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. *traveling 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 , 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(n, maxCena)*.

```
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 ciklje med staršema in jih shranimo v množico A. Prvega otroka tvorimo tako, da mu po vrsti dodamo vsak sodi cikel iz prvega stašra 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)
```

3.3. **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))
```

4. Zaključek

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.