UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Tina Bertok
Neža Habjan
Gašper Letnar

Genetski algoritem na problemu potujočega trgovca

Projekt v povezavi z OR

Ljubljana, 2018

Kazalo

1. Navodilo

Implement the genetic algorithm metaheuristic for TSP. Present the chromosomes as ordered lists i.e. as paths. Apply different variations for the crossover operations, such as order crossover (OX), paritally mapped crossover (PMX), cycle crossover (CX), etc. Experiment with different sizes of the population. You can generate some of the testing graphs yourself, and you can find some of them on the Internet.

2. Uvod

Naša naloga je, da implementiramo genetski algoritem na problemu potujočega trgovca. Pri tem bomo uporabljali različna križanja, spreminjali velikosti populacije in verjetnosti za mutacijo. Primerjali bomo rezultate pri različnih pogojih. Naš algoritem bomo testirali na podatkih, ki jih bomo generirali sami in na podatkih najdenih na internetu.

Genetski algoritem je metahevristika, navdihnjena s strani procesov naravne selekcije in spada v razred razvojnih algoritmov. Uporablja se za generiranje kvalitetnih rešitev v optimizaciji, ki temeljijo na operatorjih kot so mutacija, križanje in selekcija.
V genetskem algoritmu se uporabi množica kandidatov za rešitev, ki jih nato razvijamo do čim boljše rešitve. Vsak kandidat ima določene lastnosti, katere lahko spremenimo oziroma lahko mutirajo. Evolucija rešitev se ponavadi začne na naključni izbiri kandidatov, katere potem s pomočjo iteracije razvijamo. Na vsakem iterativnem koraku se potem oceni primernost novih kandidatov za optimizacijski problem. Najboljše kandidate potem uporabimo za naslednji korak iteracije. Algoritem se zaključi, ko je izpolnjen zaustavitveni kriterij. Ena možnost je, da algoritem ustavimo po tem, ko preteče določeno število generacij.

Problem trgovskega potnika oz. *travelling salesman problem (TSP)* je NP-težek problem v kombinatorični optimizaciji, pomemben pri operacijskih raziskavah, matematični optimizaciji in teoretičnemu računalništvu. Trgovski potnik mora obiskati določeno množico mest tako, da bo pri tem prehodil čim krajšo pot in se vrniti v izhodišče.

Pri programiranju in pisanju algoritma smo se odločili za programski jezik Python, saj imamo v njem največ znanja. Pri projektu pa smo spoznali še knjižnico *matplotlib* za delo z grafi.

3.1. **Osnovne funkcije.**

Že dobro poznan problem potujočega potnika smo se odločili predstaviti z grafom v obliki $n \times n$ matrike cen povezav. Graf smo generirali tako, da smo elemente matrike, ki predstavljajo celoštevilske cene povezav, izbrali naključno. To je prikazano v spodnji funkciji *utezi*.

```
def utezi(n, maxCena):
    utezi = np.random.randint(0, maxCena, size=(n, n))
    np.fill_diagonal(utezi, 0)
    return(utezi)
```

Nato je bilo potrebno definirati *ciljno funkcijo* oz. *fitness function*, s pomočjo katere bomo ocenjevali primernost rešitev. V našem primeru je vrednost te funkcije za neko pot kar dolžina te poti.

```
def dolzinaPoti(pot, utezi):
    dolzina = 0
    for i in range(len(pot) - 1):
        dolzina += utezi.item((pot[i] - 1, pot[i+1] - 1))
    dolzina += utezi.item((pot[len(pot)-1] - 1, pot[0] - 1))
    return(dolzina)
```

Naključno smo ustvarili začetno populacijo izbrane velikosti. Vsak element populacije ali *kromosom* predstavlja neko pot, ki obišče vsa vozlišča grafa.

```
def populacija(popVelikost, utezi):
    pot = list(range(1, len(utezi) + 1))
    poti = {}

    for i in range(popVelikost):
        random.shuffle(pot)
        dolzina = dolzinaPoti(pot, utezi)
        poti[i+1] = [pot.copy(), dolzina]

    return(poti)
```

V nadaljevanju izberemo starše, katerih gene bomo uporabili za nastanek naslednje generacije - določimo paritveni bazen. Tega se bomo lotili postopoma in sicer izbiramo po 2 starša za 2 otroka. Za selekcijo imamo dve možnosti, in sicer selekcijo s turnirjem ter proporcionalno selekcijo. Odločili smo se za turnirsko. Vsakega starša bomo izbrali tako, da bomo iz trenutne populacije naključno izbrali k kromosomov oz. poti. Naša funkcija oz. *selekcija* nam bo vrnila zmagovalca izmed teh k poti oziroma pot z najkrajšo dolžino. Torej bomo za n staršev v bazenu imeli n turnirjev.

```
def turnir(populacija, kTurnir):
    vozlisca = list(range(1, len(populacija) + 1))
    random.shuffle(vozlisca)
    izbranci = vozlisca[:kTurnir]
    minimum = math.inf
    pot = []

    for i in izbranci:
        if populacija[i][1] < minimum:
            minimum = populacija[i][1]
            pot = populacija[i][0]

    return(pot)
```

3.2. **Križanja oziroma crossoverji.**

Za ustvarjanje otrok bomo uporabili različna križanja oz. *crossoverje* staršev (poti) iz paritvenega bazena. Pri tem smo uporabili različne variacije križanj:

3.2.1. *Urejeno križanje ali ordered crossover.*

V urejenem križanju ali OX vsak starš predstavlja neko zaporedje vseh vozlišč. Na začetku oba starša razdelimo na tri podzaporedja, kjer 1. podzaporedje predstavlja vozlišča od prvega do vključno tistega na a-tem mestu, 2. podzaporedje poteka od vozlišča na a-tem do vključno vozlišča na b-tem mestu, 3. podzaporedje pa predstavlja preostanek osnovnega zaporedja. Prvi potomec ima kopirano 2. podzaporedje prvega starša na enaki poziciji. Nato od b+1 mesta naprej nadaljujemo z vozlišči drugega starša, ki jih dopolnjujemo (v primeru da je to vozlišče že vsebovano v zaporedju otroka, ga preskočimo) najprej iz 3. podzaporedja, nato 1. podzaporedja in nazadnje še iz 2. podzaporedja. Ko se zaporedje konča, skočimo na začetek in postopek nadaljujemo vse do začetka a-tega mesta. Na enak način tvorimo drugega otroka, le da vlogi staršev zamenjamo.

```
def OX(stars1, stars2):
    dolzina = len(stars1)

    rez_a = random.randint(1,len(stars1))
    rez_b = random.randint(rez_a,len(stars1)) #funkcija vzame rez_a in rez_b v odnosu rez_a <= rez_b

    ostanek1 = stars2[rez_b:] + stars2[:rez_b]
    for vozl in stars1[rez_a:rez_b]:
        ostanek1.remove(vozl)
    otrok1 = ostanek1[dolzina - rez_b:] + stars1[rez_a:rez_b] + ostanek1[:dolzina - rez_b]

    ostanek2 = stars1[rez_b:] + stars1[:rez_b]
    for vozl in stars2[rez_a:rez_b]:
        ostanek2.remove(vozl)
    otrok2 = ostanek2[dolzina - rez_b:] + stars2[rez_a:rez_b] + ostanek2[:dolzina - rez_b]

    return(otrok1, otrok2)
```

3.2.2. *Delno mapirano križanje ali partially mapped crossover.*

Drugi način križanja je delno mapirano križanje ali PMX, v katerem starša spet razdelimo na tri podzaporedja. Prvemu otroku prav tako kot v OX, prepišemo vrednosti prvega starša med obema rezoma, se pravi 2. podzaporedje. Nato vsako vozlišče(vsaka vrednost i) iz 2. podzaporedja drugega starša, ki še ni vsebovano v zaporedju otroka, dobi svoj istoležeči par v prvem staršu. Če je ta par že vsebovan v 2. podzaporedju drugega starša, temu vozlišču ponovno poiščemo par iz prvega starša in to počnemo dokler dobljeni istoležeči par v drugem staršu ne leži izven 2. podzaporedja. Takrat na njegovo mesto (v drugem staršu) zapišemo vrednost i. Na koncu postopka vsa prazna mesta v otroku zapolnimo z istoležečimi vozlišči iz drugega starša.

```
def PMX(stars1, stars2):
        l=len(stars1)
        rez_c = random.randint(1,len(stars1))
        rez_d = random.randint(rez_c,len(stars1)) #funkcija vzame rez_c in rez_d v odnosu rez_c <= rez_d

        izrez1 = stars1[rez_c:rez_d]
        izrez2 = stars2[rez_c:rez_d]
        otrok1 = [0]*l
        otrok2 = [0]*l
        otrok1[rez_c:rez_d]=izrez1
        otrok2[rez_c:rez_d]=izrez2
        u=0
        v=0
        for i in izrez2:
                if i not in izrez1:
                        u=stars2.index(i)
                        par_u=stars1[u]
                        while par_u in izrez2:
                                v=stars2.index(par_u)
                                par_u=stars1[v]
                        mesto=stars2.index(par_u)
                        otrok1[mesto]=i
        for j in range(l):
                if otrok1[j]==0:
                        otrok1[j]=stars2[j]
        for i in izrez1:
                if i not in izrez2:
                        u=stars1.index(i)
                        par_u=stars2[u]
                        while par_u in izrez1:
                                v=stars1.index(par_u)
                                par_u=stars2[v]
                        mesto=stars1.index(par_u)
                        otrok2[mesto]=i
        for j in range(l):
                if otrok2[j]==0:
                        otrok2[j]=stars1[j]

        return(otrok1, otrok2)
```

3.2.3. *Ciklično križanje ali cycle crossover.*

Zadnji način križanja, ki smo ga uporabili je ciklično križanje ali CX. Tu naša funkcija sprejme dva starša, iz katerih naredi slovar. Pri tem prvi starš predstavlja ključ, drugi pa vrednost. Najprej poiščemo vse cikle med staršema in jih shranimo v množico A. Prvega otroka tvorimo tako, da mu po vrsti dodamo vsak sodi cikel iz prvega starša in vsak lihi cikel iz drugega starša. Pri drugem otroku ravnamo ravno obratno.

```
def CX(stars1, stars2):
        slovar={key:value for key, value in zip(list(stars1), list(stars2))}
        l=len(stars1)
        otrok1=[0]*l
        otrok2=[0]*l
        cikli=[]
        cikel=[]
        A=[]
        for i in slovar.keys():
                if i not in cikli:
                        cikel=[i]
                        naslednji=slovar[i]
                        while naslednji not in cikel:
                                cikel.append(naslednji)
                                naslednji=slovar[naslednji]
                        A = A + [cikel]
                        cikli = cikli + cikel
        for C in A:
                for j in C:
                        u=stars1.index(j)
                        if A.index(C) % 2==0:
                                otrok1[u]=j
                                otrok2[u]=stars2[u]
                        else:
                                otrok2[u]=j
                                otrok1[u]=stars2[u]
        return(otrok1, otrok2)
```

3.2.4. *Naključno izbrano križanje.*

Da bi bili rezultati še bolj zanimivi, smo napisali dodatno funkcijo *nakljucno*, ki naključno izbere eno od križanj.

```
def nakljucno(stars1, stars2):
    R = [OX, PMX, CX]
    return random.choice(R)(stars1, stars2)
```

3.3. **Mutacije.**

S križanji smo dobili otroke, ki so neke nove poti ustvarjene iz dveh staršev. Da pa ohranjamo diverziteto v populaciji, je potrebno nekatere poti mutirati in sicer s *SWAP mutacijo* (zamenjali bomo dve vozlišči v poti). Vsako vozlišče poti z neko verjetnostjo mutiramo, torej zamenjamo položaj mutiranega vozlišča z nekim naključnim vozliščem te poti.

```
def mutacija(otrok, verjMutacije):
    for i in range(len(otrok)):
        if random.random() <= verjMutacije:
            j = random.randint(0, len(otrok) - 1)
            otrok[i], otrok[j] = otrok[j], otrok[i]
    return(otrok)
```

Ker pa lahko potomce dobivamo na več načinov, se pravi z različnimi križanji smo za tvorbo potomcev napisali posebno funkcijo, ki sprejme več parametrov, med njimi tudi verjetnost mutacije in način križanja.

```
def potomci(utezi, populacija, verjMutacije, kTurnir, crossover):
    potomci = {}
    for i in range(int(len(populacija)/2)):
        stars1 = turnir(populacija, kTurnir)
        stars2 = turnir(populacija, kTurnir)

        otroka = crossover(stars1, stars2)
        otrok1 = mutacija(otroka[0], verjMutacije)
        otrok2 = mutacija(otroka[1], verjMutacije)

        potomci[2*i+1] = otrok1, dolzinaPoti(otrok1, utezi)
        potomci[2*(i+1)] = otrok2, dolzinaPoti(otrok2, utezi)
    return(potomci)
```

Cilj našega genetskega algoritma je najti najkrajšo pot, ki nam jo izračuna naslednja funkcija.

```
def najkrajsaPot(nasledniki):
    minimum = math.inf
    pot = []
    for i in nasledniki:
        if nasledniki[i][1] < minimum:
            minimum = nasledniki[i][1]
            pot = nasledniki[i][0]

    return(pot)
```

Po pridobitvi vseh potrebnih podatkov, lahko na tej točki poženemo naš genetski algoritem, ki nam izmed vseh poti iz dane generacije vrne najboljšo se pravi najkrajšo.

```
def gaTsp(stGeneracij, utezi, populacija, verjMutacije, kTurnir, crossover):
    nasledniki = populacija
    for _ in range(stGeneracij):
        nasledniki = potomci(utezi, nasledniki, verjMutacije, kTurnir, crossover)
    return(najkrajsaPot(nasledniki))
```

3.4. **Primeri in spreminjanje parametrov.**

Za boljšo predstavo o delovanju našega algoritma smo na internetu poiskali nekaj že rešenih primerov in jih preizkusili tudi na našem genetskem algoritmu ter nato rezultate med sabo primerjali. Pri tem smo si za boljšo predstavo risali grafe s funkcijo *narisi* ter *gaTspGraf*.

```
def narisi(pot, lokacije):
    plt.figure()
    for stMest in range(len(pot)):
        if stMest != 0:
            sedanje = pot[stMest]
            a2, b2 = a1, b1
            a1, b1 = lokacije[sedanje-1][0], lokacije[sedanje-1][1]
            plt.plot([a1, a2], [b1, b2])
        else:
            prejsnje = pot[len(pot)-1]
            sedanje = pot[0]
            a2, b2 = lokacije[prejsnje-1][0], lokacije[prejsnje-1][1]
            a1, b1 = lokacije[sedanje-1][0], lokacije[sedanje-1][1]
            plt.plot([a1, a2], [b1, b2])
        plt.scatter(lokacije[sedanje-1][0], lokacije[sedanje-1][1])
    plt.show()
```

```
def gaTspGraf(stGeneracij, utezi, populacija, verjMutacije, kTurnir, crossover, kGraf, koordinate):
    nasledniki = populacija
    for i in range(stGeneracij):
        nasledniki = potomci(utezi, nasledniki, verjMutacije, kTurnir, crossover)

        if (i+1)%kGraf == 0:
            plt.figure()
            pot = najkrajsaPot(nasledniki)
            narisi(pot, koordinate)

    return(najkrajsaPot(nasledniki))
```

Pri naših primerjavah nas je zanimala povprečna dolžina najkrajše poti, prav tako pa nam pomemben podatek predstavlja tudi najkrajša pot v vseh ponovitvah. Funkcija *povprecje* nam je pri danih ponovitvah genetskega algoritma izračunala povprečno dolžino najkrajše poti, ter hkrati tudi vrnila najkrajšo pot in njeno dolžino.

```
def povprecje(ponovitve, stGeneracij, utezi, populacija, verjMutacije, kTurnir, crossover):
    vsota = 0
    najkrajsaPot = []
    najkrajsaDolzina = math.inf

    for _ in range(ponovitve):
        trenutnaPot = gaTsp(stGeneracij, utezi, populacija, verjMutacije, kTurnir, crossover)
        dolzinaTrenutne = dolzinaPoti(trenutnaPot, utezi)
        vsota += dolzinaTrenutne

        if dolzinaTrenutne < najkrajsaDolzina:
            najkrajsaDolzina = dolzinaTrenutne
            najkrajsaPot = trenutnaPot

    return([vsota/ponovitve, [najkrajsaPot, najkrajsaDolzina]])
```

4. Zaključek

V naši analizi smo se osredotočili na tri primere, katerih podatke in približke rešitev smo dobili na internetu. Nato smo rešitve za dane probleme, dobljene z našim genetskim algoritmom primerjali s tistimi z interneta. Omejili smo se na tri primere in sicer ulysses22, berlin52 in kroal100. Pri primeru berlin52 smo naredili veliko primerjav, da smo dobili občutek kateri parametri res vplivajo na kvaliteto rešitve. To smo potem primerjali in potrdili še na drugih dveh primerih.

Tekom analize smo spreminjali sledeče parametre:

- križanje (CX, PMX in OX)
- velikost populacije (20, 100)
- število generacij (100, 1000)
- verjetnost mutacije (0%, 0.5% in 4%)

Ohranjali pa smo število ponovitev 20 ter število kromosomov v turnirju nastavili na 5.

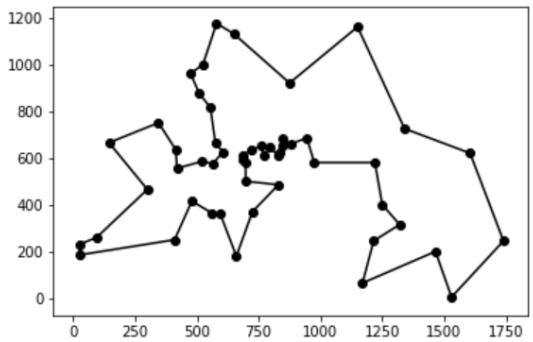
Pri primeru kroa100 je bila najbolj opazna sprememba ob večanju velikosti populacije. Dolžina najkrajše poti se je dokaj hitro bližala optimumu, kljub temu da so bile začetne vrednosti več kot dvakrat večje od optimalne vrednosti. Prav tako je bilo možno opaziti veliko spremembo ob povečanju števila generacij iz 100 na 1000, saj se je ob tem najkrajša pot za približno dvakrat zmanjšala.

Prav tako smo tudi pri primeru ulysses22 zasledili podobno gibanje rešitev, le da je bila tu konvergenca k optimumu precej hitrejša, saj sprememba števila generacij ne poveča vrednosti v tolikšni meri kot prej. Oba primera smo tu preizkušali le na urejenem križanju, med tem ko smo pri primeru berlin52 preizkusili čisto vse kombinacije.

```
In [25]: data = "berlin52.txt"
         dataPot = "berlin52opt.txt"

         lokacije = pr.preberi(data)
         razdalje = pr.mesta(lokacije)
         najkrajsa = pr.najkrajsaConcord(dataPot)

In [26]: pr.narisi(najkrajsa, lokacije)
```

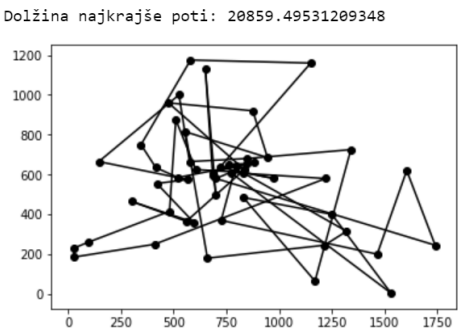



```
In [27]: pr.dolzinaPoti(najkrajsa, razdalje)
Out[27]: 7544.365901904087
```

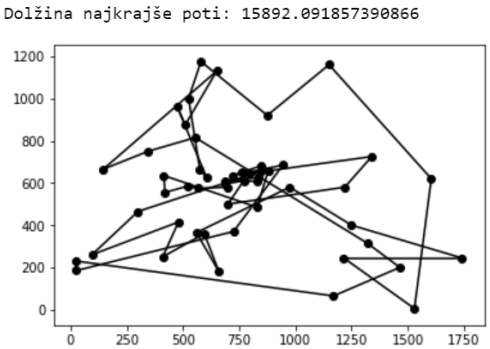
Začeli smo z cikličnim križanjem, katerega konvergenca je na začetku se pravi pri majhni populaciji, majhnem številu generacij in brez mutacije najbolj počasna od vseh treh križanj. Vrednosti naše rešitve so tudi trikratnik optimalne. Ko smo povečali velikost populacije se je rešitev rahlo izboljšala, vendar še vedno ne v meri, ki bi si jo želeli. Prav tako če smo povečali le število generacij. Ob povečanju obeh parametrov pa je bil približek rešitve že soliden. Nato smo povečali verjetnost mutacije iz 0 na 0.5 procenta in približek se je zelo izboljšal. Če smo ob tem še povečali populacijo in število generacij je bila vrednost rešitve najboljša do sedaj. Ko pa smo verjetnost mutacije povečali na 4 procente, so se rezultati začeli slabšati. Tudi ob povečanju števila generacij je dobljena dolžina poti zelo daleč od optimalne. S tem smo prišli do dejstva, da previsoka verjetnost mutacije negativno vpliva na rezultate.

Pri delno mapiranem križanju je začetna vrednost malenkost boljša od tiste pri cikličnem križanju. Povečanje populacije v tem primeru porodi bistveno boljše rešitve kot pri enakem povečanju populacije v cikličnem križanju. Opazili smo tudi, da povečanje števila generacij tu ne vpliva bistveno na izboljšanje rešitev. Pri določenih primerih lahko konvergenco celo upočasni. Povečanje verjetnosti mutacije na pol procenta tudi tu bistveno izboljša rezultat. V kombinaciji z veliko populacijo je približek poti vedno bližje pravemu. Iz primerov vidimo, da večanje števila generacij najmanj pripomore k izboljšavi rešitve. Pri visoki verjetnosti mutacije vsi rezultati niso tako dobri, kar nam pove kako pomemben je ta dejavnik. Visoka verjetnost bolj negativno vpliva na PMX kot na CX, vendar razlike niso velike.

Spodaj lahko vidimo primer izboljšanja rešitve ob povečanju populacije pri metodi PMX, verjetnosti mutacije 0 ter 100 generacijah.

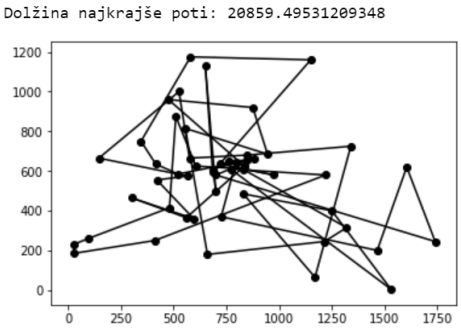


Slika 1. Populacija 20

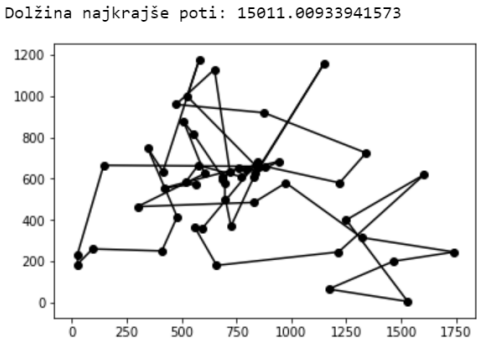


Slika 2. Populacija 100

Naslednja slika pa nam pokaže lepo primerjavo s prejšnjo, kako povečanje populacije veliko bolj vpliva na izboljšanje rešitve, kakor povečanje števila generacij. Tu so enaki pogoji kot prej, le da je stalna velikost populacije 20, spreminja pa se število generacij.



Slika 3. Število generacij 100

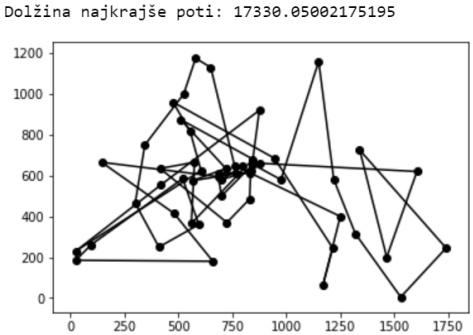


Slika 4. Število generacij 1000

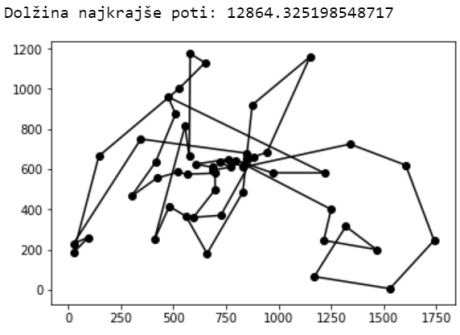
Najboljši začetni približek nam da urejeno križanje. Pri tem povečanje populacije znatno vpliva na rezultat, saj rešitev postane skoraj dvakrat boljša, medtem ko povečanje števila generacij spet pokvari rezultat. Z malo večjo verjetnostjo mutacije se rešitev najbolj približa optimalni od vseh obravnavanih primerov, vendar pa visoka mutacija (4%) na OX vpliva najslabše od vseh treh križanj.

Opazimo, da se z večanjem velikosti populacije, deloma pa tudi števila generacij, rešitve izboljšujejo. Seveda pa ne smemo zanemariti dejstva, da z večanjem teh parametrov povečujemo tudi čas, ki ga algoritem potrebuje za izračun rešitve. Glede na informacije o genetskih algoritmih smo zasledili, da naj bi se verjetnost mutacije gibala od 0.5 do 2 procenta. Verjetnost mutacije pri vrednosti 0.5% nam v analizi dejansko da najbolše rešitve. Vidimo tudi, da je verjetnost mutacije (za vse tri metode) pri vrednosti 0.005 precej blizu optimalni, saj se že pri 0.01 in 0.02 dolžine poti (npr. pri OX) zopet večajo proti 12000. Se pravi lahko pri vseh primerih zasledimo, da preveliko povečanje verjetnosti mutacije negativno vpliva na približek rešitve. Mutacija, ki je seveda zelo pomembna za ohranjanje raznolikosti rešitev, torej ne sme zajeti prevelikega deleža populacije, saj se s tem izgubljajo naše že zgrajene rešitve.

Spodnji dve sliki prikazujeta približek rešitve brez mutacije in pri mutaciji 0.5%. Tu je jasno razvidno, kako pravilna vrednost verjetnosti mutacije vpliva na boljšo konvergenco.



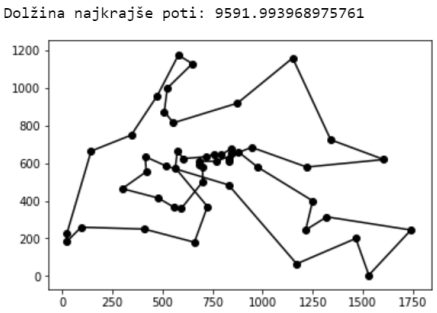
Slika 5. Verj. mutacije 0%



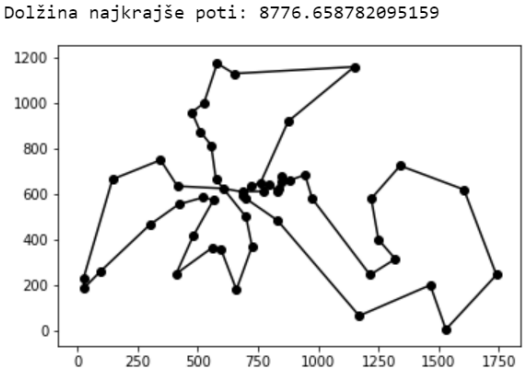
Slika 6. Verj. mutacije 0.5%

Če primerjamo izboljševanje rezultatov pri večanju populacije ali pa števila generacij opazimo, da na primer (pri OX) pri številu generacij 100 in velikosti populacije 50 dobimo rezultate okoli 12000, če desetkrat povečamo število generacij so najkrajše poti okoli 10000, pri desetkratni povečavi populacije pa okoli 8600. Torej je povečevanje populacije večjega pomena za konvergenco k optimalni rešitvi, kot povečava števila generacij.

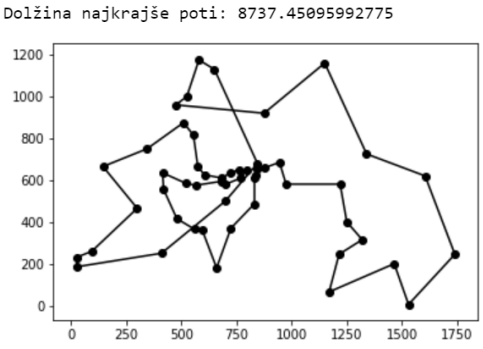
V splošnem je najhitrejša metoda z uporabo urejenega križanja ali OX. Na začetku so njeni rezultati sicer dokaj slabi, a jih lahko zelo hitro izboljšamo s povečanjem populacije in tako opazimo visoko hitrost konvergence. Zgoraj lahko vidimo primerjavo vseh treh križanj, pri velikosti populacije 100, 1000 generacijah in verjetnosti mutacije 0.5%.



Slika 7. CX



Slika 8. PMX



Slika 9. OX

Na koncu je potrebno še omeniti, da se ob večanju problema (števila mest) rešitve vedno bolj oddaljujejo od optimuma. Pri problemu ulysses22, ki je še dokaj majhen, se naša rešitev zelo približa optimalni. Na drugi strani pa imamo problem kroa100, ki je približno 4 krat večji. Tu se naše rešitve že krepko razlikujejo od optimuma.

Literatura

[1] *Genetic algorithm*, v: Wikipedia: The Free Encyclopedia, [ogled 13. 12. 201], dostopno na https://en.wikipedia.org/wiki/Genetic_algorithm.

[2] N. Kumar, Karambir in R. Kumar, *A Comarative Analysis of PMX, CX and OX Crossover operators for solving Travelling Salesman Problem*, [ogled 13. 12. 2018], dostopno na http://www.mnkjournals.com/ijlrst_files/Download/Vol%201%20Issue%202/303-%20Naveen.pdf.

[3] *The TSPLIB Symmetric Traveling Salesman Problem Instances*, [ogled 28. 12. 2018], dostopno na http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html?fbclid=IwAR2VkUJ80JoqQbLKAyGlriphhbhYjwmE7f_gH1P5K6a0asRisrHoW_zKr48