Összetett hálózatok segédlet

Horváth Árpád

2013. szeptember 26.

Az összetett hálózatok tudománya a valóságban előforduló, gráffal ábrázolható dolgokat (web, internet, ismeretségi hálózat, tápláléklánc, fehérjék kölcsönhatási hálózata, ...) vizsgálja. Általában ezek a hálózatok az idő során változnak, például a webet más gráf ábrázolta egy órája, mint most, mivel közben új weboldalak jelentek meg, de szűntek is meg, valamint a létező weboldalakon is raktak fel illetve töröltek linkeket.

A segédlet egy féléves dupla órás laborgyakorlatokra készült, melyen nem csak a hálózatok alapfogalmaival ismerkedünk meg, hanem a hálózatok jellemzőit konkrét programkönyvtárakkal – az igraph-fal és az arra épülő cxnettel – megvizsgáljuk valódi hálózatokra és modelljeikre.

Párhuzamosan fogjuk vizsgálni a valódi hálózatokat, azok modelljeit, illetve azok számítástechnikai vonatkozásait. A félév során a következő kérdésekre keressük a választ:

- Mennyire hasonlóak a valódi hálózatok? (A válasz az lesz, hogy gyakran sok szempontból nagyon is hasonlítanak egymásra, jobban, mint amit a véletlen művének tekinthetnénk)
- Milyen folyamatok alakítják ki a hálózatokat, azaz mi miatt hasonlítanak egymásra? (Ennek felderítésére hozzuk létre az egyes hálózatmodelleket.)
- Milyen jellemzőkkel érdemes leírni (az általában ábrázolhatatlanul nagy) hálózatokat?
- Hogyan tudjuk ezeket a jellemzőket számítógéppel hatékonyan vizsgálni?

A félév során több hasznos programot használunk. Lesz több otthon megoldandó programozási feladat is.

A használt programok teljesen jogtisztán használhatóak bárki számára, úgynevezett szabad szoftverek.

A feladatok megoldásának bemutatásával vagy teszttel kezdődik a legtöbb alkalom. Az egyes feladatokat nehézségtől függően egy hallgató, vagy egy hallgatópár kapja meg kidolgozásra. Megoldott feladatként csak olyan programok fogadhatóak el, amelyek ténylegesen futnak, és modulként behívva használhatóak a függvényei és osztályai; tehát csak ténylegesen kipróbált programok.

1. Python programozási alapismeretek

(A tárgy és a követelményrendszer ismertetésével együtt két dupla órában.)

A feladatok végrehajtásához több dolgot ismernünk kell. Ezeket a félév első két óráján sajátítjuk el. Az itt tanultakat folyamatosan használni fogjuk, ezért készségszinten kell tudni, és komolyan számon fogom kérni gyakorlatban.

A Vimmel és a verziókezelő rendszerekkel kapcsolatban a Linux alkalmazása segédletben találhatunk leírást.

Ismerni kell a *Vim szövegszerkesztő* használatát:

- Alapvető mozgási parancsok.
- Törlés, másolás.
- Többszörös ismétlés.

- Visszavonás.
- Kilépés mentéssel és anélkül.
- Pufferek listázása, váltás (másolás) pufferek között.
- A SnipMate használata.

Ismerni kell a *Python nyelvből*:

- A python és ipython parancsértelmezők elindítását és alapvető kezelését.
- A következő adattípusokat: int, float, list, tuple, str (később dict).
- A szeleteket (a[3:6], a[:30], a[-3:]).
- Listaértelmezés: `[érték for változó in sorozat if feltétel]`
- Objektumok metódusainak meghívását.
- A vezérlési szerkezeteket: if-elif-else, for, while.
- print és osztás a jövőben (`from __future__ import division, print_function`)
- A következő függvényeket és utasításokat: print, range, xrange, max, min, sum, all, any.
- A string-ek format és split metódusait.
- Szövegfájlok beolvasását, feldolgozását.
- A Python programok futtatását. (Futtathatóvá tétel is.)
- Modulok importálását.
- Függvények és osztályok létrehozását (def, class).
- Az assert utasítást és a tesztek futtatását (`if_name__ == "__main__": test()`)

Ismerni kell az *elosztott verziókövetés* alapfogalmait és pár dolgot a git kezeléséről:

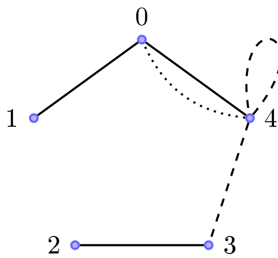
- Git-tároló klónozását és frissítését (clone, pull).
- Ágak közötti váltást (pl. master és develop a cxnet-ben)
- A github.com kezelését: felhasználó létrehozása, tárolók kezelése (fork, klónozása saját gépre).

Lásd még a linux tároló python/peldak/szamok.py fájlját. Az utolsó program, ahol magyar változóneveket használunk. `http://github.com/horvatha/linux`

$ ipython --pylab

```
# ipython --pylab (alias pylab)
li = range(5, 51, 5)
[i**2 for i in li]
[i**2 for i in li if i % 10 == 0]
li[2:5]              # 5 - 2 = 3 elem; 2-től, 5 már nincs benne
li[:3]               # első 3 elem
li[3:-4]
li[-4:]              # utolsó 4 elem
xr = xrange(5, 1000000)
# Nem foglal sok helyet, csak a meghíváskor állítja elő a következő elemet.
# Python3-ban a range majdnem ugyanaz mint python2-ben az xrange.
xr = xrange(2, 10)
list(xr)
for i in xr:
    print(i)

# Megváltoztatható (mutable) és változtathatatlan objektumok
li2 = li            # Az li és li2 nevek ugyanarra az objektumra mutatnak
li[2] = 3           # A list (és a set) megváltoztatható, li2 == li
li[3:6] = [1, 2]
li.append(0)
tup = (1, 2, 3)     # tuple
tup[2] = 3          # Hiba
st = "alma"
st[2] = "a"         # Hiba, a tuple, a str (és a frozenset) nem megváltoztatható
st[3:6]             # De mindegyik sorozattípus szeletelhető.

from __future__ import print_function
# Csak program elején. Python3-ban függvény a print, nem utasítás.
print?
print(1, 2, 3, sep=":", end=";;\n")
print(*li, sep=":", end=";;\n")
with open("repo/cxnet/mfng_hu.txt") as f:
    lines = f.readlines()
# Lezárja a fájlt, bármi hiba történik. Csak olvasható mód.

lines.append("Vége\n")
with open("temp.txt", "w") as f:
    f.writelines(lines)
with open("temp.txt", "a") as f:
    f.writelines(lines)
# Felülír (w) illetve hozzáfűz (a).
!wc temp.txt mfng_hu.txt
# 2 * (373 + 1) = 748 sor

lines[:10]
lines[101].split()
lines[101].split()[2]
int("50"), float("50"), float(5), int(5.4)
"{0}*pi =  {1:9.4f}".format(1, pi)
for i in range(1, 400, 77):
    "{0:4}*pi =  {1:9.4f}".format(i, i*pi)
"x{name:>11}x{name:^11}x {age:5.1f} {name} {age:X}".format(name="Guido", age=30)

with open("temp2.txt") as f:
    for i in range(1, 400, 77):
        print("{0:4}*pi = {1:9.4f}".format(i, i*pi), file=f)
!cat temp2.txt
```

```
$ ipython --pylab
any([True, False, True])
all([True, False, True])
any([False, False, False])
any([0, 0, 1])
# False, 0, [], "", {} hamis, többi általában igaz
any(["", "", "alma"])
li = range(5, 51, 7)
# Van-e a listában 10-zel osztható?
[(i%10) for i in li]
[(i%10 == 0) for i in li]
any([(i%10 == 0) for i in li])
[(i%7 == 0) for i in li]
# Van-e a listában 10-zel osztható?
any([(i%7 == 0) for i in li])
# Van-e a listában olyan szám, ami 10-zel osztva 5-öt ad maradékul?
any([(i%7 == 5) for i in li])
# Esetleg mind ilyen?
all([(i%7 == 5) for i in li])
max(li)
min(li)
sum(li)
```




```
A=set(range(4))
B=set(range(2,7))
A-B
A|B   # unio    (or)
A&B   # metszet (and)
A^B   # szimmetrikus differencia (xor)
A.add(8)
```

```
$ igraph
g = Graph()
summary(g)
g.add_vertices(5)
summary(g)
g.add_edges( [(0,1), (0,4), (3, 2)] )
summary(g)
g.is_connected()
g.is_loop()
any(g.is_loop())
g.is_multiple()
any(_)
g.add_edges( [(3,4), (4,4)] )
g.is_multiple()
g.is_loop()
g.is_connected()
plot(g)
summary(g)
g.add_edge(0,4)
any(g.is_multiple())
g.delete_vertices(4)
```

2. Hálózatok alapfogalmai és az igraph használata

(Két dupla órában.)

2.1. Fogalmak, elmélet

- Összetett hálózat, gráffal ábrázolható (network, graph), maximális élszám, egyszerű (simple) gráf
- Csúcs (vertex, többes száma vertices; vagy node), él (edge)
- Irányított/irányítatlan hálózat/gráf ((un)directed network)
- Teljes (full vagy complete) hálózat
- Komponens (component)
- Összefüggő (connected) hálózat
- Fa (tree), $\Delta = N - M$
  (N a csúcsok számát; M az élek számát jelöli, ha több komponens van, az i-dik komponensre $\Delta_i = N_i - M_i$)

Ha hálózatra olyan dolgot mondunk, amit gráfokra tanultunk (összefüggő, irányított, súlyozott, egyszerű, súlyozott, körmentes, átmérője, ...) akkor ugyanazt értjük rajta mint a gráfnál.

Az összefüggő körmentes gráfokat/hálózatokat nevezünk *fáknak*. Belátható, hogy egy *összefüggő* gráfra a következő két kijelentés egyenértékű:

- körmentes
- az élek száma a csúcsokénál eggyel kevesebb ($M = N - 1$)

Tehát az fákat az utóbbival is definiálhatjuk: olyan összefüggő gráfok, amelyben az élek számánál eggyel több csúcs van. Az összefüggőség feltétele fontos része mindkét definíciónak: érdemes egyszer végiggondolni, hogy nélküle hogyan változna a helyzet.

A Δ jelölést házi használatra vezettük be, mert ez a könnyen kiszámítható érték jól mutatja, hogy a hálózat adott komponense fa-e, vagy sem.

Szokás a csúcsok halmazát V-vel, az élekét E-vel jelölni, és a halmazok számosságát abszolútérték-jellel. Gyakran viszont a csúcsok számát N-nel, az élekét M-el jelöljük (gyakran kis betűvel is). Emiatt a következők azonos dolgokat jelölnek:

$$|V| \equiv N, \qquad |E| \equiv M$$

A továbbiakban mindkét jelölést fogom használni. Az utóbbit többnyire akkor, ha indexelni kell a mennyiséget, vagy programban, ahol a változónévben nem lehet abszolútérték-jel. Az utóbbi jelölést használom a `halozatok_vizsgalata*.pdf` fájlban is.

2.2. Tudnivalók a programozáshoz

Az igraph modul

- hálózat létrehozása, csúcs/él hozzáadása/törlése (élek hozzáadása, ha lehet, egyszerre, 100 élt egyszerre hozzáadni majdnem annyi idő, mint 1-et)
- VertexSeq(uence) és EdgeSeq objektumok használata: élek és csúcsok tulajdonságainak megadása és lekérdezése, szűrés, algráf létrehozása.
- gráfok kirajzolása

- csúcsszám, élszám meghatározása
- szomszédok meghatározása
- hálózat létrehozása Graph.Formula osztálymetódussal

Python és git

- függvény/metódus definiálása
- assert, raise, isinstance
- unittesztelés lényege és mikéntje
- Git a változtatások kezelése: add, commit, push

2.3. Tudnivalók a feladatokhoz 1. (a továbbiakra nézve is)

A feladatok előtt el kell készíteni a megfelelő teszteket is helyes értékek ellenőrzésére (később unitteszteket a kivételek felléptetésének ellenőrzésére). A programokat a tesztekkel együtt tar.gz vagy zip formában lehet leadni, vagy a github-ra feltölteni. A legtöbb feladatban metódust (tagfüggvényt) kell írni egy az igraph.Graph osztályból származtatott Network osztályhoz. A `Network.prufer` például ennek egy metódusát jelöli.

2.4. Feladatok

Az első feladatot (a) részét órán fogjuk elkészíteni a megfelelő tesztekkel.

2.1. feladat {} Hozzuk létre az alábbi metódusokat!

(a) Egy hálózatban megkeresi egy komponens összes csúcsát, amelyben egy megadott csúcs szerepel.

`Network.component(vertex)` $\longrightarrow$ `[csúcs, ...]`

A csúcsok sorszámmal adottak.

(b) A csúcsok listája ismeretében meghatározza a csúcsok által kifeszített részhálózatban a Δ értékét, azaz a benne szereplő élek számából levonva a csúcsok számát.

Használható a Graph.subgraph metódus.

`Network.delta(vertex_list)` $\longrightarrow \Delta$

(c) A hálózatban meghatározza a komponensek számát.

`Network.number_of_components()` $\longrightarrow$ `number`

Csak az eddig tanult igraph-függvények/metódusok használhatóak és az (a) részben található component metódus. Az igraph.Graph-objektum components metódusa esetleg tesztben.

2.2. feladat {} Hozzunk létre egy függvényt, amely teljes hálózatot hoz létre, mindegyik élt egyszer hozza létre. Nem használhatjuk a beépített Graph.Full osztálymetódust. Érdemes először összegyűjteni egy listába a hozzáadandó éleket, és egyszerre hozzáadni, mert úgy gyorsabb.

`full(N)` $\longrightarrow$ `igraph.Graph-objektum`

Hozzuk létre először a teszteket: élek számát, fokszámokat ellenőrizhetjük, és hogy nincs-e többszörös és hurokél. (`all`)

2.3. feladat {} Hozzuk létre az alábbi függvényt és metódust!

a) Éllistából és névfeloldásból hálózatot hoz létre.

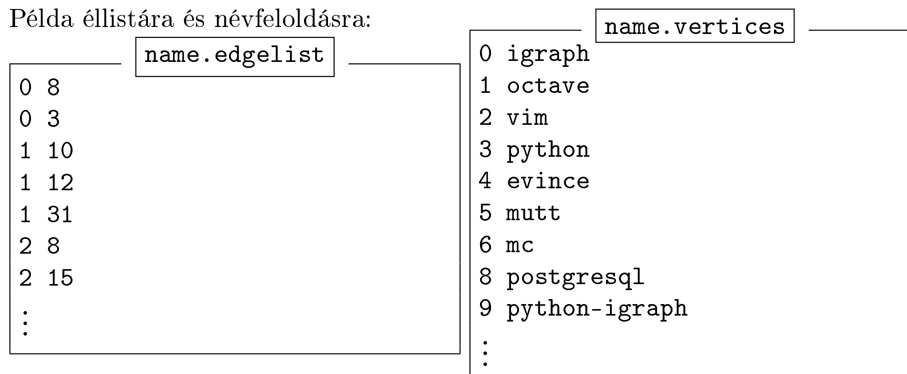
`from_edgelist(name, n=None)` $\longrightarrow$ `igraph.Graph-ból származtatott Network-objektum`

b) Kiírja a maximális fokszámú csúcs(ok) nevét.

`Network.maxdeg()` ⟶ `[(maximális fokszámú csúcs neve, fokszáma),...]`

Az igraph.maxdegree metódus csak teszthez használható.

Példa éllistára és névfeloldásra:



Hiányozhassanak vertex-azonosítók (mint fent a 7.), azaz az azonosítókat vertex-attribútumként tároljuk, ha egyáltalán tárolni kell!

2.4. feladat {} Hozzunk létre egy metódust, amely egy hálózatból létrehozza annak Prüfer-kódját ha az fa, és azzal tér vissza! Ha nem fa, `None`-t ad vissza.

`Network.prufer()` ⟶ `lista vagy None`

2.5. Függvények írása

http://docs.python.org/tutorial/controlflow.html#more-on-defining-functions

Default érték megadása

```
from __future__ import print_function

def info(net, diameter=False, welcome="Hello"):
    print(welcome)
    print("N =", net.vcount())
    print("M =", net.ecount())
    if diameter:
        print("d =", net.diameter())
    return net.vcount()

import igraph
fullnet = igraph.Graph.Full(8)
info(fullnet)
info(fullnet, True)
info(fullnet, welcome="Ciao!")  # Kulcsszó nélküli elöl.
info(welcome="Szia!", net=fullnet, diameter=True)
```

2.6. Metódusok írása hálózathoz

Először egy saját osztályt kell származtatni az eredeti hálózat (gráf) osztályból, úgy tudunk hozzáadni metódusokat.

Metódusok írása hálózathoz

maxmindeg.py

```
import igraph
class Network(igraph.Graph):
    def maxmindegree(self, write=True):
        "Returns with the maximal and minimal degree."
        deg = self.degree()
        if write:
            print("Maximal degree = {0}, "
                  "minimal degree = {1}".format(
                  max(deg),min(deg)))
        return max(deg), min(deg)

net = Network.Full(8)
maxd, mind = net.maxmindegree(write=False)
print(net.maxmindegree())
# Maximal degree = 7, minimal degree = 7
print(net.maxmindegree.__doc__)
# Returns with the maximal and minimal degree.
```

2.7. Unittest az előzőhöz

Az előző programkód végére lehetne írni két tesztet:

```
assert mind == 7
assert maxd == 7
```

Nagyobb programoknál a teszteket nem az egyes programfájlokba írjuk bele, hanem összegyűjtjük egy külön könyvtárba, hogy gyorsan tesztelhető legyen, hogy a program egy adott környezetben (Python verzió, operációs rendszer, telepített modulok), az egyes függvények jól működnek-e. Ezeket hívjuk unitteszteknek (unit test). Természetesen nem lehet mindenre unittest-et írni, de amit rutinszerűen kipróbálnánk minden változtatás után, azokra érdemes. Egy fejlesztési módszer a tesztvezérelt (test driven) fejlesztés, amikor a függvények megírása előtt írjuk meg a teszteket, hogy például milyen visszatérési értéket kell adnia a függvénynek, milyen kivételeket kell felléptetnie hibás argumentumok esetén. A unittesztek előnye a sima assert-es teszteléshez képest, hogy nem akad el az első hibánál, hanem az összes hibáról jelentést kapunk, és az eltérésekről bővebb információt nyújt. Például két lista összehasonlítása esetén kiírja, hogy hányadik elemnél milyen eltérés van.

Bővebben: a docs.python.org oldalon a Library reference-ben a unittest modul, illetve a Dive into Python 3 *Unit Testing* fejezete.

unittest

test_maxmindeg.py

```
import maxmindeg
import unittest

class testNetwork(unittest.TestCase):
    def testDegree(self):
        net = maxmindeg.Network.Full(8)
        maxd, mind = net.maxmindegree(write=False)
        self.assertEqual(maxd, 7)
        self.assertEqual(mind, 7)

if __name__ == "__main__":
    unittest.main()
```

2.8. Hálózatok, éleik és csúcsaik attribútumai

```
$ igraph
net = Graph.Formula("a-b-c-d, a-c, b-d")
plot(net)
net.vs["name"]
net.vs["label"] = net.vs["name"]
plot(net)  # label (v) megjelenik az ábrán
net.es["weight"] = net.es["width"] = [1,4]
net.vs["size"] = [10, 20, 30]
summary(net)                  # weight (e): W jelzőbit
plot(net)  # width (e) és size (v) megjelenik az ábrán
net.vs["type"] = [0, 1]         # type (v): T jelzőbit
net["name"] = "Small"
net["date"] = "2012-03-07"
summary(net) # name (g) megjelenik
# (un)directed|named|weighted|typed |V| |E| -- name
summary(net, verbosity=2)
del net.es["weight"]
# Irányított hálózat/gráf
net = Graph.Formula("a>b>c>d, a>c, b>d")
net.vs["label"] = net.vs["name"]
# Élek és csúcsok színei
net.es["color"] = ["yellow", "red"]
net.vs["color"] = ["red", "blue"]
plot(net, layout="circular")
```

2.9. Fokszám, szomszédok, szűrés, vs/es attribútumai

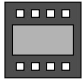
```
$ igraph
net = Graph.Formula("a-b-c-d, a-c, b-d-e-f")
deg = net.degree()
print(max(deg), min(deg))
net.neighbors(0)
nb = net.neighbors("a")
vs = net.vs.select(nb)    # vertex sequence a szomszédokból
vs.indices
vs["name"]                             # a szomszédok nevei
net.vs.select(nb)["name"]            # ugyanaz egy lépésben
net.vs.select?              # itt bőven le van írva minden
vs["color"] = "yellow"      # a szomszédok sárgák
net.vs["color"]             # a nem szomszédoknál None áll
net.vs["color"] = "red green blue green".split()
net.vs.select(color = "green")
net.es["weight"] = [1, 4, 2, 6]
es = net.es.select(weight_gt = 3)  # a 3-nál nagyobb súlyú élek
es.indices
es["color"] = "blue"

N = net.vcount()    # csúcsok (vertex) száma
M = net.ecount()    # élek (edge) száma
2 * M / N           # átlagfokszám
2.0 * M / N         # Python2-ben csak ez jó
from  __future__ import division # Python2-ben másik lehetőség
sum(deg)/len(deg)
```

3. Erdős–Rényi modell

3.1. Fogalmak, elmélet

- Erdős–Rényi modell
- Élvalószínűség (p) és kiszámítása valódi hálózatokra
- Csúcsok távolsága, hálózat átmérője (eltérő definíciók nem összefüggőre)

3.2. Tudnivalók a programozáshoz

 Az összetett hálózatok oldalról elérhető a pylab-1.ogv fájlban egy kisfilm, amelyben megtekinthető, hogyan használható a pylab.

- átmérő meghatározása (szükséges idő)
- komponens meghatározása

Pylab (matplotlib) modul

- linspace
- plot, loglog, semilogy
- függvényábrázolás (x,y) (y) (x,y,fmt) (y,fmt)
- függvény feliratai
- függvényábra mentése (pdf, svg, png, eps)

3.3. Tudnivalók a feladatokhoz 2. (a továbbiakra nézve is)

A feladatoknál gyakran van szükség egy ábra elkészítésére, vagy egy táblázat kitöltésére. Jó ábra készítéséhez esetenként sok idő is kellhet, ezért ne arra törekedjünk, hogy az ábra generálása gyorsan lefusson, hanem ha hosszú idő kell hozzá, figyeljünk, hogy az ábrát mentsük el (savefig). A táblázatok esetén néha célszerűbb, ha mi jegyezzük le az adatokat papírra vagy más formában, mintha programot írnánk, ami ilyen formában adja ki.

3.4. Feladatok

3.1. feladat {} Hozzuk létre az alábbi két függvényt!

a) Adott N csúcsszámú és p élvalószínűségű Erdős–Rényi hálózat esetén határozza meg a legnagyobb komponens méretét (=csúcsainak számát) a teljes hálózat méretéhez viszonyítva.

`largest_component_size(N, p)` $\longrightarrow$ `relatív méret` $\in (0, 1]$

b) Kirajzolja a relatív méret változását a p függvényében. A kirajzolásban legyen benne az a rész, ahol átlépi a méret az 50 %-ot. Mentsük el az $N = 1000$ esetre kapott ábrát!

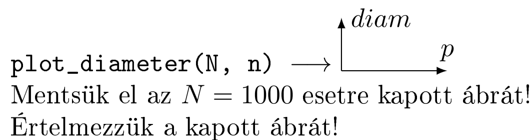
`plot_lc_size(N, p_start, p_stop, p_num)` $\longrightarrow$ `ábra`

p_num számú értéket ábrázol p_start és p_stop között.

c) Készítsünk táblázatot, amely a N függvényében megadja azt a $p_{\text{határ}}$ határt, ahol 50 % felett lesz legnagyobb komponens mérete. ($N \in [100, 10000]$)

N	$p_{\text{határ}}$
100	
500	
1000	
5000	
10000	

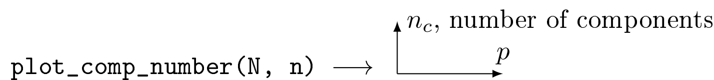
3.2. feladat {} Hozzunk létre egy függvényt, amely az Erdős–Rényi modellből kapott hálózatban (a legnagyobb komponens) átmérőjének változását rajzolja ki adott N érték esetén a p függvényében. p közel 0 és 1 között lépkedjen végig n darab értéken keresztül.



Mentsük el az $N = 1000$ esetre kapott ábrát!

Értelmezzük a kapott ábrát!

3.3. feladat {} Hozzunk létre egy függvényt, amely az Erdős–Rényi hálózat komponenseinek a számát mutatja adott N esetén p függvényében. A lineáris skála helyett – amennyiben többet mutat – használhatunk a függőleges tengelyen logaritmikusat is (plot helyett semilogy). p közel 0 és 1 között lépkedjen végig n darab értéken keresztül.



3.4. feladat {} Hozzunk létre egy függvényt, amely megbecsli adott (N, p) paraméterű Erdős–Rényi hálózat esetén a keletkezett hálózatok tényleges élvalószínűségének σ szórását n-szer generálva a hálózatot.

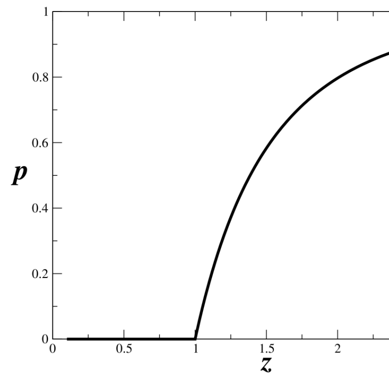
A becslés az alábbi képlet alapján történjen, ahol $\bar{p}$ a meghatározott p_i-k átlaga.

$$\sigma \approx \sqrt{\frac{\sum_{i=1}^{n}(p_i - \bar{p})^2}{n}}$$

`p_sigma(N, p, n)` $\longrightarrow \sigma$

Készítsen táblázatot a szórásról! Az n értéke mindegyik esetben legalább 10 legyen.

$p =$	0,1	0,2	0,5	0,9
N=1000				
N=500				
N=100				



1. ábra. A legnagyobb komponens mérete a teljes hálózat méretéhez képest véletlen hálózat esetén az átlagfokszám függvényében.

Példák (Most nem az igraph paranccsal indítjuk, hanem utólag importálunk.):

$ ipython3 --pylab

```
import igraph
net = igraph.Graph.Erdos_Renyi(1000, .01)
net.diameter()
cc = net.components()
sizes = cc.sizes()
sizes
# Mennyire változnak az értékek, ha az eddigieket újból végrehajtuk?
# Hogyan határozhatjuk meg a komponensek számát, legnagyobb komponens
# méretét a sizes listából?

# from __future__ import division
max(sizes)/net.vcount()
net.diameter()
cc.giant()
giant = _
igraph.summary(giant)
igraph.plot(giant)

len(sizes)
cc.membership
sizes[:20]
max(sizes)
sorted(sizes)
ix = sizes.index(max(sizes))
cc.subgraph?
sub = cc.subgraph(ix)
cc.membership
```

Erdős és Rényi azt találták, hogyha egy véletlen hálózatokból álló $\Gamma_{N,M(N)}$ sorozatot hozunk létre úgy, hogy a csúcsok N száma végtelenhez tart, az élek $M(N)$ számát pedig úgy növeljük, hogy az átlagos fokszám adott z értékhez tartson, akkor az átlagos fokszám függvényében pontosan meghatározható, hogy a csúcsok mekkora hányada lesz a legnagyobb komponensben. Ha a legnagyobb komponens csúcsainak számára a $N^*_{N,M(N)}$ jelölést használjuk, akkor akármilyen kicsi

$\varepsilon > 0$ értékre

$$\lim_{N\to\infty} Prob\left(\left|\frac{N^*_{N,M(N)}}{N} - G(z)\right| < \varepsilon\right) = 1, \tag{1}$$

ahol $G(z)$ az 1. ábrán látható pontosan meghatározható függvény. Míg a $z = 1,4$ átlagfokszám-érték körül a csúcsok fele, $z = 2$ esetén már a csúcsok 98%-a fog a legnagyobb komponenshez tartozni.

4. A Price-modell, a Barabási–Albert-modell és a fokszámeloszlás

(Két alkalom.)

Lásd a Price-modell szimulációjáról Newman könyvének 495. oldalát.

Irányított hálózatot hozok létre, kezdetben m csúcsom van. Minden lépésben egy új csúcsot adok a hálózathoz, amely m csúccsal csatlakozik a régiekhez. (Általánosabb esetben m csak átlagérték.)

A csatlakozási valószínűség arányos a (befokszám + a) értékkel, ahol $a > 0$ a modell egyik paramétere.

Az i-dik csúcs befokszámát jelölje q_i, akkor a hozzá való csatlakozás valószínűsége:

$$\Pi_{a,i} = \frac{q_i + a}{\sum_i (q_i + a)}$$

Bizonyítható, hogy ekkor sok lépés után olyan kifokszám-eloszlást kapunk, amely nagy q értékeknél hatványfüggvényhez közelít:

$$p_q \sim q^{-\gamma}, \qquad \text{ahol} \qquad \gamma = 2 + \frac{a}{m}$$

Belátható, hogy a Price-modellnek megfelelő hálózatot kapunk, ha egy régi csúcs kiválasztásakor (amihez csatlakozik az új csúcs) a következő algoritmust használom.

- $m/(m+a)$ valószínűséggel befokszámmal arányosan választok csúcsot. (Egy listában annyiszor tüntetem fel a csúcsokat, ahány él vezet hozzájuk, és random.choice függvény használatával választhatok egyet.)
- Különben egyforma valószínűséggel választok minden csúcsot.

Minden egyes választáskor meg kell nézni, hogy azt a csúcsot abban a lépésben nem választottuk-e már.

A `halozatok_vizsgálata*.pdf` fájlban találhatóak a Barabási–Albert modellel, és a fokszámeloszlással kapcsolatos anyagok.

A Barabási–Albert modell nagyjából egyezik a Price-modellel; a különbség, hogy irányítatlan hálózatot hoz létre, és benne a csatlakozás valószínűsége arányos a fokszámmal, így csak egy paramétere marad, az m. A kapott hálózat kitevője és számítógépes generálása a fentiekből megkapható.

Tulajdonképpen a generálás megoldható úgy is, hogy a Price-modell szerint generálok megfelelő paraméterekkel, majd irányítatlanná teszem a hálózatot.

Története: Herbert Simon közgazdász 1955 előtt figyelt fel több mennyiség, például a személyek vagyonának hatványfüggvény eloszlására. Sole de Price ugyanezt a hatványfüggvény-eloszlást találta cikkre hivatkozások számában, és ehhez alakította ki modelljét (1976). Barabási Albert-László és Albert Réka a web fokszámeloszlását vizsgálták, és arra kerestek modellt (1999). Nem ismerték Price modelljét. Tőlük származnak a skálafüggetlen (scale-free) hálózatok és a népszerűségi csatlakozás (preferential attachment) kifejezések.

4.1. Fogalmak, elmélet

- Price-modell
- Barabási–Albert modell
- fokszám (degree)
- átlagos fokszám kiszámítása
- fokszámgyakoriság és fokszámeloszlás (degree distribution) függvény

- összegzett fokszámeloszlás (cumulative degree distribution)
- Milyen ábrán lesz egyenes?

4.2. Tudnivalók a programozáshoz

- Listák indexei és szeletei (egyéb listaszerű objektumok)
- A range függvény részletesen
- `[ <kifejezés> for <változó> in <listaszerű obj.>]` szerkezet
- Listák max/min értéke, hossza
- loglog skálájú függvények (semilogx, semilogy, loglog)

4.3. Feladatok

4.1. feladat {} Hozzunk létre egy metódust, amely egy hálózat fokszámeloszlását kirajzolja! Lehessen választani a négyféle skála között.

4.2. feladat {}

a) Írjunk olyan függvényt, amely egy hálózat összegzett fokszámeloszlását rajzolja ki a loglog skálán.

b) Írjunk olyan függvényt, amely a Barabási–Albert modell alapján generált hálózatok összegzett fokszámeloszlását rajzolja ki az m különböző értékeire. Változik-e a függvény kitevője („loglog-os meredeksége”) az m-el?

4.3. feladat {} Hozzuk létre az alábbi függvényeket!

(a) Írjunk egy olyan függvényt, amely a Price-modell alapján készít hálózatot!

Fontos feladat. Később használjuk fel valódi hálózatokkal összevetésre!

(b) Skálafüggetlen eloszlást mutat-e a fokszám, a kifokszám és a befokszám?

(c) Írjunk egy olyan függvényt, amely a Barabási–Albert modell alapján hálózatot készít.

4.4. feladat {} Készítsünk ábrát, amely megmutatja, hogy Barabási–Albert modellből származó hálózatban, hogyan függ az átlagos fokszám az m értékétől egy kellően nagy N esetén!

Magyarázzuk meg a kapott függést!

4.5. feladat {} Hozzuk létre az alábbi függvényeket! Gondoljuk végig, hogy érdemes-e egyetlen osztály metódusaiként létrehozni!

a) Írjunk olyan függvényt, amely adott N és m esetén meghatározza egy Barabási–Albert modell alapján generált hálózat átmérőjét, és egy hasonló csúcs és élszámú Erdős–Rényi modellel generált hálózatét!

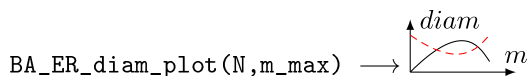
`BA_ER_diameter(N, m)` $\longrightarrow$ `(BA_diam, ER_diam)`

b) Hozzunk létre olyan függvényt, amely adott N esetén végigmegy az $m' = 1, \ldots, m_{max}$ értékeken, és meghatározza a BA illetve ER modellből származó hálózatok átmérőjét!

`BA_ER_diam(N, m_max)` $\longrightarrow$
$((d_1, d_2, d_3, \ldots, d_{m_{max}}), (d'_1, d'_2, d'_3, \ldots, d'_{m_{max}}))$

d_i: i-dik BA-átmérő, d'_i: i-dik ER-átmérő.

c) Hozzunk létre olyan függvényt, amely ábrázolja az előbbi függvény által kapott értékeket!



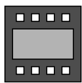
d) Mentsük el az ábrát $N = 1000$, $m_{max} = 20$ esetre, és értelmezzük!

5. Egy valódi hálózat: a csomagfüggőségi hálózat

5.1. Fogalmak, elmélet

- A deb-csomagok függőségi rendszere.
- A virtuális csomagok (pl. editor).
- Hálózatok tárolása/behívása.
- Erősen és gyengén összekapcsolt komponensek (strongly and weakly connected components, SCC, WCC).

5.2. Tudnivalók a programozáshoz

 Az összetett hálózatok oldalról elérhető a cxnet-debnetwork-1.ogv fájlban egy kisfilm, amelyben megtekinthető, hogyan hozható létre a csomagfüggőségi hálózat, a 2-esben pedig, hogy hogyan jeleníthetőek meg egy csomag szomszédai. Az egyes filmekhez tartozó Python-parancsok külön fájlban megtalálhatóak ott.

- A git elosztott verziókezelő rendszer használata.
- A cxnet modul használata.
- A cxnet modul és általában a modulok felépítése.
- Más hálózatok letöltése a cxnet-tel.
- SCC és WCC meghatározása.
- Irányítatlan átmérő meghatározása.
- Időmérés ipythonban: `%time`

5.3. Feladatok

5.1. feladat {} Vizsgáljuk meg hány komponensből áll a csomagfüggőségi hálózat! Mekkora a legnagyobb komponens mérete és átmérője (mint irányítatlan hálózatnak)?

5.2. feladat {} Elemezzük a hálózatot a fokszámok szempontjából!

összegzett fokszámeloszlás, maximális fokszám, legnagyobb fokszámú csúcsok

Melyik csomagnak van a legnagyobb fokszáma? Vajon miért?

Keressünk a csomagok között programnyelvek és standard könyvtáraik (Python, Perl, C, C++, FORTRAN, mono. . .) illetve fordítóik (gcc, gfortran, . . .) csomagjait! Milyen helyezésük van?

5.3. feladat {} Ábrázoljuk egy ábrán a csomagfüggőségi hálózat és a hozzá legjobban hasonlító Erdős–Rényi illetve Barabási–Albert hálózatok fokszámeloszlását!

Melyikhez hasonlít jobban a csomagfüggőségi hálózaté?

5.4. feladat {} Vizsgáljuk a hálózatot a 4.2. feladathoz hasonlóan, de a befokszámok szempontjából.

5.5. feladat {} Vizsgáljuk a hálózatot a 4.2. feladathoz hasonlóan, de a kifokszámok szempontjából.

6. Véletlen meghibásodások és támadások hatása

6.1. Fogalmak, elmélet

- véletlen meghibásodások modellezése
- célzott támadások modellezése
- milyen jellemzőket vizsgáljunk? (legnagyobb komponens relatív mérete, komponensek száma, átmérő, ...)

6.2. Feladatok

6.1. feladat {} Hozzunk létre egy függvényt vagy metódust, amely egy hálózat véletlen meghibásodásainak hatását vizsgálja egy irányítatlan hálózatban!

Hasonlítsuk össze az Erdős–Rényi és Barabási–Albert modellből származó hálózatokra, valamint a csomagfüggőségi hálózat irányítatlan változatára tett hatását!

6.2. feladat {} Hozzunk létre egy függvényt vagy metódust, amely egy hálózatra gyakorolt célzott támadás hatását vizsgálja egy irányítatlan hálózatban! (Célzott támadáskor minden lépésben a legnagyobb fokszámú élt távolítjuk el.)

Hasonlítsuk össze az Erdős–Rényi és Barabási–Albert modellből származó hálózatokra, valamint a csomagfüggőségi hálózat irányítatlan változatára tett hatását!

Mennyire védett az Internet célzott támadásokkal szemben? Javasoljunk stratégiákat az Internet megvédésére célzott támadásokkal szemben!

7. Gráfelméleti adatstruktúrák és hálózatok tárolása fájlokban

7.1. Fogalmak, elmélet

- Szomszédsági mátrix (adjacency matrix)
- Szomszédsági lista
- Éllista (indexeléssel vagy anélkül)

Irodalom Katona–Recski–Szabó: A számítástudomány alapjai, 3. (és 2.) fejezet, elektronikus formában fejezetenként is kapható a TypoTEX kiadótól.

- éllista
- gml-fájl
- GraphML-fájl (XML alapú)

7.2. Versenyfeladatok

A programoknak csak irányítatlan, súlyozatlan hálózatokat kell tudni vizsgálniuk.

a) Feltételek:

- tetszőleges adatstruktúrával
- tetszőleges programozási nyelven
- nem használhat hálózatkezelő programkönyvtárat, csak a nyelv standard könyvtárát
- *végrehajtási sebesség számít* hasonló hardveren
- a kevésbé gyors is kap pontot csak kevesebbet
- előny, ha Windows és Linux alatt is fut
- előny, ha kihasználja egy számítógép többprocesszormagját

b) Adottak:

- két csúcslista
  vl1 = (0,1,...,999)
  vl2: a vl1 200 elemű részhalmaza
- két éllista
  el1 = $((v_1, v_2), (v_3, v_4), (v_5, v_6), (v_7, v_8), \ldots, (v_{1999}, v_{2000}))$
  1000 elemű éllista, ahol $v_i \in vl1$
  el2: az el1 részhalmaza

c) Feladatok:

7.1. feladat {} Hozzunk létre egy olyan programot, mely ilyen sorrendben:

- hozzáveszi a hálózathoz vl1 csúcsait
- hozzáveszi a hálózathoz el1 éleit
- eltávolítja a hálózatból el2 éleit
- eltávolítja a hálózatból vl2 csúcsait a hozzávezető élekkel

7.2. feladat {} Adott egy $N = 10000$ csúcsú hálózat $vl = (0, 1, 2, \ldots, 9999)$ csúcsokkal és $10000 < M < 50000$ darab éllel.

Írjunk olyan programot, amely összeszedi azokat a csúcsokat, amelyek vl2 legalább egy csúcsával szomszédos.

7.3. feladat {} Adott egy $N = 10000$ csúcsú hálózat $vl = (0, 1, 2, \ldots, 9999)$ csúcsokkal és $10000 < M < 50000$ darab éllel.

Meghatározza a hálózat komponenseit. (Például eltárolja minden csúcsra, melyik azonosítószámú komponensben van benne.)

7.4. feladat {} Adott egy $N = 10000$ csúcsú hálózat $vl = (0, 1, 2, \ldots, 9999)$ csúcsokkal és $10000 < M < 50000$ darab éllel.

Kiszámítja minden csúcs csoporterősségi együtthatóját*, és kiírja ezek átlagát. (*Ha a fogalom még nem szerepelt, nézzenek utána vagy kérdezzenek rá.)

7.3. Elméleti összefoglaló

Az adatszerkezetek kiválasztásánál két nagyon fontos tulajdonságot figyelembe kell mindenképpen venni: (1) mekkora memória kell a tároláshoz, (2) mennyi időbe telik a gráfokon végzett művelet (módosítás, vagy információszerzés) végrehajtása.

A szükséges tárméreteket (memóriaszükségleteket) a következő példán fogjuk vizsgálni:

$$N = 100000, \quad M = 100000 \quad \Rightarrow \quad \langle k \rangle = \frac{2M}{N} = 2$$

Konkrét memóriaszükséglet számításakor feltételezzük majd, hogy az adatokat 4 bájtos egészként tároljuk.

A továbbiakban csak az élekkel kapcsolatos műveleteket vizsgáljuk, amelyek a következők: (a) élek beszúrása, (b) törlése, (c) megkeresése (van-e i. és j. között él), és (d) megszámolása (hány szomszédja van i.-nek)

Ugyanígy vizsgálhatnánk egy csúcs hozzáadásához vagy elvételéhez szükséges időt is.

Olyan összetettebb műveletek, mint az átmérő meghatározása, ilyen elemibb műveletekből állnak össze.

7.3.1. Szomszédsági mátrix (adjacency matrix)

$N \times N$-es $A = (a_{ij})$

Először az irányítatlan esettel foglalkozunk.

Ha i. és j. között n él van, akkor $a_{ij} = a_{ji} = n$.

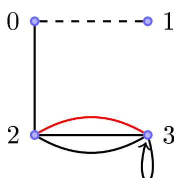
Mivel a mátrix szimmetrikus, elég lenne csak a főátlót és a felette levő értékeket tárolni, de ekkor bonyolultabb lenne az adatok elérése.

n darab hurokélnél (self-loop/self-edge) $a_{ii} = 2n$ (minden hurokélnek két végpontja van).

Egyszerű gráf: nincs benne többszörös és hurokél. Ezekben $a_{ii} = 0$ a többi érték 0 vagy 1 lehet.

A példában 10^{10} elem lesz!!! Ha 4 bájtos egészeket használunk 40 GB kell. A fenti négy művelet (a–d) viszonylag egyszerűen elvégezhető szomszédsági mátrix-szal, de ha nagy gráfot tárolunk, akkor nem biztos hogy egyszerre belefér a memóriába, ami jelentősen lelassítja a műveleteket.

Irányított hálózat esetén a mátrix nem szimmetrikus, a_{ij} az i.-ből j.-be mutató élek száma.



A fenti gráf folytonos vonallal jelzett részéből indulunk ki, ezt az alábbi A_0 mátrix írja le. Ha hozzáadjuk a szaggatott élt, akkor csupán két helyen kell megváltoztatni a mátrixot, ráadásul

tudjuk pontosan hol, nem kell keresgélni, és akkor is, ha ezek után a piros élt töröljük (A_1 és A_2 mátrixok).

$$\mathbf{A}_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 \end{bmatrix} \quad \mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 \end{bmatrix} \quad \mathbf{A}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 \end{bmatrix}$$

Ha meg akarjuk keresni, hogy két csúcs között van-e él, akkor a mátrix egy elemét kell csak kiolvasni, és megnézni, hogy 0-e vagy nem. Az eddigi műveletek időigénye független a gráf méretétől, konstans idő alatt megvalósítható.

A fokszám megkereséséhez egy teljes sort összegezni kell. Ebben az esetben az időigény a csúcsok számával arányos ($t \sim N$). Ha a csúcsok indexelését 0-val kezdjük:

$$k_i = \sum_{j=0}^{N-1} a_{ij}$$

7.3.2. Szomszédsági lista

Az alábbi mutatja a folytonos gráf szomszédsági listáját. A listák nem feltétlenül rendezettek a szomszédok sorszáma szerint.

```
i: k_i; szomszédok listája
-------------------------
0:   1; 2
1:   0;
2:   4; 0 3 3 3
3:   5; 2 2 2 3 3
```

Végiggondolható, hogy él beszúrása konstans idő alatt megoldható. A másik három művelethez (b–d) viszont egy vagy két sor elemeit végig kell nézni, amelyhez szükséges idő az átlagfokszámmal arányos $t \sim \langle k \rangle$.

A szomszédsági lista olyan gráfok esetén, amelyben a lehetséges élek töredéke létezik csak sokkal kevesebb memóriát igényel, mint a szomszédsági mátrix.

A példánkban minden él kétszer szerepel a listában, és ha tároljuk a fokszámot is, akkor a memóriaigény $\sim 2M + N$. A legelső (i) oszlop tárolása nem szükséges. Továbbra is minden adatra 4 bájtot számítva 1,2 MB elég.

7.3.3. Élpárok tárolása (indexelés nélkül)

Ha az egyes éleknek tulajdonságaik vannak, az az előző adatszerkezetekben nem tárolhatóak. A három él a 2-es és 3-as csúcs között nem rendelkezik egyedi azonosítóval, azaz megkülönböztethetetlenek.

Ha az éleket élpárokként tárolom, akkor az éleket azonosíthatja az élpárok sorszáma. Az élpárokat tárolhatom két tömbben, az elsőben az egyik végpontját, a másikban a másikat. Irányítottnál pedig a kiindulót az egyikben a célt a másikban. A tömbök neveit `src`-nek illetve `dest`-nek választottam (forrás és cél).

A fenti folytonos vonallal jelölt gráf így ábrázolható:

```
src   = [0 2 2 2 3]
dest  = [1 3 3 3 3]
```

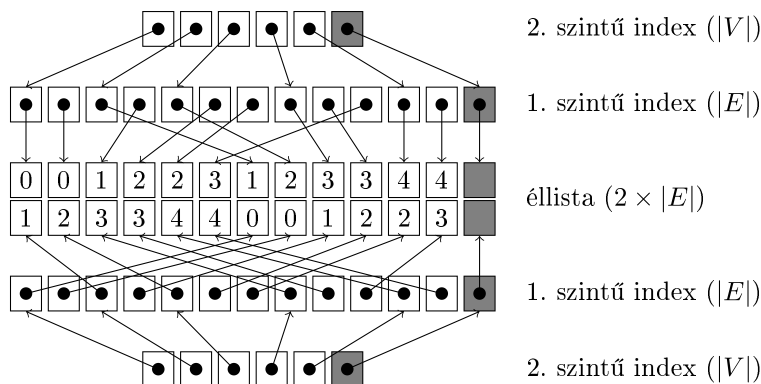
Az élek tulajdonságai (pl. egy elektromos vezetéken átvihető maximális áram egy áramhálózatban) hasonló tömbben tárolható.

(Az N értéket is tárolnunk kell, mert (1) lehet, hogy az legnagyobb sorszámú élhez például nem vezet él, és akkor az élpárokban nem szerepel a legnagyobb sorszámú csúcs, és (2) ha szerepelne is, akkor is sok idő lenne, míg megtalálom az élpárok listájában. De ez csak egyetlen szám.)

A memóriában a csúcsok számának kétszeresével megegyező számú egészet kell tárolni, ami a korábbi példával számolva 800 KB-on elfér.

7.3.4. Adatstruktúrák az igraph-ban.

Éllista kettős indexeléssel (igraph)



Összes tárigény: $4|E| + 2|V|$

A memóriában a korábbi példával számolva 2400 KB szükséges. Ez háromszorosa a sima éllistának. Korábban volt szó róla, hogy a következő jelöléseket azonosként használjuk: $M = |E|$, $N = |V|$

Gondoljuk végig, hogyan segít a kettős indexelés egy csúcs fokszámának és szomszédainak megkeresésében!

A csúcs fokszámának és a szomszédok megkeresésében ugyan segít az igraph adatszerkezete, élek beszúrása és törlése viszont jelentősen bonyolultabbá válik. Az igraph jelenlegi adatszerkezetéről azt kell tudni, hogy egyetlen él beszúrása nem sokkal kevesebb idő, mint ezeré, mivel a kettős index előállítása hosszú időt vesz igénybe. Az igraph tehát olyan esetekben jó, amikor a hálózatot kevés lépésben (esetleg egyszerre) létre tudjuk hozni, és időigényes vizsgálatokat végzünk rajta. Például az átmérő számításának ideje jelentősen lerövidül az indexelés nélküli éllistával történő munkához képest. Az igraph kimondottan alkalmas ilyen vizsgálatokra, mivel ezek a számítások C nyelven kerültek kódolásra.

7.3.5. A legfontosabb tudnivalók táblázatos összefoglalása

A tárméret képletét nem megjegyezni kell, hanem a tárolás módjának ismeretében meghatározni.

	tárigény				
adat-struktúra	általában	példában	szomszédkeresés időigénye	egyéb	
szomszédsági mátrix	$\sim N^2$	40 GB	$\sim N$ (nagy)	gyors módosítás	
szomszédsági lista	$\sim 2M + N$	1,2 MB	$\sim k_i$		
éllista	$\sim 2M$	0,8 MB	$\sim M$ (nagy)	kevés élnél (ritka gráf) kis tárigény	éltulajdonságok könnyen tárolhatók!
éllista kettős index-szel	$\sim 4M + 2N$	2,4 MB	$\sim k_i$		

8. A fokszámeloszlás simítása és a kitevő becslése, egyéb modellek

8.1. Fogalmak, elmélet

- skálafüggetlen hálózatok
- bin-ek, logaritmikus bin-elés
- maximum likelyhood módszer
- képlet a kitevőre és szórásra
- kifokszám-eloszlás, befokszám-eloszlás

8.2. Tudnivalók a programozáshoz

- osztályok a Pythonban (DegreeDistribution alapján)
- `__init__(self, ...)`
- `[ <kifejezés> for ... ]` szerkezet feltétellel (DegreeDistribution alapján) x,y változópárral
- ki- és befokszámok meghatározása

8.3. Feladatok

8.1. feladat {} Készítsünk olyan függvényt, amely egy hálózatban véletlenszerűen kiválaszt egy csúcsot, majd annak szomszédai közül véletlenszerűen egyet, és visszatér a választott szomszéd fokszámával: k'-vel.

Készítsünk egy másik függvényt, amely egy hálózatra n-szer lefuttatva az előzőt, átlagolja a fokszámot $\langle k' \rangle$, és visszatér a $\langle k \rangle / \langle k' \rangle$ (átlagfokszám/szomszéd átlagos fokszáma) értékkel.

Futtassuk le a programot egy Barabási–Albert hálózatra és egy azonos csúcsszámú, várhatóan azonos élszámú Erdős–Rényi hálózatra.

Milyen értékeket kapunk? Ha választhatunk, hogy egy véletlenül választott embert oltunk be influenza ellen, vagy egy véletlenszerűen választott ember véletlenszerűen választott ismerősét, melyiket választjuk? Egyáltalán lényeges-e választani?

(Hasznos lehet a random modul choice függvénye.)

8.2. feladat {} Készítsünk hálózatot a rátermettségi modell (fitness model) szerint!

Keressük meg a modellt az összetett hálózatok weboldalon található cikkekben! (Albert–Barabási, 2002)

Pluszpontért előadás tartható az elért eredményről.

9. A csoporterősségi együttható

9.1. Fogalmak, elmélet

- csoporterősségi együttható (clustering coefficient, csúcsé és hálózaté), az igraph-ban transitivity.
- várható értéke véletlen (Erdős–Rényi) hálózatban
- csoporterősség–fokszám függvény

9.2. Tudnivalók a programozáshoz

- csoporterősségi együttható az igraphban
  net.transitivity_local_undirected() Ez az általunk használt clustering coeff. egyes csúcsokra.
  net.transitivity_avglocal_undirected() Ez majdnem az általunk használt clustering coeff, de csak arra átlagol, ahol legalább két szomszéd van.
- argumentumok alapértelmezett értéke és a `None`
- különbség az `==` és az `is` feltételvizsgálat között

9.3. Feladatok

9.1. feladat {}

a) Hozzunk létre egy metódust, amely kiszámítja egy hálózat egy csúcsának csoporterősségi együtthatóját!

`net.clustering_coeff(vertex)` $\longrightarrow$
    `a csúcs csoporterősségi együtthatója`

b) Változtassuk meg a függvényt úgy, hogyha a csúcs nincs megadva, akkor a teljes hálózat csoporterősségi együtthatóját számítja ki!

`net.clustering_coeff(vertex=None)` $\longrightarrow$
    `a kívánt csoporterősségi együttható`

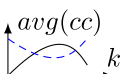
9.2. feladat {}

a) Az igraph saját csoporterősségi együtthatót számoló függvényét felhasználva írjunk olyan függvényt, amely a fokszám függvényében meghatározza az átlagos csoporterősségi együtthatót!

`net.cc_degree()` $\longrightarrow$
    $((k1, k2, k3, k4, \ldots, kmax), (cc1, cc2, cc3, cc4, \ldots, ccmax))$

ki a létező fokszámok növekvő sorrendben (i-dik), cci: az i-dik fokszámhoz tartozó csúcsok csoporterősségi együtthatóinak átlaga.

b) Ábrázoljuk a csoporterősségi együttható–fokszám függvényt a csomagfüggőségi hálózat esetén, és a hozzá legközelebb álló *módosított* Barabási–Albert modellből származó hálózat esetén! Válasszunk a tengelyeken megfelelő skálát! Elemezzük az eltéréseket a két ábra között!



c) Ábrázoljuk a világháló (www) mint hivatkozási hálózat esetén a fenti függvényt! Hasonlítsuk össze az eddig kapott grafikonokkal! A cxnet csomagban van hálózatadatok letöltését segítő függvény.

10. A multifraktálokon alapuló hálózatgenerátor

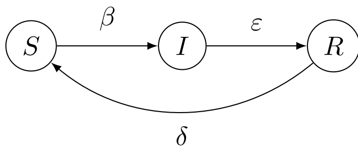
Multifractal network generator, MFNG

Az elméleti háttér a `2011mfng_ais_beamer.pdf` és `halozatok_mfng.pdf` fájlokban, a megvalósításáról a cxnet dokumentációjában.

11. Terjedésmodellek, fertőzések terjedése

11.1. Fogalmak, elmélet

- állapotok: fertőzésre érzékeny, fertőző, fertőzést legyőzte (subsceptible, infective, recovered, a kezdőbetűket elég tudni)
- SI, SIR, SIS és SIRS modellek
- kritikus pont létezése és hiánya



11.2. Tudnivalók a programozáshoz

-
-

11.3. Feladatok

11.1. feladat {} Hozzunk létre egy programot, amely $n \times n$-es négyzetrácson modellezi a fertőzés terjedését SIR modell szerint adott β [1/nap] fertőzésterjedési valószínűséggel és adott r napig tartó fertőzési állapottal!

Pl. $n = 200, \quad \beta = 0{,}1, \quad r = 7$.

Határozzuk meg, hogy adott n és r esetén van-e olyan kritikus érték β-ra, amely alatt a hálózat kicsi része lesz fertőzött, felette pedig majdnem minden csúcspont!

11.4. A network-evolution modul használata

- program futtatása Linux (UNIX) szerveren nohup-pal
- a mért jellemzők tárolása a shelves modullal
- a gyorsítás lehetséges módjai: pyrex és C/C++ modulok, pypy, jobb algoritmus, tisztán C-ben megírni, más adatstruktúra használata

Tartalomjegyzék

1. Python programozási alapismeretek **1**

2. Hálózatok alapfogalmai és az igraph használata **5**
2.1. Fogalmak, elmélet . . . 5
2.2. Tudnivalók a programozáshoz . . . 5
2.3. Tudnivalók a feladatokhoz 1. (a továbbiakra nézve is) . . . 6
2.4. Feladatok . . . 6
2.5. Függvények írása . . . 7
2.6. Metódusok írása hálózathoz . . . 7
2.7. Unittest az előzőhöz . . . 8
2.8. Hálózatok, éleik és csúcsaik attribútumai . . . 9
2.9. Fokszám, szomszédok, szűrés, vs/es attribútumai . . . 9

3. Erdős–Rényi modell **10**
3.1. Fogalmak, elmélet . . . 10
3.2. Tudnivalók a programozáshoz . . . 10
3.3. Tudnivalók a feladatokhoz 2. (a továbbiakra nézve is) . . . 10
3.4. Feladatok . . . 10

4. A Price-modell, a Barabási–Albert-modell és a fokszámeloszlás **14**
4.1. Fogalmak, elmélet . . . 14
4.2. Tudnivalók a programozáshoz . . . 15
4.3. Feladatok . . . 15

5. Egy valódi hálózat: a csomagfüggőségi hálózat **16**
5.1. Fogalmak, elmélet . . . 16
5.2. Tudnivalók a programozáshoz . . . 16
5.3. Feladatok . . . 16

6. Véletlen meghibásodások és támadások hatása **17**
6.1. Fogalmak, elmélet . . . 17
6.2. Feladatok . . . 17

7. Gráfelméleti adatstruktúrák és hálózatok tárolása fájlokban **18**
7.1. Fogalmak, elmélet . . . 18
7.2. Versenyfeladatok . . . 18
7.3. Elméleti összefoglaló . . . 19
7.3.1. Szomszédsági mátrix (adjacency matrix) . . . 19
7.3.2. Szomszédsági lista . . . 20
7.3.3. Élpárok tárolása (indexelés nélkül) . . . 20
7.3.4. Adatstruktúrák az igraph-ban. . . . 21
7.3.5. A legfontosabb tudnivalók táblázatos összefoglalása . . . 21

8. A fokszámeloszlás simítása és a kitevő becslése, egyéb modellek **22**
8.1. Fogalmak, elmélet . . . 22
8.2. Tudnivalók a programozáshoz . . . 22
8.3. Feladatok . . . 22

9. A csoporterősségi együttható **23**
9.1. Fogalmak, elmélet . . . 23
9.2. Tudnivalók a programozáshoz . . . 23
9.3. Feladatok . . . 23

10. A multifraktálokon alapuló hálózatgenerátor **24**

11. Terjedésmodellek, fertőzések terjedése **25**
11.1. Fogalmak, elmélet 25
11.2. Tudnivalók a programozáshoz 25
11.3. Feladatok 25
11.4. A network-evolution modul használata 25