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

Karolina Šavli, Klara Travnik

The firefighter problem

Projekt pri predmetu Finančni praktikum

Ljubljana, januar 2023

Kazalo

1 Naslov

Besedilo ...

1.1 Podnaslov

- 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

2 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.