inicijalizaciju, izvršavanje i promjenu programa. Promjene nad populacijom i pojedinim programima obrađuju se kasnije u zasebnim poglavljima.

4.5. Inicijalizacija programa/populacije

Inicijalizacija, kao i u običnom genetskom programiranju, podrazumijeva stvaranje nasumične populacije programa. U ovom slučaju to bi bili nizovi nasumičnih brojeva.

Iako brojevi jesu nasumični, postoje pravila koja smo ranije postavili i koja moramo poštivati. Nizovi moraju biti smisleni i slijediti formu sličnoj onoj iz primjera gdje znamo na kojim mjestima se nalaze funkcijski geni, na kojima geni veza i na kojima izlazni geni. Osim forme, adrese koje se mogu pojaviti na određenim pozicijama moraju biti ograničene na isti način kao u mreži. Funkcijski geni smiju imati samo adrese iz tablice funkcija, ali te adrese mogu se pojaviti u bilo kojem dijelu niza gdje se nalaze funkcijski geni. Geni veza (osim ako se radi o cikličkom CGP-u) smiju imati samo adrese čvorova iz stupaca lijevo od njihovog stupca, naravno s danim ograničenjem Levels-back. Informaciju o tome u kojem se stupcu nalazi koji čvor dobivamo iz kombinacije informacija: broj redova mreže, tj. koliko čvorova imamo u svakom stupcu, i praćenjem rednog broja čvora u kojem se nalazimo. Izlazni geni smiju sadržavati adrese svih čvorova i ulaza u mreži.

Slijedeći ova pravila možemo inicijalizirati cijelu populaciju programa jednostavnog zapisa u kratkom roku.

U nastavku možemo vidjeti jednu takvu populaciju mreža koje imaju šesnaest ulaza, četiri izlaza, širinu jedan, dubinu osam i maksimalan broj parametara funkcija dva. Radi preglednosti nizovi su podijeljeni na čvorove i izlaze, funkcijski geni su podcrtani, a izlazni geni su **podebljani**.

4 6 14	5 15 16	4 0 8	4 15 15	2 8 3	4 7 10	1 17 3	3 13 3	**13 23 4 10**

4 14 5	3 15 6	5 1 7	4 7 12	2 0 15	1 19 3	1 16 9	0 14 3	**20 22 1 11**

4 14 7	3 1 4	5 13 11	1 15 1	2 6 2	4 12 9	1 1 13	5 12 3	**7 23 20 20**

2 15 8	1 2 7	2 17 9	1 4 15	3 8 1	5 6 18	2 19 10	3 14 19	**0 5 10 21**

4.6. Izvođenje programa

Sad kad smo upoznati s oblikom programa i pojmom fenotipa možemo govoriti o izvođenju programa. Prije izvođenja programa potrebno ga je dekodirati kako bi radili samo s onim čvorovima koji nam utječu na izlaz iz mreže. Nakon što smo dobili fenotip, počinjemo s procesom koji je po prolazu kroz mrežu obrnut od dekodiranja. Pod pretpostavkom da radimo s necikličnom mrežom, imamo unaprijednu (eng. feed-forward) mrežu koja obrađuje i propagira podatke od ulaza prema izlazu. Izvođenje programa se zbog toga odvija stupac po stupac od ulaza prema izlazu kako ni u jednom trenutku ne bi došli do čvora koji ima nedefinirane ulazne parametre. Za svaki čvor slijedimo njegove gene veza do parametara koje uvrštavamo u funkciju adresiranu u funkcijskom genu. Rezultat funkcije zapamtimo kao izlaz iz čvora. Nastavljamo dalje čvor po čvor dok ne dođemo do izlaznih gena koji dohvaćaju izlaze čvorova ili ulaze u mrežu.

Ovime smo pokrenuli i izvršili program koji smo sastavili. Kao rezultat izvođenja programa dobili smo izlaz koji se može interpretirati kao odgovor na zadani zadatak ili dalje koristiti u svrhu rješavanja zadatka.

5. Dobrota i selekcija

U ovom poglavlju zaći ćemo dublje u pojam dobrote i proces selekcije te predložiti nekoliko načina implementacije.

5.1. Dobrota

U 3. poglavlju dotakli smo se pojma dobrote (eng. fitness) te dali kratki i jednostavan opis koji proširujemo ovdje. Dobrota je vrijednost koju koristimo kako bi mogli reći koliko je program dobar u rješavanju danog zadatka i koju koristimo pri međusobnom uspoređivanju različitih programa. Dobrota nema univerzalan način računanja niti univerzalan okvir vrijednosti, ali očekuje se da je program bolji u rješavanju zadatka što je dobrota veća.

U nadziranom učenju naši programi, uz ulazne vrijednosti, dobivaju i očekivane izlazne vrijednosti. Očekivane izlazne vrijednosti služe za uspoređivanje s izlaznim vrijednostima naših programa. Usporedbom dobivamo pogrešku (eng. error) koja nam govori koliko se rezultati naših programa razlikuju od očekivanog izlaza. Pogrešku možemo koristiti kako bi izračunali dobrotu. Način pretvorbe pogreške u dobrotu ostavlja se na izbor osobi koja će implementirati nadzirano učenje, ali ovdje će se predložiti jedna jednostavna operacija. Operacija u pitanju je cjelobrojno dijeljenje broja jedan s pogreškom ($\frac{1}{error}$). Operacija čini dvije vrijednosti obrnuto proporcionalne i u potpunosti je prihvatljiva opcija.

U podržanom učenju naši programi dobivaju niz ulaznih vrijednosti prema kojima donose "odluke". "Odluke" se mogu ili kazniti ili nagraditi. Kazna i nagrada nisu odmah uočljivi što znači da se program neće ocijeniti pri primitku prve kazne/nagrade. Kazne i nagrade se akumuliraju kroz više "odluka". Primijenimo li primjer igre, naši programi će raditi poteze kroz runde i bit će kažnjeni ili nagrađeni za njih, a na kraju igre dobivaju konačan rezultat koji je procjena koliko su bili uspješni u igri. Ako se radi o zadatku u kojem postoje elementi nasumičnosti bilo bi dobro svaki program nekoliko puta provesti kroz zadatak kako bi se uvjerili da se uvjeti u zadatku nisu poklopili na

takav način da program dobije upravo takvu procjenu uspješnosti.

Elitizam je pojam koji smo obradili u 3. poglavlju. U tom poglavlju je pojam dovoljno dobro objašnjen i ovdje ćemo samo napraviti nadogradnju vezanu za CGP. Ako se utvrdi da roditelj i dijete imaju istu dobrotu, dijete će se propagirati u sljedeću generaciju [5][6]. Testovima je dokazano kako je propagacija djeteta u takvim slučajevima izuzetno korisna i efektivna u dobivanju boljih rezultata [5].

Dobrota je izuzetno korisna, ali to ne znači da nas ne može zavarati. Problemi koje ćemo navesti nisu toliko vezani uz sam postupak dobivanja dobrote, oni mogu doći iz različitih dijelova sustava koji razvija programe, ali ti problemi utječu na dobrotu koja nas zatim može zavarati i potaknuti na krivi zaključak. Neki od problema koji se mogu vezati uz dobrotu su pretreniranost i nasumičnost zadatka.

Pretreniranost možemo najviše povezati uz nadzirano učenje. Kod pretreniranosti možemo dobiti sjajne rezultate za podatke nad kojima učimo, ali loše rezultate za neviđeni skup podataka. Tu nas dobrota tehnički ne vara, programi sjajno rade na podskupu za učenje, ali programi ne znaju riješiti generalni tip zadatka. To se događa kad se naši programi previše razviju u smjeru rješavanja jako specifičnih zadataka. Posljedica toga je ta da se pretrenirani programi kroz elitizam i selekciju, zbog visoke dobrote, propagiraju dalje i pretreniraju buduće generacije. Problem se može spriječiti boljom generalizacijom skupa za učenje i ograničavanjem broja generacija ili dobrote.

Pod nasumičnost zadatka misli se na problem koji se javlja kod podržanog učenja kad zadatak ima neke elemente nasumičnosti. Kao što je već ranije spomenuto ne znamo kako će se zadatak dalje razvijati i može se dogoditi da u ponavljanjima dođu izrazito teški ili izrazito laki zadaci. Primjerice, može nastupiti slučaj gdje generalno loš program uspješnije rješava svoje zadatka nego što generalno bolji program rješava svoje, a razlog tome je taj da su se nekoliko puta za redom poklopili posebni uvjeti koji su to omogućili. Posljedica ovog problema je propagacija lošijeg programa u sljedeće generacije. Rješenje ovom problemu može biti provođenje programa kroz zadatak više puta ili drukčiji pristup računanju dobrote.

5.2. Selekcija

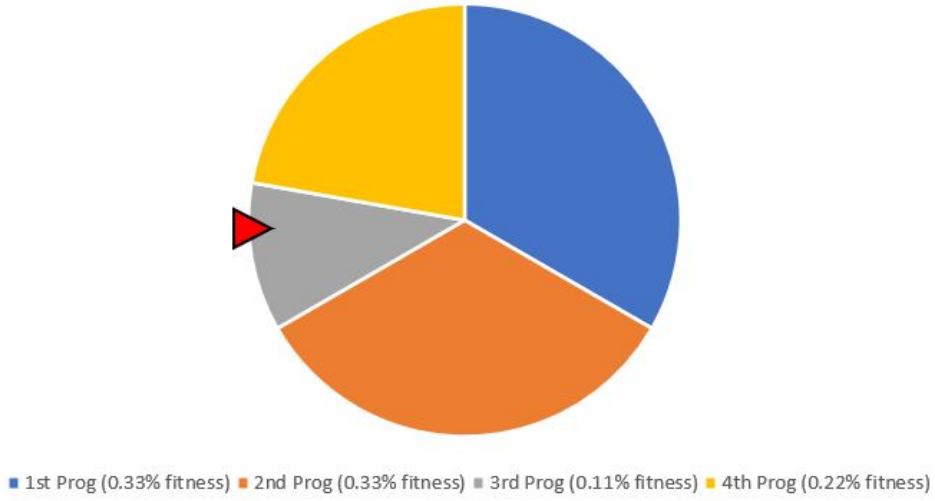
Selekciju smo spomenuli u 3. poglavlju, a ovdje ćemo ju dodatno obraditi i navesti neke načine implementacije.

Selekcijom odabiremo programe (roditelje) koji će zajedno biti temelji za nove programe koji će se uvesti u sljedećim generacijama. Povlačimo poveznicu s prirodnim svijetom gdje najuspješnije jedinke dobivaju prioritet prilikom parenja i tako osigura-

vaju prenošenje svojih gena u buduću generaciju. No, kao i u prirodnom svijetu, ne moraju se samo najbolje jedinke pariti i prenositi svoje gene. To pridonosi raznolikosti unutar vrste, a same slabosti u jednoj situaciji mogu postati snage u drugoj. Po uzoru na opisan proces, imamo nekoliko načina na koje možemo implementirati selekciju.

Prvi i najjednostavniji način koji zahtijeva samo sortiranje programa po dobroti bio bi da uzmemo najbolje programe. Naravno varijante ove implementacije su itekako moguće: jedan najbolji program i jedan najlošiji; par programa s početka, par iz sredine i par s kraja sortiranog niza; jedan program s početka i jedan program iz sredine sortiranog niza... Učinkovitost ovakve selekcije upitna je i u ovom radu neistražena tako da se predlažu neki od sljedećih načina implementacije.

Selekcija proporcionalna dobroti može se zamisliti kao kotač s kazaljkom koji je podijeljen na dijelove. Svaki dio je reprezentacija jednog programa. Veličina dijela proporcionalna je udjelu dobrote programa u zbroju dobrota svih programa. Primjerice, ako zbrojimo dobrote četiri programa i ako je dobrota jednog program pridonijela trećini zbroja, njegov dio na kotaču pokrivat će jednu trećinu kotača.



Slika 5.1: Vizualni prikaz selekcije proporcionalne dobroti

Nakon što izračunamo udio svakog programa, nasumično odabiremo programe. Vjerojatnost odabira svakog programa jednaka je njegovom udjelu. Vizualiziramo li taj događaj, zavrtimo kotač i odaberemo program na koji pokazuje kazaljka nakon što se kotač zaustavi. Ovakav oblik implementacije selekcije pokazao se izrazito uspješnim i preporučuje se.

Zadnji način implementacije selekcije koji ćemo opisati može biti nadogradnja prethodnom načinu, a naziva se turnir. Nakon što odaberemo nekoliko programa, sa-

svim nasumice ili po udjelu dobrote, uspoređujemo programe međusobno i najbolji postaju roditelji. Ovaj oblik implementacije daje priliku i slabijim programima jer oni trebaju biti najbolji samo u svojoj skupini, a ne u cijeloj populaciji programa.

Selekcija nema nekih osobitih problema koje možemo popraviti. Uvijek postoji mogućnost da ćemo za roditelje odabrati slabije programe ili da će se često odabrati isti programi. Implementacijom gore navedenih strategija pokušava se smanjiti učestalost takvih situacija, a ipak zadržati raznolikost populacije. Ispravna implementacija strategija ključna je za ispravno funkcioniranje selekcije.

Ako se tijekom selekcije u CGP-u dogodi slučaj gdje roditelj i dijete imaju istu vrijednost po kojoj se uspoređuju (npr. dobrota), a izbor je između jednog ili drugog, izabrat će se dijete zbog istog razloga koji je naveden za elitizam u prošlom potpoglavlju [5][6].

6. Križanje

Nakon što smo selekcijom dobili roditelje budućih programa, operacijom križanja pokušavamo stvoriti djecu isprepletući roditeljske gene. U CGP-u ne radimo sa stablima već s mrežama što nas potiče na drukčiji pristup od onog opisanog u 3. poglavlju. U ovom poglavlju opisuje se križanje u CGP-u, utjecaj operacije na uspješnost programa i neki od načina implementacije.

Križanje u CGP-u jako je zanimljiva tema. Tema koja se još uvijek istražuje i koja dobar dio vremena nije bila u fokusu. Primjenom operacije u nekim slučajevima dobivamo bolje rezultate nego kad se operacija ne primjenjuje, ali u nekim slučajevima i gore. Način implementacije znatno utječe na djelotvornost operacije [5][6]. Zato se križanju, posebice u CGP-u, pristupa jako pažljivo i nerijetko se izbjegava. Križanje može biti snažan alat koji će spojiti najbolje dijelove roditelja ili alat koji će poremetiti ono što je roditelje činilo uspješnim.

Originalni pokušaj ostvarenja križanja u CGP-u donekle je jednostavan. U mreži možemo odabrati nasumični gen te na mjestu odabranog gena presjeći obje kopije roditelja. Potom uzimamo prvi dio prve kopije i spajamo ga s drugim dijelom druge kopije, a zatim spajamo drugi dio prve kopije s prvim dijelom druge kopije. Vizualizaciju ovog procesa možemo vidjeti na slikama 6.1(a)-6.3(b).

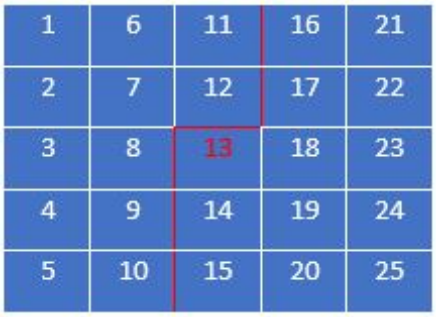
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

(a) Kopija prvog roditelja

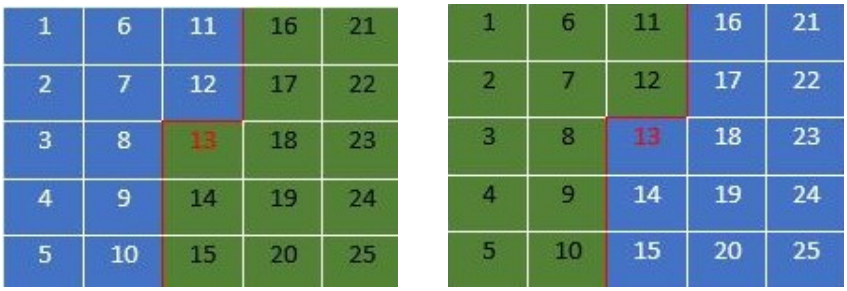
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

(b) Kopija drugog roditelja

Slika 6.1: Početne mreže roditelja



Slika 6.2: Odabir gena za presjek



(a) Prvo dijete **(b)** Drugo dijete

Slika 6.3: Djeca nastala spajanjem dva dijela roditelja

Ovakav oblik križanja pokazao se izrazito destruktivnim u drugim radovima [5]. Ipak, ovakav oblik križanja korišten je u ovom radu na mrežama širine jedan. Učinak takvog križanja na žalost nije provjeren u radu, križanje je provedeno u svim testovima. Ostavlja se kao potencijalna tema daljnjeg istraživanja.

Tijekom godina razvijeno je nekoliko drugih metoda među kojima se ističe metoda zamjene podgrafova aktivnih čvorova [3]. Metoda je slična gore navedenoj metodi, ali ne dijelimo i ne spajamo sve gene grafa već samo one kodirajuće/aktivne što bi trebalo smanjiti destruktivnost i poremećaje u novim mrežama.

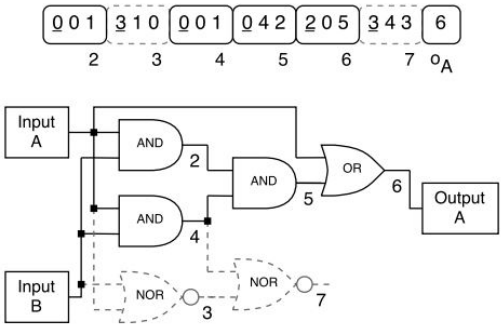
7. Mutacija

Križanjem nastaju djeca koja sadrže gene oba roditelja, ali konstantnim korištenjem istog genetskog materijala možemo postići samo određene rezultate. Mutacija nam stoji na raspolaganju kao operacija koja uvodi nove gene u populaciju. Novi geni mogu ili pospješiti ili pogoršati rezultate populacije. Ako se novi geni uvode u odgovarajućoj količini konačni učinak je pozitivan; u suprotnom je destruktivan. Promjene u bazi gena nas mogu izbaciti iz neoptimalnih rješenja u kojima bi inače ostali. Mutacija je jako korisna i važna operacija u genetskim algoritmima, ali ona je još značajnija u CGP-u gdje se koristi kao primarna genetska operacija s obzirom na nepouzdane rezultate križanja. U ovom poglavlju opisujemo mutaciju u CGP-u, utjecaj operacije na funkcioniranje mreže i načine implementacije.

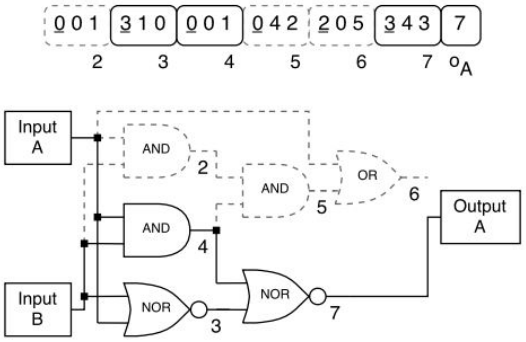
Mutacija u CGP-u podrazumijeva odabir nasumičnih gena u mreži te njihovu izmjenu u neku drugu valjanu vrijednost. S obzirom na to da su geni samo adrese funkcija ili parametara možemo ih izmijeniti u drugu valjanu adresu. Kako bi adresa bila valjana mora poštivati pravila koja smo definirali u poglavlju 4: funkcijski geni smiju poprimiti samo vrijednosti adresa navedenih u tablici funkcija, geni veza ne smiju poprimiti vrijednost adresa čvorova iz stupca u kojem se nalaze ili iz stupaca desno od njih, izlazni geni mogu poprimiti adresu bilo kojeg čvora ili ulaza. Postupak je donekle sličan inicijalizaciji programa, ali na razini zasebnih gena.

Izmjena jednog ili nekoliko gena u mreži može imati puno veći utjecaj nego što bi se očekivalo u običnom genetskom programiranju. Pošto imamo kodirajuće i nekodirajuće gene moramo razmatrati utjecaj mutacije na obje vrste.

Kod mutacije kodirajućeg gena, posebice kodirajućeg gena veze ili izlaznog gena, može doći do velike promjene fenotipa. Geni koji su prije bili nekodirajući mogu postati kodirajući, a kodirajući geni mogu postati nekodirajući kroz samo malu promjenu toka podataka od ulaza do izlaza u mrežu. Na slikama 7.1 i 7.2 možemo vidjeti primjer promjene fenotipa uz prikaz logičkih sklopova kao načina bolje vizualizacije promjene. Pune linije predstavljaju kodirajuće gene, a isprekidane linije predstavljaju nekodirajuće gene.



Slika 7.1: Logički sklop prije mutacije izlaznog gena [5]



Slika 7.2: Logički sklop nakon mutacije izlaznog gena iz 6 u 7 [5]

Mutacija nekodirajućeg gena na prvu ne pokazuje rezultate, ali je iznimno bitna. Pokazali smo kako izmjena gena fenotipa može bitno utjecati na sastav fenotipa tako da nekodirajući geni postanu kodirajući. Prema tome, svaki nekodirajući gen ima šansu postati kodirajući i sve izmjene nad njim mogu postati korisne ili štetne za rad programa. Iako nam učinak izmjene nije poznat u trenutku uspoređivanja, ovo je jedan od razloga zbog kojih u elitizmu i selekciji odabiremo djecu, a ne roditelje, ako se ispostavi da imaju jednaku dobrotu [5][6].

Jedan od oblika implementacije mutacije programa izmjena je nasumičnih gena dok se ne izmijeni određen postotak genotipa [5]. Postotak određuje osoba koja radi implementaciju. Prilikom određivanja postotka važno je imati na umu da prevelika mutacija dovodi do destruktivnih posljedica za populaciju programa.

"Single mutation" naziv je za implementaciju mutacije gdje se izmjena gena provodi dok se ne izmjeni kodirajući gen [6]. Ova implementacija cilja na promjenu fenotipa i ne postoji ograničenje na broj izmjena koje će se izvesti do ispunjenja uvjeta.

8. Implementacija

U ovom poglavlju obrađujemo implementacijske detalje sustava baziranog na CGP-u korištenog u radu i promjene učinjenje u verziji korištene igre.

8.1. Osnovne informacije

Čitava implementacija napravljena je u programskom jeziku Java (verzija 19.0.1). Sustav baziran na CGP-u napravljen je u obliku klase imena *CGP* i ukomponiran je u paket igre. Sustav stvara i koristi dodatne dvije txt datoteke u kojima se drže rezultati treniranja (*scores.txt*) i zadnja populacija programa s kojom je sustav radio (*population.txt*). Programi se zapisuju u obliku niza brojeva po uzoru na zapis opisan u poglavlju 4 uz parametar l=n_c. Modifikacije verzije igre napravljene su u postojećim klasama.

Sustav za treniranje programa koncipiran je poput podržanog učenja s brojem generacija kao uvjetom zaustavljanja. Program nakon kraja igre dobiva rezultat (eng. score) koji služi kao dio dobrote programa. Svaki program ima unaprijed zadan broj ponavljanja igranja igre. Ponavljanja su uvedena zbog elemenata nasumičnosti igre i njima se pokušava smanjiti utjecaj elemenata na ocjenu uspješnosti programa. Konačna ocjena uspješnosti programa bit će dana nakon zadnjeg ponavljanja i bit će srednja vrijednost svih dobivenih rezultata.

8.2. Programi

Implementacijska ideja kreće od koncepta programa. Koje podatke prenijeti programu? Kako bi program trebao signalizirati svoju odluku o smjeru pomicanja elementa?

Radi olakšanja rada, programi su tijekom treniranja sadržani u objektima klase *Individula*. Klasa *Individual* sadržava: originalni zapis programa, program podijeljen u

čvorove, rezultate dobivene igranjem igre, srednji rezultat igara, popis aktivnih čvorova, širinu mreže, dužinu mreže, broj čvorova, broj ulaza, broj izlaza i generacija nastanka. Svi podaci, osim rezultata, dani su pri samoj inicijalizaciji objekta.

Stanje ploče prenosi se programu u svakom koraku igre. S obzirom na to da ploča sadrži 16 polja, program ima 16 ulaza u mrežu. Stanje ploče prenosi se u obliku liste brojeva gdje se slijedno zapisuju polja ploče od gornje lijeve pozicije do donje desne. Brojevima od 0 do 12 zapisuje se sadržaj polja. 0 se koristi kao oznaka da je polje prazno, a ostali brojevi predstavljaju vrijednosti koje se mogu pojaviti u polju i možemo gledati na njih kao na eksponente na broj 2. U primjeru na slici 8.1 prva lista prikazuje početno stanje ploče. U početnom stanju vrijednost 2 pojavljuje se na dva mjesta; treći red i četvrti stupac, četvrti red i treći stupac.

```
Ploca: 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0

Ploca: 0 0 1 1 0 2 0 0 0 0 0 0 0 0 0 0

Ploca: 2 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0
```

Slika 8.1: Stanje ploče tijekom tri poteza

Završetkom obrade podataka u mreži, konačni rezultati dovode se na četiri izlaza. Svaki izlaz signalizira pokret u određenom smjeru (gore, desno, dolje, lijevo). Izlaz s najvećom vrijednošću predstavlja odluku programa.

Stanje ploče nije bilo praćeno u igri te je uvedeno tijekom ovog rada u klasu *Spavner*. Uvođenjem modifikacija u klasu, *Spavner* pohranjuje i obnavlja stanje ploče nakon svakog poteza. Ustanovi li se da je ploča puna i da više nema mjesta za novi element, ploča se čisti, a sprema se lista nula koja signalizira da je igra završila. Završetkom igre programu se šalje rezultat partije. Nakon takvog kraja igre ploča se ponovo postavlja u početno stanje spremno za novu igru.

8.3. Pokretanje treniranja

Sustav za treniranje programa može se pokrenuti na dva načina. Nakon pokretanja igre, pritiskom na tipku *0* pokreće se metoda sa slike 8.2 koja briše zapis prošle generacije s kojom je sustav radio iz *population.txt*, ako postoji; stvara se nova početna generacija i započinje treniranje.

```
public void freshStart(){
    gen=0;
    for (int i=0;i<pop_size;i++){
        List<Integer> individual=new ArrayList<>();
        int upper=numInputs;
        for(int j=0;j<width*length;j++){
            individual.add(getRandom(numFunc));
            if(j%width==0 && j!=0){
                upper+=width;
            }
            for(int k=0;k<numVar;k++){
                individual.add(getRandom(upper));
            }
        }
        for(int j=0;j<numOutputs;j++) {
            individual.add(getRandom( upper: numInputs + width * length));
        }
        population.add(make_ind(individual));
    }
     recordPop(population);
}
```

Slika 8.2: Metoda inicijalizacije nove početne populacije aktivirana pritiskom na tipku *0*

Pritiskom na tipku *4* nastavljamo treniranje populacije zapisane u *population.txt*. Populacija se zapisuje u *population.txt* prilikom inicijalizacije sasvim nove populacije ili nakon generiranja nove generacije. Dodatni parametri koji se šalju u zapis osim programa su širina i duljina mreže te trenutna generacija. Slika 8.3 prikazuje metodu koja zapisuje populaciju.

```
private void recordPop(List<Individual> population){
    try{
        File myFile=new File( pathname: "population.txt");
        if (myFile.createNewFile()) {
            System.out.println("Populations will be recorded in " + myFile.getName()+".");
        } else {
            System.out.println("Population will be rewritten.");
        }
        FileWriter myWriter=new FileWriter( fileName: "population.txt");
        myWriter.write( str: length+ " "+width+"\n");
        myWriter.write( str: gen+"\n");
        for(Individual individual:population){
            for (Integer number:individual.program){
                myWriter.write( str: number+" ");
            }
            myWriter.write( str: "\n");
        }
        myWriter.close();
        System.out.println("Population saved.");
    }catch (IOException e){
        System.out.println("An error occurred.");
    }
}
```

Slika 8.3: Metoda spremanja populacije u *population.txt*

Početni parametri programa i sustava (broj redova, broj stupaca, veličina populacije, broj ponavljanja) zadaju se u *KeyInput* klasi kod inicijalizacije objekta klase CGP. Dodatni parametri poput generacijske granice, mogućih funkcija i broja mutacija mogu se podesiti unutar klase *CGP*, s napomenom kako promjena mogućih funkcija zahtijeva više promjena implementacije i veći oprez.

Zadnji korak u pokretanju treniranja je stvaranje dretve koja će provoditi treniranje. Ovime glavnu dretvu vraćamo održavanju igre i grafičkog sučelja te postižemo paralelan rad.

8.4. Igranje igre

Dretva zadužena za treniranje kontinuirano dohvaća stanje ploče. Ako na ploči postoje elementi, stanje ploče se prosljeđuje metodi *predict* (Slika 8.4) koja izvodi trenutni program na način koji je opisan u potpoglavlju 4.6. Metoda vraća broj izlaza s najvećom vrijednosti. Odluka o pomicanju elemenata donosi se na osnovu dobivenog broja.

Ako na ploči nema elemenata to znači kako je igra završila. Spremamo rezultate i pozivamo metodu *keep_count* (slika 8.5) koja prati koji program se izvodi, koliko puta se izveo i nalazimo li se na prijelazu u novu generaciju.

```
public synchronized void CGP_play() throws InterruptedException {
    int[] ploca;
    String vidploce;
    identify_all();
    while(true || gen<gen_limit){
        ploca=spavn.getPloca();
        vidploce="";
        for (int i:ploca){
            vidploce+=i;
        }
        if(vidploce.equals("0000000000000000")){
            population.get(which).scores.add(spavn.final_score);
            population.get(which).calc_mean();
            scores.add(spavn.final_score);
            keep_count();
            Thread.sleep(1000);
        }
        else {
            int decision = predict(ploca);
            switch(decision){
                case 0: spavn.gore();
                    break;
                case 1: spavn.desno();
                    break;
                case 2: spavn.dolje();
                    break;
                case 3: spavn.lijevo();
                    break;
            }
        }
    }}
```

Slika 8.4: Metoda kontinuiranog provođenja igranja igre

```
private void keep_count(){
    if(counter<trys-1 && which<pop_size){
        counter++;
    } else if (counter>=trys-1 && which<pop_size-1) {
        which++;
        counter=0;
    }else{
        System.out.println("Generation done");
        record_scores();
        scores.clear();

        gen++;

        crossover();

        for(Individual ind:population){
            ind.clear_score();
        }
        System.out.println("\nNew population:\n");
        seePopulation();

        recordPop(population);

        counter=0;
        which=0;
    }
}
```

Slika 8.5: Metoda praćenja programa i generacija

8.5. Selekcija i križanje

Metoda *keep_count* prepoznaje kraj generacije, zapisuje vrijeme zapisa, parametre sustava, rezultate svakog programa i pokreće operaciju selekcije i križanja. Selekcija je implementirano kao turnir s nasumičnim izborom programa. Broj programa koje odabiremo je varijabilan, ali je uvijek višekratnik broja četiri. Razlog tome je taj da odabiremo dva para po dva programa. Programe u paru međusobno uspoređujemo. Programi koji su unutar para bili bolji križaju se, a gori će biti zamijenjeni djecom boljih. Bolji programi i programi koji nisu sudjelovali u turnirima prelaze u novu generaciju. Ovime se donekle ostvaruje elitizam jer najbolji programi na jedan ili na drugi način prelaze u sljedeću generaciju.

Implementirane su dvije metode križanja. Prva metoda se može smatrati destruktivnom i najmanje je korištena u testovima. Implementirana je za mreže sa širinom većom od jedan. Metoda stvara dvoje djece tako da svaki čvor djeteta ima 50% šanse doći od jednog ili od drugog roditelja. Destruktivnost proizlazi iz činjenice da se na

ovaj način genima jednog roditelja remete dijelovi genotipa drugog roditelja koji su ga činili uspješnim. (Slike 8.6 (a) i (b))

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

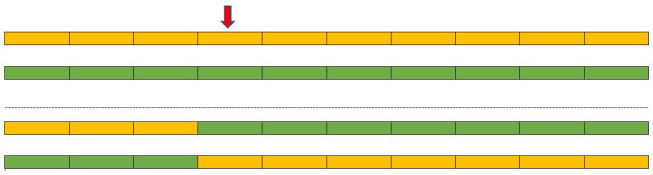
(a) Prvo dijete

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

(b) Drugo dijete

Slika 8.6: Primjer rezultata prve metode križanja

Druga metoda implementirana je za mreže širine 1 i zasnovana je na križanju s jednom točkom prijeloma. Kopije dva roditelja se prepolove u odabranom čvoru, a zatim se svaki dio spoji s dijelom drugog programa.



Slika 8.7: Križanje s jednom točkom prijeloma

8.6. Mutacija

Implementirane su dvije vrste mutacija, mutacija postotka i single mutacija, ali u testovima je korištena samo single mutacija kako bi se smanjila mogućnost potencijalno prevelike promjene fenotipa. Implementacija (Slika 8.8) dodatno omogućuje odabir broja uzastopnih single mutacija djeteta u svrhu proširenja područja testiranja. Single mutacija olakšana je činjenicom da objekti klase *Individual* već sadrže popis aktivnih gena pa stvaramo listu gena za mutaciju dok ne odaberemo gen s popisa aktivnih gena. Jednom kad se aktivni gen odabere, prestajemo tražiti gene za mutaciju, a odabrane gene izmijenimo.

Mutacijom završavamo naš proces stvaranja djece. Novonastala djeca pridružuju se već postojećoj populaciji dolaskom na mjesto programa koji su se iskazali slabijima u procesu selekcije. Time započinjemo novu generaciju, a podaci o novoj populaciji spremaju se u *population.txt*

```
private void mutateSingle(Individual individual){
    List<Integer> range=new ArrayList<>();
    for (int i=0; i<individual.program.size();i++){
        range.add(i);
    }
    List<Integer> picks=new ArrayList<>();
    int position;
    int gene;
    boolean active=false;
    do{
        position=getRandom(range.size());
        gene=range.remove(position);
        picks.add(gene);
    }while (!is_gene_active(individual,gene));
    for(int pick:picks){
        if(pick<=(width*length*(numVar+1))){
            if(pick % (numVar+1)==0){
                individual.program.set(pick,getRandom(numFunc));
            }
            else{
                int upper=numInputs+(width)*(pick/(width*(numVar+1)));
                individual.program.set(pick,getRandom(upper));
            }
        }
        else {
            individual.program.set(pick,getRandom(numInputs+width*length));
        }
    }
    individual.active=identify(individual.program);
    individual.nodes=node_split(individual.program);
}
```

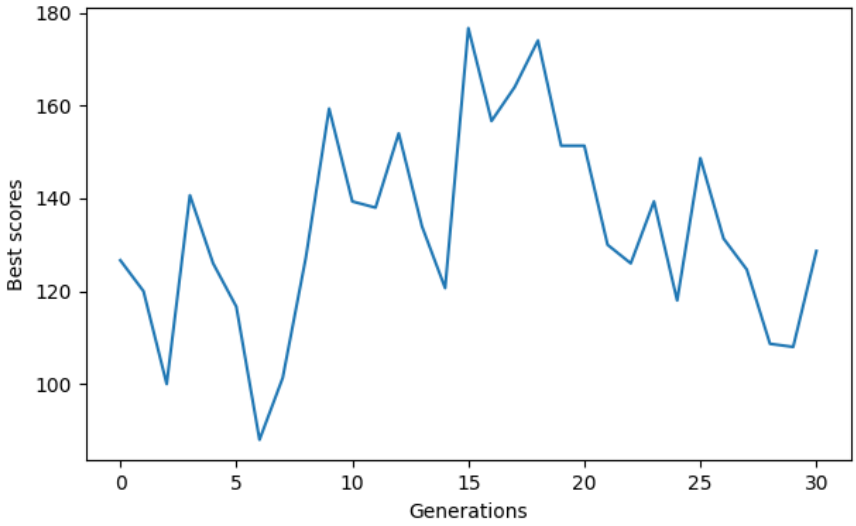
Slika 8.8: Metoda Single mutacije

9. Rezultati

Testovi su izvođeni na već opisanoj verziji igre *2048*. Verzija se ispostavila problematičnom jer je brzina izvođenja igre bitno ograničena implementacijom verzije. U svim testovima svi su programi imali tri pokušaja rješavanja igre. U slučaju veličine populacije četiri, prosječno vrijeme potrebno za prolazak svih programa generacije kroz igru bilo je petnaest minuta. U slučaju veličine populacije osam, prosječno vrijeme prolaska generacije kroz igru bilo je trideset minuta. U slučaju veličine populacije šesnaest, prosječno vrijeme prolaska generacije kroz igru bilo je šezdeset minuta. Povećanje potrebnog vremena je linearne zavisnosti i učinilo je treniranje predugim za ikakve prave rezultate. Rezultati koji su prikupljeni za različite arhitekture i parametre sustava ograničeni su na tridesetu generaciju kako bi se mogli međusobno usporediti. Ograničenje je takvo zbog toga što je to minimalna generacija do koje su došli svi testovi prije zaustavljanja daljnjeg treniranja.

Provedeno je pet testova s četiri različite arhitekture i s različitim parametrima sustava. Radi jednostavnosti zapisa ispod slika uvodimo kratice: c - broj stupaca, r - broj redova, p - veličina populacije, f - broj funkcija, m - broj mutacija po djetetu.

Prvi test izveden je nad arhitekturom mreže sa širinom šesnaest i dubinom četiri uz veličinu populacije četiri, broj funkcija četiri (+,-,*,/) i jednom single mutacijom po djetetu. Selekcija se provodi nad četiri programa. Korišteno je križanje koje je prvo opisano u potpoglavlju 8.5. Test je osmišljen kao reprezentacija općenite i plitke mreže sa širinom većom od nula, a u ovom slučaju širina je jednaka broju ulaza. Trajanje testiranja bilo je 24 sata, unutar kojih je sustav prošao kroz sedamdeset i dvije generacije. Većina rezultata nalazila se u rasponu 70-150. Na slici 9.1 možemo vidjeti najbolje srednje vrijednosti rezultata od nulte do tridesete generacije. Iako oscilacije postoje, vidimo polagani rast najboljih rezultata do petnaeste generacije gdje započinje stagnacija i lagan pad. Na osnovu rezultata daljnjih generacija, srednje vrijednosti rezultata stagniraju ili se nalaze unutar istog raspona rezultata.

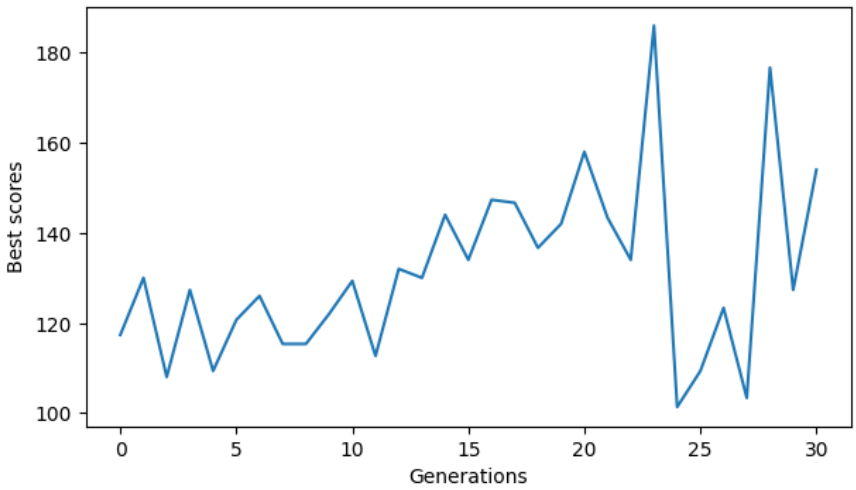


Slika 9.1: Najbolje srednje vrijednosti rezultata programa za c=4, r=16, p=4, f=4, m=1

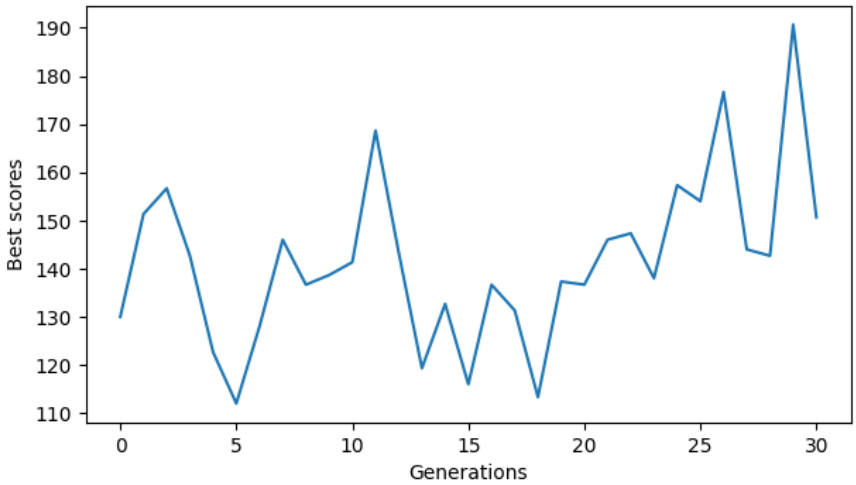
Drugi test izveden je nad arhitekturom mreže sa širinom jedan i dubinom sto uz veličinu populacije četiri, broj funkcija četiri (+,-,*,/) i jednom single mutacijom po djetetu. Selekcija je provedena nad četiri programa. Korišteno je križanje s jednom točkom prijeloma. Test je osmišljen kao reprezentacija jednostavnog slučaja mreže širine jedan gdje je naglasak na vezama između čvorova u dubini. Ideja je bila koristiti ovaj slučaj kao bazu za uspoređivanje s drugim arhitekturama i parametrima sustava. Trajanje testiranja bilo je 22 sata, unutar koji je sustav prošao kroz pedeset i tri generacije. Većina rezultata nalazila se u rasponu 90-150. Na slici 9.2 možemo vidjeti najbolje srednje vrijednosti rezultata od nulte do tridesete generacije. Primjećuje se porast najbolje srednje vrijednosti do dvadeset i treće generacije nakon čega slijedi nagli pad uspješnosti i povratak u stanje prije pada. Rezultati daljnjih generacija ukazuju na umjereno poboljšanje, ali gornja granica raspona većine rezultata ne pomiče se previše.

Treći test je gotovo po svim parametrima jednak prošlom. Jedina razlika je veličina populacije. U ovom testu veličina populacije je 8. Cilj ovog testa bio je vidjeti razliku u uspješnosti kroz generacije ako povećamo veličinu populacije. Trajanje testiranja bilo je 20 sati, unutar kojih je sustav prošao kroz trideset i dvije generacije. Većina rezultata nalazi se u rasponu od 80-150. Počinjemo uviđati iste raspone rezultata kroz testove i sumnja se kako je ovo početni raspon rezultata većine inicijaliziranih populacija. Slika 9.3 prikazuje najveće srednje vrijednosti rezultata od nulte do tridesete generacije. Stabilan porast rezultata u ovom testu kasni za onim u prošlom testu.

Minimalna i maksimalna vrijednost rezultata je veća od prošlog testa, iako je razlika možda zanemariva.



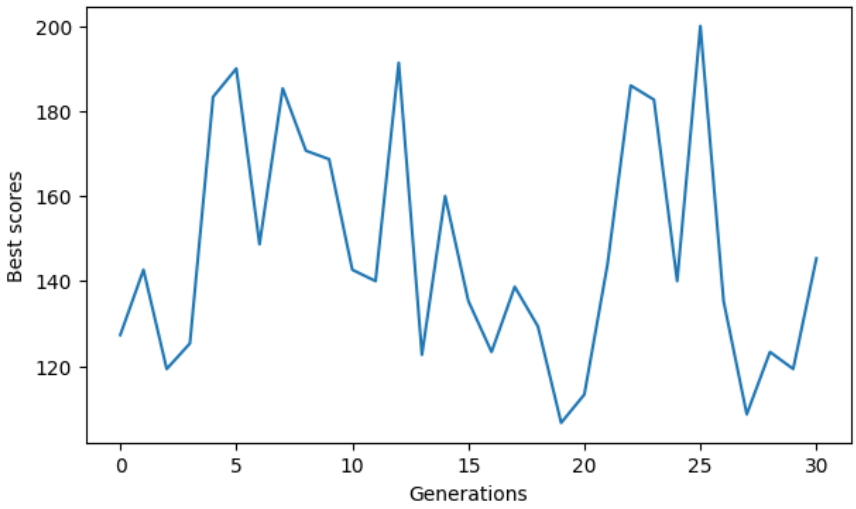
Slika 9.2: Najbolje srednje vrijednosti rezultata programa za c=100, r=1, p=4, f=4, m=1



Slika 9.3: Najbolje srednje vrijednosti rezultata programa za c=100, r=1, p=8, f=4, m=1

U četvrtom testu dubina je povećana na dvjesto i uvedene su još dvije funkcije (min i max). Testom se pokušao vidjeti utjecaj produbljavanja mreže i dodavanja dodatnih mogućih funkcija. Trajanje testiranja bilo je 20 sati, unutar kojih je sustav prošao kroz trideset i pet generacija. U početnim generacijama većina rezultata nalazi se u rasponu 100-150, ali u kasnijim generacijama raspon pada na 90-120. Slika 9.4 prikazuje najveće srednje vrijednosti rezultata od nulte do tridesete generacije. Iznos najveće

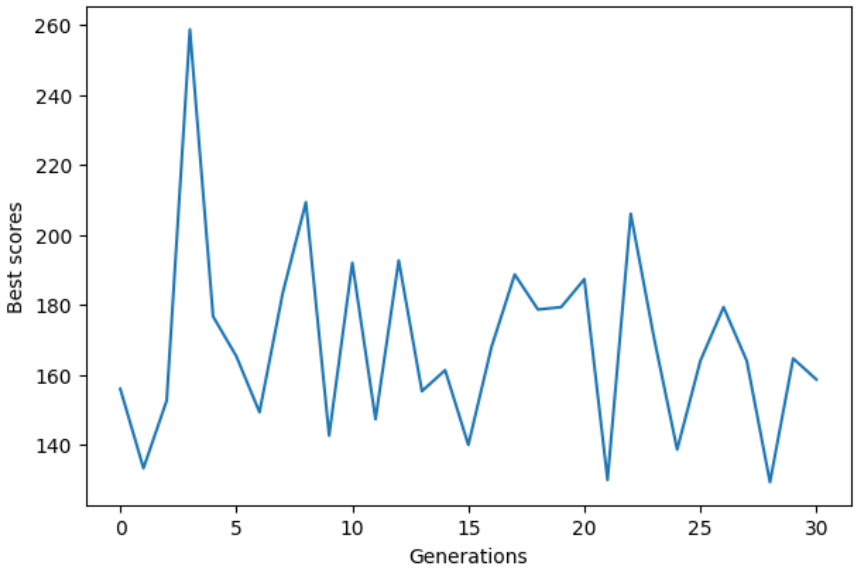
srednje vrijednosti rezultata u generaciji je nestabilan te na odsječku od trideset generacija ne možemo zaključiti previše. Prosjek rezultata po generacijama pokazuje stabilizaciju rezultata, ali i njihov pad.



Slika 9.4: Najbolje srednje vrijednosti rezultata programa za c=200, r=1, p=8, f=6, m=1

Peti test nadograđuje četvrti tako da smo mrežu produbili s dodatnih sto čvorova, veličinu populacije postavili na šesnaest, selekcija je provedena nad osam programa i broj single mutacija po djetetu povećali na dva. Cilj testa bio je vidjeti kako će povećanje kompleksnosti i promjenjivosti populacije utjecati na rezultate. Trajanje testiranja bilo je 43 sata, unutar kojih je sustav prošao kroz četrdeset i jednu generaciju. Većina rezultata nalazi se u rasponu od 90-150. Slika 9.5 prikazuje najveće srednje vrijednosti rezultata od nulte do tridesete generacije. Najveća srednja vrijednost rezultata generacije u ovom testu postiže najveći iznos u usporedbi s ostalim testovima. Na žalost ovaj test jedini pokazuje jasan pad najveće srednje vrijednosti rezultata i izrazito je nestabilan s naglim usponima i padovima.

Pregledom rezultata testova možemo pretpostaviti nekoliko učinaka koje parametri mogu imati na uspješnost programa. Veća kompleksnost može dovesti do većeg uspjeha programa, ali dovodi i do veće nestabilnosti. Prevelika razina mutacije utječe destruktivno na generacije programa. Povećavanjem populacije, bez povećanja selekcije, može dovesti do odgađanja promjene u populaciji. Sve ove pretpostavke nisu dokazane već su postavljene na osnovu dobivenih rezultata i trebaju se tretirati kao moguće hipoteze za budući rad.



Slika 9.5: Najbolje srednje vrijednosti rezultata programa za c=300, r=1, p=16, f=6, m=2

10. Zaključak

Uloga genetskih algoritama u strojnom učenju neosporiva je i značajna u današnjem svijetu. Nevjerojatna je učinkovitost metode pogreške i pokušaja koja kroz generacije programa stvara najbolja rješenja za različite probleme i okoline u kratkom roku. Fascinantnost ovakvog oblika učenja i razvoja programa uvjetovala je stvaranje ovog rada.

Tijekom ovog rada, pomoću Kartezijevog genetskog programiranja, razvijeni su agenti čiji je cilj igranje igre *2048*. Objašnjena su pravila i funkcioniranja verzija igara, objašnjeni su ključni elementi i operacije genetskog programiranja i Kartezijevog genetskog programiranja, izložena je i objašnjena implementacija rada, izvedeni su i prikazani rezultati testova implementacije.

Na žalost, verzija igre učinila se previše problematičnom za skupljanje veće količine podataka i provođenje više testova. Određeni napreci, nazadovanja i utjecaji parametara mogu se vidjeti, ali za pravi učinak morali bi moći provesti razvoj kroz podosta više generacija nego što je provedeno u ovom radu.

Implementacija otvara mogućnosti za daljnji rad i istraživanje. Potencijalne modifikacije bile bi olakšavanje unosa i korištenja određenih parametara sustava uz moguće grafičko sučelje koje bi zamijenilo potrebu za unošenjem parametara direktno u programski kod implementacije. Kombinacije različitih parametara gotovo su neograničene i ostavljaju puno prostora za daljnja testiranja. Najveći napredak može se postići dodatnim radom na korištenoj verziji igre, upotrebom druge verzije ili razvojem vlastite kojom bi se ubrzalo vrijeme treniranja i razvoja programa.

LITERATURA

[1] Gabriele Cirulli. play2048, 2018. URL https://play2048.co/.

[2] Gabriele Cirulli. 2048, 2018. URL https://github.com/gabrielecirulli/2048.

[3] Roman Kalkreuth, Günter Rudolph, i Andre Droschinsky. A new subgraph crossover for cartesian genetic programming. U James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, i Pablo García-Sánchez, urednici, *Genetic Programming*, stranice 294–310, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55696-3.

[4] J.F. Miller i S.L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2): 167–174, 2006. doi: 10.1109/TEVC.2006.871253.

[5] Julian F. Miller. *Cartesian Genetic Programming*. Springer, 2011. URL https://doi.org/10.1007/978-3-642-17310-3_2.

[6] Julian F. Miller. Ppsn 2014 tutorial: Cartesian genetic programming, 2014. URL https://cs.ijs.si/ppsn2014/files/slides/ppsn2014-tutorial3-miller.pdf.

[7] Riccardo Poli, William B. Langdon, i Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

Igranje igre *2048* korištenjem Kartezijevog genetskog programiranja

Sažetak

Cilj ovog rada je stvoriti agenta ili populaciju agenata koji će naučiti igrati igru 2048. Učenje se provodi korištenjem Kartezijevog genetskog programiranja. U sklopu ovog rada obrađuju se koncepti genetskog programiranja, Kartezijevog genetskog programiranja, razvoja agenata kroz generacije, izlaže se implementacija rada u programskom jeziku Java, prikazuju se rezultati testiranja implementacije, diskutiraju se dobiveni rezultati i predlažu se potencijalna poboljšanja.

Ključne riječi: Igra 2048, genetski algoritam, genetsko programiranje, Kartezijevo genetsko programiranje

Playing the 2048 game using Cartesian genetic programming

Abstract

The goal of this thesis is to create an agent or a population of agents that will learn to play the game 2048. The learning is performed using Cartesian genetic programming. As part of this work, the concepts of genetic programming, Cartesian genetic programming and the development of agents through generations are discussed, the implementation of the work in Java programming language and the results of implementation testing are presented, the obtained results are discussed and potential improvements are proposed.

Keywords: Game 2048, genetic algorithm, genetic programming, Cartesian genetic programming