Univerza v Ljubljani
Fakulteta za matematiko in fiziko
Finančna matematika 1. stopnja

Projekt pri predmetu Finančni praktikum

The firefighter problem

Karolina Šavli, Klara Travnik

Ljubljana, januar 2023

Kazalo

- Opis in formulacija problema (opis, predstavitev clp-ja, koda clp-ja, alijeproblemkončan, caspotreben) KLara
- Vizualizacija problema (koda barvanja, primer G2) Karolina
- (Časovna zahtevnost algoritma):
- Testiranje programa glede na število vozlišč grafa, komentar grafov Karolina
- Sklep in zaključek (uporaba problema gasilca v praksi (hiše, bolezen)) Klara

1 Opis in formulacija problema

The firefighter problem oziroma *problem gasilca* je optimizacijski problem, katerega cilj je minimiziranje števila pogorelih vozlišč na grafu.
Vhodni podatki problema so:

- graf G,
- množica vozlišč $B_{init} \subseteq V(G)$, na katerih v času 0 izbruhne požar,
- število gasilcev D.

V vsaki časovni enoti ($t > 0$) gasilci izberejo nepogorela vozlišča, ki jih bodo rešili tako, da čim bolj omejijo požar. Ta se razširi le na sosednja vozlišča pogorelih v prejšnji časovni enoti, ki jih gasilci niso uspeli rešiti. Proces se ponavlja dokler požar ni zajezen.

Opisani problem sva v programu CoCalc, v programskem jeziku SageMaths zapisali kot **celoštevilski linearni program (CLP)**:

```
def clp(G, B, gasilci):
''' vhodni podatki:
     G           izbran graf
     B           vozlišča, ki na začetku zgorijo
     gasilci     število gasilcev, ki v vsakem koraku gasijo požar
 izhodni podatki:
     seznam oblike [število časovnih enot, pogorela/burnt vozlišča po časih,
     zaščitena/defended vozlišča po časih] '''
cas = 10
while True:
    casi = range(1, cas+1) # uprabljamo pri zankah

    # CLP:
    p = MixedIntegerLinearProgram(maximization=False) # CLP
    d = p.new_variable(binary=True) # spremenljivka, defended
    b = p.new_variable(binary=True) # spremenljivka, burnt

    p.set_objective(sum(b[i, cas] for i in G)) # minimiziramo število
    pogorelih vozlišč na koncu

    for t in casi:
        for i in G:
            for j in G[i]: # j je številka v seznamu vozlišča i, sosed od i
                p.add_constraint(b[i,t] + d[i,t] - b[j,t-1] >= 0)
            p.add_constraint(b[i,t] + d[i,t] <= 1)
            p.add_constraint(b[i,t] - b[i,t-1] >= 0)
            p.add_constraint(d[i,t] - d[i,t-1] >= 0)
        p.add_constraint(sum((d[i,t] - d[i,t-1]) for i in G) <= gasilci)
```

```
        for i in G:
            p.add_constraint(b[i,0] == (1 if i in B else 0))
            p.add_constraint(d[i,0] == 0)

        k = p.solve()
        l = p.get_values(b)
        m = p.get_values(d)

        #Ali je problem končan?
        n = skrcitev(l, cas) # burnt vozlišča v cas
        e = skrcitev(m, cas) # defended vozlišča v cas
        skupaj = n + e

        koncan = 1
        # sosedi od pogorelih vozlišč so lahko pogoreli ali zaščiteni.
        Ne smejo biti prazna vozlišča
        for pogorelo_vozlisce in n:
            for sosed_od_pogorelo_vozlisce in G[pogorelo_vozlisce]:
                if sosed_od_pogorelo_vozlisce not in skupaj:
                    koncan = 0
        koncan

        if koncan == 1:
            break
        else:
            cas += 10

    return [k, l, m]
```

CLP sva želeli zastaviti tako, da v argumentu funkcije ni potrebno nastaviti časa, za katerega naj bi bil CLP končan. Zato sva nastavili nek začeten čas $cas = 10$, za katerga je bil rešen algoritem. Potem sva preverili, če je ta **rešitev končna**, torej, ali je vrednost spremenljivk v zadnji časovni enoti ustrezna. To pomeni, da za vsako vozlišče, ki je zgorelo, velja, da je vsako sosednje vozlišče le-tega tudi zgorelo, ali pa bilo rešeno. V nasprotnem primeru proces še ne bi bil končen, in čas *cas* se nastavi na večjo vrednost ter ponovimo algoritem.

Za točen čas, za katerega dobimo končno rešitev (proces se v naslednjih časih ne spreminja), sva napisali sledečo funkcijo:

```
def cas_potreben(G, B, gasilci):
''' iz p.solve() pridobi čas po katerem se nič več ne spremeni
-> dobimo potreben čas '''
 #cas = 10 #začetni cas
cas = 10
while True:
    t, burnt, defended = clp(G, B, gasilci)

    urej_burnt = sorted(burnt.items(), key=lambda tup: tup[0][1])
    #uredi glede na čas po vozliščih naraščajoče
    urej_defended = sorted(defended.items(), key=lambda tup: tup[0][1])

    vredn_burnt= []
    for i, v in urej_burnt:
        vredn_burnt.append(v)
    # pridobim ven vrednosti spremnljivk b v časih in vozliščih naraščajoče

    vredn_defended= []
    for i, v in urej_defended:
        vredn_defended.append(v)
    # pridobim ven vrednosti spremnljivk d v časih in vozliščih naraščajoče

    # from itertools import islice
    from itertools import accumulate
    dolzina = [len(G)] * (cas +1)
    #Vrednosti zgrupiram v paketke, v vsakem je toliko vrednosti,
    kolikor je vozlišč
    seznami_vrednosti_po_casih_burnt = [tuple(vredn_burnt[x - y: x])
    for x, y in zip(accumulate(dolzina), dolzina)]

    seznami_vrednosti_po_casih_defended = [tuple(vredn_defended[x - y: x])
    for x, y in zip(accumulate(dolzina), dolzina)]

    d = next(i for i in range(len(dolzina)) if all(len(set(l[i:i+2])) == 1
    for l in (seznami_vrednosti_po_casih_burnt, seznami_vrednosti_po_casih_defended

    if d < cas:
        break
    else:
        cas += 10
return d
```

Funkcija *cas_potreben* torej za vsak graf, podmnožico vozlišč B grafa ter določeno število gasilcev izračuna potreben čas za rešitev algoritma. Pomembna opomba tukaj je, da ta čas ni enak časovni zahtevnosti algoritma. Potreben čas predstavlja število časovnih enot v procesu širjenja požara in reševanju vozlišč.

2 Vizualizacija problema

2.1 Predstavitev funkcij za barvanje

Za lažjo predstavo poteka problema sva napisali funkcijo *barvanje_po_korakih*, ki z barvanjem vozlišč grafa prikazuje širjenje in zajezevanje požara. Koda funkcije je naslednja:

```
def barvanje_v_casu_t(G, B, gasilci, t):
    ''' pomožna funkcija barvanje_v_casu_t izriše graf in pobarva vozlišča v
    določenem času (t). Začetna vozlišča oz. izvor pošara pobarva v zeleno,
    pogorela v  rdečo, zaščitena pa v modro. '''

    b = skrcitev(clp(G, B, gasilci)[1], t) # burnt v času t BREZ ZAČETNIH B
    for el in B:
        b.remove(el)
    d = skrcitev(clp(G, B, gasilci)[2], t) # defended vozlišča v času t
    return G.show(partition = [b, B, d])

def barvanje_po_korakih(G, B, gasilci):
    ''' funkcija, ki za vsako časovno enoto nariše situacijo na grafu.
    Barve:
        - zelena: oglišča kjer se požar začne (B)
        - rdeča: pogorela
        - modra: zaščitena '''

    time = cas_potreben(G, B, gasilci)
    print("Število potrebnih časovnih korakov: " + str(time))
    for t in range(0, time + 1):
        print("Situacija v času " + str(t) + ":")
        barvanje_v_casu_t(G, B, gasilci, t)
```

Funkcija *barvanje_po_korakih* deluje s pomočjo funkcije *barvanje_v_casu_t*, ki predstavlja situacijo v času t. Funkcijo si za lažje razumevanje poglejmo na konkretnem primeru.

2.2 Primer na petersonovem grafu

Potek problema in barvanje predstavimo na pretersonovem grafu.
Funkcija *barvanje_po_korakih* zahteva tri argumente in sicer graf G, množico izvornih vozlišč B ter *gasilci*, ki predstavlja število gasilcev, ki bodo v vsaki časovni enoti gasili požar. Izberemo si naslednje:

```
G = graphs.PetersenGraph()
B = [1, 5]
gasilci = 2
```

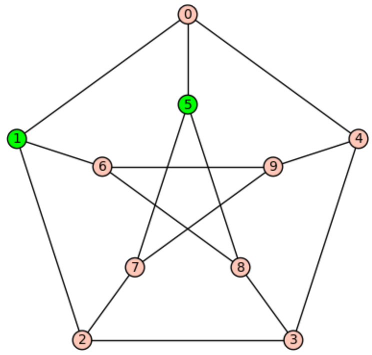
Imamo torej petersonov graf, na katerem bo požar izbruhnil v vozliščih 1 in 5 in v vsaki časovni enoti ga bosta gasila dva gasilca.
Opomba: množico vozlišč B bi lahko izbrali tudi naključno in sicer s funkcijo, ki bo predstavljana v nadaljevanju, in za aragument prejme graf ter število vozlišč, ki bi radi da so izvor požara.

Če zaženemo

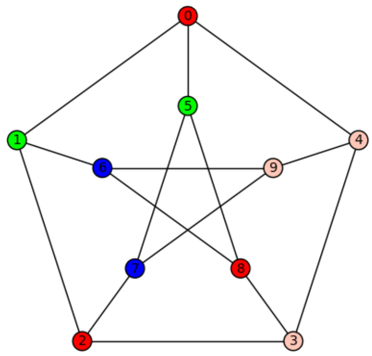
```
barvanje_po_korakih(G2, B2, gasilci2)
```

dobimo naslednje:

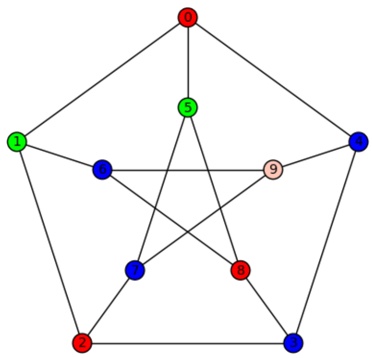
```
Število potrebnih časovnih korakov: 2
Situacija v času 0:
```




```
Situacija v času 1:
```




```
Situacija v času 2:
```



Funkcija *barvanje_po_korakih* nam na začetku izpiše število potrebnih časovnih korakov, ki ga izračuna z že predstavljeno funkcijo *cas_potreben*. Nato pa nam izriše situacijo v vsakem časovnem koraku:

- situacija <u>v času 0</u> so zgolj vozlišča izvora požara, v našem primeru vozlišči 1 in 5. Vozlišča izvora požara so tekom celotne grafične predstavitve pobarvane v zeleno.

- v naslednjem časovnem koraku, času 1 dva gasilca zaščitita požar in sicer v vozlišču 6 in 7. Zaščitena vozlišča so pobarvana z modro. Po tem ko gasilca zaščitita omenjeni vozlišči, se požar lahko naprej razširi le v vozlišče 2 in 0 (iz že zagorelega vozlišča 1) ter v vozlišče 8 (iz že zagorelega vozlišča 5). Pogorela vozlišča so pobarvana z rdečo.
- v naslednjem času, času 2 gasilca ponovno zaščitita vozlišči in sicer 3 ter 4. Z grafičnega prikaza je razvidno, da se požar ne more več razširiti naprej. Edino nepogorlo in nezaščiteno vozlišče je vozlišče 9.

3 Testiranje programa glede na število vozlišč grafa

3.1 Potek dela

Za izvedbo testiranja programa sva spisali funkcijo $seznam\_navorov\_st\_vozlisc\_cas$, ki kot argumente sprjme $seznam\_imen\_grafov$, $st\_gasilcev$ ter $stevilo\_vozlic\_v\_B$. Tekom funkcije se sprehodimo čez $seznam\_imen\_grafov$ (seznam poljubnih imen grafov) in sestavljamo nov seznam sestavljen iz naborov oblike

$$(\text{število vozlišč grafa}, \text{potreben čas reševanje problema na grafu})\,.$$

Koda funkcije:

```
import random
def nakljucno_izberi_vzolisca(graf, n):
    ''' funkcija naključno izbere n vozlišč iz grafa graf '''
    return random.sample(list(graf), n)

def seznam_naborov_st_vozlisc_cas(seznam_imen_grafov, st_gasilcev, stevilo_vozlisc_v_B):
    ''' funkcija, ki sprejme seznam v katerem so imena grafov, število gasilcev ter
        število vozlišč, ki jih želimo v začetni množici B.
        Vrne pa seznam naborov oblike (število vozlišč, potreben čas reševanje problema)'''

    # sprehodimo se po seznam_imen_grafov in sestavljamo nabor:
    seznam_naborov = []
    for graf in seznam_imen_grafov:
        B1 = nakljucno_izberi_vzolisca(graf, stevilo_vozlisc_v_B)
        B2 = nakljucno_izberi_vzolisca(graf, stevilo_vozlisc_v_B)
        potreben_cas1 = cas_potreben(graf, B1, st_gasilcev)
        potreben_cas2 = cas_potreben(graf, B2, st_gasilcev)
        seznam_naborov.append((len(graf), potreben_cas1))
        seznam_naborov.append((len(graf), potreben_cas2))

    return seznam_naborov
```

V funkciji sva si pomagali s funkcijo $nakljucno\_izberi\_vzolisca$, ki kot argument sprejme poljuben graf $graf$ ter poljubno pozitivno število n. Funkcija vrne n poljubnih vozlišč $grafa$. Funkcijo sva uporabili pri generiranju množice začetnih vozlišč B. Razlog pa je, da požar lahko izbruhne kjerkoli.

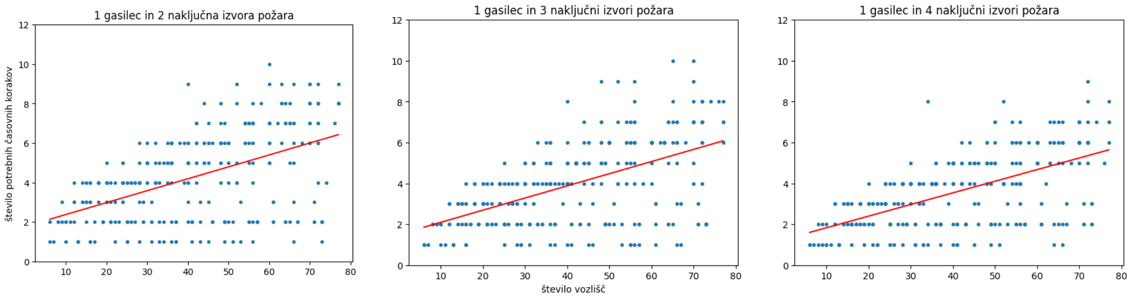
Funkcijo $seznam\_navorov\_st\_vozlisc\_cas$ sva izvedli na zelo velikem seznamu grafov. Grafi seznama so bili različnih velikosti in oblik. Obravnavo rezultatov sva razdelili glede na spremenljivki $st\_gasilcev$ ter $stevilo\_vozlic\_v\_B$.
Ločili sva primera:

- $st\_gasilcev = 1$ in $stevilo\_vozlic\_v\_B = \{2, 3, 4\}$
- $st\_gasilcev = 2$ in $stevilo\_vozlic\_v\_B = \{2, 3, 4\}$

Iz dobljenih podatkov sva nato sestavili *.csv* datoteko. Podatke sva vizualzirali s pomočjo *Python* knjižnice *pandas*. V grobem sva predstavlili odvisnost časa reševanje problema od števila vozlišč grafa. Število vozlišč obravnanih grafov je med 6 in 77. Grafične prikaze sva nazadnje nadgradili še z linarno regresijo.

3.2 1 gasilec & 2, 3 ali 4 izvori požara

Za parametre $st\_gasilcev = 1$ in $stevilo\_vozlic\_v\_B = \{2, 3, 4\}$ sva dobile naslednje grafične prikaze:



Dodatno sva izračunali še povprečno število potrebnih časovnih korakov:

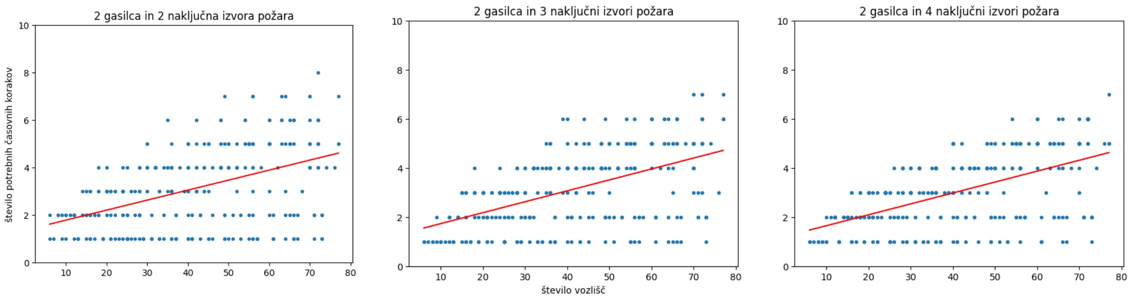
- 1 gasilec in 2 naključna izvora požara: 4.304,
- 1 gasilec in 3 naključni izvora požara: 3.993,
- 1 gasilec in 4 naključni izvora požara: 3.645.

Tako iz grafov kot tudi iz izračuna je razvidno, da se število potrebnih časovnih korakov z višanjem števila izvorov požara veča. To seveda ni presenetljivo, saj bo 1 gasilec zagotovo potreboval več časa za omejitev 4 izvorov požara, kot na primer 2 izvorov požara.

Komentar: opazimo, da so dobljena povprečja precej majhna. Slednje si lahko razlagamo z dejstvom, da je bilo v seznamu grafov več grafov z manjšim številom vozlišč (manj kot 40). Slednji grafi pa za zaključek problema potrebujejo manj časa kot grafi z večjim število vozlišč.

3.3 2 gasilca & 2, 3 ali 4 izvori požara

Za parametre $st\_gasilcev = 2$ in $stevilo\_vozlic\_v\_B = \{2, 3, 4\}$ sva dobile naslednje grafične prikaze:



Dodatno sva izračunali še povprečno število potrebnih časovnih korakov:

- 2 gasilca in 2 naključna izvora požara: 3.121,
- 2 gasilca in 3 naključni izvora požara: 3.151,
- 2 gasilca in 4 naključni izvora požara: 3.164.

V tem primeru pridemo do podobnega sklepa kot v prejšnjem (1 gasilec omejuje požar): število potrebnih časovnih korakov se z višanjem števila izvorov požara veča.

3.4 Primerjava primerov z 1 gasilcem in 2 gasilci

Primerjamo grafe in razultate, ki smo jih dobili v prejšnjih dveh primerih. Z grafa in tudi iz povprečji je opazno, da je število potrebnih časovnih enot manjše v primeru, ko imamo 2 gasilca, saj večje število gasilcev prej omeji požar.