Univerza v Ljubljani
Fakulteta za matematiko in fiziko

Žan Jernejčič in Ines Šilc

Reševanje problema trgovskega potnika s k-optimalnim in Lin-Kernighanovim algoritmom

Ljubljana, 2020

1 Definiranje problema

V projektni nalogi bova reševala Problem trgovskega potnika s pomočjo k-optimalnega in Lin-Kernighanovega algoritma.

Problem trgovskega potnika oziroma Travelling salesman problem (krajše TSP) je problem, kjer imamo podanih n mest in razdalje med vsemi (za vsak par mest imamo torej podano, koliko sta si oddaljeni). Zanima nas, ali lahko obiščemo vsako mesto in se na koncu vrnemo v prvotno mesto. Če označimo $d_{i,j}$ kot razdaljo mad i-tim in j-tim mestom, iščemo torej:

$$\min_{\pi \in S_n} \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)}$$

kjer je S_n množica vseh permutacij danih n mest.

Naivna rešitev je očitna, pogledamo $(n-1)!$ kombinacij, torej iz vsakega mesta v vsako drugo mesto, si zapišemo vse kombinacije in kakšno razdaljo smo prepotovali, ter izberemo tisto možnost, kjer je bila razdalja najkrajša.

Pred začetkom se lahko vprašamo kakšen mora biti graf, da lahko na njem izvajamo sledeča algoritma. Graf **ne sme** imeti **negativnih ciklov**, saj je najcenejši cikel potem očiten, in ustvarimo neskončno zanko, saj bo imela najboljša rešitev ceno -∞. Ker imamo opravka s cikli, lahko cikel začnemo v kateremkoli vozlišču, zato lahko vedno gledamo možne poti od začetka.

Za projekt sva uporabila programski jezik `python`. Uporabljala sva naslednje pakete:

- `networkx`: za definiranje in generiranje grafov
-
-
-
-
- `random`: za generiranje naključnih števil in seznamov
- `matplotlib`: za izrisovanje grafov
- `time`: za mertive časovne zahtevnosti
- `itertools`: za generiranje prvotne permutacije

Za preverjanje veljavnosti poti v grafu za problem potujočega trgovca, morava vedeti:

- Vsaka pot trgovca bo morala imeti vsa vozlišča primarnega grafa
- Vsaka pot trgovca bo morala imeti točno toliko povezav kot vozlišč
- Dolžina poti bo morala biti enaka številu vozlišč -1

2 k-optimalni algoritmi

K-optimalni algoritem, je lokalni algoritem, kjer iščemo rešitev Problema trgovskega potnika in sorodnih problemov. Algoritem deluje tako, da zmanjša dolžino trenutnega potovanja, dokler ne dosežemo poti, katere dolžine ne moremo izboljšati.

Poznamo več različic k-opt algoritma. V tej projektni nalogi se bova osredotočila na dve glavni različici k-opt algoritma, to sta 2-Opt in 3-Opt algoritem. 2-Opt algoritem deluje tako, da odstrani dve vozlišči grafa (dve mesti), tako razdeli pot na dva dela, poišče najboljšo rešitev podproblema, in nato poveže nazaj mesti na najboljši možen način. 3-Opt algoritem odstrani tri povezave ali vozlišča, tako dobimo 3 podomrežja mest. V naslednjem koraku analiziramo najboljšo pot med temi tremi podomrežji. To ponavljamo za druge tri povezave, dokler nismo poskusili vseh možnih poti v danem omrežju.

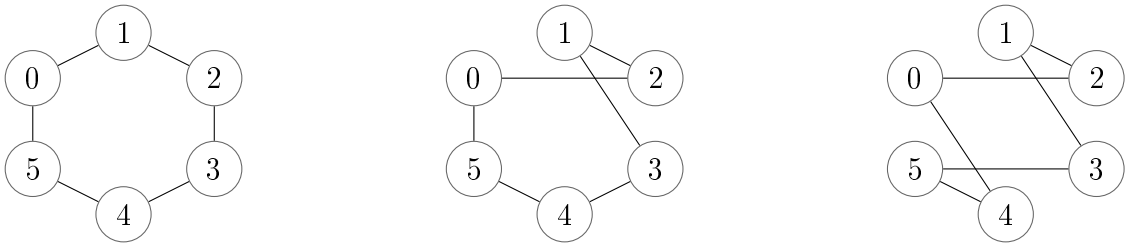
2.1 2-opt algoritem

Za začetek 2-opt postopka potrebujemo poln graf z utežmi na vsaki povezavi in danim začetnim ciklom po tem grafu. Postopoma odstranjujemo po dve vozlišči in ju povežemo drugače, nato preverimo, ali je novonastali cikel cenejši. Če je novi cikel cenejši, potem ga vzamemo za boljšo rešitev. Ta postopek nadaljujemo, dokler ne pridemo do konca poti in se moramo vrniti na začetek.

Za algoritem potrebujemo še dodatno funkcijo, ki nam izračuna ceno poti, ta prejme vhodne podatke graf in pot in sešteje uteži poti v grafu. Dejanski postopek 2-opt algoritma izgleda takole:

```
def dva_opt(graf, pot):

    najboljsa_pot = pot

    izboljsanje = True
    while izboljsanje:
        izboljsanje = False
        for i in range(1, len(pot) - 2):

            for j in range(i + 1, len(pot)):

                if j - i == 1: continue
                nova_pot = pot
                nova_pot[i:j] = pot[j - 1:i - 1:-1]

                if cena_poti(graf, nova_pot) < cena_poti(graf, najboljsa_pot):
                    najboljsa_pot = nova_pot
                    izboljsanje = True
        pot = najboljsa_pot

    return (pot, cena_poti(graf, pot))
```

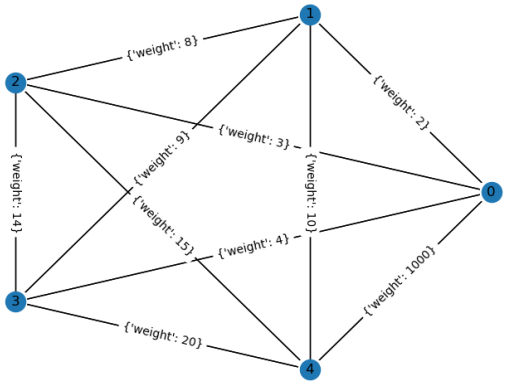
Algoritem gre postopoma po poti in zamenja zaporedje vozlišč, če je le tako bolj optimalno. Dva koraka 2-opt algoritma bi izgledala takole:



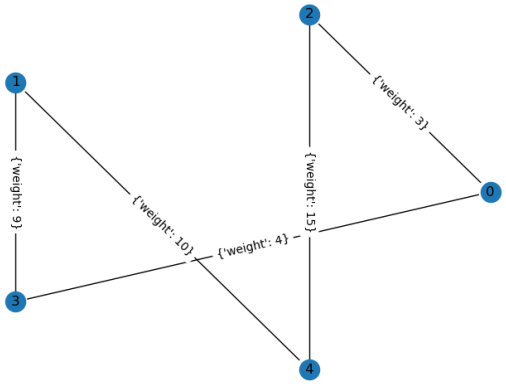
Za primer delovanja si oglejmo preprosto matriko, ki predstavlja graf. Na (i, j)-tem mestu matrike je cena med mestoma i in j. Dana je matrika (1) in njen graf je narisan na sliki 1

$$\begin{bmatrix} 0 & 2 & 3 & 4 & 1000 \\ 2 & 0 & 8 & 9 & 10 \\ 3 & 8 & 0 & 14 & 15 \\ 4 & 9 & 14 & 0 & 20 \\ 1000 & 10 & 15 & 20 & 0 \end{bmatrix} \tag{1}$$

Rešitev, ki nam jo poda 2-opt algoritem je na sliki 2:

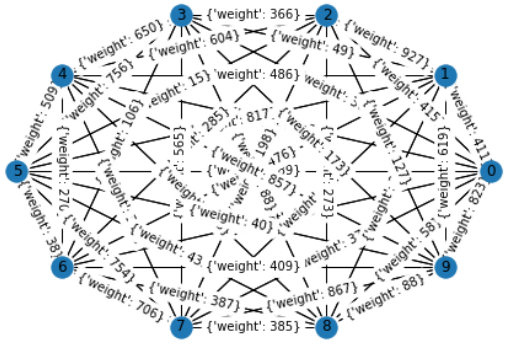


Slika 1: Primer grafa

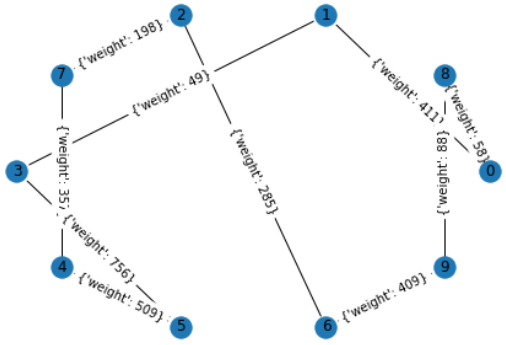


Slika 2: Rešitev primera z 2-opt

Še en primer:

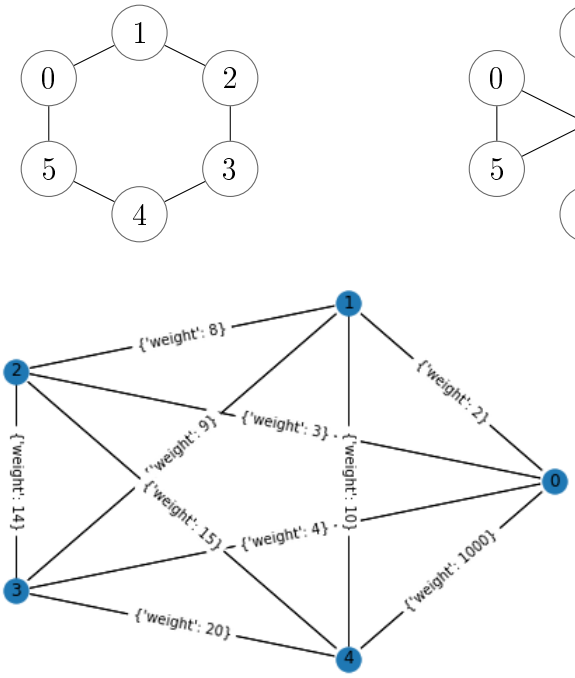


Slika 3: Primer grafa



Slika 4: Rešitev primera z 2-opt

2.2 3-opt algoritem



Slika 5: Primer grafa

Slika 6: Rešitev primera s 3-opt

3 Lin-Kernighanov algoritem

4 Viri

1. A. Hagberg, D. Schult, P. Swart:*NetworkX Refrence, Release 2.4*, [ogled 2. 1. 2020], dostopno na `https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf`

2. *Optimization with 2-OPT - Part 1*, [ogled 3. 1. 2020], dostopno na `http://pedrohfsd.com/2017/08/09/2opt-part1.html`

3. *2-opt*, [ogled 3. 1. 2020], dostopno na `https://en.wikipedia.org/wiki/2-opt`

4. *3-opt*, [ogled 4. 1. 2020], dostopno na `https://en.wikipedia.org/wiki/3-opt`

5. *3-opt: basic algorithm*, [ogled 4. 1. 2020], dostopno na `https://tsp-basics.blogspot.com/2017/03/3-opt-iterative-basic-algorithm.html`