UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Katarina Brilej, Sara Kovačič

Uporaba metahevristike GRASP na problemu potujočega trgovca

Projekt OR pri predmetu Finančni praktikum

Mentor: prof. dr. Riste Škrekovski

Ljubljana, 2019

Kazalo

1. Uvod

Metahevristika je algoritemski način reševanja kombinatoričnega optimizacijskega problema, pri katerem na začetku izberemo množico kandidatov za rešitev, in jo iterativno izboljšujemo (glede na neko vnaprej izbrano funkcijo zaželenosti), ter po dovolj korakih vrnemo najboljši element iz te množice. Metahevristike torej vrnejo približne rešitve, a veliko hitreje kot eksaktni postopki. V projektu bova na problem potujočega trgovca implementirali metahevristiko GRASP (*greedy randomized adaptive search procedure*). Problem potujočega trgovca bova rešili tudi kot celoštevilski linearni program in primerjali rešitve. Generirali bova nekaj zanimivih grafov in na njih preizkusili algoritem. Rezultate bova primerjail tudi z rezultati iz spleta in rezultati skupine 7, ki bo na problem potujočega trgovca implementirala genetski algoritem.

2. Problem potujočega trgovca

Problem potujočega trgovca ("travelling salesman problem"/TSP) se glasi:

- **Formulacija v vsakdanjem jeziku:** danih je n mest in razdalja med poljubnim parom mest (od mesta do mesta lahko potujemo po zgolj eni poti). Najdi najkrajšo (najcenejšo) pot, ki se začne in konča v istem mestu ter obišče vsako mesto natanko enkrat.
- **Formulacija v matematičnem jeziku**: v (neusmerjenem enostavnem) polnem grafu K_n z uteženimi povezavami (pozitivne vrednosti) najdi najkrajši cikel, ki vsebuje vsa vozlišča. Ciklom, ki vsebujejo vsa vozlišča grafa, pravimo Hamiltonovi cikli.

3. Grasp

GRASP (*greedy randomized adaptive search procedure*) je metahevristika, ki sestoji iz dveh faz: *greedy randomized construction* in *local search.* V prvi fazi na pameten način (odvisno od problema) izberemo izmed vseh možnih rešitev CL (candidate list) množico začetnih približkov RCL (restricted candidates list). To storimo deloma deterministično in deloma stohastično, da zagotovimo, da so začetni približki obetavni, a dovolj razpršeni po celotni množici CL, da bo druga faza pregledala čimvečji del CL. V drugi fazi za vsako izmed teh rešitev $s \in RCL$ pregledamo elemente $s' \in CL$ v njeni okolici (kaj je okolica je od problema in načina reševanja odvisno). Če najdemo boljšo rešitev s', jo dodamo v RCL ter s odstranimo. To ponavljamo dokler zaustavitveni pogoj (npr. št. iteracij, zahtevana natančnost) ni izpolnjen.

3.1. **Greedy randomized construction.** Kot smo že omenili je GRASP sestavljen iz dveh delov. Najprej bomo predstavili tako imenovan greedy randomized construction. Tu parameter alpha predstavlja moč množice začetnih približkov (RCL), v našem primeru torej dolžino seznama RCL. Za vhodni podatek imamo tudi incidenčno matriko cen povezav velikosti nxn. g je simetrična matrika, z ničlami po diagonali. Na začetku jo s pomočjo funkcije slovar_cen spremenimo v slovar, sestavljen iz elementov, ki jo imajo za ključ povezavo oblike $(x_i,\ x_j)$, za vrednost pa ceno/razdaljo, ki ji pripada. Vsak začetni približek $t = (l, 1, v_2, \ldots, v_n)$ iz RCL konstruiramo tako, da določimo $v_1 := 1$ nato iteraticno za$i = 2, \ldots, n$ za vi izberemo naključno med p % najbližjih vozlišč do $vi - 1$, ki še niso v t. Na koncu še izračunamo dolžino poti s prej definirano funkcijo dolzina_poti in jo postavimo na ničto mesto. Take cikle t konstruiramo toliko 2casa, da RCL napolnimo. Če delamo z matrikami velikosti manj kot 5, moramo nastaviti tudi parameter deles, saj mora vrednost n // delez presegati število 1.

```python
def slovar_cen(g):
    r = range(len(g))
    cena = {(i+1, j+1): g[i][j] for i in r for j in r}
    return cena
```

```python
def dolzina_poti(g,pot):
    n = len(g)
    slovar = slovar_cen(g)
    dolzina = 0
    for i in range(1,n):
            dolzina += slovar[(pot[i],pot[i+1])]
    dolzina += slovar[(pot[n],pot[1])]
    return dolzina
```

```python
def greedy_construction(g, alpha, delez = 5):
    RCL = [0] * alpha
    slovar = slovar_cen(g)
    n = len(g)
    p = n // delez
    for j in range(0, alpha):
        t = [0] * (n+1)
        t[1] = 1
        mesta = [h for h in range(2,n+1)]
        for i in range(2,n+1):
            povezave = [(t[i-1],m) for m in mesta]
            cene = { key:value for key, value in slovar.items() if key in povezave }
            urejene_povezave = sorted(cene, key=cene.__getitem__)
            (_,vi) = random.choice(urejene_povezave[:p])
            t[i] = vi
            mesta.remove(vi)
        t[0] = dolzina_poti(g,t)
        RCL[j] = t
    return RCL
```

3.2. **Local search.** V tem delu se bomo posvetili drugemu delu algoritma imenovanemu local search. Obravnavali bomo dve metodi 2opt in 3opt. Funkcija local_search zato poleg matrike g, parametra alpha in števila iteracij sprejme še parameter metodo. Na začetku definiramo RCL, ki predstavlja začetni seznam približkov. Te nato uredimo po dolžini, primerjamo jih po prvem elementu, ki predstavlja dolžino poti. Nato naključno izberemo t iz RCL, toda z linerano padajočo verjetnostjo. Najverjetneje izbereme cikel na vrhu RCL , torej najkrajši. Ko imamo izbran t se na podlagi parametra metoda odločimo za 2opt ali pa 3opt. Obe metodi poizkušata približek izboljšati, če jima uspe, vrneta novi_t. Neodvisno od metode nato v primeru, da je izboljšava uspela v RCL dodamo izboljšan približek in starega odstranimo. Kasneje si bomo ogledali še kako posamezna metoda deluje. Ponavljamo tolikokrat kot je predpisano, to nam določa iter.

```
def local_search(g,k,iter,metoda):
    RCL = greedy_construction(g,k)
    n = len(g)
    slovar = slovar_cen(g)
    #urejen seznam zacetnih priblizkov
    RCL.sort(key=lambda x: x[0])
    stevec = 0
    while stevec < iter:
        utezi = [i * 2/((k+1)*k) for i in range(k,0,-1)]
        indeks = np.random.choice(len(RCL), size = 1, p = utezi)
        t = RCL[indeks[0]]

        if metoda == "dva_opt":
            novi_t = dva_opt(n,t,g)
        elif metoda == "tri_opt":
            novi_t = tri_opt(n,t,g)

        if novi_t:
            RCL.append(novi_t)
            RCL.remove(t)
        stevec += 1
        RCL.sort(key=lambda x: x[0])
    RCL.sort(key=lambda x: x[0])
    return RCL[0]
```

Če si zdaj ogledamo še kako delujeta posamezni metodi. Začnimo s preprostejšo, 2opt. Ko naključno izberemo cikel t, ga želimo izboljšati. Krajši cikel iščemo v okolici, ki je definirana kot monožica vseh ciklov t' iz CL, ki jih dobimo iz t tako, da mu zamenjamo dve vozišči, torej naključno zamenjamo dve vozlišči t. Namesto, da bi shranjevali vse možne t' in na koncu preverili, če je najkrajši krajši od trenutnega t, je bolj učinkovito, če sproti preverjamo, če posamezna menjava prinese izboljšavo. Na začetku zato definiramo spremenljivko razlika, ta meri, če je dana menjava boljša. Nato moramo v zanki ločiti dva primera, saj v primeru, da je j = n, razdremo povezavo s prvim elementom. Nato izračunamo change, če je ta manjši od trenutne razlike, jo posodobimo, saj smo dobili krajši cikel. Shranimo si tudi optimalni i in j, da bomo na koncu vedeli s katero menjavo smo dosegli najkrajši cikel. Če smo uspeli izboljšati t, bo razlika negativna in dobili bomo novi_t. Tega konstruiramo tako, da na starem t izvedemo menjavo določeno z optimalnim i in j. Namesto, da znova računamo celotno dolžino cikla, samo prištejemo razliko, ki je nagativna.

```
def dva_opt(n,t,g):
    slovar = slovar_cen(g)
    razlika = 0
    for i in range(2,n):
        for j in range(i+1,n+1):
            if j != n:
                change = slovar[(t[i-1],t[j])] + slovar[(t[i],t[j+1])]
  - slovar[(t[i-1],t[i])] - slovar[(t[j],t[j+1])]
```

```
            else:
                change = slovar[(t[i-1],t[j])] + slovar[(t[i],t[1])] - slovar[(t[i-1],t[i])] - slovar[(t[j],t[1])]
            if change < razlika:
                razlika = change
                opt_i =  i
                opt_j = j
    if razlika < 0:
        novi_t = [t[m] for m in range(0,n+1)]
        novi_t[opt_i:opt_j+1] = novi_t[opt_i:opt_j+1][::-1]
        novi_t[0] = t[0] + razlika
        return novi_t
    else:
        return None
```

3opt je nekoliko bolj dodelana različica local searcha, zamenjamo namreč kar tri vozlišča. Tako kot 2opt sprejme velikost matrike, matriko in cikel t. Zaradi peglednosti na začetku definiramo spremenljivke X1, X2, Y1, Y2, Z1, Z2. Sedaj imamo 7 možnih menjav. Tri od teh so 2opt, štiri pa 3opt. Za vsako od možnih menjav nato izračunamo razliko. Ker pri 3opt menjavah odstranimo enake povezave, lahko to shranimo pod spremenljivko odstej. Ko izračunamo vse razlike, poiščemo minimum, s tem ko si zabeležimo, pri katerem indeksu je bila minimalna razlika dosežena, bomo kasneje vedeli katero menjavo izvesti, saj so te urejene po vrsti v seznamu spremembe. Če je change manjša od trenutne razlike, pomeni, da lahko t izboljšamo. Posodobimo razliko, indeks in si zabeležimo optimalne i,j in k, ki jih bomo potem potrebovali, da izvedemo ustrezno menjavo. Torej če je na koncu razlika manjša kot 0 lahko zgradimo novi_t. Ta je odvisen od vrste menjave, ki jo moramo izvršiti, ta pa je določena z indeksom in optimalnimi i,j in k. Tako kot pri 2opt tudi tukaj novo dolžino cikla izračunamo tako, da prištejemo razliko.

```
def tri_opt(n, t, g):
    slovar = slovar_cen(g)
    razlika = 0
    for i in range(2,n-1):
        for j in range(i+1,n):
            for k in range(j+1,n+1):
                X1, X2, Y1, Y2, Z1, Z2 = t[i-1], t[i], t[j-1], t[j], t[k-1], t[k]
# 2 opt moves
                change1 = slovar[(X1,Z1)] + slovar[(X2,Z2)] -  slovar[(X1,X2)] - slovar[(Z1,Z2)]
                change2 = slovar[(Y1, Z1)] + slovar[(Y2, Z2)] -  slovar[(Y1, Y2)] - slovar[(Z1, Z2)]
                change3 = slovar[(X1, Y1)] + slovar[(X2, Y2)] -  slovar[(X1, X2)] - slovar[(Y1, Y2)]
# 3 opt moves
                odstej = slovar[(X1, X2)] + slovar[(Y1, Y2)] + slovar[(Z1, Z2)]
# v vseh treh primerih odstranimo enake povezave
                change4 = slovar[(X1, Y1)] + slovar[(X2, Z1)] + slovar[(Y2, Z2)] -  odstej
```

```
                change5 = slovar[(X1, Z1)] + slovar[(Y2, X2)] + slovar[(Y1, Z2)] - odstej
                change6 = slovar[(X1, Y2)] + slovar[(Z1, Y1)] + slovar[(X2, Z2)] - odstej
                change7 = slovar[(X1, Y2)] + slovar[(Z1, X2)] + slovar[(Y1, Z2)] - odstej
# izracunamo najmanjso vrednost razlike
                spremembe = [change1,change2, change3, change4, change5,change6,change7]
                change = min(spremembe)
                ind = np.argmin(spremembe) + 1
                if change < razlika:
                    razlika = change
                    indeks = ind
                    opt_i =  i
                    opt_j = j
                    opt_k = k
    if razlika < 0:
        novi_t = menjava(indeks,n,t,opt_i,opt_j,opt_k)
        novi_t[0] = t[0] + razlika
        return novi_t
    else:
        return None
```

V naslednji funkciji so opisane menjave, ki jih moramo narediti na trenutnem ciklu t, odvisne pa so od indeksa.

```
def menjava(indeks,n,t,opt_i,opt_j,opt_k):
    novi_t = [t[m] for m in range(0,n+1)]
    if indeks == 1:
        novi_t[opt_i:opt_k] = novi_t[opt_i:opt_k][::-1]
    if indeks == 2:
        novi_t[opt_j:opt_k] = novi_t[opt_j:opt_k][::-1]
    if indeks == 3:
        novi_t[opt_i:opt_j] = novi_t[opt_i:opt_j][::-1]
    if indeks == 4:
        novi_t[opt_i:opt_j] = novi_t[opt_i:opt_j][::-1]
        novi_t[opt_j:opt_k] = novi_t[opt_j:opt_k][::-1]
    if indeks == 5:
        tmp = novi_t[opt_j:opt_k][::-1] + novi_t[opt_i:opt_j]
        novi_t[opt_i:opt_k] = tmp
    if indeks == 6:
        tmp = novi_t[opt_j:opt_k] + novi_t[opt_i:opt_j][::-1]
        novi_t[opt_i:opt_k] = tmp
    if indeks == 7:
        tmp = novi_t[opt_j:opt_k] + novi_t[opt_i:opt_j]
        novi_t[opt_i:opt_k] = tmp
    return novi_t
```

4. Celoštevilski linearni program in primerjava z GRASP

Problem potujočega trgovca lahko predstavimo kot *celoštevilski linearni program*. Označimo mesta s števili $1, \ldots, n$. Strošek (ali razdalja) potovanja iz mesta i v mesto j je $c_{i,j}$, ($1 \leq i, j \leq n$). Minimiziramo strošek potovanja. Definiramo:

$$X_{i,j} := \begin{cases} 1 \,; & \text{potnik gre iz mesta } i \text{ v mesto } j, \\ 0 \,; & \text{sicer,} \end{cases}$$

y_i ...katero po vrsti obiščemo mesto i

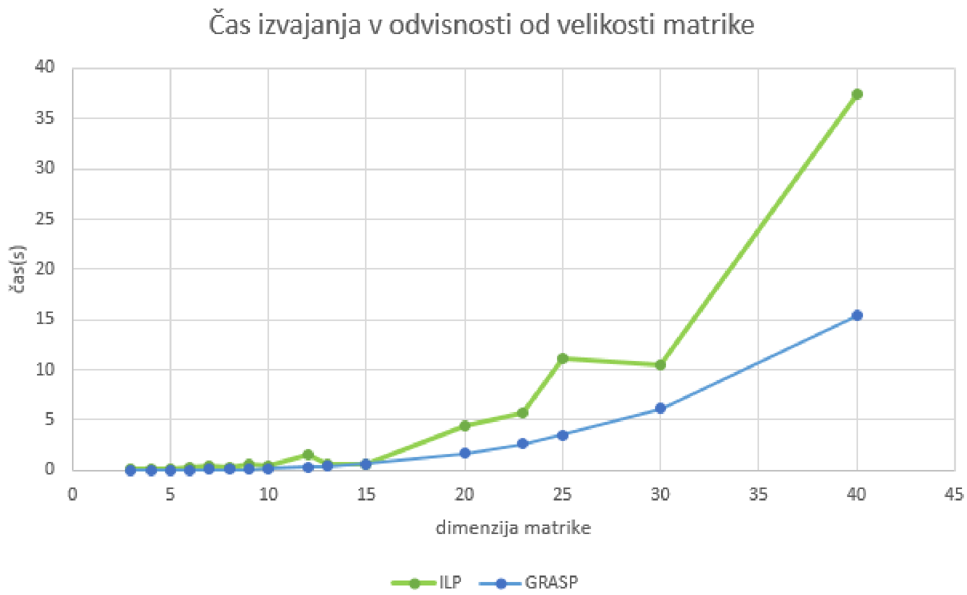
$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} \cdot c_{i,j} \\
\text{p. p.} \quad & \sum_{i=1}^{n} x_{i,j} = 1, \text{ za vsak } j \\
& \sum_{j=1}^{n} x_{i,j} = 1, \text{ za vsak } i \\
& x_{i,j} \in \{0, 1\}, \text{ za vsak } i \text{ in vsak } j \\
& y_i \in \{1, \ldots, n\}; \text{za vsak } i \\
& y_i + 1 - n + n \cdot x_{i,j} \leq y_j; \text{za vsak } i \text{ in vsak } j > 1
\end{aligned}$$

Problem potujočega trgovca se da v Pythonu elegantno zapisati kot celoštevilski linearni program s pomočjo knjižnice PuLP. PuLP za reševanje problema izbere enega od vgrajenih algoritmov. V našem primeru je PuLP za reševanje izbral PULP_CBC_CMD. S pomočjo knjižnice PuLP sva napisali funckijo, ki sprejme matriko cen povezav, izpiše minimalno ceno potovanja in vrne urejen seznam obiskanih mest.

```python
from pulp import *
def tsp_as_ilp(g):
    razdalje = slovar_cen_ilp(g)   # slovar razdalj
    mesta = [x+1 for x in range(len(g))]  # seznam mest (od 1 do n)
    prob = LpProblem("salesman", LpMinimize) # definiramo problem
    x = LpVariable.dicts('x', razdalje, 0,1, LpBinary)
    cena = lpSum([x[(i,j)]*razdalje[(i,j)] for (i,j) in razdalje])
    prob += cena
    for k in mesta:  # pri pogojih:
        prob += lpSum([ x[(i,k)] for i in mesta if (i,k) in x]) == 1
        prob += lpSum([ x[(k,i)] for i in mesta if (k,i) in x]) == 1
    u = LpVariable.dicts('u', mesta, 0, len(mesta)-1, LpInteger)
    N = len(mesta)
    for i in mesta:
        for j in mesta:
            if i != j and (i != 1 and j!= 1) and (i,j) in x:
                prob += u[i] - u[j] <= (N)*(1-x[(i,j)]) - 1
    prob.solve() # PuLP sam izbere metodo
    sites_left = mesta.copy()
```

```
    org = 1
    potovanje = []
    potovanje.append(sites_left.pop(sites_left.index(org)))
    while len(sites_left) > 0:
        for k in sites_left:
            if x[(org,k)].varValue == 1:
                potovanje.append(sites_left.pop(sites_left.index(k)))
                org = k
    potovanje.append(1)
    print("Optimalna cena potovanja = ", value(prob.objective))
    return potovanje
```

Z merjenjem časa reševanja določenih primerov različnih velikosti ugotovimo, da napisana funkcija *ilp* porabi več časa za reševanje, kot metahevristika *GRASP*.



5. Primerjave

Za različne primerjave smo se osredotočili na pet grafov, vendar pa ne bomo povsod primerjali vseh. Poleg grafov skupine 7, ti so ulysses22, berlin52 in kroA100 smo si izbrali še swiss42 in st70. Vse smo dobili iz TSPLIB, kjer so zapisane tudi trenutne znane optimalne rešitve, ki jih bomo tekom naslednjega poglavja tudi primerjali. Ker pa so podatki za TSP podani v različnih oblikah, smo napisali še tri različne funkcije za uvoz. Za primerjavo GRASP in ILP pa smo sami generirali nekoliko manjše matrike. Te so simetrične s celoštevilskimi vrednostmi od 1 pa do maksimalne vrednosti, ki jo določimo sami.

```
def TSP(N, max_pot, min_pot = 1):
    " funkcija vrne nakljucno matriko cen povezav, ki predstavlja
    problem potujocega trgovca"
    a = np.random.randint(min_pot, max_pot, size= (N,N))
    m = np.tril(a) + np.tril(a, -1).T
    for i in range(N):
        m[i][i] = 0
    return m
```

5.1. **Primerjava parametra alpha.**

5.2. **Primerjava ILP in GRASP.**

5.3. **Primerjava s skupino 7.**

6. Zaključek

Literatura

[1] Travelling salesman problem
http://en.wikipedia.org/wiki/Travelling_salesman_problem