

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
3. Grasp	3
3.1. Greedy randomized construction	3
3.2. Local search	4
4. Celoštevilski linearni program	7
5. Primerjave	8
5.1. Primerjava ILP in GRASP	9
5.2. Primerjava parametra alpha	9
5.3. Primerjava glede na število iteracij	10
5.4. Primerjava s skupino 7	11
5.5. Vizualizacija	12
6. Zaključek	13
Literatura	15

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.

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 pamenten 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

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 $n // \text{deles}$ presegati število 1.

```

1 def greedy_construction(g, alpha, deles = 5):
2     RCL = [0] * alpha
3     slovar = slovar_cen(g)
4     n = len(g)
5     p = n // deles
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 α in števila iteracij sprejme še parameter metoda. 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 izboljšan 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        if metoda == "dva_opt":
13            novi_t = dva_opt(n, t, g)

```

```

14     elif metoda == "tri_opt":
15         novi_t = tri_opt(n,t,g)
16     if novi_t:
17         RCL.append(novi_t)
18         RCL.remove(t)
19         stevec += 1
20         RCL.sort(key=lambda x: x[0])
21     RCL.sort(key=lambda x: x[0])
22     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 množica 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])]
12             if change < razlika:
13                 razlika = change
14                 opt_i = i
15                 opt_j = j
16         if razlika < 0:
17             novi_t = [t[m] for m in range(0,n+1)]
18             novi_t[opt_i:opt_j+1] = novi_t[opt_i:opt_j+1][::-1]
19             novi_t[0] = t[0] + razlika
20             return novi_t
21         else:
22             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 preglednosti 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.

```

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            # izracunamo najmanjso vrednost razlike
29            spremembe = [change1, change2, change3, change4, change5
30                , change6, change7]
31            change = min(spremembe)
32            ind = np.argmin(spremembe) + 1
33            if change < razlika:
34                razlika = change
35                indeks = ind
36                opt_i = i
37                opt_j = j
38                opt_k = k
39            if razlika < 0:
40                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

```

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

```

1 def menjava(indeks, n, t, opt_i, opt_j, opt_k):
2     novi_t = [t[m] for m in range(0, n+1)]
3     if indeks == 1:
4         novi_t[opt_i:opt_k] = novi_t[opt_i:opt_k][::-1]
5     if indeks == 2:
6         novi_t[opt_j:opt_k] = novi_t[opt_j:opt_k][::-1]
7     if indeks == 3:
8         novi_t[opt_i:opt_j] = novi_t[opt_i:opt_j][::-1]
9     if indeks == 4:
10        novi_t[opt_i:opt_j] = novi_t[opt_i:opt_j][::-1]
11        novi_t[opt_j:opt_k] = novi_t[opt_j:opt_k][::-1]
12    if indeks == 5:
13        tmp = novi_t[opt_j:opt_k][::-1] + novi_t[opt_i:opt_j]
14        novi_t[opt_i:opt_k] = tmp
15    if indeks == 6:
16        tmp = novi_t[opt_j:opt_k] + novi_t[opt_i:opt_j][::-1]
17        novi_t[opt_i:opt_k] = tmp
18    if indeks == 7:
19        tmp = novi_t[opt_j:opt_k] + novi_t[opt_i:opt_j]
20        novi_t[opt_i:opt_k] = tmp
21    return novi_t

```

4. CELOŠTEVILSKI LINEARNI PROGRAM

Problem potujočega trgovca lahko predstavimo kot *celoštevilski 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. V nadaljevanju je prikazan del funkcije *tsp_as_ilp*, v katerem definiramo problem in ga rešimo. Na repozitoriju se nahaja celotna funkcija, ki vrne urejen seznam obiskanih mest.

```

1 from pulp import *
2 def tsp_as_ilp(g):
3     razdalje = slovar_cen_ilp(g) # slovar razdalj
4     mesta = [x+1 for x in range(len(g))] # seznam mest (od 1 do n)
5     prob = LpProblem("salesman", LpMinimize) # definiramo problem
6     x = LpVariable.dicts('x', razdalje, 0, 1, LpBinary)
7     cena = lpSum([x[(i,j)]*razdalje[(i,j)] for (i,j) in razdalje])
8     prob += cena
9     for k in mesta: # pri pogojih:
10         prob += lpSum([ x[(i,k)] for i in mesta if (i,k) in x]) == 1
11         prob += lpSum([ x[(k,i)] for i in mesta if (k,i) in x]) == 1
12     u = LpVariable.dicts('u', mesta, 0, len(mesta)-1, LpInteger)
13     N = len(mesta)
14     for i in mesta:
15         for j in mesta:
16             if i != j and (i != 1 and j != 1) and (i,j) in x:
17                 prob += u[i] - u[j] <= (N)*(1-x[(i,j)]) - 1
18     prob.solve() # PuLP sam izbere metodo
19     print("Optimalna cena potovanja = ", value(prob.objective))

```

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.

```

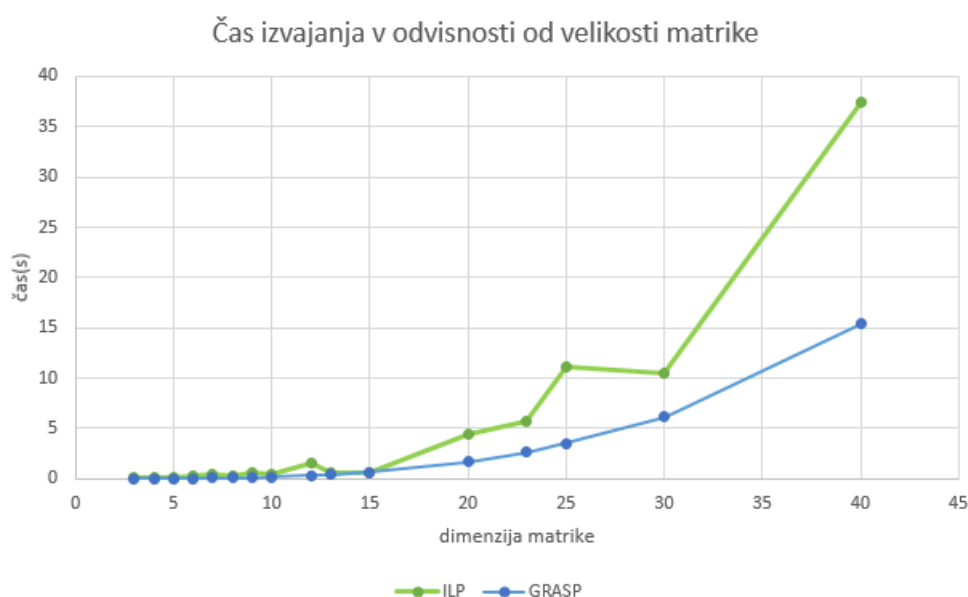
1 def TSP(N, max_pot, min_pot = 1):
2     " funkcija vrne naključno matriko cen povezav, ki predstavlja
   problem potujočega trgovca"
3     a = np.random.randint(min_pot, max_pot, size= (N,N))
4     m = np.tril(a) + np.tril(a, -1).T
5     for i in range(N):
6         m[i][i] = 0
7     return m

```

5.1. Primerjava ILP in GRASP.

Z merjenjem časa reševanja določenih primerov različnih velikosti ugotovimo, da napisana funkcija *ilp* porabi več časa za reševanje, kot metaheuristika *GRASP*. Za potrebe primerjave, smo za parametre pri *GRASP* vzeli $\alpha=3$, število iteracij=100 in metodo *3opt*. V nadaljevanju bomo ugotovili, da na čas izvajanja *local_search*-a vplivata tudi število iteracij in predvsem metoda, zato časovna primerjava funkcij *tsp_as_ilp* in *GRASP* ni najbolj točna. Ker za majhne grafe *GRASP* tudi pri 100 iteracijah vrne pravilno (optimalno) rešitev, lahko vseeno sklepamo, da deluje hitreje kot *tsp_as_ilp*.

SLIKA 1. Časovna primerjava ILP in GRASP



5.2. Primerjava parametra α .

Za primerjavo parametra α smo se osredotočili na štiri parametre in sicer 3, 5, 10 in 15. Parameter α nam določa velikost RCL (restricted client list) na začetku algoritma *GRASP* v sklopu greedy randomized construction. Torej če je α enak 5, bo algoritem skonstruiral 5 začetnih rešitev problema. Te bo nato *local search* izboljševal. Primerjavo smo naredili za obe metodi na 100 iteracijah, razen v zadnjem primeru na desetih, saj je bila matrika že prevelika. Prav tako smo naredili deset ponovitev algoritma. Če si najprej ogledamo najmanjši graf Ulysses22,

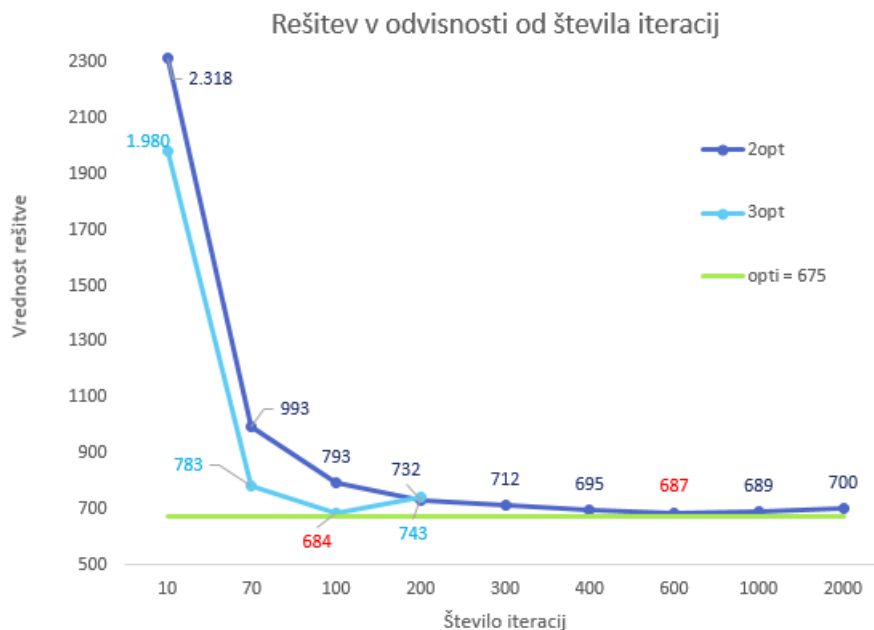
lahko vidimo, da izbira parametra ne vpliva na rezultat, saj v vseh primerih vrne znan optimalni rezultat 1273. Pri malo večjem grafu Swiss42 že opazimo, da se rezultat oddaljuje od optimuma, večji kot je α . Vendar pa je pri 2-opt slabšanje očitno hitrejše. Pri 3-opt celo kljub slabšem rezultatu pri $\alpha = 10$, pri $\alpha = 15$ vrne optimalen rezultat. Optimalen rezultat za Berlin52 je 7544, mi pa smo dobili 7616, kar je precej blizu. Se pa tu rezultat še hitreje poslabša z včanjem α . Spet pri 2-opt bolj kot pri 3-opt. Tudi pri St70 najboljši rezultat vrne 3-opt, zopet pri najmanjšem izbranem α . Za KroA100 so rezultati že močno oddaljeni od optimuma in vse bolj kot se povečuje α . Če zaključimo. Opazimo, da razen v primeru zelo majnih grafov, z uporabo manjšega α (npr. 3), dobimo boljši rezultat.

TSP		3	5	10	15	opt
Ulysses22						7013
	2-opt	7013	7013	7013	7013	
	3-opt	7013	7013	7013	7013	
Swiss42						1273
	2-opt	1273	1273	1420	1592	
	3-opt	1273	1273	1316	1273	
Berlin52						7542
	2-opt	7716	8130	8629	10464	
	3-opt	7616	7684	7735	8705	
St70						675
	2-opt	714	888	1211	1386	
	3-opt	684	754	878		
KroA100						21282
	2-opt	37162	54119	74951	74462	
	3-opt					

5.3. Primerjava glede na število iteracij.

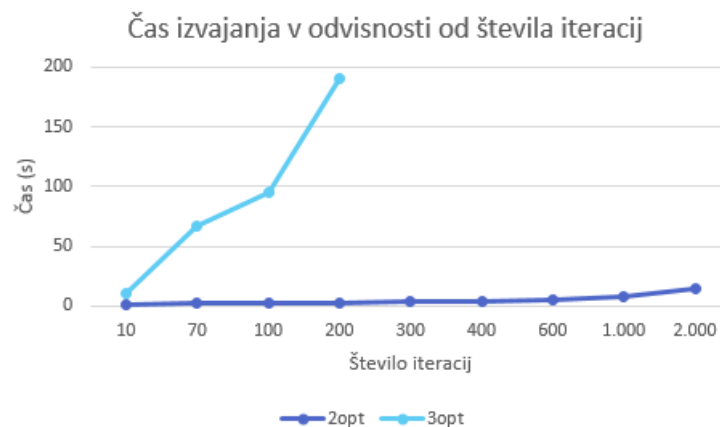
Pri določanju parametrov algoritma ima pomembno vlogo tudi število iteracij. Odvisnost vrednosti rešitve od števila iteracij smo preizkusili na primeru St70 (velikosti 70). Iz prejšnjega podpoglavja ugotovimo, da je za ta primer najboljši parameter $\alpha = 3$. Zaradi velike časovne zahtevnosti metode 3opt, smo za ta primer naredili le 200 iteracij. Ugotovimo, da se vrednost rešitve izboljšuje z večanjem iteracij. Metoda 3opt je pri preizkušanju dosegla najboljši rezultat (684) že pri 100 iteracijah. Podobno rešitev pri metodi 2opt dobimo še le pri 600 iteracijah.

SLIKA 2. Odvisnost rešitve od števila iteracij



Zaradi manjše časovne zahtevnosti metode 2opt, lahko pri izvajanju GRASP-a uporabimo več iteracij kot pri metodi 3opt. Metoda 3opt uporabi $\mathcal{O}(n^3)$ operacij, 2opt pa ima kvadratično časovno zahtevnost oz. $\mathcal{O}(n^2)$.

SLIKA 3. Odvisnost časa izvajanja od števila iteracij



5.4. Primerjava s skupino 7.

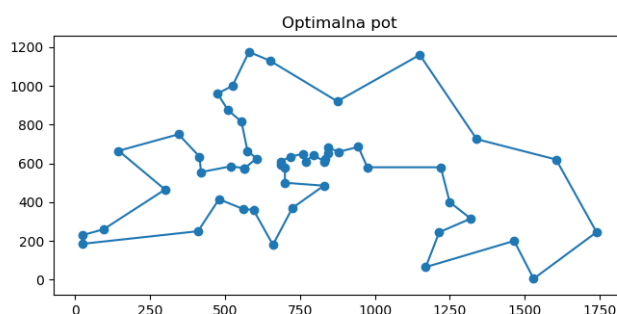
	GRASP	GA
Ulysses22	7013	7112
Berlin52	7544	8737
KroA100	21761	36408

Skupina 7 je na problem potujočega trgovca implementirala Genetski Algoritem (GA). Iz njihovega poročila sva poiskali njihove najboljše rezultate posameznega

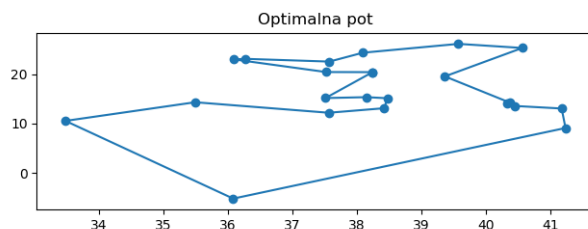
grafa in jih zapisali v tabelo. Iz tabele razberemo, da se je algoritem GRASP odrezal bolje od genetskega algoritma. Tudi iz vira [6] razberemo, da je genetski algoritem sicer natančnejši, vendar se na primerih iz prakse pokaže, da natančnost ni tako pomembna, in zato GRASP deluje bolje od GA.

5.5. Vizualizacija. Za boljšo predstavo sva napisali še dve funkciji, ki narišeta podano omrežje in rešitev na njem. Če imamo podane koordinate, funkcija *narisi* nariše točke kot koordinate v 2D in jih poveže glede na dano rešitev.

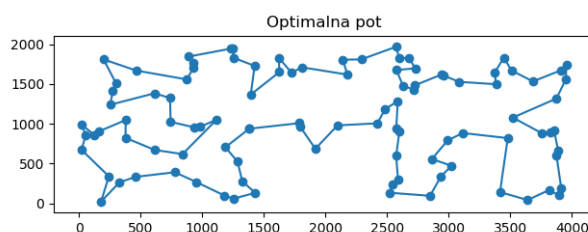
SLIKA 4. Najkrajša pot primera berlin52 dolga 7544



SLIKA 5. Najkrajša pot primera ulysses22, dolga 7013

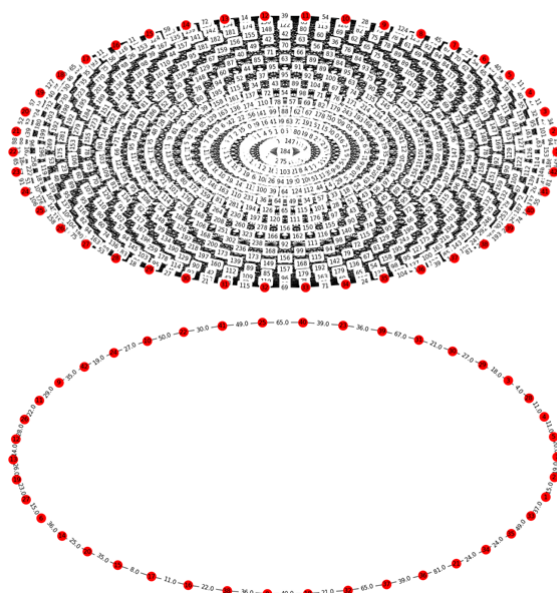


SLIKA 6. Najkrajša pot primera kroA100, dolga 21761



Če pa koordinat nimamo podanih, pa funkcija *narisigraf(matrika)* sprejme matriko uteži povezav in nariše usmerjen graf z utežmi na povezavah. Napisali sva še funkcijo, ki rešitev algoritma GRASP spremeni v matriko, tako da še lahko narišemo rešitev. Opomba: pri velikih grafih je risanje vseh povezav nesmiselno, zato te funkcije nisva izboljševali.

SLIKA 7. Poln graf in primer rešitve za swiss42



6. ZAKLJUČEK

Problem potujočega trgovca je NP-težak problem v kombinatorični optimizaciji, ki je pomemben v operacijskih raziskavah in računalniških znanostih. Problem je bil prvič oblikovan leta 1930 in je eden izmed najbolj intenzivno proučenih problemov optimizacije. Na problem sva implementirali metahevrstiko GRASP, ki sestoji iz dveh faz (Greedy randomized construction in Local search). V Local search-u sva obravnavali metodi 2opt in 3opt, kjer je glavna razlika to, da pri 2opt zamenjamo dve vozlišči, pri 3opt pa tri. Metoda 3opt vsebuje tri for zanke, zato je njena časovna zahtevnost kar $\mathcal{O}(n^3)$. Vsak 3-opt premik je bodisi enak 2-opt premiku ali je enak zaporedju dveh ali treh 2-opt potez. Čeprav je 3-opt bolj zapleten in počasnejši, se uporablja, ker je možno, da obstaja zaporedje 2-opt premikov, ki izboljšajo pot vendar se začne z 2-opt premikom, ki poveča dolžino poti, zaradi česar to zaporedje premikov ne bo izvedeno. V Pythonu lahko problem potujočega trgovca zapišemo kot celoštevilski linearni program (ILP) s pomočjo knjižnice PULP, ki za reševanje problema izbere enega od vgrajenih algoritmov. Z merjenjem časa izvajanja funkcij, sva ugotovili, da GRASP (100 iteracij, metoda 3opt) deluje hitreje kot ILP. Primerjave različnih parametrov metode GRASP sva izvedli na grafih ulysses22, berlin52, kroA100, swiss42 in st70. Najprej sva primerjali parameter alpha, ki določa koliko različnih začetnih rešitev problema bo skonstruiral algoritem. Na rešitve manjših grafov parameter alpha ni vplival, pri večjih grafih pa je manjši alpha pomenil boljšo rešitev. Zato sva v nadaljevanju uporabljali vrednost $\alpha = 3$. Vrednost rešitve se izboljšuje z večanjem števila iteracij, vendar se s tem veča tudi časovna zahtevnost. Metoda 3opt je na primeru st70 za 200 iteracij potrebovala že skoraj 200 sekund, vendar je že pri 100 iteracijah vrnila boljši rezultat kot 2opt pri 1000 ali 2000. Metoda 2opt tudi za 2000 iteracij potrebuje manj kot 30 sekund. Rezultate algoritma GRASP sva na primerih ulysses22, berlin52 in kroA100 primerjali tudi s skupino 7, ki je izvajala Genetski algoritem (GA). Ugotovili sva, da so se najinim rezultatom najbolj približali pri najmanjšem grafu ulysses22, kjer GRASP zelo hitro vrne kar

optimalno rešitev (7013). Pri grafu velikosti 100 (KroA100), pa je GRASP deloval veliko bolje, saj se je od znane optimalne rešitve (21282) razlikoval za 479, rešitev genetskega algoritma pa se je od te rešitve razlikovala za kar 15126. Ugotovimo, je da algoritem zaradi svoje enostavnosti na splošno zelo hiter in je zmožen producirati kar dobre rezultate v kratkem času.

LITERATURA

- [1] Travelling salesman problem
http://en.wikipedia.org/wiki/Travelling_salesman_problem
- [2] C. Blum, A. Roli, Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, online.
https://www.iiaa.csic.es/~christian.blum/downloads/blum_rol_i_2003.pdf
- [3] S. Luke, Essentials of Metaheuristics: a set of undergraduate lecture notes, online.
<https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf>
- [4] TSP Basics
<https://tsp-basics.blogspot.com/2017/03/>
- [5] R. Škrekovski, Zbornik seminarjev iz hevristik [Elektronski vir]: izbrana poglavja iz optimizacijskih metod (2010-11) / Riste Škrekovski, Vida Vukašinovič - El. knjiga. - Ljubljana: samozal. R. Škrekovski, 2012.
https://www.fmf.uni-lj.si/~skreko/Gradiva/Zbornik_Hevristike.pdf
- [6] B. Bernai, S. Deleplanque, A. Quilliot, Routing on Dynamic Networks: GRASP versus Genetic.
https://annals-csis.org/Volume_2/pliks/52.pdf