

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

Projekt pri predmetu Finančni praktikum

The firefighter problem

Karolina Šavli in Klara Travnik

Ljubljana, januar 2023

Kazalo

1	Opis in formulacija problema	3
1.1	Predstavitev problema	3
1.2	Program	3
2	Vizualizacija problema	6
2.1	Predstavitev funkcij za barvanje	6
2.2	Primer na Petersonovem grafu	6
3	Časovna zahtevnost na konkretnih primerih	9
4	Testiranje programa glede na število vozlišč grafa	11
4.1	Potek dela	11
4.2	1 gasilec & 2, 3 ali 4 izvori požara	12
4.3	2 gasilca & 2, 3 ali 4 izvori požara	12
4.4	Primerjava primerov z 1 gasilcem in 2 gasilci	13
5	Zaključek	14

1 Opis in formulacija problema

1.1 Predstavitev 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 in
- 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 lahko razširi le na sosede že pogorelih vozlišč, ki niso še zaščitena. V naslednji časovni enoti gasilci ponovno gasijo požar in opisan proces se ponavlja dokler požar ni zajezjen.

Predpostavke problema:

- če je bilo vozlišče v nekem času t rešeno ali je pogorelo, obvelja rešeno oz. pogorelo tudi v vseh prihodnjih časih,
- vozlišče v ne more biti hkrati označeno kot pogorelo in rešeno,
- v vsakem času t je na voljo fiksno število gasilcev; vsak lahko reši eno vozlišče, torej je v vsakem času največ D na novo rešenih.

1.2 Program

Opisani problem sva v programu *CoCalc*, v programskem jeziku *SageMath* 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 [št. 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

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

        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 v katerem naj bi bil CLP končan. Slednje sva dosegli z *while True* zanko in nastavitvijo začetnega časa, *cas = 10*. V tem času je problem lahko končan ali ne. Če problem v tem času ni končan, torej rešitev ni končna, sva čas povečali za 10 ter algoritem ponovili. Da je rešitev končna pomeni, da za vsako vozlišče, ki je zgorelo, velja, da je vsako sosednje vozlišče le-tega tudi zgorelo, ali pa bilo rešeno.

Za število potrebnih časovnih korakov, da dobimo končno rešitev (torej se proces/situacija v naslednjih časih ne spreminja), sva napisali naslednjo funkcijo:

```

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

        # uredi glede na čas naraščajoče:
        urej_burnt = sorted(burnt.items(), key=lambda tup: tup[0][1])
        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 spremenljivk b v časih in vozliščih naraščajoče

        vredn_defended = []
        for i, v in urej_defended:
            vredn_defended.append(v)
        # pridobim ven vrednosti spremenljivk 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
seznam_i_vrednosti_po_casih_burnt = [tuple(vredn_burnt[x - y: x]) for x, y in zip(
    accumulate(dolzina), dolzina)]

seznam_i_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 (seznam_i_vrednosti_po_casih_burnt, seznam_i_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.

Opomba: potreben čas ni enak časovni zahtevnosti algoritma. Potreben čas predstavlja število časovnih enot v procesu širjenja požara in reševanju vozlišč. Časovno zahtevnost pa bova obravnavali v nadaljevanju.

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 požara pobarva v zeleno,
        pogorela v rdečo, zaščitena pa v modro. '''
    b = skrcitev(clp(G, B, gasilci)[1], t) # burnt vozlišča v času t BREZ ZAČETNIH VOZLIŠČ 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 pogledjmo na konkretnem primeru.

2.2 Primer na Petersonovem grafu

Potek problema in barvanje predstavimo na Petersonovem 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 argument prejme graf ter število vozlišč, ki bi radi da so izvor požara.

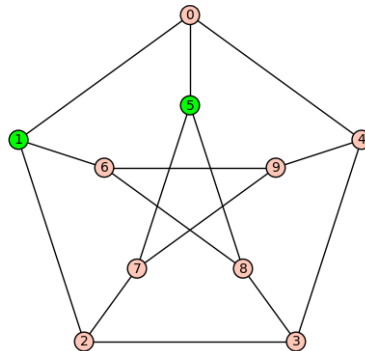
Če zaženemo

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

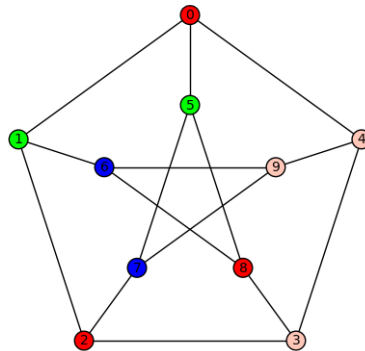
dobimo naslednje:

Število potrebnih časovnih korakov: 2

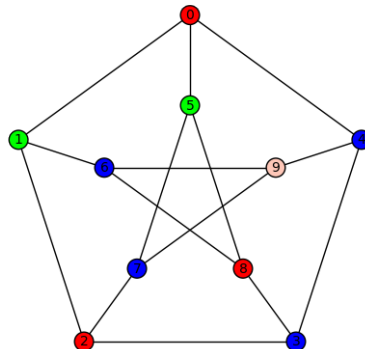
Situacija v času 0:



Situacija v času 1:



Situacija v času 2:



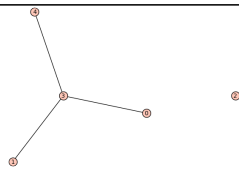
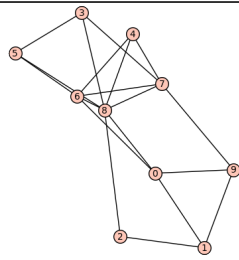
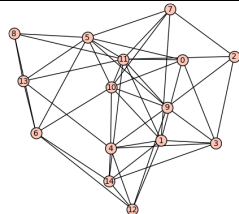
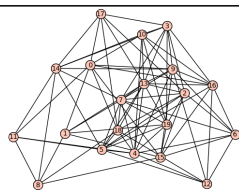
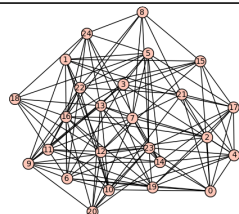
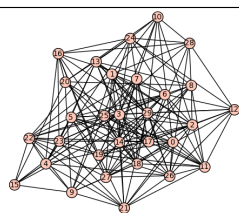
Funkcija *barvanje_po_korakih* nam na začetku izpiše število potrebnih časovnih korakov, ki ga izračuna z že predstavljeno funkcijo *cas_potreben*. Nato pa nam izriše situacijo v vsakem časovnem koraku:

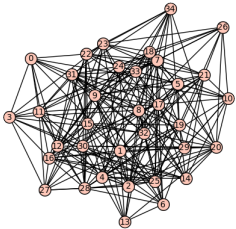
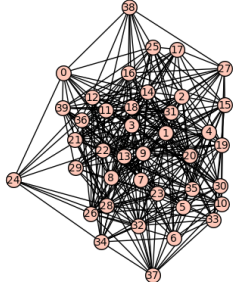
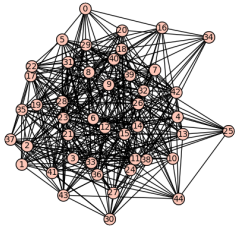
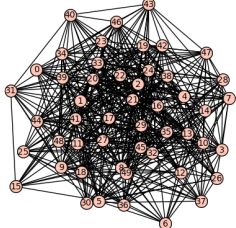
- situacija v času 0 so zgolj vozlišča izvora požara, v našem primeru vozlišči 1 in 5. **Vozlišča izvora** požara so tekom celotne grafične predstavitve pobarvane v **zeleno**.
- v naslednjem časovnem koraku, času 1 dva gasilca zaščitita požar in sicer v vozlišču 6 in 7. **Zaščitena vozlišča** so pobarvana 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 pa je vozlišče 9.

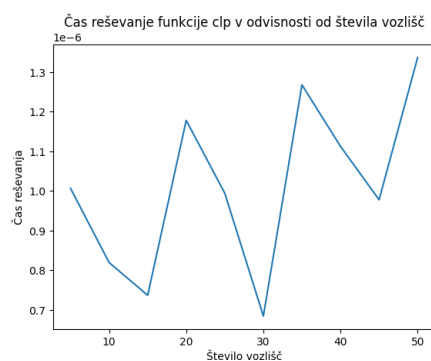
3 Časovna zahtevnost na konkretnih primerih

Kot zanimivost sva s pomočjo *Python* knjižnice *timeit* generirali funkcijo, ki nam vrne čas reševanja oziroma časovno zahtevnost funkcije *clp*, opisane v prvem poglavju. Kot že znano, funkcija *clp* sprejme tri argumente: graf, množico izvorov požara in število gasilcev. Funkciji sva podali poljuben graf, naključno vozlišče (eno samo) ter določili sva, da bo požar gasil samo en gasilec. Slednje sva naredili na desetih grafih z različnim številom vozlišč in dobili sva naslednje rezultate:

število vozlišč	slika obravnavanega grafa	časovna zahtevnost
5		1.0068004485219717e-06
10		8.19600245449692e-07
15		7.370006642304361e-07
20		1.1784009984694422e-06
25		9.944007615558803e-07
30		6.845992174930871e-07

število vozlišč	slika obravnavanega grafa	časovna zahtevnost
35		1.2680000509135426e-06
40		1.1128009646199644e-06
45		9.780007530935108e-07
50		1.3369994121603667e-06

Dobljene rezultate predstavimo še na grafu:



Opazimo, da čas reševanje grafa za naše primere ni v nobeni relaciji s številom vozlišč. Predvidevamo pa lahko, da je čas reševanje v splošnem večji za grafe z večjim številom vozlišč. To bi lahko s simulacijo tudi pokazale in sicer na podoben način kot sva izvedli testiranje v naslednjem poglavju.

4 Testiranje programa glede na število vozlišč grafa

4.1 Potek dela

Za izvedbo testiranja programa sva spisali funkcijo *seznam_naborov_st_vozlisc_cas*, ki kot argumente sprejme *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

(število vozlišč grafa, 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 *graf*-a. Namen funkcije je generirane naključne množice začetnih vozlišč *B*, saj požar lahko izbruhne kjerkoli.

Funkcijo *seznam_naborov_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*.

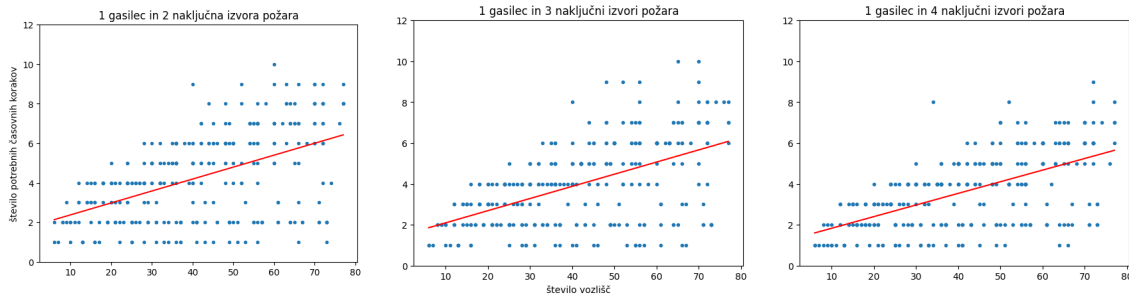
Ločili sva primeri:

- *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 in podatke vizualizirali s pomočjo *Python* knjižnice *pandas*. V grobem sva predstavljali odvisnost časa reševanje problema od števila vozlišč grafa. Število vozlišč obravnavanih grafov je med 6 in 77. Grafične prikaze sva nazadnje nadgradili še z linearno regresijo.

4.2 1 gasilec & 2, 3 ali 4 izvori požara

Za parametre $st_gasilcev = 1$ in $stevilo_vozlic_v_B = \{2, 3, 4\}$ sva dobili naslednje grafične prikaze:



Dodatno sva izračunali še povprečno število potrebnih časovnih korakov:

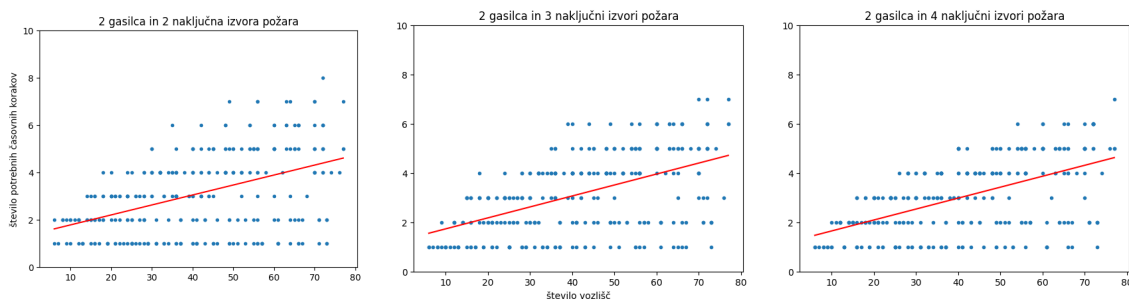
- 1 gasilec in 2 naključna izvora požara: 4.304,
- 1 gasilec in 3 naključni izvori požara: 3.993,
- 1 gasilec in 4 naključni izvori požara: 3.645.

Tako iz grafov kot tudi iz izračuna je razvidno, da se število potrebnih časovnih korakov z višanjem števila izvorov požara večja. 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 številom vozlišč.

4.3 2 gasilca & 2, 3 ali 4 izvori požara

Za parametre $st_gasilcev = 2$ in $stevilo_vozlic_v_B = \{2, 3, 4\}$ sva dobili 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 večja.

4.4 Primerjava primerov z 1 gasilcem in 2 gasilci

Primerjamo grafe in rezultate, 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.

5 Zaključek

V praksi najdemo več različic obravnavanega optimizacijskega problema gasilca. Na primer:

- **širjenje bolezni** v manjši skupnosti, kjer z vozlišči grafa predstavimo ljudi in s povezavami stike z drugimi. Če je vozlišče v grafu okuženo, ostane nalezljivo za A časovnih enot in lahko okuži sosede v tem časovnem okviru. Gasilce pa v tem primeru nadomestijo ukrepi proti širjenju (e.g. razkuževanje, nošenje mask, ...).
- V igri **policistov in roparja** rešujemo problem, ali lahko D policistov ujame roparja, ki se premika po povezavah grafa.
- Podobnost pa lahko najdemo tudi pri vzpostavljanju **komunikacijske mreže med člani uporniškega gibanja** tako, da se minimizira število izdaj članov.

Delo najinega projekta zajema reševanje, obravnavo in uporabo celoštevilskega linearnega programa. Pri slednjem sva si pomagali s članki “García-Martínez, Blum, Rodríguez in Lozano”[2] ter “Fomin, Heggenes in van Leeuwen”[1].

Problema bi se pa seveda lahko lotili še na kakšen drug način. V vsakem časovnem koraku iteracije bi lahko na primer kot rešeno označili vozlišče z največjo stopnjo. Tako bi preprečili (v primeru da zgori), da se ogenj razširi na sosednja vozlišča le-tega. V nasprotnem primeru bi bila zajezeitev požara težja. S takim načinom bi potem morali pri testiranju pogledati razmerje med najvišjo stopnjo grafa in časom. Vendar pa na tak način najverjetneje ne bi dobili najboljše rešitve, kot pri linearnem programu.

Glavni sklepi:

Literatura

- [1] Fedor V. Fomin, Pinar Heggernes, and Erik Jan van Leeuwen. The firefighter problem on graph classes. *Theoretical Computer Science*, 613:38–50, 2016.
- [2] Carlos García-Martinez, Christian Blum, F. J. Rodriguez, and Manuel Lozano. The firefighter problem: Empirical results on random graphs. *Computers & Operations Research*, 60:55–66, 2015.