SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5347

Rješavanje problema obilaska skakača svih polja na šahovskoj ploči korištenjem genetskih algoritama

Kristijan Palić

Zagreb, lipanj 2018.

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

Zagreb, 9. ožujka 2018.

ZAVRŠNI ZADATAK br. 5347

Pristupnik: **Kristijan Palić (0036492988)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Rješavanje problema obilaska skakača svih polja na šahovskoj ploči korištenjem genetskih algoritama**

Opis zadatka:

Opisati i definirati problem obilaska skakača svih polja na šahovskoj ploči dimenzija NxN polja. Opisati genetski algoritam koji je prilagođen rješavanju problema obilaska skakača. Posebnu pažnju posvetiti genetskim operatorima koji ne stvaraju nemoguća rješenja. Ostvariti programski sustav s odgovarajućim grafičkim sučeljem za zadavanje i rješavanje problema obilaska skakača te za prikaz dobivenih rješenja. Usporediti dobivene rezultate s poznatim optimalnim rješenjima iz literature. Uz rad priložiti izvorne tekstove programa i citirati korištenu literaturu.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 15. lipnja 2018.

Mentor:
Prof. dr. sc. Marin Golub

Predsjednik odbora za završni rad modula:
Prof. dr. sc. Siniša Srbljić

Djelovođa:
Doc. dr. sc. Tomislav Hrkać

SADRŽAJ

1. Uvod

Ubrzanim razvojem računala omogućava nam se rješavanje problema koji su do nedavno bili samo teorijski rješivi, ili rješivi za neke manje, jednostavnije primjere. No koliko god naša računala danas bila snažna, postoje problemi koje nije moguće riješiti samo čistim iteriranjem, odnosno primjenom grube sile (engl. *brute force*). Tu do izražaja dolaze razni algoritmi optimizacije, od kojih je jedan i genetski algoritam, inspiriran evolucijom. U ovom radu će biti predstavljen problem obilaska skakača svih polja na šahovskoj ploči te će se genetskim algoritmom pokušati doći do konačnog rješenja. Primarni cilj ipak nije postavljen na konačno rješenje, već na sam postupak i dolazak do rješenja te ujedno upoznavanje s optimizacijskim algoritmima.

2. Skakačev put

Skakačev put (engl. *Knight's tour*) je putanja skakača po šahovskoj ploči tako da se svaki kvadrat posjeti samo jednom. Problem skakačevog puta (engl. *Knight's tour problem*) je matematički problem kojim se pokušava naći barem jedno moguće rješenje. Nije poznato tko je prvi postavio problem, ali on vuče svoje korijene još iz stare indijske civilizacije, samo naravno, ne u obliku u kakvom ga danas poznajemo. Među najpoznatijim matematičarima koji je posvetio svoje vrijeme rješavanju problema te javno objavljivao radove bio je Leonhard Euler[1]. Prvo poznato rješenje u obliku algoritma bilo je ono H.C. von Warnsdorfa[2], koji je 1823. godine predstavio rad na 68 stranica u kojemu je opisao algoritam. Algoritam je nazvan Warnsdorfovo pravilo, no više o njemu u poglavlju 4.2.

2.1. Matematička pozadina

Rješavanje problema obilaska skakača svih polja na šahovskoj ploči je matematički prilično složen račun. Predstavlja samo općenitiji problem pronalaska Hamiltonovog ciklusa koji spada u kategoriju NP-potpunih problema. Iako je problem star više od 1000 godina, i unatoč svoj snazi računala koju danas poznajemo, točan broj svih mogućih rješenja za ploču dimenzija 8×8 nije poznat (tablica 2.1). Valja napomenuti kako je dokazano koliko je zatvorenih puteva [5], ali onih koji ne završavaju u istoj točki u kojoj su počeli se samo procjenjuje da ima 1.305×10^{35}, no nije dokazano.

Allen J. Schwenk[3] je 1991. predstavio rad [6] u kojemu je dokazao kako je skakačev put za ploču veličine $m \times n$, gdje je $m \leq n$, uvijek moguć (pritom se misli na pronalazak zatvorenog puta, ne općenito pronalaženje rješenja), osim ako su zadovoljeni sljedeći uvjeti:

[1] Leonhard Euler – švicarski matematičar, fizičar, astronom, logičar
[2] H.C. von Warnsdorfa – njemački matematičar
[3] Allen J. Schwenk – američki matematičar

1. m i n su neparni
2. m = 1,2 ili 4
3. m = 3 i n = 4,6 ili 8

Tablica 2.1: Rješenja problema za ploče različitih dimenzija

Dimenzije ploče	Broj rješenja
3×3	0
4×4	0
5×5	1728
6×6	6,637,920
7×7	165,575,218,320
8×8	13,267,364,410,532 (samo zatvorenih)

Često se u literaturi može pronaći brojka dvostruko veća od ove prikazane u tablici za ploču dimenzija 8×8. To je zato što se u tim slučajevima broje rute u oba smjera - od početne do krajnje točke i od krajnje do početne što je identično rješenje pa ga nema smisla brojati dva puta.

Magični kvadrat

Jedno od prvih ponuđenih rješenja u sebi skriva neke nevjerojatne podatke. Radi se naime o magičnom kvadratu kojeg tvore brojevi koraka pronađenog rješenja. Svaki stupac i svaki redak se zbrajaju u istu sumu - 260. Nadalje, zbroj brojeva svakog kvadranta daje sumu 520, a svaki redak i svaki stupac svakog kvadranta daje sumu 130. Zaista zvuči nevjerojatno, a jedino što nedostaje da rješenje tvori potpuni magični kvadrat je suma dijagonala. Ona bi u pravom magičnom kvadratu trebala biti jednaka za obje dijagonale. Pokazalo se kasnije kako postoje rješenja koja zadovoljavaju i taj uvjet.

Zanimljivost Na jednom matematičkom natjecanju dano je pitanje može li se pronaći skakačev put na standardnoj šahovskoj ploči, oduzmu li se:

1. Dva polja koja cijelom stranom leže jedno uz drugo
2. Dva međusobno suprotna kuta ploče

Odgovori su da i ne, no treba ih malo argumentirati. Oduzme li se šahovskoj ploči dva susjedna polja, uz uvjet da se polja cijelom jednom dužinom dodiruju, s ploče će se

1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

Slika 2.1: Skakačev put i magični kvadrat

uzeti jedno crno i jedno bijelo polje, a maknu li se međusobno suprotni kutevi, to će biti polja iste boje. Sada treba primijetiti kako se kreće skakač - s polja jedne boje na polje suprotne boje, i nikada ne drugačije. Tako da, makne li se jednak broj bijelih i crnih polja s ploče skakač će pronaći svoj put, no maknu li se polja jednake boje, tada je i matematički nemoguće da skakač sva polja na ploči obiđe samo jednom.

2.2. Prijašnji radovi

Ako se malo pregleda po internetu, može se pronaći vrlo širok spektar radova na varijaciju ove teme, od pristupa rješavanju uz pomoć optimizacijskih algoritama do rješavanjem heuristikom ili pretraživanjem u dubinu. Problem je prilično zanimljiv i može mu se pristupiti na jako puno načina.

Kao referentni rad poslužio je rad nekoliko profesora s američkih sveučilišta[1]. U tom su radu problemu pristupili prvo korištenjem samo genetskih algoritama, a onda svakim novim pokušajem ubacivanjem nekog novog mehanizam u algoritam kako bi se dobili bolji rezultati. Više o samom algoritmu te kako on radi i usporedbu s ponuđenim rješenjem u ovome radu može se pronaći u poglavlju 5.

Još jedan zanimljiv rad koji će također biti ukratko opisan u poglavlju 5, a koji rje-

šavanju problema pristupa optimizacijskim algoritmom kolonije mrava, je rad dvojice profesora s britanskih sveučilišta[4].

2.3. Pristupi rješavanju

Kako će se opširno opisati pristup rješavanju ovog problema genetskim algoritmom, valjalo bi napomenuti još neke pristupe koji su više tipični te ne spadaju u domenu optimizacijske obitelji. Jedan od njih je primjenom klasične grube sile (engl. *brute-force*) gdje se ispituju sve mogućnosti dok god se ne dođe do rješenja. Iako taj pristup daje rezultate za manje dimenzije (primjerice 5×5), kod malo većih ploča taj bi proces mogao trajati godinama. Primjerice već se za ploču dimenzija 7×7 na rješenje mora čekati nekoliko sati, a za veće dimenzije vrijeme čekanja eksponencijalno raste.

Kada ploča postane prevelika za jednostavnu primjenu grube sile može se pristupiti problemu na sljedeći način - razbijanjem ploče na manje jednake dijelove i za svaki dio pronaći put, pazeći pritom da je moguće iz zadnjeg koraka prijeći s trenutnog dijela ploče na novi dio. Iako zanimljiv, ovakav pristup ima svojih mana. Jedan od njih je taj što jako puno mogućih rješenja propada zbog činjenice da se mora prelaziti s ploče na ploču (pritom se misli na dijelove ploče) te će se zbog toga teže pronalaziti rješenje.

Postoje i razni drugi algoritmi koji vrlo uspješno rješavaju problem, ali su vremenski ili memorijski jako skupi te kao takvi nisu pogodni za računanje s velikim pločama. Zato se problemu u ovom radu pristupa pomoću jednog od najpoznatijih optimizacijskih algoritama - genetskim algoritmom.

3. Genetski algoritam

3.1. Optimizacija

Što zapravo znači optimizirati neki postupak? Što je to optimalno rješenje? U nastavku je dano par primjera iz stvarnog života kako bi se dobio dojam o pojmu optimizacije..

Primjer 1. Neka je građevinar dobio zadatak premjestiti kamen iz zgrade iz prizemlja na drugi kat, a cigle s drugog kata u prizemlje. Kako će to izvesti? Prvi način je da jedan po jedan kamen prebaci iz prizemlja, zatim jednu po jednu ciglu prebaci s drugog kata, no jasno je da to nikako nije optimalno rješenje jer radi "dupli posao". Bolje rješenje bi bilo da prebacuje kamen do drugog kata, a na povratku u prizemlje uzme ciglu i tako skraćuje posao za duplo. No, višestruko bi ga mogao skratiti ako sa sobom nosi maksimalnu količinu materijala koju može ponijeti.

Naravno da je prethodni primjer poprilično banalan, ali vrlo jasno pokazuje kako je optimizacija vrlo primjenjiva i zastupljena, čak i kada toga nismo svjesni. Slijedi malo kompleksniji primjer koji pokazuje da ipak nisu stvari uvijek tako jednostavne.

Primjer 2. Skupina mladih turista želi posjetiti najljepše gradove Europe kroz cijelo ljeto. Naravno, popis je dugačak, no oni žele potrošiti što je moguće manje novaca na putovanje, ali isto tako ne putovati rutama koje će im oduzimati najviše vremena. Takav slučaj će zapravo vrlo vjerojatno imati više od jednog prihvatljivog rješenja. Teško je moguće da postoji put koji će trajati najmanje sati i koji će koštati najmanje novaca. Vjerojatnije je da će postojati nekoliko puteva koji bi bili prihvatljivi. Npr. neka **put1** ima cijenu $c1$, a duljinu $d1$. Iste varijable neka imaju i **put2** te **put3**. Neka je:

- $c1 = 100, d1 = 200$
- $c2 = 150, d2 = 150$
- $c3 = 200, d3 = 100$

Koji bi put grupa trebala izabrati? Ovdje se može primijetiti kako postoji više kriterija po kojima se bira optimalno rješenje te nije moguće na jednostavan način odabrati

kojim putem krenuti.

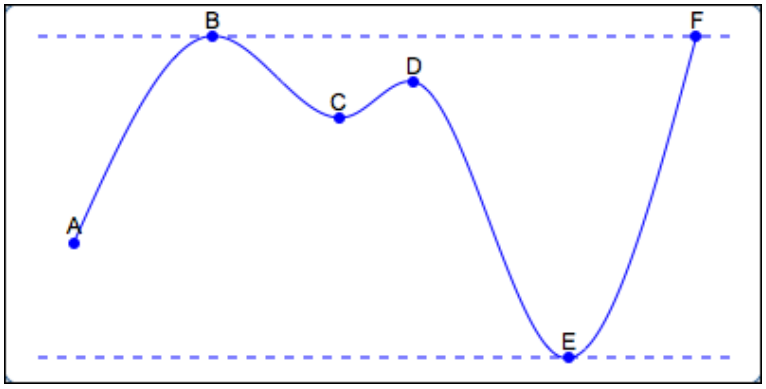
Prethodni primjeri pokazuju kako se optimizacijski problemi mogu podijeliti u dvije velike skupine, **problemi jednokriterijske optimizacije** i **problemi višekriterijske optimizacije**. Postoje i razlike između optimizacijskih problema istih skupina, ali gruba podjela je na ova dva tipa.

3.1.1. Problemi jednokriterijske optimizacije

Ovaj tip problema vrlo se jednostavno može opisati - problem koji jednostavnom evaluacijom daje prikaz kvalitete rješenja. Primjer 1 je tip ovakvog problema. Neka se evaluacijom dođe do broja sati potrebnih da radnik obavi posao, dolazi se do vrlo dobrog prikaza kvalitete rješenja (uzimajući u obzir kako su svi radnici isti, nitko ne radi brže, niti sporije, svi mogu nositi istu količinu materijala u jednoj turi, itd.). Sličan primjer bi bio pronaći vrijednost varijable x za koju neka funkcija $f(x)$ daje maksimalnu vrijednost na nekoj domeni $[x1, x2]$.

Globalni optimum je maksimalna vrijednost koju neka funkcija može poprimiti. Rješenje problema jednokriterijske optimizacije je prihvatljivo ako i samo ako se evaluacijom rješenja postigne globalni maksimum funkcije.

Lokalni optimum je maksimalna vrijednost koju neka funkcija može poprimiti na prostoru δ. Takva rješenja u problemima jednokriterijske optimizacije najčešće nisu od prevelikog značaja, odnosno nisu prihvatljiva. Postoje neke situacije kada to jesu, ali uglavnom se koriste mehanizmi koji bi spriječili algoritam da zapne u lokalnom maksimumu (poveća se broj koraka pretraživanja prostora, razni oblici selekcije, povećana mutacija kod genetskog algoritma, itd.). Kako uopće zaglaviti u lokalnom maksimumu? Pa prilično jednostavno, ali opet dosta neintuitivno. Tako što će se nedeterministički birati samo najbolja rješenja. Nepisano je pravilo uzeti rješenje koje je trenutno najbolje te njega poboljšavati, no postoji velika mogućnost da nije izabrano rješenje koje je krenulo putem globalnog maksimuma te tako konačno rješenje može zaglaviti u lokalnom maksimumu.



Slika 3.1: Globalni i lokalni maksimum

Na slici se jako dobro može vidjeti kada bi nastao problem. Ako se uđe u područje između točaka B i D te se nastavi poboljšavati samo to rješenje, ono će zapeti u lokalnom minimumu, bez mogućnosti dolaska do točke E. No ako se uz rješenje između B i D uključi neko između D i F, s vremenom će se pokazati kako je to rješenje kudikamo bolje, odnosno točnije. Tako da treba posebnu pozornost obratiti na biranje rješenja koje će se poboljšavati.

3.1.2. Problemi višekriterijske optimizacije

Kod višekriterijske optimizacije problem je uspoređivanje rješenja. Ako je jedno rješenje $r1 \rightarrow (100, 200)$, drugo $r2 \rightarrow (200, 300)$, vrlo je jednostavno odabrati bolje, no ako su to $r1 \rightarrow (100, 200)$ i $r2 \rightarrow (150, 150)$, dolazak do konačne odluke više nije trivijalan. Danas postoje algoritmi i načini kako se ti problemi rješavaju, no oni nisu primarni cilj ovog rada te se dalje neće obrađivati.

3.1.3. Pretraživanje

Da bi metaheuristički algoritmi uspjeli doći do nekog rješenja, potrebno je proći cijeli prostor pretraživanja. Najbolji način da se to napravi je prva rješenja generirati u potpunosti nasumično, čime se povećava mogućnost pokrivanja kompletnog prostora. Zatim svako rješenje na neki način poboljšavati, bilo međusobnim križanjem ili alternacijom trenutnog rješenja. Pretraživanje se dijeli na dvije faze:

1. **Gruba pretraga** Primarni cilj je pokriti što je veći mogući prostor, kako bi se smanjila mogućnost zapinjanja u lokalnom maksimumu

2. **Fina pretraga** Kada se pronađe obećavajući δ prostor, poboljšava ga se i dolazi do globalnog maksimuma

Naravno da nije moguće uvijek dobiti rješenje koje je očekivano, optimizacijski algoritmi ne rade tako, ali je moguće maksimalno minimizirati loša rješenja.

3.2. Prikaz jedinke

Podatci koji opisuju jednu jedinku zapisani su u kromosomu. Ti kromosomi dalje mutacijom ili međusobnim križanjem stvaraju nove kromosome, odnosno nova rješenja. Uobičajeno se uzima prikaz pomoću prirodnog binarnog koda zbog jednostavnosti i učinkovitosti. Naime, vrlo veliku ulogu u djelotvornosti genetskog algoritma ima prikaz jedinke. Ako je on prekompleksan, usložnjavaju se genetski operatori te samim time i dolazak do rješenja. S druge strane, ako je prikaz jedinke nepotpun na način da ne sadrži sve informacije koje jedinka mora sadržavati da bi se uspješno došlo do rješenja, tada takav prikaz nije prihvatljiv. Danas se može vrlo lako pronaći specijalizirana literatura koja govori samo o prikazu jedinki kod genetskih algoritama (ili optimizacijskih algoritama općenito) te njihovoj važnosti.

3.2.1. Prikaz binarnim kodom

Prikaz rješenja uz pomoć binarnog koda je vrlo često korišten način prikaza jedinke. Prvenstveno zbog svoje jednostavnosti, takav način zapisa omogućava vrlo malu memorijsku težinu te definira vrlo jednostavne genetske operatore. Ali nije sve tako sjajno. Budući da je binarni kod uglavnom nerazumljiv čovjeku, javlja se potreba za dekodiranjem rješenja kako bi ono postalo razumljivo.[8]

3.2.2. Prikaz poljem

Prikaz poljem (bitova, cijelih brojeva, decimalnih brojeva, itd.) je prikaz koji se najčešće koristi kada se kao rješenje traži nekakav put. Primjerice *Problem trgovačkog putnika* ili *Rješavanje problema obilaska skakača svih polja na šahovskoj ploči* su idealni primjeri za korištenje takvog tipa podataka za prikaz rješenja. Tada će svaki član polja predstavljati jedan korak u rješenju. Primjerice neka je to:

0000–0111–1001–0011–0101–0110–1011...

0–7–9–3–5–6–11...

Oba rješenja predstavljaju polja, samo su različitih podataka, prvo su bitovi, drugo su cijeli brojevi. Neka su to rješenja *Problema trgovačkog putnika*, tada se rješenje čita na sljedeći način - putnik će prvo posjetiti 0. grad, zatim 7. grad, pa 9., itd.[8]

3.2.3. Prikaz složenijim strukturama podataka

Nerijetko je potrebno imati više od jedne informacije u genetskom kodu jedinke, te je prikaz takvog rješenja nekim od jednostavnih oblika teško izvediv. Moglo bi se to riješiti tako da se kodira u binarni kod, ali je onda upitna učinkovitost prilikom dekodiranja i izvedbi genetskih operatora - uglavnom nije preporučljivo. No prikaz složenijim strukturama podatak sa sobom nosi dodatne probleme. Potrebno je definirati genetske operatore primjenjive na takvo rješenje te njihovu izvedbu, što nije uvijek najjednostavnija stvar za načiniti. Preporučljivo je zato svesti složenije strukture na samo malo kompleksnije prikaze nekih jednostavnijih oblika ranije prikazanih. Tako će definiranje operatora i njihova izvedba biti samo neznatno složenija nego kod jednostavnijih prikaza.[8]

3.3. Selekcija

Selekcija je ključan element pronalaska rješenja budući da se u selekcijskom mehanizmu krije odluka tko će sudjelovati u generiranju novih rješenja. Već je spomenuto kako je vrlo bitno ne birati isključivo najbolja rješenja, no bolja rješenja svakako trebaju imati određenu prednost. U nastavku će se obraditi nekoliko najčešće korištenih mehanizama selekcije.[2][3][8]

3.3.1. Proporcionalne selekcije

Jednostavna proporcionalna selekcija

Jednostavna proporcionalna selekcija (engl. *Roulette-wheel selection*) će svakoj jedinki rješenja dati određenu šansu da bude izabrana s obzirom na evaluaciju njene funkcije dobrote. Tako će najbolje jedinke imati veću šansu da sudjeluju u kreiranju novog genetskog materijala, ali one lošije također neće biti zanemarene.
Primjer Ako postoji 5 jedinki te vrijednosti njihove funkcije dobrote:

1. $10 \rightarrow 50\%$
2. $5 \rightarrow 25\%$
3. $3 \rightarrow 15\%$
4. $1 \rightarrow 5\%$
5. $1 \rightarrow 5\%$

Lijevo u primjeru je vrijednost funkcije dobrote, a desno je vjerojatnost da će baš ta jedinka biti izabrana kao nositelj genetskog materijala. Može se primijetiti kako je moguće da neke jedinke nemaju šansu biti zabrane (u slučaju da im je vrijednost funkcije dobrote jednaka 0), no to je u redu jer takva rješenja vrlo rijetko vode nečemu smislenom.

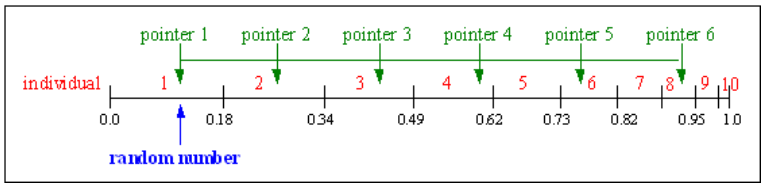
Stohastička univerzalna selekcija

Stohastička univerzalna selekcija (engl. *Stochastic universal sampling*) je vrlo slična jednostavnoj proporcionalnoj selekciji. Jedinke su raspoređene na vektoru od 0 do 1, gdje su one s najvećom dobrotom na početku i zauzimaju najviše mjesta (0 – 0.5, ako se gleda gornji primjer i prva jedinka). Zatim se uzima zadan broj N koji predstavlja koliko se jedinki treba izabrati. Tada se primjenjuje sljedeći postupak:

$$x = 1/N$$

Nasumično se odabere jedan broj iz $[0 - x]$ i označi sa poc

N brojeva se bira na način: $iduci = poc + x$, $poc = iduci$



Slika 3.2: Stohastička univerzalna selekcija

3.3.2. Rangirajuće selekcije

Selekcija najboljih

U ovu skupinu spadaju dvije selekcije, (μ, λ)-selekcija i $(\mu+\lambda)$-selekcija, a zajedničko im je to što smanjuju veličinu populacije.

Kod (μ, λ)-selekcije se slučajno bira μ roditelja i njihovim križanjem nastaje λ djece, ali uz uvjet $\lambda \geq \mu$. Tada se μ najbolje djece odabire kao nositelje genetskog materijala za sljedeću generaciju.

Kod $(\mu + \lambda)$-selekcije se slučajno bira μ roditelja i njihovim križanjem nastaje λ djece. Postupak se ili ponavlja dok se ne popuni predodređen broj za sljedeću generaciju ili se samo kopira trenutni izbor onoliko puta koliko je potrebno.[3]

k-turnirska selekcija

k-turnirska selekcija, gdje je $k \in [2, N]$, radi tako da se nasumično odabere k jedinki iz populacije sastavljene od N jedinki. Zatim se biraju dvije najbolje jedinke iz tog skupa koje će biti nositelji daljnjeg genetskog materijala, bilo da će se samo preslikati u iduću generaciju ili će se njihovim križanjem dobiti novo dijete koje će se prenijeti u novu generaciju. Postoje dvije mogućnosti kako ovakav način može funkcionirati - ili će se oba roditelja birati iz istog skupa jedinki pa je moguće da se izaberu dva ista roditelja, ili će se odabrati prvi roditelj iz početnog skupa, a drugi iz skupa iz kojeg je uklonjen prvi roditelj te je tako mogućnost istog roditelja puno manja (nije nemoguća jer je još uvijek moguće da se u skupu nalaze dvije iste jedinke, no to je trenutno manje bitno).

3.3.3. Elitizam i selekcijski pritisak

Kod selekcije dvije stvari želimo izbjeći - eliminiranje najboljih jedinki i selektiranje samo najboljih jedinki.

Elitizam je mehanizam koji omogućava da se sačuva jedna ili više najboljih jedinki od eliminacije. Taj mehanizam je u određenoj mjeri (ne u potpunosti) ugrađen u *k-turnirsku selekciju* jer će se nasumično odabrati k jedinki te je vrlo vjerojatno da će biti sačuvan jednak broj dobrih kao i onih lošijih jedinki. Naravno to ne mora uvijek biti slučaj, no uz dobar odabir parametara i pažljiv pristup selekciji to će uglavnom biti ostvareno. Drugi način za ostvariti elitizam je jednu ili više najboljih jedinki odmah preslikati u iduću generaciju. Tako će jedinke preživljavati dok god obećavaju, odnosno dok god je njihova funkcija dobrote superiorna nad ostalim članovima populacije.

Selekcijski pritisak je svojstvo GA koji pokazuje koliko će kakvih rješenja biti proslijeđeno u iduću generaciju. Ako je selekcijski pritisak prevelik (primjerice pretjerani elitizam), onda će posljedica biti rana konvergencija ka lokalnom optimumu. No ako je premali, rješenje se svodi na slučajno pretraživanje prostora rješenja. Zbog toga treba biti oprezan da se ne dolazi u situaciju jednog od dva ekstrema, već održavati zlatnu sredinu.[3]

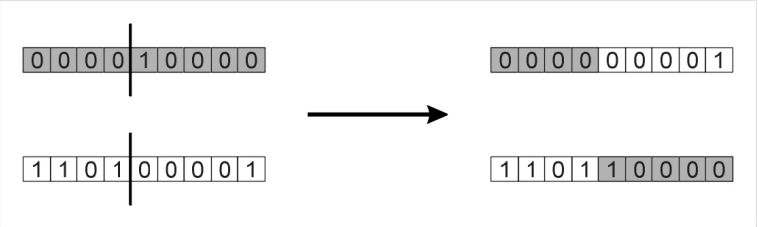
3.4. Genetski operatori

Selekcija je vrlo važan korak svakog genetskog algoritma. Međutim, samom selekcijom se nikada neće napredovati, samo će se sačuvati trenutno najbolje jedinke (ovisi

o implementaciji). Tu na scenu stupaju dva najvažnija genetska operatora - križanje i mutacija.[2][3][8]

3.4.1. Križanje

Koliko je često u svakodnevnom životu čuti izraze poput "Imaš oči na majku." ili "Osmijehom si isti tata." - to pokazuje kako se geni nasljeđuju od roditelja. Po istom principu funkcionira i križanje jedinki kod genetskog algoritma. Selekcijom se biraju roditelji iz kojih će nastati jedno ili više djece. Najčešće je izvedeno tako da se po jedan gen naslijedi od svakog roditelja, izmjenjujući se. No, samo križanje će ovisiti o implementaciji. Ponekad nije moguće križanje izvesti na jednostavan način (posebice ako postoje neke restrikcije i pravila koja se moraju poštivati, što je, pokazat će se, bio problem u implementaciji ovog rješenja) pa se tada javlja potreba za dodatnim parametrima koji bi utjecali na križanje.



Slika 3.3: Križanje

3.4.2. Mutacija

Mutacija dolazi nakon križanja i bitno utječe na daljnji razvoj jedinki. No zašto je mutacija toliko bitna. Neka se kao primjer uzme genetski algoritam koji bi tražio matematički zapis neke funkcije na temelju danih točaka. Neka je prvi roditelj savršeno došao do rješenja za prvu trećinu izraza, a drugi za zadnju trećinu izraza, njihovim križanjem dobiti će se jako dobar izraz koji opet ima slabosti u sebi koje će teško izvući daljnjom rekombinacijom. Takav problem može riješiti mutacija. Ona se uglavnom zadaje kao parametar genetskom algoritmu. Nakon križanja, ako se dogodi mutacija, biti će promijenjen jedan gen dobivene jedinke. To ne mora nužno riješiti problem istog trena, ali je vrlo vjerojatno da će prije ili kasnije za x generacija promijeniti točno onaj dio koji je najslabiji na jedinki te tako poboljšati rješenje. Naravno uvijek postoji rizik da će mutacija izmijeniti dobro rješenje i učiniti ga lošim, no za pretpostaviti

je da postoji još nekoliko jednako dobrih jedinki koje mutacija neće upropastiti te će prenijeti taj dobar genetski materijal na sljedeću generaciju.

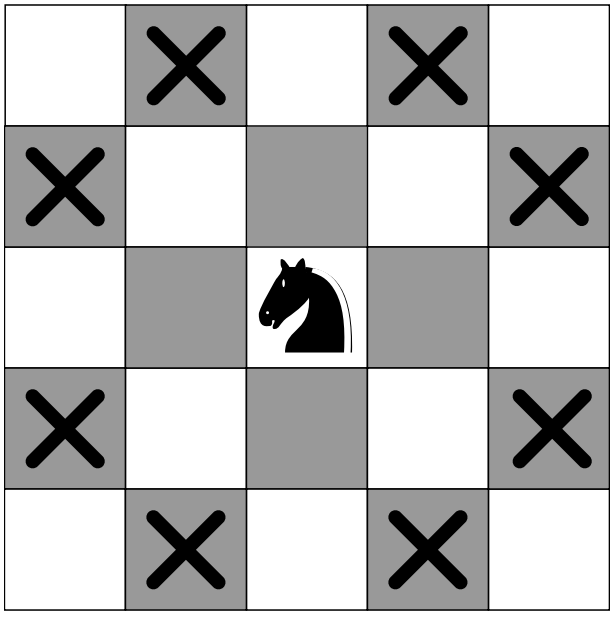
Postoji i nekoliko čestih modifikacija koje se ubacuju u genetske algoritme. Jedna od njih je zaštita najbolje jedinke (ili nekoliko njih) od mutacije, upravo iz iznad navedenog razloga. Druga, puno češća preinaka, je smanjenje mogućnosti mutacije što se dolazi bliže rješenju. Vrlo je vjerojatno da će prva, slučajno generirana, rješenja biti izuzetno loša i neprihvatljiva, no kako generacije odmiču, tako će jedinke postajati sve razvijenije i bolje te će dolaziti sve bliže rješenja. A što su bliže konačnom rješenju, pokušava se smanjivati mogućnost mutacije kako nebi više dobrih rješenja bilo promijenjeno. To se pokazalo kao jako dobra praksa te će se nerijetko u radovima viđati upravo ovakvi mehanizmi.

4. Praktični rad

4.1. Algoritam

Prikaz jedinke

Budući da je već objašnjeno na koje načine se sve jedinka može prikazati, treba odabrati najbolji način za prikazati jedinku specifičnu za problem kojim se ovaj rad bavi. Više je mogućnosti s kojima se krenulo u razmišljanje. Prvotno je ideja bila prikazati jedinku kao polje cijelih brojeva, gdje bi svaki broj bio u rasponu $[1, 8]$. Ti brojevi bi predstavljali korak koji je napravljen (npr. skakač se pomaknuo gore desno, ili dolje lijevo).



Slika 4.1: Skakačevi mogući potezi

Svaki x na slici predstavlja jedan potez, a njihovo označavanje je manje bitno, recimo da je 1 gore desno pa se u smjeru kazaljke na sat brojevi povećavaju. No, gdje je problem s takvim načinom prikaza jedinke? Nekako je trebalo zabraniti nemoguće

poteze, odnosno izlazak skakača sa šahovske ploče, što je, pokazati će se kasnije, još dodatnih izračuna koristi li se ovaj način zapisa jedinke.

Druga ideja je bila prikazati jedinku pomoću polja cijelih brojeva. No tu dolaze drugi problemi. Ideja je od početka bila generalizirati problem i ne fokusirati se na ploču dimenzija 8×8 već proširiti problem na ploču dimenzija $m \times n$, tako da samo zapis koje je polje trenutno posjećeno malo komplicira stvari prilikom izračuna sljedećeg koraka, odnosno sljedećih dozvoljenih koraka.

Na kraju je odluka pala na prikaz složenijom strukturom podataka. Jedinka je prikazana poljem *BoardPosition* objekata.

```
public class BoardPosition {
    private int x;
    private int y;
    private int positionNumber;
    private int stepId;

    public BoardPosition(int x, int y, int positionNumber,
              StepId stepId) {
        this.x = x;
        this.y = y;
        this.positionNumber = positionNumber;
        this.stepId = stepId.value;
    }
}
```

Isječak 4.1: Razred BoardPosition

x i *y* predstavljaju koordinate ploče trenutnog koraka, *positionNumber* je redni broj pozicije na ploči na kojoj se skakač trenutno nalazi $[0, m \times n - 1]$. *stepId* se dobiva is enumeracije *StepId* koja je prikaz poteza koji je odigran da bi se došlo na trenutno poziciju (isto kao što je bilo zamišljeno rješenje u prvom pokušaju). Korak označen kao *ZERO* je zapravo nulta pozicija, odnosno početna točka skakača.

Ovakav prikaz jedinke je vrlo zahvalan, u smislu da je vrlo jednostavno dodati novu informaciju bez puno promjene ako se pokaže da će ona biti potrebna. Cijena koja se plaća za takvu slobodu je memorija, no za ovakav tip problema i uz današnje standarde, ta cijena je zaista simbolična.

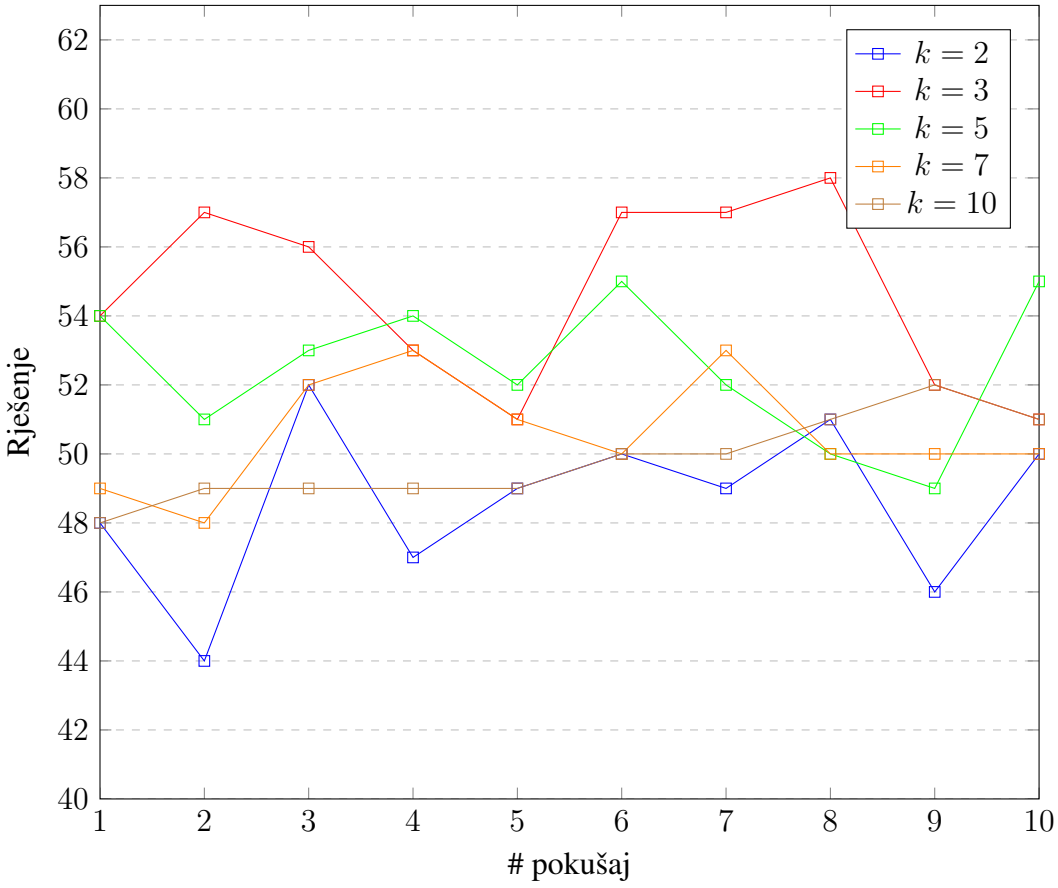
```
public enum StepId {
    ONE(0),
    TWO(1),
    THREE(2),
    FOUR(3),
    FIVE(4),
    SIX(5),
    SEVEN(6),
    EIGHT(7),
    ZERO(-1);

    public final int value;

    private StepId(int value) {
        this.value = value;
    }
}
```

Isječak 4.2: Enumeracija StepId

Selekcija

Selekcija je, kao što je već opisano, jako bitan faktor genetskog algoritma, stoga je potrebno posvetiti dovoljno pozornosti na taj dio kako bi se izveo na najbolji mogući način. Prvotno je ideja bila implementirati *jednostavnu proporcionalnu selekciju*, no jednostavnosti radi i zbog više mogućnosti koje se trebaju istraživati, na kraju je odluka pala na *k-turnirsku selekciju*. Trebalo je zatim odrediti k koji će davati najbolje rezultate. Iako je rad napisan s $k = 10$, na kraju je napravljen test da se vidi koji k će davati najbolje rezultate (već se može primijetiti kako rezultati dobiveni samo pomoću genetskog algoritma nisu zadovoljavajući).

Može se primijetiti kako za $k = 3$ graf odskače od ostalih, no valja napomenuti kako je testiranje i ispitivanje parametara za ovaj slučaj ni približno dovoljno za donijeti konačan zaključak. Postojala je ideja napisati skriptu koja bi kvalitetnije testirala ovakve slučajeve, međutim, to će morati pričekati. Na kraju je za selekciju odabrana *k-turnirska selekcija* s parametrom $k = 3$.



Slika 4.2: Ovisnost rješenja o parametru k

```
public static Solution kTournamentSelection(
          Board board, Solution[] population,
          int tournamentSize, boolean mutation) {

  List<Solution> parents=new ArrayList<>(tournamentSize);
    for(int i = 0; i < tournamentSize; i++) {
      parents.add(population[
        Util.random.nextInt(population.length)
      ]);
    }

    Collections.sort(parents);

  return crossover(board, parents.get(0),
          parents.get(1), mutation);
}
```

Isječak 4.3: Implementacija *k-turnirske selekcije*

Trenutno *board* nije bitan, on se nalazi ovdje radi nekih implementacijskih detalja nevažnih za sam algoritam. Selekcija radi tako da nasumično odabere 3 primjerka jedinke (jedinka je prikazana razredom *Solution* u kojemu se nalazi polje objekata *BoardPosition*), sortira ih[1] te odbacuje najlošiju, a preostale dvije križa i dobiva novu jedinku koja se prenosi u novu generaciju.

Postoje razni oblici kako selekcija može biti izvedena, oba roditelja se prenose u novu generaciju, bolji roditelj se prenosi, križanjem se dobiva samo jedno dijete, križanjem se dobiva dvoje djece i razne druge kombinacije kojih se može nabrajati i nabrajati. No ovdje je odluka pala na oblik gdje se niti jedan od roditelja ne prenosi izravno u iduću generaciju, već se prenosi dijete koje nastane križanjem. U idućih nekoliko odlomaka će biti objašnjeno kako je križanje izvedeno, sada je samo bitno da nastaje jedno dijete. Pokazalo se kako to na kraju i nije bilo najspretnije rješenje, rezultati su mogli biti i znatno bolji da je izabran malo drugačiji pristup, no o tome više u paragrafu **Zaključak o algoritmu**.

Križanje

Križanje je genetski operator koji za ovaj tip problema nije bilo jednostavno izvesti. Naime, stvar bi bila poprilično jednostavna kada generiranje nemogućih rješenja nebi bio problem, no budući da se generiranje nemogućih rješenja pokušava izbjeći, stvari su puno kompliciranije no što bi netko na prvu pomislio. Vidjet će se nešto kasnije kako je nedovoljno dobro rješenje dobiveno samo genetskim algoritmom zapravo izravna posljedica križanja. No kako je križanje izvedeno?

Prvotno je ideja bila napraviti klasično križanje - dijete će naslijediti po jedan gen od svakog roditelja naizmjence. Naravno, to za ovaj problem nije moguće. Neka su roditelji dani varijablama *parent1* i *parent2* i neka je n-ti gen djeteta uzet od roditelja *parent1*. Vjerojatnost da je $n+1$-ti gen jedinke roditelja *parent2* legalan potez n-tog gena djeteta je vrlo mala. Ako i je moguća, to bi se trebalo podudarati $m \times n$ puta, gdje su m, n dimenzije polja. Očito je kako takav pristup nije zadovoljavajuć jer je gotovo sigurno da će se prije ili kasnije generirati nemoguće rješenje.

Idući logičan korak bio je povećati broj gena koji će dijete naslijediti od svakog roditelja. Jeli to uopće moguće ako uzmemo pola gena od svakog roditelja? Moguće je, ali opet, vrlo je vjerojatno da se neće podudarati svaki put tako da problem ostaje isti kao u prethodnom primjeru. No ideja nije loša, mogla bi voditi do nekog smislenog,

[1] sortiranje se radi na temelju vrijednosti evaluacije funkcije dobrote (engl. *fitness*), silazno od najveće prema najmanjoj

zadovoljavajućeg rješenja.

Kako nije uspjelo uzimanje pola gena od svakog roditelja za genetski materijal djeteta, sasvim je razumno taj korak smanjiti. No, niti to ne rješava problem generiranja nemogućih rješenja. Stoga je rješenje konstruirano na sljedeći način:

```
currentParent = parent1
otherParent = parent2
for i in length(parent)
  if field >= K && isLegalMove(otherParent, i)
    switchParents()
    field = 0
  child[i] = currentParent[i]
  field++
```

Isječak 4.4: Pseudokod križanja

Prije iteriranja po listi gena bilo kojeg roditelja (ovdje je nebitno po kojem roditelju se iterira budući da je bitna samo duljina, a ona je uvijek ista), pripremaju se nove varijable *currentParent* i *otherParent* koje predstavljaju roditelje djeteta (opet je sasvim nebitno koji je roditelj gdje, dok god se isti roditelj ne stavi u obje varijable). Zatim se u svakoj iteraciji provjerava jeli već preslikano K gena[2] iz roditelja *currentParent* i jeli taj potez legalan za roditelja *otherParent* te ako je, roditelji se zamijene i brojač se resetira. Ako potez nije legalan, u dijete će se nastaviti preslikavati geni iz trenutnog roditelja, a brojač će se nastaviti povećavati.

Možda se na prvi pogled čini kao nešto što ne radi baš najbolje, nešto što će vrlo rijetko uzimati gene oba roditelja jer će se vrlo rijetko podudarati baš na mjestu na kojem bi trebali, no to nije slučaj. Pokazalo se kako se uglavnom roditelji izmjenjuju u preslikavanju gena na dijete svakih $K, K+1, K+2$ gena, vrlo se rijetko događalo da se zamjena roditelja dogodi nakon više od $K+2$ gena.

[2] K je jedan od parametara koji se zadaju u genetskom algoritmu, a predstavlja broj gena koji će biti preslikan iz svakog roditelja

```
Solution current = parent1;
Solution other = parent2;
int fieldSizeCounter = 0;

for(int i = 0; i < parent1.getSteps().length; i++) {
  if(i == 0) {
    steps[i] = current.getSteps()[i];
    continue;
  } else {
  if(fieldSizeCounter >= 3) {
    while(true) {
      if(isLegalMove(board, other.getSteps()[i],
              steps[i - 1])) {
        current = current.equals(parent1) ?
            parent2 : parent1;
        other = other.equals(parent1) ?
            parent2 : parent1;
        fieldSizeCounter = 0;
        break;
      }

    steps[i] = current.getSteps()[i];
    if (i == current.getSteps().length - 1) break;
    i++;
    }
  }

  steps[i] = current.getSteps()[i];
  fieldSizeCounter++;
  }
}
```

Isječak 4.5: Implementacija križanja jedinki

Mutacija

Mutacija je još jedan od parametara koje zadaje korisnik kao cijeli broj u rasponu od $[0 - 100]$, gdje taj broj predstavlja vjerojatnost da će se mutacija dogoditi. No, i kod mutacije je bio problem kako mutirati jedinku bez da se generira nemoguće rješenje.

Jasno je odmah kako promjena samo jednog gena jedinke nije rješenje koje bi uvijek generiralo samo moguća rješenja, dapače, to gotovo nikada nebi bio slučaj.

Moglo bi se to izvesti na način da se mutirani gen prilagodi genu iza i ispred sebe, ali to bi postavilo previše restrikcija te bi se na kraju svelo na jednu ili, eventualno, dvije mogućnosti.

Rješenje koje je implementirano prilikom rješavanja ovog problema je to da se jedinka mutira od nasumično odabranog gena. Znači neće se mutirati samo jedan gen, već svi geni o tog jednog gena pa do kraja, ali pazeći pritom da se ne naruši rješenje, odnosno da se mutiranjem ne dobije neželjena situacija. Metoda koja mutira jedinku zapravo zove metodu koja će tu istu jedinku popuniti od tog određenog gena pa skroz do kraja.

```
for(int i = fromStep; i < steps.length; i++) {
  if(i == 0) {
    steps[i] = board.getStart();
  } else {
    steps[i] = generateMove(steps[i-1], board);
  }

  beenToMatrix[steps[i].getX()][steps[i].getY()] = true;
}
```

Isječak 4.6: Implementacija mutacije

Tako se vrlo efikasno izbjeglo nemoguće rješenje, a opet se izgenerirala nova jedinka koja se bitno razlikuje od originala. To da se bitno razlikuje možda i nije pogodno te je, pokazalo se, upravo to jedna od manjkavosti implementiranog algoritma.

Zaključak o algoritmu

Dan je opis glavnih dijelova genetskog algoritma implementiranog za rješavanje problema skakača. Natuknuto je već na nekoliko mjesta koje su zapravo manjkavosti algoritma, no sada će biti objašnjene na jednome mjestu. Iz do sada viđenog jasno je kako genetski algoritam ne daje konačno rješenje za *skakačev put*, niti ovdje opisan niti bilo koji drugi koji je do sada isproban i implementiran. Naravno, moguće je da ponekad i uspije GA doći do konačnog rješenja, no te su situacije vrlo rijetke i uglavnom se dogode u zanemarivom broju slučaja.

Koje su zapravo manjkavosti ovdje opisanog genetskog algoritma? Prvi bitan nedostatak događa se kod same selekcije. Selekcija kao takva nije loša, dapače dosta je dobra, ali način da se iz selekcije biraju dva roditelja od kojih će nastati samo jedno dijete koje će se prenijeti u iduću generaciju nije dobar, barem ne za ovakav tip pro-

blema. Naime lako je moguće da su oba roditelja bolja od nastalog djeteta, no njihov se genetski materijal jednostavno izgubi. Možda nebi bilo loše da se to izvelo na način da iz roditelja nastane dijete te se to dijete i bolji od roditelja prenosi dalje u iduću generaciju. Ili da nastanu dva djeteta pa se bolje dijete i bolji roditelj prenosi dalje. Ili neka varijacija na temu, ali je važno da je bit jasna - opisan pristup kako algoritam to radi nije najspretnije rješenje.

Nadalje se kao nedostatak pokazuje samo križanje. Iako je spomenuto kako križanje funkcionira dobro te uzima otprilike jednak broj gena iz svakog roditelja, zapravo ako je uzeto više od jednog gena, nova jedinka će sa sobom povlačiti probleme koje su imale prijašnje jedinke u preuzetim genima. To bi se križanjem trebalo smanjivati (barem u većini slučajeva), no pokazuje se kako to na ovaj način ne funkcionira najbolje. Jedinka nastala križanjem je vrlo rijetko sadržavala bolji genetski materijal od svojih roditelja.

Zadnji nedostatak je način na koji je izvedena mutacija. Već je objašnjeno kako je teško mutirati jedinku bez da se naruši pravilo o generiranju nemogućih rješenja, ali to da se jedinka promjeni od nekog n gena pa sve do kraja može rezultirati uništavanjem vrlo dobrih jedinki. To bi se dalo spriječiti zaštitom dobrih jedinki, međutim to ovdje nije implementirano.

4.2. Warnsdorfovo pravilo

Genetskim algoritmom nije moguće uvijek doći do rješenja, niti opisanim algoritmom niti bilo kojim drugim do sada implementiranim. Iako sam cilj ovoga rada nije bio pronaći rješenje, kao završna točka ubačena je heuristika koja bi pogurala genetski algoritam i pomogla mu doći do konačnog rješenja. Korištena heuristika je Warnsdorfovo pravilo.

Kao izazov čitaču neka bude da pokuša riješiti problem koji je tema ovoga rada na nekoj manjoj ploči ,primjerice 5×5 ili 6×6, ili barem prvih 10-ak koraka. Kako većina ljudi započne traženje rješenja? Zasigurno bez puno razmišljanja krenu sasvim nasumično gledati gdje bi skakač iduće mogao doći ili ga smještaju tamo gdje je put do tada bio najrjeđi, što je sve sasvim logično. No takav način rješavanja gotovo sigurno nikada neće dovesti do konačnog rješenja. Ovaj mali eksperiment je zapravo samo pokazatelj kako ljudski mozak intuitivno krene razmišljati te kako nitko ni ne pomišlja krenuti na način kako to kaže Warnsdorfovo pravilo. Pravilo je samo po sebi vrlo jednostavno, skakač će od svih mogućih pozicija skočiti na onu koja će imati najmanje idućih mogućih pozicija. Pritom se mora paziti da se ne skače na polje koje nema

mogućih poteza, osim ako to nije posljednji potez. To zapravo rezultira time da skakač ide prema zidovima te se na početku rješava polja koja ga ograničavaju. No, razmisli li se malo, skakač će, prođe li uz sve zidove, zapravo stvoriti novi zid te smanjiti ploču po kojoj se može kretati te tu Warnsdorfovo pravilo dolazi do izražaja. Vidimo kako ono zapravo djeluje rekurzivno, stvaranjem novih "zidova" i smanjivanjem ploče, skakač je primoran ponovno se kretati uz "zidove" te tako sve dok ne dođe do posljednjeg poteza. Pravilo je vrlo jednostavno, kako za objasniti tako za implementirati, a daje izvrsne rezultate - ali i točne, što je najbitnije od svega!

Ova heuristika je ugrađena u genetski algoritam na način da se uključuje nakon pređenih 30% generacija od ukupnog predviđenog broja generacija. Tada će algoritam vrlo brzo pronaći rješenje (govorimo o 8×8 ploči, no rješenje će se pronaći i za manje, ali i ploče nešto većih dimenzija). Netko bi mogao pomisliti kako heuristika obavlja glavninu posla u rješavanju problema i nebi puno pogriješio. Međutim, to ne predstavlja problem, ionako čistim genetskim algoritmom nije moguće doći do rješenja, pa zašto se nebi iskoristilo nešto što garantira uspješnost, barem za ploče razumnih dimenzija.[7]

4.3. Analiza implementiranog GA

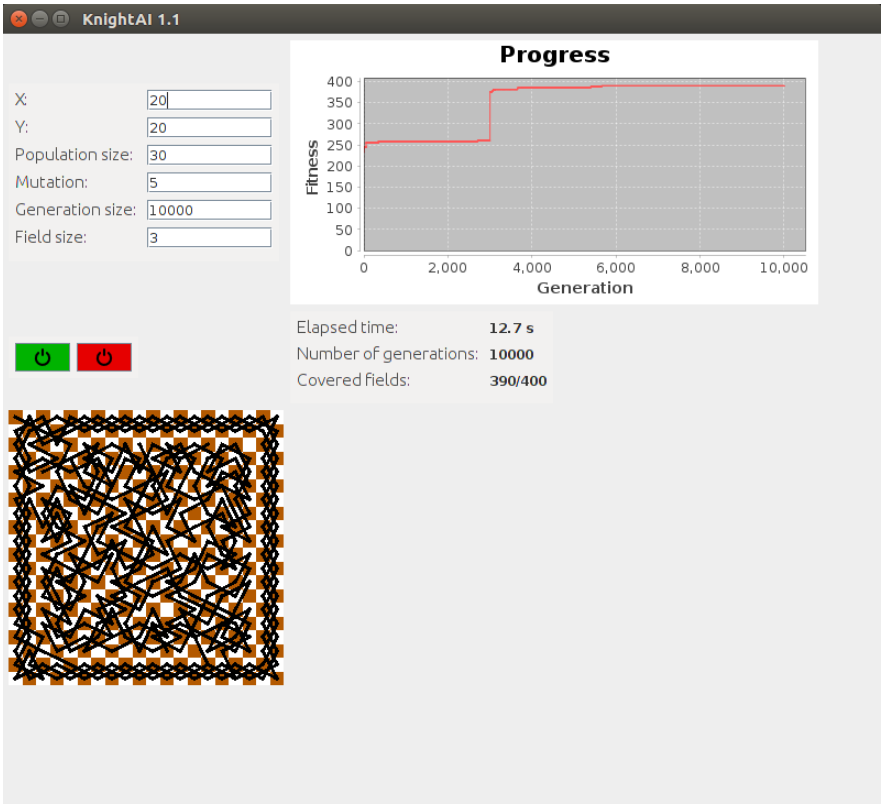
U ovom će se dijelu pokazati usporedba algoritma kako radi s heuristikom i bez nje te tako pokazati zašto genetski algoritam ne daje rezultate za ovaj problem.

Na prvoj ploči, na slici 4.5, primjećuje se kako niti sa heuristikom, niti bez nje, algoritam nije uspio doći do rješenja, što je zapravo prilično zanimljivo budući da za tu ploču postoji rješenje. Objašnjenje toga se može naći u manjkavosti genetskog algoritma i nepotpunosti korištene heuristike.

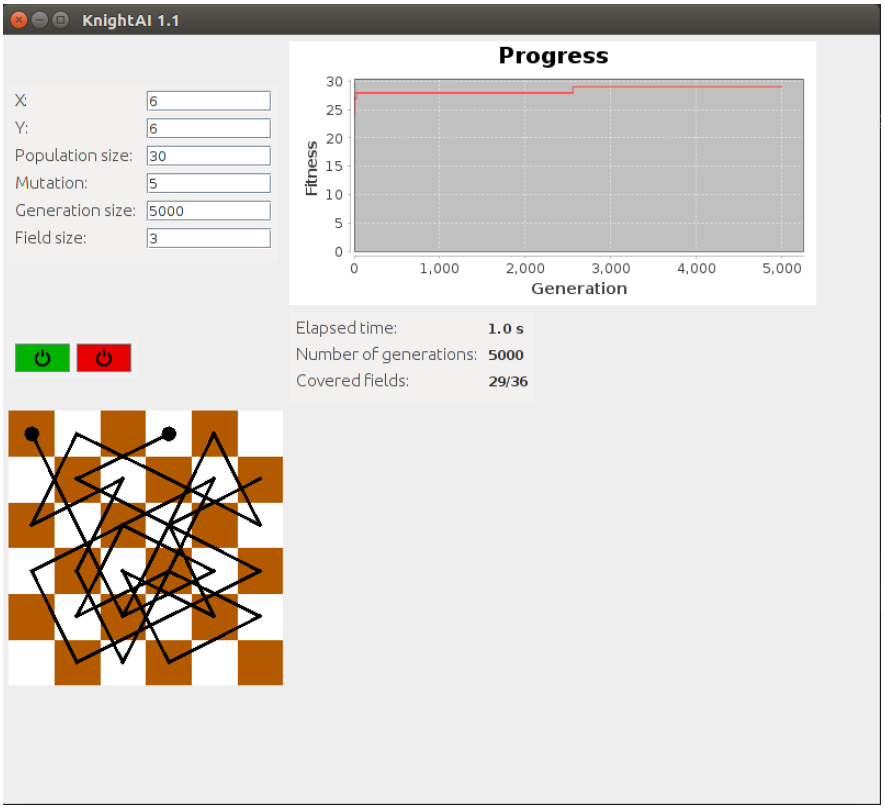
Možda i najbolji prikaz koliko je neprimjenjiv genetski algoritam na ovaj problem može se vidjeti na originalnoj veličini ploče, onoj dimenzija 8×8. Svaki skakačev potez je u potpunosti nasumičan, drži se jedino ograničenja ploče, dok ga kazne za stajanje na već posjećena polja ne tjeraju na evoluiranje. Može se primijetiti i kako je napredak generacija izuzetno spor te u većini slučajeva gotovo zanemariv. Tu se najviše osjeća taj nedostatak loše mutacije uzrokovane striktnim ograničenjima (vidi 4.1). Ako se pogleda primjer gdje je heuristika primijenjena nije niti potrebno znati kada se ona uključuje, već je po samom grafu jasno. Radi se o onom velikom skoku u 1500. generaciji. Nakon toga, potrebno je svega par koraka kako bi se došlo do konačnog rješenja, ali ne radi genetskog algoritma, već heuristike.

Zadnji primjer koji će ovdje biti prikazan je onaj za ploču dimenzija 20×20. Iako

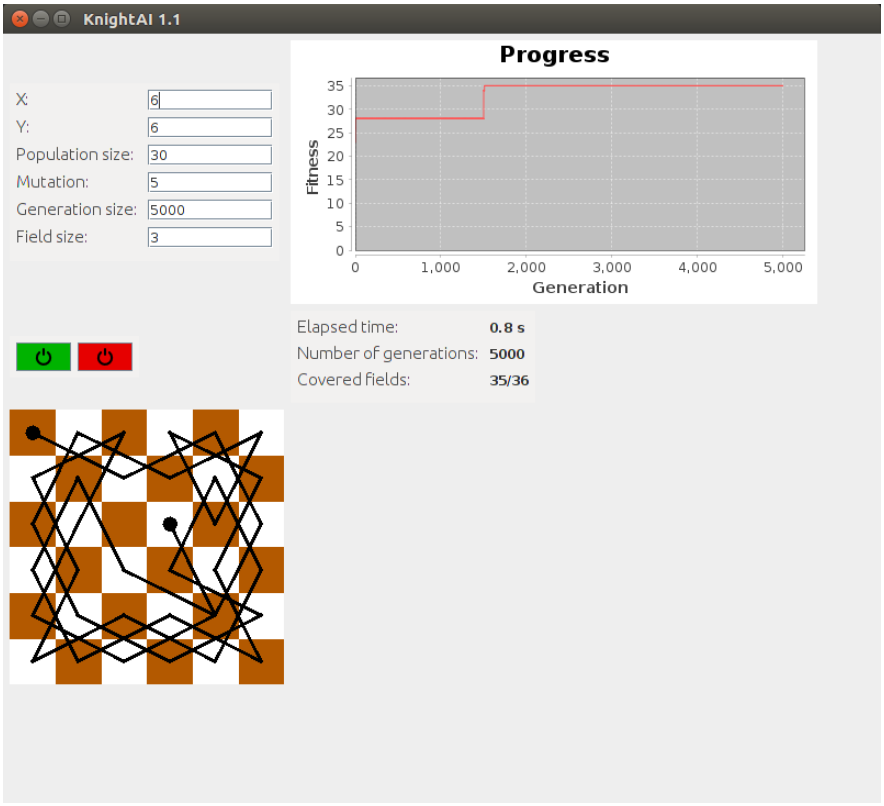
GA niti uz pomoć heuristike nije uspio doći do rješenja, najbolje se prikazuje kako Warnsdorfovo pravilo funkcionira, prolazeći prvo uz zidove pa tek onda prema sredini.



Slika 4.3: Primjer na ploči dimenzija 20×20 (GA + heuristika)

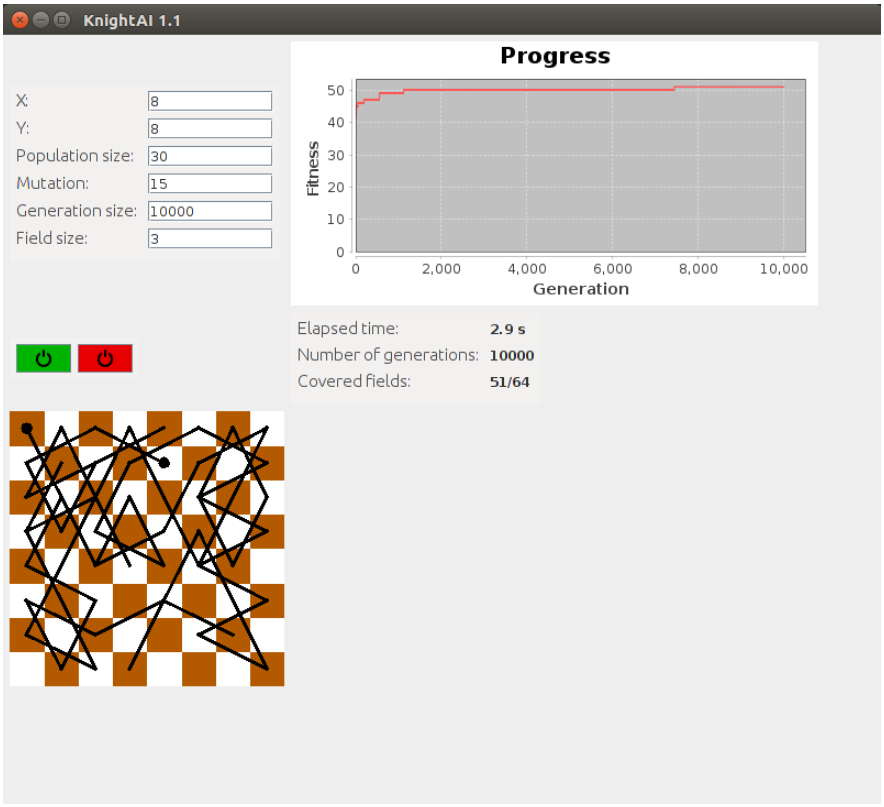


(a) Bez heuristike

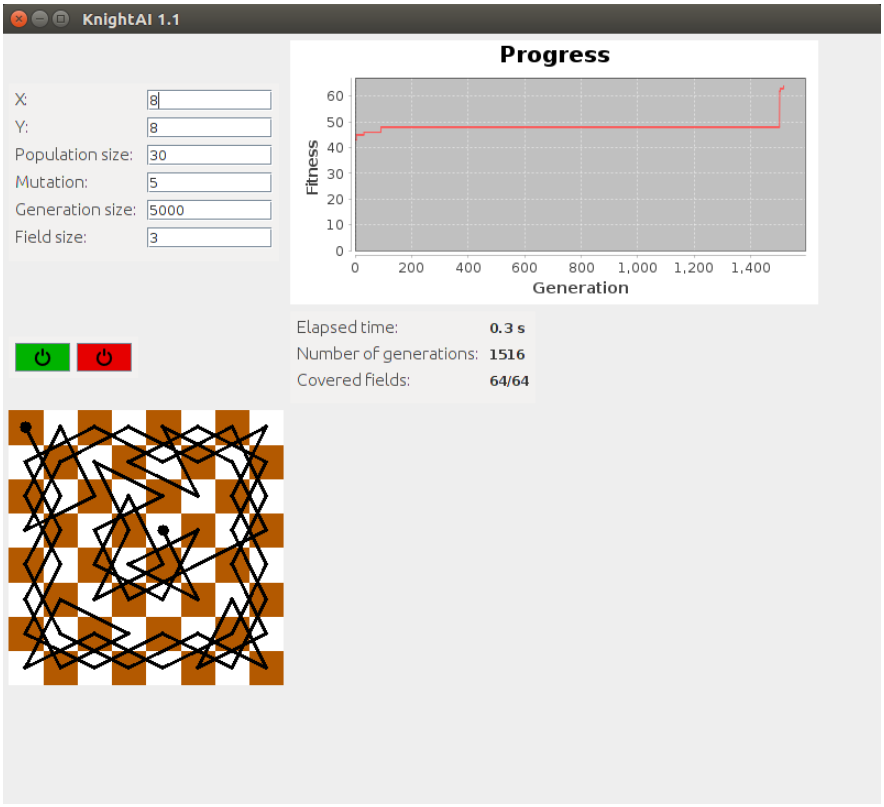


(b) S heuristikom

Slika 4.4: Primjer na ploči dimenzija 6×6



(a) Bez heuristike



(b) S heuristikom

Slika 4.5: Primjer na ploči dimenzija 8×8

5. Usporedba s referentnim radom

5.1. Genetski algoritam i heuristika

Sada kada je opisan programski ostvaren algoritam, istaknute su njegove manjkavosti i pokazano je zašto korištenje GA nije najspretnije rješenje, može se napraviti usporedba s radom koji je poslužio kao referentan ovome [1].

Jedinka je predstavljena kao binarno kodirana lista poteza koje je skakač napravio, gdje su potezi u rasponu $[000 - 111]$.
„Each chromosome represents a sequence of 63 moves plus the start square with a total of length 64*3 = 192 bits. The start state can be set to any square on the board. Each 3-bit substring represents the next move on the board according to the current state.“
Može se primijetiti kako ovdje nije naglasak na generiranju samo mogućih rješenja, ovdje je vrlo izgledno da skakač neće završiti u polju, budući da nema dodatnih provjera.

Što se tiče križanja i mutacije, korišteno je križanje po samo jednoj točki (engl. *one point crossover*), što bi značilo da se odredila točka koja je predstavljala granicu te su se geni do te točke uzimali od jednog roditelja, a od te točke od drugog. Razlika od ovdje opisanog algoritma je u tome što nisu križanje nužno obavljali svaki put, već s nekom vjerojatnošću. Mutaciju su pak implementirali tako da se promijeni samo jedan gen, što je naravno moguće ako se ne pazi na generiranje nemogućih rješenja.
„In our approach we used a one point crossover that is selected randomly, the crossover rate is between 85% and 95% with 0.5% mutation rate.“

Za selekciju je odabrana *jednostavna proporcionalna selekcija*.
„Chromosomes are selected using Russian Roulette wheel selection, where the better chromosomes have a greater chance of being selected. Using this selection method, a maximum of a quarter of the chromosomes (worst chromosomes in generation) are replaced by better chromosomes in the generation. This gives a greater chance for the best chromosomes to participate in the crossover to produce the next generation, and reduces the chances of the worst chromosomes to produce the next generation.“

Jasno je kako ovakvo rješenje nije moglo dati prihvatljive rezultate, stoga su napravljene neke izmjene. Izvjesno je kako će se gore navedenim pristupom vrlo vjerojatno generirati nemoguće rješenje. Taj su problem riješili tako što su svaki puta kada bi skakač povukao nemogući potez ili se vratio na poziciju koju je već posjetio iskoristili heuristiku. Korištena heuristika je Warnsdorfovo pravilo, opisana i u ovome radu. Naravno, prije provođenja heuristike, zadnji potez bi se poništio i zamijenio novim potezom dobivenim heuristikom. Postupak bi se ponavljao dok god nebi došlo do situacije da više nije moguće generirati legalan potez - to je značilo da je pronađeno rješenje.
„For each partial solution (chromosome in a generation), when a knight jumps off the board or returns to a previously visited square, a modification with heuristic is applied. We try to replace the 3-bit string that represents the illegal move with another 3-bit string that allows the knight to proceed. Before every move, we examine the squares that can be reached with legal moves from the current square. Then for each of those possible next squares, we count the number of legal moves at that square. . The knight then moves to the square with the lowest number of new choices. We repeat for each chromosome until a substitution cannot be made, and the evaluation of that chromosome then ends.“

Tablica 5.1: Usporedba rezultata u radu [1]

	GA + heuristika	Heuristika
# pronađenih puteva	12,084	1,979
# iteracija	800,000	800,000
Postotak uspješnosti iteracija	1.51%	0.25%
Prosječan # puteva u svakom kvadratu	189	31
Neuspješne iteracije	1%	20%
Najveći broj pronađenih rješenja u iteraciji	22	-
Prosječan broj generacija za dobivanje rješenja	4.9	-

Jasno se vidi kako ovdje heuristika bez genetskog algoritma daje puno slabije rezultate, no taj aspekt je teško usporediti s algoritmom opisanom u ovome radu. Budući da nije implementirano rješenje koristeći samo heuristiku, ne mogu se usporediti ti rezultati. Isto tako, budući da ovdje opisan algoritam staje nakon pronađenog samo jednog rješenja, nije moguće usporediti broj dobivenih rješenja u ovome i referentnom radu. Ono što se može usporediti je prosječan broj generacija potrebnih da se dobije rješenje koristeći GA i heuristiku (bez da se heuristika uključuje tek nakon 30% generacija) -

2.5! Tako da konačno možemo zaključiti kako rješenje implementirano u ovome radu nije loše, ali zbog raznih ograničenja (heuristika se ne uključuje odmah, generiranje samo mogućih rješenja, itd.) do rješenja se dolazilo nakon puno više koraka nego što bi to bilo realno.

5.2. Kolonija mrava

Mravi su fascinantna bića koja unatoč gotovo potpunom nedostatku vida uspijevaju pronaći najkraći put do izvora hrane. Taj prirodni fenomen moguć je zbog feromona kojeg izlučuju i kojim se vode prilikom otkrivanja najkraćeg puta. *Rješavanje problema obilaska skakača svih polja na šahovskoj ploči* moguće je riješiti pomoću algoritma kolonije mrava i to puno efikasnije za veće ploče gdje Warnsdorfovo pravilo zakazuje, kako u brzini tako i u pronalasku rješenja. Algoritam radi tako da jedan mrav rješenja traži tako da na svakom polju koje posjeti istražuje koje bi iduće polje bilo najidealnije za posjetiti, uzimajući u obzir količinu feromona na svakom polje - vjerojatnije je da će ići tamo gdje je koncentracija feromona veća. Na taj način dolazak do rješenja nije u potpunosti nasumičan te je kao takav puno efikasniji za pronalazak *skakačevog puta*, pogotovo za ploče većih dimenzija ($m, n > 20$).[4]

Iako ovdje algoritam kolonije mrava nije u detalje objašnjen već samo njegova ideja, on izgleda jako obećavajuće i ostavlja se mogućnost implementacije algoritma za budući rad.

6. Zaključak

U ovom radu dan je kratak uvod u optimizacijske problema, zašto su oni potrebni te kada se koriste. Opisan je problem skakača te se rješenju tog problema pristupilo korištenjem jednog od prirodom inspiriranih optimizacijskih algoritama. Opisan je implementirani algoritam i na kraju je napravljena usporedba s referentnim radom.

Već je više puta napomenuto kako genetski algoritam ne radi najbolje u problemima gdje postoje neke zabrane ili ograničenja. Pritom se na ograničenja misli ona da se mora paziti da ne izađe s ploče ili slična, ne na ograničenja domene svakog gena. On će upravo zbog svojeg svojstva nasumičnosti raditi najbolje za probleme gdje ograničenja nema te će za takve probleme davati očekivano dobre rezultate. U radu [1] GA je dobio malo slobode tako što je ograničenje izlaska s ploče samo strože kažnjavano ili popravljano, no sam potez nije zabranjen.

Unatoč neuspješnom pokušaju rješavanja problema obilaska skakača svih polja na šahovskoj ploči uz pomoć genetskog algoritma, ideja rješavanja tog problema nekim drugim prirodom inspiriranim optimizacijskim algoritmom i dalje je vrlo interesantna te je ostavljena kao mogućnost u daljnjem radu pokušati pristupiti problemu nekim od tih algoritama.

LITERATURA

[1] Jafar Al-Gharaibeh, Zakariya Qawagneh, i Hiba Al-Zahawi. Genetic algorithms with heuristic - knight's tour problem. Technical report, University of Idaho, Jordan University for Science and Technology, University of Utah, 2007.

[2] Marin Golub. *Genetski algoritam prvi dio*. FER, 2004.

[3] Marin Golub. *Genetski algoritam drugi dio*. FER, 2004.

[4] Philip Hingston i Graham Kendall. Enumerating knight's tours using an ant colony algorithm. Technical report, Edith Cowan University, The University of Nottingham, 2005.

[5] Brendan D. McKay. Knight's tours of an 8 x 8 chessboard. Technical report, Computer Science Department, Australian National University, 1997.

[6] Allen J. Schwenk. Which rectangular chessboards have a knight's tour? *Mathematics Magazine*, 64(5):325–332, 1991.

[7] Douglas Squirell i Paul Cull. A warnsdorff-rule algorithm for knight's tours on square chessboards. Technical report, 1996.

[8] Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi*. FER, 2013.

Rješavanje problema obilaska skakača svih polja na šahovskoj ploči korištenjem genetskih algoritama

Sažetak

U ovom radu dan je kratak uvod u optimizacijske probleme i genetski algoritam te se genetskim algoritmom pristupilo problemu rješavanja obilaska skakača svih polja na šahovskoj ploči. Također je uvedena heuristika kao dodatak genetskom algoritmu. Pokazani su nedostatci primjene GA na ovakav tip problema te je dano nekoliko načina rješavanja koji su bolji i efikasniji.

Ključne riječi: Optimizacija, genetski algoritam, heuristika, šah

Solving Knight's Tour Problem by Using Genetic Algorithms

Abstract

In this paper short intro to optimization problems and genetic algorithm is given. Genetic algorithm was used to solve knight's tour problem, and heuristic has been implemented when GA was found to be non-optimal solution. Flaws of GA were stated and also some other approaches that would bear better results.

Keywords: Optimization, genetic algorithm, heuristic, chess