Nanotorusi

Tomas Rode
Enej Kovač

29. november 2019

1 Navodila

Nanotorus je 3-regularen graf na torusu. Vsak nanotorus lahko dobimo tako, da na šeskotni mreži enačimo nasprotni stranici danega paralelograma. Torej je vsak nanotorus določen z dvema vektorjema v ravnini, (k, l) in (m, n), za katera velja $k^2 + l^2 \neq 0$ in $m^2 + n^2 \neq 0$. Projekt je sestavljen iz štirih podnalog:

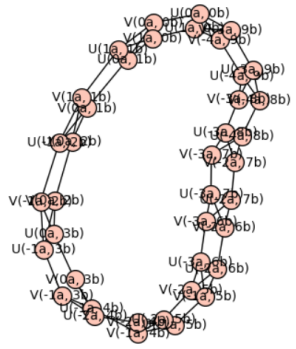
1. V prvem delu naloge ustvarite funkcijo, ki v *Sage* konstruira nanotorus, za dane k, l, m in n.
2. S pomočjo funkcij v *Sage* preučite nekaj lastnosti nanotorusov: za dan (k, l, m, n) določite število vozlišč, premer, tranzitivnost, ...
3. Za $v \in V(T)$, poljubno vozlišče nanotorusa, določite število vozlišč na razdaljah $i : \ 1 \leq i \leq \text{diam}(T)$. Poizkusite poiskati formulo za dane i, k, l, m, n.
4. Naj bo T nanotorus tipa (k, l, m, n). Ugotovite ali obstaja nanotorus tipa $(k', 0, m', n')$, izomorfen T. Če obstaja, ugotovite povezavo med (k, l, m, n) in (k', m', n').

2 Funkcija nanotorus

S pomočjo objektnega programiranja sva v *Sage* zapisala funkcijo, ki konstruira nanotorus s k, l, m in n. Na sponjih slikah lahko vidimo nekaj primerov.

```
G = nanotorus(2,0,-5,10)
print(G.is_vertex_transitive(), G.order(), G.size(), G.degree().count(3) == len(G.degree()), G.diameter())
#(tranzitivnost, št. vozlišč, št. povezav, vse točke reda 3, premer)
G.plot()
```

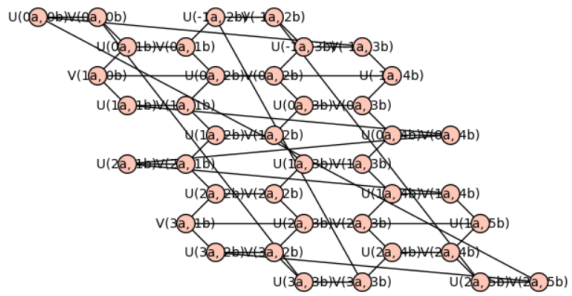
```
(True, 40, 60, True, 10)
```



Na prvi sliki sva dovolila, da *Sage* prerazporedi točke po prostotru tako, da se čim bolje vidi oblika grafa. Pri tem se nekoliko izgubi šestkotna mreža, na kateri smo graf ustvarili. Pri vsakem grafu sva preverila še tranzitivnost, število vozlišč in povezav, red vozlišč in premer grafa.

```
G = nanotorus(4,2,-2,4)
print(G.is_vertex_transitive(), G.order(), G.size(), G.degree().count(3) == len(G.degree()), G.diameter())
#(tranzitivnost, št. vozlišč, št. povezav, vse točke reda 3, premer)
G.plot()
```

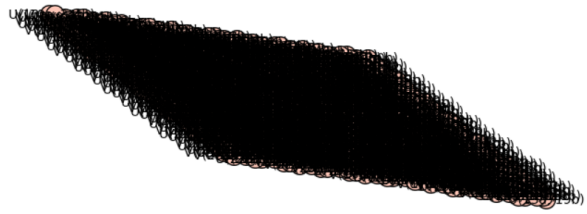
```
(True, 40, 60, True, 6)
```



Pri drugem grafu sva ohranila koordinate vozlišč na šestkotni mreži. Izmed vseh grafov je torej tukaj izvorna šestkotna mreža najbolj razvidna.

```
G = nanotorus(10,-35,13,20)
print(G.is_vertex_transitive(), G.order(), G.size(), G.degree().count(3) == len(G.degree()), G.diameter())
#(tranzitivnost, št. vozlišč, št. povezav, vse točke reda 3, premer)
G.plot()
```

```
(True, 1310, 1965, True, 35)
```



Tretji graf ima veliko vozlišč, kar ga dela zelo nepreglednega, omogoči pa, da preverimo omenjene lastnosti nanotorusov tudi za velike grafe.

2.1 Opis delovanja funkcije nanotorus

Funkcijo nanotorus sva si zamislila tako, da deluje v naslednjih korakih:

1. Definicija razreda točk tipa U in V
2. Konstrukcija mreže šestkotnikov
3. Določanje, katere točke so vsebovane v paralelogramu
4. Konstrukcija povezav v nanotorusu
5. Končna funkcija `nanotorus`

2.1.1 Definicija razreda točk tipa U in V

V razredu `U` in `V` so oglišča šestkotnikov. Točke `U` so leva oglišča v vodoravnih daljicah, točke `V` pa desna. Točko `U(a, b)` dobimo tako, da seštejemo a vektorjev $A1$ in b vektorjev $A2$. Iz tega sva smiselno definirala atribute razreda. Najbolj ključni za nadaljnje delo so sledeči:

```
class U:
    def __init__(self, a, b):
        self.a = a # a konstanta ... koliko A1 vektorjev uporabimo
        self.b = b

    def koordinate(self):
        vektor = self.a * A1 + self.b * A2
        x = vektor[0]
        y = vektor[1]
        return [x, y]
```

```
    def sosedi(self):
        s = set({V(self.a, self.b), V(self.a, self.b - 1),
         V(self.a + 1, self.b - 1)})
        return s

    def premakni(self, c, d):
        return U(self.a + c, self.b + d)

class V:
    def koordinate(self):
        vektor = self.a * A1 + self.b * A2 + vector([2, 0])
        x = vektor[0]
        y = vektor[1]
        return [x, y]

    def sosedi(self):
        s = set({U(self.a, self.b), U(self.a - 1, self.b + 1),
         U(self.a, self.b + 1)})
        return s
```

2.1.2 Konstrukcija mreže šestkotnikov

S funkcijo `grid(k, l, m, n)` sva zajela vse točke, ki jih vzamemo v obzir pri konstrukciji nanotora. Pri tem velja poudariti, da ne bomo vseh točk uporabili v nanotoru, ampak le tiste, ki ležijo znotraj paralelograma, določenega z vektorjema `(k, l)` in `(m, n)`.

2.1.3 Določanje, katere točke so vsebovane v paralelogramu

Na tej točki določimo, katere točke ležijo v paralelogramu. Pri tem si pomagamo s funkcijo `zavoj(u, v, w)`, ki pove, kje leži točka `w` glede na vektor `uv`. Če točka leži levo od vektorja, je `zavoj(u, v, w) == 1`, če točka leži desno, bo `zavoj(u, v, w) == -1`, če je točka na premici, ki jo določa vektor, bo pa `zavoj(u, v, w) == 0`. Funkcija `v_paralelogramu(G, u0, u1, u2, u3, u4)` nam pove, katere točke danega grida so v paralelogramu, določenim z navedenimi točkami. Če sta vektorja kolinearna, je paralelogram izrojen. To uporabimo v funkciji `vozlisca_na_torusu(k, l, m, n)`:

```
def vozlisca_na_torusu(k, l, m, n):
     G = grid(k, l, m, n)

     u_00 = U(0,0)
     u_kl = U(k, l)
     u_klmn = U(k + m, l + n)
     u_mn = U(m, n)
```

```
    # ugotovimo orientacijo
    if zavoj(u_kl, u_mn, u_00) == 0:
        Uji, Vji = "Izrojen", "Izrojen" # niso mnozice
    elif zavoj(u_kl, u_mn, u_00) > 0:
        # u_00 lezi levo od vektorja u_kl u_mn
        Uji, Vji = v_paralelogramu(G, u_00, u_kl, u_klmn, u_mn)
    elif zavoj(u_kl, u_mn, u_00) < 0:
        # u_00 lezi desno od vektorja u_kl u_mn
        Uji, Vji = v_paralelogramu(G, u_00, u_mn, u_klmn, u_kl)

    return Uji, Vji
```

2.1.4 Konstrukcija povezav v nanotorusu

Določiti želimo povezave med vozlišči na torusu. Če so vsi sosedi na mreži dane točke tudi na torusu, potem lahko v graf te povezave kar dodamo. Sicer moramo soseda na mreži prestaviti za ustrezen vektor, bodisi `(k, l)`, bodisi `(m, n)`.Pri tem je izjema točka `U(0, 0)`, ki jo lahko dobimo tudi tako, da jo prestavimo za vektor `(k + m, l + n)`. Tako sva dobila funkcijo `povezave_na_torusu(k, l, m, n)`

```
def povezave_na_torusu(k, l, m, n):
    Uij, Vij = vozlisca_na_torusu(k, l, m, n)
    if Uij == "Izrojen":
        return Uij # paralelogram je izrojen
    else:
        vozlisca = union(Uij, Vij)
        # vozlisca znotraj paralelograma (brez stranic)
        pregledane = set()
        povezave = []
        for tocka in vozlisca:
            for sosed in (tocka.sosedi() - pregledane):
            # tocka.sosedi().difference(pregledane)
                sosed_kl = sosed.premakni(k, l)
                sosed_mn = sosed.premakni(m, n)
                if sosed in vozlisca:
                    povezave.append((tocka, sosed))
                else:
                    if sosed_kl in vozlisca:
                        povezave.append((tocka, sosed_kl))
                    elif sosed_mn in vozlisca:
                        povezave.append((tocka, sosed_mn))
                    elif sosed == U(k + m, l + n):
                        povezave.append((tocka, U(0, 0)))
```

```
                    # to je sosed_klmn, nasprotno oglisce
                pregledane.add(tocka)
        return povezave
```

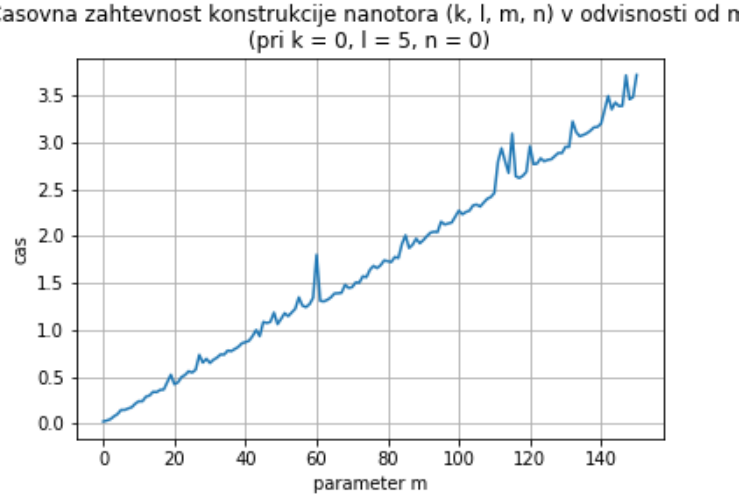
2.1.5 Končna funkcija `nanotorus`

Z zgornjimi funkcijami je zdaj preprosto definirati novo funkcijo, ki nam vrne nanotorus, dobljen z danima vektorjema, saj nam *Sage* dovoljuje, da graf definiramo samo s povezavami iz grafa. Imamo torej:

```
def nanotorus(k, l, m, n):
    p = povezave_na_torusu(k, l, m, n)
    if p == "Izrojen":
        print("Paralelogram je izrojen!")
    else:
        G = Graph(p, multiedges = True)
        G._pos = {v: v.koordinate() for v in G}
        return G
```

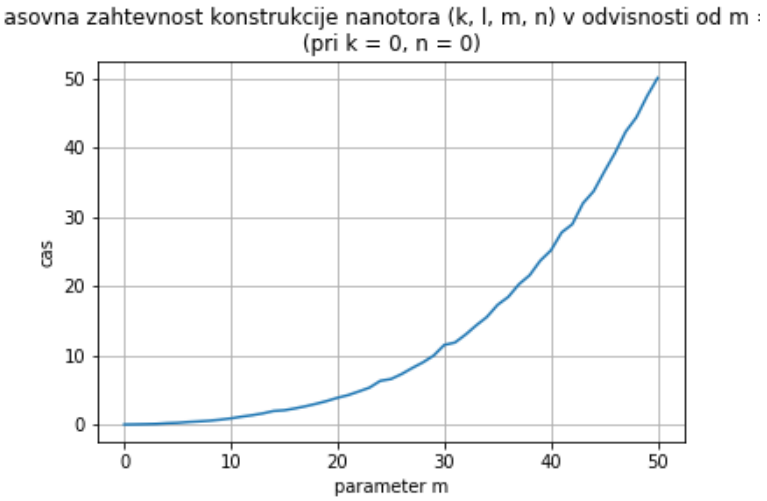
3 Časovna zahtevnost funkcije nanotorus

Časovna zahtevnost funkcije `nanotorus` je najbolj odvisna od velikosti mreže šestkotnikov, na kateri je treba preveriti vsebovanost točk v paralelogramu. Na sliki lahko vidimo koliko časa potrebuje funkcija za konstruiranje grafa v odvisnosti od enega parametra.



Vidimo, da je časovna zahtevnost v tem primeru linearna, kar lahko pojanimo s tem, da se ob podaljševanju paralelograma v eno smer tudi mreža

šestkotnikov poveča sorazmerno. Na naslednji sliki pa lahko vidimo, da v primeru podaljševanja paralelograma v dve različni smeri časovna zahtevnost funkcije raste kvadratično, saj se v tem primeru tudi mreža šestkotnikov povečuje v obe smeri.



4 Število točk do dane razdalje

Število točk na razdaljah, krajših od danega d, sva dobila s sledečo funkcijo. Ob pisanju poročila sva ugotovila, da bi jo bilo mogoče optimizirati, če bi dodala množico že pregledanih vozlišč.

```
def vozlisca_na_razdalji_d(graf, start, dodane, d=0):
    # dodane je na zacetku prazna mnozica
    # start je vozlisce, kjer zacnemo
    if d == 0:
        dodane.add(start)
    elif d < 0:
        print("Negativna dolzina!")
    else:
        dodane.add(start)
        for povezava in graf[start]:
            vozlisca_na_razdalji_d(graf, povezava, dodane, d - 1)
    return dodane
```

To funkcijo sva preizkusila na zelo različnih grafih z namenom, da bi našla formulo za izražanje števila točk z d, k, l, m in n. Pri enostavnih grafih sva našla nekatere formule, ampak niso držale za vse razdalje. Na večjih grafih pa preprosto nisva uspela dobiti povezave, ki bi zajemala, kar se dogaja s številom točk pri spremembi parametrov.

In [86]:
```
J = nanotorus(3,0,0,10)
r = J.diameter()
for i in range(r):
    print("{} tock na razdalji {}".format(len(vozlisca_na_razdalji_d(J, U(0,0), set(), i)), i))
```

```
1 tock na razdalji 0
4 tock na razdalji 1
10 tock na razdalji 2
18 tock na razdalji 3
26 tock na razdalji 4
33 tock na razdalji 5
39 tock na razdalji 6
45 tock na razdalji 7
51 tock na razdalji 8
57 tock na razdalji 9
```

In [87]:
```
J = nanotorus(4,0,0,10)
r = J.diameter()
for i in range(r):
    print("{} tock na razdalji {}".format(len(vozlisca_na_razdalji_d(J, U(0,0), set(), i)), i))
```

```
1 tock na razdalji 0
4 tock na razdalji 1
10 tock na razdalji 2
19 tock na razdalji 3
30 tock na razdalji 4
41 tock na razdalji 5
51 tock na razdalji 6
60 tock na razdalji 7
68 tock na razdalji 8
76 tock na razdalji 9
```

5 Izomorfnost