Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Abt Dávid Ottó

2022

Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Másodrendű sajátvektor centralitások vizsgálata

Szakdolgozat

Készítette:
Abt Dávid Ottó
programtervező informatikus BSc
hallgató

Témavezető:
Dr. Vinkó Tamás
egyetemi docens

Szeged

2022

Feladatkiírás

Hálózatok központisági értéke egy olyan függvény, amely a gráf csúcsaihoz rendel egy vagy több valós számot.

Több száz ilyen központisági értéket definiáltak már. Ezek általános jellemzője, hogy lényegében csúcspárok közötti összefüggéseken alapszanak, így ezek tekinthetőek elsőrendű centralitásoknak. A triviális fokszám központiság után talán az egyik legklasszikusabb fogalom a sajátvektor központiság.

A szakirodalomban nemrégiben több olyan cikk is megjelent, amelyekben magasabb rendű központisági fogalmakat definiáltak. [2] Egy lehetséges ilyen kivitelezés az, amelyben a szomszédsági mátrix helyett tenzorokat használunk.

A hallgató feladata, hogy felfedezze a másodrendű sajátvektor központiság világát. Elkészítendő egy szimulációs környezet, amelyben lehetőségünk van különféle tenzorok használatával többféle variánst is használni. Egy szemléletes demonstrációs példa elkészítése is feladat.

Tartalmi összefoglaló

A téma megnevezése

Másodrendű sajátvektor centralitások vizsgálata

A megadott feladat megfogalmazása

Egy szimulációs program elkészítse, amely képes másodrendű sajátvektor központiságok kiszámítására tetszőleges gráf és adott paraméterek esetén, képes az eredmények összehasonlítására, valamint adott krtiériumnak megfelelő gráf keresésére.

A megoldásmód

Az általános sajátvektor modell egyenlete definiálja a sajátvektor centralitás problémát, amely megoldásának a közelítésére alkalmas a hatványmódszer. Az eredményként kapott vektorok elemeinek sorrendjének az összehasonlítására a Kendall rangkorrelációs együttható kézenfekvő eszköz.

Alkalmazott eszközök, módszerek

A fejlesztés a *VSCodium* szövegszerkesztő segítségével zajlott python nyelven, a *NetworkX, NumPy, Matplotlib* és egyéb matematikai függvénykönyvtárak használatával. A program számára minta adatok a Netwrok Repository weboldalról lettek letöltve.

Elért eredmények

Egy könnyen kezelhető parancssori alkalmazás, amely tetszőleges gráf másodrendű sajátvektor centralitásainak vizsgálatára, összehasonlítására, valamint a feltételeknek megadott gráf keresésére alkalmas.

Kulcsszavak

gráf, mátrix, tenzor, sajátvektor, centralitás, másodrendűség, NetworkX

Tartalomjegyzék

1. fejezet

Bevezetés

A technológia és az internet fejlődésével nagymértékben megnőtt az emberi interakciók száma a különböző online hívás- és chatszolgáltatásoknak, valamint közösségi médiai platformoknak köszönhetően. Sokkal több emberrel lépünk kapcsolatba minden nap, mint néhány évtizeddel ezelőtt.

Nem csak az online emberi kapcsolatok mennyiségében történt meg ez a hirtelen növekedés az utóbbi harminc évben, hanem például a számítógép hálózatok esetén is: ma már több mint 10 milliárd eszköz csatlakozik az internetre és kommunikál egymással, ami több, mint ahány ember él jelenleg a Földön.

Ezeknek az interakcióknak a formalizálásához, tárolásához, illetve a rajtuk végzett számításokhoz újfajta modellekre illetve módszerekre volt szükség. Itt gyakran nem magukon az entitásokon van a hangsúly, hanem a köztük létrejött kapcsolatokon. Az ilyen fajta adatok leírására talán a legalkalmasabb eszköz egy gráf.

Egy gráfnak sok triviális tulajdonsága van, mint például a fokszáma, éleinek száma, illetve utóbbiak aránya, amivel egyszerű, de látványos kimutatásokat lehet készíteni. Azonban magas szintű, komplex kérdések megválaszolásához, vizsgálatokhoz nem elegendő ezeket az atomi tulajdonságokat kielemezni. Például egy baráti társaságban ki az összetartó kapocs? Az, aki a legtöbb embert közvetlenül ismeri? Az akit a legtöbben ismernek? Az aki ha kiesne a társasából, akkor sok olyan ember lenne, akinek nincs közös ismerőse?

Sokan a mesterséges intelligencia felől, mélytanulással, neuronhálókkal közelítik meg az efféle problémákat. Azonban ezek a módszerek gyakran feketedobozként viselkednek, nehéz velük konkrét tulajdonságokat kinyerni egy gráfból.

Egy másik megközelítés a centralitások használata. Ez nem olyan elhíresült kifejezés manapság, mint az AI, de komoly feladatok elvégzésében nyújtanak segítséget, mint például a Google keresésekben a találatok relevanciájának kiszámításában. A centralitás gyakorlatilag egy gráf csúcsaihoz értékeket rendel. Ennek is vannak triviális változatai, mint például a fokszámcentralitás, ami minden csúcshoz a fokszámát rendeli, azonban ennél sokkal összetettebb, akár kettőnél több csúcs kapcsolatát magába foglaló tulajdonságok is léteznek, ezeknek a vizsgálatával foglalkozunk a továbbiakban.

2. fejezet

Alapfogalmak

2.1. Gráf

Egy gráf definiálható egy $G = (V, E)$ párként, ahol V a gráf csúcsainak, E az éleinek halmaza. Az E halmaz V-beli elempárokat tartalmaz. [3] ($E \subseteq V \times V$)

Ez a matematikai struktúra alkalmas különböző entitások közti bináris kapcsolatok tárolására. Ezek lehetnek akár egy szociális médiai platform felhasználói közti ismeretségek, vagy egy internetes enciklopédia kulcsszavai közti összefüggések. A gráfok ezen tulajdonságát használja ki például a *neo4j* nevű gráf adatbázis kezelő rendszer is.

2.2. Mátrix

A mátrix egy kétdimenziós listaként képzelhető el, ami nem csupán egy $m*n$ elemet tároló struktúra, hanem az elemek elhelyezkedésének köszönhetően jóval több információt tárol mint egy $m * n$ elemű halmaz.

2.3. Gráf ábrázolások, mátrix

A gráfokat ábrázolására a két legnépszerűbb adatszerkezet a szomszédsági lista, illetve a szomszédsági mátrix.

A másodrendű sajátvektor centralitások vizsgálatához a szomszédsági mátrixokat fogjuk használni, mivel mátrixokon jóval könnyebb az ezekhez szükséges műveleteket elvégezni mint láncolt listákon.

A szomszédsági mátrix egy $n \times n$-es mátrix, ahol n a gráf csúcsainak száma. Az m-edik sor n-edik eleme megadja, hogy a gráf m-edik és n-edig csúcsa közt vezet-e él.

2.4. Sajátérték, sajátvektor

A mátrixokat felfoghatjuk egy geometriai transzformációként is amit egy vektoron végzünk el. Például az **A** mátrix a $\vec{v}$ vektort az alábbiak szerint módosítja:

$$\mathbf{A}\vec{v} = \vec{w}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a * x + b * y \\ c * x + d * y \end{bmatrix}$$

Ha a kapott $\vec{w}$ vektor a kiinudlási $\vec{v}$ vektornak skalárszorosa, akkor $\vec{v}$ vektor az **A** mátrix egy sajátvektora, a skalár szorzó pedig a hozzá tartozó sajátérték.

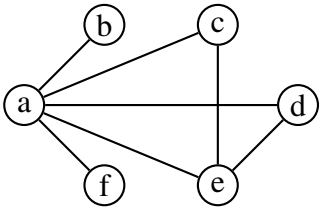
A transzformáció irányát és mértékét a sajátvektor és a sajátérték jellemzi. Definíció szerint A mátrix v sajátétvektora és λ sajátértéke közt az összefüggés:

$$A\lambda = v\lambda, \quad ahol\ \vec{v} \neq \vec{0}, \lambda \in \mathbb{C}$$

2.5. Centralitás

A gráfok atomi tulajdonságain (pl.: van-e él két csúcs között) kívül sok értékes információ kinyerhető különböző metrikákkal. Például a csúcsok ”fontosságának” egy kimutatására szolgálnak a centralitások, amik egy sorrendet képeznek a gráf pontjai közt. [1]

A legegyszerűbb centralitás a fokszám centralitás, ami az adott csúcs fokszáma szerint képez sorrendet.



$$\rightarrow \qquad \begin{bmatrix} deg(a) \\ deg(b) \\ deg(c) \\ deg(d) \\ deg(e) \\ deg(f) \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 2 \\ 3 \\ 1 \end{bmatrix} \tag{2.1}$$

A ábrán szereplő 6 pontú gráfhoz egy 6 hosszúságú vektort rendelt, melynek elemei az egyes csúcsok fokszámával egyeznek meg.

A fokszám centralitás kiszámítható gráfot leíró $n \times n$ szomszédsági mátrix valamint egy n elemű, egyesekből álló vektor szorzataként:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 2 \\ 3 \\ 1 \end{bmatrix} \tag{2.2}$$

2.6. Tenzor

Egy skalár egy szám, egy n elemű vektort n rendezett érték, egy $n \times m$-es mátrix $n * m$ számstruktúra tárolására képes. Ezeket a matematikai fogalmakat általánosíthatjuk a tenzorokkal. A skalár egy nullad-, a vektor egy első-, a mátrix pedig egy másodrendű tenzornak tekinthető.

Amiért igazán fontos behozni a tenzor fogalmát az az, hogy nem állunk meg a másodrendűségnél, léteznek magasabb rendű tenzorok is. Például egy $3 \times 3 \times 3$-as tenzor 27 elemből áll, és felfogható egy három dimenziós tömbként.

3. fejezet

Másodrendű sajátvektor centralitások

3.1. Elsőrendű centralitás

A korábban látott fokszám centralitás egy elsőrendő centralitás volt, mivel fokszám az adott csúcshoz kapcsolódó élek számát jelenti, egy él pedig két csúcs között teremt kapcsolatot.

Az elsőrendű centralitások kiszámításához elég volt egy mátrixszorzást végrehajtani egy vektoron, hiszen a mátrix felfogható egy kétdimenziós tömbként is, aminek az $A[i][j]$-edig eleme jellemzi az összefüggést az i-edik és j-edik csúcs között.

3.2. Másodrendű centralitás

A másodrendű centralitás ezzel szemben csúcshármasok közti kapcsolatok alapján fogja jellemezni a gráfot. Három-három csúcs viszonyának tárolására a mátrixok nem elegendőek, harmadrendű tenzorok viszont igen. Ezt fogjuk felhasználni a következőkben.

3.3. Általános sajátvektor modell

A másodrendű centralitások számításához egy operátorra lesz szükség, amely végrehajtja a megfelelő leképezést.

Egy $\boldsymbol{T} \in \mathbb{R}^{n \times n \times n}$ tenzorhoz és egy p valós paraméterhez definiálunk egy $\boldsymbol{T_p} : \mathbb{R}^n \to \mathbb{R}^n$ operátort, ami egy n elemű vektorhoz egy n elemű vektort rendel. Az eredményvektor i-edik elemét a következőképp kaphatjuk meg:

$$\boldsymbol{T_p}(\boldsymbol{x})_i = \sum_{j,k=1}^{n} \boldsymbol{T}_{ijk} \mu_p(x_j, x_k)$$

ahol $\mu_p(x_j, x_k)$ a hatványközép, azaz:

$$\mu_p(x_j, x_k) = \left(\frac{|a|^p + |b|^p}{2} \right)^{1/p}$$

Ahhoz, hogy hatékonyan vizsgálatokat tudjunk végezni a különböző centralitásokon szükségünk lesz egy egységes kiszámítási módra.

Legyen $\alpha \in \mathbb{R} : 0 \leq \alpha \leq 1$ skalár, ami az első- és másodrendűség lineáris kombinációjának az arányát adja meg, illetve legyen $M \in \mathbb{R}^{n \times n}$ nemnegatív elemekből álló, a gráfhoz tartozó négyzetes mátrix, valamint $\boldsymbol{T} \in \mathbb{R}^{n \times n \times n}$ szintén nemnegatív elemegkből álló, gráfot jellemző tenzor. Ekkor definiálhatunk egy $\mathcal{M} : \mathbb{R}^n \to \mathbb{R}^n$ operátort, hogy

$$\mathcal{M}(\boldsymbol{x}) = \alpha M \boldsymbol{x} + (1 - \alpha)\boldsymbol{T}_p(\boldsymbol{x})$$

Ekkor az első- és a másodrendű centralitás i-edik csúcsbeli értéke $x_i \geq 0$ ahol x kielégíti a sajátérték problémát:

$$\mathcal{M}(\boldsymbol{x}) = \lambda \boldsymbol{x}$$

α megfelelő megválasztásával kiszámíthatjuk csak az elsőrendű centrailtás értéket ($\alpha = 1$), vagy csak a másodrendű centralitás értéket ($\alpha = 0$), illetve a kettő súlyozott kombinációját is.

3.4. M és $\boldsymbol{T}$ megválasztása

Az M mátrix az elsőrendű összetevője, tehát valamilyen elsőrendű információt kellene elkódolnia. Erre egy kézenfekvő lehetőség a szomszédsági mátrix.

A fontosabb szerepet pedig a $\boldsymbol{T}$ tenzor tölti be, hiszen ez tartalmazza a vizsgálandó másodrendű kapcsolatokat.

Bináris háromszögtenzor

A másodrendű kapcsolatoknak egyik legjobb példája, ha az adott három csúcs egy háromszöget zár be, azaz páronként össze vannak kötve egy-egy éllel. Ennek az elkódolásához egy olyan tenzorra van szükségünk, amelynek az első dimenziója szerinti $i.$, második dimenzója mentén $j.$, harmadik dimenziója mentén $k.$ elem akkor 1, ha a gráfnak az $i., j., k.,$ csúcsa egy háromszöget alkot, különben 0:

$$(\boldsymbol{T}_B)_{ijk} = \begin{cases} 1 & \text{ha } i, j, k \text{ csúcsok között páronként szerepel él} \\ 0 & \text{különben} \end{cases} \tag{3.1}$$

Random séta háromszögtenzor

A random séta háromszögtenzor egy normalizált változata a bináris háromszögtenzorznak, azzal a trükkel, hogy ha i, j, k egy háromszöget alkot, akkor nem 1 lesz a tenzor adott értékének az eleme, hanem a (j, k) élet tartalmazó háromszögek számának a reciproka. Egy adott (j, k) élet tartalmazó gráfbeli csúcsháromszögek számát könnyen ki lehet számolni a szomszédsági mátrix alapján:

$$\Delta(j, k) = (A \circ A^2)_{jk}$$

ahol a $\circ$ művelet a mátrixok elemenkénti szorzását jelöli.

A kapott tenzor ezek alapján:

$$(\boldsymbol{T}_W)_{ijk} = \begin{cases} \frac{1}{\Delta(j,k)} & \text{ha } i, j, k \text{ csúcsok között páronként szerepel él} \\ 0 & \text{különben} \end{cases} \tag{3.2}$$

Fontos megemlíteni, hogy $\boldsymbol{T}_W$ jól definiált, hiszen ha i, j, k csúcsok háromszöget alkotnak, akkor $\Delta(j, k)$ legalább 1.

Klaszterező-együttható háromszögtenzor

Egy másik megközelítés lehet, ha a csúcshármasoktól nem várjuk el, hogy alkossanak háromszöget, hanem csak annyi, hogy egy ”szeletben” vegyenek részt, azaz legalább kettő élhez kapcsolódjanak. Tehát az i csúcsot érintő háromszögek számának reciproka helyett az i csúcsból kiinduló irányítatlan szeletek számának reciprokát vesszük alapul.

Ha d_i az $i.$ csúcs fokszámát jelöli, akkor $d_i(d_i - 1)$ szorzat pont a keresett értéket adja meg. Ezek alapján a tenzor:

$$(\boldsymbol{T}_C)_{ijk} = \begin{cases} \frac{1}{d_i(d_i-1)} & \text{ha } i, j, k \text{ csúcsok között páronként szerepel él} \\ 0 & \text{különben} \end{cases} \tag{3.3}$$

Ezesetben is $\boldsymbol{T}_C$ jól definiált, mivel ha i, j, k csúcsok háromszöget alkotnak, akkor $d_i(d_i - 1)$ legalább 2.

Local closure háromszögtenzor

Az utolsó tenzor, amit vizsgálni fogunk ismét egy másik megközelítése a másodrendű kapcsolatoknak. Most az adott csúcsból kiinduló kettő hosszúságú utakat vesszük figyelembe a számításnál a háromszögek illetve a szeletek keresése helyett. Ezt megkaphatjuk úgy, hogy összeadjuk az adott csúccsal szomszédos csúcsok fokszámát, és kivonjuk belőle az adott csúcs fokszámát. Vagy pedig összeadjuk az adott csúccsal szomszédos csúcsok fokszámánál egyel kisebb értékeket:

$$w(i) = \sum_{j \in N(i)} d_j - d_i = \sum_{j \in N(i)} (d_j - 1)$$

ahol $N(i)$ az i-vel szomszédos mezők halmazát jelöli.

Ebben az esetben a tenzor a következő:

$$(\boldsymbol{T}_L)_{ijk} = \begin{cases} \frac{1}{w(i)} & \text{ha } i, j, k \text{ csúcsok között páronként szerepel él} \\ 0 & \text{különben} \end{cases} \tag{3.4}$$

Amely szintén jól definiált, hiszen ha i, j, k háromszöget alkot, akkor biztosan létezik nullánál több kettő hosszúságú i-ből kiinduló út a gráfban.

4. fejezet

Implementáció

4.1. Fejlesztői környezet

A programot GNU/Linux operációs rendszer alatt VSCodium-ban fejlesztettem (Visual Studio Code nyílt forráskódú és szabad változa). A python nyelvet [7] választottam, ami egy széles körben elterjedt szkriptnyelv, egyszerű szintaxissal, és sokrétű matematikai függvénykönyvtárakkal rendelkezik, így a célnak tökéletesen megfelel.

A NetworkX [6] a használt csomagok közül a legfontosabb, a gráfok beolvasását, kezelését, illetve a rajtuk elvégzett műveleteket is támogatja. Ezen kívül említésre méltó még a NumPy és a Pandas, amiket mátrixműveletek kiszámítására használtam, valamint az argparse, amit parancssori argumentumok beolvasásában segített, illetve a Matplotlib, amivel az eredményeket kirajzoltattam.

4.2. A program magja

Ahhoz, hogy gráfok centralitását összehasonlítsuk, először ki kell tudni számolni őket. Az első lépés ennek az implementációja volt. Bemenetként egy gráfot kapunk, és az elvárt kimenet a $\mathcal{M}(\boldsymbol{x}) = \lambda \boldsymbol{x}$ egyenlet egy közelítő megoldása.

Az egyenlet kiszámításához egy iterációs módszerre lesz szükségünk, a hatványmódszerre. Ez az eljárás iteratív lépések segítségével a legnagyobb abszolútértékű sajátértékhez és a hozzá tartozó sajátvektorhoz ad közelítő megoldást. [4] Az algoritmus alapja ez az iteráció:

$$\boldsymbol{y}^{(k)} = A\boldsymbol{x}^{(k)}$$

$$\boldsymbol{x}^{(k+1)} = \frac{\boldsymbol{y}^{(k)}}{||\boldsymbol{y}^{(k)}||}$$

ahol az első egyenlet egy mátrix-vektor szorzás, a második pedig ennek a részeredménynek a normált értékét adja meg. A kiinudlási ($\boldsymbol{x}^{(0)}$) vektor nem lehet merőleges a legnagyobb abszolútértékű sajátértékhez tartozó sajátvektorra.

A mi esetünkben ez a következőképpen néz ki:

$$\boldsymbol{y}^{(k)} = \mathcal{M}(\boldsymbol{x})$$
$$\boldsymbol{x}^{(k+1)} = \frac{\boldsymbol{y}^{(k)}}{||\boldsymbol{y}^{(k)}||}$$

vagyis

$$\boldsymbol{y}^{(k)} = \alpha M\boldsymbol{x} + (1-\alpha)\boldsymbol{T}_p(\boldsymbol{x})$$
$$\boldsymbol{x}^{(k+1)} = \frac{\boldsymbol{y}^{(k)}}{||\boldsymbol{y}^{(k)}||}$$

4.3. Az egyenlet paraméterei

A kiszámítás előtt szükségünk lesz az α-ra, az M-re, illetve $\boldsymbol{T_p}$-re, és a hozzá szükséges tenzorra. Alfa egyszerű skalár paraméter, M-et, mint szomszédsági mátrixot pedig megkaphatjuk a NetworkX `adjacency_matrix` nevű függvényével.

A tenzorok előállítását saját függvényekkel oldottam meg. Az input itt is egy gráf (NetworkX-es `Graph` típusban eltárolva), és hátrom egymásba ágyazott for ciklus tölti fel a három dimenziós tömbként tárolt tenzorokat. Ezek a függvények felépítésükben nagyban hasonlítanak, egyedül a tenzor értékét adó logika tér el. Itt említést érdemel a NumPy csomag `ndarray, multiply, matmul` és `ones` függvényei, amik sok for ciklus megírását spórolták meg, ezzel rövidebbé, és átláthatóbbá téve a kódot.

A $\boldsymbol{T_p}$ operátornak egy három paraméteres függvényt írtam, ami a tenzort, az $\boldsymbol{x}$ vektort, illetve p-t kapja inputként. A vektort itt egy hagyományos pyhton-os listaként kezeltem.

Még egy fontos paraméter, ami az egyenletben nincs explicit jelölve, az az iterációk száma. Mivel iterációs módszerről beszélünk, így meg kell adni egy egész értéket is, hogy hány iteráció után szeretnénk visszatérni az eredménnyel. Minél tovább iterálunk, annál pontosabb eredményt kapunk, azonban annál több erőforrást és időt vesz igénybe a program.

$\boldsymbol{x}$ kezdeti értékét eleinte egy random n hosszúságú vektorra állítottam be, aztán a determinisztikus működés miatt áttértem az egyesekből álló vektorra, így könnyebb volt a későbbiekben ellenőrizni a program helyességét.

4.4. Teszt adatok

A NetworkX a gráfokkal kapcsolatos függvényeken és eljárásokon kívül tartalmaz néhány gráfot is, például Zachary karate klubjának a gráfját [8]. Én is ezt használom a teszteléshez,

illetve a program ezzel számol alapértelmezett gráfként, ha nem adunk meg mást.

Egyéb gráfok beszerzésére a Netwrok Repository [5] oldalt használtam, ahol több ezer gráf áll rendelkezésre ingyenesen. Igyekeztem kisebb fokszámú ($|V| < 100$), de nem triviális gráfokat keresni.

4.5. A keretprogram

A szoftvert egy mérésre, kísérletezésre alkalmas eszközként képzeltem el, amiben a matematikai számításokon van a hangsúly, nem pedig a felhasználói felületen. Így a Unix filozófiához hűen egyszerű parancssori alkalmazásban gondolkoztam, ami „egy dolgot csinál, de azt jól csinálja”.

A program fejlesztésének első fázisában probléma volt, hogy az aktuális haladást mindig ki akartam próbálni, esetleg az eredményeket képeken ábrázolni, és aztán változtatni rajta, újra kipróbálni. Erre kézenfekvő megoldás volt, hogy az alkalmazásnak a számításhoz szükséges bemeneti paramétereken kívül meg lehessen adni, hogy milyen funkciót végezzen el. Így ha valami újfajta mérést szerettem volna leimplementálni, akkor elég volt létrehozni neki egy új funkciót, amit a helyes paraméterezés esetén végrehajt.

4.6. Parancssori argumentumok

A program futtatásához kell egy funkciót megadni parancssori paraméter formájában, amivel meghatározzuk, hogy milyen számítást hajtson végre a program. Ez az egyetlen kötelező paraméter.
Például:

```
python main.py solve
```

ahol a `solve` funkciót szeretnénk elvégeztetni a programmal.

Ezen kívül lehetőségünk van a következő paraméterek állítására is:

- `--graph`, röviden `-g`, a gráf megadására, amin számolni szeretnénk
- `--tensor`, röviden `-t`, a tenzor(ok) megadására, amivel a másodrendű kapcsolatokat jellemezzük. (Egyes funkciókhoz több tenzort is meg lehet adni.)
- `--alpha`, röviden `-a`, az egyenlet α paramétere
- `-p` a gráf megadására, az egyenlet p paramétere
- `--num_iter`, röviden `-n`, a hatványmódszerben az elvégezendő iterációk szám

- `--treshold`, röviden `-x`, a `find-similar` funkciókban az optimális hasonlóság mértéke

Ezeket a paramétereket, és a hozzájuk tartozó rövid magyarázatot a

```
python main.py --help
```

utasítással le tudjuk kérdezni. Példa egy paraméterezésre:

```
python src/main.py --graph input_graphs/soc-firm-hi-tech.txt --tensor
random_walk --alpha 0.6 -p 2 --num_iter 15 solve
```

5. fejezet

Funkciók

5.1. Solve

Az első funkció, amit a program nagyon korai fázisában implementálva lett, az a `solve`, ami gyakorlatilag kiszámítja az általános sajátvektor modell egyenletét az kapott paraméterek alapján, hatványmódszer segítségével. Az eredményvektort a képernyőre írja ki. Az előző oldalon szereplő példa esetén az output a következő:

```
[0.00840449 0.12371296 0.00840449 0.22103893 0.51571457 0.10745514
0.06594145 0.38736454 0.20666928 0.25458575 0.11321281 0.26577601
0.11070488 0.2408121  0.31856498 0.0964164  0.11184042 0.14277145
0.10582157 0.0454091  0.08996091 0.04470695 0.17091334 0.18363112
0.05318976 0.07146028 0.03518014 0.05129317 0.003085   0.02035976
0.01828195 0.00239007 0.00617164]
```

5.2. Compare

A centralitások vizsgálatának az egyik legkézenfekvőbb formája az összehasonlítás. Minden centralitás a gráf valamely jellemzője alapján felállít egy sorrendet a csúcsok között a hozzájuk rendelt értékekkel. Ezeknek a sorrendeknek összehasonlításához a Kendall rangkorrelációs együtthatót (röviden: Kendall tau) vesszük igénybe, ami egy $x \in \mathbb{R} : 0 \leq x \leq 1$ értékkel jellemzi a centralitások eredményeként kapott vektorok hasonlóságát a sorrendiség szempontjából.

A Kendall tau kiszámításához az `scipy` csomag `stats` alkönyvtárának a `kendalltau` függvényét használjuk fel, ami a két összehasonlítandó vektort várja paraméterként, és a Kendall rangkorrelációs együtthatóval, valamint a p-értékkel tér vissza.

A program képes egyszerre akár kettőnél több tenzorhoz tartozó centrailtás összehasonlítására is a `compare` funkcióval. Az eredmény egy táblázat, aminek első sorában és oszlopában szerepelnek az összehasonlított centralitásokhoz tartozó tenzorok nevei, az egyes

mezőkben pedig az adott sorhoz és oszlophoz tartozó centralitás korrelációja: a főátló felett a Kendall korrelációs rangegyütthatók, a főátló alatt a p-értékek.
Példa a funkció használatára:

```
python src/main.py compare -t binary -t random_walk -t clustering_coefficient -t local_closure
```

Ehhez a bemenethez tartozó eredmény:

```
p_value \ tau              binary  random_walk  clustering_coefficient  local_closure
binary                1.00000000   0.64349376              0.66131907     0.64349376
random_walk           0.00000009   1.00000000              0.98217469     1.00000000
clustering_coefficient 0.00000004   0.00000000              1.00000000     0.98217469
local_closure         0.00000009   0.00000000              0.00000000     1.00000000
```

5.3. Comparison

Gyakran ha összehasonlítást szeretnénk elvégezni, nem csak egy adott paraméterezéses mérésnek az eredményére vagyunk kíváncsi, mint amit az előző funkció produkál, hanem például több α érték esetén lennénk kíváncsiak a hasonlóságokra. Erre szolgál a `comparison` funkció, ami szintén a megadott tenzorokhoz tartozó centralitásokat hasonlítja össze, azonban itt nem az általunk megadott α-ra, hanem 0-tól, 1-ig 0.1-es lépéssel minden α-ra elvégzi az összehasonlítást.

Ez esetben nem lenne célszerű az eredményt a standart outputon megjeleníteni, így ehelyett inkább egy adatbázisba kerülnek a kapott korrelációs együtthatók. A célnak tökéletesen megfelel az sqlite adatbázis, hiszen relatíve kevés adatot szeretnénk letárolni, és így nem kell semmiféle harmadik féltől származó szoftvert telepíteni az adatbázisba íráshoz.

A program a $comparison$ funkció futtatása esetén létrehoz egy táblát az adatbázisban - ha az még nem létezik - és letárolja a gráf nevét, az aktuális α értéket, p paramétert, illetve a centralitáspárok tartozó korrelációs értékeket.

Például:

```
python src/main.py comparison -p 2 -t degree -t binary -t random_walk -t clustering_coefficient -t local_closure
```

paraméterezés esetén az adatbázisba a következő adatok kerültek:

	id	graph_name	alpha	p	degree_binary	degree_random_walk	degree_clustering_coefficient	degree_local_closure	binary_random_walk	binary_clustering_coefficient	binary_local_closure
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Zachary's Karate Club	0.0	0.0	0.482858848619439	0.611774322389714	-0.0642446602311196	0.3053758153814	0.62857383133016	0.460061599147874	0.735804394306496
2	2	Zachary's Karate Club	0.1	0.0	0.505266980808089	0.746240463962715	0.660733744133654	0.734580456713298	0.443850267379679	0.46524064171123	0.440285204991087
3	3	Zachary's Karate Club	0.2	0.0	0.551907009805758	0.734580456713298	0.699600434965046	0.734580456713298	0.46524064171123	0.486631016042781	0.472370766488414
4	4	Zachary's Karate Club	0.3	0.0	0.614093715135985	0.734580456713298	0.715147111297602	0.734580456713298	0.522281639928699	0.547237076648841	0.529411764705882
5	5	Zachary's Karate Club	0.4	0.0	0.668507082299933	0.730693787630159	0.722920449463881	0.730693787630159	0.590017825311943	0.60427807486631	0.590017825311943

A kép részlet, `sqlitebrowser` alkalmazásban készült. A korrelációs értékeket tároló mezőket a $tenzor1\_tenzor2$ konvenció alapján neveztem el, és a Kendall tau értékét tartalmazzák.

5.4. Find-similar

5.4.1. Keresési probléma

A program eddigi funkciói arra nyújtottak eszközt, hogy meglévő gráfokon végezhessünk méréseket, összehasonlításokat. Azonban gyakran előfordul, hogy olyanra esetekre vagyunk kíváncsiak, amit még nem láttunk konkrét gráfok esetén. Tehát nem a centralitás, illetve a centralitások hasonlósága a kérdés, hanem hogy az általunk kívánt korrelációk milyen gráfok esetén állnak fent.

A `find-similar` funkciócsalád olyan gráfok keresésére szolgál, ahol a megadott sajátvektor centralitások majdnem megegyeznek. Természetesen a triviális gráfokon kívül érdekes ez a kérdés, hiszen például egy teljes gráf esetén a csúcsoknak nincs kitüntetett szerepük, minden centralitás ugyan azt az értéket rendelné a pontokhoz.

5.4.2. Kiindulás

Ahhoz, hogy egy ilyen keresésbe belekezdjünk, szükséges egy kiindulási gráf. A program eddigi adottságait kihasználva ez szükségszerűen lehet egy parancssori argumentumként kapott gráf, hiszen több, tetszőleges kezdőállapotból elindítva a keresést nagyobb valószínűséggel találunk az elvárásainknak megfelelő, vagy legalábbis ahhoz hasonlító gráfot.

5.4.3. Algoritmus

Az alap ötlet az, hogy kiszámítjuk a kezdeti gráfon a sajátvektor centralitások korrelációját, majd változtatunk valamit a gráfon, például egy élet elhagyunk, vagy egy újat hozzáadunk és megint kiszámítjuk a centralitásokat, és a korrelációs együtthatót. Ha az érték jobb (nagyobb) mint az előző, akkor az új gráffal megyünk tovább, ha nem, elvetjük a változtatásokat, és újra próbálkozunk. Ez egy hegymászó algoritmus, azaz kiszámítjuk az adott helyben lévő értéket, lépünk egyet, kiszámítjuk az új értéket. Ha jobb, megtartjuk. Ha rosszabb, visszalépünk. Ennek az egyik előnye az, hogy nincs akkora számítás- és időigénye mintha az összes lehetséges élkombináción végig próbálnánk menni, ami nagyobb gráfok esetén szinte kivitelezhetetlen lenne.

5.4.4. Első megvalósítás

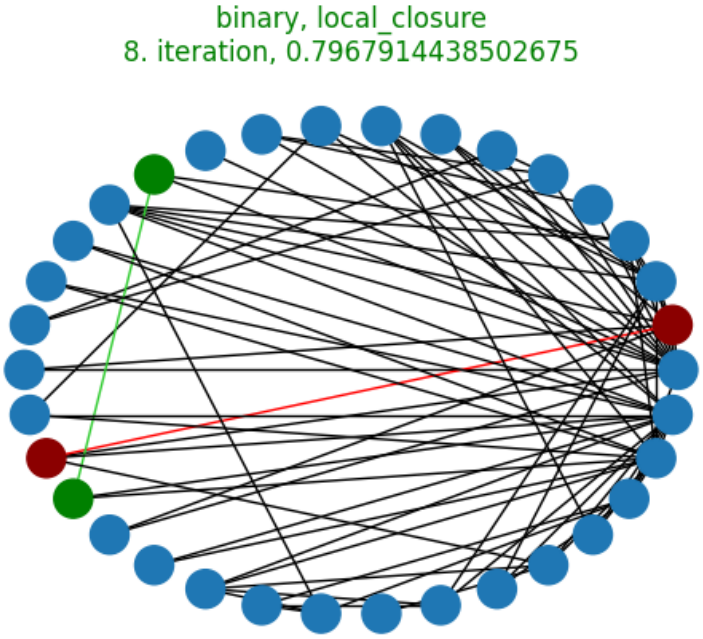
Az algoritmus implementálása során egy hagyományos ciklusos maximuxkeresős megoldás volt a legcélszerűbb, ahol nem egy adott strukrúra értékein végigiterálva történt a keresés, hanem minden iterációban kiszámolásra került az aktuális gráf centralitásainak korrelációja. Kezdetben ez az előző funkciókban már elkészített függvényeket használta. Kettőnél több centralitás összehasonlítása esetén a korrelációk összegének a maximumát kereste az algoritmus.

Minden iterációban 50% eséllyel kitörölt egy élet a gráfból, 50% eséllyel pedig létrehozott egyet két még nem összekötött csúcs között. Az algoritmus természetesen figyelembe vette az aktuális élek számát, tehát például teljes gráf esetén biztosan elvesz élet, nem fut hibára.

5.4.5. A részeredménynek ábrázolása

Az algoritmus futása közben kapott gráfokat, illetve a hozzájuk tartozó értéket nem lenne sem informatív, sem látványos a standard outputon mátrix formában megjeleníteni, így arra a következtetésre jutottam, hogy NetworkX és a Matplotlib beépített függvényeit kihasználva az egyes gráfokat kirajzoltatom. A gráfon végbement változtatások kiemelése érdekében a hozzáadott, illetve elvett csúcsokat a végpontjaikkal együtt zöldre, illetve pirosra színezi a program, hogy emberi szemmel is könnyen látható legyen, hogy mi változott.

Példa:



Ezen a képen a Bináris háromszögtenzorhoz, illetve a Local closure háromszögtenzorhoz tartozó centralitások korrelációja a célfüggvény, a 8. iterációnál (hegymászó lépésnél) tart az algoritmus, és jelenleg 0.7967... a korrelációs együttható értéke.

5.4.6. Javítások

Az első megvalósításban gyakran előfordult, hogy a program egy olyan irányba vitte a kiindulási gráfot, hogy annak az éleinek a száma nagymértékben elkezdett növekedni, vagy épp ellenkezőleg, gyorsan elkezdett csökkenni. Az ilyen lefutások gyakran egy kevésbé természetes, jóval inkább triviális gráfhoz vezettek, ami nem túl ideális eset számunkra.

E probléma kiküszöbölésére az a megoldás született, hogy az algoritmus élszámtartó legyen, azaz mindig pontosan egy élet vegyen el a gráfból (továbbra is figyelembe véve azt, hogy a gráf összefüggő maradjon), illetve egy még nem létező élet adjon hozzá. Ez a megközelítés egy teljesen új gráf alkotása helyett a kiindulási gráf éleit rendezi csak át az optimális megoldást keresve.

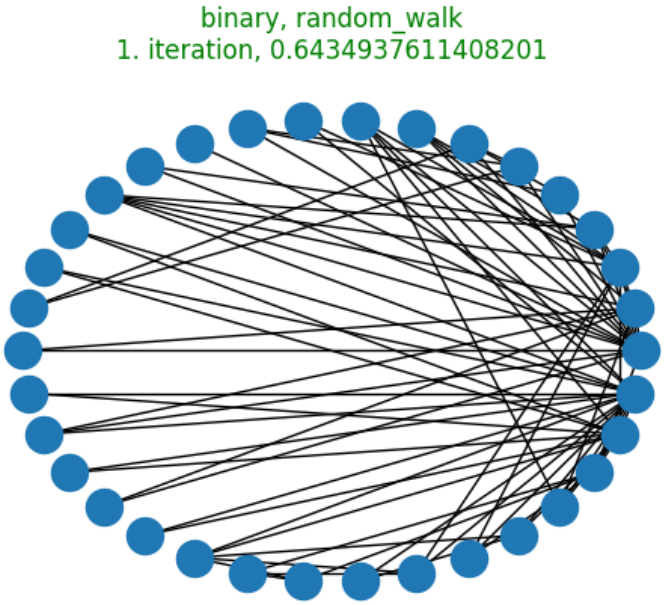
A másik probléma az volt, hogy kettőnél több centralitás esetén az hegymászás nem bizonyult hatékonynak. Kezdetben a program ugyanis a páronként összehasonlított centralitások korrelációs együtthatóinak az összegét maximalizálta. Ezzel probléma lehet, hogy valamely két centrailtás ténylegesen közelebb kerül egymáshoz egy részeredmény gráf esetén, de a többivel való hasonlóság kissé romlik, ami hátráltat az optimális értékhez való eljutásban.

Erre az a megoldás kínálkozott, hogy az egyes centralitások hasonlóságát páronként vizsgáljuk, és páronként történik a hegymászás. Ha a korrelációs együttható eléri az elvárt értéket egy centrailtás pár esetén egy gráfnál, akkor a következő párral folytatjuk az algoritmust.

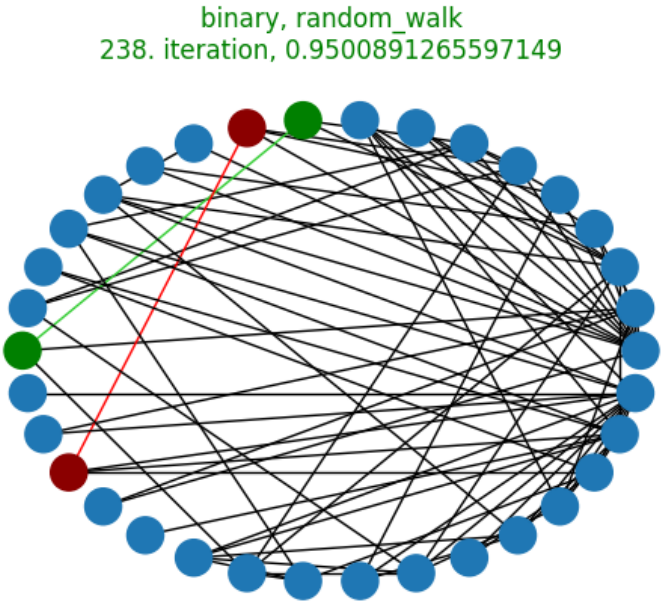
Utóbbi megoldással érezhetően megnövekedett az algoritmus sebessége, olyan értelemben, hogy hamarabb eljutottunk az elvárt korreláció együttható értékig.

5.4.7. Eredmények

Az funckió futtatásának eredményei a Karate klub gráf esetén:
Kiindulási gráf:

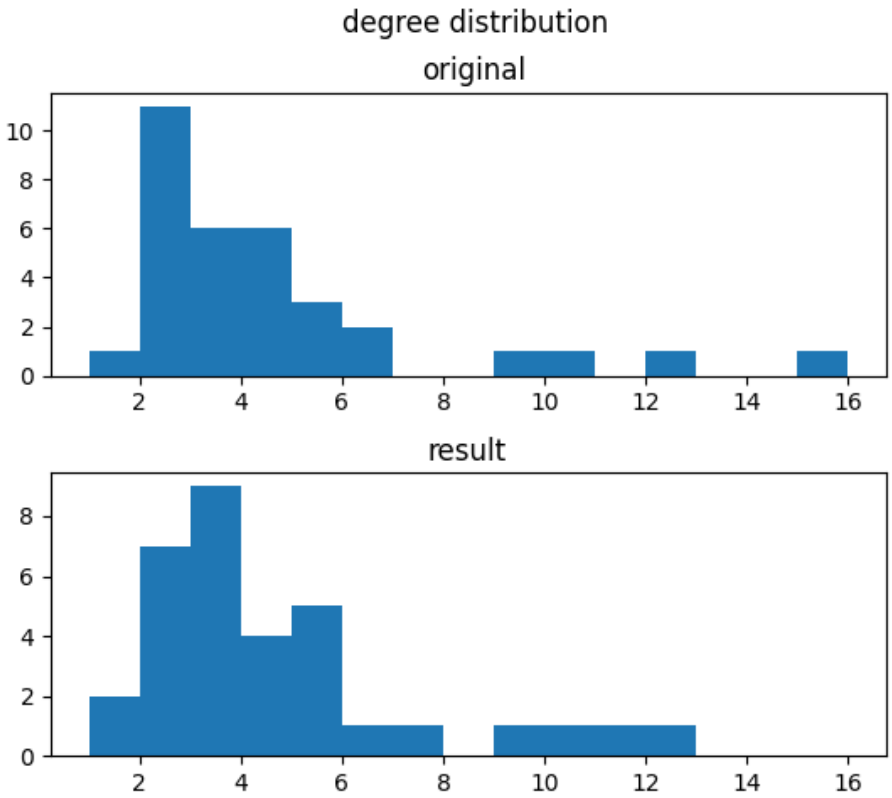


Utolsó iteráció eredménye:

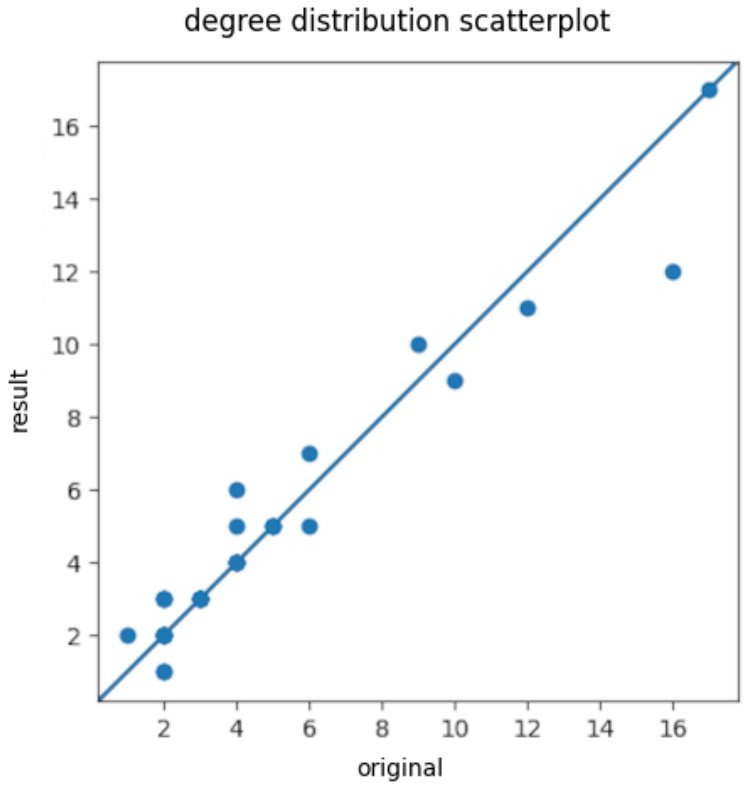


A program a png képeken kívül az iterációkból készített gif-et is.

A kiindulási és az eredményül kapott gráf élszámairól készít hisztogramot is a szoftver:

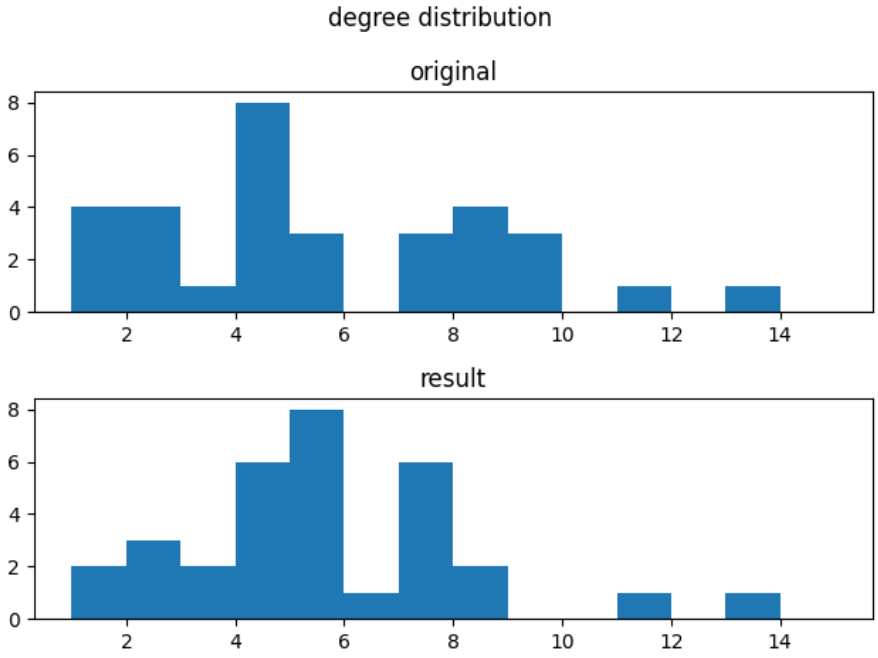


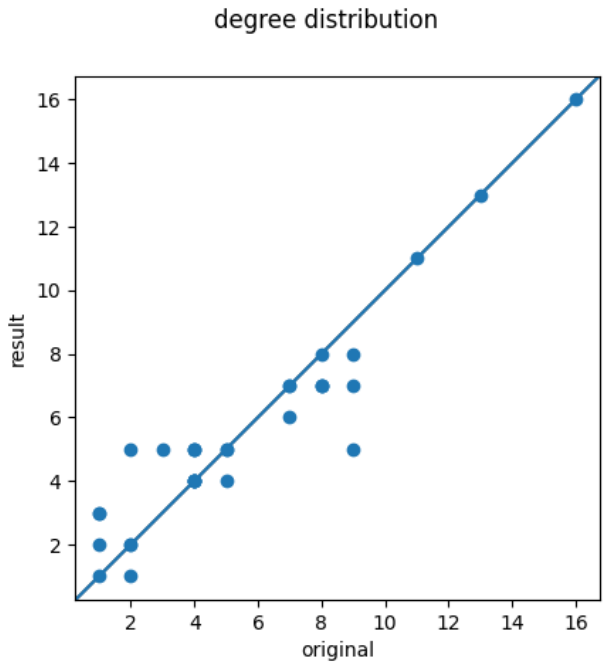
valamint scatterplot-ot is:



A példában használt Karate Klub gráf esetén jól látszik, hogy az eredmény gráfban a kiindulásihoz képest az élszámok egyenletesebben oszlanak el, a nagyobb fokszámú, illetve a nagyok kis (x<3) fokszámú csúcsok száma csökkent, több csúcs esett az eredmény gráfban a 3-5 fokszámú csúcsok halmazába.

Egy másik gráf esetén, ami egy csúcstechnológiai cégen belüli kapcsolatokat tartalmaz, hasonló eredményeket ért el az algoritmus, itt is egyenletesebb lett a csúcsok fokszámainak az eloszlása az eredmény gráf esetén, ahol 97%-ban korrelálnak a sajátvektor centralitások:





6. fejezet

Konklúzió

Összességében tehát egy (a python csomagoktól eltekintve) függőségmentes alkalmazásunk van, ami saját paraméterekkel futtatható a parancssorból, így tehát még grafikus felületet sem igényel.

A végleges funkicók, amik teljes körűen működőképesek:

1. `solve`, az általános sajátvektor modell egyenletének megoldása
2. `compare`, másodrendű sajátvektor centralitások összehasonlítása egy gráfban
3. `comparison`, a `compare` eredménye több α esetén
4. `find-similar`, olyan gráf keresése hegymászó algoritmussal, ahol hasonlóak a másodrendű sajátvektor centralitások eredményei
5. `find-similar-pair`, a `find-similar` funkció, de az algoritmus páronként hasonlítja össze a centralitásokat

Tehát az alkalmazás megfelel az elvárásoknak, képes másodrendű sajátvektor centraiiltások kiszámítására, összehasonlítására, és ezen felül egy hegymászó algoritmust is tud futtatni tetszőleges gráfra, amely adott tulajdonságú gráfot keres.

Irodalom

[1] London András. *Hálózattudomány, 2. előadás*.
URL: http://www.inf.u-szeged.hu/~london/Halozatok/halozat2.pdf.

[2] Francesco Tudisco Francesca Arrigo Desmond J. Higham.
A framework for second order eigenvector centralities and clustering coefficients.
URL: https://arxiv.org/pdf/1910.12711.pdf.

[3] Laczik Sándor János. *MSc Gráfelmélet*. URL: http://www.math.u-szeged.hu/~hajnal/courses/MSc_Grafelmelet/MSc_graf10/ea01.pdf.

[4] Gelle Kitti. *Közelítő és szimbolikus számítások, 7. gyakorlat*.
URL: http://inf.u-szeged.hu/~kgelle/sites/default/files/upload/07_hatvanymodszer.pdf.

[5] *Network Repository*. URL: https://networkrepository.com/.

[6] *NetworkX documentation*.
URL: https://networkx.org/documentation/networkx-1.9/.

[7] *Python 3.10.4 documentation*. URL: https://www.python.org/.

[8] *Zachary's karate club*.
URL: https://en.wikipedia.org/wiki/Zachary%27s_karate_club.

Nyilatkozat

Alulírott Abt Dávid Ottó, Programtervező Informatikus hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Számítógépes Optimalizálás Tanszékén készítettem, alapszakos diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2022. május 10. .....

aláírás

Köszönetnyilvánítás

Szeretném megköszönni a **MIT OpenCourseWare** YouTube csatorna megalkotóinak és **Gilbert Strang** professzornak, hogy közzétették a általa oktatott MIT Linear Algebra előadássorozatot, amely annó felkeltette az érdeklődésemet ezen a szakterületen, valamint a mátrixokkal és tenzorokkal kapcsolatos alaptudást megszerezhettem innen.

Köszönöm a **NetworkRepository** oldal üzemeltetőinek, és azoknak akik hozzájárultak, hogy szinte bármilyen méretű és tulajdonságú gráfra találhattam példát, amit ingyen fel is használhattam a programom tesztelése közben.

Továbbá köszönöm azoknak a **MathOverflow** és **StackOverflow** felhasználóknak a válaszait, akik önszánukból részletes példákon keresztül igyekeztek megválaszolni előttem feltett kérdéseket, ami bennem is megfolgalmazódtak e szakdolgozat készítése közben.