SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 797

Rješavanje problema bojanja grafa evolucijskim algoritmom

Filip Penzar

Zagreb, svibanj 2023.

SADRŽAJ

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.

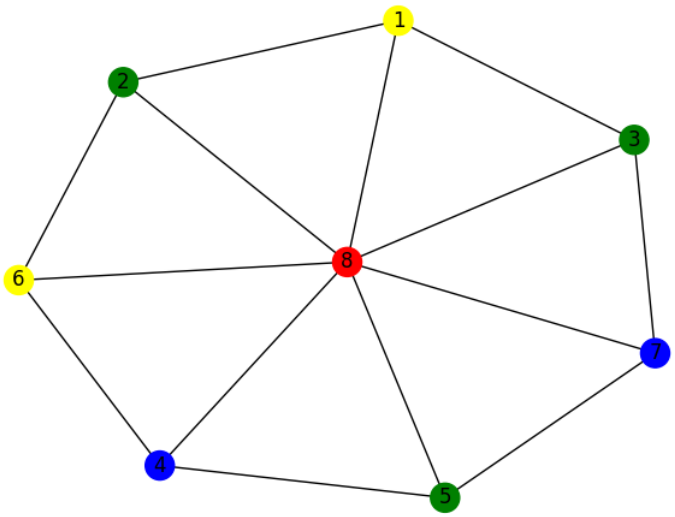
Peto poglavlje opisuje način određivanja kromatskog broja grafa genetskim algoritmom, implementaciju rješenja te korištene parametre i razloge njihovog odabira. U šestom se poglavlju iznose i uspoređuju rezultati testiranja različitih varijacija algoritma. Sedmo poglavlje objašnjava dobivene podatke 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 isrcpnog pretreaživanja 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 slikom 2.2.

```
def provjeri_ispravnost_bojanja(obojani_vrhovi, matrica_povezanosti):
    for vrh in matrica_povezanosti:
        for susjedan_vrh in matrica_povezanosti[vrh]:
            if obojani_vrhovi[vrh] == obojani_vrhovi[susjedan_vrh]:
                return False

    return True


def iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, k, indeks_vrha):
    # kraj rekurzije
    if indeks_vrha == len(vrhovi):
        if provjeri_ispravnost_bojanja(vrhovi, matrica_povezanosti):
            return vrhovi
        else:
            return None

    for boja in range(k):
        vrhovi[indeks_vrha] = boja
        bojanje_vrhova = iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, k, indeks_vrha + 1)
        if bojanje_vrhova is not None:
            return bojanje_vrhova

    return None
```

Slika 2.2: Algoritam iscrpnog pretraživanja

Jedan iterativni pristup pronalaska kromatskog broja je pokušati obojati graf s $k = n$ boja, gdje je n broj vrhova. Ukoliko 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 $\chi(G) = k = n - m + 1$. Algoritam je prikazan slikom 2.3.

```
def odredi_kromatski_broj(vrhovi, matrica_povezanosti):
    ispravno_bojanje = None
    for k in range(len(vrhovi), 0, -1):
        bojanje_vrhova = iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, k, 0)
        if bojanje_vrhova is None:
            return ispravno_bojanje, k + 1
        ispravno_bojanje = copy.copy(bojanje_vrhova)
    return ispravno_bojanje, 1
```

Slika 2.3: Iterativno određivanje kromatskog broja s iscrpnim pretraživanjem

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 slikom 2.4.

```
def pohlepno_pretrazivanje(vrhovi, matrica_povezanosti, vrhovi_po_prioritetu, k, indeks):
    if indeks == len(vrhovi):
        return vrhovi

    dostupne_boje = [i for i in range(k)]
    susjedni_vrhovi = matrica_povezanosti[vrhovi_po_prioritetu[indeks]]
    for i in range(indeks):
        if vrhovi_po_prioritetu[i] in susjedni_vrhovi:
            boja_susjeda = vrhovi[vrhovi_po_prioritetu[i]]
            if boja_susjeda in dostupne_boje:
                dostupne_boje.remove(boja_susjeda)

    if len(dostupne_boje) == 0:
        return None

    vrhovi[vrhovi_po_prioritetu[indeks]] = dostupne_boje[0]
    return pohlepno_pretrazivanje(vrhovi, matrica_povezanosti, vrhovi_po_prioritetu, k, indeks + 1)
```

Slika 2.4: Pohlepno pretraživanje

2.2.3. Usmjereno iscrpno pretraživanje

Algoritam usmjerenog iscrpnog pretraživanja radi vrlo slično kao i pohlepno pretraživanje, uz bitnu preinaku da ukoliko ne pronađe legalno bojanje, traži dalje. Algoritam odredi moguće boje za bojanje trenutnog vrha kao i pohlepni algoritam, ali ukoliko 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 slikom 2.5.

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

```python
def usmjereno_iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, vrhovi_po_prioritetu, k, indeks):
    if indeks == len(vrhovi):
        return vrhovi

    dostupne_boje = [i for i in range(k)]
    susjedni_vrhovi = matrica_povezanosti[vrhovi_po_prioritetu[indeks]]
    for i in range(indeks):
        if vrhovi_po_prioritetu[i] in susjedni_vrhovi:
            boja_susjeda = vrhovi[vrhovi_po_prioritetu[i]]
            if boja_susjeda in dostupne_boje:
                dostupne_boje.remove(boja_susjeda)

    for boja in dostupne_boje:
        vrhovi[vrhovi_po_prioritetu[indeks]] = boja
        bojanje = usmjereno_iscrpno_pretrazivanje(vrhovi, matrica_povezanosti, vrhovi_po_prioritetu, k, indeks + 1)
        if bojanje is not None:
            return bojanje
        if indeks == 0:
            return None

    return None
```

Slika 2.5: Usmjereno iscrpno pretraživanje

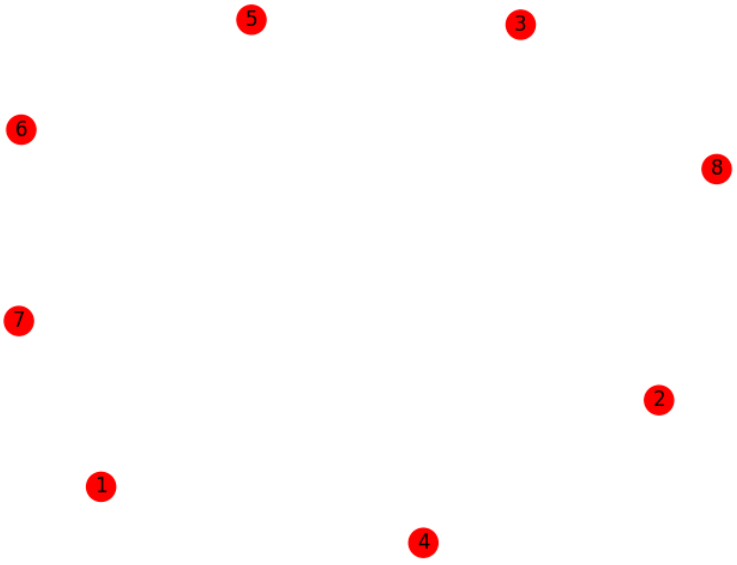
2.3. Gornja ograda na kromatski broj

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

Maksimalan stupanj grafa, Δ, je najveći broj bridova koji su incidentni s bilo kojim pojedinačnim vrhom u grafu.

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 2.6: Graf s $\chi(G) = 1$

3. Evolucijski algoritmi

Evolucijski algoritmi su algoritmi koji inspiraciju vuku iz prirode, iz Darwinonve 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.

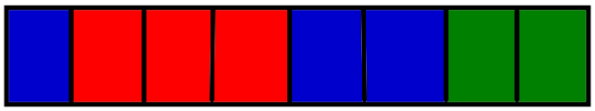
3.1. Genetski algoritmi

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 generacije. 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].

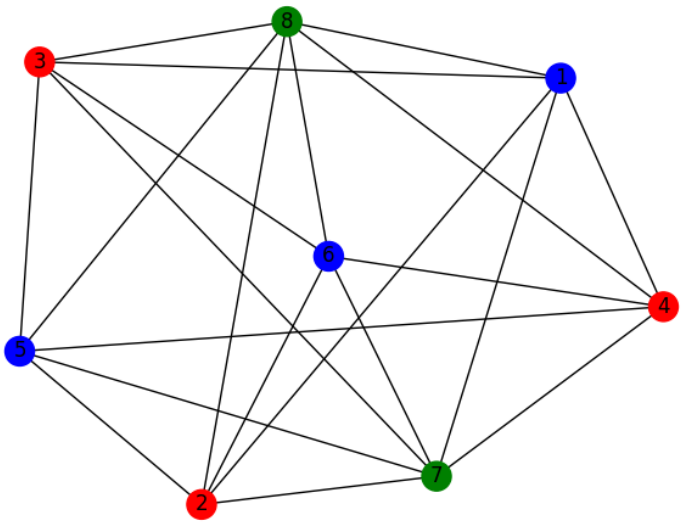
3.1.1. Kodiranje genoma

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 pobojani s k različitih boja tako da pripadajući spojeni vrhovi nemaju istu vrijednost. Ukoliko 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]. Implementacija algoritma prikazana je slikom 3.3.

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

Slika 3.3: Funkcija cijene implementirana u Pythonu

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
- Roullete wheel
- Rang selekcija

Slučajna selekcija

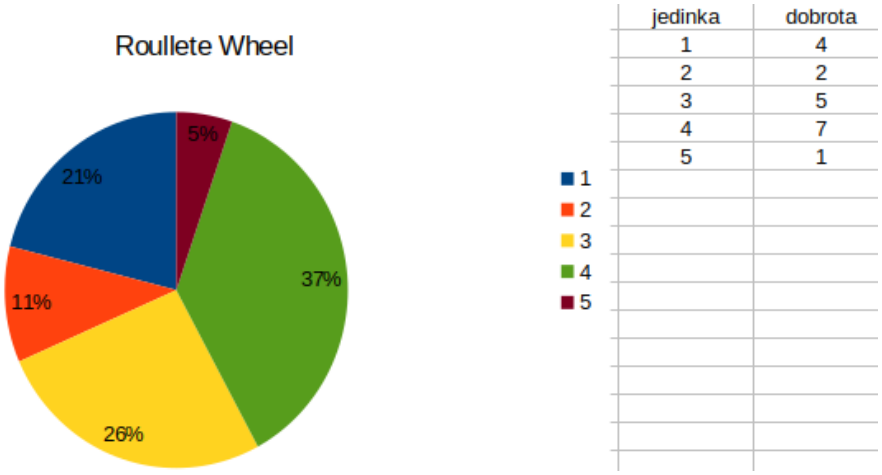
Kao naivnu metodu selekcije roditelja koristi se Slučajna selekcija. Iz populacije se slučajnim odabirom izabiru dvije jedinke kao roditelji novoj jedinci. Ova metoda je manje uspješna od ostalih metoda selekcije i zato nije pretjerano korištena. Razlog manjoj uspješnosti je što cijena izračunata za 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 dobrota djece i djeca se dodaju populaciji. Iz populacije se tada izbacuju dvije jednike 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.

Roullete wheel

Nakon što su izračunate cijene za sve jedinke u generaciji, svakoj se pridjeljuje udio na *roullete wheel-u*. Udio je proporcionalan cijenama jedinki; one s nižom cijenom (većom dobrotom) će imati veći udio na *roullete wheel-u*. 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.



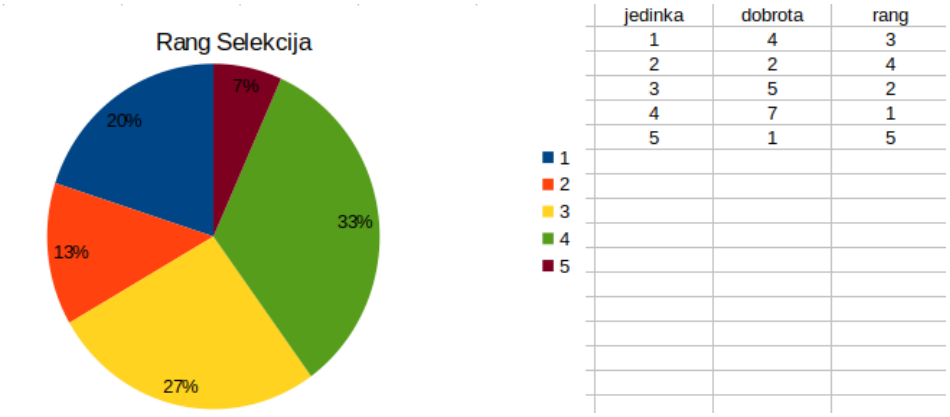
Slika 3.4: Primjer *roullete wheel-a*

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.

Rang selekcija

Rang selekcija po principu je slična *roullete wheel-u*, ali se umjesto proporcionalnog udjela vjerojatnosti prema cijeni, jedinke prvo rangiraju prema cijeni, a udjeli na kotaču

se zatim dodjeljuju proporcionalno rangu.



jedinka	dobrota	rang
1	4	3
2	2	4
3	5	2
4	7	1
5	1	5

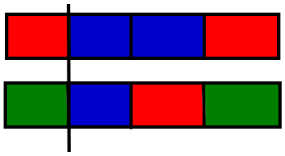
Slika 3.5: Primjer rang selekcije

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

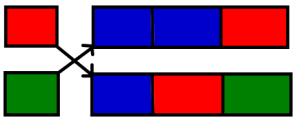
Križanje je postupak kojim se iz izabranih genoma roditelja dobiva genom potomka. Potomak se sastoji od pomješanog genoma svojih roditelja. Križanje se izvodi tako da se na slučajnom mjestu prerežu genomi oba roditelja u točki prijeloma, zamijene se prvi dijelovi genoma i dobiju se dvije nove jedinke; 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.

Postupak križanja vizualno je prikazan na slikama 3.6, 3.7, 3.8.

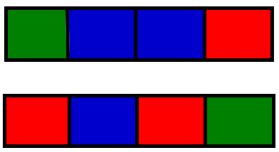


Slika 3.6: Kromosomi roditelja

Ukoliko smo koristili dobru metodu za odabir roditelja, djeca će se sastojati od pomiješanih gena uspješnih roditelja, približiti će se optimumu. Često se dogodi da



Slika 3.7: Križanje



Slika 3.8: Kromosomi djece

su djeca manje dobrote od svojih roditelja, no prosječno gledano na cijeloj populaciji, dobrota djece se povećava.

3.1.5. Mutacije

Mutacije su promjene gena na kromosomima jedinki. One su vrlo bitne u genetskim algoritmima jer nam omogućuju da pretražujemo rješenje na širem prostoru. Ukoliko ne bismo uveli 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 naš skup jedinki. Ukoliko 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.

Slika 3.9 prikazuje kromosom prije mutacije, a slika 3.10 kromosom poslije mutacije na trećem i osmom genu.

Slika 3.9: Kromosom prije mutacije

Klasično se mutacije implementiraju 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 3.10: Kromosom poslije mutacije

3.2. Implementacija 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, pronađe optimalno rješenje problema ili se ne pređe određen broj iteracija. Pseudokod genetskog algoritma dan je slikom 3.11.

```
def generiraj_pocetnu_populaciju(velicina_populacije):
    pocetna_populacija = []
    for _ in range(velicina_populacije):
        pocetna_populacija.append(generiraj_slucajnu_jedinku())
    return pocetna_populacija


def izaberi_roditelje(populacija):
    roditelji = []
    for _ in range(len(populacija)):
        roditelji.append(rang_selekcija())
    return roditelji


def krizanje(roditelji):
    djeca = []
    for i in range(0, len(roditelji), 2):
        dijete_1, dijete_2 = generiraj_djecu(roditelji[i], roditelji[i+1])
        djeca.append(dijete_1)
        djeca.append(dijete_2)
    return djeca


def mutiraj(djeca):
    for dijete in djeca:
        for gen in dijete.kromosom:
            if slucajan_dogadaj():
                gen = slucajan_gen()
    return djeca


def genetski_algoritam(broj_iteracija, limit_dobrote, velicina_populacije):
    populacija = generiraj_pocetnu_populaciju(velicina_populacije)
    for _ in range(broj_iteracija):
        najbolja_dobrota, jednika_s_najboljom_dobrotom = maksimalna_dobrota(populacija)
        if najbolja_dobrota >= limit_dobrote:
            return True, jednika_s_najboljom_dobrotom

        roditelji = izaberi_roditelje(populacija)
        djeca = krizanje(roditelji)
        populacija = mutiraj(djeca)

    return False, jednika_s_najboljom_dobrotom
```

Slika 3.11: Pseudokod genetskog algoritma

4. Korištene biblioteke i podaci

4.1. PyGAD

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

4.1.1. Instalacija

Instalacija biblioteke vrši se preko PIP-a sljedećom naredbom:

```
$ pip install pygad
```

4.2. 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 biblioteke vrši se preko PIP-a sljedećom naredbom:

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

4.3. House of Graphs

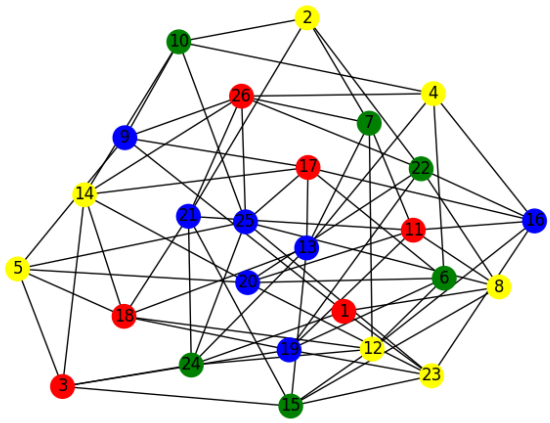
House of Graphs [4] je Web-stranica koja sadržava bazu grafova s karakteristikama koja dozvoljava njihovo besplatno preuzimanje. House of Graphs razvijen je u suradnji sveučilišta Ghent i sveučilišta KU Leuven.

U sklopu 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
umber 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 dana je slikom 4.1, a ispravno bojanje za taj graf dano je slikom 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 biblioteka.

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 bibliotekom 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 dobrota 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 prijeloma.

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 kompleksnije funkcije mutacije.

5.4.1. Slučajna mutacija na krivim 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]. Implementacija je dana u Pythonu slikom 5.1.

```
def slucajna_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k):
    for dijete in djeca:
        for vrh, boja_vrha in enumerate(dijete):
            potrebno_novo_bojanje = False
            for susjedan_vrh in matrica_susjedstva[vrh]:
                if boja_vrha == dijete[susjedan_vrh]:
                    potrebno_novo_bojanje = True
                    break
            if not potrebno_novo_bojanje:
                continue
            nova_boja = random.randint(k)
            dijete[vrh] = nova_boja
    return djeca
```

Slika 5.1: Slučajna mutacija na krivim vrhovima

5.4.2. Ciljana mutacija na krivim vrhovima

U ovoj metodi, slično kao i u potpoglavlju 5.4.1, mutacija se odvija samo na vrhovima koji su obojani isto 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. Ukoliko 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 ukoliko to mogu [3]. Implementacija je dana u Pythonu slikom 5.2.

```
def ciljana_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k):
    for dijete in djeca:
        for vrh, boja_vrha in enumerate(dijete):
            potrebno_novo_bojanje = False
            boje_susjeda = set()
            for susjedan_vrh in matrica_susjedstva[vrh]:
                boje_susjeda.add(dijete[susjedan_vrh])
                if boja_vrha == dijete[susjedan_vrh]:
                    potrebno_novo_bojanje = True

            if not potrebno_novo_bojanje:
                continue

            if len(boje_susjeda) == k:
                nova_boja = random.randint(k)
            else:
                dostupne_boje = []
                for boja in range(k):
                    if boja not in boje_susjeda:
                        dostupne_boje.append(boja)
                nova_boja = random.choice(dostupne_boje)
            dijete[vrh] = nova_boja
    return djeca
```

Slika 5.2: Ciljana mutacija na krivim vrhovima

5.4.3. Kombinacija slučajne i ciljane mutacije na krivim 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. Ukoliko je dobrota najbolje jedinke blizu optimalne, nula, tada se koristi *Slučajna mutacija na krivim vrhovima* jer se želi izbjeći zaustavljanje u lokalnom optimumu. Kada je dobrota najbolje jedinke daleko od optimalne, koristi se *Ciljana mutacija na krivim vrhovima* jer se želi usmjerenije pretražiti prostor i na taj način stići do optimuma [3]. Implementacija je dana u Pythonu slikom 5.3.

```
def kombinacija_slucajne_i_ciljane_mutacije(djeca, matrica_susjedstva, k, najbolja_dobrota, granica):
    if najbolja_dobrota >= granica:
        return slucajna_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k)
    else:
        return ciljana_mutacija_na_krivim_vrhovima(djeca, matrica_susjedstva, k)
```

Slika 5.3: Kombinacija slučajne i ciljane mutacije na krivim vrhovima

5.5. Ostali parametri

Ostali korišteni parametri dani 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

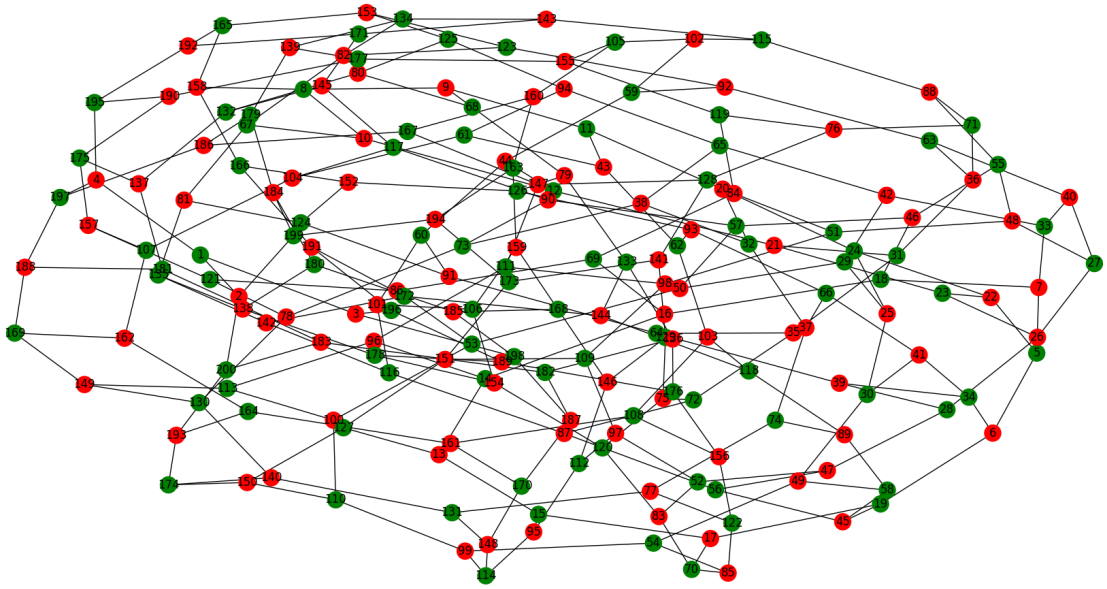
Nakon parsiranja grafova preuzetih s House of Graphs (potpoglavlje 4.3), za svaki od grafova određuje se kromatski broj. Početna vrijednost k je postavljena na $\Delta + 1$. Ako se genetskim algoritmom pronašlo ispravno bojanje sa zadanim k (dobrota neke jedinke je jednaka nuli), k se smanjuje za 1, a najbolja jedinka se pohranjuje. Algoritam prekida s radom u dva slučaja; ukoliko 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. Implementacija je dana u Pythonu slikom 5.4.

```
def odredi_kromatski_broj(graf, parametri_genetskog_algoritma):
    delta = graf.maksimalan_stupanj_grafa
    kromatski_broj = graf.kromatski_broj
    k = delta + 1
    prethodno_rjesenje = None
    while(k >= kromatski_broj):
        algoritam = genetski_algoritam(parametri_genetskog_algoritma, graf, k)
        rjesenje = algoritam.run()
        if rjesenje is None:
            return k + 1, prethodno_rjesenje
        prethodno_rjesenje = rjesenje
        k = k - 1
    return k + 1, prethodno_rjesenje
```

Slika 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 kromatkog broja. Uz to, program je u mogućnosti vizualizirati dobivene rezultate crtanjem obojenog grafa.



Slika 5.5: Bojanje bipartitnog grafa s 200 vrhova

6. Rezultati praktičnog dijela

Testirane 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 1226 grafova s rasponom broja vrhova od 200 do 249.

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

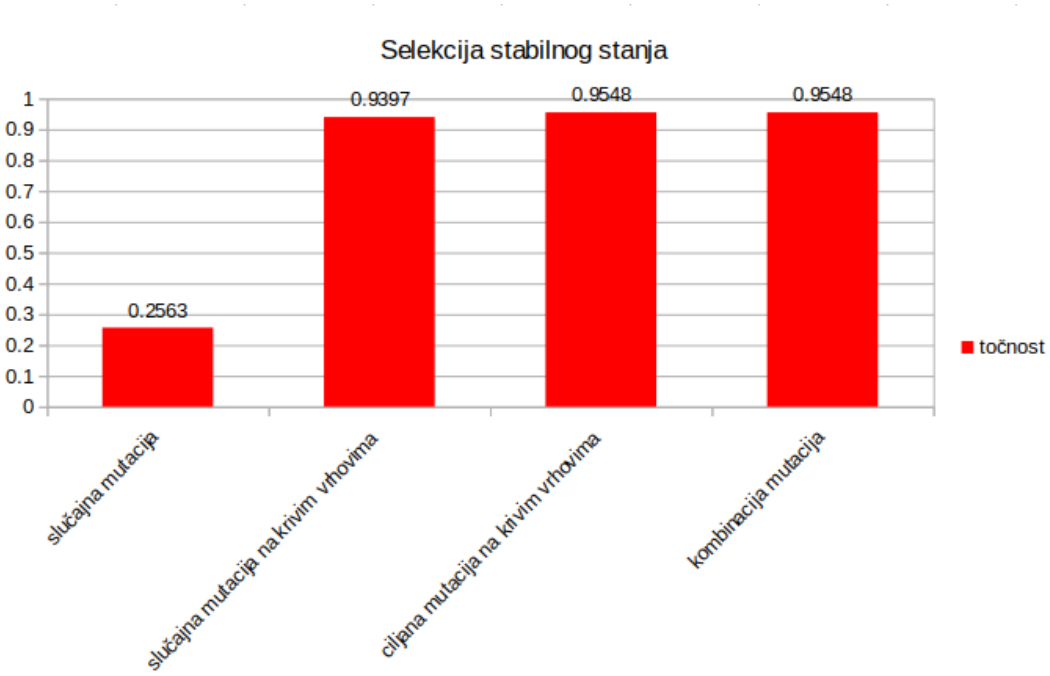
6.1. Manji skup podataka

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.

Selekcija stabilnog stanja

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

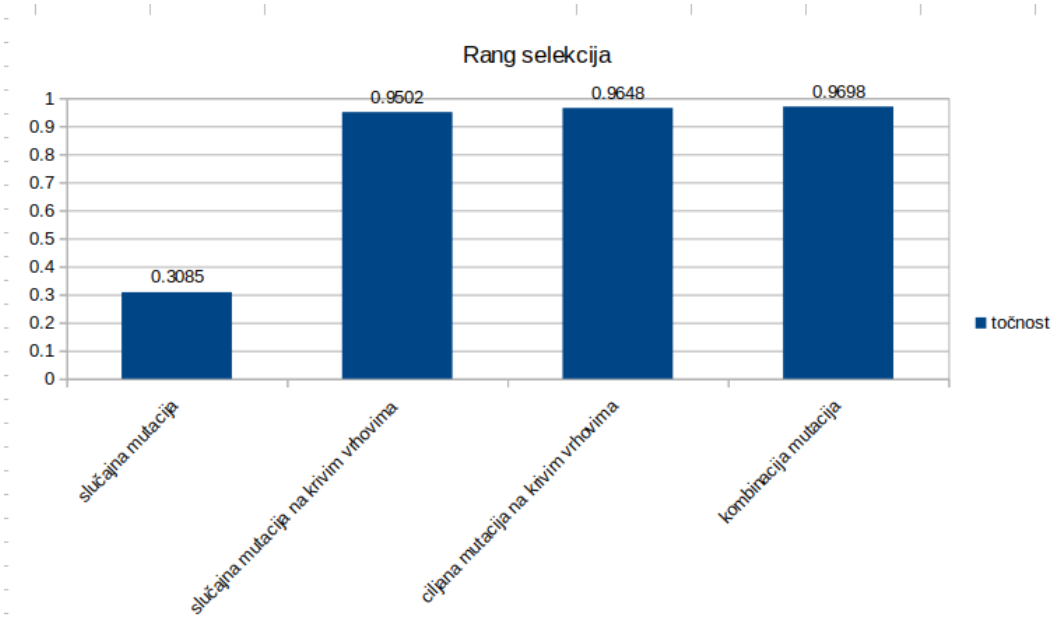


Slika 6.1: Grafikon točnosti algoritama selekcije stabilnog stanja

Iz grafikona sa slike 6.1 vidimo da je točnost daleko najlošija kod algoritma sa slučajnom mutacijom. Ostale tri varijacije se čine podjednako uspješne, s točnosti od oko 95%. 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.

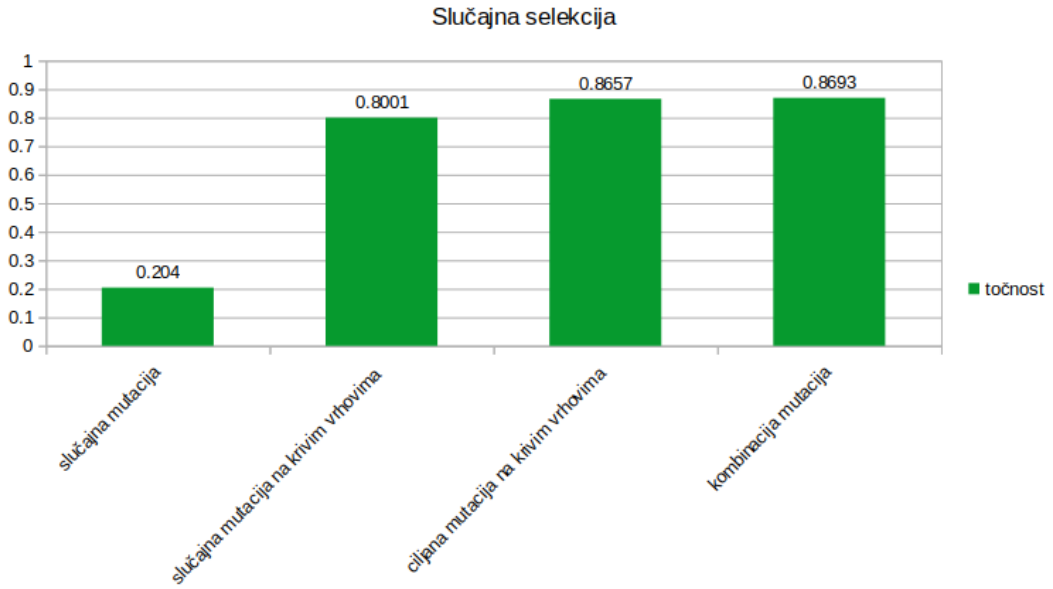


Slika 6.2: Grafikon točnosti algoritama rang selekcije

Kao i kod selekcije stabilnog stanja, iz grafikona sa slike 6.2 vidimo da je slučajna mutacija postigla najlošije rezultate. Ostale tri varijacije su podjednako uspješne, s točnosti od preko 95%. Uspoređujući rang selekciju i selekciju stabilnog tipa, vidimo da je rang selekcija ostvarila bolje rezultate u svakoj od varijacija algoritma.

Slučajna selekcija

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

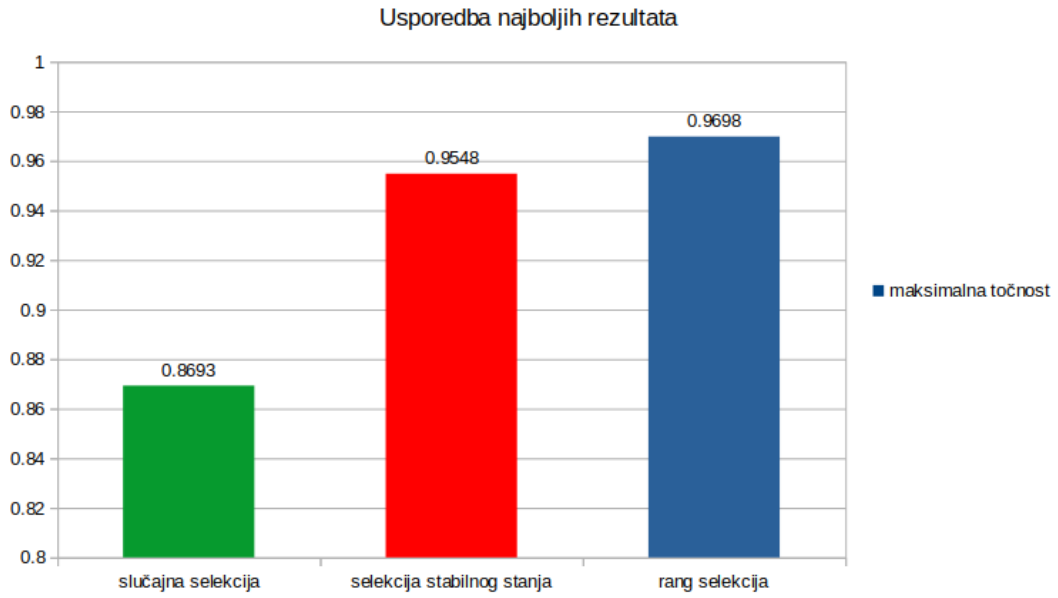


Slika 6.3: Grafikon točnosti algoritama slučajne selekcije

Iz grafikona sa slike 6.3 uočavamo da je točnost manja za sve strategije mutacija u odnosu na rang selekciju i selekciju stabilnog stanja. Slučajna mutacija i dalje ima najlošiju točnost od samo 20%.

6.1.1. Usporedba rezultata

U nastavku su prikazane najviše postignute točnosti grupirane po strategijama odabira roditelja.



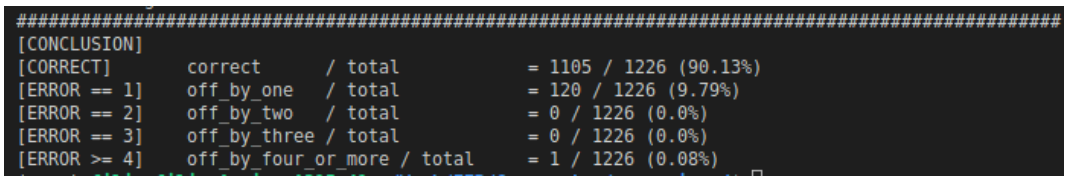
Slika 6.4: Grafikon točnosti najboljih varijacija algoritama

Iz grafikona sa slike 6.4 vidimo da je rang selekcija postigla najbolju točnost, dok je slučajna selekcija najmanje uspješna. Razlika postignute točnosti između te dvije strategije odabira roditelja je čak 10%. Selekcija stabilnog tipa je vrlo blizu rang selekciji s 95%-tnom točnošću.

6.2. Veći skup podataka

Na skupu podataka od 1226 grafova s brojem vrhova između 200 i 249 testirane su samo varijacije algoritama koje su ostvarile najbolje rezultate na skupu od 200 podataka.

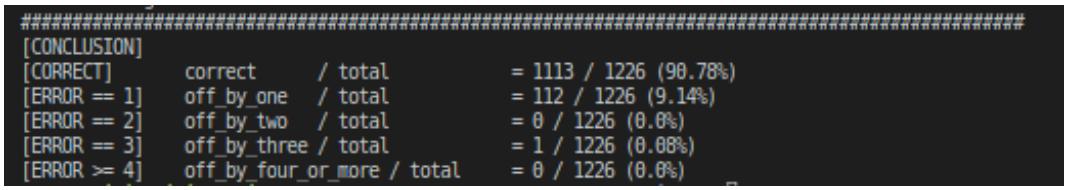
Selekcija stabilnog stanja s kombiniranom funkcijom mutacije



Slika 6.5: Selekcija stabilnog stanja s kombiniranom funkcijom mutacije

Iz rezultata sa slike 6.5 vidimo da je algoritam postigao točnost od 90%. U samo jednom slučaju je pogriješio za 4 ili više, a u preostalim slučajevima je pogriješio za samo jednu boju više.

Rang selekcija s kombiniranom funkcijom mutacije



Slika 6.6: Rang selekcija s kombiniranom funkcijom mutacije

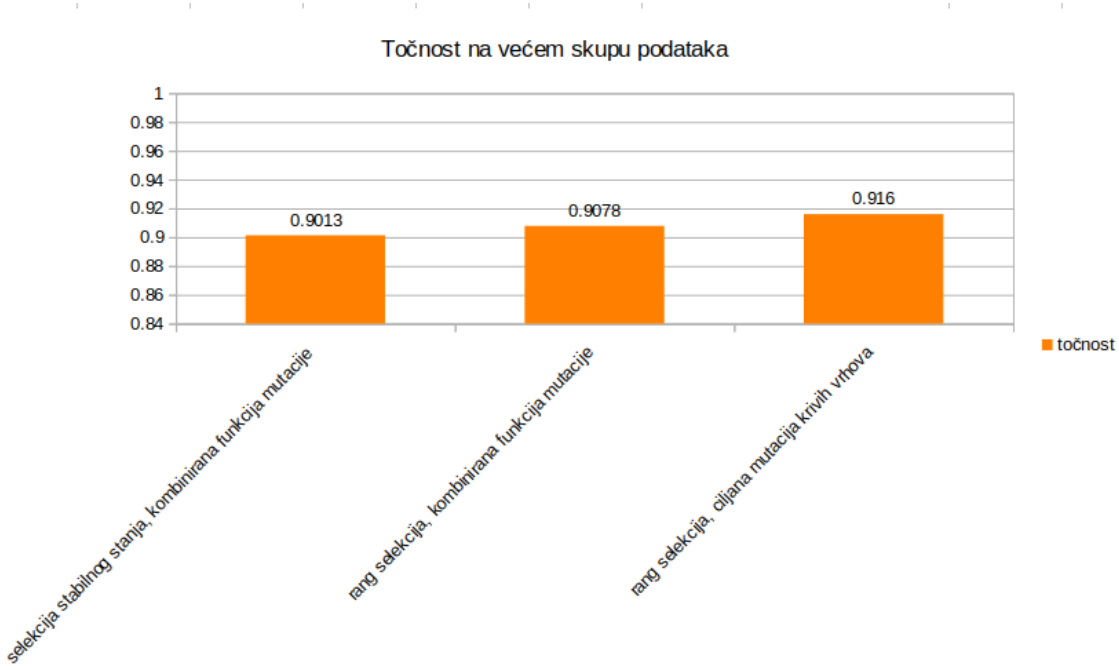
Iz rezultata sa slike 6.6 vidimo da je algoritam postigao točnost od skoro 91%. U samo jednom slučaju je pogriješio za 3, a u preostalim je slučajevima pogriješio samo za jednu boju više.

Rang selekcija s ciljanom mutacijom krivih vrhova

```
##################################################################################################
[CONCLUSION]
[CORRECT]       correct      / total           = 1123 / 1226 (91.6%)
[ERROR == 1]    off_by_one   / total           = 102 / 1226 (8.32%)
[ERROR == 2]    off_by_two   / total           = 0 / 1226 (0.0%)
[ERROR == 3]    off_by_three / total           = 0 / 1226 (0.0%)
[ERROR >= 4]    off_by_four_or_more / total    = 1 / 1226 (0.08%)
```

Slika 6.7: Rang selekcija s ciljanom mutacijom krivih vrhova

Iz rezultata sa slike 6.7 vidimo da je algoritam postigao točnost od preko 91.5%. U samo jednom slučaju je pogriješio za 4 ili više, a u preostalim je slučajevima pogriješio samo za jednu boju više.



Slika 6.8: Grafikon točnosti algoritama na većem skupu podataka

Iz grafikona sa slike 6.8 vidimo da su točnosti algoritama vrlo blizu. Selekcija stabilnog stanja s kombinacijom funkcija mutacije postigla je najlošiji rezultat, a rang selekcija s ciljanom mutacijom krivih vrhova najbolji.

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 jednu boju.

Kao najbolja kombinacija parametara pokazala se rang selekcija kao strategija odabira roditelja i kombinirana funkcija mutacije ili ciljana mutacija krivih vrhova kao strategije mutacije (slike 6.4, 6.8). Kombinirana funkcija mutacije pokazala se nešto boljom na manjem skupu podataka, a ciljana mutacija krivih vrhova kao najbolja strategija mutacije na većem skupu podataka.

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. Strategija selekcije stabilnog stanja nije toliko uspješna jer se po generaciji generira manji broj novih jedinki i na taj način se sporije pretražuje prostor. Budući da je broj iteracija bio limitiran na 1000, takva se strategija nije uspjela proširiti po prostoru dovoljno brzo i pronaći optimalno rješenje.

Najlošije rezultate postigle su strategije koje su se oslanjale na slučajnost (slika 6.3). 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.

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