Univerza v Ljubljani
Fakulteta za matematiko in fiziko

Anja Kokalj in Lina Ivanova

Najdaljši Hamiltonov cikel brez presečišč

Seminarska naloga pri predmetu Finančni praktikum

Mentor: prof. dr. Sergio Cabello, asist. dr. Janoš Vidali

Ljubljana, 2021

Kazalo

1 Uvod

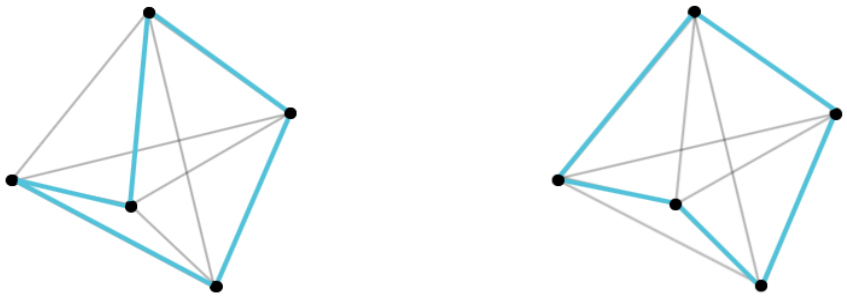
Osrednji problem projektne naloge je iskanje Hamiltonovega cikla brez presečišč, ki se razteza nad množico točk P v evklidski ravnini. Predstavile bova dva optimizacijska problema: Iskanje najdaljšega Hamiltonovega cikla in iskanje cikla, katerega najkrajša uporabljena povezava bo izmed vseh najkrajših uporabljenih povezav najdaljša.
Skozi seminarsko nalogo bomo opazili, da reševanje le-tega ni enostavno. Za implementacijo sva si pomagali s programskim jezikom **SageMath**, kateri je že imel vgrajeno potrebno funkcijo `MixedIntegerLinearProgram()` za reševanje problemov s celoštevilskim linearnim programom.
Proti koncu bova s pomočjo generiranega programa, opazovali kako povečanje števila točk v množici P vpliva na optimalno vrednost navedenih problemov in na čas samega delovanja. Prav tako, bova grafično predstavile razliko dolžin najkrajše uporabljene povezave in skupno dolžino vseh uporabljenih povezav v obeh dobljenih ciklih.

1.1 Matematični okvir

Za začetek sva definirali neusmerjeni graf $G = (P, E)$, kjer je

- $P = \{1, 2, ..., n\}$ množica točk oziroma vozlišča grafa G v splošnem položaju v ravnini,
- E množica povezav grafa G, pri čemer je vsaka povezava par točk.

Hamiltonov cikel v grafu G je cikel, ki obhodi vse točke iz dane množice P natanko enkrat ter sta začetna in končna točka enaki.



Slika 1: Primer dveh Hamiltonovih ciklov na grafu G

2 Celoštevilski linearni program

2.1 Poimenovanje spremenljivk

Kot že v uvodu povedano, sva s P označile množico, ki vsebuje n točk, $p_i \in \mathbb{R}^2$, $i = 1, 2, ..., n$ za $n \in \mathbb{N}$. Daljica uv predstavlja neusmerjeno povezavo med točkama $p_u \in P$ in $p_v \in P$. Razdaljo le-te sva označili z

$$d_{uv} =\mid p_u - p_v \mid,$$

ki sva jo potrebovale za izračun maksimalne vrednosti vsote uporabljenih povezav uv in maksimalne vrednosti najkrajše povezave uv. Ker sva problem reševali s celoštevilskim linearnim programiranjem sva potrebovali tudi binarno spremenljivko

$$x_{uv} = \begin{cases} 1, \text{ če uporabimo povezavo med točkama } p_u \text{ in } p_v \\ 0, \text{ sicer.} \end{cases}$$

Definiriali sva spremenljivki $t \in \mathbb{R}$ in $d_{max} = max(d_{u,v})$, ki sva jo uporabile pri maksimizaciji najkrajšega roba.

2.2 Pogoji

Prvi pogoj je, da imamo povezave, ki se med seboj ne sekajo:

$$x_e + x_f \leq 1 \text{ za vsak e,f } \in \text{I} \subseteq \text{E(G)} \times \text{E(G)}$$

Temu sledi pogoj, da lahko iz vsake točke vodita natanko dve povezavi:

$$\sum_{e=uv \in E(G)} x_e = 2, \text{ za vsak v = 1,2,...,n.}$$

Prav tako sva dodali pogoj, da za določene particije množic vozlišč na dve neprazni množici obstajata vsaj dve povezavi med tema množicama.

$$\sum_{e \in E(G), e \in S \times \overline{S}} x_e \geq 2, \text{ za vsak S} \subseteq \text{P(G).}$$

Pri drugem problemu sva poleg prej omenjenih pogojev potrebovali tudi pogoj, da bo spremenljivka t vedno manjša ali enaka najkrajši povezavi, ki jo bomo uporabili:

$$t \leq r_{u,v} + (1 - x_{u,v} \cdot r_{max}),$$

za $t \in \mathbb{R}$ in $u, v \in 1, 2, .., n$.

Cilj problema maksimizacije vsote dolžin povezav cikla, sva zapisali kot:

$$max \sum_{u=1}^{n} \sum_{v=1}^{n} (x_{u,v} \cdot d_{u,v}), \text{ za vsak u = 1,2,...,n.}$$

Pri problemu maksimizacije dolžine najkrajše uporabljene povezave pa kot:

$$max(t),$$

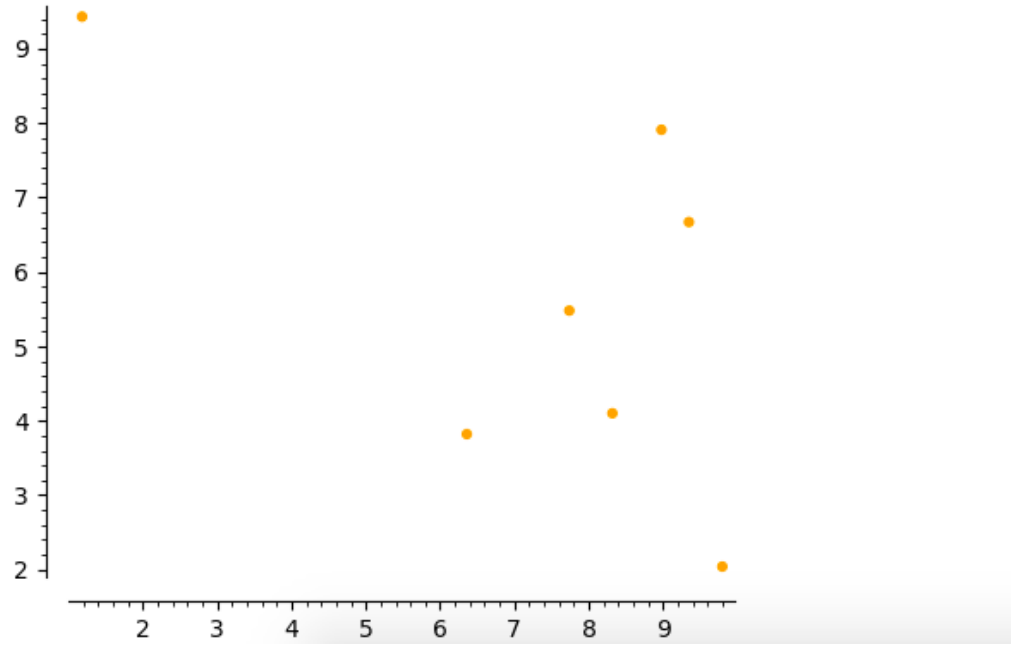
pri čemer je $t \in \mathbb{R}$.

3 Opis programa

Pisanja zgoraj opisanega celoštevilskega linearnega programa sva se lotile v programu **SageMath**. Za začetek sva definirali n naključno razporejenih točk v evklidski ravnini `[0,10)x[0,10)`. To sva storile s funkcijo `generate_random_point` in jih shranile v seznam `list_points`.

```
def generate_random_points(n, max_x = 10, max_y = 10):
    return [(max_x * random(), max_y * random()) for _ in range(n)]
```

```
[[1.18, 9.43], [9.8, 2.04], [8.98, 7.91], [7.74, 5.48], [9.35, 6.67],
[8.32, 4.1], [6.36, 3.82]]
```



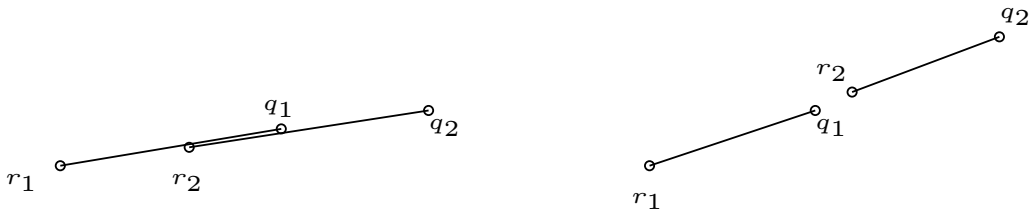
Slika 2: Primer 7 naključno izbranih točk v evklidski ravnini

Za vsak par poljubnih točk sva poiskali razdaljo med njima s pomočjo funkcije `distances`:

```
def distance(p, q):
    return math.sqrt((p[0] - q[0])^2 + (p[1] - q[1])^2)
```

Upoštevati sva tudi morali pogoj ne sekanja povezav. To sva razrešile s funkcijo `orientation`. Funkcija vzame tri točke in pove kakšna je orientacija med

izbranimi točkami. Torej je lahko enaka 0, če so točke med seboj kolinearne, 1, če so v smeri urinega kazalca in -1, če so v nasprotni smeri urinega kazalca. Kolinearnost preverimo tako, da pogledamo pravokotne projekcije točk na x in y os. Če se intervala med točkami sekata potem se bosta sekali tudi gledani povezavi.

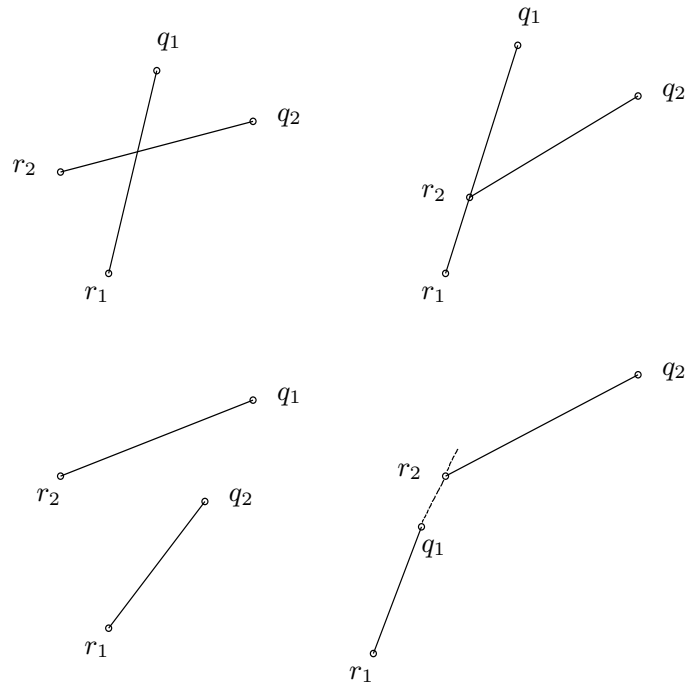


Slika 3: Primer sekanja dveh povezav, kjer so si točke med seboj kolinearne. Pri prvi sliki se daljici sekata, pri drugi se ne.

Pri nekolinearnosti točk pa pogledamo njihovo orientacijo urejene trojice. Dve povezavi se bosta sekali, če izpolnjujeta naslednji pogoji:

- Trojici (r_1, q_1, r_2) in (r_1, q_1, q_2) imata različni orientaciji,
- Trojici (r_2, q_2, r_1) in (r_2, q_2, q_1) imata različni orientaciji.

Tukaj se prva gledana povezava razteza med točkama r_1 in q_1 in druga med r_2 in q_2. Upoštevali sva tudi pogoj, da se povezavi ne sekata, če se sekata v isti točki.



Slika 4: Zgornji sliki prikazujeta različno orientacijo trojic (r_1, q_1, r_2) in (r_1, q_1, q_2), ter (r_2, q_2, r_1) in (r_2, q_2, q_1), spodnji sliki prikazujeta enako orientacijo trojic (r_2, q_2, r_1), (r_2, q_2, q_1) in različno (r_1, q_1, r_2), (r_1, q_1, q_2).

Povezave, ki se med seboj sekajo sva shranile v seznam `intersection`.

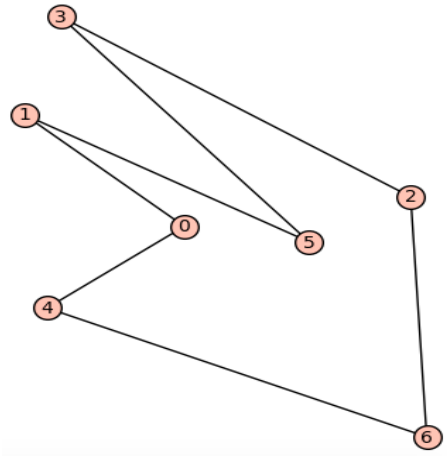
Z zgoraj definiranimi funkcijami sva pridobile vse potrebne zahteve, ki jih vpišemo v celoštevilski linearni program:

- **Pogoj ne sekanja:** Če se dve povezavi sekata, smemo uporabiti največ eno.
- Iz vsakega vozlišča vodita natanko dve povezavi.
- Za določene particije množic vozlišč na dve neprazni množici obstajata vsaj dve povezavi med tema množicama.

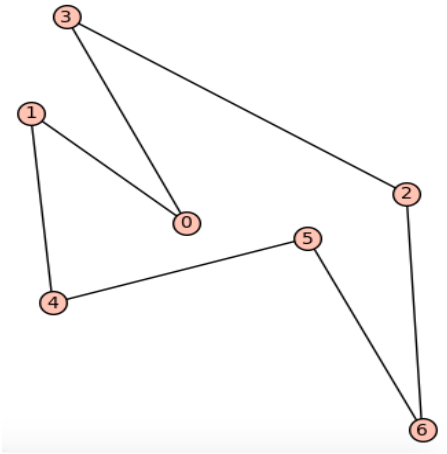
Ko računalnik reši CLP, nam izpiše vsoto najdaljših med seboj ne sekajočih povezav.

Pri reševanju drugega problema sva navedenim zahtevam dodale še pogoj iskanja najkrajše povezave v ciklu. Tu nama linearni program vrne dolžino najkrajše povezave v iskanem ciklu.

Spodaj je primer rešitev obeh problemov na naključno izbranih 7 točkah.



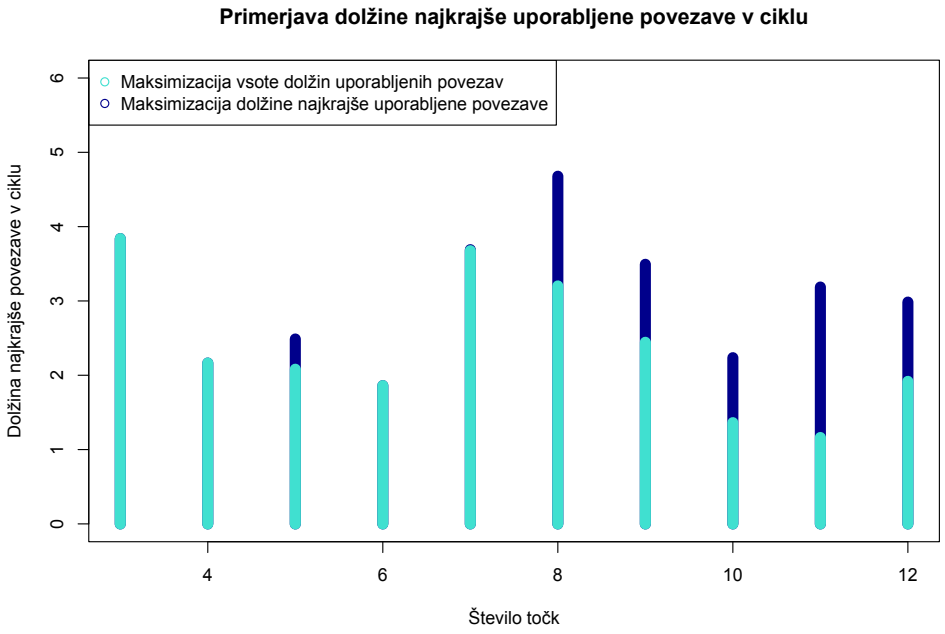
Slika 5: Primer rešitve CLP, Maksimizacija vsote dolžin uporabljenih povezav v ciklu, na 7 točkah



Slika 6: Primer rešitve CLP, Maksimizacija najkrajše uporabljene povezave v ciklu, na 7 točkah

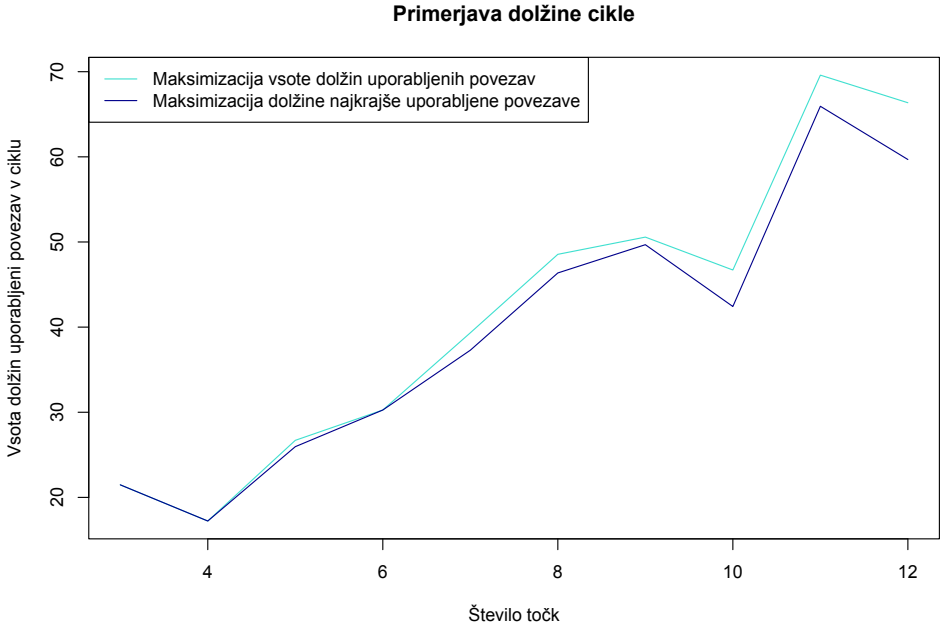
Pri prvi sliki vidimo, da je najkrajša povezava med vozliščema 0 in 4, pri drugi pa se nahaja med vozliščema 0 in 1, katere razdalja je večja od prve. Za primer sva vzeli graf, na katerem nama je program izpisal dva različna cikla za lažjo predstavitev namena najine seminarske naloge. Vendar pri večkratnem opravljanju poizkusov in spreminjanjem število točk pa je prišlo do enakih podgrafov, še posebej za grafe, ki so se raztezali nad manj kot 6

točkami. Poizkuse sva po večini delali na grafih z največ 12 točkami, kajti pri večanjem števila točk je računalnik potreboval predolgo časa za izpis rešitve linearnega programa. Meniva, da če bi bili današnji računalniki sposobnejši bi lahko prišle do ugotovitve, da z večanjem števila n dobimo več možnosti uporabe različnih povezav in s tem različne rešitve za oba problema.
V grafu primerjava dolžin najkrajše uporabljene povezave v ciklu sva najino domnevo le poizkušale potrditi že za izbrane grafe ki se raztezajo nad 8 točkam.



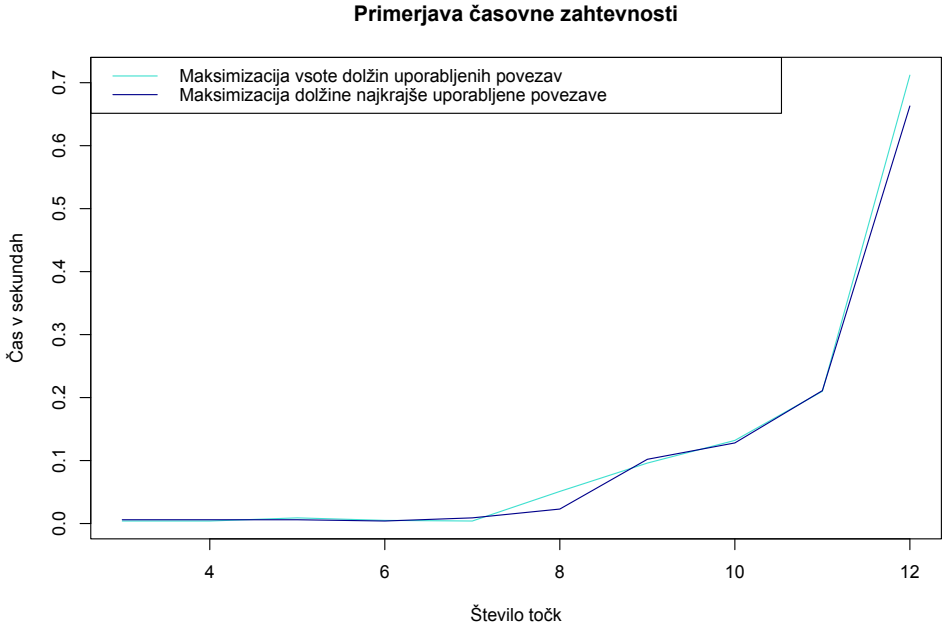
Slika 7: Primerjava dolžine najkrajše uporabljene povezave v ciklu

Zanimalo naju je tudi, kako se spreminja vsota vseh povezav, ko povečujeva število točk v evklidski ravnini. Kot pričakovano se optimalna vrednost povečuje z povečanjem števila n. Rezultate sva predstavile v spodnjem grafu, ki sva ga oblikovale s pomočjo programa **R**.



Slika 8: Primerjava vsote dolžin uporabljenih povezav v ciklu

Navsezadnje sva preverile še čas porabljen pri maksimiziranju vsot in najkrajše povezave. Ker sva imele problem s programiranjem funkcije, ki nama bi izračunala čas delovanje celoštevilskega linearnega programa, sva pisanje programa prestavili na **CoCalc**, ki nama je skupaj z izvajanjem programa računal še čas izvajanja. Tudi tukaj sva rezultate predstavile grafično. Vidimo, da se z večanjem številom točk drastično poveča čas delovanja.



Slika 9: Primerjava časovne zahtevnosti

4 Zaključek

Skozi seminarsko nalogo sva ugotovili, da je bil problem precej težak. Veliko časa sva porabile s programiranjem pogoja o ne sekanju daljic. Prav tako sva opazile, da je program časovno zelo zahteven za veliko število točk in, da se čas izvajanja programa eksponentno povečuje. Enako se z n povečuje tudi vsota dolžin uporabljenih povezav. O problemu maksimiziranja najkrajše povezave na majhnem številu točk pa ne moramo kaj dosti povedati, lahko pa sklepamo, da bo cikel z maksimalno vsoto dolžine stranic imel najkrajšo povezavo krajšo oziroma v velikih primerih enako dolgo kot cikel pri drugem problemu.

5 Priloge

```
# Linearni program 1
# Maksimizacija vsote dolžin uporabljenih povezav.

p = MixedIntegerLinearProgram()
x = p.new_variable(binary=True)

# ciljna funkcija: max skupna dolžina uporabljenih povezav
p.set_objective(sum(x[e] * d for e, d in distances.items()))

# POGOJI

# Če se povezavi sekata, smemo uporabiti največ eno
for e, f in intersection:
    p.add_constraint(x[e] + x[f] <= 1)

# Vsota vseh povezav v vozlišču je enaka 2
for v in g.vertices():
    edges = g.edges_incident(v, labels = False)
    p.add_constraint(sum(x[Set(e)] for e in edges) == 2)

p.solve()

# Za določene particije množic vozlišč na dve neprazni množici obstajata
# vsaj dve povezavi med tema množicama.
while True:

    x_sol = p.get_values(x)
    edges = [tuple(e) for e, i in x_sol.items() if i == 1]

    g_tmp = g.subgraph(vertices = g.vertices(), edges = edges)
    cc = g_tmp.connected_components()

    if len(cc) == 1:
        break

    boundary = g.edge_boundary(cc[0], labels = False)

    p.add_constraint(sum([x[e] for e in boundary]) >= 2)
    p.solve()

value1 = p.solve()
solution1 = p.get_values(x)
g_tmp.plot()
```

Slika 10: CLP, Maksimizacija vsote dolžin uporabljenih povezav v ciklu

```
# Linearni program 2
# Maksimizacija dolžine najkrajše uporabljene povezave

p = MixedIntegerLinearProgram()
x = p.new_variable(binary=True)

# ciljna funkcija: max skupna dolžina uporabljenih povezav
p.set_objective(sum(x[e] * d for e, d in distances.items()))

# POGOJI

# Če se povezavi sekata, smemo uporabiti največ eno
for e, f in intersection:
    p.add_constraint(x[e] + x[f] <= 1)

# Vsota vseh povezav v vozlišču je enaka 2
for v in g.vertices():
    edges = g.edges_incident(v, labels = False)
    p.add_constraint(sum(x[Set(e)] for e in edges) == 2)

# Maksimiziranje realne spremenljivke t, ki naj bo navzgor omejena
# z dolžinami uporabljenih povezav.
dmax = max(distances.values())
for e, d in distances.items():
    p.add_constraint(p['t'] <= d + (1 - x[e]) * dmax)
p.set_objective(p['t'])

p.solve()

# Za določene particije množic vozlišč na dve neprazni množici obstajata
# vsaj dve povezavi med tema množicama.
while True:

    x_sol = p.get_values(x)
    edges = [tuple(e) for e, i in x_sol.items() if i == 1]
    g_tmp = g.subgraph(vertices = g.vertices(), edges = edges)
    cc = g_tmp.connected_components()

    if len(cc) == 1:
        break

    boundary = g.edge_boundary(cc[0], labels = False)

    p.add_constraint(sum([x[e] for e in boundary]) >= 2)

    p.solve()

value2 = p.solve()
solution2 = p.get_values(x)
```

Slika 11: CLP, Maksimizacija najkrajše uporabljene povezave v ciklu