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

Slike

Tabele

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 bomo na problem potujočega trgovca implementirali metahevristiko GRASP (*greedy randomized adaptive search procedure*). Problem potujočega trgovca bomo rešili tudi kot celoštevilski linearni program in primerjali rešitve. Generirali bomo nekaj zanimivih grafov in na njih preizkusili algoritem. Rezultate bomo 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 obe-tavni, 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 faz. Najprej bomo predstavili prvo fazo oz. *greedy randomized construction*. Tu parameter alpha predstavlja moč množice začetnih približkov (RCL), v našem primeru torej dolžino seznama RCL. Algoritmu podamo tudi slovar razdalj (cen povezav) med mesti, n pa predstavlja dolžino slovarja oz. število mest. Slovar je sestavljen iz elementov, ki imajo za ključ povezavo oblike (x_i, x_j), za vrednost pa razdaljo/ceno, ki ji pripada. Vsak začetni približek $t = (l, 1, v_2, \ldots, v_n) \in$ RCL konstruiramo tako, da določimo $v_1 := 1$ nato iterativno za $i = 2, \ldots, n$ 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 časa, da RCL napolnimo.

```
1  def greedy_construction(slovar,n,alpha):
2      RCL = [0] * alpha
3      if n > 5: # p mora biti vsaj 1
4          p = n // 5
5      else:
6          p = 1
7      for j in range(0, alpha):
8          t = [0] * (n+1)
9          t[1] = 1
10         mesta = [h for h in range(2,n+1)]
11         for i in range(2,n+1):
12             povezave = [(t[i-1],m) for m in mesta]
13             slovar_povezav = { key:value for key, value in
14             slovar.items() if key in povezave }
15             urejene_povezave = sorted(slovar_povezav,
16             key=slovar_povezav.__getitem__)
17             (_,vi) = random.choice(urejene_povezave[:p])
18             t[i] = vi
19             mesta.remove(vi)
20         t[0] = dolzina_poti(slovar,n,t)
21         RCL[j] = t
22     return RCL
```

3.2. **Local search.** Drugo fazo algortima predstavlja *local search*. Obravnavali bomo dve metodi: 2-opt in 3-opt. Funkcija *local_search* zato poleg matrike g, parametra alpha in števila iteracij sprejme še parameter *metoda*. g je podana kot incidenčna matrika razdalj med mesti/cen povezav velikosti $n \times n$, je simetrična ter z ničlami na diagonali. Funkcija $slovar_razdalj$ matriko g pretvori v slovar kot je opisan v prvi fazi. V vrstici 2 definiramo RCL, ki predstavlja začetni seznam približkov. Te nato uredimo po dolžini, primerjamo jih po prvem elementu, ki predstavlja dolžino poti. Nato naključno izberemo $t \in$ RCL, toda z linerano padajočo verjetnostjo. Zato v vrstici 8 definiramo uteži. Najverjetneje izberemo cikel na vrhu RCL , torej najkrajši. Ko imamo izbran t, se na podlagi parametra *metoda* odločimo za 2-opt ali 3-opt. Obe metodi poizkušata približek izboljšati, če jima uspe, vrneta *novi_t*. Neodvisno od metode nato v primeru, da smo približek uspeli izboljšati, 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 parameter *iter*.

```
1  def local_search(g,alpha,iter,metoda):
2      n = len(g)
3      slovar = slovar_razdalj(g)
4      RCL = greedy_construction(slovar,n,alpha)
5      RCL.sort(key=lambda x: x[0])
6      stevec = 0
7      while stevec < iter:
8          utezi = [i * 2/((alpha+1)*alpha) for i in range(alpha,0,-1)]
9          indeks = np.random.choice(len(RCL), size = 1, p = utezi)
10         t = RCL[indeks[0]]
11         if metoda == "dva_opt":
```

```
            novi_t = dva_opt(n,t,slovar)
        elif metoda == "tri_opt":
            novi_t = tri_opt(n,t,slovar)
        if novi_t:
            RCL.append(novi_t)
            RCL.remove(t)
        stevec += 1
        RCL.sort(key=lambda x: x[0])
    return RCL[0]
```

Sedaj si oglejmo kako delujeta posamezni metodi. Začnimo s preprostejšo, 2-opt. 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. 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 (če smo uspeli skrajšati pot). 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 optimalnima i in j. Namesto, da znova računamo celotno dolžino cikla, samo prištejemo razliko, ki je nagativna.

```
def dva_opt(n, t, slovar):
    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
```

3-opt je nekoliko bolj dodelana različica *local searcha*, zamenjamo namreč kar tri vozlišča. 3-opt metoda nam omogoča pobeg v primeru, ko bi se z 2-opt metodo ujeli v lokalnem minimumu in ne bi mogli ven. Tako kot 2-opt funkcija sprejme velikost matrike, cikel t in slovar povezav. Zaradi preglednosti na začetku definiramo spremenljivke X1, X2, Y1, Y2, Z1, Z2. Sedaj imamo 7 možnih menjav. Tri od teh so

2-opt (torej čeprav razdremo tri povezave, nato eno spet povežemo nazaj), štiri pa 3-opt (vse tri ostanejo nepovezane). Za vsako od možnih menjav nato izračunamo razliko. Ker pri 3-opt menjavah odstranimo enake povezave, lahko to shranimo pod spremenljivko *odstej*. Ko izračunamo vse razlike, poiščemo minimum. S tem ko si zabeležimo, pri katerih indeksih 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 2-opt tudi tukaj novo dolžino cikla izračunamo tako, da prištejemo razliko. Če opišemo se eno bolj zahtevno menjavo, recimo 3-opt menjavo change4. Prekinemo povezave (X1,X2), (Y1, Y2) in (Z1, Z2), zato te vrednosti odštejemo. Na novo pa v tem primeru zgradimo povezave (X1, Y1), (X2, Z1) in (Y2, Z2), zato te vrednosti prištejemo. Na ta način izračunamo razliko v dolžini cikla, ki jo ta menjava prinese, kako natančno pa to menjavo izvedemo bomo opisali pri funkciji *menjava*.

```
def tri_opt(n, t, slovar):
    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)]
                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
                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
```

Funkcija *menjava* sprejme indeks oz. katero mejavo želimo narediti, velikost matrike n, naključno izbrani cikle $t \in$ RCL ter optimalne i,j in k pri katerih je bila dosežena minimalna razlika. Če opišemo menjavo 4. Nove povezave (X1, Z1),(Y2, X2) in (Y1, Z1) dobimo tako da obrnemo del cikla med Y1 in X2 ter med Z1 in Y2. Vse ostale menjave, prav tako pa izračuni razlik so opisani v datoteki GRASP.py.

```
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

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 smo napisali funckijo, 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 funckija, ki vrne urejen seznam obiskanih mest.

```
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
    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

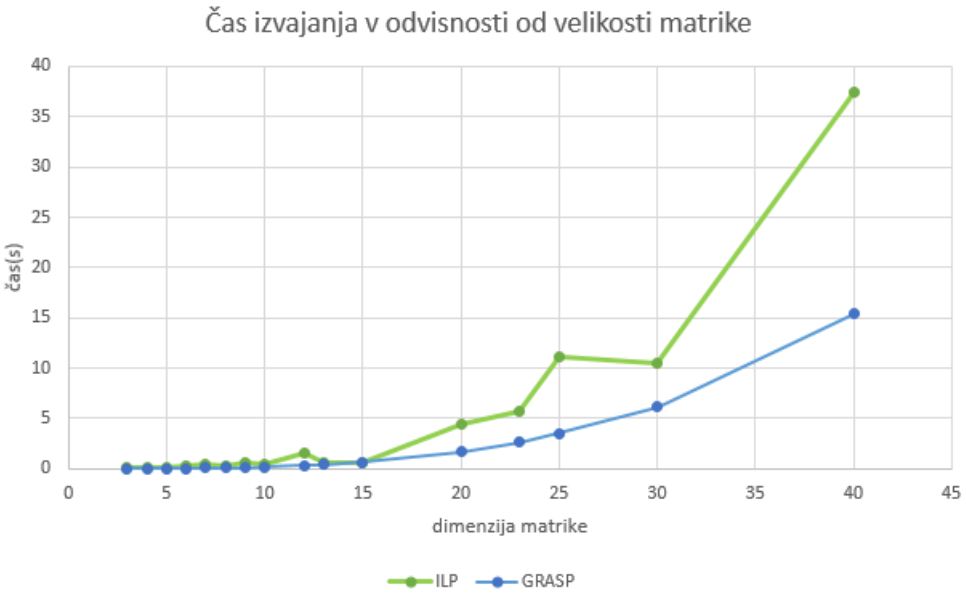
nekoliko manjše matrike. Te so simetrične s celoštevilskimi vrednostmi od 1 pa do maximalne vrednosti, ki jo določimo sami.

```
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 ILP in GRASP.**

Z merjenjem časa reševanja določenih primerov različnih velikosti ugotovimo, da napisana funkcija *ilp* porabi več časa za reševanje, kot metahevristika *GRASP*. Za potrebe primerjave, smo za parametre pri *GRASP* vzeli alpha=3, št. iteracij=100 in metodo 3-opt. 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 alpha.**

Za primerjavo parametra alpha sva se osredotočili na štiri parametre in sicer 3, 5, 10 in 15. Parameter alpha nam določa velikost RCL, torej število začetnih približkov. Če je alpha enak 5, bo funkcija *greedy_construction* skonstruirala 5 začetnih približkov. Te bo nato *local_search* izboljševal. Za začetek smo primerjavo naredili za obe metodi na 100 iteracijah, naredili smo deset ponovitev algoritma. Če si najprej ogledamo najmanjši graf *Ulysses22*, lahko vidimo, da izbira parametra ne

vpliva na rezultat, saj v vseh primerih algoritem vrne znan optimalni rezultat 1273. Pri malo večjem grafu *Swiss42* že opazimo, da se rezultat oddaljuje od optimuma, večji kot je alpha. 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 večanjem alpha, zopet pri 2-opt bolj kot pri 3-opt. Tudi pri *St70* najboljši rezultat vrne 3-opt, spet pri najmanjšem izbranem alpha. Za *KroA100* so rezultati že močno oddaljeni od optimuma vse bolj kot se povečuje alpha. Če povzamemo. Opazimo, da razen v primeru zelo majnih grafov, z uporabo manjšega alpha (npr. 3), dobimo boljši rezultat. Večanje alpha pa bolj vpliva na metodo 2-opt kot 3-opt, saj se ta začne hitreje odmikati od optimuma. Prav tako ima spreminjanje parametra večji vpliv pri večjih grafih. Opazimo tudi, da pri enakem številu iteracij metoda opt-3 načeloma vrne boljši rezultat. To je smiselno, saj vsakič pregleda večjo okolico cikla, ki ga želim oizbolj

Tabela 1. Primerjava parametra alpha, iter = 100, ponovitve = 10

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	941	
KroA100						21282
	2-opt	37162	54119	74951	74462	
	3-opt	22917	26938	41953	46026	

Za naslednjo primerjavo smo se odločili primerjati še različne parametre alpha pri številu iteracij = 1000. Zdaj smo primerjali samo metodo 2-opt, dodali pa smo še alpha = 30. Zopet pa smo naredili 10 ponovitev. Opazimo, da zdaj tudi za primer *Swiss42* metoda vrne optimalno vrednost, neodvisno od alpha. Pri grafu *Berlin52* se vrednosti za vse prejšnje alpha močno izboljšajo, nekolikšno odstopanje lahko zasledimo le pri alpha = 30. Tudi pri primerih *St70* in *KroA100* se vrednosti precej bolje približajo optimalni, tudi oddaljevanje od optimuma v odvisnosti z večanjem alpha je počasnejše kot pri iter = 100. Zopet lahko najmanjše vrednosti v povprečju opazimo pri manjših alpha, vendar pa se zdaj vpliv izbranih parametrov alpha zmanjša. Sklepamo lahko torej, da si lahko pri večjem številu iteracij lahko "privošcimo"večji alpha in bo algoritem še vedno vračal optimalno rešitev. Prav tako so rešitve pri danem številu iteracij boljše za manjše grafe.

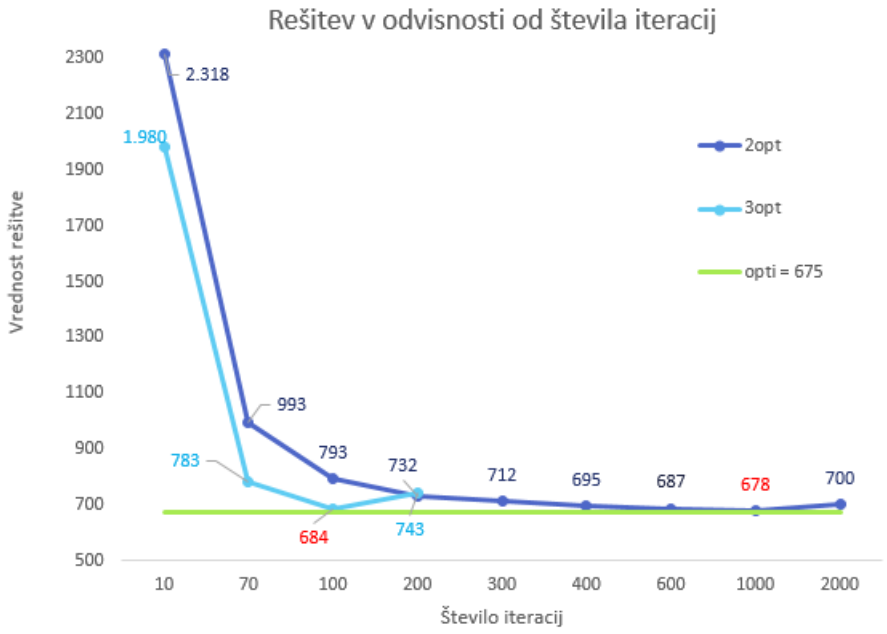
TABELA 2. Primerjava parametra alpha, iter = 1000, ponovitve = 10

TSP		3	5	10	15	30	opt
Swiss42							1273
	opt-2	1273	1273	1273	1273	1273	
Berlin52							7542
	opt-2	7544	7544	7544	7544	7560	
St70							675
	opt-2	678	685	678	682	685	
KroA100							21282
	opt-2	21381	21709	21642	22009	24275	

5.3. **Primerjava glede na število iteracij.**

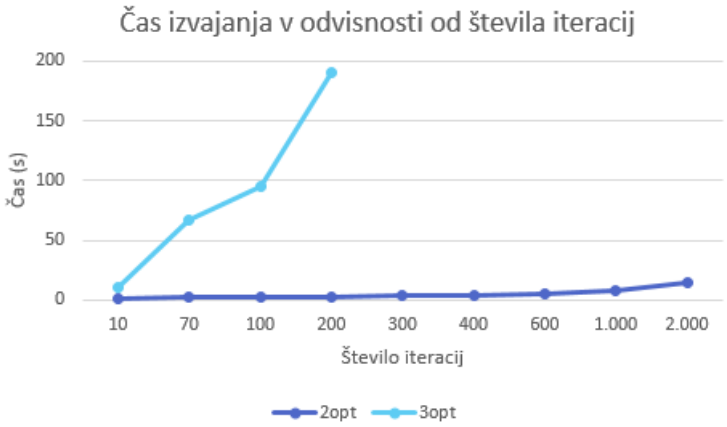
Pri določanju parametrov algoritma ima pomembno vlogo tudi število iteracij. Odvisnost vrednosti rešitve od števila iteracij smo preizkusili na primeru St70, velikosti 70. Iz prejšnjega podpoglavja ugotovimo, da je za ta primer najboljši parameter alpha = 3. Zaradi velike časovne zahtevnosti metode 3-opt, smo za ta primer naredili le 200 iteracij. Ugotovimo, da se vrednost rešitve izboljšuje z večanjem iteracij. Metoda 3-opt je pri preizkušanju dosegla dober rezultat (684) že pri 100 iteracijah. Podobno rešitev pri metodi 2-opt dobimo šele pri 600 iteracijah, najboljšo pa pri 1000.

SLIKA 2. Odvisnost rešitve od števila iteracij



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

Slika 3. Odvisnost časa izvajanja od števila iteracij



5.4. **Primerjava s skupino 7.**

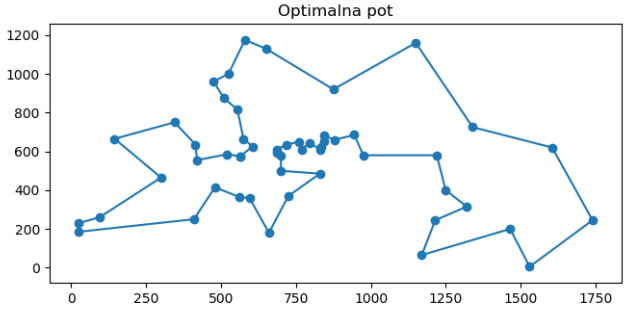
	GRASP	GA
Ulysses22	7013	7112
Berlin52	7544	8737
KroA100	21381	36408

Skupina 7 je na problem potujočega trgovca implementirala Genetski Algoritem (GA). V njihovem poročilu smo poiskali 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.

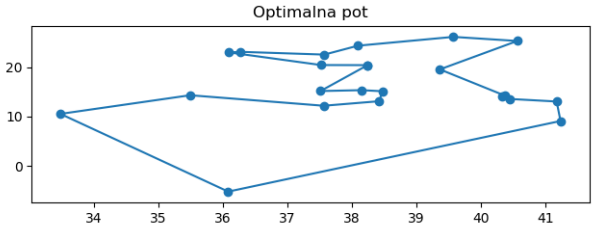
5.5. **Vizualizacija.**

Za boljšo predstavo smo 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.

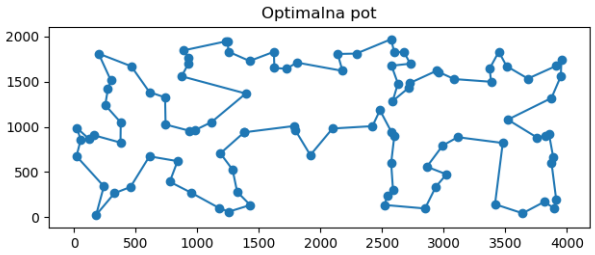
Slika 4. Najkraša pot primera Berlin52 dolga 7544



Slika 5. Najkrajša pot primera Ulysses22, dolga 7013

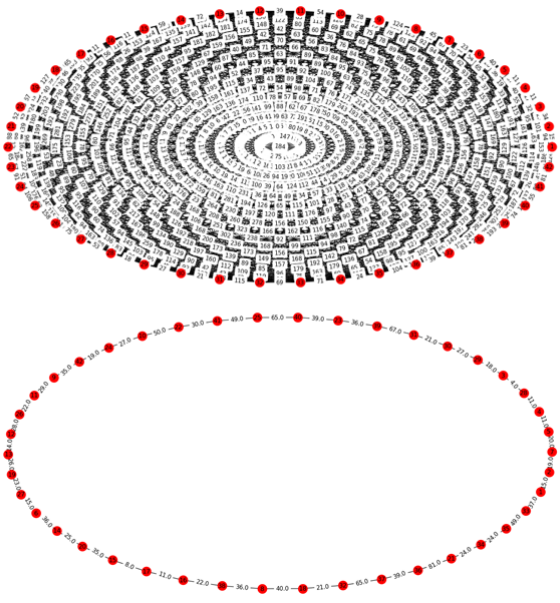


Slika 6. Najkrajša pot primera KroA100, dolga 21381



Če pa koordinat nimamo podanih, pa funckija $narisigraf(matrika)$ sprejme matriko uteži povezav in nariše usmerjen graf z utežmi na povezavah. Napisali smo 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 nismo 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 proučevanih problemov optimizacije. Na problem smo implementirali metahevristiko GRASP, ki sestoji iz dveh faz (Greedy randomized construction in Local search). V Local search-u smo obravnavali metodi 2-opt in 3-opt, kjer je glavna razlika to, da pri 2-opt zamenjamo dve vozlišči, pri 3-opt pa tri. Metoda 3-opt 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, smo ugotovili, da GRASP (100 iteracij, metoda 3opt) deluje hitreje kot ILP. Primerjave različnih parametrov metode GRASP smo izvedli na grafih *Ulysses22*, *Berllin52*, *KroA100*, *Swiss42* in *St70*. Najprej smo primerjali parameter alpha, ki določa koliko 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 smo v nadaljevanju uporabljali vrednost alpha = 3. Prav tako smo ugotovili, da lahko pri večjem številu iteracij vzamemo večji alpha da bo rešitev še vedno optimalna. Tako pri metodi 2-opt in 3-opt smo z večanjem alpha zasledili oddaljevanje od optimuma (vsaj pri večjih grafih), oddaljevanje pa je bilo hitrejše pri metodi 2-opt. Vrednost rešitve se izboljšuje z večanjem števila iteracij, vendar se s tem veča tudi časovna zahtevnost. Metoda 3-opt je na primeru *St70* za 200 iteracij potrebovala že skoraj 200 sekund, vendar je že pri 100 iteracijah vrnila podoben rezultat kot 2-opt pri 1000 ali 2000. Metoda 2opt tudi za 2000 iteracij potrebuje manj kot 30 sekund. Rezultate algoritma GRASP smo na primerih *Ulysses22*, *Berllin52* in *KroA100* primerjali tudi s skupino 7, ki je izvajala Genetski algoritem (GA). Ugotovili smo, da so se našim rezultatom najbolje 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 99, rešitev genetskega algoritma pa se je od te rešitve razlikovala za kar 15126. Ugotovimo, da je 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.iiia.csic.es/ christian.blum/downloads/blum_roli_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.pdff