SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6494

OPTIMIRANJE RASPOREĐIVANJA VOJNIKA U IGRI OBRANE TVRĐAVA

Marin Njirić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6494

OPTIMIRANJE RASPOREĐIVANJA VOJNIKA U IGRI OBRANE TVRĐAVA

Marin Njirić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6494

Pristupnik:	**Marin Njirić (0036506204)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	prof. dr. sc. Marin Golub

Zadatak: **Optimiranje raspoređivanja vojnika u igri obrane tvrđava**

Opis zadatka:

Definirati optimizacijski problem raspoređivanja vojnika u strateškoj igri obrane tvrđava. Razmotriti mogućnost rješavanja specifičnog problema raspoređivanja vojnika za slučaj konstantnih vrijednosti tvrđava uz pomoć evolucijskih algoritama. Detaljno opisati odabrani evolucijski algoritam i njegove operatore. Programski ostvariti sustav koji omogućuje zadavanje parametara igre, optimiranje rasporeda vojnika za problem sa zadanim vrijednostima tvrđava i igranje igre za proizvoljan broj igrača. Na nekoliko primjera različitih početnih parametara ispitati djelotvornost optimizacijskog algoritma.

Rok za predaju rada: 12. lipnja 2020.

Sadržaj

1. Uvod 1

2. Evolucijsko računanje 2

2.1. Genetski algoritmi 4

3. Opis igre 6

4. Praktični rad 7

4.1. Genetski algoritam 7

4.1.1. Inicijalizacija 7

4.1.2. Selekcija 10

4.1.3. Križanje 12

4.1.4. Mutacija 14

4.2. Programski sustav 16

5. Eksperimentalni rezultati 18

5.1. Osvrt na rezultate i praktični rad 22

6. Primjene u praksi 23

7. Zaključak 24

8. Literatura 25

1. Uvod

Iako se na razvoj tehnologije i umjetnu inteligenciju najčešće gleda kao na „budućnost“ ili „nešto što dolazi“, možemo se uvjeriti da već živimo u svijetu u kojem oni imaju velik utjecaj. Umjetna inteligencija je ostvarila velik broj postignuća za koje se nije vjerovalo da će biti moguća tako rano. Tako je već 1997. godine računalo Deep Blue pobijedilo svjetskog prvaka Garryja Kasparova u šahu. To pokazuje ogromnu moć izračunavanja velikog broja poteza u kratkom vremenu. Da računalo može konkurirati ljudima i u kreativnom razmišljanju, dokazao je 2011. IMB-ov Watson koji je pobijedio najbolje natjecatelje u igri Jeopardy!. 2018. godine su počela testiranja autonomnih vozila na cestama.

Ovi primjeri, uz brojne druge, pokazuju veliki napredak umjetne inteligencije u područjima za koja se smatralo da ih samo čovjek može savladati. Danas možemo pronaći primjene umjetne inteligencije u medicini, upravljanju robotima, obrazovanju, transportu i mnogim drugim područjima i industrijama. S obzirom na to da mogu pretražiti velik prostor stanja, algoritmi umjetne inteligencije često će dati rješenja kojih se čovjek ne bi dosjetio. Također, ponekad čovjeku neće biti jasno zašto algoritam izvodi određene poteze, ali se ispostavlja da upravo oni vode prema boljem rješenju problema.

Igre su odlična platforma za ispitivanje i razvoj algoritama umjetne inteligencije. Sve imaju neki način bodovanja prema kojem se može mjeriti napredak. Tako možemo reći da je jedno rješenje bolje od drugog ako na kraju igre ima veći broj bodova. Broj različitih rješenja je vrlo velik i ide prema beskonačnosti, stoga ne možemo jednostavno pretražiti sva moguća rješenja i proglasiti najbolje. To nije izvedivo u realnom vremenu. Za rješavanje ovakvih problema postoje drugačije tehnike.

2. Evolucijsko računanje

Evolucijsko računanje je skupina optimizacijskih algoritama koji su inspirirani prirodom. Ti algoritmi simuliraju ponašanje određenih skupina životinja, procese evolucije i druge oblike optimizacije koji se mogu vidjeti u prirodi. To je područje umjetne inteligencije koje se temelji na metodi pokušaja i pogrešaka, što znači ponavljanje pokusa dok se ne ostvari dovoljno dobro rješenje.

Evolucijsko računanje se koristi za rješavanje složenih problema koje ne bi bilo moguće riješiti drugim tehnikama. Ti algoritmi nikada ne mogu garantirati optimalno rješenje zbog načina izvođenja algoritma, ali će uglavnom dati rješenja koja su dovoljno dobra i blizu optimalnih.

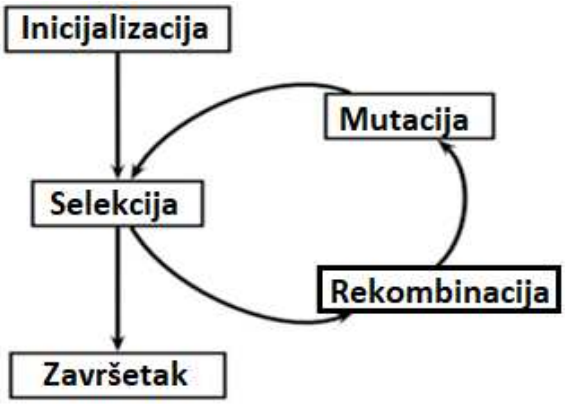
U evolucijskom računanju, nasumično se stvara početni skup kandidata rješenja. Svaka nova generacija kandidata se proizvodi tako da se uklone manje poželjna rješenja, a na ostale kandidate se primjene manje nasumične promjene. To simulira prirodnu selekciju i mutaciju. Kao rezultat ovog postupka, populacija postepeno evoluira i daje sve bolja rješenja problema.

Tehnike evolucijskog računanja su vrlo popularne u računarskoj znanosti jer daju dovoljno optimizirana rješenja za vrlo širok spektar problema.

Tijekom godina razvijeni su mnogobrojni algoritmi u svrhu optimizacije koji oponašaju pojave iz prirode. Jedan od njih je pčelinji algoritam koji je inspiriran načinom na koji pčele pronalaze novo mjesto za košnicu. Pčele izviđači traže nova mjesta za košnicu i zatim pčelinjim plesom promoviraju kvalitetu svoga mjesta. Na temelju plesa svih izviđača određuje se mjesto nove košnice. Mravlji algoritam je nastao temeljem ponašanja mravi pri pronalasku najkraćeg puta do izvora hrane. Mravi prilikom kretanja za sobom ostavljaju feromonski trag koji vremenom ispari. Kroz kraći put se češće prolazi pa je gustoća feromona veća nego na duljim putovima. Mrav se kreće slučajno, ali s većom vjerojatnošću u smjeru u kojem osjeti jači feromonski trag. Kao rezultat, kada jedan mrav nađe dobar put, drugi mravi imaju veću vjerojatnost da ga slijede. Algoritam kiše je algoritam nastao promatranjem kretanja kapljica kiše u skladu s Newtonovim zakonima. Promatraju se kapljice različite mase i visine na kojoj se nalaze. Kapljice jednolikim ubrzanjem slobodnog pada putuju do najnižeg mjesta na tlu. To mjesto je funkcija koju promatra ovaj algoritam.

Implementacija evolucijskog algoritma

1. Nasumično stvaranje inicijalne populacije jedinki
2. Izračunavanje funkcije dobrote svake jedinke te populacije
3. Ponavljanje do prihvatljivog rješenja
    - Izbor najboljih jedinki za reprodukciju (roditelji)
    - Stvaranje nove jedinke kroz rekombinaciju i mutaciju
    - Izračunavanje funkcije dobrote novih jedinki
    - Zamjena najlošijih jedinki novim jedinkama



Slika 2.1. Implementacija evolucijskog algoritma

Metode koje spadaju u područje evolucijskih algoritama:

1. Genetski algoritmi
2. Evolucijske strategije
3. Genetsko programiranje
4. Evolucijsko programiranje
5. Klasifikatorski sustavi sa sposobnošću učenja

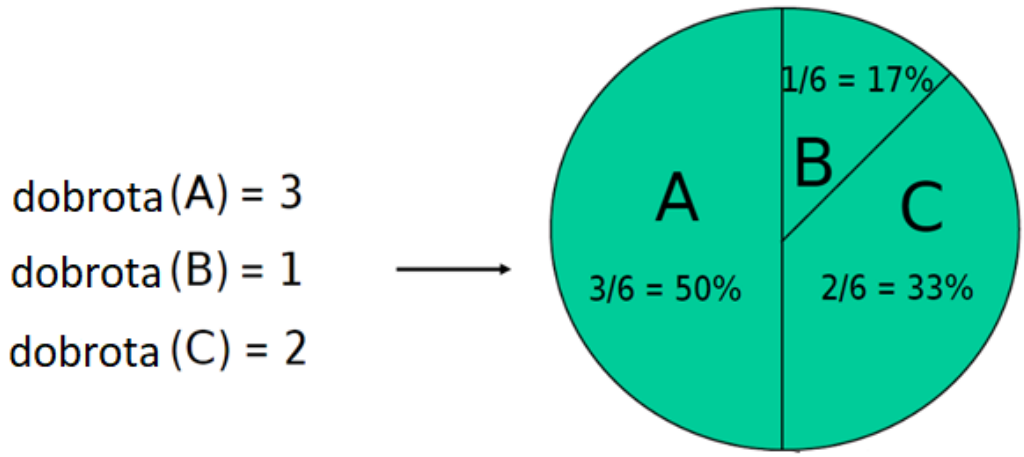
2.1. Genetski algoritmi

Genetski algoritmi su vrsta evolucijskih algoritama koji koriste mehanizme kao što su reprodukcija, mutacija, križanje i selekcija. Potencijalna rješenja imaju ulogu jedinki u populaciji. Mjera kvalitete rješenja, odnosno stupanj zadovoljavanja postavljenog problema, određena je funkcijom dobrote.

Reprodukcija je proces stvaranja novih jedinki koje tvore novu populaciju. Cilj reprodukcije je stvaranje populacija koje će imati sve veću vrijednost funkcije dobrote, odnosno davat će sve bolja rješenja postavljenog problema.

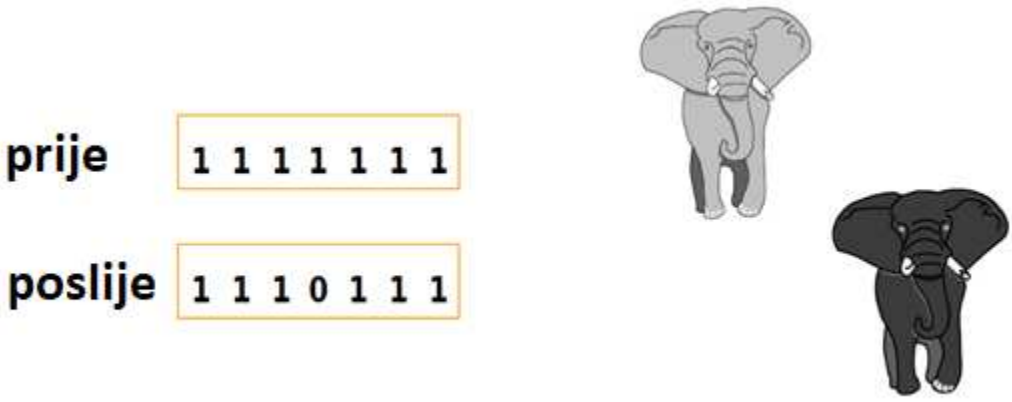
Jedinke koje ulaze u proces stvaranja nove populacije određuju se temeljem procesa selekcije, mutacije i rekombinacije.

Selekcija daje jedinkama bolje šanse za stvaranje sljedeće populacije (da postanu „roditelji“) i bolje šanse za preživljavanje. Selekcijom populacija napreduje prema boljem rješenju i ima sve veći iznos funkcije dobrote. Neke od metoda selekcije su selekcija funkcijom dobrote, turnirska selekcija, jednostavna selekcija i eliminacijska selekcija. Na slici 2.1.1. je prikazana selekcija funkcijom dobrote.



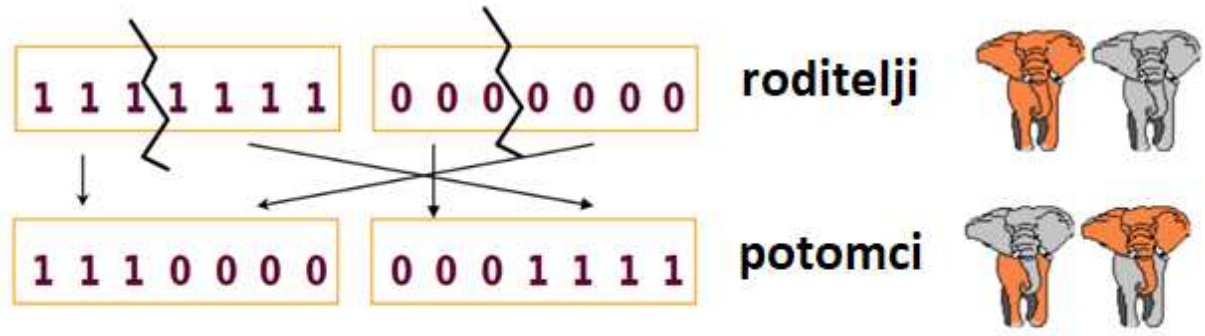
Slika 2.1.1. Selekcija funkcijom dobrote [1]

Mutacija uzrokuje male nasumične promjene jedinke. Služi za održavanje genetske raznolikosti u populaciji. Mutacija se događa uz neku vjerojatnost koja treba biti dovoljno niska jer bi se u suprotnom algoritam pretvorio u nasumično pretraživanje. Razlikujemo nekoliko vrsta mutacija: jednostavna mutacija, miješajuća mutacija, invertirajuća miješana mutacija itd. Slika 2.1.2. je primjer jednostavne mutacije.



Slika 2.1.2. Mutacija [1]

Križanje (rekombinacija) se koristi za kombiniranje genetske informacije dva roditelja za stvaranje nove jedinke (potomka). To je način da se stohastički stvore nova rješenja iz postojeće populacije. Novostvorena rješenja obično prolaze proces mutacije prije nego budu dodana u novu populaciju. Križanje je definirano proizvoljnim brojem prekidnih točaka. Na slici 2.1.3. je prikazano križanje s jednom točkom prekida.



Slika 2.1.3. Križanje [1]

3. Opis igre obrane tvrđava

Igra se temelji na strateškom raspoređivanju određenog broja vojnika na tvrđave s definiranim vrijednostima. Svaki igrač raspoređuje svojih n vojnika na tvrđave na način koji smatra da će mu donijeti najviše bodova. Igra se provodi tako da igra svaki igrač sa svakim. Gleda se jedna po jedna tvrđava. Onaj igrač koji ima više vojnika na tvrđavi dobiva onoliko bodova koliko ta tvrđava vrijedi. Na primjer, imamo 2 igrača i 100 vojnika na raspolaganju, a vrijednosti tvrđava su prikazane na slici 3.1.

Slika 3.1. Primjer vrijednosti tvrđava

Strategija igrača A = (10 10 10 10 10 10 10 10 10 10)

Strategija igrača B = (5 5 5 5 5 15 15 15 15 15)

U ovoj igri igrač A ima više vojnika na prvih 5 tvrđava i dobiva 15 bodova (zbroj vrijednosti prvih 5 tvrđava 1+2+3+4+5), a igrač B ima više vojnika na posljednjih 5 tvrđava i dobiva 40 bodova (zbroj vrijednosti tih tvrđava 6+7+8+9+10). Ovaj izračun provodi se za svaki par igrača i ukupni bodovi se zbrajaju. Igrač koji na kraju ima najviše bodova je pobjednik.

Niti jedan igrač ne zna kakav će raspored vojnika odabrati njegovi protivnici pa želi odabrati strategiju kojom će ih što više pobijediti i donijeti mu što više bodova. Za ovaj primjer sa 100 vojnika i 10 utvrda postoje $\binom{109}{9}$ ~$4.2 * 10^{12}$ različitih rasporeda vojnika. Očito da se ne mogu ispitati sve moguće kombinacije i da tehnika grube sile za ovaj problem neće funkcionirati. Zbog toga će se optimalna strategija pokušati pronaći koristeći genetski algoritam.

4. Praktični rad

Svaki igrač predstavlja jedinku populacije. Kromosom je određen strategijom igrača (niz od 10 brojeva). Svaki broj unutar tog kromosoma (broj vojnika na pojedinim tvrđavama) predstavlja jedan gen. Početna populacija je stvorena nasumično, a kroz iteracije se odbacuju lošiji kromosomi izvode se promjene na preostalima. Tim postupkom populacija daje sve bolja rješenja i na kraju zadnje iteracije se proglašava najbolje rješenje.

4.1. Genetski algoritam

4.1.1. Inicijalizacija

Postoji više načina za nasumično stvaranje brojeva koji imaju određen zbroj, a odabir samo jedne istraži samo dio prostora rješenja. Iz tog razloga programski su ostvarene 3 različite funkcije za stvaranje jedinke u početnoj populaciji.

Prva funkcija stvara kromosom tako da za svaki gen nasumično odredi broj između 0 i 100 i zatim skalira sve brojeve tako da je ukupan zbroj jednak zadanom parametru. Na kraju poziva funkciju `adaptSoldiers` da izvede korekcije zbog zaokruživanja brojeva. Ta funkcija nasumično raspodijeli višak ili manjak vojnika na ostale tvrđave.

```
public static String randomStrategy(int numOfSoldiers) {
            Integer[] randomStrategy = new Integer[10];
            int sum = 0;

            for(int i = 0; i < 10; i++) {
               randomStrategy[i] = (int) (Math.random()*100);
               sum += randomStrategy[i];
            }

            for(int i = 0; i < 10; i++) {
               randomStrategy[i] = numOfSoldiers*randomStrategy[i]/sum;
            }

            // correction due to rounding to integer
            randomStrategy = adaptSoldiers(randomStrategy);

            String strategy =
      Arrays.toString(randomStrategy).replaceAll("\\[|\\]|,", "");

            return strategy;

      }
```

Kod 1. – prva funkcija stvaranja jedinke

Druga funkcija uzima u obzir broj vojnika na ostalim tvrđavama i određuje broj na temelju preostalih vojnika. Također, ova funkcija uzima u obzir i vrijednosti tvrđava, na način da je veća vjerojatnost da će se veći broj vojnika rasporediti na tvrđavu s većom vrijednosti.

```
public static String randomStrategy2(int numOfSoldiers, Tower tower) {
            Integer[] randomStrategy = new Integer[10];
            int remainingSoldiers = numOfSoldiers;

            for(int i = 0; i < 10; i++) {
            randomStrategy[i] =
      (int)(Math.random()*((float)tower.getTowerValue(i)/tower.getTower
      Sum())*remainingSoldiers*3)%remainingSoldiers;
                  remainingSoldiers -= randomStrategy[i];
                  if(i == 8) {
                        randomStrategy[9] = remainingSoldiers;
                        break;
                  }
            }
                  String strategy =
      Arrays.toString(randomStrategy).replaceAll("\\[|\\]|,", "");

            return strategy;
      }
```

Kod 2. – druga funkcija stvaranja jedinke

Treća funkcija najprije raspoređuje vojnike na tvrđave s većom vrijednosti tako da stvara nasumični broj između 0 i broja preostalih vojnika. Na ovaj način, na vrjednije tvrđave postavlja se širi spektar broja vojnika, a na manje vrijedne utvrde ide ono što preostane.

```
public static String randomStrategy3(int numOfSoldiers, Tower tower) {
            Integer[] randomStrategy = new Integer[10];
            int remainingSoldiers = numOfSoldiers;
            Integer[] sortedTowers = tower.getSortedTowers();

            for(int i = 0; i < 10; i++) {
                  randomStrategy[sortedTowers[i]] =
                  (int)(Math.random()*remainingSoldiers);
                  remainingSoldiers -= randomStrategy[sortedTowers[i]];
                  if(i == 8) {
                        randomStrategy[sortedTowers[i+1]] =
                        remainingSoldiers;
                        break;
                  }
            }
            String strategy =
      Arrays.toString(randomStrategy).replaceAll("\\[|\\]|,", "");

            return strategy;
            }
```

Kod 3. – treća funkcija stvaranja jedinke

Broj igrača zadaje se kao parametar igre. Početna populacija stvara se tako da svaka od 3 navedene funkcije stvara po trećinu populacije. Ovako je pokriven puno veći prostor rješenja nego da je samo jedna funkcija odabrana. Za različite vrijednosti tvrđava neke funkcije će se pokazati boljima u stvaranju jedinki od drugih. Ovo se može ispitati igranjem n igrača početne populacije bez primjene genetskog algoritma. Za svaki primjer igra 30 igrača, pa svaka funkcija stvara po 10 jedinki.

Pokazuje se da za jednolike vrijednosti tvrđava prva funkcija daje najbolje rezultate. Na slici 4.1.1.1. prikazan je rezultat igre gdje je vrijednost svake tvrđave postavljena na 10. Jedinka stvorena prvom funkcijom označena je s A, drugom s B, a trećom s C. Možemo vidjeti da je čak 7 od najboljih 10 stvoreno prvom funkcijom.

	Name	Score	Strategy
1	A5	1835	19 17 20 7 14 3 14 1 3 2
2	A1	1810	5 14 7 16 2 11 10 14 10 11
3	B9	1810	0 16 20 17 5 11 8 6 2 15
4	A10	1775	11 15 17 17 5 1 3 14 11 6
5	B2	1755	24 15 10 10 6 6 3 3 4 19
6	A6	1750	11 14 3 5 13 3 19 13 9 10
7	A2	1745	3 3 9 19 17 3 22 13 8 3
8	A3	1740	17 14 15 4 7 14 4 3 7 15
9	A8	1740	13 14 15 8 12 6 11 0 9 12
10	B10	1735	1 19 7 0 15 15 4 11 2 26

Slika 4.1.1.1. Rezultati za jednolike vrijednosti tvrđava

Druga funkcija je najbolja kada su vrijednosti tvrđava linearno raspoređene. Na primjer, za vrijednosti tvrđava (1 2 3 4 5 6 7 8 9 10) rezultati su prikazani na slici 4.1.1.2. Treća funkcija se pokazuje najboljom kada su velike razlike vrijednosti tvrđava. To pokazuje slika 4.1.1.3. na kojoj su prikazani rezultati za vrijednosti tvrđava (1 2 4 8 16 32 64 128 256 512).

	Name	Score	Strategy
1	B2	995	0 5 1 18 15 4 5 11 16 25
2	B8	975	1 3 9 0 9 17 13 10 14 24
3	B10	948	4 6 8 2 21 1 9 17 9 23
4	A8	935	15 3 5 5 2 4 14 26 24 2
5	A1	927	11 11 5 11 5 6 14 7 16 14
6	B4	926	4 2 13 5 10 12 12 3 13 26
7	B9	920	3 7 4 11 19 1 16 9 9 21
8	B7	883	2 10 9 17 11 9 8 12 2 20
9	B1	860	3 3 11 13 14 18 7 3 13 15
10	A6	856	0 19 12 13 0 0 18 6 17 15

Slika 4.1.1.2. Rezultati za linearne vrijednosti tvrđava

	Name	Score	Strategy
1	C3	19667	1 0 0 0 0 0 1 4 20 74
2	C1	19011	1 0 0 0 0 1 1 15 10 72
3	B3	18945	0 0 0 0 4 5 16 22 12 41
4	B7	18857	0 0 0 1 3 3 2 15 37 39
5	C4	18847	1 0 1 0 2 2 9 0 8 77
6	B10	18489	0 0 0 1 3 7 11 20 20 38
7	C7	18263	1 1 0 0 0 1 1 3 48 45
8	C9	18215	1 1 0 0 0 0 1 6 31 60
9	C5	18035	1 0 0 0 0 3 9 11 51 25
10	B8	18033	0 0 0 2 4 6 6 2 22 58

Slika 4.1.1.3. Rezultati za eksponencijalne vrijednosti tvrđava

4.1.2. Selekcija

Nakon stvaranja početne populacije započinje djelovanje genetskih operatora selekcije, križanja i mutacije koji se odvijaju iterativno. U svakoj iteraciji selekcijom se bira koje jedinke će ostati u populaciji i koje jedinke će ući u procese križanja i mutacije.

Za ovaj problem programski je ostvarena kombinacija eliminacijske i turnirske selekcije. U svakoj iteraciji najlošijih 10% jedinki je izbačeno iz populacije i umjesto njih stvaraju se nove jedinke. Na taj način populacija se obnavlja. Nove jedinke stvorene su pomoću jedne od 3 već spomenute funkcije. Ovim postupkom spriječeno je da jedinke konvergiraju prema početnom najboljem rješenju. Također, nova jedinka može biti iz područja koje nije „istraženo" početnom populacijom pa tako može doprinijeti poboljšanju konačnog rješenja.

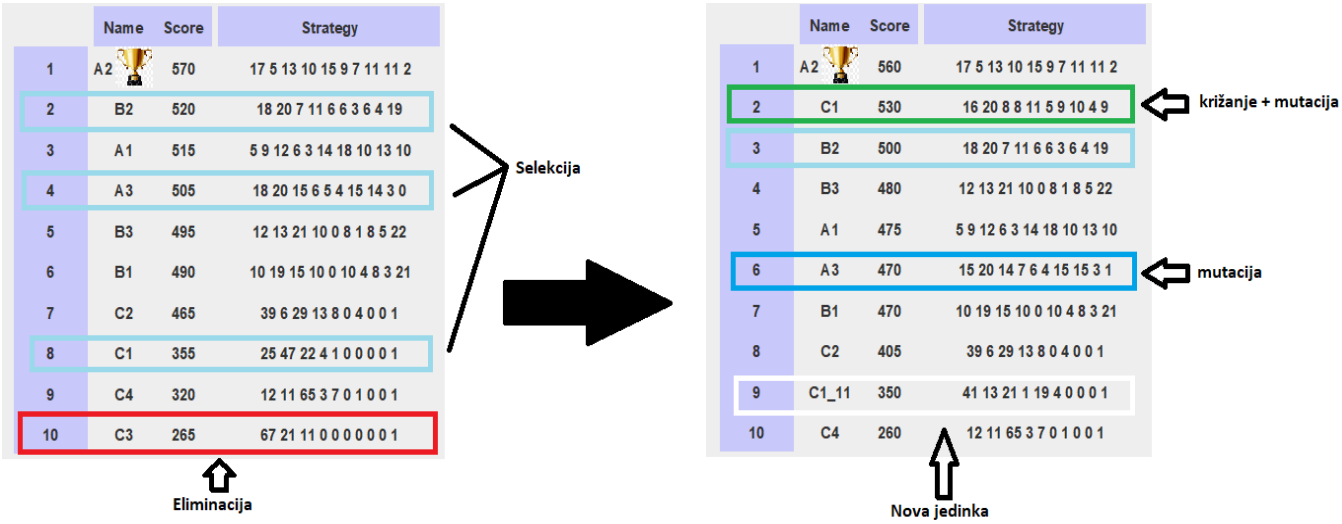
Nakon eliminacijske selekcije slijedi turnirska selekcija. Među preostalim jedinkama nasumično se odabiru 3 jedinke. Najbolja od njih postaje prvi roditelj i u sljedeću iteraciju ulazi nepromijenjena. Druga jedinka postaje drugi roditelj, ali prije nego što uđe u sljedeću iteraciju prolazi kroz proces mutacije. Najlošija od ove 3 izabrane jedinke gubi svoj kromosom, a novi dobiva križanjem kromosoma dva roditelja. Taj kromosom je također mutiran prije nego što uđe u sljedeću iteraciju. Broj turnirskih selekcija ovisi o veličini populacije. Tako će se za svaku iteraciju odviti $0.1 * brojIgrača$ turnirskih selekcija.

```
for(int j = 0; j < numOfPlayers/10; j++) {
    Set<Player> selectedPlayers = GAProcesses.selection(results);
    GAProcesses.crossover(newPlayers, selectedPlayers);
    GAProcesses.mutation(newPlayers,selectedPlayers,mutationProbability);
}
```

Kod 4. – turnirska selekcija

Ova petlja nalazi se unutar petlje koja ide po svim iteracijama. U varijablu `selectedPlayers` spremljene su 3 jedinke iz turnirske selekcije. Zatim se unutar metode `crossover` kromosom najlošije jedinke zamijeni kombinacijom dva roditelja. Na kraju se kromosomi drugog roditelja i najlošije jedinke metodom *`mutation`* promijene uz određenu vjerojatnost i spremni su za ulazak u sljedeću iteraciju.

Primjer izvođenja prikazan je na slici 4.1.2.1. Na lijevoj strani prikazani su rezultati nakon stvaranja početne populacije. Na desnoj strani su rezultati nakon izvođenja prve iteracije i provedbe procesa selekcije, križanja i mutacije.



Slika 4.1.2.1. Izvođenje genetskih operatora

Najprije djeluje eliminacijska selekcija i najlošijih 10% (ovdje 1) je izbačeno iz populacije. Može se vidjeti da jedinke C3 označene crveno više nema na desnoj slici koja predstavlja sljedeću iteraciju. U ovom primjeru izabrane su jedinke B2, A3 i C1 u turnirsku selekciju. B2 kao najbolja ostaje nepromijenjena u sljedećoj iteraciji. Kod A3 možemo vidjeti manje promjene uzrokovane mutacijom. Jedinka C1 nastala je kombinacijom kromosoma roditelja B2 i A3 i nakon provedene mutacije zapravo je postala bolja od svojih roditelja. Mjesto u populaciji od C3 popunila je nova jedinka C1_11 s potpuno novim nasumično stvorenim kromosomom. Ovaj postupak ponavlja se broj puta koji je određen parametrom igre. Svakom iteracijom populacija napreduje i daje sve bolja rješenja.

4.1.3. Križanje

Nakon selekcije opisane u prethodnom poglavlju, najslabija jedinka od 3 odabrane dobiva novi kromosom križanjem kromosoma roditelja. Kombiniranjem dva bolja rješenja, ta jedinka poboljšava svoj položaj u populaciji (povećava iznos funkcije dobrote). Time dobiva veće šanse da će dulje ostati u populaciji i sudjelovati u stvaranju novih kromosoma.

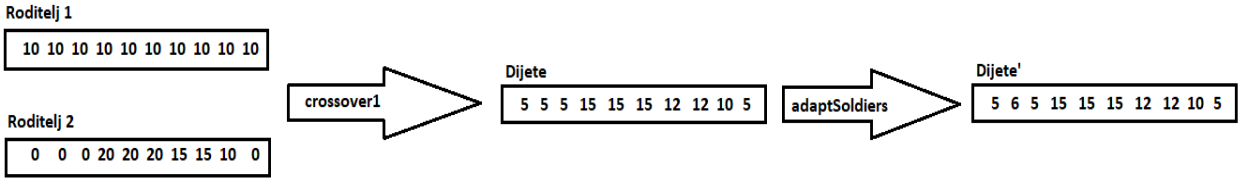
Postoje različite vrste križanja, a u ovom programskom rješenju se koriste tri. Svaki put kad su jedinke poslane u proces križanja, nasumično se odabere i provede jedna od te tri vrste.

```
public static void crossover(List<Player> players, Set<Player>
selectedPlayers) {

      Player parent1 = (Player) selectedPlayers.toArray()[0];
      Player parent2 = (Player) selectedPlayers.toArray()[1];
      Player weakestPlayer = (Player) selectedPlayers.toArray()[2];

      String[] parent1Strategy = parent1.getStrategy().split(" ");
      String[] parent2Strategy = parent2.getStrategy().split(" ");
      Integer[] newStrategy = new Integer[10];

      int crossoverType = (int)(Math.random()*3);
      switch(crossoverType) {
            case(0): newStrategy =crossover1(parent1Strategy,
parent2Strategy, newStrategy);
            break;
            case(1): newStrategy = crossover2(parent1Strategy,
parent2Strategy, newStrategy);
             break;
            case(2): newStrategy = crossover3(parent1Strategy,
parent2Strategy, newStrategy);
            break;
            }
}

      newStrategy = MainThread.adaptSoldiers(newStrategy);
      …
}
```

Kod 5. – križanje

Funkcije *crossover1*, *crossover2* i *crossover3* stvaraju novi kromosom spremljen u varijablu `newStrategy` na način objašnjen u nastavku. Ukupan broj vojnika svakog igrača zadan je parametrom igre i mora biti konstantan. Nakon stvaranja kromosoma, vrlo je vjerojatno da ukupan broj vojnika neće odgovarati zadanom parametru. Iz tog razloga poziva se funkcija *adaptSoldiers* koja će osigurati da ukupan broj vojnika ostane nepromijenjen.

Funkcija `crossover1` stvara novi kromosom u kojemu je svaki gen jednak prosjeku gena na tim mjestima u kromosomima roditelja. Broj vojnika mora biti cijeli pa se prosjek zaokružuje na manji cijeli broj, a funkcija `adaptSoldiers` će riješiti problem nedostatka vojnika ako bude potrebno.



Slika 4.1.3.1. crossover1

Funkcija `crossover2` predstavlja programsko ostvarenje križanja s jednom točkom prekida. Nasumično se odabire broj n između 0 i 10. Broj vojnika na prvih n tvrđava se preuzima od prvog roditelja, a ostali od drugog roditelja. Ovim postupkom ukupan zbroj vojnika može se dosta razlikovati od zadanog parametra. Zbog toga funkcija `scale` jednoliko raspoređuje višak ili manjak vojnika ako je on veći od 10.

Slika 4.1.3.2. crossover2

Funkcija `crossover3` izvodi uniformno križanje. Za svaku tvrđavu nasumično se odredi roditelj od kojeg će broj vojnika biti preuzet. Iz istog razloga kao i u prethodnoj funkciji, poziva se funkcija `scale` i funkcija `adaptSoldiers`.

Slika 4.1.3.3. crossover3

4.1.4. Mutacija

Mutacija uzrokuje manje promjene na kromosomu uz određenu vjerojatnost. Ta vjerojatnost zadaje se kao parametar igre. Kroz proces mutacije će proći jedinka s novim kromosomom dobivenim križanjem roditelja i lošiji roditelj. Bolji roditelj nepromijenjen ide u sljedeću iteraciju. Mutacijom se ostvaruje lokalna pretraga i ona daje šansu da drugi roditelj u sljedećoj iteraciji postane bolji od prvoga. Također, mutacijom se održava genetska raznolikost u populaciji.

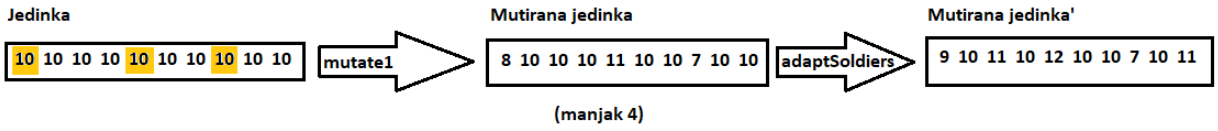
Kao što je bio slučaj i kod križanja, u ovom programskom rješenju koriste se tri različite funkcije mutacije. Funkcija se odredi nasumično i u nju se pošalju dvije spomenute jedinke.

```
public static void mutation(List<Player> players, Set<Player>
selectedPlayers, float mutationProbability) {

  Player p1 = (Player) selectedPlayers.toArray()[1];
  Player p2 = (Player) selectedPlayers.toArray()[2];

  String[] p1Strategy = p1.getStrategy().split(" ");
  String[] p2Strategy = p2.getStrategy().split(" ");

  Integer[] mutatedStrategy1 = new Integer[10];
  Integer[] mutatedStrategy2 = new Integer[10];

  int mutationType = (int)(Math.random()*3);
   switch(mutationType) {
     case(0): mutatedStrategy1 = mutate1(p1Strategy, mutationProbability);
              mutatedStrategy2 = mutate1(p2Strategy, mutationProbability);
              break;
     case(1): mutatedStrategy1 = mutate2(p1Strategy, mutationProbability);
              mutatedStrategy2 = mutate2(p2Strategy, mutationProbability);
              break;
     case(2): mutatedStrategy1 = mutate3(p1Strategy, mutationProbability);
              mutatedStrategy2 = mutate3(p2Strategy, mutationProbability);
              break;

   }

…
}
```

Kod 6. – mutacija

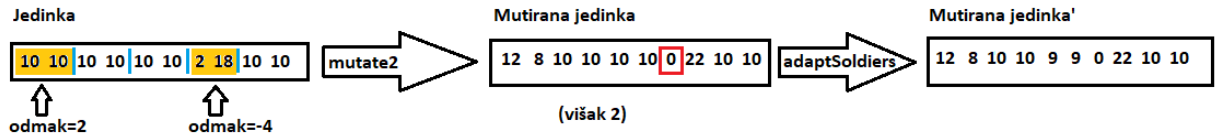
Funkcije *mutate1*, *mutate2* i *mutate3* vraćaju niz cijelih brojeva koji predstavlja mutirani kromosom. Nakon izvršavanja mutacije (u dijelu označenom sa „…“) kromosom jedinke poslane u mutaciju se briše i zamjenjuje se novim. Izvođenje svake od funkcija mutacije objašnjeno je u nastavku. Za svaku funkciju uvjet je li se vjerojatnost mutacije ostvarila ispituje se na sljedeći način: `if(Math.random() < mutationProbability)`.

Funkcija `mutate1` iterira po svim tvrđavama. Ako se dogodi mutacija, broj vojnika na toj tvrđavi promjeni se za nasumični broj između -5 i 5. Za ovu funkciju treba paziti da broj vojnika ne postane manji od 0 ili veći od ukupnog broja vojnika. Uz to, na kraju potrebno je dobiveni niz poslati u funkciju `adaptSoldiers` da bi se održao konstantan ukupni broj vojnika.



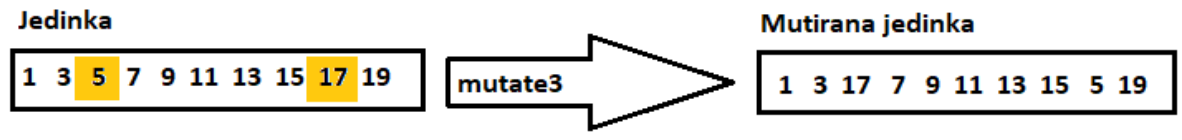
Slika 4.1.4.1. mutate1

Funkcija `mutate2` iterira po parovima tvrđava (tvrđava 2i i tvrđava 2i+1). Ako dođe do mutacije izračuna se odmak nasumično u rasponu -5 do 5. Prvoj tvrđavi se pribroji odmak, a drugoj oduzme. Također treba paziti na granice mogućeg broja vojnika na tvrđavi i zbog toga je potreban poziv `adaptSoldiers` na kraju.



Slika 4.1.4.2. mutate2

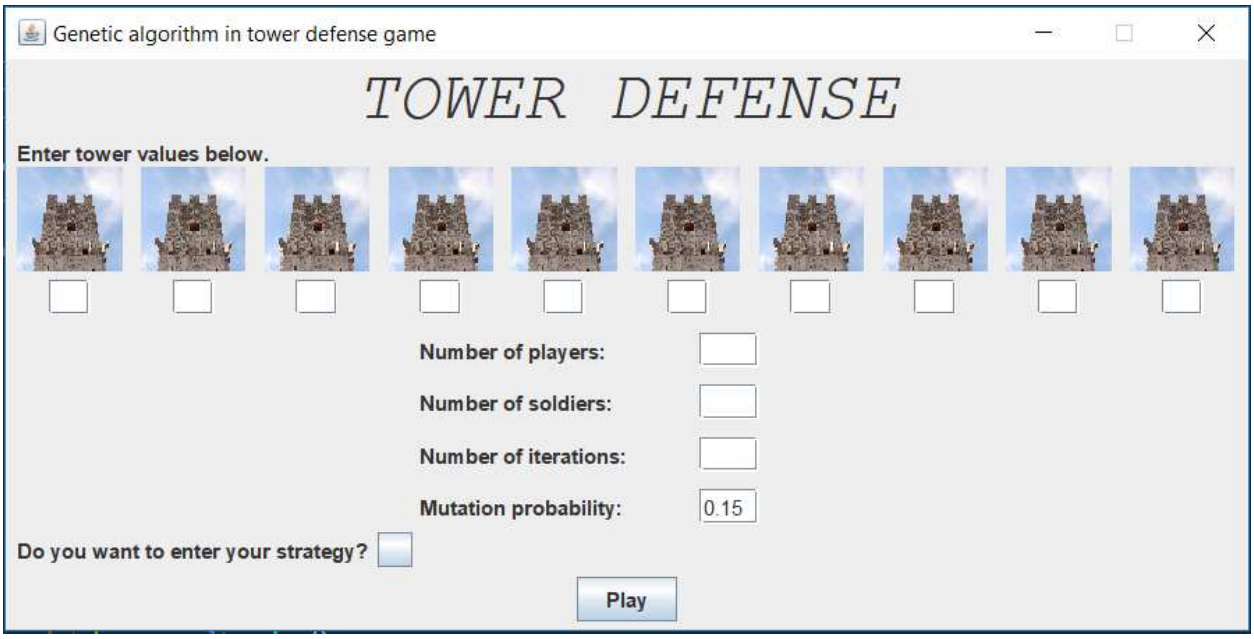
Funkcija `mutate3` je programsko ostvarenje jedne vrste miješajuće mutacije. Nasumično izabire dva gena unutar kromosoma i zamijeni im vrijednosti. U ovoj funkciji poziv korekcijske funkcije `adaptSoldiers` nije potreban jer ukupan zbroj vojnika ostaje stalan.



Slika 4.1.4.3. mutate3

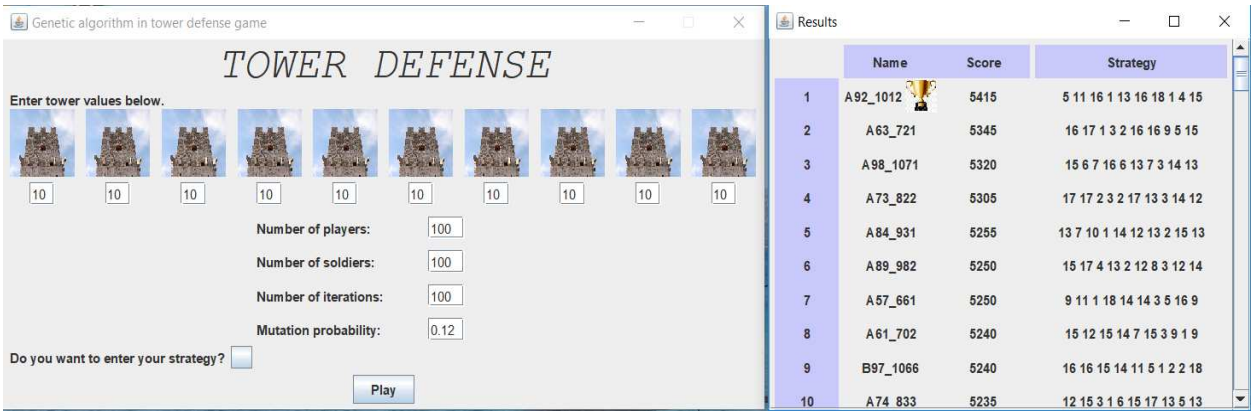
4.2. Programski sustav

Programsko sučelje za testiranje igre i genetskog algoritma izrađeno je koristeći Java Swing alat. Izgled početnog ekrana nakon pokretanja programa prikazan je na slici 4.2.1.



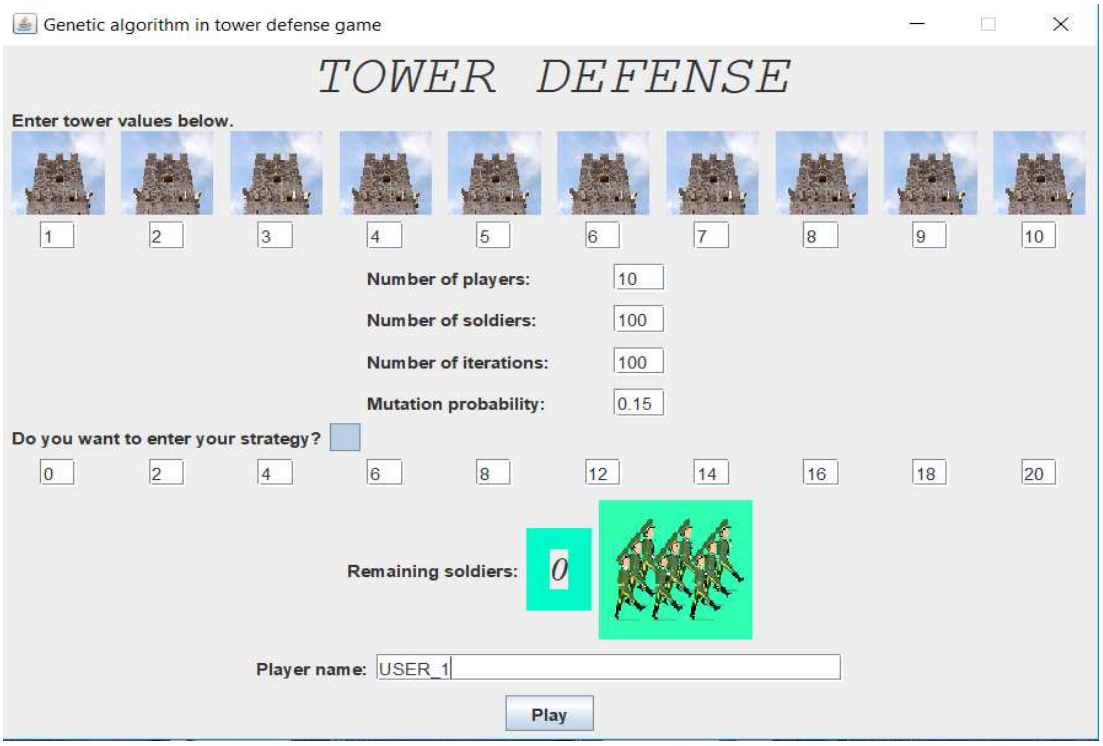
Slika 4.2.1. Početni ekran

U prazne kućice korisniku je omogućeno upisivanje parametara igre. Vrijednosti tvrđava upisuju se ispod sličica tvrđava. Nakon toga upisuje se broj igrača, ukupan broj vojnika za svakog igrača i broj iteracija genetskog algoritma. Vjerojatnost mutacije postavljena na zadanu vrijednost 0.15 može se proizvoljno mijenjati. Razlog postavljanja vrijednosti u to polje je da je korisniku jasno da se za separator koristi točka, a ne zarez. Pokretanje igre izvršava se gumbom Play. Primjer rezultata igre prikazan je na slici 4.2.2.

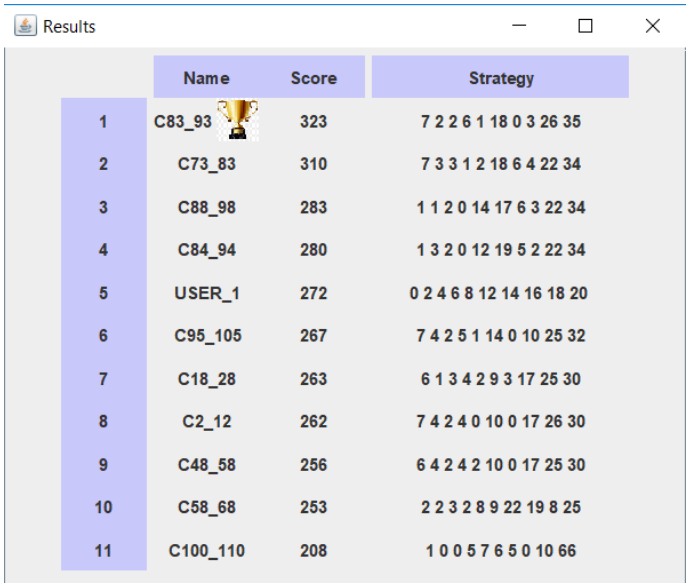


Slika 4.2.2. Primjer rezultata

Korisniku je također omogućen unos vlastite strategije u igru. Klikom na gumb pored teksta „Do you want to enter your strategy?“ otvara se prostor za unos korisničke strategije i korisničkog imena. Nakon provedbe genetskog algoritma, korisnik je ubačen u skup n igrača i igra se provodi još jedan put s n + 1 igračem. Zatim se prikazuju ukupni rezultati igre.



Slika 4.2.3. Polje za upis podataka korisnika



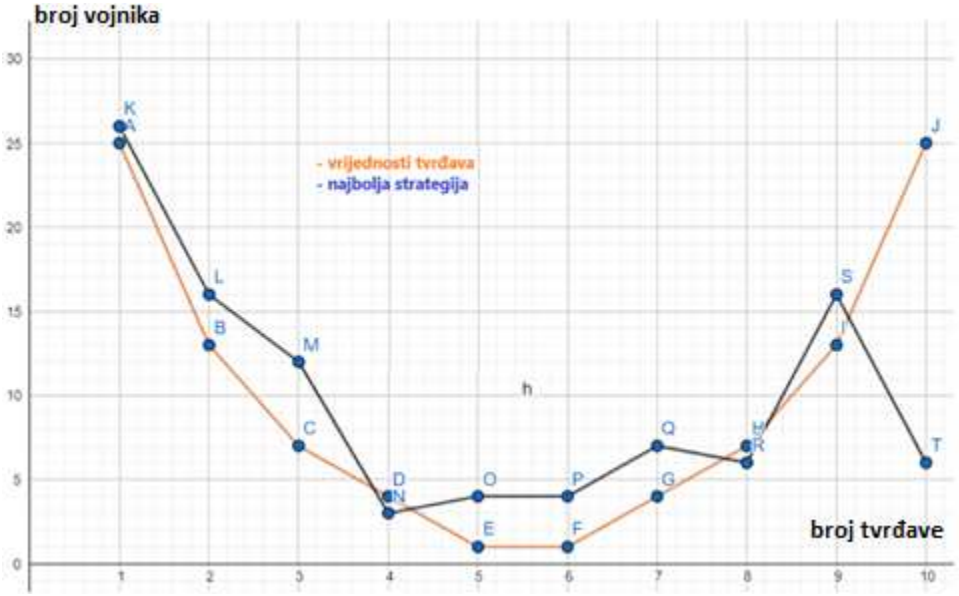
	Name	Score	Strategy
1	C83_93	323	7 2 2 6 1 18 0 3 26 35
2	C73_83	310	7 3 3 1 2 18 6 4 22 34
3	C88_98	283	1 1 2 0 14 17 6 3 22 34
4	C84_94	280	1 3 2 0 12 19 5 2 22 34
5	USER_1	272	0 2 4 6 8 12 14 16 18 20
6	C95_105	267	7 4 2 5 1 14 0 10 25 32
7	C18_28	263	6 1 3 4 2 9 3 17 25 30
8	C2_12	262	7 4 2 4 0 10 0 17 26 30
9	C48_58	256	6 4 2 4 2 10 0 17 25 30
10	C58_68	253	2 2 3 2 8 9 22 19 8 25
11	C100_110	208	1 0 0 5 7 6 5 0 10 66

Slika 4.2.4. Rezultati s korisnikom

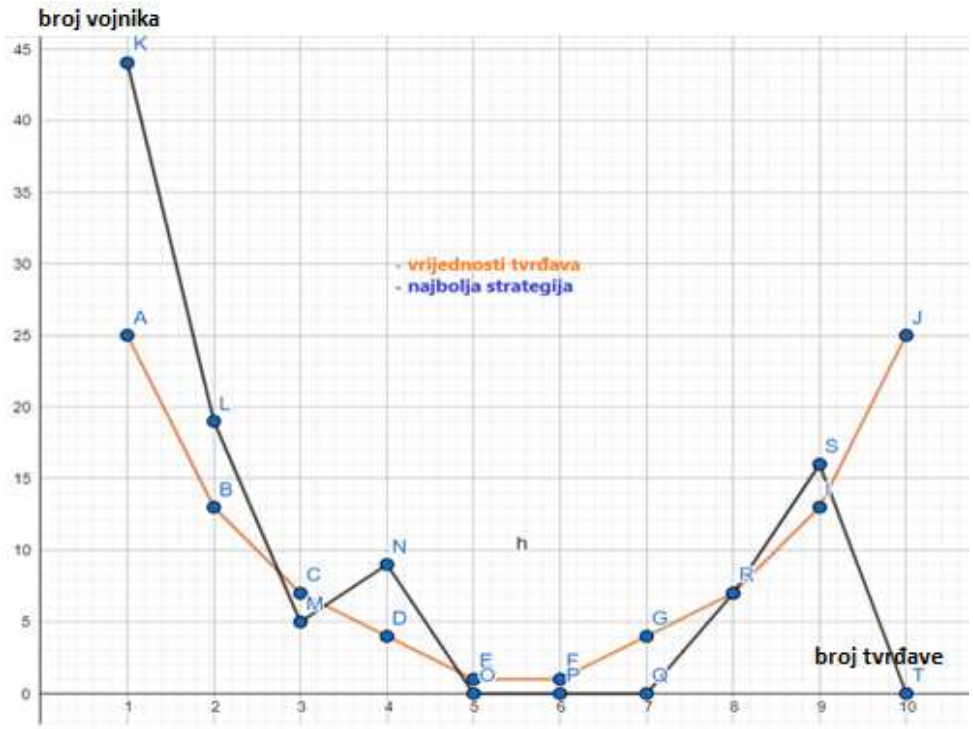
5. Eksperimentalni rezultati

Ovo poglavlje prikazuje rezultate genetskog algoritma pokrenutog s različitim parametrima. Iako se ukupan broj vojnika može mijenjati, ovdje se za svaki primjer uzima da je broj vojnika 100. Dobivena strategija se onda može interpretirati kao postotak vojnika koji bi se rasporedio na pojedinu tvrđavu. Također, pri usporedbi rezultata, vjerojatnost mutacije će biti postavljena na konstantnu vrijednosti 0.15 s očekivanjem pojave mutacije na jednoj do dvije tvrđave. Razlog ovome je što igra zahtijeva da suma unutar kromosoma bude konstantna. Zbog toga funkcija prilagodbe, `adaptSoldiers` na neki način izvodi mutaciju pridjeljivanjem ili oduzimanjem vojnika nasumično odabranim tvrđavama. Stoga se promatraju rezultati određenih vrijednosti tvrđava s različitim parametrima broja igrača i broja iteracija.

Najprije se promatraju rezultati za vrijednosti tvrđava 25 13 7 4 1 1 4 7 13 25. Očekivano je da će za veći broj iteracija jedinke biti bolje istrenirane i davati bolje rezultate nego za manji broj iteracija. To se promatra na primjeru igre u kojoj sudjeluju 30 igrača i mijenja se broj iteracija sa 100 na 1000. Za 100 iteracija algoritam daje najbolje rješenje 26 16 12 3 4 4 7 6 16 6 što je vrlo slično vrijednostima tvrđava kao što prikazuje slika 5.1. S istim parametrima, ali s 1000 iteracija najbolje rješenje je 44 19 5 9 0 0 0 7 16 0. To rješenje odskače od proporcionalnog rasporeda po vrijednosti tvrđava (slika 5.2.), ali se pokazuje da je bolje od onog rješenja sa 100 iteracija (slika 5.3.)

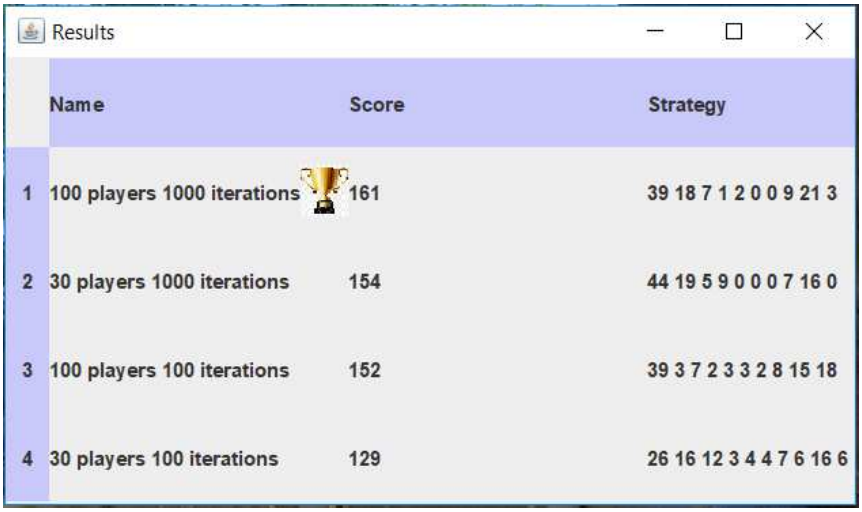


Slika 5.1. Rezultati sa 30 igrača i 100 iteracija



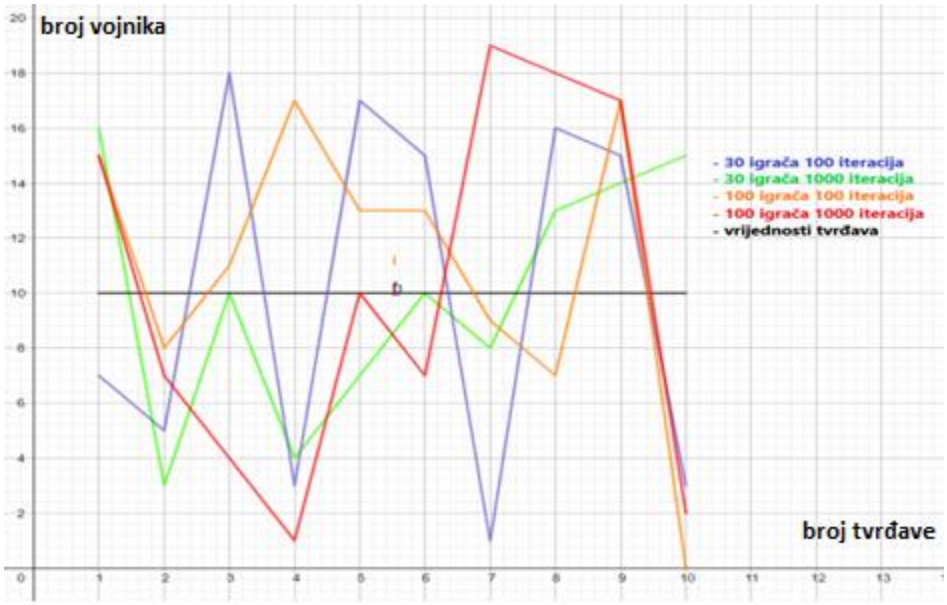
Slika 5.2. Rezultati sa 30 igrača i 1000 iteracija

Kada se populacija poveća s 30 na 100 igrača, također je očekivano da će algoritam davati bolja rješenja. Tako se za igru sa 100 igrača i 100 iteracija dobije rješenje 39 3 7 2 3 3 2 8 15 18, a za igru sa 100 igrača i 1000 iteracija 39 18 7 1 2 0 0 9 21 3. Očekivano (slika 5.3.) rezultat igre s najvećim brojem iteracija i najvećim brojem igrača pokazao se najboljim, a s manjim brojem igrača i iteracija najlošijim. Povećanje samo broja iteracija daje vrlo slične rezultate kao povećanje samo broja igrača (razlika 2 boda).



Slika 5.3. Rezultati za vrijednosti kula 25 13 7 4 1 1 4 7 13 25

Za drugi primjer promatraju se vrijednosti tvrđava 10 10 10 10 10 10 10 10 10 10. S obzirom na to da je vrijednost svake tvrđave jednaka, bitno je samo na koliko će kula neki igrač imati više vojnika od drugoga. Kao i u prethodnom primjeru, promatraju se rezultati za 30 i 100 igrača, te za broj iteracija 100 i 1000. Dobivena rješenja prikazana su na slici 5.4.



Slika 5.4. Rješenja za vrijednosti tvrđava 10 10 10 10 10 10 10 10 10 10

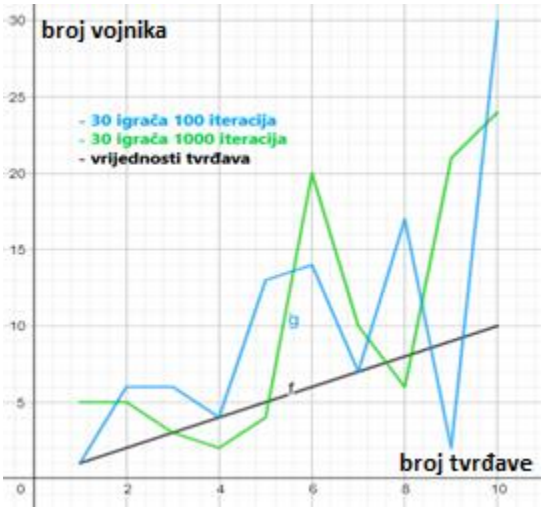
Vidi se da za ovaj slučaj vrijednosti tvrđava broj vojnika i iteracija ne igra toliko važnu ulogu. Pokazuje se da rješenja koja na pojedinim utvrdama žrtvuju više vojnika da bi na ostalima mogli imati više, bolje prolaze od rješenja koja imaju jednolik raspored vojnika.

Results

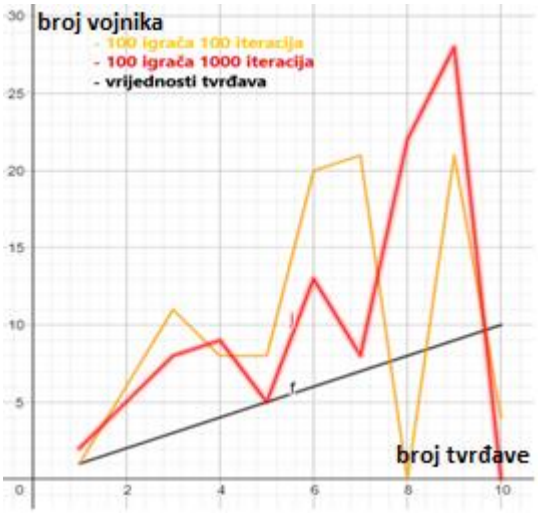
	Name	Score	Strategy
1	100 players 100 iterations	160	15 8 11 17 3 13 9 7 17 0
2	30 players 100 iterations	160	7 5 18 3 17 15 1 16 15 3
3	100 players 1000 iterations	150	15 7 4 1 10 7 19 18 17 2
4	30 players 1000 iterations	130	16 3 10 4 7 10 8 13 14 15

Slika 5.5. Rezultati dobivenih rješenja

Kao posljednji primjer promatraju se rezultati za linearno raspoređene vrijednosti tvrđava 1 2 3 4 5 6 7 8 9 10. Očekuje se da će se na tvrđave s većom vrijednosti rasporediti veći broj vojnika. Najbolja rješenja za određene parametre prikazana su na slikama 5.6. i 5.7.



Slika 5.6. Rezultati s 30 igrača



Slika 5.7. Rezultati sa 100 igrača

Vidi se da se rješenja drže raspoređivanja vojnika proporcionalno vrijednostima tvrđava uz neka odstupanja. U oba slučaja igre sa 100 igrača najbolje rješenje žrtvuje najvrjedniju tvrđavu da bi na ostale mogao rasporediti više vojnika. Tim potezom pobjeđuje strategije dobivene igrom 30 igrača koje najviše vojnika raspoređuju na tu tvrđavu.

Results

	Name	Score	Strategy
1	100 players 100 iterations	91	1 6 11 8 8 20 21 0 21 4
2	100 players 1000 iterations	84	2 5 8 9 5 13 8 22 28 0
3	30 players 100 iterations	79	1 6 6 4 13 14 7 17 2 30
4	30 players 1000 iterations	74	5 5 3 2 4 20 10 6 21 24

Slika 5.8. Rezultati za vrijednosti tvrđava 1 2 3 4 5 6 7 8 9 10

5.1. Osvrt na rezultate i praktični rad

Sukladno očekivanjima, izvođenje programa s većim brojem igrača i većim brojem iteracija uglavnom će davati bolja rješenja nego kad su ti brojevi manji. Međutim, iz priloženih rezultata može se vidjeti da to nije uvijek slučaj. Tako je na primjeru izvođenja programa sa svim vrijednostima tvrđava postavljenih na 10 rješenje dobiveno iz 100 iteracija bilo bolje od rješenja dobivenog iz 1000 iteracija.

Ova nepravilnost uzrokovana je samim načinom izvođenja igre. Najboljim rješenjem proglašava se ono rješenje koje ostvari najveći broj bodova u igri protiv ostalih rješenja iz iste populacije. To predstavlja specifičnost uporabe genetskog algoritma pri rješavanju ovog problema. U klasičnim primjenama genetskog algoritma jedinka je ocijenjena na temelju njezine izvedbe određenog problema i ta ocjena ne ovisi o drugim jedinkama. S druge strane, u ovoj igri jedinke su ocijenjene isključivo po tom kriteriju. Ne postoji apsolutni kriterij koliko je neko rješenje dobro, već se kvaliteta rješenja određuje relativno, u odnosu na druge jedinke.

Također, još jedan problem pri izvođenju ove igre je nepostojanje optimalnog rješenja. Neovisno o tome koliko je neko rješenje dobro u igri s drugima, uvijek će postojati neko drugo rješenje koje će biti bolje. Za svaki raspored vojnika A možemo stvoriti raspored vojnika B tako da s neke tvrđave manje vrijednosti premjestimo jednog vojnika na tvrđavu s većom vrijednosti. Na taj način rješenje B će pobijediti rješenja A u međusobnoj igri, ali to ne garantira da će B proći bolje od A kada igra veći broj igrača. Moguće je da A osvaja veći broj bodova od B u igri protiv ostalih igrača. Iz tog razloga, A će se proglasiti najboljim rješenjem s najvećim brojem bodova iako u međusobnoj igri protiv B izgubi.

Ova problematika igre dovodi do zaključka da će rješenje biti to bolje što veći broj igrača sudjeluje u igri. Iako je poznato da svako rješenje može biti pobijeđeno, rješenja stvorena s većim brojem igrača će dominirati nad većim skupom rješenja i time će imati veće šanse za pobjedu u ostalim igrama. Ovo potvrđuju eksperimentalni rezultati. U svakom primjeru s različitim vrijednostima tvrđava, u igri najboljih rješenja s različitim parametrima, najboljim se pokazalo ono koje je imalo veći broj igrača (slike 5.3, 5.5 i 5.8).

6. Primjene u praksi

Igra raspoređivanja vojnika na utvrde može se u stvarnom svijetu interpretirati kao raspoređivanje ograničenih resursa na komponente područja u kojem želimo biti što bolji. Osim očite poveznice ove igre s ratom i raspoređivanjem trupa na bojišta, ova igra pronalazi primjenu i u mnogim drugim područjima današnjeg svijeta.

Izbori u SAD-u se ne pobjeđuju brojem glasova birača, već brojem osvojenih saveznih zemalja. Za osvajanje izbora, svaki kandidat želi osvojiti što veći broj zemalja. Resursi, koje u ovom slučaju predstavlja novac, kandidati raspoređuju na „tvrđave“, odnosno troše u pojedinim zemljama na reklamiranje i promoviranje svog sadržaja. Kandidat koji najbolje rasporedi svoje resurse ima najveću vjerojatnost da će osvojiti izbore.

Drugi primjer je odabir najboljeg kandidata za zapošljavanje. Svaki kandidat ima određene vještine i znanja u raznim poljima važnima za taj posao. Tako se na primjer ocjenjuju obrazovanje, vještine komuniciranja, intervju, radno iskustvo, tehničke vještine itd. Kandidati su ocijenjeni po ovim elementima koji predstavljaju tvrđave igre i svaki element nosi različitu težinu ovisno o poslu za koji se kandidat prijavljuje. Izbor najboljeg kandidata ovisi o najboljem rasporedu tih vještina koje odgovaraju traženom poslu.

Još jedan primjer je primjena ove igre na sport. U svakom sportu postoji više elemenata na temelju kojih se ocjenjuje koliko je igrač dobar. Tako na primjer u tenisu postoje servis, udarci s osnovne linije, voleji, *slice*, *spin*, *smash*, brzina, snaga, izdržljivost itd. Svaki od tih elemenata su važni za sport, ali nisu svi jednako. Postoji ograničeno vrijeme do početka nekog turnira. Svaki igrač raspoređuje to vrijeme na usavršavanje pojedinih komponenti. S obzirom na ograničeno vrijeme, igrač mora odrediti kako najpametnije rasporediti to vrijeme da bi na turniru imao što veće šanse za pobjedu.

7. Zaključak

Ugledanje na prirodu kao inspiraciju za rješavanje složenih problema u računarstvu omogućilo je pronalazak rješenja do kojih ne bi znali doći koristeći klasične tehnike programiranja. U ovakvim problemima u kojima je broj mogućih rješenja često veći od broja atoma u svemiru, algoritmi evolucijskog računanja pronalaze optimalno rješenje ili rješenje blizu optimalnog u razumnom vremenu. Gledajući izvođenje ovakvih algoritama, vrlo često ne možemo zaključiti kako je algoritam došao do nekog rješenja.

Glavna značajka ovakvih algoritama, kao i drugih algoritama umjetne inteligencije, je ta da programer ne zadaje korake algoritmu kako doći do rješenja, nego samo određena ograničenja kojih se algoritam mora pridržavati. Algoritam sam uči na temelju ocjena i kazni koje zadaje programer za pojedina rješenja. Tako metodom pokušaja i pogrešaka dolazi do sve boljih rješenja.

Konkretno, genetski algoritam simulira evoluciju populacije živih bića. Teško je predvidjeti kako će ta populacija izgledati za tisuću, sto tisuća ili milijun godina. Evolucija je pokazala da temeljem selekcije, križanja i mutacije jedinke populacije postaju sve bolje prilagođene na okruženje u kojemu žive. Ova ideja inspirirala je stvaranje računalne simulacije koja će kopirati procese iz prirode. Razlika je u tome što zbog brzine izvođenja naredbi u računalu, vrijeme za stvaranje jedne generacije jedinki mjeri se u mikrosekundama. Tako je moguće simulirati veliki broj generacija u razumno kratkom vremenu i stvoriti jedinku koja predstavlja optimalno rješenje problema.

Fleksibilnost evolucijskih algoritama u pronalaženju optimalnog rješenja daje ovim tehnikama prednost nad klasičnim procedurama numeričke optimizacije. Također, još jedna važna značajka je konceptualna jednostavnost algoritma. Svaki algoritam se može sažeti u dva osnovna koraka: inicijalizacija, koja može biti nasumično prikupljanje mogućih rješenja, nakon koje slijedi iterativna varijacija i selekcija. Ovaj proces pridjeljuje brojčanu vrijednost svakom od mogućih rješenja čime možemo razlikovati bolje rješenje od lošijeg. Ovi algoritmi se mogu primijeniti na praktički svaki problem koji može biti formuliran kao problem optimizacije. Iz navedenih razloga možemo očekivati sve češću uporabu ovih algoritama u problemima u kojima se čini da je vrlo teško doći do optimalnog rješenja.

8. Literatura

[1] A.E.Eiben: Introduction to evolutionary computing; 2003.

[2] David B. Fogel: The Advantages of Evolutionary Computation; 1997.

[3] https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html

[4] Devrim Balköse: Chemical Science and Engineering Technology; 2019.

[5] Vikhar, P. A. (2016). "Evolutionary algorithms: A critical review and its future prospects"

[6] https://www.coursera.org/lecture/model-thinking

[7]https://www.valluriorg.com/blog/artificial-intelligence-and-its-applications/

[8]https://www.theverge.com/2019/1/28/18197520/ai-artificial-intelligence-machine-learning-computational-science

[9]https://www.forbes.com/sites/bernardmarr/2018/12/31/the-most-amazing-artificial-intelligence-milestones-so-far

Optimiranje raspoređivanja vojnika u igri obrane tvrđava

Sažetak

Genetski algoritam je algoritam evolucijskog računanja koji simulira procese selekcije, križanja i mutacije pri pronalasku najboljeg rješenja. Programski je ostvaren za problem raspoređivanja vojnika na tvrđave s različitim vrijednostima. Cilj je pronaći strategiju koja će se pokazati najboljom u igri s drugim igračima. Najbolja strategija je ona koja na kraju igre ima najveći broj bodova. Ovaj algoritam koristi 2 operatora selekcije, 3 operatora križanja i 3 operatora mutacije. Programski sustav omogućava korisniku zadavanje parametara igre i unos vlastite strategije. Eksperimenti su izvedeni s različitim parametrima i prikazani su rezultati.

Ključne riječi: genetski algoritam, selekcija, križanje, mutacija, jedinka, kromosom, vojnici, tvrđave

Optimizing the deployment of soldiers in the tower defense game

Abstract

Genetic algorithm is an evolutionary computation algorithm that simulates processes of selection, crossover and mutation when searching for the best solution. It is implemented for problem of deploying soldiers to towers with different values. The goal is to find a strategy that turns out to be the best against other players. The best strategy is the one which has the most points in the end of the game. This algorithm uses 2 selections operators, 3 crossover operators and 3 mutation operators. Software system enables user to enter parameters of the game and his own strategy. Experiments have been made with various parameters and results are shown.

Key words: genetic algorithm, selection, crossover, mutation, unit, chromosome, soldiers, towers