Predpostavka o Randićevem indeksu in radiusu

Poročilo pri predmetu Finančni praktikum

Avtorja:
Jaka Munda, Anja Žavbi Kunaver

Ljubljana, januar 2019

Kazalo

1 Opis problema

Računalniški program Graffiti je postavil lemo, da za enostaven povezan graf $G = (V, E)$ velja,

$$Ra(G) \geq rad(G) - 1.$$

Domnevo je potrebno testirati na različne načine na manjših in večjih grafih. Z uporabo metahevristične populacije je domnevo potrebno preizkusiti na večjih grafih in upati na njeno ovrgbo.

Opombe:

1. Graf je enostaven, če ne vsebuje zank in je brez vzporednih povezav.
2. Graf je povezan, če lahko iz vsake točke pridemo do vsake druge točke v grafu.
3. Ekscentričnost vozlišča v je razdalja do njegovega najbolj oddaljenega vozlišča; tj. $\max\{d(v, u) : u \in V(G)\}$.
4. Radius grafa $rad(G)$ pomeni minimum ekscentričnosti vozlišč grafa.
5. $Ra(G)$ je Randićev indeks grafa G. Definiran je kot

$$Ra(G) = \sum_{uv \in E(G)} \frac{1}{\sqrt{d(u)d(v)}}.$$

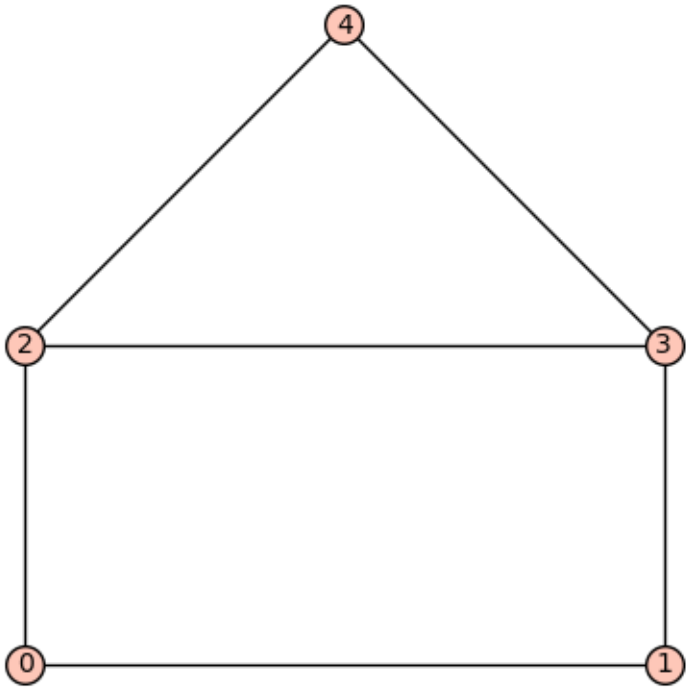
6. $d(x)$ predstavlja stopnjo vozlišča x oz. število povezav, ki imajo vozlišče x za svoje krajišče.

2 Potek dela

Programiranje sva opravila v programu Sage, ki ima že vgrajene funkcije za delo z grafi. Najprej sva napisala program, ki je lemo testiral na manjših enostavnih povezanih grafih. S tem programom sva uspela lemo potrditi za grafe s številom vozlišč $n \leq 9$. Za grafe z večjim številom vozlišč pa program ni deloval, zato sva se dela lotila z metodo populacijske metahevristike, in sicer z genetskim algoritmom.

3 Primer

Za lažje razumevanje prilagava primer enostavnega povezanega grafa s 5 vozlišči.



$$radius = 2$$

$$Ra(G) = \frac{1}{\sqrt{4}} + \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{9}} = \frac{5 + 4\sqrt{6}}{6} \doteq 2.47$$

Vidimo, da na tem grafu lema drži, saj je $2.47 > 2 - 1$.

4 Algoritmi

Oba algoritma sta dostopna na najinem repozitoriju na GitHubu (https://github.com/ZavbiA/Graffiti-conjecture-on-Randic-index-vs.-radius). Algoritem za male grafe se nahaja pod imenom *mali_grafi.ipynb*, genetski algoritem za večje grafe pa pod imenom *genetic_algorithm.ipynb*.

4.1 Algoritem za manjše grafe

Najprej sva definirala funkcijo, ki za graf vrne Randićev indeks po definirani formuli. Funkcija za izračun radiusa grafa je že vgrajena.

```
def randic(graf):
    '''vrne randicev indeks za nek graf'''
    vsota = 0
    povezave = graf.edges()
    stopnje = graf.degree()
    for edge in povezave:
        u, v,_ = edge
        d_u = stopnje[u]
        d_v = stopnje[v]
        vsota += 1/((d_u * d_v)**(1/2))
    return(vsota)
```

Nato sva definirala funkcijo, ki za vse grafe velikosti n in manj preveri, ali domneva drži. Tukaj je bila v pomoč funkcija $list(graphs.nauty\_geng(str(i)+-c"))$, ki vrne seznam vseh enostavnih povezanih grafov velikosti n.

```
def preveri_za_vse(n):
    '''Preveri, če neenakost velja na vseh enostavnih povezanih grafih z n vozlišči in manj.'''
    for i in range(2, n):
        grafi = list(graphs.nauty_geng(str(i)+" -c"))
        for graf in grafi:
            if randic(graf) < graf.radius() - 1:
                return False
    return True
```

V kolikor lema za kakšen graf ne bi držala, bi funkcija vrnila $False$, vendar pa se to v nobenem primeru ni zgodilo. Ta funkcija deluje za grafe do števila vozlišč $n \leq 9$. Ker je tak algoritem zelo potraten, sva se odločila za uporabo genetskega algoritma.

4.2 Algoritem za večje grafe

Genetski algoritem je metoda populacijske metahevristike in spada v razdred razvojnih algoritmov. Temelji na ideji evolucije in naravne selekcije, uporablja pa se za generiranje rešitev v optimizaciji s križanjem, mutacijami ipd.

Sestavila sva genetskim algoritmom, s katerim sva poskušala ovreči domnevo. Ponovno sva najprej definirala Randićev indeks, enako kot pri majhnih grafih.

Nato sva definirala funkcijo $fitness$, ki vrne vrednost neenakosti, torej $Ra(G) - rad(G) + 1$. Če bi vrnila negativno vrednost, bi bila lema ovržena.

```
def fitness(graf):
    '''Vrne vrednost naše neenakosti. Če je vrednost negativna, lema ne drži.'''
    return randic(graf) - graf.radius() + 1
```

Funkcija $fitness\_populacije$ naredi seznam naborov oblike $(graf, njegov fitness)$. Koristna je za hitrejše delovanje algoritma, saj za grafe, za katere je bil $fitness$ že izračunan, le tega ne rabi ponovno računati.

```
def fitness_populacije(populacija):
    '''Naredi seznam naborov oblike (graf, njegov fitnes).'''
    seznam = []
    for graf in populacija:
        fitnes = fitness(graf)
        seznam.append((graf, fitnes))
    return seznam
```

Naslednja funkcija *tournament_selection* med t naključno izbranimi grafi izbere tistega, ki ima najmanjšo vrednost funkcije $fitness$. Najprej naključno izbere enega izmed grafov in ga spravi v 'najbolsi', nato pa pregleduje ostale grafe in če najde boljšega, ga zamenja. Ravno tako hkrati v $fitnes\_najbolsi$ spravi njegov $fitness$. (Populacija je že urejena v seznam naborov oblike $(graf, njegov fitness)$.)

```
def tournament_selection(populacija, t):
    '''Med t naključno izbranimi grafi izbere tistega z najmanjšim fitnessom.'''
    velikost_populacije = len(populacija)
    n = randint(0, velikost_populacije - 1)
    najbolsi, fitnes_najbolsi = populacija[n]
    for i in range(1, t):
        n = randint(0, velikost_populacije - 1)
        izbrani, fitnes = populacija[n]
        if fitnes < fitnes_najbolsi:
            najbolsi = izbrani
            fitnes_najbolsi = fitnes
    return (najbolsi, fitnes_najbolsi)
```

Definirala sva funkcijo za Poissonovo porazdelitev, ki pride v poštev kasneje. Z njo bova izbirala število povezav, ki jih bova funkciji *mutiraj* odstranila oziroma dodala. V funkciji *crossover* nama pove, koliko povezav bova dodala potomcu.

```
def poisson(t = 1, lambd = 1/2):
    '''poissonova porazdelitev'''
    N = 0
    S = 0
    while S < t:
        N += 1
        S += expovariate(lambd)
    return N
```

Funkcija *mutiraj* prejme graf in ga mutira. To naredi tako, da najprej naredi kopijo grafa in naključno izbere neko verjetnost. Če je ta verjetnost $\leq \frac{1}{3}$, doda povezavo, če je $> \frac{1}{3}$ in $\leq \frac{2}{3}$, odstrani povezavo in če je $> \frac{2}{3}$, doda in odstrani povezavo. Ko odstranjujemo povezave, moramo biti pozorni, da graf ostane povezan. V kolikor ni več povezan, je potrebno povezavo dodati nazaj. *Poissonovo* funkcijo sva uporabila za določitev maksimalnega števila povezav,

ki jih grafu lahko dodamo ali odstranimo.

```
def mutiraj(graf):
    '''Funkcija mutira graf. Z verjetnostjo p = 1/3 doda povezavo, z p = 1/3 odstrani povezavo in z p = 1/
3 doda in odstrani povezavo.'''
    kopija = Graph(graf)
    verjetnost = random()
    plus_povezave = poisson(lambd = 2) # max število povezav, ki jih bomo dodali
    minus_povezave = poisson(lambd = 2) # max število povezav, ki jih bomo odstranili
    if verjetnost <= 1/3:
        for k in range(plus_povezave):
            u, v = kopija.random_vertex(), kopija.random_vertex()
            if u != v:
                kopija.add_edge(u, v)
        return kopija
    elif verjetnost > 1/3 and verjetnost <= 2/3:
        for k in range(minus_povezave):
            povezava = kopija.random_edge()
            kopija.delete_edge(povezava)
            if not kopija.is_connected(): # če graf ni več povezan, ko odstranimo povezavo, jo moramo doda
ti nazaj
                kopija.add_edge(povezava)
        return kopija
    elif verjetnost > 2/3:
        for k in range(plus_povezave):
            u, v = kopija.random_vertex(), kopija.random_vertex()
            if u != v:
                kopija.add_edge(u, v)
        for k in range(minus_povezave):
            povezava = kopija.random_edge()
            kopija.delete_edge(povezava)
            if not kopija.is_connected():
                kopija.add_edge(povezava)
        return kopija
```

Funkcija *crossover* prejme dva grafa (a in b) in ju križa med seboj ter vrne njunega potomca. Najprej naredi podgraf grafa a, kjer je vsako vozlišče vsebovano z verjetnostjo $\frac{1}{2}$ in prav tako naredi podgraf grafa b. Sledi pogoj, da sta oba grafa povezana, imata skupaj n vozlišč in da noben od njiju ni prazen. V *potomec* najprej shrani povezave, ki med grafoma niso skupne in v *nove_povezave* shrani nove povezave, število katerih se seveda veča s številom n. Nove povezave naredi s pomočjo *Poissonove* funkcije. Nato z njimi oblikuje *potomca*. Vmes sva uporabila vgrajeno funkcijo *.relabel*(), ki poskrbi za primerno oštevilčenje vozlišč v novem grafu.

```
def crossover(a, b):
    '''Križa dva grafa med sabo in vrne njunega potomca.'''
    n = len(a)
    while True:
        podgraf_a = a.random_subgraph(0.5) # vrne podgraf grafa a, kjer je vsako vozlišče vsebovano z verj
etnostjo 1/2
        podgraf_b = b.random_subgraph(0.5) # vrne podgraf grafa b, kjer je vsako vozlišče vsebovano z verj
etnostjo 1/2
        if len(podgraf_a.vertices()) + len(podgraf_b.vertices()) == n and len(podgraf_a.vertices()) >= 1 a
nd len(podgraf_a.vertices()) < n and podgraf_a.is_connected() and podgraf_b.is_connected():
            # z zgornjim pogojem želimo da sta oba podgraf povezana in da imata skupaj n vozlišč in da ni
 en podgraf prazen drugi pa ima n vozlišč
            podgraf_a.relabel()
            podgraf_b.relabel()
            potomec = podgraf_a.disjoint_union(podgraf_b) # povezave, ki niso skupne med grafoma
            nove_povezave = poisson(lambd = log(n/2)) # večji kot je n, več povezav bomo dodal
            for k in range(nove_povezave):
                u, v = podgraf_a.random_vertex(), podgraf_b.random_vertex()
                potomec.add_edge((0, u), (1, v))
            potomec.relabel()
            break
    return potomec
```

Funkcija *zacetna_populacija* je potrebna zato, da se naredi začetno populacijo (seznam grafov), kjer imajo grafi n vozlišč. S parametrom *velikost* je določena velikost populacije, s parametrom n pa število vozlišč grafa. Vsaka povezava v grafu je z neko naključno verjetnostjo. Funkcija poskrbi tudi, da so vsi grafi povezani.

```
def zacetna_populacija(velikost, n):
    '''Naredi začetno populacijo, kjer imajo grafi n vozlišč.'''
    populacija = []
    trenutna_velikost = 0
    while trenutna_velikost < velikost:
        graf = graphs.RandomGNP(n, random()) # Naredi nek naključen graf z n vozlišči. Vsaka povezava je v
 grafu z neko naključno verjetnostjo.
        if graf.is_connected():
            populacija.append(graf)
            trenutna_velikost += 1
    return populacija
```

Funkcija *min_fitness* prejme seznam grafov, med katerimi poišče tistega, ki ima najmanjšo vrednost funkcije *fitness*. To pa zato, ker manjša kot je ta vrednost, večja je verjetnost, da bo lema ovržena. Želiva namreč priti pod vrednost 0, večje vrednosti pa lemo le potrdijo za en določen graf. Poleg grafa funkcija vrne tudi njegov *fitness*.

```
def min_fitness(seznam):
    '''V seznamu grafov poišče graf z najmanjšim fitnessom.'''
    najbolsi, najbolsi_fitnes = seznam[0]
    for nabor in seznam[1:]:
        graf, fitnes = nabor
        if fitnes < najbolsi_fitnes:
            najbolsi_fitnes = fitnes
            najbolsi = graf
    return (najbolsi, najbolsi_fitnes)
```

S funkcijo *nova_populacija* iz obstoječe populacije narediva novo populacijo. To narediva z *while* zanko z mutacijami in križanjem. Funkcija je že tako napisana, da takoj naredi seznam naborov.

```
def nova_populacija(populacija,t):
    '''S križanjem in mutacijo grafov iz podane populacije naredi novo populacijo.'''
    nova_populacija = []
    trenutna_velikost = 0
    velikost_stare = len(populacija)
    while trenutna_velikost < velikost_stare:
        graf1 = tournament_selection(populacija, t)
        graf2 = tournament_selection(populacija, t)
        mutacija1 = mutiraj(graf1[0])
        mutacija2 = mutiraj(graf2[0])
        fit_mut1 = fitness(mutacija1)
        fit_mut2 = fitness(mutacija2)
        krizan_graf = crossover(mutacija1, mutacija2)
        fit_kriz = fitness(krizan_graf)
        minimum = min_fitness([(krizan_graf, fit_kriz), (mutacija1, fit_mut1), (mutacija2, fit_mut2), graf
1, graf2])
        nova_populacija.append(minimum)
        trenutna_velikost += 1
    return nova_populacija
```

Še zadnja funkcija *genetic_algorithm* v vsaki ponovitvi s križanjem in mutiranjem naredi novo populacijo. Če v tej populaciji najde graf, za katerega je vrednost *fitness* manjša od nič, vrne 'Lema ne drži' in pripadajoči graf. Če

se to ne zgodi pri nobenem grafu, vrne 'Ne najdem protiprimera'. Argument *cas_izvajanja* je podan zato, da program ne teče v neskončnost. n je velikost populacije (število grafov), k pa je število vozlišč vsakega grafa.

```
def genetic_algorithm(n, k, cas_izvajanja, t=4):
    '''Funkcija v vsaki ponovitvi naredi novo populacijo z križanjem in mutiranjem.'''
    populacija = zacetna_populacija(n, k)
    populacija = fitness_populacije(populacija)
    for i in range(cas_izvajanja):
        for nabor in populacija:
            graf, fitnes = nabor
            #print(round(fitnes, 0))
            if fitnes < 0:
                print('Lema ne drži!')
                return graf
        populacija = nova_populacija(populacija,t)
    return 'Ne najdem protiprimera.'
```

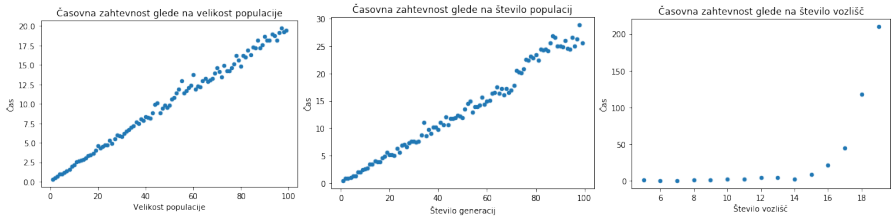
Lemo sva nato testirala z vnašanjem različnih vrednosti v zadnjo funkcijo in ob tem spremljala časovno zahtevnost.

5 Ugotovitve

5.1 Splošne ugotovitve

Ugotovila sva, da imajo polni grafi z več kot enim vozliščem radius vedno enak 1. Iz tega dokaj očitno sledi, da neenakost velja (celo stroga neenakost). Z algoritmom za male grafe sva uspela dokazati, da neenakost drži za vse enostavne povezane grafe s številom vozlišč $n \leq 9$. Z genetskim algoritmom za večje grafe pa nama ni uspelo najti protiprimera. Na podlagi tega leme ne moreva ovreči ali potrditi v celoti.

5.2 Časovna zahtevnost



Z večkratnimi poskusi sva prišla do ugotovitve, da je časovna zahtevnost najinega algoritma enaka $O(\check{c}as\_izvajanja * velikost\_populacije * n^2)$. Čas izvajanja se torej najbolj poveča, če povečava število vozlišč. To pa zato, ker ima funkcija *randic* casovno zahtevnost $O(n^2)$ in posledično ima isto časovno zahtevnost tudi funkcij *fitness*. Ker je v funkciji *nova_populacija* potrebno za vsak graf izračunati nov *fitness*, je časovna zahtevnost te funkcije enaka $O(velikost\_populacije * n^2)$. V genetskem algoritmu je sicer odvisno, koliko časa se izvaja, ampak je potrebno v vsaki ponovitvi narediti novo populacijo, zato ima v najinem primeru algoritem časovno zahtevnost $O(\check{c}as\_izvajanja * velikost\_populacije * n^2)$.

6 Literatura

[1] M. Cygan, M. Pilipczuk, R. Škrekovski, *On the Inequality between Radius and Randic Index for Graphs*, MATCH, 2011.

[2] S. Luke, *Essentials of Metaheuristics*, Department of Computer Science, George Mason University, Online Version 2.2, 2015.