Minimalno particionisanje grafa na klike

Seminarski rad u okviru kursa Računarska inteligencija
Matematički fakultet
Univerzitet u Beogradu

Jelena Milivojević
mi16004@alas.matf.bg.ac.rs

Septembar 2022

Sažetak

U ovom radu obradjene su različite metaheuristike za rešavanje problema minimalnog particionisanja grafa na klike i uporedjeni su njihovi rezultati. Problem minimalnog particionisanja grafa na klike sveden je na problem minimalnog bojenja grafa.

Sadržaj

1 Uvod

Neka je dat neusmereni graf G = (V, E) sa skupom čvorova V i skupom ivica E. Tada je njegov **komplement** graf $\overline{G} = (V, \overline{E})$, gde je $\overline{E}$={(i,j) | i,j $\in V \wedge (i,j) \notin E$}.

Klika C grafa G je podskup čvorovova, $C \subseteq V$, takav da su svaka dva susedna čvora medjusobno povezana (kompletan graf).

Problem minimalnog particionisanja grafa G na klike je problem podele čvorova grafa G na minimalan broj podskupova čvorova koji su klike. Ovaj problem je NP-težak. Ovako definisan problem minimalnog particionisanja na klike može se svesti na problem minimalnog bojenja grafa, tj. odredjivanje minimalnog particionisanja grafa G na klike ekvivalentno je minimalnom bojenju komplemetnog grafa $\overline{G}$ [1].

Bojenje čvorova predstavlja pridruživanje boja čvorima grafa G tako da nikoja dva susedna čvora nisu obojena identičnom bojom. **K-bojenje** je bojenje grafa koje koristi najvise k različitih boja da oboji sve čvorove grafa. Najmanji prirodan broj k za koje se graf G moze obojiti zove se **hromatski broj** $\chi(G)$. Bojenje čvorova grafa je takodje NP-težak problem.

U nastavku, rešavanje problema minimalnog particionisanja na klike biće svedeno na rešavanje problema minimalnog bojenja.

2 Algoritam grube sile

Algoritam grube sile iscrpno ispituje sve varijante i dolazi do egzaktnog rešenja. Funkcija brute poziva se iterativno za broj boja `num_colors` = 1, dok se ne dobije ispravno bojenje grafa - za prvi broj različitih boja koji je dovoljan da dobijemo ispravno bojenje grafa algoritam se zaustavlja. Budući da ionako velika složenost algoritma, O(n^n), eksponencijalno raste sa povećanjem ulaza, algoritam je neupotrebljiv za primenu nad grafovima većih dimenzija.

```
function brute(graf, br_boja, rbr_cvora, bojenje)
    if (rbr_cvora == |V|):
        if (ispravnoBojenje(graf, bojenje)):
            return True
        return False

    for (svaka boja od mogucih boja):
        bojenje[rbr_cvora] = boja
        if brute(graf, br_boja, rbr_cvora, bojenje):
            return True
        bojenje[rbr_cvora] = 0

    return False
```

3 Pohlepni algoritam

Algoritam prolazi kroz listu čvorova dodeljujući boju svakom od čvorova koje obradjuje. Boje su predstavljene rednim brojevima od 1 do | $\overline{V}$ | +1. Svaka boja i čvor kome je dodeljena ta boja čuvaju se kao par u rečniku color. Svakom čvoru iz liste se dodeljuje boja sa najmanjim rednim

brojem koja nije zauzeta od strane nekog od susednih čvora. Na ovaj način algoritam redukuje broj ispitivanja mogućih rešenja.

Iako je dati algoritam vremenski efikasan, pokazuje se da sam redosled čvorova koji se ispituju značajno utiče na rešenje. Ukoliko se čvorovi obilaze bez konkretno definisanog redosleda obilaska čvorova, mozemo dobiti neoptimalno rešenje.

```
function greeedy(graf, bojenje)
    dodeli boju prvom cvoru u grafu
    br_boja = 1
    for (ostali cvorovi iz liste cvorova):
        odredi skup zauzete_boje koji sadrzi boje suseda cvora
        for (svaka boja od mogucih boja)
            ako se boja ne nalazi u skupu zauzete_boje:
                break

        bojenje[cvor] = boja
        ako je boja > br_boja:
            br_boja = boja
    return br_boja
```

3.1 Welsh-Powell algoritam

Da bi se rešio problem neoptimalnih rešenja zbog redosleda obilaska čvorova možemo koristiti Welsh-Powell heuristiku. Kod ovog pristupa odredjuje se stepen svakog od čvorova u grafu i čvorovi se sortiraju opadajuće prema stepenu čvorova. Kao rezultat toga dobijamo varijantu pohlepnog algoritma koja koristi najviše $max_i\ min\{degree(x_i) + 1, i\}$ boja, najviše jednu boju više od najveće vrednosti stepena čvorova u grafu [2].

```
function WelshPowell(graf, bojenje, stepeni)
    sortiraj cvorove u opadajucem redosledu prema stepenu cvorova
    br_boja = 0
    for (svaki cvor i iz liste cvorova):
        if (cvoru nije dodeljena boja):
            br_boja = br_boja + 1
            bojenje[i] = br_boja
        for (svaki cvor j iz liste cvorova koji ima manji stepen od i):
            if (i i j nisu susedi):
                bojenje[j] = br_boja
    return br_boja
```

4 Simulirano kaljenje

Simulirano kaljenje je optimizacioni algoritam koji pripada S-metaheuristikama baziranim na unapredjenju jednog rešenja. Naziv algoritma potiče od procesa kaljena koji se koristi u metalurgiji.

Ovaj proces obuhvata zagrevanje i potom postepeno hladjenje metala, što dovodi do promena njegovih fizičkih svojstava tj. materijal se oplemenjuje. Sto duže traje hladenje materijala, to on postaje čvršći.

Algoritam prvo generiše početno rešenje iz prostora dopustivih rešenja, a zatim dok nije zadovoljen kriterijum zaustavljanja pronalazi novo rešenje iz okoline trenutnog rešenja. Ukoliko je novo rešenje bolje od trenutnog ažurira se trenutno rešenje, u suprotnom uporedjuje se vrednost q i verovatnoća prihvatanja p, gde je $q \in U(0,1)$, a p = $1/\sqrt{i}$ za i iteracija. Ako je vrednost q manja od verovatnoće prihvatanja novo rešenje postaje trenutno rešenje. Ovo uslovno prihvatanje lošijeg rešenja odgovara procesu postepenog hladenja kod fizičkog kaljenja, tj. imamo postepeno smanjivanje verovatnoće prihvatanja lošijeg rešenja sa povećanjem broja iteracija.

U našem slučaju kriterijum za zaustavljanje je dostizanje maksimalnog broja iteracija, a takodje je moguć preuranjen izlazak iz algoritma ukoliko je dostignuta optimalna vrednost. Rešenje je predstavljeno nizom celih brojeva gde se na i-toj pozciji u nizu nalazi boja za i-ti čvor. Rešenje iz okoline trenutnog rešenja dobija se dodelom nove boje jednom od čvorova grafa. Ako novo rešenje nije u skupu dopustivih rešenja, boja čvora se vraća na prethodnu i prelazimo u narednu iteraciju.

```
function simulatedAnnealing(graf, br_iter, hromatski_broj, m_stepen)
    trenutno_resenje = inicijalizuj(graf, m_stepen)
    najbolje_resenje = trenutno_resenje
    for (nije zadovoljen kriterijum zaustavljanja):
        novo_resenje = generisiSusednoResenje(resenje, graf, m_stepen)
        if (novo_resenje nije dopustivo):
            continue
        if f_cilja(novo_resenje) < f_cilja(trenutno_resenje):
            trenutno_resenje = novo_resenje
            if f_cilja(novo_resenje) < fja_cilja(najbolje_resenje):
                najbolje_resenje = novo_resenje
        else:
            q = U(0,1)
            if q < verovatnoca_prihvatanja:
                trenutno_resenje = novo_resenje
    return fja_cilja(najbolje_resenje)
```

5 Optimizacija rojem čestica

Za razliku od simuliranog kaljena, optimizacija rojem čestica pripada P-metaheuristikama koje rade sa populacijom rešenja - čestica. Ovaj algoritam je zasnovan na prirodnom ponašanju pojedinačnih jedinki, čestica, unutar grupe (npr. jato ptica). Ako je neka jedinka jata pronašla hranu, ostale jedinke će se ugledati na nju i početi da modifikuju svoje kretanje tako da teže ka hrani. Svaka jedinka neće slepo pratiti najbolju jedinku, jedinku koja je pronašla hranu, već ce se takodje voditi i svojim instiktom da nezavisno od jata traži hranu. Ukoliko ova jedinka pronadje bolji izvor hrane, ostale jedinke će sada početi da se ugledaju na nju. Dakle, kod inteligencije rojeva svaka čestica roja istovremeno se ugleda na ponašanje drugih jedinki, ali takodje sledi neki svoj instikt.

U našem algoritmu, pozicije čestica su predstavljene kao moguća bojenja grafa iz dopustivog skupa rešenja. Kvalitet čestice tj. rešenja zavisi od broja iskorišćenih boja za bojenje grafa, što je

manji broj to je čestica bolja. Nakon ažuriranja pozicija čestice, ukoliko rešenje ne upada u skup dopustivih rešenja, rešenje se popravlja tako da postane dopustivo.
Vrednost brzine se ažurira korišćenjem formule:

$$v_i(t+1) = c_i * v_i(t) + r_l * c_l * (min_i - f(x_i)) + r_g * c_g * (global\_min - f(x_i))$$

Gde su c_i, c_l, c_g unapred zadati parametri, r_l, r_g vrednosti iz U(0, 1), min_i najbolja vrednost rešenja za česticu i, *global_min* vrednost rešenja globalno najbolje pozicije, a f je funkcija cilja.

```
function PSO(graf, br_cestica, br_iteracija)
    for (svaka cestica i iz roja):
        inicijalizuj poziciju cestice x_i
        min_vrednost_i = f(x_i)
        v_i = 0
        if f(x_i) < f (globalni_min):
            globalni_min = x_i
            globalni_min_vrednost = f(globalni_min)

    while (nije zadovoljen kriterujem zaustavljanja):
        izaberi r_l i r_g iz U(O, 1)
        azuriraj vektor brzine
        azuriraj poziciju cestice
        if f(x_i) < min_vrednost_i:
            min_vrednost_i = f(x_i)
            if f(x_i) < f(global_min):
                global_min_vrednost = f(x_i)
                global_min = x_i
    return global_min_vrednost, global_min
```

6 Genetski algoritam

Genetski algoritam je jedan od evolutivnih algoritama inspirisanih Darvinovom teorijom evolucije. On takodje spada u algoritme koji su zasnovani na populaciji rešenja, P-metaheuristike.

U našem algoritmu, jedinke populacije su predstavljene nizom celih brojeva koji odgovaraju bojama čvorova u grafu, gde su indeksi niza odgovarajući čvorovi. Vrednost funkcije cilja odgovara broju različitih boja neophodnih da se oboji graf, dok funkcija prilagodjenosti pored vrednosti funkcije cilja sadrži i odgovarajući penal. Budući da ne proveravamo dopustivost rešenja, tj. jedinke, uvodimo penal. Penal nam govori koliki je broj nepravilno obojenih čvorova - čvor je nepravilno obojen ukoliko je iste boje kao i neki od njegovih suseda.

U prvom koraku inicijalizujemo početnu populaciju jedinki. Nakon toga elitizmom prebacujemo u novu populaciju `ELITISM_SIZE` najboljih jedinki. Potom dok broj jedinki u novoj populaciji ne dostigne `POPULATION_SIZE` pratimo sledeći postupak: operatorom selekcije biramo svakog od roditelja, dobijamo decu ukrštanjem roditelja, nakon toga mutiramo decu i dodajemo ih u novu populaciju. Ovaj postupak formiranja nove populacije ponavaljamo za svaku generaciju dok ne dostignemo najveći dozvoljen broj generacija ili dok funkcija cilja ne dostigne vrednost unapred poznatog optimalnog broja boja. Za selekciju je korišćena turnirska selekcija, kod koje imamo unapred zadatu veličinu turnira na osnovu čega uzimamo odgovarajući broj jedinki koje se medjusobno

takmiče i jedinka sa najmanjim fitnesom pobedjuje. Roditelji koji su dobijeni kao rezultat takve selekcije se potom ukrštaju operatorom jednopozicionog ukrštanja, slučajno biramo jednu poziciju i do te pozicije u prvom detetu uzimamo deo prvog roditelja do te pozicije, a od te pozicije do kraja prebacujemo vrednosti drugog roditelja od te pozicije, obrnuto radimo za drugo dete. Nad decom koristimo operator mutacije koji je implementiran tako da sa velikom verovatnocom (0.8) menja vrednost boje jednog nasumično odabranog čvora. Na najbolju jedinku u svakoj generaciji primenjujemo simulirano kaljenje.

```
function GeneticAlgorithm(graf, hromatski_broj, broj_generacija)
    populacija = inicijalizujPopulaciju()
    while (nije dostignut broj generacija):
        prebaci k najboljih jedinki iz populacija u nova_populacija
        for (nije dostignuta duzina populacije):
            roditelj1 = selekcija(populacija)
            roditelj2 = selekcija(populacija)
            dete1, dete2 = ukrstanje(roditelj1, roditelj2)
            mutacija(dete1)
            mutacija(dete2)
            dodaj dete1 i dete2 u nova_populacija
            populacija = nova_popuLlacija
    return br_boja, najbolja_jedinka, br_iteracija
```

7 Rezultati

Algoritmi su testirani na laptopu sa sledećim specifikacijama:

Marka:	Dell
Model:	Vostro 3578
OS:	16.04.7 LTS (Xenial Xerus), 64-bit
RAM:	8GB
Procesor:	Intel® Core™ i7-8550U CPU @ 1.80GHz X 8
Broj jezgara:	4

Za testiranje korišćeni su grafovi u DIMACS formatu za koje već postoje poznata optimalna bojenja grafa.
U tabeli $\chi(G)$ predstavlja poznato optimalno rešenje, a reš. je najbolje rešenje dobijeno primenom implementiranih algoritama iz ovog rada. Broj iteracija iter. se odnosi na broj iteracija potrebnih za dobijanje datog rešenja za algoritme simuliranog kaljenja, optimitazije rojem cestica i genetski algoritam. Kod PSO algoritma dodat je kriterijum izlaska zbog dužine izvršavanja.

8 Zaključak

Na osnovu dobijenih rezultata može se uočiti da se najbolje pokazao pohlepni algoritam sa obilaskom čvorova u opadajućem redosledu prema stepenu čvora, Welsh-Powell varijanta pohlepnog algorita. Od algoritama metaheuristika najbolje rezultate pronalazi genetski algoritam, na osnovu

Graf	Greedy(WP)		SA		PSO		GA	
	$\chi(G)$/reš.	Vreme	$\chi(G)$/reš.	Vreme|iter.	$\chi(G)$/reš.	Vreme|iter.	$\chi(G)$/reš.	Vreme|iter.
myciel3.col	4/4	0.0000391s	4/4	0.000379s|63	4/4	0.017s|4	4/4	0.0039s|2
myciel4.col	5/5	0.00016s	5/6	0.089s|9999	5/7	215.499s|10000	5/5	0.497s|237
myciel5.col	6/6	0.00062s	6/11	2.413s|99999	6/15	90.215s|1000	6/8	53.644s|10000
myciel6.col	7/7	0.0021s	7/21	6.497s|99999	7/34	471.407s|1000	7/14	153.916s|10000
david.col	11/11	90.00148s	11/23	4.511s|99999	11/35	23.187s|100	11/14	88.694s|10000
huck.col	11/11	0.00112s	11/15	3.127s|99999	11/29	493.766s|10000	11/13	69.0016s|10000
jean.col	10/10	0.00108s	10/15	2.927s|99999	10/28	1367.840s|10000	10/12	63.686s|10000

činjenice da pronalazi najmanje vrednosti za broj boja grafa, ali takodje uzimajući u obzir i činjenicu da za razliku od SA algoritma nije pokretan za 100000 iteracija(zbog vremena izvršavanja). Nakon njega sledeći po efikasnosti je algoritam simuliranog kaljenja, koji verovatno zbog proveravanja dopustivosti svakog generisanog rešenja ima lošije performanse od genetskog, koji to za razliku od njega ne radi.

Literatura

[1] A. Paz and S. Moran. “Non deterministic polynomial optimization problems and their approximations”. In: *Theoretical Computer Science* 15.3 (1981), pp. 251–277.

[2] D. J. A. Welsh and M. B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems”. In: *The Computer Journal* 10.1 (Jan. 1967), pp. 85–86.