SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4424

Optimizacija parametara strateških igara evolucijskim algoritmima

Marko Stanić

Zagreb, lipanj 2016.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 15. ožujka 2016.

ZAVRŠNI ZADATAK br. 4424

Pristupnik: **Marko Stanić (0036473130)**
Studij: Računarstvo
Modul: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Optimizacija parametara strateških igara evolucijskim algoritmima**

Opis zadatka:

Opisati problem optimizacije parametara strateških igara u cilju omogućavanja igrivosti. Definirati vremenski ovisan model strateške igre, postojećih ograničenja i mogućnosti izbora igrača. Istražiti načine prikaza poteza igrača pogodne za uporabu u optimizacijskom algoritmu. Ostvariti programski sustav koji uključuje model simulacije strateške igre i postupke optimizacije poteza igrača uz pomoć evolucijskih algoritama. Ispitati mogućnosti ostvarenog sustava u pronalaženju niza poteza koji vodi do pobjede igrača, s obzirom na zadana ograničenja. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 17. lipnja 2016.

Mentor:

Izv. prof. dr. sc. Domagoj Jakobović

Djelovođa:

Doc. dr. sc. Ivica Botički

Predsjednik odbora za završni rad modula:

Prof. dr. sc. Krešimir Fertalj

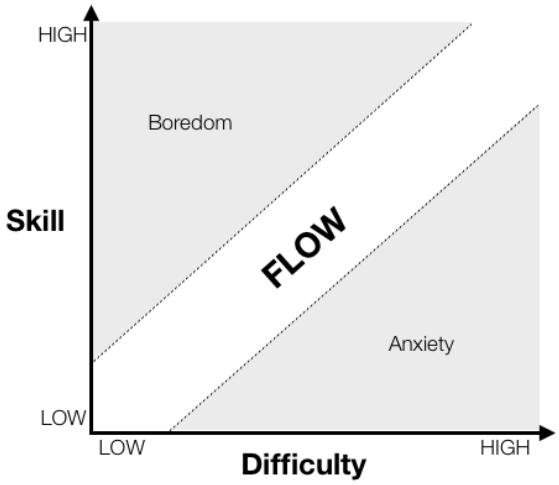
Zahvaljujem se mentoru Domagoju Jakoboviću na slobodi pri izboru teme.

SADRŽAJ

1. Uvod

Razvijanje računalnih igara nije kao razvijanje ostalog softvera - ne samo zato što igrači često ne znaju što žele, nego i zato što je jako teško napraviti igru s takvim parametrima da pruža dobar izazov igraču, a da ga ne frustrira.

Stoga su u industriji računalnih igara često potrebni *testeri* - ljudi koji igraju igre tražeći propuste i probleme u ravnoteži. Problemi u ravnoteži igre se mogu manifestirati na više načina - igra je preteška, igra je prelaka, ili ako se radi o igri s više igrača, neki igrači imaju nepoštene prednosti pred drugim igračima, što narušava zabavu igre i može udaljiti igrača (slika 1.1).



Slika 1.1: Povezanost toka, dosade i nervoze s težinom igre i vještinom igrača. [2]

Međutim, testerima često treba puno vremena da dođu do zaključka jer su ljudi. Budući da je posao koji obavljaju rutinski, postavlja se pitanje je li moguće taj proces automatizirati tako da ga radi računalo umjesto osobe. Ovaj rad se bavi traženjem odgovora na to pitanje i ostvarenjem jednostavne igre u kojoj će genetski algoritam ispitati je li moguće riješiti određeni dio igre.

2. Strateške igre na računalu

Strateške igre su žanr računalnih igara kod kojih se od igrača očekuje vješto planiranje i sposobnost taktičkog razmišljanja. Igrač je obično u ulozi neke vrste zapovijednika koji daje naredbe jedinicama (engl. *units*) i upravlja ekonomijom, diplomacijom i sličnim aspektima, ovisno o igri. Ovakve igre se dijele na dvije skupine: strateške igre na poteze (engl. *turn-based strategy game*) i strateške igre u realnom vremenu (engl. *real-time strategy game*).

Kod strateških igara na poteze, svaki igrač može upravljati svojim jedinicama samo za vrijeme svojeg poteza, a kad završi potez, drugi igrač dolazi na red.

Kod strateških igara u realnom vremenu ne postoje potezi, već svaki igrač ima slobodu upravljati jedinicama kontinuirano, bez potrebe za čekanjem da drugi igrač završi. Budući da se radnja igre odvija kontinuirano, vrijeme je bitan faktor i predstavlja pritisak, što potiče igrača da brzo donosi odluke. [1]

2.1. Povijest strateških igara na računalu

Većina prvih strateških igara na računalu nastala je iz pokušaja da se reproduciraju društvene igre, poput poznate igre *Šah*. Prednosti igranja strateških igara na računalu je mogućnost da računalo upravlja složenim skupom pravila igre bez potrebe da igrač poznaje sva pravila. [1]

Prva strateška igra na računalu, *Invasion*, nastala je 1972. godine i imala je skup pravila po uzoru na popularnu društvenu igru *Risk*. Uslijedilo je nekoliko pokušaja da žanr postane popularan na računalu, ali prije 90-ih godina 20. stoljeća, većina nije bila značajna.

Smatra se da je najznačajnija strateška igra na poteze *Civilization* (slika 2.1), koja je izdana 1991. godine, i postavila temelje za brojne druge strateške igre u budućnosti. [4]

Slika 2.1: *Civilization*

Dune II (slika 2.2), nastala 1992. godine, se smatra prvom pravom strateškom igrom u realnom vremenu. [1] U toj igri su uspostavljeni standardi koji se još i danas koriste u strateškim igrama u realnom vremenu. Izgradnjom zgrada omogućuje se proizvodnja jedinica koje služe za borbu, istraživanje ili sakupljanje resursa. Za izgradnju zgrada i proizvodnju jedinica su potrebni resursi, nad kojima se vojske bore za kontrolu - vojska s više resursa pod svojom kontrolom je u prednosti. Bolja preglednost omogućena je minijaturiziranim prikazom bojišta (engl. *minimap*).

Slika 2.2: *Dune II*

Dune II je bila napredna igra za svoje vrijeme, međutim, nije imala mod igranja za

više igrača (engl. *multiplayer*). Teško je reći koja je zaista bila prva strateška igra koja je omogućavala igru više igrača preko mreže, ali jedna od najznačajnijih je *Warcraft: Orcs & Humans*, koja je podržavala igru preko mreže putem IPX protokola.[9] To je bilo vrlo važno jer je mogućnost mrežnog igranja davala strateškim igrama društvenu komponentu, zbog čega su polako postajale sve popularnije. Uz to, igranje protiv igrača je bio veći izazov od igranja protiv računala, budući da je umjetna inteligencija u to doba bila relativno teška za izvesti zbog ograničenja računala.

Razvojem hardvera, internetskih protokola i infrastrukture, i strateške igre su počele sve više dobivati na kvaliteti i popularnosti. Primjerice, u igri *Supreme Commander*, izdanoj 2007. godine, u mrežnim igrama se mogu nadmetati do 8 igrača, s tisućama jedinica u realnom vremenu, što je u 90-im godinama prošlog stoljeća bilo nezamislivo.

2.2. Problem ravnoteže strateških igara na računalu

U strateškim igrama se obično pojavljuje problem ravnoteže, koji razlikujemo kod načina igre za jednog igrača (engl. *singleplayer*) i kod načina igre za više igrača (engl. *multiplayer*).

Kod načina igre za jednog igrača, dobra ravnoteža obično znači da je težina igre adekvatna za publiku kojoj je igra namijenjena. Drugim riječima, igra ima dobru ravnotežu ako nije ni preteško ni prelagano da igrač u njoj napreduje. Ako je igra preteška, obično će frustrirati i odbiti igrača, a ako je prelagana, igrač će obično odustati zbog nedostatka izazova.

U igri *Dune II*, ravnoteža je riješena tako da igrač na nijednoj razini nema lošije razvijene jedinice od protivnika. Zbog slabije umjetne inteligencije, protivniku kojim upravlja računalo je omogućeno varanje (izgradnja zgrada koje nisu spojene s ostalim zgradama, primjerice).

Kod načina igre za više igrača, ravnoteža znači da igrači sličnih vještina imaju barem približno jednaku šansu za pobjedu iako počinju igru s različitim parametrima (ovisno o igri - rasa, lokacija, broj početnih resursa i slično). [8]

Slika 2.3: *Warcraft: Orcs & Humans*

U igri *Warcraft: Orcs & Humans* (slika 2.3), ravnoteža u načinu igre za više igrača je riješena jednostavno - obje igrive rase imaju jedinice s identičnim atributima (brzina kretanja, jačina napada, trošak, i slično), a razlikuju se samo u jednoj jedinici koja je jedinstvena za svaku rasu - napravljena je tako da kod jedne rase ima bolji napad, a kod druge obranu. Mape u igri su napravljene simetrično, tako da nijedan igrač nije u povoljnijem položaju od drugog s obzirom na resurse i lokaciju.

3. Evolucijski algoritmi

3.1. Općenito o evolucijskim algoritmima

Evolucijski algoritmi (engl. *evolutionary algorithms, evolutionary computing*) su postupci optimiranja, učenja i modeliranja koji se temelje na mehanizmu prirodne evolucije. To su formalni sustavi koji nastoje biti izomorfni s prirodnom evolucijom. Evolucijski algoritmi nastali su iz dviju pobuda: želje za boljim razumijevanjem prirodne evolucije i pokušaja primjene načela prirodne evolucije pri rješavanju različitih zadaća. [7]

Za bolje razumijevanje evolucijskih algoritama potrebno je poznavati znanstvenu teoriju evolucije. Sva živa bića imaju skupove gena koji čine DNK - nukleinsku kiselinu koja sadrži upute za biološki razvoj. Živo biće gene naslijeđuje od roditelja, i prenosi ih na svoje potomke kod razmnožavanja. Međutim, potomci gotovo nikad nisu potpuno identični roditeljima, već postoje varijacije koje se mogu dogoditi slučajno ili rekombinacijom gena prilikom razmnožavanja. Ako su te varijacije korisne, potomak ima veću šansu za preživljavanje i razmnožavanje, što znači da će takav potomak nakon nekoliko generacija obično biti znatno brojniji od potomaka koji imaju manje korisne kombinacije gena. Promjenom okoline i životnih uvjeta preživljavaju pojedinci koji imaju gene prilagođene novim uvjetima, te se oni dalje razmnožavaju, a njihove genetičke varijacije se preko nekoliko generacija akumuliraju. Kada populacija potomaka ima bitno različite gene u odnosu na neke pretke, mogu se smatrati novom vrstom. [6]

Primjerice, promatra se populacija životinja nalik jelenu. Svi mužjaci posjeduju rogove koji služe za privlačenje ženki i borbu s drugim mužjacima za pravo na parenje. Parenjem jednog mužjaka i ženke nastane potomak koji ima genetsku mutaciju zbog koje ima rogove jače i veće od prosjeka, koji mu omogućuju da lakše privuče ženke i pobjeđuje u borbi, što povlači veću vjerojatnost razmnožavanja. U sljedećoj generaciji, njegovi potomci naslijede tu korisnu mutaciju, te i oni imaju veće šanse za razmnožavanje. Promatranjem populacije nakon nekoliko generacija može se ustanoviti da će mužjaci s manjim rogovima biti potisnuti, i da će prevladati mužjaci s većim

rogovima.

Kada se taj koncept preslika na računalne programe, postaje dobar alat za rješavanje problema koje je teško riješiti uobičajenim metodama.

Međutim, treba imati na umu da ovakvi algoritmi ne daju nužno najbolje moguće rješenje, tj. nalaze lokalni maksimum (ili minimum) funkcije dobrote, a ne globalni. Primjer toga se može naći i u prirodi, u žirafama. Žirafa ima povratni grkljanski živac (slika 3.1), koji ima duljinu veću od 4 metra jer ide iz mozga cijelom duljinom vrata, oko lijeve aorte i nazad cijelom duljinom vrata do grkljana, gdje ima funkciju upravljanja glasnicama, što nije optimalno jer su mozak i grkljan žirafe udaljeni desetak centimetara. [3] Budući da je predak žirafe imao znatno kraći vrat, duljina tog živca je bila zanemariva, ali kako se evolucijom vrat žirafe produljivao, živac je postajao sve dulji umjesto da je jednostavno promjenio putanju. Budući da evolucijski algoritmi oponašaju prirodnu evoluciju, taj problem se javlja i kod njih.

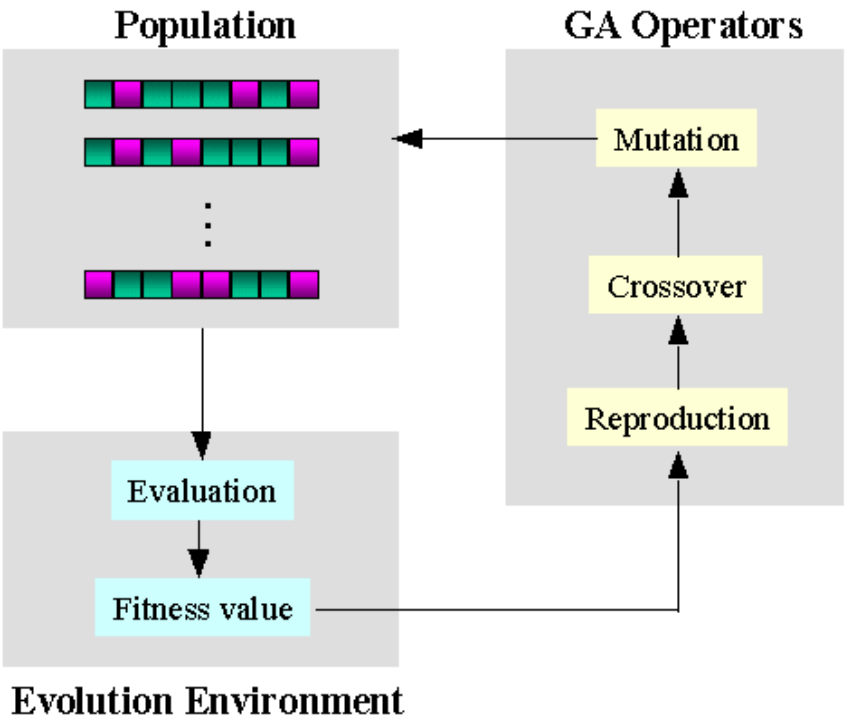
Slika 3.1: Ilustracija povratnog grkljanskog živca kod žirafe

3.2. Genetski algoritmi

Genetski algoritmi čine podskup evolucijskih algoritama. Najčešće se koriste za pronalazak rješenja optimizacijskog problema ili rješenja problema pretrage.

Genetski algoritam se provodi nad populacijom jedinki (engl. *individuals*) koje predstavljaju potencijalna rješenja za problem koji se rješava. Svaka jednika ima skup svojstava koja se mogu mijenjati i mogu biti izvedena na više načina (niz nula i jedinica, niz realnih brojeva, itd.)

Evolucija je iterativan proces, a obično počinje s populacijom jedinki sa slučajno generiranim svojstvima. Populacija u svakoj iteraciji se naziva generacijom, a u svakoj generaciji se određuje dobrota (engl. *fitness*) svake jedinke, koja se računa funkcijom određenom s obzirom na to kakav problem se rješava. U svakoj generaciji, uzimaju se najbolje jedinke te se od njih sastavlja sljedeća generacija na način da se njihovi geni rekombiniraju ili mutiraju, te se postupak ponavlja dok se ne pronađe dovoljno dobro rješenje, ili dok broj generacija postane jako visok, ovisno o problemu koji se rješava. Dakle, za rješavanje problema genetskim algoritmom potrebno je domenu rješenja predstaviti u obliku gena, te je potrebno odrediti funkciju dobrote. Tok genetskog algoritma je prikazan na slici 3.2.



Slika 3.2: Tok genetskog algoritma

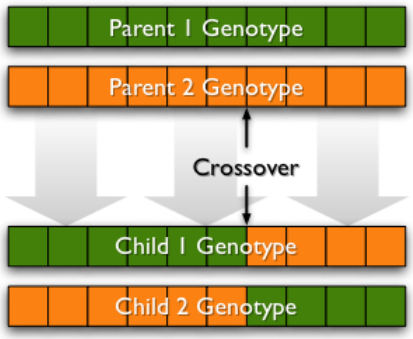
Struktura genetskog algoritma je prikazana na slici 3.3.

```
Genetski_algoritam
{
   t = 0
   generiraj početnu populaciju potencijalnih rješenja P(0);
   sve dok nije zadovoljen uvjet završetka evolucijskog procesa
   {
        t = t + 1;
        selektiraj P'(t) iz P(t-1);
        križaj jedinke iz P'(t) i djecu spremi u P(t);
        mutiraj jedinke iz P(t);
   }
   ispiši rješenje;
}
```

Slika 3.3: Struktura genetskog algoritma [5]

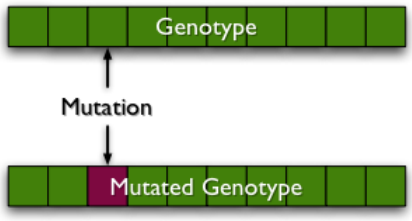
Selekcija (engl. *selection*) omogućuje da se bolja rješenja reproduciraju i prenesu svoja svojstva na sljedeću generaciju u algoritmu. Postoji više vrsta selekcije, ali najčešće korištene su jednostavna selekcija (engl. *roulette wheel selection*) i turnirska selekcija (engl. *tournament selection*).

Križanje (engl. *crossover*) je proces kombiniranja svojstva roditelja da se stvori potomak u novoj generaciji. Najčešće se koristi križanje s jednom točkom prekida (engl. *single point crossover*) kao što je prikazano na slici 3.4.



Slika 3.4: Križanje s jednom točkom prekida

Mutacija (engl. *mutation*) služi kao sredstvo sprječavanja konvergencije rješenja lokalnom minimumu. Primjerice, ako je genom predstavljen kao niz bitova, dodjeljuje mu se vjerojatnost mutacije koja označava da svaki bit može biti invertiran tom vjerojatnošću (slika 3.5).



Slika 3.5: Jednostavna mutacija

Za provođenje genetskog algoritma ključno je da se koriste sva tri operatora.

4. Programsko ostvarenje

U ovom poglavlju je opisano kako je ostvarena optimizacija težine igre genetskim algoritmom na primjeru jednostavne strateške igre s obrambenim tornjevima (engl. *tower defense game*).

4.1. Ideja rješenja

Cilj rada je bio ostvariti model jednostavne igre s obrambenim tornjevima (slika 4.1) koji će omogućiti optimiziranje težine pojedinačnih razina genetskim algoritmom po izboru.

Slika 4.1: Primjer iz igre *Lost Earth* koja je pripadnik istog žanra

Igra se odvija na dvodimenzionalnoj plohi podijeljenoj na polja, na kojoj je cesta

koja ide od točke A do točke B. Neprijatelji se stvaraju u točki A i kreću se po cesti prema točki B. Ukoliko neprijatelj dođe do točke B, igraču se umanjuje brojač koji predstavlja zdravlje (engl. *hitpoints*), i ako zdravlje postane 0, igrač gubi. Cilj igrača je spriječiti neprijatelje da dođu do cilja tako da gradi obrambene tornjeve, kojih ima različitih vrsta. Ne postoji univerzalan toranj koji je dobar protiv svake neprijateljske jedinice, već svaki toranj ima svoje prednosti i mane. Svaki toranj neprijateljima radi štetu ovisno o tome kakva je vrsta neprijateljevog oklopa, i kakva je vrsta tornjevog napada. Kada je neprijateljska jedinica uništena, igrač dobiva određenu količinu valute koju koristi za izgradnju tornjeva. Igra završava kada igrač izgubi, ili kad igrač uništi sve neprijateljske jedinice.

Razine se dizajniraju na način da se odredi oblik ceste, dimenzije mape i sastavi lista neprijatelja. Neprijatelja može biti proizvoljno mnogo, a u igri dolaze po redu u konstantnim vremenskim razmacima.

U industriji igara, dizajner obično sastavi razinu i testiranje tada rade ljudski igrači. Ideja ovog ostvarenja je da dizajner sastavi razinu i nad njom izvrši genetski algoritam koji "igra" razinu i provjerava postoji li rješenje, tj. je li razina uopće rješiva. Ako se ispostavi da je razina rješiva, dizajner može pregledati koji redoslijed naredbi je bio pobjednički, i na temelju toga zaključiti je li dovoljno jednostavan da bi ga mogao odigrati čovjek. Ovo je korisno jer se djelomično uklanja potreba za ljudima koji testiraju igru.

4.2. Pravila igre

U igri postoji nekoliko vrsta tornjeva i neprijatelja. Ponašanje tornjeva je sljedeće: ako nema metu, toranj čeka da mu neprijatelj uđe u domet, nakon čega ga smatra metom tako dugo dok ne izađe iz dometa ili ga ne uništi, te potom traži novu metu. Ako ima metu, toranj je napada svakih nekoliko sekundi (ovisno o tipu tornja) te svakim napadom čini štetu određenu tipom tornja. Svaki toranj ima tip napada kojim se određuje hoće li napraviti veću ili manju štetu jedinici koju napada, ovisno o njezinoj vrsti oklopa.

Tipu napada je moguće dodijeliti i modifikator koji svakim napadom na metu stavlja određeni efekt. Primjerice, za tip napada *Frost*, pogođena jedinica dobiva modifikator koji usporava jedinicu za 50% njezine osnovne brzine kretanja na određeno vrijeme.

Kao i tornjeva, postoji nekoliko vrsta neprijatelja. Njihovo ponašanje je jednostavno - kreću se po cesti dok ne dođu do cilja i tada igraču čine štetu ovisnu o tipu ne-

prijatelja. Svaki neprijatelj ima određeno zdravlje, brzinu kretanja te količinu resursa koju igrač dobije kada je uništi. Neprijatelji posjeduju oklop koji utječe na količinu štete koju nanose različiti tipovi tornjeva.

Učinci različitih napada na različite oklope prikazani su u tablici 4.1.

Tip napada	Light	Organic	Heavy	Posebni efekti
Normal	125%	100%	75%	-
Frost	100%	75%	100%	Usporava jedinice
Energy	75%	100%	125%	-

Tablica 4.1: Prikaz količine štete ovisno o tipu napada i tipu oklopa

Igrač može graditi tornjeve samo uz cestu. Svaki toranj ima određenu cijenu i igrač ga ne može kupiti ako nema dovoljnu količinu resursa. Igrač može prodati određeni toranj, čime se oslobađa mjesto za gradnju i igrač dobiva natrag pola iznosa njegove početne cijene.

4.3. Tehnologije

Kod ostvarenja rješenja korištene su sljedeće tehnologije:

- **Microsoft Visual Studio** kao razvojna okolina i (engl. *compiler*) za program koji je pisan u jeziku C++
- **Evolutionary Computation Framework (ECF)** - biblioteka koja pruža funkcionalnost evolucijskih algoritama
- **SDL2** - biblioteka za prikaz multimedijalnih sadržaja, dohvaćanje ulaza s tipkovnice i miša, itd.

4.4. Ostvarenje rješenja

Program se sastoji od tri dijela - igre, grafičkog sučelja za upravljanje igrom i simulatora.

4.4.1. Igra

Razred *Game* sadrži logiku same igre i izveden je tako da mu nije potrebno grafičko korisničko sučelje, već je njime moguće upravljati programskim sučeljem, što je korisno za genetski algoritam.

```
Game(GameInterface* gameInterface, Map* map, bool
   aiControlled);
```

Kod instanciranja razreda, kao *gameInterface* se može poslati pokazivač na grafičko sučelje ili *NULL* (ako se igra koristi samo u genetskom algoritmu). Parametar *map* predstavlja pokazivač na podatke o razini igre.

Ako parametar *aiControlled* ima vrijednost true, ručna kontrola igre će biti onemogućena i umjesto igrača će igrati umjetna inteligencija, koja interpretira red naredbi za izgradnju (engl. *build order*). Red naredbi se proslijeđuje igri preko nekoliko funkcija, a moguće ga je konstruirati iz niza realnih brojeva ili niza bitova. Također, moguće je i niz naredbi učitati iz datoteke.

Struktura naredbe je jednostavna:

```
struct BuildCommand{
  unsigned int towerType;
  unsigned int socket;
}
```

U strukturi, član *towerType* odgovara indeksu tipa tornja u globalnom nizu *types* u prostoru imena *TowerTypes*. Član socket predstavlja indeks slobodnog mjesta uz cestu. Budući da je tornjeve moguće graditi isključivo uz cestu (zbog pravila igre), nema smisla voditi računa o koordinatama, već jednostavno o rednom broju slobodnog mjesta. Ovo također poboljšava efikasnost genetskog algoritma - ne troši se vrijeme na pokušaje da se gradi na nedozvoljenim poljima.

Nakon inicijalizacije, igra ulazi u glavnu petlju koja se izvršava dok igra ne završi. Ako je uključena umjetna inteligencija, svaki ciklus petlje izvršava sljedeće:

```
if (this->aiControlled)
{
  if (!commandQueue.empty())
  {
    BuildCommand b = commandQueue.front();
    if (buyTower(this->socketTiles[b.socket],
       &TowerTypes::types[b.towerType]))
```

```
      {
        commandQueue.pop();
      }
    }
}
```

Funkcija *buyTower* provjerava ima li igrač dovoljno resursa da kupi toranj, i ako ima, izgrađuje ga na određenom praznom mjestu (engl. *socket*), te vraća *true*. Ako na željenom mjestu već postoji toranj, uništit će se i igrač će dobiti pola njegove cijene za izgradnju natrag, te će se izgraditi novi toranj na istom mjestu i funkcija će opet vratiti *true*. Inače funkcija vraća *false*.

Razina je spremljena uobliku dvodimenzionalnog polja (razred *Map*), a tornjevi i neprijateljske jedinice se drže u dinamičkim nizovima pokazivača.

Kraj igre i dobrota

Igra traje dok varijabla *state* u razredu *Game*, koja predstavlja stanje igre, ima vrijednost *Playing*. Kada igrač pobijedi, stanje postaje *Won*, a kada izgubi, postaje *Lost*.

Dobrota određenog niza naredbi se računa na sljedeći način:

```
double Game::getFitness(){
  return this->timeElapsed;
}
```

Dakle, boljim igračem se smatra onaj koji dulje izdrži protiv neprijatelja.

Vrste tornjeva

Struktura vrste tornja je sljedeća:

```
struct TowerType{
  std::string name;

  float baseDamage;
  float baseAttackCooldown;
  float baseCost;
  float attackRange;

  AttackType attackType;
```

```
  unsigned int spriteSeed;
};
```

Član *attackRange* predstavlja domet tornja. Period napada je određen članom *baseAttackCooldown*, a količina štete članom *baseDamage*. Član *attackType* određuje tip napada. Cijena tornja određena je članom *baseCost*. Član *spriteSeed* predstavlja broj na temelju kojeg se generira izgled tornja.

Struktura tipa napada je sljedeća:

```
enum AttackType{
  Normal, //extra damage vs. light, reduced damage to heavy
     armor
  Frost, //apply slow Modifier, reduced damage to organic
     armor
  Energy, //extra damage to heavy, reduced damage to light
     armor
};
```

Razred konkretnog modifikatora napada je naslijeđen od razreda *EnemyModifier*, te mu nadograđuje funkciju *update*. Primjerice, deklaracija konkretnog razreda *FrostModifier* je sljedeća:

```
class FrostModifier : public EnemyModifier {
public:
  FrostModifier(Enemy* enemy, float timer);
  virtual void update(float deltaTime);
};
```

Svaki neprijatelj ima listu modifikatora koji su trenutno na njemu, a efekti modifikatora se događaju svaki ciklus glavne petlje igre (engl. *frame*), tj. u funkciji *update*, za čije je pozivanje zadužen razred *Enemy* u svojoj funkciji *update*. Sama logika modifikatora je jednostavna:

```
void FrostModifier::update(float deltaTime){
  EnemyModifier::update(deltaTime); //base class handles
     modifier expiration

  this->enemy->setSpeed(this->enemy->getType()->baseSpeed *
     0.5f);
}
```

Vrste neprijatelja

Struktura vrste neprijatelja je sljedeća:

```
struct EnemyType{
  std::string name;

  float baseHealth;
  float baseDamage;
  float baseSpeed;
  unsigned int baseReward;

  ArmorType armorType;

  unsigned int spriteSeed;
};
```

Član *baseHealth* određuje količinu štete koju neprijatelj može podnijeti. Brzina neprijatelja u pikselima po sličici (engl. *pixels per frame*) je određena članom *baseSpeed*. Član *baseDamage* određuje količinu štete koju neprijatelj čini kada stigne do cilja. Količina resursa koju igrač dobiva uništenjem neprijatelja je određena članom *baseReward*. Član *armorType* određuje vrstu oklopa jedinice:

```
enum ArmorType{
  Light,
  Organic,
  Heavy
};
```

Tipovi napada i oklopa su podložni promjenama po procjeni dizajnera. Moguće je lako dodavati nove ili uređivati postojeće, te dodavati nove tipove modifikatora za pojedine tipove napada.

Format razine

Razine u igri su spremljene kao tekstualne datoteke. Prve dvije linije datoteke moraju imati sljedeći format:

```
WIDTH <sirina u poljima>
HEIGHT <visina u poljima>
```

Nakon toga slijedi linija koja označava da počinje blok s vrstama polja. Polja se označavaju znamenkama, kod kojih vrijedi preslikavanje u tablici 4.2.

Znamenka	Tip polja
0	Normalno tlo
1	Cesta po kojoj se neprijateljske jedinice kreću
9	Početak ceste, tj. polje s kojeg neprijatelji dolaze

Tablica 4.2: Preslikavanje znamenaka na vrste polja

Primjerice, za priloženu razinu *map_01.map*, blok ima sljedeću strukturu:

```
BEGIN_MAP_DATA
0090000000000000
0010000000000000
0011111111111100
0000000000000100
0000000000000100
0000000000000100
0000000000000100
0000000000000100
0000000011110100
0000000010010100
0000000010011100
0011111110000000
0010000000000000
0011111111111000
0000000000001000
END_MAP_DATA
```

Nakon definiranja izgleda razine, slijedi definiranje redoslijeda neprijateljskih jedinica. Neprijateljske jedinice dolaze u konstantnim vremenskim razmacima, a dizajner treba samo odrediti njihove tipove i redoslijed. To se čini tako da se navede broj koji označava indeks vrste neprijatelja u izvornom kodu. Za svakog neprijatelja navodi se broj u zasebnom retku, a kod izvršenja se oni izvode po redu.

```
(...)
0
1
0
```

```
1
```

Nakon što se pročita zadnji indeks neprijatelja, razina se smatra učitanom te slijedi njezina inicijalizacija.

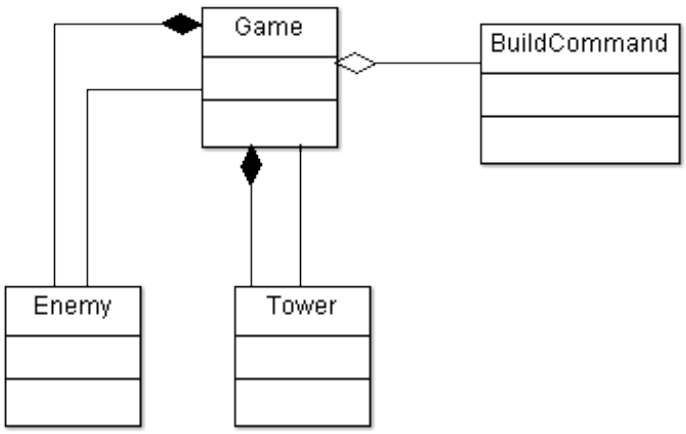
Kod inicijalizacije se stvara staza (engl. *path*) koji slijede svi neprijatelji tako da se provede varijanta *flood-fill* algoritma počevši od prvog polja ceste. Staza se zatim pohranjuje u niz dvodimenzionalnih vektora koji neprijatelji koriste kod kretanja. Zatim slijedi određivanje dozvoljenih mjesta za gradnju, koja mogu biti isključivo uz cestu, a određuju se jednostavnih prolaskom kroz sva polja i provjerom nalazi se polje uz cestu. Na kraju se stvara red neprijatelja iz učitanih indeksa:

```
for (int i = 0; i < map->enemyOrder.size(); i++)
     this->enemyQueue.push(new Enemy(this->evolutionMode,
        &EnemyTypes::types[map->enemyOrder[i]], this));
```

Struktura igre

Svaka igra sadrži nekoliko kolekcija pokazivača kojima prati tornjeve, neprijatelje i redoslijed preostalih neprijatelja (Slika 4.2).

Iako je omogućen rad igre bez grafičkog sučelja, igra je zadužena za dio iscrtavanja ako je grafičko sučelje prisutno da se spriječi dodatna složenost koda.

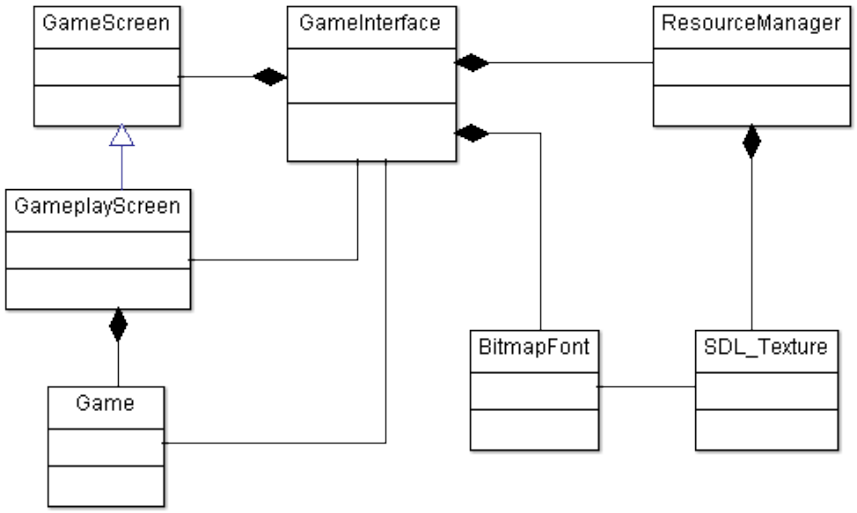


Slika 4.2: UML Dijagram modula igre

4.4.2. Grafičko sučelje za upravljanje igrom

Budući da svrha rada nije bila napraviti potpunu igru, grafičko sučelje je minimalno.

Razred *GameInterface* sadrži logiku grafičkog sučelja za upravljanje igrom. Sučelje upravlja scenama preko apstraktnog sučelja *GameScreen* koji predstavlja jednu scenu u igri. Primjerice, glavni izbornik je jedna scena, sama igra je druga scena, itd. Kod za upravljanje samom igrom se nalazi u razredu *GameplayScreen* (slike 4.3).



Slika 4.3: UML Dijagram grafičkog sučelja

Glavna petlja igre ima sljedeću strukturu:

```
while (!this->quit)
  {
    while (SDL_PollEvent(&event))
    {
      if (event.type == SDL_QUIT)
      {
        this->quit = true;
      }
      currentScreen->handleEvents(&event);
    }
    this->currentScreen->update(1 / 60.0f);
    // (...) render code (...)
  }
```

Na početku svake iteracije petlje se obrađuju svi prispjeli događaji (engl. *events*), pozivom funkcije *SDL_PollEvent* u petlji. Nakon što su obrađeni događaji koji su globalni, za cijeli program, obrađivanje preostalih događaja se delegira trenutno aktivnoj sceni igre (*currentScreen*).

Nakon obrađivanja događaja poziva se funkcija *update* trenutne scene koja provodi kontinuiranu logiku igre. Argument koji joj se šalje je vremenski korak (engl. *timestep*). Iako većina modernih igara podržava varijabilni vremenski korak (engl. *variable timestep*), ovdje se to neće koristiti jer bi razlika između simulacije i igranja bila prevelika, te situacija u simulaciji ne bi nužno odgovarala stvarnom scenariju.

Razred *ResourceManager* je zadužen da učitavanje tekstura iz datoteka, njihovo smještanje i dohvaćanje, te njihovo oslobađanje iz memorije nakon što više nisu potrebne.

Razred *BitmapFont* je jednostavna implementacija tekstualnog fonta s jednakim razmacima (engl. *monospace font*) koji se generira iz slike u kojoj su znakovi poredani ASCII redoslijedom (slika 4.4).

Slika 4.4: Primjer korištenog fonta

Za iscrtavanje i prihvat ulaza s miša i tipkovnice se koristi biblioteka SDL2. Većina grafike u igri je iscrtana način da se tekstura iz datoteke učita u memoriju pomoću razreda *ResourceManager*, i tada iscrta na zaslon pomoću funkcija biblioteke SDL2. Iznimka ovome su tornjevi i neprijatelji - za njih su teksture generirane proceduralno da se izbjegne potreba za crtanjem pojedinačnih grafika. U strukturi *TowerType* postoji

```
unsigned int spriteSeed;
```

na temelju kojeg se u biblioteci *Spriteral* generira tekstura za određeni tip neprijatelja. To je jednostavna biblioteka nastala u svrhu ovog rada, koja omogućuje proceduralno generiranje dvodimenzionalne grafike, tako da joj se postavi željeni oblik grafike (engl. *sprite mask*) i broj koji predstavlja *seed*. Moguće je generirati grafiku simetričnu po horizontalnoj osi ili po horizontalnoj i vertikalnoj. Biblioteka vraća niz okteta u RGBA redoslijedu, na temelju kojeg *ResourceManager* stvara teksturu za upotrebu u stvarnom vremenu. Primjer neprijatelja generiranih na ovaj način prikazan je na slici 4.5.

Slika 4.5: Primjer proceduralno generiranih neprijatelja

Maska po kojoj su generirani neprijatelji s slike 4.5 je sljedeća:

```
Spriteral::Uint8 shipMask[16*16] =
{
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 3, 3, 1, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 1, 3, 3, 1, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 1, 3, 3, 1, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 1, 1, 1, 3, 3, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 0, 1, 1, 1, 3, 3, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 0, 0,
  0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 1, 0, 0,
```

```
  0, 0, 1, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 0, 0,
  0, 0, 1, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

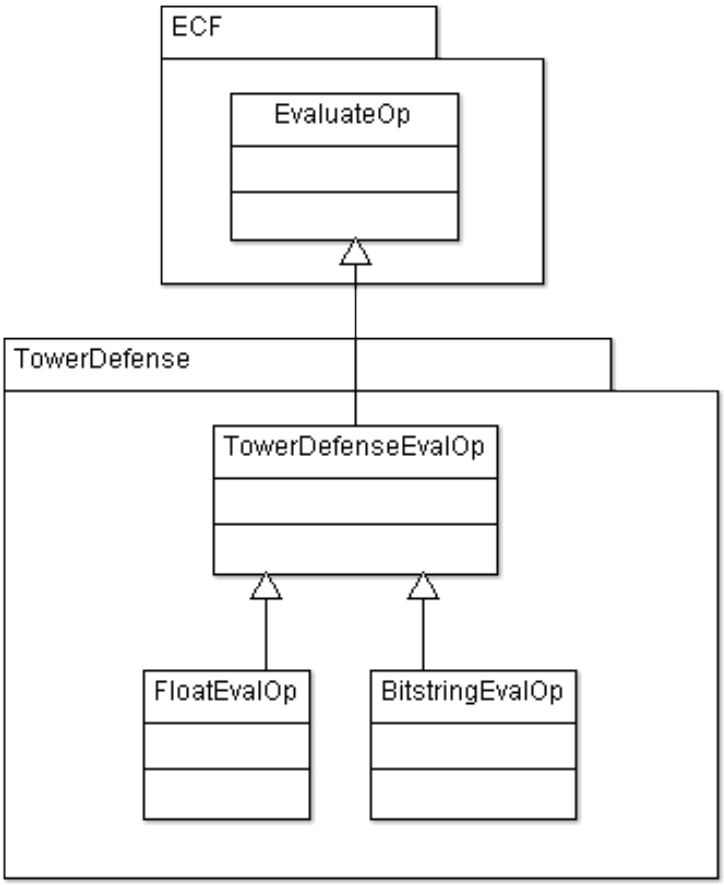
Moguće je stvarati vlastite maske i na temelju njih generirati različite oblike neprijatelja i tornjeva. Mogu se stvarati grafike i većih dimenzija, ali se gubi na kvaliteti i detaljima jer je biblioteka namijenjena za stvaranje grafike manjih dimenzija (engl. *pixel art*).

Malom promjenom u maski moguće je generirati sasvim drukčiji izgled neprijatelja. Primjerice, na slici 4.6 su prikazani generirani neprijatelji nalik tenkovima.

Slika 4.6: Proceduralno generirane grafike tenkova

4.4.3. Simulator

Simulator je dio programa zadužen za provođenje genetskog algoritma nad igrom, a logika mu je smještena u funkciji *main()*. Evaluacijski operatori su smješteni u razrede naslijeđene od razreda *TowerDefenseEvalOp*, koji sadrži pomoćne zajedničke varijable, a taj razred je naslijeđen od razreda *EvaluateOp* koji pripada biblioteki ECF (slika 4.7).



Slika 4.7: UML dijagram razreda evaluacijskih operatora

Evaluacijski operatori imaju funkciju evaluate koja određuje dobrotu na sljedeći način:

1. Stvori instancu razreda *Game* i inicijaliziraj ga
2. Pretvori genotip u red naredbi razreda *BuildCommand*
3. Dodijeli stvoreni red naredbi instanci razreda *Game*
4. Pozivaj funkciju *Game::update()* sve dok igra ne završi
5. Dohvati vrijednost dobrote koja je izračunata u instanci razreda *Game*

Konkretan evaluacijski operator se postavlja ovisno o argumentima programa.

Floating-point Evaluation Operator

Floating-point Evaluation Operator je smješten u razred *FloatEvalOp*, i red naredbi stvara iz niza realnih brojeva u rasponu [0, 1].

Budući da su tornjevi izvedeni kao strukture dva cijela broja (Slika 4.8) (tip tornja i redni broj mjesta na kojem će se graditi), veličina niza realnih brojeva treba biti duplo veća od broja naredbi.

Slika 4.8: Serijalizacija niza naredbi

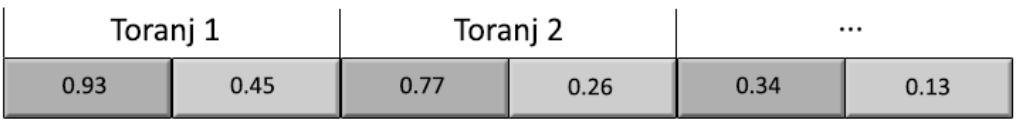
Pojedinačni članovi dobivenog niza realnih brojeva se potom mogu preslikati u članove reda naredbi pomoću sljedeće funkcije:

```
static int mapFloat(float value, float from, float to, int
    low, int high)
{
  float f = (value - from) / (to - from) * (high - low) + low;
  return (int)f;
}
```

Varijabla *value* sadrži vrijednost broja koji se preslikava, *from* i *to* su donja i gornja granica intervala iz kojeg se preslikava, a *low* i *high* su donja i gornja granice intervala brojeva na koji se preslikava.

Budući da je ovo općenita funkcija, a u svrhe ovog rada je potrebno samo uzeti u obzir interval [0, 1], koristi se pojednostavljena verzija:

```
static int mapFloatNormal(float value, int low, int high)
{
  return mapFloat(value, 0.0f, 1.0f, low, high);
}
```



Slika 4.9: Primjer niza realnih brojeva

Primjerice, postupak preslikavanja niza realnih brojeva sa slike 4.9 je sljedeći:

Uzme se prvi broj, te se u funkciju *mapFloatNormal* uvrsti kao *value*, a granice intervala *low* i *high* su 0 i broj vrsta tornjeva. Funkcija vraća cijeli broj koji je indeks vrste tornja u globalnom nizu vrsta tornjeva. Zatim se uzima drugi broj, uvrsti se kao *value*, a *low* i *high* postaju 0 i broj mjesta na kojima je moguće graditi. Funkcija vraća indeks slobodnog mjesta na kojem će se graditi toranj. Postupak se ponavlja za sve parove brojeva, te se na kraju pretvara u red naredbi koji koristi umjetna inteligencija u samoj igri.

Konstruiranje naredbe na temelju dobivenih indeksa je jednostavno:

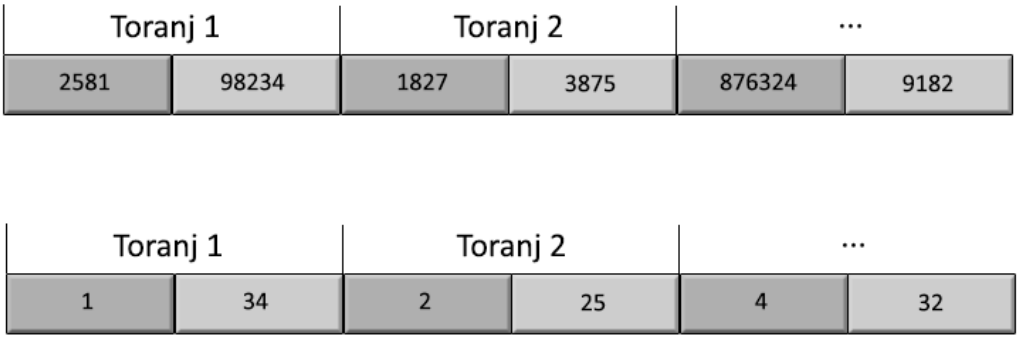
```
int towerIndex = Utility::mapFloatNormal(towerValue, 0,
   towerTypeCount);
int socketIndex = Utility::mapFloatNormal(socketValue, 0,
   socketCount);

BuildCommand command(towerIndex, socketIndex);
commandQueue.push(command);
```

Bitstring Evaluation Operator

Bitstring Evaluation Operator je smješten u razred *BitstringEvalOp*, i red naredbi stvara iz niza bitova.

Red naredbi se iz niza bitova stvara tako da se niz podijeli na skupine od 32 bita, i svaka se skupina pročita kao cijeli broj, te se zatim ograniči na odgovarajući raspon primjenom modulo operatora (slika 4.10).



Slika 4.10: Primjer konstruiranja niza naredbi iz niza bitova, uz pretpostavku da je 5 tipova tornjeva i 50 slobodnih mjesta

Najprije se iz niza bitova konstruira cijeli broj:

```
unsigned int towerValue = 0;
for (int j = i; j < i + 32; j++)
{
  int bit = 0;
  if (bits[j] == true)
    bit = 1;

  towerValue = (towerValue << 1) | bit;
}
```

Zatim se dobiveni cijeli broj ograniči na raspon [0, broj_tornjeva - 1]:

```
unsigned int towerIndex = towerValue % towerTypeCount;
```

Isto se ostvari i za redni broj slobodnog mjesta, te se na temelju tih brojeva konstruira naredba.

```
BuildCommand command(towerIndex, socketIndex);
commandQueue.push(command);
```

4.5. Korištenje ostvarenog rješenja

Program se može pokrenuti na 2 načina - evolucijski i ispitni.

Evolucijski način je izveden bez grafičkog sučelja, a služi da se nad željenom mapom pokrene genetski algoritam koji optimizira nizove naredbi koji igraju igru. Moguće je odabrati jedan od dva tipa genoma koje program podržava - niz bitova ili niz realnih brojeva.

4.5.1. Korištenje evolucijskog načina rada

Na operacijskom sustavu *Windows*, evolucijski način se pokreće na sljedeći način:

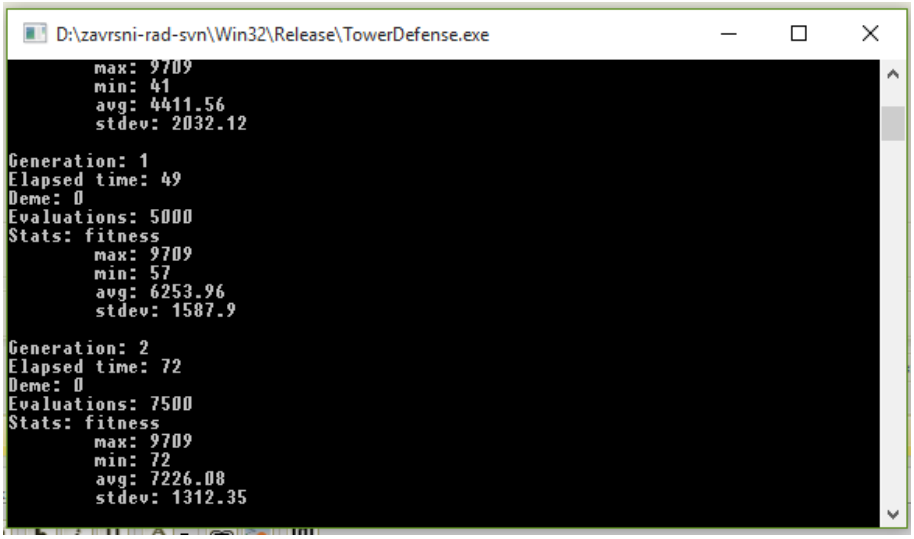
```
TowerDefense.exe path_to_map –evolution evolution_type
```

Gdje je *path_to_map* putanja do datoteke s definicijom razine, a *evolution_type* tip genoma koji će se koristiti u genetskom algoritmu, što može biti *-float* ili *-bitstring*. Parametar *-evolution* znači da pokrećemo program u evolucijskom načinu rada. Primjerice, pokretanje evolucije nad razinom pod imenom "map_01.map" u kazalu "Maps",

s nizom realnih brojeva kao genomom, će se izvršiti ovako:

```
TowerDefense.exe Maps/map_01.map -evolution -float
```

Pokretanjem programa na ovaj način počinje izvođenje genetskog algoritma (slika 4.11) koje nakon određenog broja generacija prestaje, te se najbolja jedinka zapisuje u datoteku *best-bitstring.bin* ili *best-float.bin*, ovisno o korištenom genomu.



Slika 4.11: Prikaz evolucijskog načina rada

Broj generacija, veličinu populacije, konkretan genetski algoritam, vjerojatnost mutacije, i slične parametre je moguće podešavati uređivanjem tekstualne datoteke *parameters-bitstring.txt* ili *parameters-float.txt*, ovisno o korištenom genomu.

Primjerice, datoteka *parameters-bitstring.txt* ima sljedeću strukturu:

```
<ECF>
  <Genotype>
    <BitString>
      <Entry key="size">1600</Entry>
    </BitString>
  </Genotype>
  <Registry>
    <Entry key="log.filename">ECF/log.txt</Entry>
    <Entry key="population.size">2500</Entry>
    <Entry key="mutation.indprob">0.3</Entry>
    <Entry key="term.maxgen">10</Entry>
  </Registry>
</ECF>
```

Gdje unos *size* odgovara veličini genotipa, što je u slučaju niza bitova broj bitova, a u slučaju niza realnih brojeva broj realnih brojeva. Unos *population.size* odgovara veličini populacije, a *term.maxgen* je maksimalan broj generacija, tj. broj generacija nakon kojeg izvršenje genetskog algoritma prestaje. Unos *mutation.indprob* označava vjerojatnost mutacije i zadaje se u rasponu [0, 1]. Moguće je i ispisati dnevnik pomoću unosa *log.filename*.

4.5.2. Korištenje ispitnog načina rada

Ispitni način ne pokreće genetski algoritam, već otvara binarnu datoteku u kojoj je upisan najbolji niz naredbi dobiven evolucijom, te prikazuje kako ga umjetna inteligencija koristi za igranje određene razine igre.

Na operacijskom sustavu *Windows*, ispitni način se pokreće na sljedeći način:

```
TowerDefense.exe path_to_map evolution_type
```

Gdje je *path_to_map* putanja do datoteke s definicijom razine, a *evolution_type* tip genoma koji će se koristiti u genetskom algoritmu, što može biti *-float* ili *-bitstring*. Primjerice, pokretanje ispitnog načina nad razinom pod imenom "map_01.map" u kazalu "Maps", s nizom bitova kao genomom, će se izvršiti ovako:

```
TowerDefense.exe Maps/map_01.map –bitstring
```

Pokretanjem programa na ovaj način se pokreće i interaktivno grafičko sučelje (slika 4.12), te umjetna inteligencija počinje izvoditi red naredbi učitan iz datoteke u kojoj je spremljen najbolji red za određen genom.

Slika 4.12: Prikaz ispitnog načina rada

Prelaskom pokazivača miša preko tornjeva prikazuju se njihovi atributi. Crvena linija povučena od tornja do jedinice označava projektil koji je toranj ispalio na jedinicu. Tamna izbočena polja označavaju prazna mjesta na kojima je moguće graditi tornjeve. Iznad svake jedinice se prikazuje traka koja prikazuje trenutno zdravlje jedinice (engl. *healthbar*). U lijevom donjem kutu prikazuje se količina resursa kojom igrač raspolaže, a u donjem desnom kutu prikazuje se koliko je igraču ostalo zdravlja.

5. Rezultati

5.1. Parametri simulacije

Metodom pokušaja i pogreške ustanovljeno je da je dobro prekinuti algoritam nakon 50000 evaluacija. Pokazalo se da ako algoritmu treba dulje da nađe rješenje, ono vjerojatno ne postoji ili je presloženo da ga reproducira čovjek.

Prilikom razvijanja programskog ostvarenja pokazalo se da program najbolje rezultate daje za razine veličine 16x15 polja, budući da su dovoljno velike da se igraču omogući sloboda izbora, a dovoljno male da evolucija ne traje predugo.

Isprobane su varijante s nekoliko različitih vremenskih koraka (engl. *timestep*) i ustanovljeno je optimalna vrijednost 24 sličice po sekundi (engl. *frames per second, FPS*). Kod uzimanja većih vremenskih koraka, smanjilo se vrijeme potrebno za evoluciju, međutim, kvaliteta simulacije se znatno smanjila i često se nije ponašala očekivano jer je povlačila lošu preciznost mjerača vremena na kojima se baziraju učestalosti napada i brzina kretanja jedinica. Kod uzimanja manjih vremenskih koraka, kvaliteta simulacije se povećala, ali je vrijeme potrebno za evoluciju znatno poraslo.

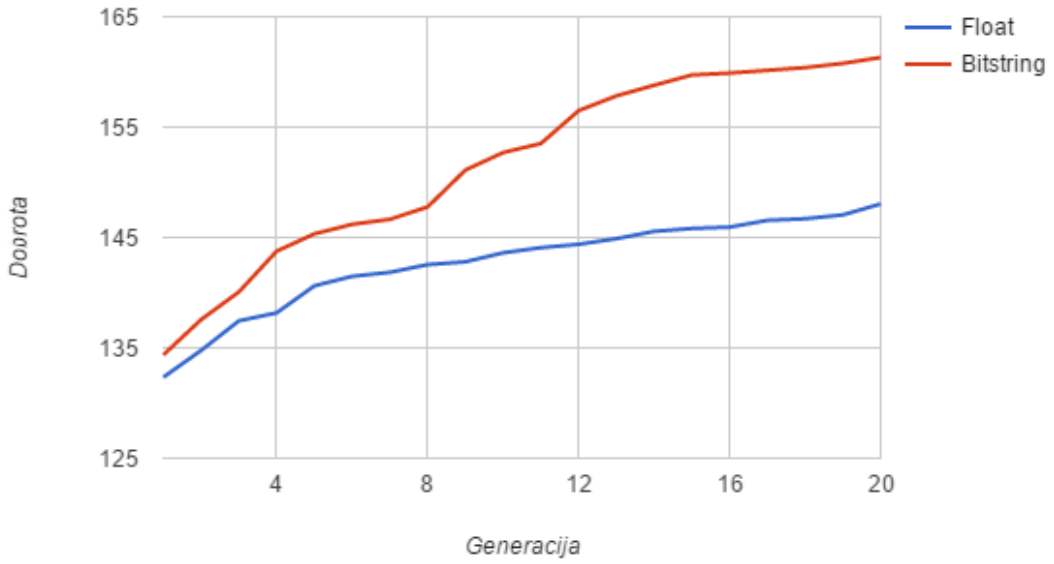
Tijekom razvoja rješenja korištena je sljedeća funkcija dobrote:

```
double Game::getFitness()
{
  return this->health + this->enemiesKilled;
}
```

Međutim, pokazalo se da je ta funkcija loša - iako je davala prednost boljim jedinkama, loše jedinke su svejedno bile evaluirane kao dobre. To je bilo ispravljeno uvođenjem nove funkcije dobrote koja jedinku smatra boljom ako je dulje izdržala u igri.

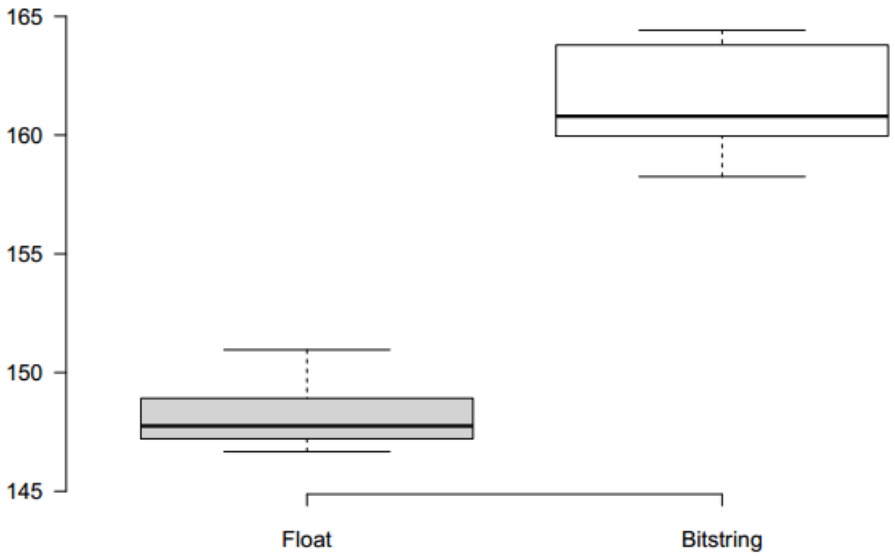
5.2. Rezultati simulacije

Korištena su dva genotipa, i ovisnost maksimalne dobrote o njima kroz generacije, mjerena na populaciji od 2500 jedinki, je prikazana na slici 5.1.



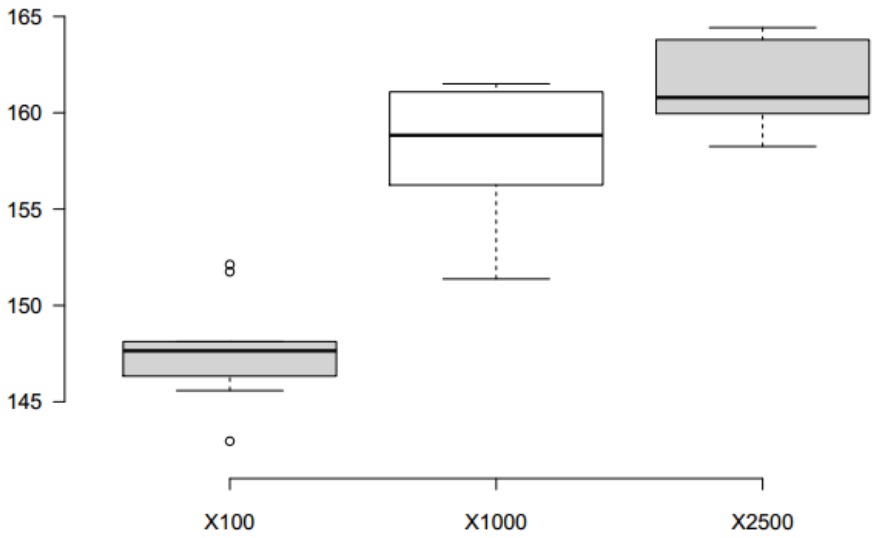
Slika 5.1: Ovisnost maksimalne dobrote o genotipu kroz generacije

Na slici 5.2 vidljivo je kretanje dobrote najboljih jedinki kroz 10 pokretanja, mjereno na populaciji od 2500 jedinki.

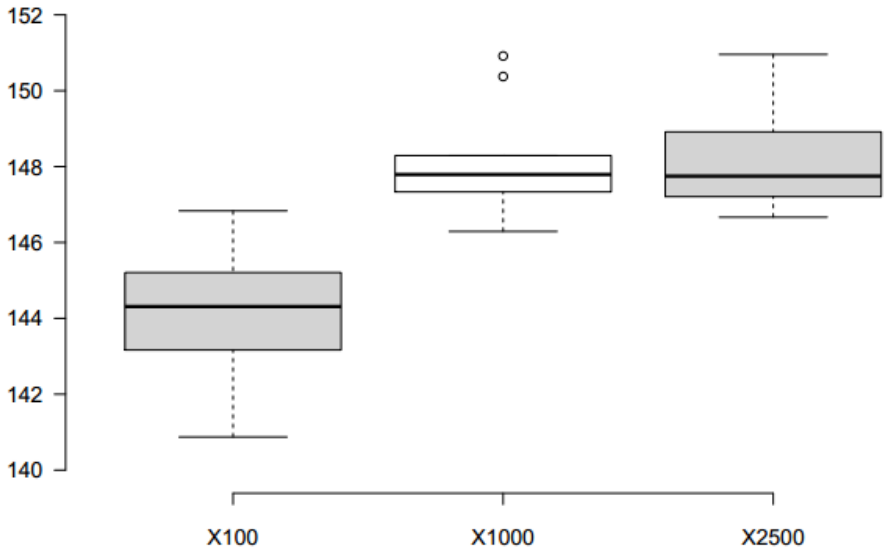


Slika 5.2: Boxplot dobrote najboljih jedinki

Na slici 5.3 vidljiva je ovisnost dobrote o veličini populacije, mjerena na 10 najboljih jedinki iz populacija od 100, 1000 i 2500 za genotip u obliku niza bitova. Na slici 5.4 je ista ovisnost vidljiva za niz realnih brojeva.

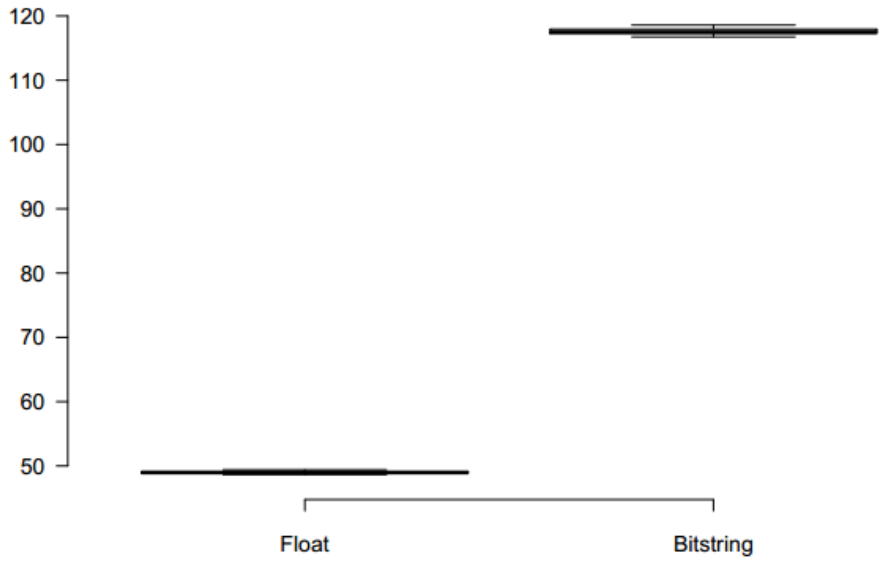


Slika 5.3: Boxplot dobrote ovisne o veličini populacije za niz bitova



Slika 5.4: Boxplot dobrote ovisne o veličini populacije za niz realnih brojeva

Provođenjem evolucije nad nerješivom razinom rezultira stagnacijom kretanja maksimalne dobrote za oba genotipa, kao što je vidljivo na slici 5.5. Razina nad kojom je provedena evolucija je jednostavna ravna staza po kojoj idu samo najjače jedinice.



Slika 5.5: Boxplot maksimalne dobrote na nerješivoj razini

Za evoluciju populacije od 2500 jedinki preko 20 generacija na osobnom računalu bilo je potrebno približno 20 minuta. Za 1000 jedinki je bilo potrebno približno 10 minuta, a za 100 jedinki manje od 5 minuta.

Gledano iz perspektive dizajnera, program postiže relativno dobre rezultate - moguće je u razumnom vremenu dobiti odgovor je li razina otprilike rješiva. Naravno, ovom metodom nije moguće dokazati da je neka razina nerješiva, ali ako se dopusti da se algoritam izvršava dovoljno dugo i nađe rješenje, može se zaključiti da će igrač također teško doći do takvog rješenja. S druge strane, ako algoritam u prosjeku prebrzo nađe rješenje razine, može se zaključiti da je razina prelaka i da je treba otežati.

6. Zaključak

Iako je ostvarena igra jednostavna za današnje standarde, ovakvu metodu je moguće primjeniti i na složenijim igrama, pod uvjetom da omogućuju dovoljno brzo simuliranje.

Genetski algoritam pronalazi rješenje relativno brzo, ali ipak ne uklanja u potpunosti ljudski faktor (dizajnera), čak i kod ovako jednostavne igre. Međutim, omogućuje da se uštedi barem jedan dio vremena potreban za razvijanje razina u igri.

Razvojem hardvera se omogućuje da se ovakve metode pronalaska rješenja koriste sve efikasnije, čak i na osobnim računalima. Međutim, budući da se radi o igrama, koje danas obično rade na 60 ciklusa po sekundi, teško je u razumnom vremenu simulirati one sa složenijim izračunima fizike i kontinuiranim kretanjem. Zbog toga bi bolje bilo da se provođenje ovakvih genetskih algoritama paralelizira da se može izvršavati na računalima s više jezgri.

Najveći problem kod razvoja je prilagođavanje igre da se omogući rad s genetskim algoritmom - potrebno je napraviti model igre koji će raditi bez grafičkog prikaza, preko programskog sučelja koje koristi genetski algoritam, što obično jako povećava složenost programa.

LITERATURA

[1] Ernest Adams Andrew Rollings. *Andrew Rollings and Ernest Adams on Game Design*. New Riders Publishing, 2003.

[2] Sean Baron. *Cognitive Flow: The Psychology of Great Game Design*. URL http://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php.

[3] Marshall Cavendish Corporation. *Mammal Anatomy: An Illustrated Guide*. Marshall Cavendish Corporation, 2010.

[4] Benj Edwards. *The History of Civilization*. URL http://www.gamasutra.com/view/feature/1523/the_history_of_civilization.php?print=1.

[5] Marin Golub. *Genetski Algoritam (prvi dio)*. URL http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf.

[6] T. Ryan Gregory. *Understanding Natural Selection: Essential Concepts and Common Misconceptions*. 2009.

[7] Darko Grundler. *Evolucijski Algoritmi*. URL hrcak.srce.hr/file/8743.

[8] David Sirlin. *Balancing multiplayer games*. URL http://www.sirlin.net/article-archive/.

[9] Patrick Wyatt. *The making of Warcraft*. URL http://www.codeofhonor.com/blog/the-making-of-warcraft-part-1.

Optimizacija parametara strateških igara evolucijskim algoritmima

Sažetak

Opisuje se problem optimizacije parametara strateških igara u cilju omogućavanja igrivosti. Definira se vremenski ovisan model strateške igre, postojećih ograničenja i mogućnosti izbora igrača. Istražuju se načini prikaza poteza igrača pogodnih za uporabu u optimizacijskom algoritmu. Ostvaruje se programski sustav koji uključuje model simulacije strateške igre i postupke optimizacije poteza igrača uz pomoć evolucijskih algoritama. Ispituju se mogućnosti ostvarenog sustava u pronalaženju niza poteza koji vodi do pobjede igrača, s obzirom na zadana ograničenja.

Ključne riječi: Genetski algoritam, evolucijski, igra, strategija, strateška igra, toranj, *tower-defense*, optimizacija

Strategy game parameter optimization with genetic algorithms

Abstract

Describes the problem of optimizing strategy game parameters with playability in mind. Defines the time-dependent model of a strategy game, its limits and player's choices. Describes the ways of implementing player moves suitable to be used in a genetic algorithm. Implements a program which includes a strategy game model and optimization of player moves using evolutionary algorithms. Tests the possibilities of the implemented program in showing possible moves that lead to player victory, with established limits in mind.

Keywords: Genetic algorithm, evolutionary, game, strategy, tower, tower-defense, optimization