

*Zahvaljujem se mentoru, prof. dr. sc. Marinu Golubu na usmjeravanju i savjetima  
tijekom izrade ovog rada.*

# SADRŽAJ

<b>1. Uvod</b>	<b>1</b>
<b>2. Kromatski broj grafa</b>	<b>2</b>
2.1. Motivacija . . . . .	2
2.2. Određivanje kromatskog broja . . . . .	3
2.2.1. Iscrpno pretraživanje . . . . .	3
2.2.2. Pohlepno pretraživanje . . . . .	4
2.2.3. Usmjereno iscrpno pretraživanje . . . . .	4
2.3. Gornja ograda na kromatski broj . . . . .	5
2.4. Donja ograda na kromatski broj . . . . .	6
<b>3. Evolucijski algoritmi</b>	<b>7</b>
3.1. Genetski algoritmi . . . . .	7
3.1.1. Kodiranje genoma . . . . .	8
3.1.2. Funkcija cijene . . . . .	8
3.1.3. Određivanje roditelja . . . . .	9
3.1.4. Križanje . . . . .	11
3.1.5. Mutacije . . . . .	12
3.2. Programsko ostvarenje genetskog algoritma . . . . .	13
<b>4. Korištene knjižnice programa i podaci</b>	<b>14</b>
4.1. Knjižnica programa PyGAD . . . . .	14
4.1.1. Instalacija . . . . .	14
4.2. Paket NetworkX . . . . .	14
4.2.1. Instalacija . . . . .	14
4.3. Web stranica House of Graphs . . . . .	15
<b>5. Praktični dio</b>	<b>16</b>
5.1. Određivanje dobrote jedinki . . . . .	16

5.2. Određivanje roditelja . . . . .	16
5.3. Križanje . . . . .	16
5.4. Mutacije . . . . .	17
5.4.1. Slučajna mutacija na pogrešno obojanim vrhovima . . . . .	17
5.4.2. Ciljana mutacija na pogrešno obojanim vrhovima . . . . .	17
5.4.3. Kombinacija slučajne i ciljane mutacije na pogrešno obojanim vrhovima . . . . .	18
5.5. Ostali parametri . . . . .	19
5.6. Postupak određivanja kromatskog broja . . . . .	19
5.7. Vizualizacija rezultata . . . . .	20
<b>6. Rezultati praktičnog dijela rada</b>	<b>21</b>
6.1. Manji skup podataka . . . . .	21
6.1.1. Usporedba rezultata po strategiji selekcije roditelja . . . . .	25
6.2. Veći skup podataka . . . . .	26
<b>7. Zaključak</b>	<b>28</b>
<b>Literatura</b>	<b>30</b>

# 1. Uvod

Kod NP-teških problema pretraživanja nije moguće pronaći optimalno rješenje iscrpnim pretraživanjem u realnom vremenu za velike skupove podataka. Iz tog se razloga koriste heuristički algoritmi koji uz različite pretpostavke o problemu pronalaze rješenja u prihvatljivom vremenu, smanjujući prostor pretraživanja. Upravo zbog smanjivanja prostora pretraživanja rješenja pronađena takvim algoritmima nisu uvijek optimalna.

Bojanje grafa, pronalazak njegovog kromatskog broja, je NP-težak problem određivanja minimalnog broja boja kojima se graf može obojati tako da su svaka dva susjedna vrha obojana drugom bojom. Evolucijski algoritmi su heuristički algoritmi inspirirani procesima u prirodi, a posebno su učinkoviti kod rješavanja optimizacijskih problema. Ovaj rad pobliže opisuje način rada evolucijskih algoritama, njihov podskup genetskih algoritama, problem bojanja grafa te primjenu i analizu evolucijskih algoritama pri određivanju kromatskog broja grafa.

U drugom poglavlju daje se kratko objašnjenje i uvod u problem određivanja kromatskog broja grafa. U poglavlju se analiziraju iscrpni algoritmi traženja kromatskog broja. Zatim se u trećem poglavlju objašnjava princip rada evolucijskih algoritama te se detaljnije obrađuju genetski algoritmi, njihov način rada te različite varijacije parametara i funkcija koje utječu na kvalitetu dobivenog rješenja. U četvrtom poglavlju dan je kratak pregled korištenih biblioteka i podataka.

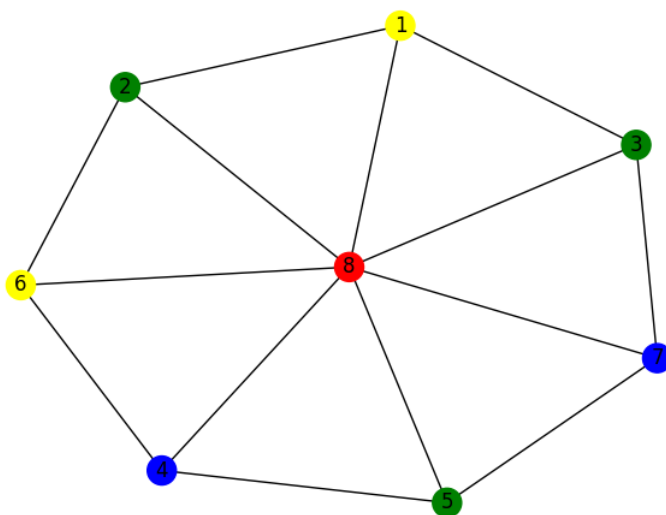
Peto poglavlje opisuje način određivanja kromatskog broja grafa genetskim algoritmom, implementaciju rješenja te korištene parametre i razloge njihovog odabira. U šestom se poglavlju iznose i uspoređuju rezultati testiranja različitih varijacija algoritma. Sedmo poglavlje objašnjava dobivene rezultate i iznosi zaključak.

## 2. Kromatski broj grafa

Kromatski broj grafa,  $\chi(G)$ , predstavlja minimalan broj boja s kojima se mogu obojati vrhovi grafa, tako da niti jedna dva susjedna vrha (vrhovi spojeni bridom) nisu obojana istom bojom [5].

### 2.1. Motivacija

Problem bojanja grafa vrlo je bitan problem zbog njegove široke primjene. Kromatski broj se koristi u rješavanju mnogih problema; problema raspoređivanja, dodjele registara u procesoru, rješavanja Sudoku, dodjele radio frekvencije, sparivanje uzoraka, sinkronizacije paralelnih procesa, bojanje karata itd. Svi ovi problemi mogu se reprezentirati grafom, a njihovo rješenje bojanjem zadanog grafa. Pronalaskom kromatskog broja rješavamo problem koristeći minimalne potrebne resurse [7].



**Slika 2.1:** Graf s osam vrhova i  $\chi(G) = 4$

## 2.2. Određivanje kromatskog broja

Pronalazak kromatskog broja grafa je NP-težak problem, što znači da je vremenska složenost njegovog pronalaska eksponencijalna [8]. Zbog toga nije praktično koristiti se algoritmima iscrpnog pretraživanja na grafovima s većim brojem vrhova.

### 2.2.1. Iscrpno pretraživanje

Jednostavan algoritam iscrpnog pretraživanja prošao bi sve kombinacije  $k$  boja i  $n$  vrhova; njegova vremenska složenost je  $O(k^n)$ . Implementacija iscrpnog algoritma određivanja kromatskog broja rekurzijom u programskom jeziku Python dana je izvornim tekstom programa 2.1.

```
1 def provjeri_ispravnost_bojanja(obojeni_vrhovi, matrica_povezanosti):
2     for vrh in matrica_povezanosti:
3         for susjedan_vrh in matrica_povezanosti[vrh]:
4             if obojeni_vrhovi[vrh] == obojeni_vrhovi[susjedan_vrh]:
5                 return False
6     return True
7
8 def iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, k, indeks_vrha):
9     # kraj rekurzije
10    if indeks_vrha == len(vrhovi):
11        if provjeri_ispravnost_bojanja(vrhovi, matrica_povezanosti):
12            return vrhovi
13        else:
14            return None
15    for boja in range(k):
16        vrhovi[indeks_vrha] = boja
17        bojanje_vrhova = iscrpno_pretrazivanje(vrhovi,
18                                                matrica_povezanosti,
19                                                k, indeks_vrha + 1)
20        if bojanje_vrhova is not None:
21            return bojanje_vrhova
22    return None
```

Izvorni tekst programa 2.1: Iscrpno pretraživanje

Jedan iterativni pristup pronalaska kromatskog broja je pokušati obojati graf s  $k = n$  boja, gdje je  $n$  broj vrhova. Ako uspijemo pronaći takvo rješenje, pokušavamo s  $k = n - 1$  bojom [2]. Postupak nastavljamo sve do koraka  $m$  : ( $m < n$ ) kada više ne pronalazimo ispravno bojanje. Tada zaključujemo da je  $\chi(G) = k = n - m + 1$ .

```
1 def odredi_kromatski_broj(vrhovi, matrica_povezanosti):
2     ispravno_bojanje = None
3     for k in range(len(vrhovi), 0, -1):
4         bojanje_vrhova = iscrpno_pretrazivanje(vrhovi,
5                                                matrica_povezanosti,
6                                                k, 0)
7         if bojanje_vrhova is not None:
8             return ispravno_bojanje, k + 1
9     ispravno_bojanje = copy.copy(bojanje_vrhova)
10    return ispravno_bojanje, 1
```

Izvorni tekst programa 2.2: Određivanje kromatskog broja

### 2.2.2. Pohlepno pretraživanje

Pohlepan algoritam kreće od vrha s najvećim stupnjem, onim s najviše susjednih vrhova. Njega boja prvom dostupnom bojom. Zatim pokušava obojati sljedeći vrh s najvećim stupnjem, uzimajući prvu boju s kojom može legalno obojati taj vrh, tako da je različite boje od njemu susjednih vrhova. Postupak se ponavlja do zadnjeg vrha, ili do kad više nema dostupnih boja [1]. Složenost ovakvog algoritma je  $O(n^2)$ , jer za svaki vrh mora proći po svim dotad obojanim vrhovima i vidjeti kojim bojama može obojati trenutni vrh. Ovakvo pretraživanje ne daje uvijek rješenje kad ono postoji, no prednost mu je što je manje vremenske složenosti od iscrpnog pretraživanja. Algoritam je dan u nastavku izvornim tekstom programa 2.3

```
1 def pohlepno_pretrazivanje(vrhovi, matrica_povezanosti,
2                             vrhovi_po_prioritetu,
3                             k, indeks):
4     if indeks == len(vrhovi):
5         return vrhovi
6
7     dostupne_boje = [i for i in range(k)]
8     susjedni_vrhovi = matrica_povezanosti[vrhovi_po_prioritetu[indeks]]
9     for i in range(indeks):
10         if vrhovi_po_prioritetu[i] in susjedni_vrhovi:
11             boja_susjeda = vrhovi[vrhovi_po_prioritetu[i]]
12             if boja_susjeda in dostupne_boje:
13                 dostupne_boje.remove(boja_susjeda)
14
15     if len(dostupne_boje) == 0:
16         return None
17
18     vrhovi[vrhovi_po_prioritetu[indeks]] = dostupne_boje[0]
19     return pohlepno_pretrazivanje(vrhovi, matrica_povezanosti,
20                                   vrhovi_po_prioritetu, k,
21                                   indeks + 1)
```

Izvorni tekst programa 2.3: Pohlepno pretraživanje

### 2.2.3. Usmjereno iscrpno pretraživanje

Algoritam usmjerenog iscrpnog pretraživanja radi vrlo slično kao i pohlepno pretraživanje, uz bitnu preinaku da ako ne pronađe legalno bojanje, traži dalje. Algoritam odredi moguće boje za bojanje trenutnog vrha kao i pohlepni algoritam, ali ako ne uspije obojati graf s prvom bojom, proba s drugom mogućom, pa trećom i tako dalje. Ovakav će algoritam uvijek pronaći ispravno bojanje i upravo će to bojanje biti ono s minimalnim brojem boja, odnosno kromatski broj grafa. Algoritam je dan u nastavku izvornim tekstom programa 2.4.

```
1 def usmjerenno_iscrpno_pretrazivanje(vrhovi, matrica_povezanosti,
2                                       vrhovi_po_prioritetu, k, indeks):
3     if indeks == len(vrhovi):
```

```

4         return vrhovi
5
6     dostupne_boje = [i for i in range(k)]
7     susjedni_vrhovi = matrica_povezanosti[vrhovi_po_prioritetu[indeks]]
8     for i in range(indeks):
9         if vrhovi_po_prioritetu[i] in susjedni_vrhovi:
10             boja_susjeda = vrhovi[vrhovi_po_prioritetu[i]]
11             if boja_susjeda in dostupne_boje:
12                 dostupne_boje.remove(boja_susjeda)
13
14     for boja in dostupne_boje:
15         vrhovi[vrhovi_po_prioritetu[indeks]] = boja
16         bojanje = usmjereno_iscrpno_pretrazivanje(vrhovi,
17                                                    matrica_povezanosti,
18                                                    vrhovi_po_prioritetu,
19                                                    k, indeks + 1)
20
21         if bojanje is not None:
22             return bojanje
23         if indeks == 0:
24             return None
25     return None

```

#### Izvorni tekst programa 2.4: Usmjereno iscrpno pretraživanje

Svaki graf s  $n$  vrhova možemo obojati s  $n$  boja, no uvjet na gornju ogradu kromatskog broja možemo postrožiti.

## 2.3. Gornja ograda na kromatski broj

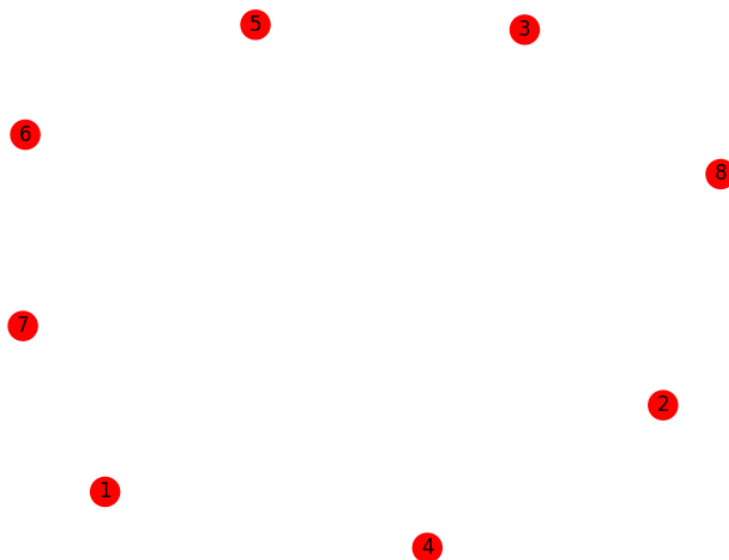
Prema Brookovom teoremu, kromatski broj svakog grafa je maksimalno  $\Delta + 1$ , gdje je  $\Delta$  maksimalan stupanj grafa [1].

Maksimalan stupanj grafa,  $\Delta$ , je najveći broj bridova koji su incidentni s bilo kojim pojedinačnim vrhom u grafu. Incidentnost označava povezanost brida i vrha.



## 2.4. Donja ograda na kromatski broj

Donja ograda na kromatski broj je 1. Ovo se postiže samo za grafove u kojima niti jedna dva vrha nisu međusobno povezana.



**Slika 2.2:** Graf s  $\chi(G) = 1$

## 3. Evolucijski algoritmi

Evolucijski algoritmi su algoritmi koji inspiraciju vuku iz prirode, iz Darwinove teorije evolucije. Probleme rješavaju kroz procese koji emuliraju ponašanje živih bića i sustava u prirodi.

Rješenja problema u evolucijskim algoritmima predstavljaju pojedine jedinke unutar populacije. Populacija je na početku određena slučajno, a kasnije kroz naraštaje, odnosno iteracije algoritma, bolja rješenja preživljavaju i reproduciraju se, dok se lošija rješenja miču, izumiru. Ovime je osigurano da su jedinke novog naraštaja potomci najboljih jedinki prethodnog naraštaja čime se postiže konvergencija prema optimalnom rješenju [9].

Evolucijski algoritmi odlični su kod pronalaženja rješenja za optimizacijske probleme. Kroz naraštaje, pojedina rješenja postaju sve bolja i bliža optimalnom. Bitno je imati na umu da su evolucijski algoritmi podskup heurističkih algoritama, što znači da pronađena rješenja nisu nužno optimalna. Uzrok tome je što se ne pretražuje cijeli prostor već samo njegov podskup koji je ograničen početnom populacijom i parametrima evolucijskog algoritma, određenima na temelju neke heuristike.

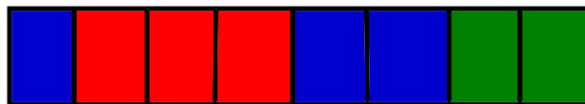
### 3.1. Genetski algoritmi

Genetski algoritmi podskup su evolucijskih algoritama. Jedinke, rješenja, kodirana su u obliku genoma. Iz početne populacije jedinki biraju se one najbolje pomoću funkcije dobrote (eng. *fitness*) ili pomoću funkcije cijene (eng. *cost*) koje će biti roditelji novoj generaciji. Iz genoma odabranih roditelja se pomoću procesa križanja određuju genomi djece. Osim procesa križanja i odabira roditelja, važnu ulogu igraju i mutacije gena na genomu. Proces mutacije događa se nakon križanja roditelja, kada se s određenom vjerojatnošću ili heuristikom mijenjaju neki od gena na genomu. Mutacije su bitne jer se njima proširuje prostor pretraživanja, uvode nove varijacije rješenja te se sprječava ostanak u području lokalnog optimuma rješenja [9].

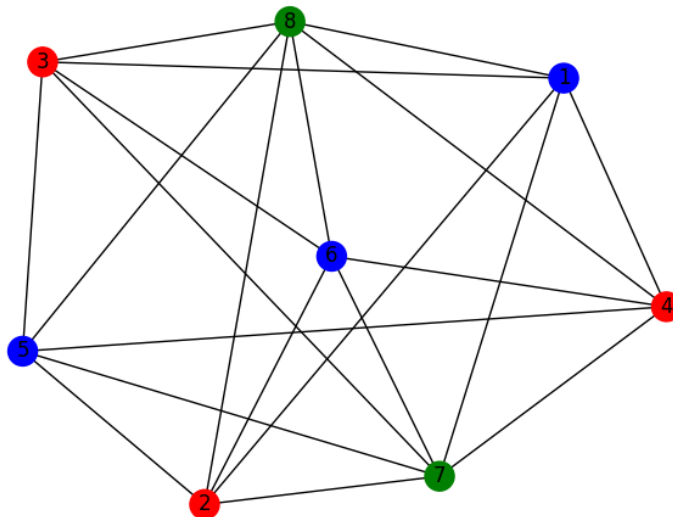
### 3.1.1. Kodiranje genoma

Genom (kromosom) u sklopu genetskih algoritama predstavlja jedan specifičan organizam, odnosno jedno moguće rješenje. Genom se sastoji od vektora gena. Svaki gen može poprimiti određenu vrijednost. Za ispravan rad genetskog algoritma nužno je pravilno kodirati rješenja u obliku genoma.

Kod problema bojanja grafa, genom je vektor od  $n$  elemenata gdje je  $n$  broj vrhova u grafu. Vrijednost svakog gena je iz skupa od  $k$  elemenata, gdje  $k$  predstavlja broj različitih boja [6]. Konačno rješenje je vektor u kojemu su elementi pobožani s  $k$  različitih boja tako da pripadajući spojeni vrhovi nemaju istu vrijednost. Ako su dva vrha spojena i obojana su istom bojom, takvu jedniku ne smatramo rješenjem problema. Na slici 3.1 prikazan je genom jedne jedinke koja je optimalno rješenje bojanja grafa na slici 3.2.



Slika 3.1: Vektor gena - genom,  $k = 3$ ,  $n = 8$



Slika 3.2: Pripadajući obojani graf,  $\chi(G) = 3$

### 3.1.2. Funkcija cijene

Određivanje koliko je koja jedinka dobra određuje se pomoću funkcije cijene. Ona mjeri koliko je trenutno rješenje udaljeno od ciljnog. Što je cijena jedinke manja, to je

rješenje bolje. Kada cijene neke jedinke dosegne 0, pronađeno je ispravno rješenje.

Suprotno funkciji cijene je funkcija dobrote, prema kojoj bolje jedinke imaju veću vrijednost.

U primjeru bojanja grafova, ciljno rješenje ne smije imati niti jedna dva susjedna vrha obojana istom bojom. Za svaka dva krivo spojena vrha, funkcija cijene raste za jedan [2]. Algoritam je dan izvornim tekstom programa 3.1.

```
1 def funkcija_cijene(vrhovi, matrica_povezanosti):
2     cijena = 0
3     for vrh in matrica_povezanosti:
4         for susjedan_vrh in matrica_povezanosti[vrh]:
5             if vrhovi[vrh] == vrhovi[susjedan_vrh]:
6                 cijena = cijena + 1
7     return cijena / 2
```

**Izvorni tekst programa 3.1:** Funkcija cijene

### 3.1.3. Određivanje roditelja

Roditelji se iz populacije biraju na temelju njihove izračunate cijene. Jedinke s manjom cijenom imaju prednost pri reproduciranju nad jedinkama s većom cijenom. Postoji mnogo različitih strategija odabira roditelja u sklopu genetskih algoritama, a neke od njih su:

- slučajna selekcija,
- selekcija stabilnog stanja,
- jednostavna selekcija i
- rang selekcija.

#### Slučajna selekcija

Kao naivnu metodu selekcije roditelja koristi se slučajna selekcija. Iz populacije se slučajnim odabirom izabiru dvije jedinke kao roditelji novoj jedinci. Ova metoda je manje uspješna od ostalih metoda selekcije i zato nije pretjerano korištena. Razlog manjoj uspješnosti je što cijena izračunata za pojedine jedinke nimalo ne utječe na izbor jedinke kao roditelja. Bolje jedinke nemaju nikakvu prednost kod reprodukcije što znači da rješenja sporije konvergiraju, ili uopće ne konvergiraju k optimumu.

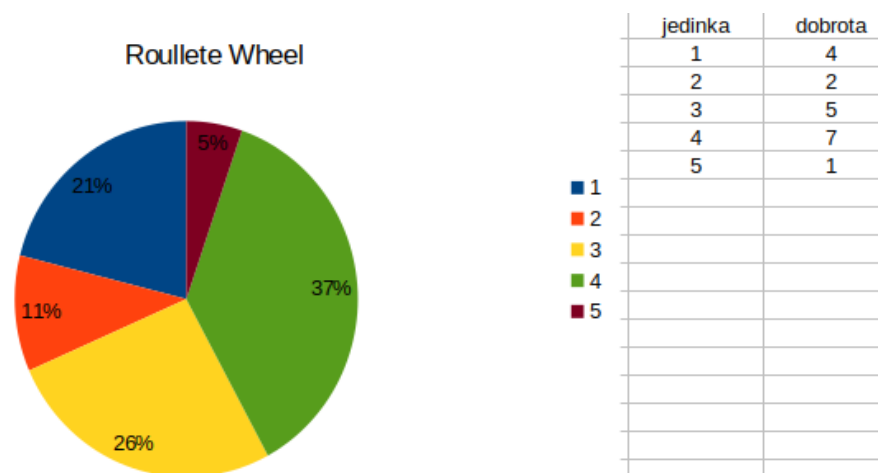
#### Selekcija stabilnog stanja

Kod selekcije stabilnog stanja (*eng. Steady State Selection*), među populacijom se biraju dvije jedinke s najboljom dobrotom. Iz njih se stvaraju dvije nove jedinke,

njihova djeca, i na njima se provodi mutacija. Zatim se izračunava dobrotu djece i djeca se dodaju populaciji. Iz populacije se tada izbacuju dvije jedinke s najmanjom dobrotom. Ovakav pristup odabira roditelja i njihove djece proizvodi svega dvije nove jedinke po generaciji, ali su one uvijek potomci dvoje roditelja s najboljom dobrotom.

### Jednostavna selekcija

Nakon što su izračunate cijene za sve jedinke u generaciji, svakoj se pridjeljuje udio na kotaču ruleta (*eng. roulette wheel*). Udio je proporcionalan cijenama jedinki; one s nižom cijenom (većom dobrotom) će imati veći udio na kotaču ruleta. Time se osigurava da bolje jedinke imaju veću vjerojatnost da budu odabrane kao roditelji sljedećoj generaciji. Rješenje se tako brže približava optimumu jer će djeca naslijediti gene od uspješnijih roditelja.

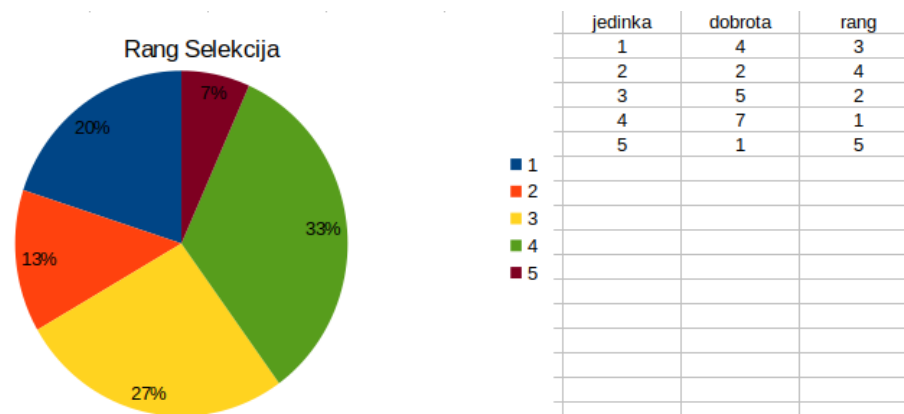


**Slika 3.3:** Primjer kotača ruleta

Nedostatak ove metode je što se može dogoditi da neka jedinka bude nesrazmjerno dobra u odnosu na druge jedinke, ali vodi samo lokalnom optimumu. Algoritam će tada odabrati takvu jedinku kao roditelja i sva njezina djeca će posljedično biti relativno uspješna, ali neće konvergirati optimalnom rješenju. Još jedan veliki nedostatak je i često biranje istih roditelja ako su oni znatno veće dobrote od ostatka populacije čime se povećava vjerojatnost generiranja duplikata.

### Rang selekcija

Rang selekcija po principu je slična jednostavnoj selekciji, ali se umjesto proporcionalnog udjela vjerojatnosti prema cijeni, jedinke prvo rangiraju prema cijeni, a udjeli na kotaču se zatim dodjeljuju proporcionalno rangu.



**Slika 3.4:** Primjer rang selekcije

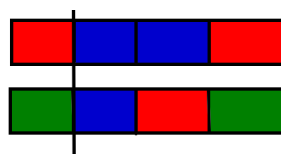
Na taj način svaki rang uvijek ima jednaku vjerojatnost biti izabran. Bolje jedinke imaju veću vjerojatnost biti izabrane kao roditelji sljedećoj generaciji, a ne može se dogoditi da neke jedinke koje vode do lokalnog optimuma imaju preveliku prednost nad ostalim jedinkama te da zbog toga algoritam ne konvergira u globalno rješenje.

### 3.1.4. Križanje

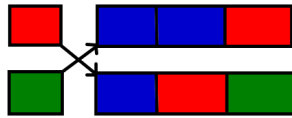
Križanje je postupak kojim se iz izabranih genoma roditelja dobiva genom potomka. Potomak se sastoji od pomješanog genoma svojih roditelja. Križanje se izvodi tako da se na slučajnom mjestu prerežu genomi oba roditelja u točki prijeloma, zamijene se prvi dijelovi genoma i dobiju se dvije nove jedinke; prva jedinka s prvim dijelom genoma od prvog roditelja i drugim dijelom genoma od drugog roditelja, i druga jedinka s prvim dijelom genoma od drugog roditelja i drugim dijelom genoma od prvog roditelja. Osim križanja sa samo jednom točkom prijeloma, postoji i križanje s više točaka prijeloma.

Ako smo koristili dobru metodu za odabir roditelja, djeca će se sastojati od pomiješanih gena uspješnih roditelja, približiti će se optimumu. Često se dogodi da djeca imaju manju vrijednost funkcije dobrote od svojih roditelja, no prosječno gledano na cijeloj populaciji, dobrota djece se povećava.

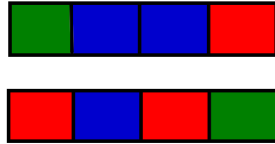
Postupak križanja vizualno je prikazan na slikama 3.5, 3.6, 3.7.



**Slika 3.5:** Kromosomi roditelja



**Slika 3.6:** Križanje



**Slika 3.7:** Kromosomi djece

### 3.1.5. Mutacije

Mutacije su promjene gena na kromosomima jedinki. One su vrlo bitne u genetskim algoritmima jer omogućuju da se pretražuje rješenje na širem prostoru. Ako ne bi bilo mutacije na genima, geni početne populacije bi u potpunosti određivali mogući prostor pretraživanja. Sve nove jedinke mogle bi biti jedino kombinacije početnog skupa gena. Mutacije također ubrzavaju pretraživanje prostora, jer će neke jedinke "odskočiti" od trenutne populacije, uvesti novinu u skup jedinki. Ako su mutacije nepovoljne, takve jedinke imati će manju dobrotu, neće se uspješno razmnožavati i takva mutacija će izumrijeti. Pozitivne će mutacije s druge strane, povećati dobrotu jedinke i tako povećati vjerojatnost reprodukcije jedinke s takvim genom. Gen će se dalje propagirati i dovesti cijelu populaciju bliže optimumu.

Uobičajeno se mutacije ostvaruju tako da svaki gen ima nezavisnu vjerojatnost mutacije. Mutacija se provodi nakon križanja. Kod nekih specifičnih problema, poput bojanja grafa, bolje rezultate daju usmjere mutacije na genima koje ovise o trenutnom stanju svih gena jedinke.

Slika 3.8 prikazuje kromosom prije mutacije, a slika 3.9 kromosom poslije mutacije na trećem i osmom genu.



**Slika 3.8:** Kromosom prije mutacije



Slika 3.9: Kromosom poslije mutacije

## 3.2. Programsko ostvarenje genetskog algoritma

Nakon slučajnog izbora početne populacije, nad svakom jedinkom izračunava se njezina dobrota. Na temelju dobrote se odabiru roditelji nove generacije. Roditelji se križaju, a zatim se nad dobivenom djecom izvode mutacije. Djeca zamjenjuju roditelje u populaciji. Ovaj se postupak ponavlja sve dok se ne postigne željena dobrota, pronade optimalno rješenje problema ili se ne pređe određen broj iteracija. Genetski algoritam dan je izvornim tekstom programa 3.2.

```

1 def generiraj_pocetnu_populaciju(velicina_populacije):
2     pocetna_populacija = []
3     for _ in range(velicina_populacije):
4         pocetna_populacija.append(generiraj_slucajnu_jedinku())
5     return pocetna_populacija
6
7 def izaberi_roditelje(populacija):
8     roditelji = []
9     for _ in range(len(populacija)):
10        roditelji.append(rang_selekcija())
11    return roditelji
12
13 def krizanje(roditelji):
14    djeca = []
15    for i in range(0, len(roditelji), 2):
16        dijete_1, dijete_2 = generiraj_djecu(roditelji[i], roditelji[i+1])
17        djeca.append(dijete_1)
18        djeca.append(dijete_2)
19    return djeca
20
21 def mutiraj(djeca):
22    for dijete in djeca:
23        for gen in dijete.kromosom:
24            if slucajan_dogadaj():
25                gen = slucajan_gen()
26    return djeca
27
28 def genetski_algoritam(broj_iteracija, limit_dobrote, velicina_populacije):
29    populacija = generiraj_pocetnu_populaciju(velicina_populacije)
30    for _ in range(broj_iteracija):
31        najbolja_dobrota, najbolja_jedinka = maksimalna_dobrota(populacija)
32        if najbolja_dobrota >= limit_dobrote:
33            return True, najbolja_jedinka
34
35        roditelji = izaberi_roditelje(populacija)
36        djeca = krizanje(roditelji)
37        populacija = mutiraj(djeca)
38
39    return False, najbolja_jedinka

```

Izvorni tekst programa 3.2: Genetski algoritam



## 4. Korištene knjižnice programa i podaci

### 4.1. Knjižnica programa PyGAD

PyGAD je knjižnica programa otvorenog koda za programski jezik Python pomoću koje se vrlo jednostavno mogu izgraditi modeli genetskih algoritama. Knjižnica dolazi s mnoštvom ugrađenih metoda, specifičnima za genetske algoritme. PyGAD je vrlo fleksibilan i dozvoljava definiranje vlastitih funkcija izračuna dobrote, izbora roditelja, križanja i mutacije.

#### 4.1.1. Instalacija

Instalacija knjižnice vrši se preko PIP-a sljedećom naredbom:

```
$ pip install pygad
```

### 4.2. Paket NetworkX

NetworkX je Python paket za stvaranje, rukovanje i proučavanje strukture, dinamike i funkcija složenih mreža. U sklopu ovog projekta korišten je kod vizualizacije grafova te njihovog bojanja.

#### 4.2.1. Instalacija

Instalacija paketa obavlja se preko PIP-a sljedećom naredbom:

```
$ pip install networkx[default]
```

### 4.3. Web stranica House of Graphs

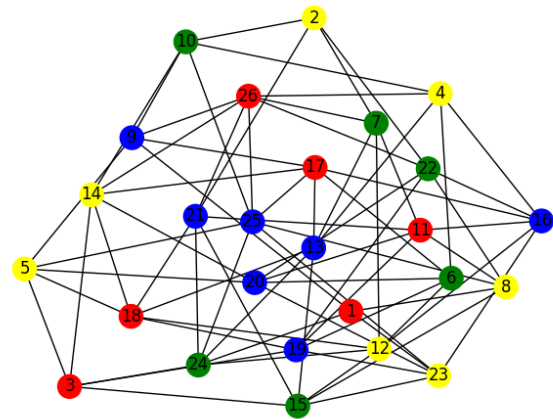
House of Graphs [4] je Web-stranica koja sadržava bazu grafova s karakteristikama koja dozvoljava njihovo besplatno preuzimanje. House of Graphs razvijen je u suradnji sveučilišta Ghent i sveučilišta KU Leuven.

U okviru projekta preuzeti su grafovi s brojem vrhova između 1 i 250. Za svaki od grafova definirana je matrica susjedstva kao i kromatski broj grafa. Na taj se način rješenje dobiveno genetskim algoritmom moglo usporediti sa stvarnom vrijednošću kromatskog broja grafa. Za neke od kompleksnijih grafova nije izračunat kromatski broj (*Chromatic number: Computation timeout*) te su takvi grafovi preskočeni kod izračuna točnosti rada genetskog algoritma.

```
1: 8 9 23 24
2: 7 10 21 22
3: 5 14 15 19 24
4: 6 10 13 16 26
5: 3 9 18 20 25
6: 4 12 17 19 20
7: 2 11 12 13 26
8: 1 11 15 22 25
9: 1 5 10 17 26
10: 2 4 9 14 25
11: 7 8 16 19 20 21
12: 6 7 15 16 18 24
13: 4 7 15 17 18 24
14: 3 10 17 18 20 26
15: 3 8 12 13 21 23
16: 4 11 12 17 22 23
17: 6 9 13 14 16 25
18: 5 12 13 14 19 21
19: 3 6 11 18 22 23
20: 5 6 11 14 22 23
21: 2 11 15 18 24 26
22: 2 8 16 19 20 26
23: 1 15 16 19 20 25
24: 1 3 12 13 21 25
25: 5 8 10 17 23 24 26
26: 4 7 9 14 21 22 25

Chromatic Number: 4
Maximum Degree: 7
Number of Edges: 73
Number of Vertices: 26
```

Slika 4.1: Definicija grafa s 26 vrhova



Slika 4.2: Bojanje grafa s 26 vrhova s 4 boje

Definicija grafa preuzetog s House of Graphs Web-stranice s 26 vrhova prikazana je na slici 4.1, a ispravno bojanje za taj graf prikazano je na slici 4.2.

## 5. Praktični dio

Projekt je pisan u programskom jeziku Python. Python je izabran zbog jednostavnosti korištenja i dobre potpore rješavanju problema u obliku slobodno dostupnih knjižnica programa.

### 5.1. Određivanje dobrote jedinki

Funkcija cijene korištena u sklopu projekta opisana je u potpoglavlju 3.1.2. Funkcija je prilagođena kako bi bila kompatibilna s PyGAD knjižnicom programa koja zahtjeva korištenje funkcije dobrote. Kod određivanja ispravnog bojanja grafa prirodno je koristiti funkciju cijene, kao mjeru broja vrhova obojanih krivom bojom. Ispravno bojanje grafa određeno je maksimumom funkcije dobrote koji je nula (niti jedan vrh nije obojan istom bojom kao i njegov susjed), a za svaki krivo obojani vrh dobrotu pada za jedan. Lošije jedinke tako imaju manju dobrotu.

### 5.2. Određivanje roditelja

Roditelji su određivani na tri načina; slučajnim odabirom, rang selekcijom i selekcijom stabilnog stanja. Slučajan odabir izabran je kako bi se pokazali nedostatci te metode, ali i dao uvid u rad ostatka algoritma i odabira ostalih parametara. Rang selekcija i selekcija stabilnog stanja korištene su kako bi se postigli optimalni rezultati.

### 5.3. Križanje

Za metodu križanja je korišteno križanje s jednom točkom prekida.

## 5.4. Mutacije

Najbitniju ulogu u samom algoritmu imale su mutacije. Na početku su se koristile slučajne mutacije s vjerojatnošću mutiranja na svakom genu od 20%. Ovaj relativno visok postotak uzet je kako bi se populacija brže širila po prostoru pretraživanja te ne bi zapela u lokalnim optimumima [2]. Problem je što geni unutar svake jedinke nisu nezavisni, pa se na slijepo pogađanje nisu dobili zadovoljavajući rezultati. Iz tog razloga isprobane su i neke složenije funkcije mutacije.

### 5.4.1. Slučajna mutacija na pogrešno obojanim vrhovima

Prvi pristup mutaciji slučajno je mijenjao vrijednosti samo onih gena koji su obojani istom bojom kao i njihovi susjedi. Na ovaj način su vrhovi koji su bili dobro obojani ostali netaknuti. Rješenje na ovaj način konvergira prema pravom jer se jednom pronađena dobra bojanja ne mijenjaju. Slučajna mutacija na krivim vrhovima ne uzima u obzir njihove susjede kod odabira nove boje. Zbog toga se prostor pretraživanja pretražuje na relativno širokom području [3]. Algoritam je dan izvornim tekstom programa 5.1.

```
1 def slucajna_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k):
2     for dijete in djeca:
3         for vrh, boja_vrha in enumerate(dijete):
4             potrebno_novo_bojanje = False
5             for susjedan_vrh in matrica_susjedstva[vrh]:
6                 if boja_vrha == dijete[susjedan_vrh]:
7                     potrebno_novo_bojanje = True
8                     break
9             if not potrebno_novo_bojanje:
10                continue
11            nova_boja = random.randint(k)
12            dijete[vrh] = nova_boja
13    return djeca
```

**Izvorni tekst programa 5.1:** Slučajna mutacija na pogrešno obojanim vrhovima

### 5.4.2. Ciljana mutacija na pogrešno obojanim vrhovima

U ovoj metodi, slično kao i u potpoglavlju 5.4.1, mutacija se odvija samo na vrhovima koji su obojani jednako kao i neki od njihovih susjeda. Mutacija na takvim vrhovima nije slučajna, već se gledaju susjedi takvog vrha i za njegovo ponovno bojanje bira se boja kojom njegovi susjedi nisu obojani. Ako takva boja ne postoji, slučajnim odabirom izabire se neka od boja iz skupa svih boja. Ovakva implementacija osigurava da dobro obojani vrhovi ostanu netaknuti, a oni krivo obojani ciljano promijene svoju

boju ne narušavajući ispravna bojanja ostalih vrhova ako to mogu [3]. Algoritam je dan izvornim tekstom programa 5.2.

```

1 def ciljana_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k):
2     for dijete in djeca:
3         for vrh, boja_vrha in enumerate(dijete):
4             potrebno_novo_bojanje = False
5             boje_susjeda = set()
6             for susjedan_vrh in matrica_susjedstva[vrh]:
7                 boje_susjeda.add(dijete[susjedan_vrh])
8                 if boja_vrha == dijete[susjedan_vrh]:
9                     potrebno_novo_bojanje = True
10
11             if not potrebno_novo_bojanje:
12                 continue
13
14             if len(boje_susjeda) == k:
15                 nova_boja = random.randint(k)
16             else:
17                 dostupne_boje = []
18                 for boja in range(k):
19                     if boja not in boje_susjeda:
20                         dostupne_boje.append(boja)
21                 nova_boja = random.choice(dostupne_boje)
22                 dijete[vrh] = nova_boja
23     return djeca

```

**Izvorni tekst programa 5.2:** Ciljana mutacija na pogrešno obojanim vrhovima

### 5.4.3. Kombinacija slučajne i ciljane mutacije na pogrešno obojanim vrhovima

Ovisno o dobroti najbolje jedinke iz prethodne populacije, ovaj algoritam bira jednu od mutacija iz potpoglavlja 5.4.1 i potpoglavlja 5.4.2. Ako je dobrota najbolje jedinke blizu optimalne, nula, tada se koristi *Slučajna mutacija na pogrešno obojanim vrhovima* jer se želi izbjeći zaustavljanje u lokalnom optimumu. Kada je dobrota najbolje jedinke daleko od optimalne, koristi se *Ciljana mutacija na pogrešno obojanim vrhovima* jer se želi usmjerenije pretražiti prostor i na taj način stići do optimuma [3]. Algoritam je dan izvornim tekstom programa 5.3.

```

1 def kombinacija_slucajne_i_ciljane_mutacije(djeca,
2                                             matrica_susjedstva, k,
3                                             najbolja_dobrota, granica):
4     if najbolja_dobrota >= granica:
5         return slucajna_mutacija_na_krivim_vrhovima(djeca,
6                                                         matrica_susjedstva,
7                                                         k)
8     else:
9         return ciljana_mutacija_na_krivim_vrhovima(djeca,
10                                                         matrica_susjedstva,
11                                                         k)

```

**Izvorni tekst programa 5.3:** Kombinacija slučajne i ciljane mutacije na pogrešno obojanim vrhovima

## 5.5. Ostali parametri

Ostali korišteni parametri su:

- broj generacija,
- broj jedinki u generaciji,
- elitizam i
- broj gena po jedinci.

Njihove vrijednosti dane su tablicom 5.1.

**Tablica 5.1:** Ostali parametri genetskog algoritma

Parametar	Vrijednost
broj generacija	1000
jedinke po generaciji	50
elitizam	istina
broj gena	$n$

Broj generacija i broj jedinki po generaciji odabrani su kao kompromis između vremena izvođenja algoritma i uspješnosti algoritma [3]. Broj gena odgovara broju vrhova,  $n$ , jer se svaki vrh kodira svojim genom.

Elitizam je strategija evolucijskih algoritama kojom se najbolje jedinke, u kontekstu ovog projekta jedna, automatski prenose u sljedeću generaciju. Na taj način se čuvaju uspješne jedinke, imaju više potomaka i algoritam brže konvergira optimalnom rješenju.

## 5.6. Postupak određivanja kromatskog broja

Nakon parsiranja grafova preuzetih s House of Graphs (potpoglavlje 4.3), za svaki od grafova određuje se kromatski broj. Početna vrijednost  $k$  je postavljena na  $\Delta + 1$ . Ako se genetskim algoritmom pronašlo ispravno bojanje sa zadanim  $k$  (dobrota neke jedinke je jednaka nuli),  $k$  se smanjuje za 1, a najbolja jedinka se pohranjuje. Algoritam prekida s radom u dva slučaja; ako je pronašao ispravno bojanje s  $k = \chi(G)$  (točan kromatski broj je zadan zajedno s definicijom grafa) ili algoritam nije pronašao ispravno bojanje za trenutni  $k$ . U potonjem slučaju algoritam vraća  $k + 1$  kao zadnji ispravan broj boja. Algoritam je dan izvornim tekstom programa 5.4.

```

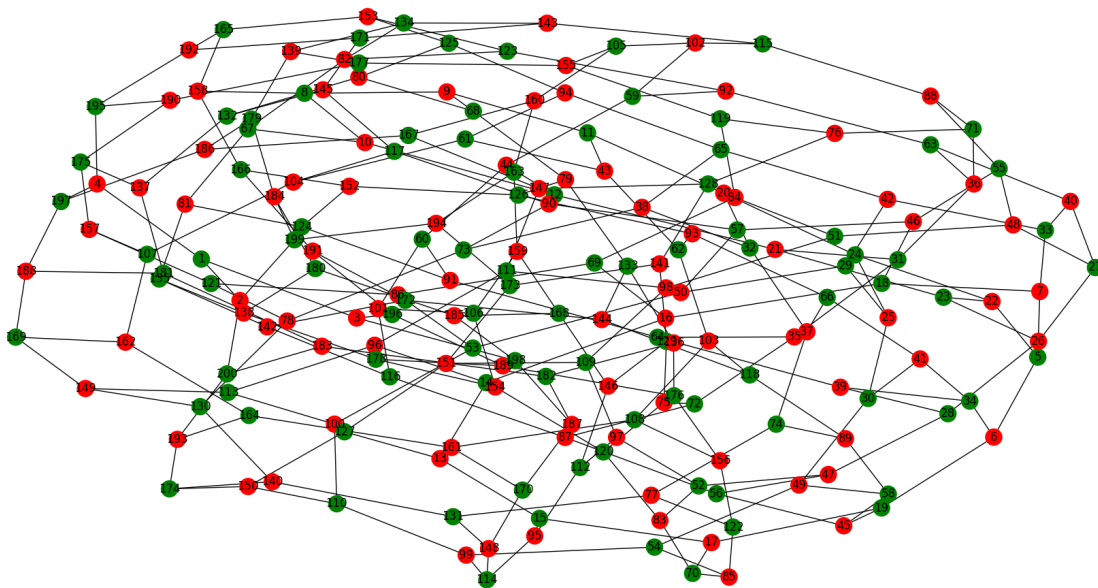
1 def odredi_kromatski_broj(graf, parametri_genetskog_algoritma):
2     delta = graf.maksimalan_stupanj_grafa
3     kromatski_broj = graf.kromatski_broj
4     k = delta + 1
5     prethodno_rjesenje = None
6     while(k >= kromatski_broj):
7         algoritam = genetski_algoritam(parametri_genetskog_algoritma,
8                                         graf, k)
9         rjesenje = algoritam.run()
10        if rjesenje is None:
11            return k + 1, prethodno_rjesenje
12        prethodno_rjesenje = rjesenje
13        k = k - 1
14    return k + 1, prethodno_rjesenje

```

**Izvorni tekst programa 5.4:** Određivanje kromatskog broja genetskim algoritmom

## 5.7. Vizualizacija rezultata

Podaci o uspješnom pronalasku kromatskog broja se pohranjuju, tako da je na kraju izvođenja programa moguće odrediti postotak uspješnih pronalazaka kromatskog broja. Uz to, program je u mogućnosti vizualizirati dobivene rezultate crtanjem obojenog grafa.



**Slika 5.1:** Bojanje bipartitnog grafa s 200 vrhova

## 6. Rezultati praktičnog dijela rada

Ispitane su različite kombinacije parametara i funkcija s ciljem pronalaska optimalnih parametara. Testiranje se provodilo na dva skupa podataka; 200 grafova s rasponom broja vrhova od 1 do 250, te 1066 grafova s rasponom broja vrhova od 150 do 199.

Kao mjera uspješnosti, korištena je *točnost*, omjer broja točno određenih kromatskih brojeva i ukupnog broja grafova za koje se određivao kromatski broj.

### 6.1. Manji skup podataka

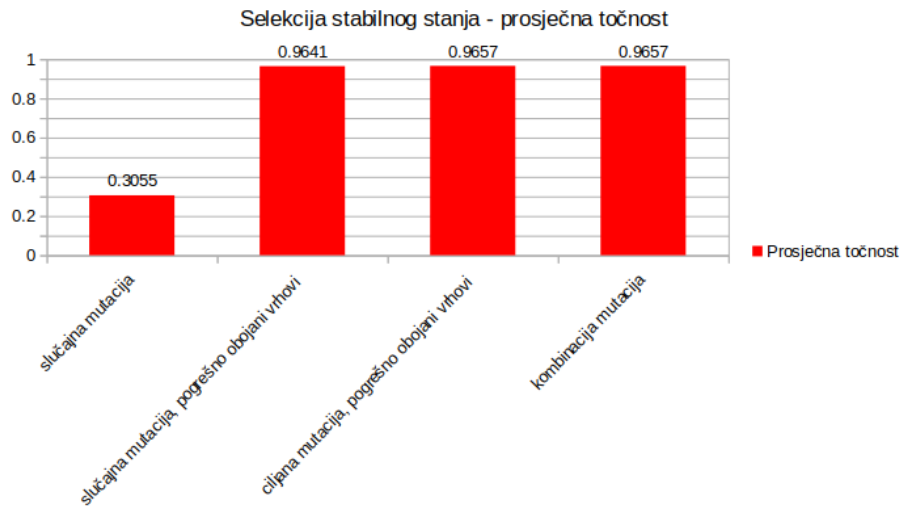
Na manjem skupu podataka od 200 grafova željelo se ustanoviti koliko su algoritmi dobri u pronalaženju rješenja na grafovima s većim rasponom broja vrhova. Pretpostavka je bila da će sve varijacije algoritma biti relativno dobre kod pronalaženja kromatskog broja za jednostavne grafove (mali broj vrhova i bridova), ali da će algoritmi koji koriste slučajan odabir biti lošiji.

Rezultati su podijeljeni po strategiji odabira roditelja, a zatim po strategiji mutacije. Za svaku korištenu kombinaciju strategije odabira roditelja i strategije mutacije postupak je ponovljen 10 puta. Razlog ovako malog broja ponavljanja je bio nedostatak računalne snage.

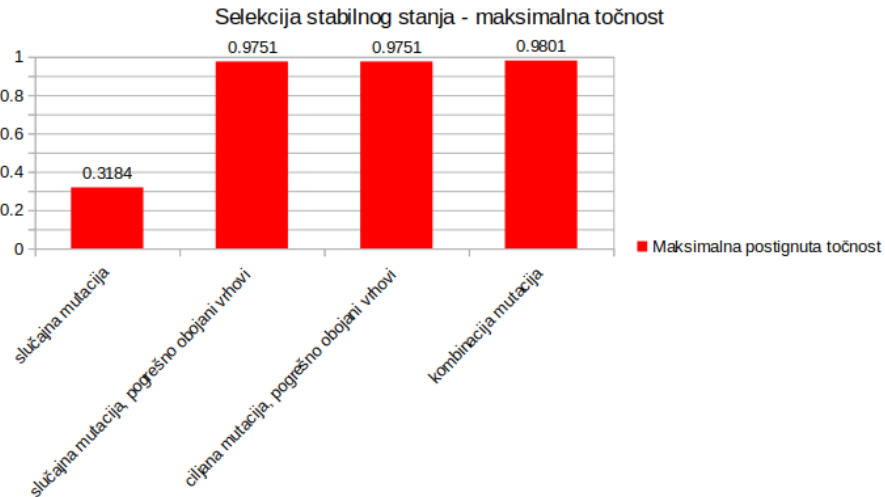


## Selekcija stabilnog stanja

U nastavku su prikazani rezultati algoritama koji koriste selekciju stabilnog stanja za odabir roditelja, uz korištenje različitih strategija mutacije.



**Slika 6.1:** Grafikon prosječne točnosti algoritama selekcije stabilnog stanja

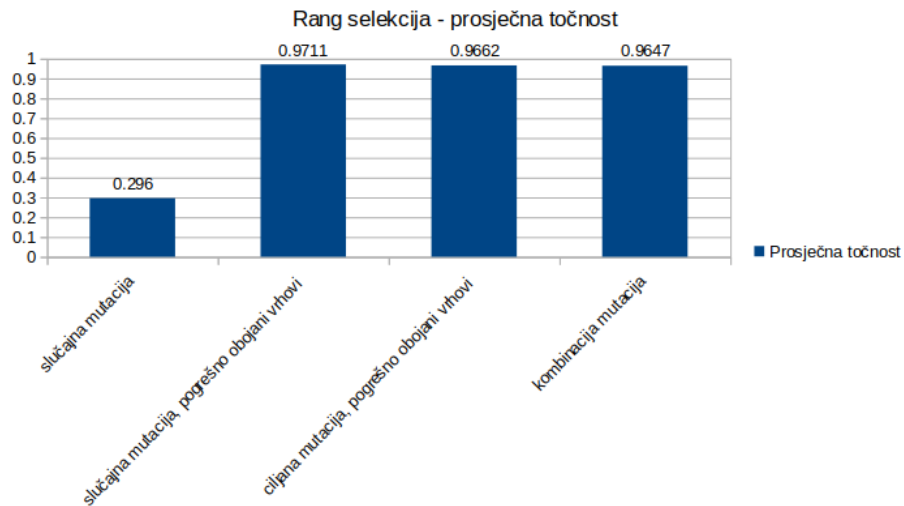


**Slika 6.2:** Grafikon maksimalne točnosti algoritama selekcije stabilnog stanja

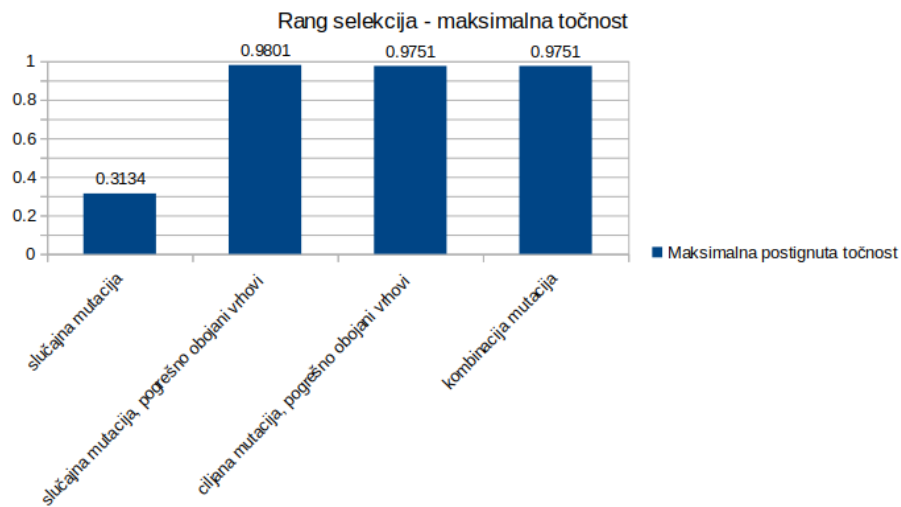
Iz grafikona sa slika 6.1 i 6.2 vidljivo je da prosječna točnost kao i maksimalna postignuta točnost daleko najlošija kod algoritma sa slučajnom mutacijom. Ostale tri strategije čine se podjednako uspješnim, s prosječnom točnosti od oko 96%. Slučajna mutacija na krivim vrhovima ima malo nižu točnost od preostale dvije strategije mutacije.

## Rang selekcija

U nastavku su prikazani rezultati algoritama koji koriste rang selekciju za odabir roditelja, uz korištenje različitih strategija mutacije.



**Slika 6.3:** Grafikon prosječne točnosti algoritama rang selekcije

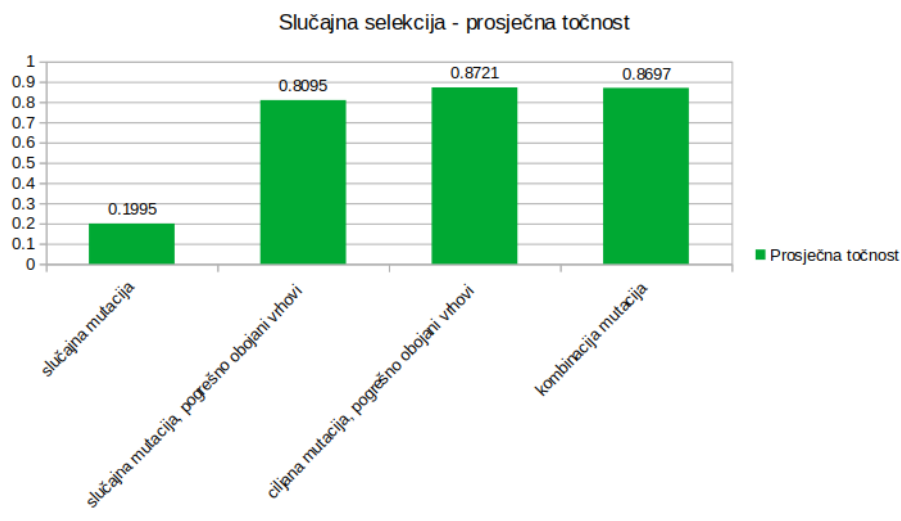


**Slika 6.4:** Grafikon maksimalne točnosti algoritama rang selekcije

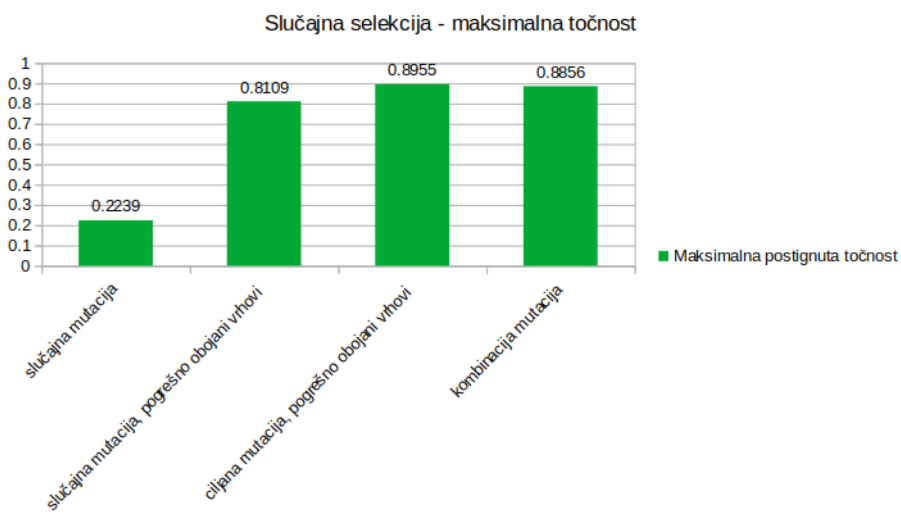
Kao i kod selekcije stabilnog stanja, iz grafikona sa slika 6.3 i 6.4 vidljivo je da je slučajna mutacija postigla najlošije rezultate. Ostale tri varijacije su podjednako uspješne, s prosječnom točnosti od preko 96%.

## Slučajna selekcija

U nastavku su prikazani rezultati algoritama koji koriste slučajnu selekciju za odabir roditelja, uz korištenje različitih strategija mutacije.



Slika 6.5: Grafikon prosječne točnosti algoritama slučajne selekcije

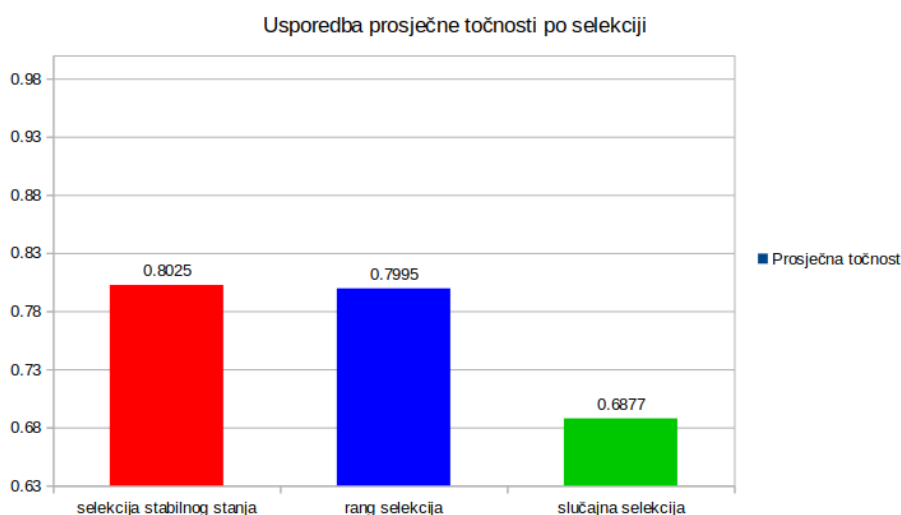


Slika 6.6: Grafikon maksimalne točnosti algoritama slučajne selekcije

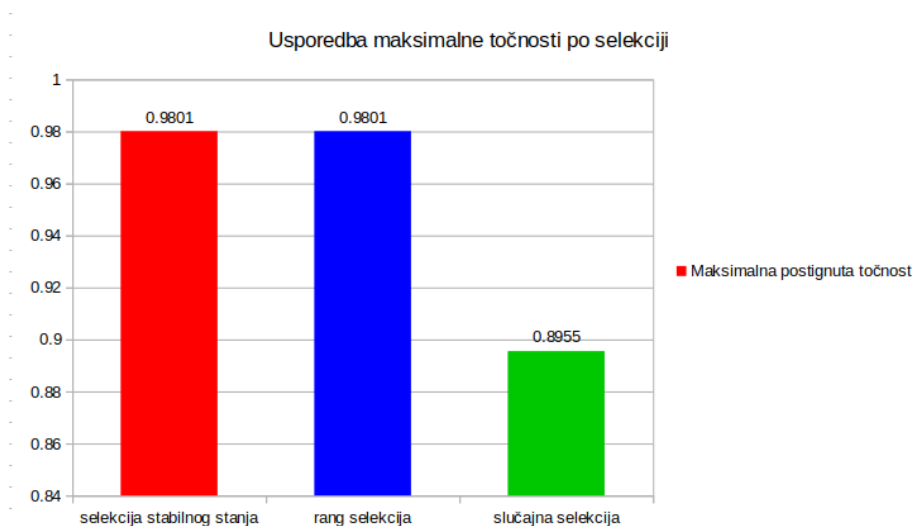
Iz grafikona sa slika 6.5 i 6.6 vidljivo je da je kod slučajne selekcije točnost manja za sve strategije mutacija u odnosu na rang selekciju i selekciju stabilnog stanja. Slučajna mutacija ima najlošiju prosječnu točnost od samo 20%.

### 6.1.1. Usporedba rezultata po strategiji selekcije roditelja

U nastavku su prikazani rezultati grupirani po strategiji selekcije roditelja. Prosječna točnost selekcije računata je kao prosjek prosječnih točnosti za svaku strategiju mutacije, dok je maksimalna točnost računata kao maksimalna vrijednost maksimalnih postignutih točnosti za svaku strategiju mutacije.



**Slika 6.7:** Grafikon prosječne točnosti po strategijama selekcije roditelja



**Slika 6.8:** Grafikon maksimalne postignute točnosti po strategijama selekcije roditelja

Iz grafikona sa slika 6.7 i 6.8 vidljivo je da je selekcija stabilnog stanja postigla najbolju prosječnu točnost, dok je slučajna selekcija najmanje uspješna. Rang selekcija je vrlo blizu selekciji stabilnog stanja i po prosječnoj točnosti i po maksimalno postignutoj točnosti.

## 6.2. Veći skup podataka

Na skupu podataka od 1066 grafova s brojem vrhova između 150 i 199 testirane su samo varijacije algoritama koje su ostvarile najbolje rezultate na skupu od 200 podataka. Za svaku korištenu kombinaciju strategije odabira roditelja i strategije mutacije postupak je ponovljen 6 puta. U nastavku su prikazani najbolji postignuti rezultati.

### Selekcija stabilnog stanja s kombiniranom funkcijom mutacije

```
BEST SCORE
[CORRECT]:    1023 / 1066 = (95.96622889305816%)
[ERROR == 1]: 43 / 1066 = (4.033771106941838%)
[ERROR == 2]: 0 / 1066 = (0.0%)
[ERROR == 3]: 0 / 1066 = (0.0%)
[ERROR >= 4]: 0 / 1066 = (0.0%)
```

**Slika 6.9:** Selekcija stabilnog stanja s kombiniranom funkcijom mutacije

Iz rezultata sa slike 6.9 vidljivo je da je algoritam postigao točnost od skoro 96%. U preostalim slučajeva pogriješio je samo za jednu boju; bojanje koje je pronašao bilo je za jednu boju veće od stvarnog kromatskog broja.

### Rang selekcija s kombiniranom funkcijom mutacije

```
BEST SCORE
[CORRECT]:    1015 / 1066 = (95.21575984990619%)
[ERROR == 1]: 51 / 1066 = (4.784240150093809%)
[ERROR == 2]: 0 / 1066 = (0.0%)
[ERROR == 3]: 0 / 1066 = (0.0%)
[ERROR >= 4]: 0 / 1066 = (0.0%)
```

**Slika 6.10:** Rang selekcija s kombiniranom funkcijom mutacije

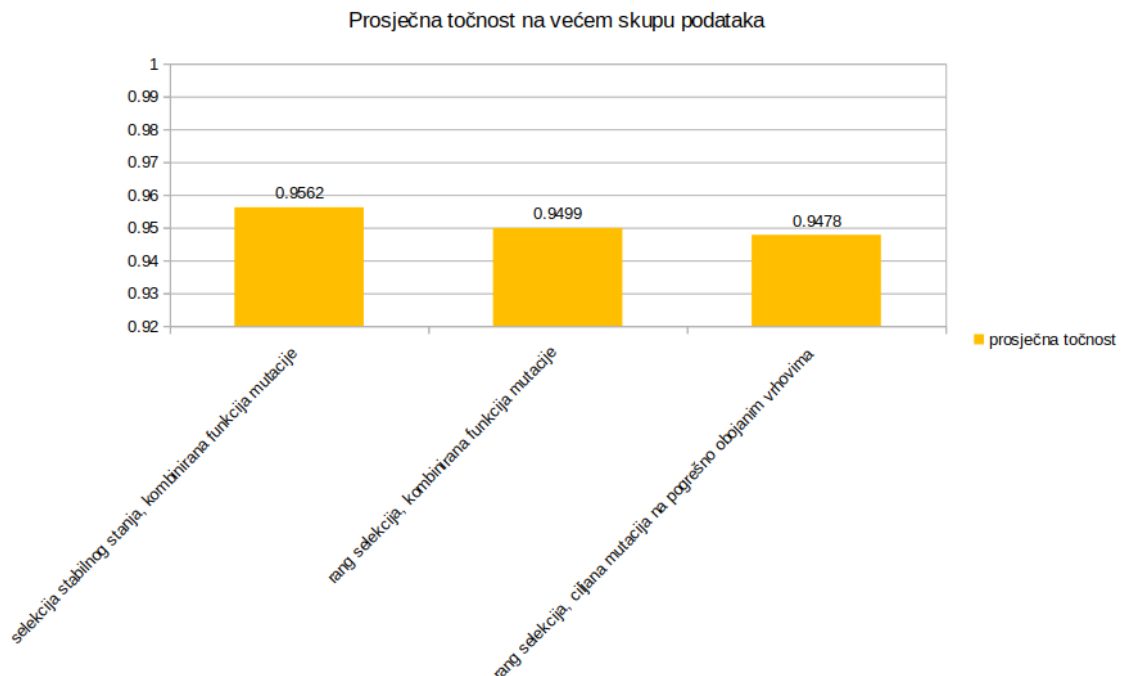
Iz rezultata sa slike 6.10 vidljivo je da je algoritam postigao točnost od oko 95%. U preostalih 5% slučajeva pogriješio je samo za jednu boju.

## Rang selekcija s ciljanom mutacijom krivih vrhova

```
BEST SCORE  
[CORRECT]:      1016 / 1066 = (95.30956848030019%)  
[ERROR == 1]:   50 / 1066 = (4.690431519699812%)  
[ERROR == 2]:    0 / 1066 = (0.0%)  
[ERROR == 3]:    0 / 1066 = (0.0%)  
[ERROR >= 4]:   0 / 1066 = (0.0%)
```

**Slika 6.11:** Rang selekcija s ciljanom mutacijom krivih vrhova

Iz rezultata sa slike 6.11 vidljivo je da je algoritam postigao točnost od oko 95%. U preostalih 5% slučajeva pogriješio je samo za jednu boju.



**Slika 6.12:** Grafikon točnosti algoritama na većem skupu podataka

Iz grafikona sa slike 6.12 vidljivo je da su prosječne točnosti vrlo blizu. Selekcija stabilnog stanja s kombinacijom funkcija mutacija postigla je najbolji rezultat, a rang selekcija s ciljanom mutacijom na pogrešno obojanim vrhovima najlošiji.

## 7. Zaključak

Dobiveni rezultati ukazuju na uspješnost korištenja genetskih algoritama za rješavanje problema bojanja grafa. Iako nemaju 100%-tnu preciznost određivanja kromatskog broja, u velikoj većini slučajeva ne griješe za više od jedne boje.

Kao najbolja kombinacija parametara pokazala se selekcija stabilnog stanja kao strategija odabira roditelja i kombinirana funkcija mutacije kao strategija mutacije (slike 6.8, 6.12). Rang selekcija s kombiniranom funkcijom mutacije i ciljanom mutacijom na pogrešno obojanim vrhovima također se pokazala vrlo uspješnom. Bitno je naglasiti da su razlike točnosti među boljim algoritmima relativno zanemarive s obzirom na veličinu testnog skupa.

Mogući razlog uspješnosti ovih strategija je što uz davanje prednosti boljim jedinkama pri razmnožavanju, ciljano mijenjaju gene tako da se oni vrhovi koji su pravilno obojani ne mijenjaju.

Najlošije rezultate postigle su strategije koje su se oslanjale na slučajnost (slike 6.5, 6.8). Slučajna strategija odabira roditelja pokazala se inferiornijom naspram ostalih strategija biranja roditelja, baš kao i slučajna mutacija gena naspram ostalih strategija mutacije. Razlog je što kod odabira roditelja bolje jedinke nisu dobile prednost, djeca su naslijedila gene od slučajnih roditelja te tako samo rješenje nije konvergiralo prema optimumu. Kod slučajne mutacije gena, problem je što su geni za ovaj specifičan problem unutar jedinki zavisni. Na taj se način moglo dogoditi da su neki već dobro postavljeni geni postali pogrešni, bilo zbog vlastite mutacije ili mutacije susjeda.

Općenito se odabir ispravne strategije mutacije čini utjecajnijim na točnost od odabira strategije selekcije roditelja (slike 6.1, 6.3, 6.5, 6.7). Funkcija mutacije omogućuje pretraživanje šireg prostora i na taj način osigurava pronalazak optimalnog rješenja. Ona također spriječava da algoritam ostane u lokalnom optimumu i zaslužna je za genetsku raznolikost unutar populacije. Sama selekcija roditelja nema toliko utjecaj jer se ispravnim mutacijama u ovom specifičnom problemu sve jedinke ciljano poboljšavaju. Razlika između dobroti roditelja nije toliko bitna jer se nakon križanja (koje unosi element slučajnosti) svaki gen zasebno i ciljano mijenja uz određenu heuristiku. Time je

osigurano da dijete poslije mutacije u prosjeku ima bolju dobrotu nego prije mutacije.

Iako su za neke generalne probleme slučajne strategije u sklopu genetskih algoritama korisne, za specifičan problem poput bojanja grafa bolje su se pokazale usmjerene strategije koje uz neku heuristiku ciljano mijenjaju i biraju vrijednosti za sljedeću generaciju.



# LITERATURA

- [1] Daniel W Cranston i Landon Rabern. Brooks' theorem and beyond. *Journal of Graph Theory*, 80(3):199–225, 2015.
- [2] P. Gupta i H. V. Goenka. Genetic algorithms for graph colouring, 2021. URL <https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/>.
- [3] Musa M Hindi i Roman V Yampolskiy. Genetic algorithm applied to the graph coloring problem. U *Proc. 23rd Midwest Artificial Intelligence and Cognitive Science Conf*, stranice 61–66, 2012.
- [4] S. D'hondt K. Coolsaet i J. Goedgebeur. House of graphs 2.0: A database of interesting graphs and more, discrete applied mathematics, 325:97-107, 2023. URL <https://houseofgraphs.org>.
- [5] D. Kovačević, M. Krnić, A. Nakić, i M. Osvin Pavčević. *Diskretna matematika 1*. FER, 2020.
- [6] Raja Marappan i Gopalakrishnan Sethumadhavan. A new genetic algorithm for graph coloring. U *2013 Fifth International Conference on Computational Intelligence, Modelling and Simulation*, stranice 49–54. IEEE, 2013.
- [7] M Rajagaspar i S Senthil. Applications of graph coloring using vertex coloring. *JOURNAL OF ALGEBRAIC STATISTICS*, 13(2):3447–3454, 2022.
- [8] J Shen. Solving the graph coloring problem using genetic programming, 2003.
- [9] M. Čupić. *Prirodom inspirirani optimizacijski algoritmi*. FER, 2010.

## **Rješavanje problema bojanja grafa evolucijskim algoritmom**

### **Sažetak**

U ovom je radu prikazan i opisan način rada evolucijskog algoritma i njegova primjena na rješavanje problema bojanja grafa. Navedene su prednosti i mane pojedinih strategija odabira roditelja i strategija mutacije kod genetskog algoritma. Analiziran je utjecaj različitih strategija na točnost određivanja kromatskog broja grafa te su rezultati međusobno uspoređeni i objašnjeni.

**Ključne riječi:** evolucijski algoritam, genetski algoritam, kromatski broj, bojanje grafa, strategija mutacije, strategija odabira roditelja

## **Solving graph coloring problem using evolutionary algorithm**

### **Abstract**

This paper presents and describes how the evolutionary algorithm works and its application to solving the graph coloring problem. The advantages and disadvantages of individual parent selection strategies and mutation strategies in the genetic algorithm are listed. The influence of different strategies on the accuracy of determining the chromatic number of the graph was analyzed, and the results were compared and explained.

**Keywords:** evolutionary algorithm, genetic algorithm, chromatic number, graph coloring, mutation strategy, parent selection strategy