Graffiti conjecture 143

Poročilo

Projekt v povezavi s predmetom Operacijske raziskave

Avtorici:
Jona Gričar, Tjaša Mehle

Ljubljana, november 2018

Kazalo

1 OPIS PROBLEMA

Graffiti je računalniški program, ki je namenjen generiranju matematičnih domnev oziroma odprtih problemov. Nekatere izmed domnev se izkažejo za resnične, nekatere pa za napačne.
Najina naloga pri tem projektu je, da ovrževa ali potrdiva domnevo številka 143.

Domneva številka 143 pravi:

Vsak enostaven povezan graf G ustreza zahtevi

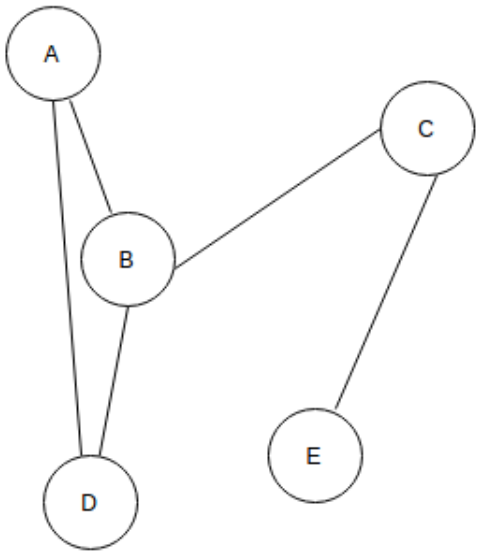
$$tree(G) \geq \frac{g(G)+1}{\sigma(G)}.$$

Nekaj opomb:

- graf je **ENOSTAVEN**, če nima niti zank (povezava, katere začetna točka je tudi končna točka) niti vzporednih povezav (dve povezavi, ki imata skupno začetno in končno točko),
- graf je **POVEZAN**, če za poljubni vozlišči $u, v \in V(G)$ obstaja sprehod (oziroma pot) od u do v,
- $g(G)$ je notranji obseg grafa G (ang. girth) - tj. dolžina (število vozlišč) najkrajšega cikla grafa G, v primeru,da v grafu ni cikla je $g(G) = \infty$
- $\sigma(G)$ je druga najmanjša stopnja v zaporedju stopenj grafa G, ki je enako $d_1 \leq d_2 \leq ... \leq d_{n-1} \leq d_n$,
- $tree(G)$ - število vozlišč v največjem induciranem drevesu grafa G (velikost največje podmnožice vozlišč grafa G, ki inducira drevo - torej da je graf na izbranih vozliščih in vseh povezavah med njimi povezan in brez ciklov),
  - drevo je graf, pri katerem za vsak par vozlišč u in v obstaja natanko ena pot od u do v,
  - graf $H = (U, F)$ je podgraf grafa $G = (V, E)$, $H \subseteq G$, če velja $U \subseteq V$ in je$F \subseteq E$,
  - H je induciran podgraf grafa G, če $H \subseteq G \wedge \forall e = xy :$ $(x, y \in V(H) \Rightarrow e \in E(H))$.

2 PRIMER

G = enostaven povezan graf s petimi vozlišči



- $g(G) = 3$
- $\sigma(G) = 2$
- $tree(G) = 4$

$$tree(G) \geq \frac{g(G)+1}{\sigma(G)}$$
$$4 \geq \frac{3+1}{2}$$
$$4 \geq 2$$

domneva drži

3 NAČRT DELA

- **Testiranje hipoteze:**
  Dela sva se lotili tako, da sva v Sage najprej napisali pomožne funckije, ki jih potrebujemo v neenačbi oziroma domnevi. Na koncu sva definirali še glavno funkcijo, ki s pomočjo prej definiranih pomožnih funkcij preveri domnevo $tree(G) \geq \frac{g(G)+1}{\sigma(G)}$. Končna funkcija nam tako tudi sporoči oziroma vrne izpis, ali je domneva ovržena ali ne.

- **Iskanje protiprimera:**
  V primeru, da pri testiranju hipoteze domneva ne bo ovržena, se bova lotili iskanja protiprimera. Pri tem si bova pomagali z metahevristiko.

4 TESTIRANJE HIPOTEZE

4.1 POMOŽNE FUNKCIJE

Domneva oziroma neenačba vsebuje tri spremenljivke (parametre) za vsak graf G, ki jih je potrebno predhodno izračunati. Zato sva najprej definirali tri pomožne funkcije, ki za graf G vrnejo iskan podatek.

1. **notranji obseg grafa**

   - IDEJA: Najprej preverimo, ali ima graf vsaj en cikel - v primeru da ga ni, nastavimo vrednost $g(G) = \infty$. V nasprotnem primeru prevedemo na linerno programiranje in sicer iščemo:

   $$\begin{gathered} \min \textstyle\sum_{v \in G} b_v \\ \text{p.p.} \\ \forall u \in G; \textstyle\sum_{uv \in G} b_u v = 2b_v \\ \textstyle\sum_{v \in G} b_v \geq 1 \end{gathered}$$

   Spremenljivko p nastavimo na mixed integer linear program in ker iščemo minimum, nastavimo maximization na false. S pomočjo novega slovarja b, ki sprejme vrednosti 0 in 1 in vgrajene funkcije .set_objective, podamo funkcijo, ki jo želimo minimizirati. Z .add_constraint dodamo željena pogoja in nato s p.solve() dobimo željeno vrednost.

- KODA:

```
def girth(G):
    if G.cycle_basis() == []:
        return oo
    else:
        p = MixedIntegerLinearProgram(maximization = False)
        b = p.new_variable(binary = True)
        p.set_objective(sum([b[v] for v in G]))
        p.add_constraint(sum([b[v] for v in G]) >= 1)

        for v in G:
            edges = G.edges_incident(v, labels = False)
            p.add_constraint(sum([b[Set(e)] for e in edges]) == 2 * b[v])

    return p.solve()
```

2. **druga najmanjša stopnja**

- IDEJA: Iz seznama stopenj vseh vozlišč, ki je urejen po velikosti od največje do najmanjše stopnje, vzamemo predzadnji element. Seznam dobimo tako, da uporabimo že vgrajen ukaz *degree_sequence*(), ki nam za vozlišča vrne seznam stopenj.
  Opomba: privzeli sva, da je druga najmanjša stopnja definirana kot drugi vnos v seznamu, ki je urejen po velikosti od najmanjše do največje (možne ponovitve) in ne kot drugi vnos v seznamu vrednosti.

- KODA:

```
def second_smallest_degree(G):
     stopnje = G.degree_sequence()
     return stopnje[-2]
```

3. **velikost največje podmnožice vozlišč grafa G, ki inducira drevo**

- IDEJA: Za izračun uporabimo backtracking oziroma rekurzivno preiskovanje. Najprej ustvarimo množico drevesa, ki hrani že pregledane množice vozlišč, da istih množic ne preiščemo večkrat - če je bilo drevo že pregledano, rekurzivni klic končamo.
  Da pa bi lahko uporabili backtracking, potrebujemo dodatno

funkcijo znotraj funkcije za $tree(G)$. Ta dodatna funkcija sprejme tri argumente: trenutno drevo T (oziroma množico njegovih vozlišč), množico kandidatov za razširitev S (to so vozlišča, ki imajo v T natanko enega soseda) in množico prepovedanih vozlišč X (to so vozlišča, ki imajo v T več kot enega soseda). Velikost sprva nastavimo kar na velikost trenutnega drevesa T. Za vsakega naslednjega kandidata, se ustvari novo drevo TT - če je drevo že pregledano, potem nadaljujemo, drugače pa novo drevo dodamo med pregledana.
Vsakič potrebujemo tudi množico sosedov N. Za vsakega naslednjega kandidata, se ustvari nova množica kandidatov SS: med kandidate dodamo sosede novega vozlišča, ki niso že v drevesu in niso že sosedi katerega od njih, in odstrani tiste kandidate, ki so tudi sosedi novega vozlišča. Prav tako se poveča množica prepovedanih vozlišč XX. Oznake $+$, $-$, $\&$ in $\wedge\wedge$ pomenijo unijo, razliko, presek in simetrično razliko množic. Če je velikost novega drevesa večja od trenutno največjega, kar nam podaja max, potem to število povečamo. Dodatna funkcija nam vrne največjo možno velikost.
Na koncu algoritma dobimo absolutno največjo velikost, ko so preiskana vsa vozlišča in vse možnosti dreves v grafu G.
Opomba: ta algoritem ima eksponentno časovno zahtevnost in za večje grafe ne bo končal v doglednem času.

- KODA

```
def tree(G):
    drevesa = set()
    def tree_backtrack(T, S, X):
        velikost = len(T)
        for v in S:
            TT = T + Set([v])
            if TT in drevesa:
                continue
            drevesa.add(TT)
            N = Set(G[v])
            SS = (S ^^ N) - TT - X
            XX = X + (S & N)
            velikost = max(velikost, tree_backtrack(TT, SS, XX))
        return velikost
    return max(tree_backtrack(Set([u]), Set(G[u]), Set()) for u in G)
```

4.2 GLAVNA FUNKCIJA

S pomočjo pomožnih funkcij, ki sva jih združili v glavno, se da preveriti domnevo za vse enostavne povezane grafe z n vozlišči, kjer je $n \in \{1, 2, 3, 4, 5, 6, 7, 8\}$.
Seznam vseh enostavnih povezanih grafov (EPG) dobimo s pomočjo vgrajene funkcije $list(graphs.nauty\_geng(str(n) + -c"))$.

Opomba: algoritem dobro deluje za majhne n, pri večjih pa je časovna zahtevnost velika in zato ne konča v doglednem času.

```
def testiranje_hipoteze(n):
    EPG = list(graphs.nauty_geng(str(n) + " -c"))
    for i in range(len(EPG)):
        if girth(EPG[i]) < oo:
            if tree(EPG[i]) < (girth(EPG[i]) + 1) / second_smallest_degree(EPG[i]):
                return "DOMNEVA JE OVRZENA"
    return "DOMNEVA NI OVRZENA"
```

4.3 UGOTOVITVE

- Pri vseh enostavnih povezanih grafih, ki so brez cikla (to so ravno drevesa), je $g(G) = \infty$ in posledično domneva za take grafe **ne drži**. Zato sva testiranje omejili na vse enostavne povezane grafe, ki imajo vsaj en cikel. (Druga možnost za $g(G)$ pa je ta, da 'ožina' ni definirana in v tem primeru celotna domneva prav tako ni definirana.)
- Za vse enostavne povezane grafe z vsaj enim ciklom pri testiranju za vsak izbran n **domneva ni bila ovržena**. To pomeni, da je potrebno poiskati protiprimer, če morda obstaja.

5 ISKANJE PROTIPRIMERA

Kot že rečeno, najin poskus testiranja domneve na grafih z vozlišči, kjer $n \in \{1, 2, 3, 4, 5, 6, 7, 8\}$, ni obrodil sadov - domneve ni bilo mogoče ovreči. Zato sva se odločili, da poskušava poiskati protiprimer. Iskanja sva se lotili na dva načina - uporabili sva **genetic algorithm** in **simulated annealing**.

1. **GENETIC ALGORITHM:**

   - IDEJA: Zgeneriramo naključno populacijov enostavnih povezanih grafov z n vozlišči (velikost populacije določimo sami). Tu nam prav pride že vgrajena funkcija $graphs.RandomGNP(n, p)$, ki zgenerira naključen graf z n vozlišči, pri katerem sta dva vozlišča povezana z verjetnostjo p.
     Grafe ocenjujemo glede na to, kako majhna je naslednja razlika: $tree(G) - (girth(G) + 1)/second\_smallest\_degree(G)$ - manjša kot je, boljši je graf ("fitness"grafa). To preverimo na vsakem grafu v populaciji in populacijo uredimo po velikosti od najboljših do najslabših grafov. Nato iz te populacije izberemo željeno število najboljših grafov in še nekaj naključnih. Dobimo manjšo populacijo in na njej naredimo naključne mutacije, tj. naključno izberemo grafe iz populacije in jim dodamo ali odstranimo povezavo ali pa iz dveh naključnih grafov vzamemo njuna subgrafa tako da je skupna vsota vozlišč enaka n in ju povežemo z naključnim številom povezav, da dobimo nov graf z n-vozlišči in ga dodamo v populacijo. Sedaj imamo novo populacijo, na kateri spet vse ponovimo.
     Opomba: pri mutacijah moramo paziti, da ostaneta pogoja, da je graf povezan in da vsebuje vsaj en cikel.

   - KODA: GitHub / Graffiti-conjecture-143

2. **SIMULATED ANNEALING:**

   - IDEJA: Zgeneriramo naključen enostaven povezan graf z n vozlišči. Za množico sosedov definiramo množico vseh mutacij začetnega grafa (tj. dodamo ali odstranimo povezavo med dvema naključnima vozliščema). Iz množiče sosedov izberemo naključnega "soseda" in primerjamo fitness začetnega grafa s fitnessom soseda (v kodi uporabljeno ime E_sosed). V primeru da bo fitness soseda manjši, se bomo premaknili na soseda z verjetnostjo 1. V nasprotnem primeru je premik na soseda odvisen od verjernostne funkcije, ki jo definiramo kot : $p = e^{(-(E\_sosed - E\_osebek)/(T))}$. Ta je odvisna tudi od parametra T, ki označuje temperaturo - to je na začetku potrebno čimbolj optimalno nastaviti in potem po vsakem premiku izvesti "cooling_schedule". Te korake ponavljamo dokler ne najdemo optimalne rešitve oziroma dokler se program ne izčrpa.

   - KODA: GitHub / Graffiti-conjecture-143

6 ZAKLJUČEK

Med izdelovanjem projekta sva ugotovili, da je sama domneva 143 izredno težka za dokazovanje oziroma preverjanje. Prvi problem je ta, da je algoritem za testiranje hipoteze primeren le za zelo majhne n - za večje n algoritem ne konča v doglednem času in nam žal ne koristi. Algoritem sva zato uporabili za majhne n, kjer $n \in \{1, 2, 3, 4, 5, 6, 7, 8\}$. Sprva sva dobivali rezultat, da je domneva ovržena. Ugotovili sva, da takšen rezultat povzročajo grafi brez ciklov in se zato potem omejili le na grafe, ki vsebujejo vsaj en cikel. Prilagojen algoritem je za izbrane n vsakič vrnil rezultat, da domneva ni ovržena. To pa naju je privedlo do tega, da sva se odločili poiskati protiprimer, če bo to le možno. Poizkusile sva *genetic algorithm* in *simulated annealing*, a nama žal pri nobenem načinu ni uspelo sestaviti popolnega algoritma in posledično nisva našli protiprimera. Na tem mestu lahko projekt zaključiva - domnevo za majhne n lahko potrdiva z empirično podlago, za večje n pa problem domneve 143 še vedno ostaja odprt - prav tako tudi problem uspešnega iskanja protiprimera.