Graffiti conjecture 299

Poročilo projekta

Projekt v povezavi s predmetom Operacijske raziskave

Avtorja:
Stefan Đekanović, Katja Marina

Ljubljana, november 2018

Kazalo

1 Opis projekta

Graffiti je računalniški program, ki programira grafsko-teoretične domneve. Program pozna določene grafe in je sposoben ovrednotiti določene formule oblikovane iz teoretičnih invariant grafov. Če nobeden od grafov, s katerimi je seznanjen Graffiti, ni protiprimer za formulo, to formulo imenujemo domneva. Glavni problem je število teh domnev, še posebej tistih, ki so trivialne.
Najin projekt je testiranje domneve 299, ki pravi:

Če je G enostaven povezan graf, potem je

$$g_t(G) \leq Annihilation(G) + |S|,$$

kjer je S množica vozlišč, ki imajo stopnjo 2.

2 Definicije

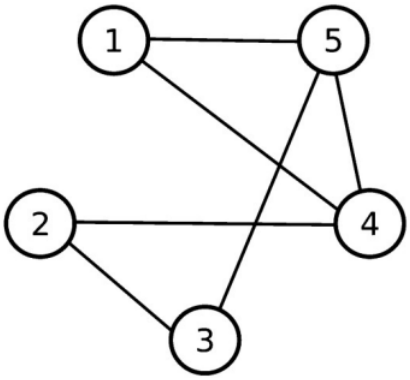
- Graf G je *povezan*, če za poljubni vozlišči $u, v \in G$ obstaja pot od u do v.
- Graf G je *enostaven*, če nima zank (začetna točka ni tudi njena končna točka) in vzporednih povezav (povezavi, ki imata skupno začetno in končno točko).
- *Dominantna množica* oziroma *dominating set* je podmnožica vozlišč grafa G, za katero velja, da dominira vse ostale točke v grafu. Točka v grafu dominira drugo točko, če sta enaki, ali pa sta med seboj povezani (sosedni).
- *Totalna dominantna množica* oziroma *total dominanting set* $D(G)$ grafa G je dominantna množica vozlišč grafa G, v kateri ima vsako vozlišče iz dominantne množice tudi vsaj enega soseda iz te množice.
- *Moč totalne dominantne množice* označimo z $g_t(G)$.
- *Totalno dominantno število grafa G* je moč najmanjše totalne dominantne množice grafa.
- *Stopnja točke* v neusmerjenem grafu G je število vseh njenih sosednih točk.
- *Annihilation(G)* oziroma »uničenje« grafa G – $A(G)$ definiramo tako: Naj ima graf G n vozlišč in naj bo $d_1, d_2, ..., d_n$ nepadajoče zaporedje stopenj grafa G. $A(G)$ je največje celo število k, da je vsota prvih k členov zaporedja največ $\frac{(d_1+d_2+...+d_n)}{2}$.

3 Načrt dela

Delo sva začela s pisanjem pomožnih funkcij, ki so nama v nadaljevanju pomagale pri glavni funkciji za testiranje hipoteze. Funkcija na koncu vrne, ali je bila domneva ovržena, ali pa jo potrdi. Vendar pa ta program učinkovito deluje le za (približno) $n \in \{1, 2, ..., 8\}$ zaradi časovne zahtevnosti, zato sva se lotila iskanja protiprimera z eno od populacijskih metahevrističnih metod, in sicer z genetskim algoritmom.

4 Primer

Prikazala bova primer testiranja domneve na manjšem enostavnem povezanem grafu s 5 vozlišči.



$$S = \{1, 2, 3\} \Rightarrow |S| = 3$$
$$g_t(G) = |\{3, 5\}| = 2$$
$$d_1 \leq d_2 \leq d_3 < d_4 < d_5 :$$
$$d_1 = 2, d_2 = 2, d_3 = 2, d_4 = 3, d_5 = 3$$
$$\frac{d_1 + d_2 + d_3 + ... + d_5}{2} = \frac{2 + 2 + 2 + 3 + 3}{2} = 6$$
$$d_1 + d_2 + d_3 = 6 \Rightarrow k = 3, \text{torej je } A(G) = 3.$$

Najina domneva pravi:

$2 \leq 3 + 3 = 6$, torej v tem primeru drži.

5 Pomožne funkcije

Najina enačba je sestavljena iz treh spremenljivk, ki jih bova izračunala s pomočjo pomožnih funkcij.

5.1 degrees(G)

Najprej sva definirala funkcijo, ki vrne seznam vozlišč grafa, s katero si bova pomagala pri računanju *annihilation(G)*.

```
def degrees(G):
    stopnje = G.degree_sequence()
    return stopnje
```

5.2 powerset_S

Druga pomožna funkcija se imenuje *powerset_S*, ki iz seznama stopenj vozlišč vrne število vozlišč stopnje 2.

```
def powerset_S(G):
    velikost_S = degrees(G).count(2)
    return velikost_S
```

5.3 annihilation(G))

Funkcija *annihilation(G)* najprej vzame seznam stopenj vozlišč, in ga obrne, tako da so v naraščajočem zaporedju. Potem izračuna polovično vsoto vseh stopenj, kar rabimo v nadeljevanju funkcije. Z zanko for se sprehodimo po naraščajočem seznamu stopenj, in ko je vsota i-tih stopenj še manjša ali pa enaka polovični vsoti stopenj, vrne število i.

```
def annihilation(G):
    stopnje = degrees(G)
    stopnje_obr = stopnje[::-1]
    vel = int(sum(stopnje_obr)/2)
    sest = 0
    if stopnje_obr == [0]:
        return 1
    else:
        for i, k in enumerate(stopnje_obr):
            sest += k
            if sest > vel:
                return i
```

5.4 total_domination_number(G))

Naslednja pomožna funkcija je *total_domination_number(G)*, ki z mešanim celoštevilskim linearnim programom izračuna totalno dominantno število grafa. Pri tem sva ločila še primer, ko ima graf samo eno vozlišče in ima totalno dominantno število 0, in pa primer, ko sta vozlišči dve, kjer je totalno dominantno število 1.

```
def total_domination_number(G):
    if len(G.vertices()) == 1:
        return 0
    p = MixedIntegerLinearProgram(maximization = False)
    b = p.new_variable(binary = True)
    p.set_objective( sum([b[v] for v in G]) )

    for u in G:
        p.add_constraint( sum([b[v] for v in G[u]]) >= 1 )

    p.solve()
    b = p.get_values(b)
    return len([v for v,i in b.items() if i])
```

6 Glavna funkcija

Vse pomožne funkcije, ki sva jih definirala, na koncu združiva v funkciji *testiranje_domneve*, ki deluje za majhne grafe, do približno n = 8. Če za n vstaviva večje število, najin algoritem deluje počasneje zaradi velike časovne zahtevnosti.

```
def testiranje_domneve(n):
    grafi = list(graphs.nauty_geng(str(n)+" -c"))
    for graf in grafi:
        if total_domination_number(graf) > (annihilation(graf))
+ (powerset_S(graf)):
            show(graf)
            print(annihilation(graf))
            return ("Domneva je ovrzena.")
    return "Domneva ni ovrzena."
```

7 Genetski algoritem za iskanje protiprimera

Glavna funkcija v nobenem primeru ne ovrže, za število vozlišč $n \geq 9$ (približno) pa je bistveno prepočasna in neučinkovita, zato se morava problema

lotiti še drugače, to pa je z uporabo metahevristike, bolj natančno populacijske metode.

Genetski algoritem temelji na ideji evolucije in naravne selekcije. Če hoče populacija preživeti, se mora konstantno izboljševati in prilagajati okolju. Poleg močnejših osebkov v populaciji mora preživeti tudi nekaj slabih, saj se s tem ohranja raznolikost. V našem primeru jo nujno potrebujemo, saj lahko brez raznolikosti v populaciji hitro skonvergiramo k lokalni rešitvi. S tem, da tudi nekaj slabih osebkov preživi, in se naprej razmnožujejo, torej zagotovimo, da pridemo s časom do globalne rešitve.

Najprej določimo velikost vsake generacije in število vozlišč grafa, nato pa začnemo s prvo generacijo, kjer osebke (grafe) določimo naključno s funkcijo $graphs.randomGNP$. Funkcija $graphs.randomGNP$ zahteva še argument p, ki nam pove, s kakšno verjetnostjo naj bojo vozlišča povezana. Fitnes vsakega grafa določimo preko hipoteze s funkcijo $fitness(G)$:

$$0 \leq annihilation(G) + |S| - g_t(G)$$

$$fitness(G) = annihilation(G) + |S| - g_t(G)$$

Boljši graf je torej tisti, ki je bližji številu 0. Če najdemo graf s fitnesom manjšim od 0, je naša domneva ovržena. Ta graf pa je globalna rešitev problema. Ko vsakemu osebku določimo fitnes, izberemo za razmnoževanje najboljše osebke. Poleg teh pa naključno določimo še nekaj slabih osebkov, da ohranimo raznolikost v populaciji.

Nato sledi proces razmnoževanja in mutacije. Najprej izberemo osebke, ki bodo ustvarili potomce. Osebek je lahko v procesu izbran večkrat. Večjo verjetnost za izbiro imajo osebki z močnejšim fitnesom. Pri vsakem razmnoževanju se določi en potomec, ki ga dodamo k naši populaciji. Kolikokrat pride do razmnoževanja osebkov v populaciji nam določi Poissonov proces.

Primer: Simulacija Poissonovega procesa s časom $t = 50$ in intenzivnostjo $\lambda = 1/2$, kjer je 50 velikost populacije.

[11, 14, 15, 17, 18, 18, 18, 18, 19, 19, 19, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 32, 32, 33, 33, 33, 33, 34, 34, 36, 36, 37]

Po "crossoverju" je pomembno še, da pride do mutacije nekaterih osebkov. To nam zagotovi še večjo raznolikost v populaciji. Naključno izberemo nekaj osebkov in jih "mutiramo" tako, da jim dodamo ali odstranimo povezave.

Želimo si, da je mutacij, kjer odstranimo/dodamo samo eno povezavo, največ. Sledijo mutacije z dvema povezavama, še redkeje so mutacije s tremi povezavami itd. Pri tem nam pomaga Poissonov proces s časom $t = 1$.

Primer: Simulacija Poissonovega procesa s časom $t = 1$ in intenzivnostjo $\lambda = 1/2$.

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

Na tem koraku imamo v populaciji zaradi novih potomcev več osebkov kot na začetku, kjer so sedaj nekateri osebki tudi mutirani. Ko celoten postopek ponovimo, se populacija zmanjša na začetno velikost, saj za preživetje in razmnoževanje spet izberemo najboljše in nekaj naključno pridobljenih osebkov. V tej populaciji so sedaj osebki, ki so bližje naši rešitvi problema.

8 Zaključek in ugotovitve

Vse funkcije sva združila v funkcijo *testGA*, ki za parametre vzame število vozlišč n, velikost populacije *pop_size*, poljubno izbrano število najboljših za reprodukcijo *best* in pa število generacij, torej *stevilo_generacij*. Funkcija sproti prikazuje fitnes populacije za vsako generacijo, urejen po vrsti od najboljšega do najslabšega. Tako kot funkcija *testiranje_domneve* vrne, ali je domneva ovržena, in smo našli primer, ali pa ni. Ker pa z večkratnim testiranjem nisva dobila nobenega fitnesa manjšega od 0, protiprimera nisva našla, in ga tudi ne bova.