

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Klemen Hovnik
Matija Gubanec Hančič
Jan Rudof

**Predpisana drevesa z najmanjšim/največjim Wienerjevim
indeksom**

Projekt v povezavi z OR

Ljubljana, 2018

KAZALO

1. Navodilo	3
2. Uvod	3
3. Opis dela	3
3.1. Enostaven algoritem	3
3.2. Genetski algoritem	5
4. Zaključek	8

1. NAVODILO

We want to analyze the structure of trees on a fixed number of vertices n and fixed maximum degree Δ that have Wiener index (i.e. total distance) as small as possible. Similarly, we want to find the structure of trees on a fixed number of vertices n with fixed diameter d that have Wiener index (i.e. total distance) as large as possible. In order to get the answer for very small values of n first, apply an exhaustive search, and next, for larger n , apply a genetic algorithm or any other metaheuristic. Verify for how large n your exhaustive search and your genetic algorithm implementations are efficient.

2. UVOD

Naj bo $G = (V(G), E(G))$ enostaven povezan neusmerjen graf. *Wienerjev indeks* (oziroma *Wienerjevo število* $W(G)$) je definiran kot

$$(1) \quad W(G) = \frac{1}{2} \sum_{u \in V(G)} \sum_{v \in V(G)} d_G(u, v).$$

Tukaj označimo z $d_G(u, v)$ razdaljo med vozliščem u in v v grafu G .

Naša naloga je, da analiziramo lastnosti dreves z določenim številom vozlišč in fiksno maksimalno stopnjo vozlišč, ki imajo najmanjši Wienerjev indeks. Podobno nas zanimajo tudi lastnosti dreves na določenem številu vozlišč s fiksnim premerom, ki imajo največji možni Wienerjev indeks.

Za izvedbo projekta smo si izbrali programski jezik *Sage*, saj ta že vsebuje orodja za delo z grafi, prav tako pa ima tudi generator dreves in že vgrajeno funkcijo za izračun Wienerjevega indeksa.

3. OPIS DELA

Najprej smo se lotili izračuna Wienerjevih indeksov na preprostih grafih z malo vozlišči, da vidimo, kako naj bi ta struktura grafov z minimalnimi oziroma maksimalnimi indeksi izgledala v splošnem.

3.1. Enostaven algoritem.

3.1.1. Maksimalen Wiener index na drevesih s fiksnim premerom.

Definirali smo funkcijo $drevesa(n)$, ki nam izpiše seznam vseh dreves s številom vozlišč n . Potem smo to funkcijo uporabili v funkciji $drevesa_premer(n)$, ki nam iz prejšnjih dreves generira slovar, kjer so ključi možni premeri naših dreves, vrednosti ključev pa so pripadajoča drevesa. Tako smo si pripravili podlago za enostaven algoritem iskanja maksimalnega Wiener indexa za drevesa z določenim premerom. Sestavili smo funkcijo $max_Weinerindex(n, N)$, kjer je N fiksni premer. Ta funkcija nam je za vsa drevesa s številom vozlišč n in premerom N izpisala maksimalni Wiener index in nam drevo, kjer je ta indeks dosežen tudi izrisala.

```
def drevesa(n):
    t = graphs.trees(n)
    T= next(t)
    k= []
    k.append(T.edges())
    for T in t:
        m = T.edges()
        k.append(m)
    return k
```

```
def drevesa_premier(n):
    L = []
    k = drevesa(n)
    for i in range(len(k)):
        premier1 = Graph(k[i]).diameter()
        L.append([premier1, k[i]])
    from collections import defaultdict
    d1 = defaultdict(list)
    for l, v in L:
        d1[l].append(v)
    d = dict((l, tuple(v)) for l, v in d1.iteritems())
    return d
```

```
def max_Wienerindex(n,N):
    d = drevesa_premier(n)
    x = []
    for i in range(len(d[N])):
        x.append(Graph(d[N][i]).wiener_index())
    print 'drevesa, na ', n, 'vozliščih, s premerom', N, 'imajo maksimalni Wiener index:', max(x)
    for i in range(len(x)):
        if x[i] == max(x):
            Graph(d[N][i]).show()
```

3.1.2. Minimalen Wiener index na drevesih s fiksno stopnjo.

Za iskanje najmanjših Wienerjevih indeksov pri določenem številu vozlišč n in pri fiksni maksimalni stopnji m smo definirali funkcijo *fiksna_stopnja*, ki nam je iz seznama, ki ga vrne funkcija *drevesa(n)* izpisala drevesa z maksimalno stopnjo m . Izmed teh optimalnih dreves pa smo potem s funkcijo *.wiener_index()* izračunali najmanjši Wiener index, ter drevo s tem indeksom tudi izrisali.

```
def fiksna_stopnja(n,m):
    k = drevesa(n)
    zaporedje = []
    optimalna_drevesa = []
    index = []
    for j in range(len(k)):
        zaporedje.append(Graph(k[j]).degree_sequence())

    for l in range(len(zaporedje)):
        T = Graph(k[l])
        if max(zaporedje[l]) == m:
            optimalna_drevesa.append(k[l])
            index.append(T.wiener_index())
    u = min(index)
    pozicija = [i for i, j in enumerate(index) if j == u]
    print 'Minimalni Wiener index s fiksno stopnjo', m, 'je', u
    return Graph(optimalna_drevesa[pozicija[0]]).plot()
```

Hitro smo prišli do ugotovitve, da je naš algoritem za izračun indeksov časovno prepotraten. Zato smo se problema iskanja Wienerjevih indeksov lotili na drugačen način. In sicer z *genetskim algoritmom* katerega ideja je, da ustvarimo populacijo začetnih osebkov, ki jih potem kombiniramo s križanjem in mutacijami, da prihajamo do vedno boljših rezultatov.

3.2. Genetski algoritem.

Genetski algoritem je metahevrstika, 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 optimalne rešitve. Vsak kandidat ima določene lastnosti, ki jih 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 in tako dalje. Na koncu se algoritem zaključi, ko doseže maksimalno število iteracijskih korakov oziroma, ko dobi najboljši približek optimalni rešitvi.

Naša začetna množica kandidatov bodo drevesa na n vozliščih, ki sta jim skupna ali premer ali pa največja stopnja. Nato smo ustvarili genetski algoritem, ki iz začetne množice dreves generira naslednjo generacijo dreves. Ta postopek nato iterativno nadaljujemo (in pri tem selekcioniramo iz novo nastalih dreves le najboljše za naslednje korake), dokler ne bomo prišli do optimalnih dreves za določeno število vozlišč. Pri tem bomo morali paziti, da se bo ohranjala maksimalna stopnja vozlišč, oziroma v drugem primeru, premer.

3.2.1. *Genetski algoritem za drevesa z največjo stopnjo.*

Pri tem algoritmu smo iskali grafe z določenim številom vozlišč in določeno največjo stopnjo vozlišč z najmanjšim Wienerjevim indeksom. Najprej definiramo funkcijo, ki zgenerira graf na $st_vozlisc$ vozliščih z največjo stopnjo $max_stopnja$. Pri preučevanju karakteristik optimalnih grafov smo ugotovili, da imajo najmanjše Wienerjeve indekse ravno zvezde. Zato smo funkcijo spisali tako, da najprej vzame graf zvezde in nato naključno dodaja povezave, medtem ko pazi, da katero vozlišče ne bi preseglo največje dovoljene stopnje. S pomočjo generatorja dreves nato ustvarimo začetno populacijo osebkov.

```
def nakljucni_graf(st_vozlisc, max_stopnja): #ustvarjamo naključne grafe z določenim številom vozlišč in max stopnjo
    zvezda = graphs.StarGraph(max_stopnja)
    i = max_stopnja
    while i < (st_vozlisc - 1):
        zvezda1 = Graph(zvezda)
        zvezda1.add_edge(random.randint(1, i), i + 1)
        if zvezda1.degree_sequence()[0] <= max_stopnja:
            zvezda = Graph(zvezda1)
            i = i + 1
    return zvezda

def zacetna_populacija(st_vozlisc, max_stopnja, stevilo_osebkov): #ustvari zacetno populacijo, ki jo bomo potem razvijali
    zacetna_populacija = []
    i = 0
    while i < stevilo_osebkov:
        osebek = nakljucni_graf(st_vozlisc, max_stopnja)
        zacetna_populacija.append(osebek.to_dictionary())
        i = i + 1
    return zacetna_populacija
```

Potem definiramo še funkcijo *fitness*, ki med danim seznamom dreves poišče tistega, ki ima najboljšo iskano lastnost. V našem primeru recimo izmed izbranih dreves poišče tistega z minimalnim Wienerjevim indeksom. Naslednja funkcija, ki jo definiramo je funkcija *mutate*. S to funkcijo v algoritmu vpeljemo še možnost mutacije, kjer se v primeru, da do mutacije pride, eden od listov drevesa naključno prestavi na neko drugo mesto.

```
def fitness(seznam_dreves): #iscemo iskano optimalno lastnost, se pravi minimalen Wienerjev indeks
    index = 1000000000 #neko veliko število, da bojo naslednji indeksi zihr manjši
    for drevo in seznam_dreves:
        graf_drevesa = Graph(drevo)
        if graf_drevesa.wiener_index() < index:
            index = graf_drevesa.wiener_index()
            drevo_min_index = drevo
    return drevo_min_index

def mutate(drevo, verjetnost): #mutacija grafa (odstranimo listek in ga pripnemo drugam)
    kopija_drevo = Graph(drevo)
    listki = []
    if random.random() <= verjetnost:
        for i in range(0, len(kopija_drevo.degree())): #iščemo vse listke v grafu, katere lahko odstranimo
            if kopija_drevo.degree()[i] == 1:
                listki.append(i)
        izbrano_vozlisce_listka = random.choice(listki)
        kopija_drevo.delete_edge(izbrano_vozlisce_listka, kopija_drevo.neighbors(izbrano_vozlisce_listka)[0])
        vozlisca_drevo = kopija_drevo.vertices()
        del vozlisca_drevo[izbrano_vozlisce_listka] #treba je izločiti vozlišče, ki ga pripnemo
        kopija_drevo.add_edge(izbrano_vozlisce_listka, random.choice(vozlisca_drevo)) #pripnemo listek drugemu vozlišču
        if kopija_drevo.degree_sequence()[0] == Graph(drevo).degree_sequence()[0]:
            return kopija_drevo.to_dictionary()
        else:
            return mutate(drevo, verjetnost)
    else:
        return drevo
```

Predzadnja funkcija, ki smo jo napisali, je funkcija *crossover*. Ta iz dveh podanih dreves s križanjem sestavi dve novi drevesi. To stori tako, da naključno v teh drevesih izbere vozlišči in njuna soseda, nato povezavo med izbranim vozliščem in sosedom prekine ter navzkrižno poveže del prvega grafa z delom drugega in obratno. Nato preveri, ali karakteristike dobljenih dreves ustrezajo iskanim. V primeru, da ustrezajo, vrne dobljena grafa, v nasprotnem primeru pa križanje izvaja toliko časa, dokler ne dobi dveh ustreznih grafov.

Zadnja funkcija, ki smo jo spisali, je funkcija *nova_generacija*. Parametra, ki ju podamo tej funkciji sta *zacetna_generacija* in *verjetnost*. S pomočjo te funkcije iz začetne generacije z mutacijami in križanjem dreves ustvarimo novo generacijo. Iz začetne generacije naključno vzamemo dva osebk, ju podvržemo možnosti mutacije in nato prekrižamo. Nato izmed dobljenih dreves in njunih staršev izberemo najboljša dva, ki ju pošljemo v naslednjo generacijo. Na ta način se najboljša drevesa

```

def crossover(drevo1, drevo2): #križanje dveh grafov, kjer dobimo ven dva nova (za naslednjo generacijo vzamemo le optimalnega)
    kopija_drevo1 = Graph(drevo1)
    kopija_drevo2 = Graph(drevo2)

    izbrano_vozlisce1 = random.randint(0, kopija_drevo1.order() - 1) #izberemo vozlišče kjer bomo razpolovili graf
    izbrano_vozlisce2 = random.randint(0, kopija_drevo2.order() - 1)
    sosed_vozlisca1 = random.choice(kopija_drevo1.neighbors(izbrano_vozlisce1)) #izberemo soseda vozlišca, kjer bomo razpolovili graf
    sosed_vozlisca2 = random.choice(kopija_drevo2.neighbors(izbrano_vozlisce2))

    kopija_drevo1.delete_edge(izbrano_vozlisce1, sosed_vozlisca1) #naključno odstranimo povezavo
    kopija_drevo2.delete_edge(izbrano_vozlisce2, sosed_vozlisca2)

    kopija_drevo1_a = kopija_drevo1.subgraph(kopija_drevo1.connected_component_containing_vertex(izbrano_vozlisce1)) #naredimo podgrafe
    kopija_drevo1_b = kopija_drevo1.subgraph(kopija_drevo1.connected_component_containing_vertex(sosed_vozlisca1))
    kopija_drevo2_a = kopija_drevo2.subgraph(kopija_drevo2.connected_component_containing_vertex(izbrano_vozlisce2))
    kopija_drevo2_b = kopija_drevo2.subgraph(kopija_drevo2.connected_component_containing_vertex(sosed_vozlisca2))

    novo_drevo1 = kopija_drevo1_a.disjoint_union(kopija_drevo2_b) #povezemo dva podgrafa v nov graf
    novo_drevo1.add_edge((0, izbrano_vozlisce1), (1, sosed_vozlisca2))
    novo_drevo1.relabel()

    novo_drevo2 = kopija_drevo2_a.disjoint_union(kopija_drevo1_b) #povezemo dva podgrafa v nov graf
    novo_drevo2.add_edge((0, izbrano_vozlisce2), (1, sosed_vozlisca1))
    novo_drevo2.relabel()

    if novo_drevo1.degree_sequence()[0] == novo_drevo2.degree_sequence()[0] and novo_drevo1.degree_sequence()[0] ==
    Graph(drevo1).degree_sequence()[0] and novo_drevo1.order() == novo_drevo2.order():
        #preverimo, če se max stopnja in število vozlišč ohrani
        return [novo_drevo1.to_dictionary(), novo_drevo2.to_dictionary()]
    else:
        return crossover(drevo1, drevo2)

```

skoraj vedno prenesejo v naslednjo generacijo. Na koncu nam preostane še, da vse, kar smo do sedaj spisali uporabimo v simulaciji, ki ustvari začetno populacijo in nato *stevilo_generacij*-krat iz prejšnje generacije ustvari novo, kar bi v teoriji po dovolj velikem številu korakov moralo pripeljati do zelo dobrega približka optimalne rešitve.

```

def nova_generacija(zacetna_generacija, verjetnost): #definira novo generacijo osebkov
    nova_generacija = []
    i = 0
    while i < len(zacetna_generacija):

        drevo1 = random.choice(zacetna_generacija)
        drevo2 = random.choice(zacetna_generacija)
        mutirano_drevo1 = mutate(drevo1, verjetnost)
        mutirano_drevo2 = mutate(drevo2, verjetnost)
        novi_drevesi = crossover(mutirano_drevo1, mutirano_drevo2)
        novo_drevo1 = novi_drevesi[0]
        novo_drevo2 = novi_drevesi[1]
        optimalno_drevo = fitness([novo_drevo1, novo_drevo2, mutirano_drevo1, mutirano_drevo2, drevo1, drevo2])

        nova_generacija.append(optimalno_drevo)

        i = i + 1

    return nova_generacija

def simulacija(st_vozlisc, max_stopnja, stevilo_osebkov, stevilo_generacij, verjetnost): #požene program in išče graf z min wiener indeksom
    populacija = zacetna_populacija(st_vozlisc, max_stopnja, stevilo_osebkov)
    i = 0
    while i < stevilo_generacij:
        populacija = nova_generacija(populacija, verjetnost)
        i = i + 1
    return fitness(populacija), Graph(fitness(populacija)).wiener_index(), Graph(fitness(populacija)).show()

```

3.2.2. Genetski algoritem za drevesa s fiksnim premerom.

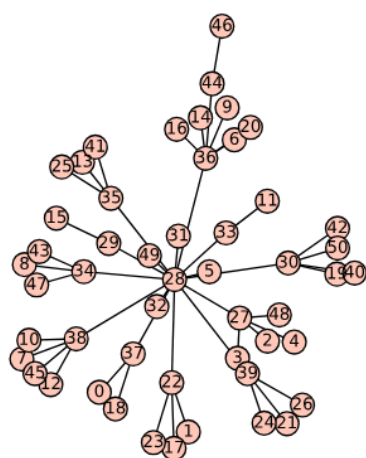
Pri drevesih s fiksnim premerom je princip precej podoben tistemu, ki ga uporabimo pri drevesih z določeno maksimalno stopnjo vozlišč. Le, da smo tukaj iskali maksimalen Wienerjev indeks. Uporabljene funkcije so podobne, razlikujejo se po tem, da se ne osredotočajo na maksimalno stopnjo, temveč preverjajo premer grafa. Poigrali smo se tudi z dodajanjem elitizma v funkcijo *nova_generacija*. Na ta način v naslednjo generacijo osebkov pošljemo nekaj teh iz začetne generacije z najboljšo lastnostjo in zagotovimo, da se najboljše drevo nikoli ne izgubi. Celoten algoritem si lahko pogledate na [github](#) repozitoriju.

4. ZAKLJUČEK

Funkciji, ki iščeta optimalna drevesa prek izčrpane metode, oziroma načina, da preverita vsa drevesa in vrneta tistega z iskano optimalno lastnostjo, naj bo to maksimalni al minimalni Wienerjev indeks, sta, kot že rečeno, zelo potratni in dobro delujeta le do okoli 18 vozlišč.

Nasprotno genetski algoritem za iskanje dreves z minimalnim Wienerjevim indeksom dela zelo dobro, testirali smo ga na drevesih s 50 vozlišči, z največjo stopnjo 15, velikostjo začetne populacije 100, številom generacij 100 in z verjetnostjo 0.05, da pride do mutacije. Rezultat vrne v malo manj kot dveh minutah, kar se nam zdi precej dobro.

116 simulacija(50, 15, 100, 100, 0.05)



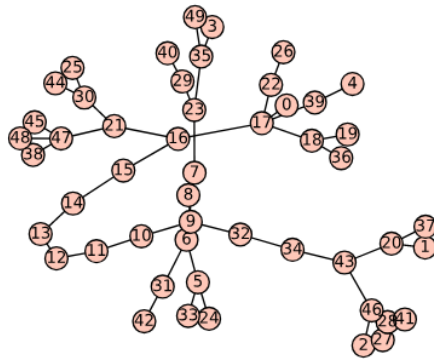
```
({0: [37], 1: [22], 2: [27], 3: [27], 4: [27], 5: [28], 6: [36], 7: [38], 8: [34], 9: [36], 10: [38], 11: [33], 12: [38], 13: [35], 14: [36], 15: [29], 16: [36], 17: [22], 18: [37], 19: [30], 20: [36], 21: [39], 22: [1, 17, 23, 28], 23: [22], 24: [39], 25: [35], 26: [39], 27: [48, 2, 3, 4, 28], 28: [32, 33, 49, 34, 35, 36, 5, 37, 22, 38, 39, 27, 29, 30, 31], 29: [28, 15], 30: [50, 19, 40, 42, 28], 31: [28], 32: [28], 33: [11, 28], 34: [8, 43, 28, 47], 35: [25, 41, 28, 13], 36: [16, 20, 6, 9, 28, 44, 14], 37: [0, 18, 28], 38: [7, 10, 12, 28, 45], 39: [21, 24, 26, 28], 40: [30], 41: [35], 42: [30], 43: [34], 44: [36, 46], 45: [38], 46: [44], 47: [34], 48: [27], 49: [28], 50: [30]}, 4124, None)
```

Za grafe, kjer smo iskali minimalni Wienerjev indeks, smo ugotovili, da to optimalno lastnost dosežejo bolj razvejani grafi. Že pri enostavnih algoritmihi smo ugotovili, da bi pri določitvi le števila vozlišč imeli najmanjše indekse pri grafih zvezdaste oblike. Ker pa smo morali fiksirati tudi maksimalno stopnjo, pa graf ne more biti vedno zvezda. V teh primerih smo opazili, da je graf zelo razvejan. Seveda pa je razvejanost odvisna od maksimalne stopnje vozlišč (pri višjih maksimalnih stopnjah so grafi veliko bolj razvejani).

Približno enako hitro deluje tudi genetski algoritem za iskanje največjega Wienerjevega indeksa. Za iskana drevesa smo ugotovili, da največje indekse dosežejo poti. Ker imamo podan največji dovoljeni premer, ne moremo vedno dobiti poti. Ima pa drevo z največjim Wienerjevim indeksom lastnost, da ni preveč razvejano, temveč osnovo grafa sestavlja pot, ki se nato razcepi, večinoma pa so vozlišča nabrana pri koncih osnovne poti.


```
140 1 simulacija(50,15,100,100,0.05)
```

```
141 ▾
```



```
({0: [17], 1: [20], 2: [46], 3: [35], 4: [39], 5: [33, 6, 24], 6: [5, 7, 31], 7: [6, 23, 8], 8: [7, 9], 9:
[32, 8, 10], 10: [9, 11], 11: [10, 12], 12: [11, 13], 13: [12, 14], 14: [13, 15], 15: [16, 14], 16: [17, 21,
15], 17: [16, 0, 18, 22, 39], 18: [17, 19, 36], 19: [18], 20: [1, 37, 43], 21: [16, 30, 47], 22: [17, 26],
23: [35, 7, 29], 24: [5], 25: [30], 26: [22], 27: [46], 28: [46], 29: [23, 40], 30: [21, 25, 44], 31: [6,
42], 32: [34, 9], 33: [5], 34: [32, 43], 35: [49, 3, 23], 36: [18], 37: [20], 38: [47], 39: [17, 4], 40:
[29], 41: [46], 42: [31], 43: [34, 20, 46], 44: [30], 45: [47], 46: [2, 41, 27, 43, 28], 47: [48, 21, 38,
45], 48: [47], 49: [35]}, 10227, None)
```