

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Katarina Brilej, Sara Kovačič

**Uporaba metahevrstike GRASP na problemu potujočega
trgovca**

Projekt OR pri predmetu Finančni praktikum

Mentor: prof. dr. Riste Škrekovski

Ljubljana, 2019

KAZALO

| | |
|---|---|
| 1. Uvod | 3 |
| 2. Problem potujočega trgovca | 3 |
| 2.1. Celoštevilski linearni program in primerjava z GRASP | 3 |
| 3. Grasp | 4 |
| 3.1. Greedy randomized construction | 4 |
| 3.2. Local search | 5 |
| 4. Primerjave | 7 |
| 4.1. Primerjava parametra α | 7 |
| 4.2. Primerjava ILP in GRASP | 7 |
| 4.3. Primerjava s skupino 7 | 7 |
| 5. Zaključek | 7 |
| Literatura | 8 |

1. UVOD

Metahevrstika 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. Metahevrstike torej vrnejo približne rešitve, a veliko hitreje kot eksaktni postopki. V projektu bova na problem potujočega trgovca implementirali metahevrstiko 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 primerjali 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.

2.1. Celoštevilski linearni program in primerjava z GRASP. Problem potujočega trgovca lahko predstavimo kot *celoštevilske linearni program*. Označimo mesta s števili $1, \dots, n$. Strošek (ali razdalja) potovanja iz mesta i v mesto j je $c_{i,j}$, ($1 \leq i, j \leq n$). Minimiziramo strošek potovanja. Definiramo:

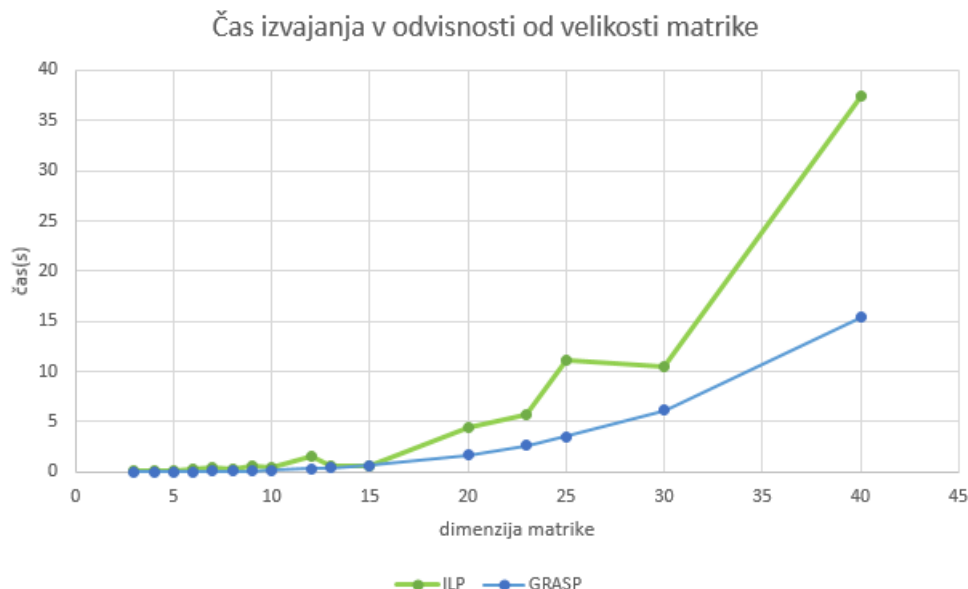
$$X_{i,j} := \begin{cases} 1 & \text{; potnik gre iz mesta } i \text{ v mesto } j, \\ 0 & \text{; sicer,} \end{cases}$$

$y_i \dots$ katero po vrsti obiščemo mesto i

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

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 funkcijo, ki sprejme matriko cen povezav, izpiše minimalno ceno potovanja in vrne urejen seznam obiskanih mest. Z merjenjem časa reševanja določenih primerov različnih velikosti ugotovimo, da napisana funkcija *ilp* porabi več časa za reševanje, kot metahevrstika *GRASP*.



3. GRASP

GRASP (*greedy randomized adaptive search procedure*) je metahevrstika, 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 α predstavlja moč množice začetnih približkov (RCL), v našem primeru torej dolžino seznama RCL. Za vhodni podatek imamo tudi incidenčno matriko cen povezav velikosti $n \times n$. 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 $t = (l, 1, v_2, \dots, v_n)$ iz RCL konstruiramo tako, da določimo $v_1 := 1$ nato iterativno za $i = 2, \dots, n$ za v_i izberemo naključno med $p\%$ najbližjih vozlišč do $v_i - 1$, ki še niso v t . Na koncu še izračunamo dolžino poti s prej definirano funkcijo `dolzina_poti` in jo postavimo na ničo 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 // deles` presežati število 1.

```

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

```

```

1 def dolzina_poti(g, pot):
2     n = len(g)
3     slovar = slovar_cen(g)
4     dolzina = 0
5     for i in range(1, n):
6         dolzina += slovar[(pot[i], pot[i+1])]
7     dolzina += slovar[(pot[n], pot[1])]
8     return dolzina

```

```

1 def greedy_construction(g, alpha, delez = 5):
2     RCL = [0] * alpha
3     slovar = slovar_cen(g)
4     n = len(g)
5     p = n // delez
6     for j in range(0, alpha):
7         t = [0] * (n+1)
8         t[1] = 1
9         mesta = [h for h in range(2, n+1)]
10        for i in range(2, n+1):
11            povezave = [(t[i-1], m) for m in mesta]
12            cene = {key: value for key, value in slovar.items() if key
13                    in povezave}
14            urejene_povezave = sorted(cene, key=cene.__getitem__)
15            (_, vi) = random.choice(urejene_povezave[:p])
16            t[i] = vi
17            mesta.remove(vi)
18            t[0] = dolzina_poti(g, t)
19            RCL[j] = t
20    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 izberemo 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 izboljššan približek in starega odstranimo. Kasneje si bomo ogledali še kako posamezna metoda deluje. Ponavljamo tolikokrat kot je predpisano, to nam določa iter.

```

1 def local_search(g,k,iter,metoda):
2     RCL = greedy_construction(g,k)
3     n = len(g)
4     slovar = slovar_cen(g)
5     #urejen seznam zacetnih priblizkov
6     RCL.sort(key=lambda x: x[0])
7     stevec = 0
8     while stevec < iter:
9         utezi = [i * 2/((k+1)*k) for i in range(k,0,-1)]
10        indeks = np.random.choice(len(RCL), size = 1, p = utezi)
11        t = RCL[indeks[0]]
12
13        if metoda == "dva_opt":
14            novi_t = dva_opt(n,t,g)
15        elif metoda == "tri_opt":
16            novi_t = tri_opt(n,t,g)
17
18        if novi_t:
19            RCL.append(novi_t)
20            RCL.remove(t)
21            stevec += 1
22            RCL.sort(key=lambda x: x[0])
23        RCL.sort(key=lambda x: x[0])
24    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 negativna.

```

1 def dva_opt(n,t,g):
2     slovar = slovar_cen(g)
3     razlika = 0
4     for i in range(2,n):
5         for j in range(i+1,n+1):
6             if j != n:
7                 change = slovar[(t[i-1],t[j])] + slovar[(t[i],t[j+1])]
8                     - slovar[(t[i-1],t[i])] - slovar[(t[j],t[j+1])]
9             else:
10                change = slovar[(t[i-1],t[j])] + slovar[(t[i],t[1])] -
11                    slovar[(t[i-1],t[i])] - slovar[(t[j],t[1])]

```

```

10         if change < razlika:
11             razlika = change
12             opt_i = i
13             opt_j = j
14         if razlika < 0:
15             novi_t = [t[m] for m in range(0,n+1)]
16             novi_t[opt_i:opt_j+1] = novi_t[opt_i:opt_j+1][::-1]
17             novi_t[0] = t[0] + razlika
18             return novi_t
19         else:
20             return None

```

3opt je nekoliko bolj dodelana različica local searcha.

```

1 def tri_opt(n, t, g):
2     slovar = slovar_cen(g)
3     razlika = 0
4     for i in range(2,n-1):
5         for j in range(i+1,n):
6             for k in range(j+1,n+1):
7                 X1, X2, Y1, Y2, Z1, Z2 = t[i-1], t[i], t[j-1], t[j], t[
8                     k-1], t[k]
9             # 2 opt moves
10             change1 = slovar[(X1,Z1)] + slovar[(X2,Z2)] - slovar[(
11                 X1,X2)] - slovar[(Z1,Z2)]
12             change2 = slovar[(Y1, Z1)] + slovar[(Y2, Z2)] - slovar
13                 [(Y1, Y2)] - slovar[(Z1, Z2)]
14             change3 = slovar[(X1, Y1)] + slovar[(X2, Y2)] - slovar
15                 [(X1, X2)] - slovar[(Y1, Y2)]
16             # 3 opt moves
17             odstej = slovar[(X1, X2)] + slovar[(Y1, Y2)] + slovar[(
18                 Z1, Z2)]
19             # v vseh treh primerih odstranimo enake povezave
20             change4 = slovar[(X1, Y1)] + slovar[(X2, Z1)] + slovar
21                 [(Y2, Z2)] - odstej
22             change5 = slovar[(X1, Z1)] + slovar[(Y2, X2)] + slovar
23                 [(Y1, Z2)] - odstej
24             change6 = slovar[(X1, Y2)] + slovar[(Z1, Y1)] + slovar
25                 [(X2, Z2)] - odstej
26             change7 = slovar[(X1, Y2)] + slovar[(Z1, X2)] + slovar
27                 [(Y1, Z2)] - odstej
28             spremembe = [change1, change2, change3, change4, change5
29                 , change6, change7]
30             change = min(spremembe)
31             ind = np.argmin(spremembe) + 1
32             if change < razlika:
33                 razlika = change
34                 indeks = ind
35                 opt_i = i
36                 opt_j = j
37                 opt_k = k
38             if razlika < 0:
39                 novi_t = menjava(indeks, n, t, opt_i, opt_j, opt_k)

```

```
31     novi_t[0] = t[0] + razlika
32     return novi_t
33 else:
34     return None
```

4. PRIMERJAVE

4.1. Primerjava parametra alpha.

4.2. Primerjava ILP in GRASP.

4.3. Primerjava s skupino 7.

5. ZAKLJUČEK

LITERATURA

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