

**SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK**



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Szociális jelenségek vizsgálata egy valós cég email kommunikációs
hálózatán

DIPLOMADOLGOZAT

Témavezetők:
Dr. Kátai Zoltán,
Egyetemi docens
Oltean-Péter Boróka,
Egyetemi tanárseg

Végzős hallgató:
Tankó Tamás

2023

**UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
SPECIALIZAREA INFORMATICĂ**



**UNIVERSITATEA
SAPIENTIA**

Studierea fenomenelor sociale într-o rețea de comunicare prin
email a unei companii reale

LUCRARE DE DIPLOMĂ

Coordonator științific:
Dr. Kátai Zoltán,
Conferențiar universitar
Oltean-Péter Boróka,
Asistent universitar

Absolvent:
Tankó Tamás

2023

**SAPIENTIA HUNGARIAN UNIVERSITY OF
TRANSYLVANIA
FACULTY OF TECHNICAL AND HUMAN SCIENCES
COMPUTER SCIENCE SPECIALIZATION**



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Examining social phenomena in a real company's email
communication network

BACHELOR THESIS

Scientific advisor:
Dr. Káta Zoltán,
Associate professor
Oltean-Péter Boróka,
Assistant professor

Student:
Tankó Tamás

2023

Declarație

Subsemnatul/a TANKO' TAMA'S, absolvent(ă) al/a specializării
INFORMATICA, promoția 2023 cunoscând
prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a
Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta
lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală,
cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de
specialitate sunt citate în mod corespunzător.

Localitatea, TÂRGU MUREȘ
Data: 14.VI.2023

Absolvent

Semnătura... T. Tama's

Kivonat

Dolgozatom témája a szociális jelenségek vizsgálata egy valós cég e-mail kommunikációs hálózatán, amely során különböző gráfokra alkalmazott algoritmusok futásidejét hasonlítom össze. Ezen műveletekkel összekapcsolom a szociális jelenségeket az informatikai algoritmikával és gráfelmélettel. Előnyös esetként nem egy kitalált (fiktív) adatbázis elemeivel dolgoztam, hanem az ENRON valós e-mail kommunikációs adatbázisán végeztem. A kutatás során adatbányászatot végeztem el, hogy kizárólag a vállalaton belüli kapcsolatokat vizsgáljam meg, ezzel csökkentve a gráfban használt csúcsok számát.

Az interdiszciplináris kutatásom során különböző gráfelméleti algoritmusokat használtam, például a mélységi bejárás és a Girvan-Newman módszer.

A valós adatbázis lehetőséget nyújt a hálózat dinamikájának vizsgálatára, ezáltal a kapcsolatok kialakulása is követhető, amit vizualizálással is megjeleníték.

Rezumat

Románra fordítva: Tema lucrării mele este investigarea fenomenelor sociale într-o rețea de comunicare prin email a unei companii reale, în cadrul căreia compar duratele de execuție ale algoritmilor aplicați asupra diferitelor grafuri. Prin aceste operațiuni, conectez fenomenele sociale cu algoritmica informatică și teoria grafurilor. Într-un scenariu favorabil, nu am lucrat cu elemente fictive, ci am utilizat baza de date reală a comunicărilor prin email ale companiei ENRON. În cadrul cercetării, am efectuat și minerit de date pentru a examina exclusiv relațiile interne din cadrul companiei, reducând astfel numărul de noduri utilizate în graf.

În timpul cercetării mele interdisciplinare, am utilizat diferite algoritme de teoria grafurilor, precum parcurgerea în adâncime și metoda Girvan-Newman.

Baza de date reală oferă posibilitatea de a investiga dinamica rețelei, permițându-ne să urmărim formarea conexiunilor, care pot fi vizualizate.

Abstract

The topic of my paper is the examination of social phenomena in a real company's email communication network, where I compare the execution times of algorithms applied to different graphs. Through these operations, I connect social phenomena with computer algorithms and graph theory. In a favorable scenario, I did not work with fictional elements, but used the real email communication database of the ENRON company. In the research, I also conducted data mining to exclusively examine internal relationships within the company, thereby reducing the number of nodes used in the graph.

During my interdisciplinary research, I utilized various graph-theoretical algorithms such as depth-first search and the Girvan-Newman method.

The real database provides the opportunity to examine the dynamics of the network, allowing us to track the formation of connections, which can be visualized as well.

Tartalomjegyzék

1. Bevezető	10
1.1. Cím értelmezése és a kutatás célja	11
2. A hálózat vizsgálatának lépései	12
2.1. Adatbányászat, adatszűrés	13
2.2. Adatvizualizáció	14
2.3. Összességében	19
3. A hálózaton kipróbált algoritmusok	20
3.1. A pletyka terjedése	20
3.2. A csomósodási együttható kiszámítása	24
3.3. Az hálózat kapcsolatainak vizsgálata	26
4. Girvan-Newman módszer	30
4.1. Általános tudnivaló a módszerről	30
Összefoglaló	37
Köszönetnyilvánítás	38
Ábrák jegyzéke	39
Táblázatok jegyzéke	40
Irodalomjegyzék	41

1. fejezet

Bevezető

Az emberek életében létfontosságú a kapcsolatok fenntartása, ezáltal minden embernek létrejön egy szociálisan vizsgálható környezete, amelynek következménye egy hálózat létrejötte. Ezen hálózatok sokasága könnyen ábrázolható gráfként, amelyet különböző vizsgálatoknak lehet alávetni. Az egyik legkiemelkedőbb mérési lehetőség a kapcsolatokban észlelhető hidak felismerése, valamint különböző algoritmusok alkalmazása a hálózaton (például a Girvan-Newman módszer).

A 21. században a számítógépekkel való kezelhetőség és a mérhetőség jelenti a legjellemzőbb képességet az emberi kapcsolatokból származó hálózatok szempontjából. Az egyik legjobb példa erre az internet, amelyet napjainkban több millió ember használ naponta. Ha elgondolkodunk ezen, láthatjuk, hogy az interneten keresztül létrejött hálózaton rengeteg adat és személy közötti kapcsolat tárolható. Az ilyen hálózatok elérhető adatainak kutatása ma népszerűvé vált, és több informatikai témakört egyesít. Egy felhasználható valós adatbázis a kutatások szempontjából az Enron kommunikációs hálózata. [Adi05] Ezen adatbázist már többen kutatták, de én szerettem volna egy olyan kutatást végezni, amely hozzájárul az egyetemi tananyag bővítéséhez. Ezért átnéztem több kutatást [KPX11], és saját elképzelésem alapján alkalmaztam néhány algoritmust a kapott hálózaton. Dolgozatom elkészítése során az algoritmusok összehasonlítása mellett adatbányászattal és kommunikációs hálózatok vizsgálatával is foglalkoztam. Minden hálózatnak megvannak sajátos tulajdonságai, amelyek alapján különböznek egymástól, és így kijelenthető, hogy bár lehetnek közös vonások, nem egyformák.

Az egyik szempont, amely alapján sorolhatóak a hálózatok, a kapcsolatok sűrűsége, azaz az általam választott esetben az Enron vállalatban dolgozók közötti üzeneteket tekintettem élnek a gráfokban. Mikor lesz egy kapcsolati hálózat, gráf sűrűsége a lehető legnagyobb? Egy adott gráfnak akkor lesz nagyobb sűrűsége, ha a csúcsok közötti élek száma közelíti a maximális értéket (Egy n csúcsból álló irányítatlan gráfnak maximálisan $n*(n-1)/2$ éle lehet).

A mérésekhez irányított gráfokként használtam fel az adatokat, ugyanakkor az adatbázis méretét a lehető legkisebbre próbáltam csökkenteni a cég által biztosított fiókok szempontjából. Az így kapott személyeket azonnal vizsgálatok alá lehetett helyezni, valamint az így nyert adatok egy részéből adatvizualizációt is létrehoztam a hálózat átláthatóságának érdekében.

1.1. Cím értelmezése és a kutatás célja

A cím, "Szociális jelenségek vizsgálata egy valós cég e-mail kommunikációs hálózatán", felhívja figyelmünket arra, hogy ez a hálózat nem egy általunk tervezett gráfot hoz létre, hanem egy cég munkatársai között létrejövő kapcsolatokat ábrázol. Az e-mail alapján határozhatók meg a gráf élei, a küldő és címzett részek alapján.

De miért nevezzük szociális jelenségek vizsgálatának ezt a kutatást? Ez nagyon egyszerű válasz. Mivel az e-mailen keresztül történő kapcsolat emberi interakciókat foglal magába. Az így kialakult kapcsolatokban számos szociális jelenség jelentkezhethet, például:

- kommunikációs minták, melyre egy jó illusztráció, ha egy személy előnybe részesíti egy bizonyos csoporttal a kapcsolattartást.
- információáramlás és annak hatása, tehát egy emberi véleményt is befolyásolhat különböző üzenetben szereplő információ
- társas támogatás és kapcsolatépítés, ezen jelenség arra mutat rá, hogy az üzenetváltásokkal tudjuk támogatni egymást, vagy szoros kapcsolatokat is tudunk kialakítani
- utolsó példaként a hálózati struktúrát sorolnám fel, amelyet én is választottam ezen kapcsolati lánc elemzésére.

Miért választottam a hálózati struktúrát a szociális jelenségek közül?

A hálózatokat átalakíthatjuk gráfokká, ezáltal a jelenségek kutathatóak és vizsgálhatóak gráfelméleti szinten. Így a szociális jelenségeket összekapcsolhatom az informatikával, amelynek tudományával az elmúlt években foglalkoztam.

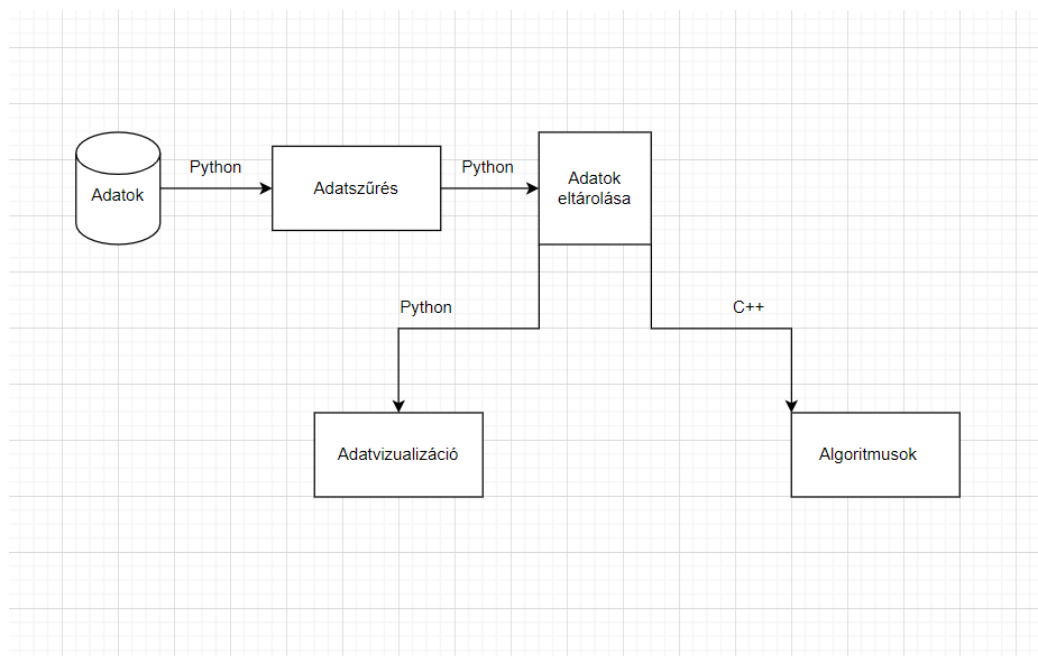
A kutatás célja, hogy létrehozzak egy olyan programot, amely megtalálja a legfontosabb személyt ebben a hálózatban, valamint meghatározza a hidakat, amelyek nélkül a hálózat több részre szakadna. A program fejlesztésével segíteni szeretnék az informatikát tanuló diáktársaknak, hogy valós cégből származó adathálózattal dolgozhassanak, és ne kelljen kitalálniuk egy adott gráfot. Emellett néhány vizualizált eredménnyel is szolgálni szeretnék a Girvan-Newman módszer alkalmazásával.

2. fejezet

A hálózat vizsgálatának lépései

A hálózat vizsgálatát több részre osztottam, mivel különböző programozási nyelveket használtam az eredmények kimutatásához.

Az alábbi ábra szemlélteti a kutatási folyamat lépéseit:



2.1. ábra. Folyamat ábra

- Első lépésként az adatszűrés , adatbányászat , amely következtében feldolgozható adatokat átállítottam elő és ezen lépés után két részre választottam a műveleteket.
- Egyik rész az adatok vizualizációja. Itt kirajzolásra kerül a hálózat kis rész gráfja és annak fejlődése .
- Második részben pedig a gráfelméletben megjelenő szociális jelenségek vizsgálata és ahhoz köthető algoritmusok megírása volt a fő célpont.

A programozási nyelvek közül két különböző nyelvet, Python-t és C++-ot választottam, és használtam az adatokkal való műveletek és vizsgálatok során. A Python egy

interpretált nyelv, ami azt jelenti, hogy fordítás nélkül futtatható. Ezzel szemben a C++ nyelvű forráskódot gépi kóddá kell fordítani a futtatható állomány létrehozása érdekében. A Python dinamikusan típusos nyelv, ami azt jelenti, hogy a változók típusát futásidőben határozza meg, nem kell előre meghatároznunk. Ez a tulajdonság nem alkalmazható a C++ nyelvre, de érdemes megemlíteni, hogy a C++ egy hatékony nyelv, amely lehetővé teszi a hardverközeli programozást és a memóriakezelést precízen irányítja.

Miért választottam a Python és a C++ nyelveket a kutatás során? Először is, minden műveletet el lehetett volna végezni más nyelveken is. Például az adatokat adatbázisban tárolhatnánk és az adatbáziskezelő nyelvét, például SQL-t használhatnánk a műveletek végrehajtásához. Egy másik lehetőség az adatok feldolgozására és vizualizációjára az adatbázisból való lekérdezések segítségével. Azonban én a Python és a C++ nyelveket választottam, mert a gráfelméleti algoritmusokat C++ nyelven tanultam, amihez a diákok már korábban is kapnak betekintést középiskolai tanulmányaik során. Ezért egy olyan nyelvet választottam, amelyet már ismernek, és könnyen át tudják gondolni az algoritmusok lényegét, ahelyett, hogy egy új nyelvet kellene elsajátítaniuk a téma mellett.

Ezzel a döntéssel a diákoknak lehetőségük van jobban megérteni és követni a programozási részleteket, és könnyebben alkalmazni a gráfelméleti algoritmusokat a kutatás során.

2.1. Adatbányászat, adatszűrés

Az adatbányászathoz a Python nyelvet használtam, amely segítségével sikerült megszerezni az összes Enron cég által biztosított e-mail címet. Konkrétan 5653 e-mail cím van, amelyek "@enron.com" végződéssel rendelkeznek. A kommunikációs hálózat létrehozásához az üzenetek jelentik az építőelemet, tehát ki kellett bányásznom és el kellett mentenem az üzeneteket annak érdekében, hogy létrehozhassak egy gráfot, amelyet vizsgálhatok. A szűrés ebben az esetben fontos volt, hogy csak az Enron-os e-mailek közötti üzeneteket használjam fel, ezzel csökkentve az adatok méretét. Még így is rengeteg adatot kaptam, pontosan 88975 üzenetet, amelyekről csak a minimális információt (küldő, címzett és dátum) mentettem el.



```
['robert.badeer@enron.com' 'dale.neuner@enron.com'
'mona.petrochko@enron.com' ... 'james.lawrence@enron.com'
'all.group@enron.com' 'paul.schiavone@enron.com']
szemelyek szama:
5653
Uzenetek szama
88975.8
```

2.2. ábra. Adatok számlálásának eredménye

Az alábbi kódrészlettel kezdtem az adatokat kinyerni a megkapott adattömegből

```
1 from pathlib import Path
2 import regex as re
3 from tqdm import tqdm
4 import csv
5
6 # open the file in the write mode
7 csv_file = open('eredmeny.csv', 'w', encoding='UTF8', newline='')
8 writer = csv.writer(csv_file)
9 for i in range(1, 840):
10     result = list(Path("../").rglob("*/all_documents/%s" %i))
11     print(result)
12     for file in tqdm(result):
13         with open(file, "rt") as f:
14             data = f.read()
15             dateof = re.search(r'(?:(?!(0-9)+) ([A-Z]+) ([0-9]+)', data)
16             sender = re.search(r'([fFrRoOmM]+: )(\w.+)+@(\w-)+\.(?!(\w-)+)', data)
17             to = re.search(r'([tToO]+: )(\w.+)+@(\w-)+\.(?!(\w-)+)', data)
18
19             print(dateof)
20             if sender and to and dateof:
21                 writer.writerow([sender.group(0).lower().replace('from: ', ''), to.group(0).lower().replace('to: ', ''), dateof.group(0).lower().replace('dateof: ', '')])
```

2.3. ábra. Adatbányászat

- első lépésként létrehoztam egy csv fájlt, amelyben a megkapott adatokat tudom tárolni
- a következő lépés egy ciklus létrehozása, hogy minden személyt leellenőrizzünk
- minden mappában több email szerepelt ezért először meghatároztam a fájlok listáját
- ezen lista elemeit, tehát az összes fájlt meg kellett nyitnom és kikeresni a számomra szükséges adatokat(küldő,címzett és dátum)
- ezen adatok kiválasztásához regex kifejezéseket használtam amellyel nagyon egyszerűen lehetett rátalálni a helyes adatokra
- minden olyan esetben ha volt dátum, küldő és címzett akkor az adathármaszt kiírtam a program elején létrehozott fájlba
- ugyanakkor az itt kapott eredményt még át kellett futtatnom egy szűrési algoritmuson amelyet már az adatok biztonságos tárolása miatt Drive-ra töltöttem fel és onnan használtam fel .
- a szűréshez is Python nyelvet használtam amelyet már a gyors mentési lehetőséggel élve Colab-ban írtam meg.
- a legfontosabb szempont az volt a helyes adatok kiválasztásában, hogy felhasználók e-mail címei "@enron.com"-végződéssel rendelkezzenek ugyanakkor az üzenetekben mindkét személyre jellemző legyen ezen tulajdonság.

2.2. Adatvizualizáció

A következő részben röviden bemutatom a hálózat megjelenítését és annak fejlődését.

Az ábrákon és adatokon, amelyeket eredményként bemutatok, az első 50 e-mail címet vettem figyelembe, valamint a közöttük lévő szigorú kapcsolatot és üzenetváltást. A vizualizáció előnye, hogy nem csak elképzelhetjük a kapcsolatok fejlődését, hanem ténylegesen láthatjuk azokat, és könnyebben észlelhetjük a változásokat.

A részhálózat megjelenítése előtt két műveletet hajtottam végre, amelyek segítették az algoritmusom gyorsabbá tételét.

- Első művelet ,szűrés, amely fontos az optimálisabb futási időhöz , az adatok csökkentése, amelyet a következő ábrán látható program végezett el.

```
1 with open('enronosok.txt', 'r') as file:
2     lines = file.read()
3     persons=lines.strip().split()
4     with open('kapcsolatok.txt', 'r') as file:
5         email = file.read()
6
7     emails = email.strip().split("\n")
8     print(emails[1].split(" "))
9
10    outfile=open('kivalasztott.txt','w')
11
12    for e in emails:
13
14        t = e.split(" ")
15
16        try:
17            index = persons.index(t[0])
18            email1 = index
19
20        except ValueError:
21            email1 = -1
22
23        try:
24            index = persons.index(t[1])
25            email2 = index
26        except ValueError:
27            email2 = -1
28        if email1 >= 0 and email2 >= 0:
29            outfile.write(e+ "\n")
```

2.4. ábra. Szűrési művelet

- A fenti ábrán a program megnyitja a 2 fájlt amelyben az adatok tárolva vannak, elsőben a személyeket tároltam, amiből tárolom egy tömbbe az e-mail címeket, míg a második fájlból az üzenetek adatait fogom eltárolni .
- Harmadik fájlnevnél már látható hogy ezt fájlt nem olvasásra használom, mert ebbe fognak bekerülni a szűrésen áthaladó adatok egy része
- Számomra azon adatok kell hogy megmaradjanak, amelyek az első 50 személy közötti üzeneteket képezik
- Ehhez a szűrésnek köszönhetően elmondhatjuk hogy az 5653 felhasználóból kiválasztott első 50-hez nem 88976 üzenetet kell átnézek a következő adatvizualizációnál hanem csak 3969-et .

- Második művelet ,rendezés, amely a kronológiai sorrendbe állítja az üzeneteket, kapcsolatokat, ehhez szükséges rendezést a képen látható kódrészlet által oldottam meg:

```

12 for i in range(len(emails)):
13 | t=emails[i].split(" ")
14 month=monthToInt(t[3])
15 johlen=0
16 if i==0:
17     rendezett.append(emails[i])
18 else:
19     rendezett.append(rendezett[i-1])
20     s=i-1
21     while s>0:
22         rendezett[s]=rendezett[s-1]
23         s-=1
24     rendezett[0]=emails[i]
25     j=0
26     while johlen == 0:
27         if j >= i - 1:
28             johlen = 1
29         seged1=rendezett[j].split(" ")
30         seged2 = rendezett[j+1].split(" ")
31         if seged1[4] < seged2[4]:
32             johlen=1
33         elif seged1[4] == seged2[4]:
34             if monthToInt(segед1[3]) < monthToInt(segед2[3]):
35                 johlen=1
36             elif monthToInt(segед1[3]) == monthToInt(segед2[3]):
37                 if seged1[2] <= seged2[2]:
38                     johlen=1
39                 else:
40                     rendezett[j]=rendezett[j+1]
41                     rendezett[j+1]=emails[i]
42                     j+=1
43             else:
44                 rendezett[j] = rendezett[j + 1]
45                 rendezett[j + 1] = emails[i]
46                 j += 1
47         else:
48             rendezett[j] = rendezett[j + 1]
49             rendezett[j + 1] = emails[i]
50             j += 1

```

2.5. ábra. Rendezés kronológiailag

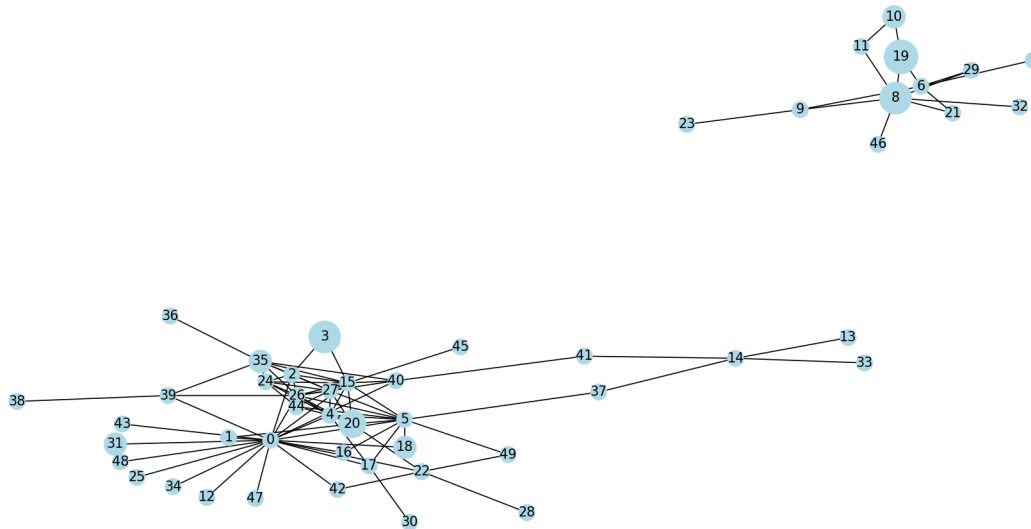
- Az említett rendezési módszer, amit leírtál, hasznos a listaelemek sorba rendezésére. A módszeredben az elemeket egymással összehasonlítva és cserélve helyezed őket a megfelelő pozícióba. Emellett említetted, hogy egy saját függvényt használsz a hónapok helyes sorrendjének meghatározására.

Miután rendezted a listát, elmentetted az eredményt egy új fájlba, amit később felhasználhatsz az adatvizualizációhoz.

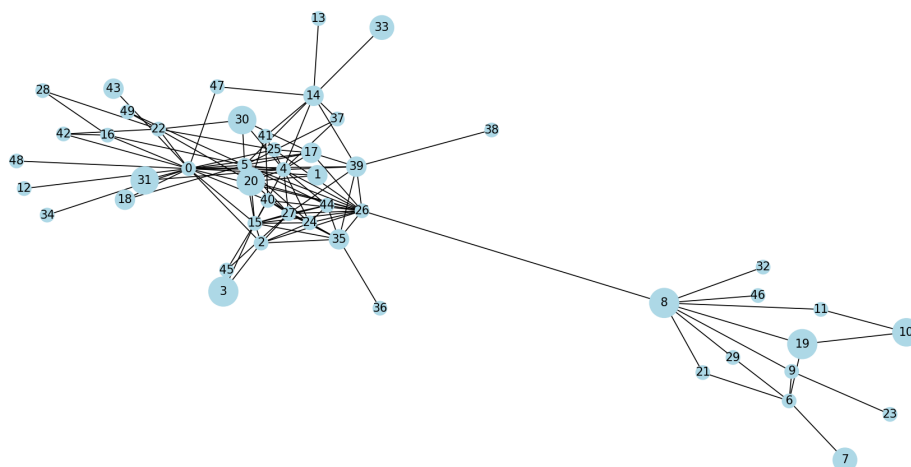
Ez a rendezési és mentési folyamat segít abban, hogy a hálózatot könnyebben áttekinthető és vizualizálható formában tudd megjeleníteni.

A szükséges adatcsökkentés és rendezés után következett a hálózat megjelenítése, amelyet az eddigi műveletekhez hasonló módon Pythonban készítettem el. Ez a kis rendszer egy

felületet hoz létre, amely 1000 üzenet után frissíti a kapcsolati ábrát, így jól láthatók a különbségek akár 1000, akár 2000 üzenetküldés között. A rendezett e-mail lista segítségével ezek a változások idővel arányosan jelennek meg. Az alábbiakban látható a hálózat 1000 üzenet után, valamint a második képen az összes üzenet elküldése után.



2.6. ábra. 1000 üzenet utáni hálózat



2.7. ábra. Összes üzenet utáni hálózat

Az ábrákon észrevehető, hogy különböző méretűek a csúcspontok, amelyek az elküldött üzenetekkel arányosan növekednek. Ezen méretbeli eltérést az alábbi kód segítségével állítottam be:

A kódrészletben feltüntetett 65. sorban látható egy ellenőrzés/feltétel, amelyben megnézem, hogy mindkét e-mail cím létezik-e. Ezen lépés után mindig a küldő részénél növelem az elküldött üzenetek számát, amelyet egy 50 elemű tömbben tárolok, és ezáltal tudom meghatározni, hogy az adott személy mikor kerül más területre, mikor kell


```

65 if email1 >= 0 and email2 >= 0:
66     sendings[email1]+=1
67     edge = (email1, email2)
68     edges.append(edge)
69     if sendings[email1]<21:
70         sizes[email1] = 200
71
72     elif sendings[email1] >20 and sendings[email1] < 51: #lightblue
73         sizes[email1]=400
74
75     elif sendings[email1] >50 and sendings[email1]<101: #blue
76         sizes[email1]=600
77
78     elif sendings[email1] >100 and sendings[email1]<250: #red
79         sizes[email1]=800
80
81     else:
82         sizes[email1]=900
83
84 if num%1000 == 0 or num == len(emails):
85     plt.clf()
86     G.add_edges_from(edges)
87     nx.draw(G, with_labels=True, node_color=colors, node_size=sizes, font_size=12, font_color='black')
88     plt.pause(2)
89
90
91 plt.show()

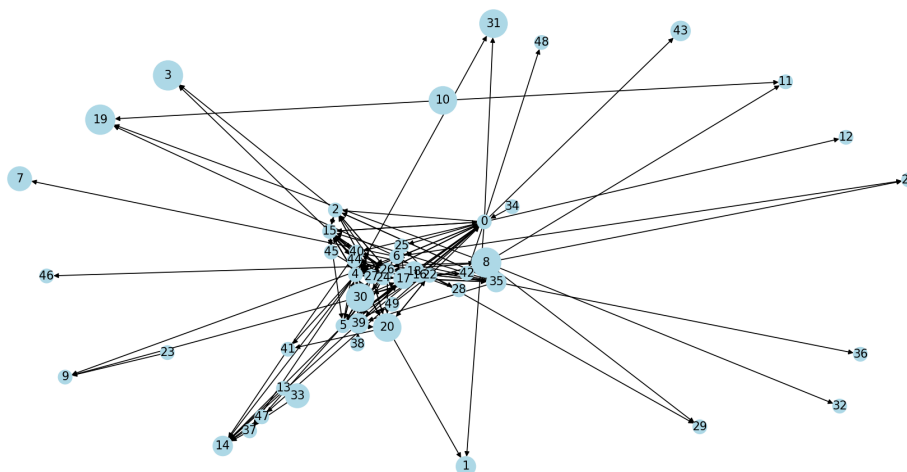
```

2.8. ábra. Csucspontok méretének beállítása és kijarzolása

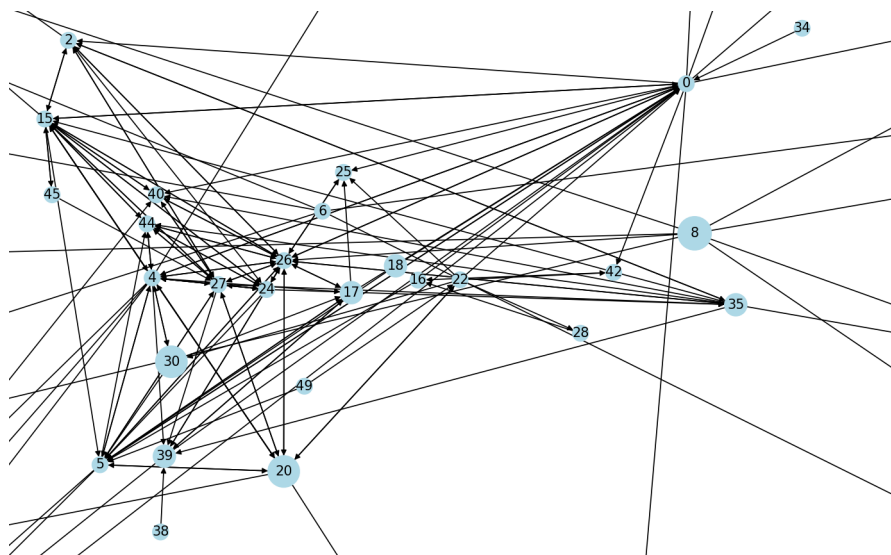
növelnem az ő vizuális méretét. Ugyanakkor látható egy másik dinamikus tömb (edges névvel), amelyben számpárokat, pontosabban a gráf éleit tárolom.

Ha végeztünk az összes üzenettel, akkor a feltöltött tömbök segítségével létrehozuk a gráfot. A tömb feltöltése több részben fog történni, mert így a kapcsolatok létrehozása során lehet kirajzoltatni a gráfot. Az általam választott lépés 1000 egységnyi nagyságú, és 2 mp-ig tart, hogy kirajzolja a gráfot, majd a következő lépés, változat következik, addig amíg eléri az utolsó üzenet feldolgozását.

Utolsó lépésként az adatvizualizációt bővítettem egy irányított gráf kirajzolásával. Az eddig használt 50 fős hálózat kapcsolatai irányokkal ellátva, valamint ráközelítve a program által generált ábra sűrített részébe.



2.9. ábra. Csucspontok kijarzolása irányított gráfként



2.10. ábra. Kis nagyítás a gráf középpontjára

2.3. Összességében

A fenti esetekhez azért használtam a Python nyelvet, mert úgy éreztem, hogy ez az a nyelv, amely teljesen új volt számomra az egyetem kezdetén, valamint ezzel a nyelvvel könnyedén lehet adatokat szűrni, akár összehasonlítani egymással. Ugyanakkor nagyon sok könyvtárral rendelkezik a grafikonok vizualizációjához. Ezen könyvtárak közül választottam a Matplotlib-et, amely többek között támogatja a különböző diagramtípusokat, illetve lehetővé teszi a részletes testreszabást. Emellett nagyon elterjedt könyvtár, így rengeteg dokumentáció és példa áll rendelkezésre a különböző hibák kiküszöböléséhez.

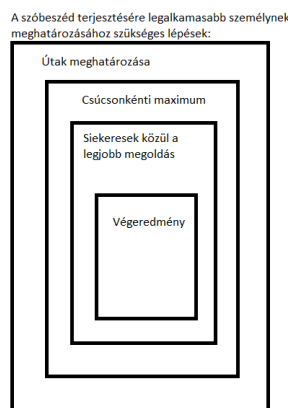
3. fejezet

A hálózaton kipróbált algoritmusok

Ebben a fejezetben bemutatom az általam megírt algoritmusokat, amelyeket ezen hálózaton teszteltem, és egyben összehasonlítom egy méretben kisebb, fiktív gráfon is alkalmazva. Az algoritmusok programozási nyelve a C++, amely eltér az eddig használt nyelvtől. Ugyanakkor ezt a nyelvet választottam, mert ezeket a programokat később más diáktársak is felhasználhatják a gráfelméleti algoritmika elsajátításakor. Így viszonyítani tudom a sok fontos információt és tanácsot az egyetemnek, ha ez csak egy kis töredéke annak, amit én tanulhattam az elmúlt évek során.

Következzenek az általam megvalósított algoritmusok, amelyeket a hálózat szociális jellegű vizsgálatára készítettem :

3.1. A pletyka terjedése

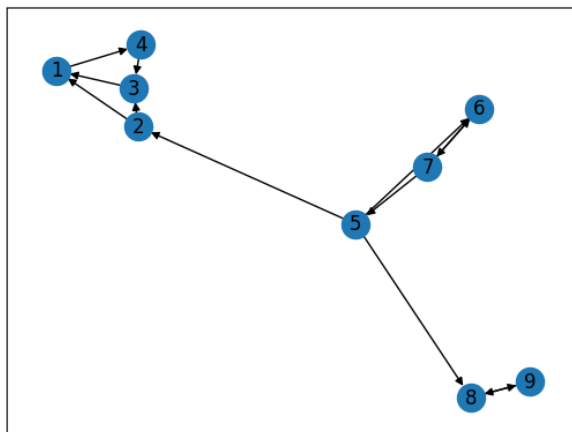


3.1. ábra. Szoftver lépései

Az első algoritmus a két pont közötti út meghatározására használt algoritmus továbbképzése, amelynek köszönhetően a szociológiai elemként tartott pletyka (szóbeszéd) milyen gyorsan tud terjedni, akár e-mailek által, egy nagyobb hálózaton keresztül. Ezen program felépítése több lépésből áll, amelynek első lépése egy mélységi bejárás egy irányított gráfon, majd a második lépésben minden csúcsra meghatározzuk az utat, vagyis hogy az üzenet egyik személytől a másik személyig hány emberen keresztül jut el.

Ugyanakkor az utolsó lépésben meghatározom, hogy a hálózatban ki lenne a legjobb központ, ki által terjed a legjobban a pletyka, valamint melyik személy nem tudja eljuttatni mindenhová a szóbeszédet.

Látványos áttekintésként az alábbi gráfot hoznám fel, amely segítségével könnyebb megérteni a fent említett információkat.



3.2. ábra. Gráf a szóbeszédhez

Az ábrán látható egy 9 csúcspontról álló irányított gráf, amelyet azért készítettem, hogy érthetőbb legyen az algoritmus magyarázata.

Az első lépés a mélységi bejárás által meghatározni az utakat. A képen is látható, hogy az 1 pontból csak a 4 és 3 pontig jutunk el, tehát itt az üzenet csak 2 személyhez jut el. Hasonlóképpen nem jutunk messze a 2., 3. és 4. pontokkal sem. Az 5. pont már minden más csúcsba el tudja küldeni az üzenetet. A leghosszabb út, amit megtesz az üzenet, a 4-es elemhez való információ közlést alkotja. Ehhez át kell mennie 2 ponton (2. és 1. csúcsokon), ami pontosan 3 él hosszú. Ezt a leghosszabb utat eltároljuk, valamint azt is, hogy összesen hány él mentén kellett bejárnia az utakat. Az utóbbi adat azért szükséges, ha van még egy hasonló elem, akkor ez dönti el, ki van a középpontban. De térjünk át egy másik olyan pontra, amely be tudja járni az egész gráfot, ez a pont a 7-es. A kiválasztott pont annyiban tér el az 5-östől, hogy a leghosszabb útja 4 él hosszú. Ezért a döntés az 5-ösre esik, így ő lesz a hálózat központja. Tehát az ő által küldött üzenetek érnek el mindenhová, és mindezt a leghatékonyabb idő alatt tudja elérni.

A kis példa után áttérnék az általam felhasznált 50 fős adattömegre, amelynek vizualizációját már bemutattam. Ugyanakkor bemutatom a programot, amelyet használtam az előző gráfnál is. Mint tudjuk, a gráfok nagysága és éleinek száma nagyban eltér, ezért összehasonlítottam a futási időket. A hamarosan megjelenített program a kis gráfra, amely 9 csúcsot és 12 élet tartalmaz, 0,0094 másodperc alatt végezte el a középpont megkeresését és annak pontos meghatározását, míg az 50 ezres és még szűrés nélküli, 80 000 fölötti üzeneteket tároló fájl használata során 0,0467 másodpercet volt képes elérni.

Következzen is a C++ szoftver bemutatása és annak magyarázata:

```

200 auto start = chrono::high_resolution_clock::now();
201 Pontok pontok[50];
202 int lepes = 0;
203 int siker = 0;
204 for (int i = 0; i < namelistsize; i++) {
205     siker = 0; seged1 = 0; seged2 = 0;
206     for (int j = 0; j < namelistsize; j++) {
207         int jart[50] = {0};
208         if (i != j) {
209             lepes = utakmeghatarozasa(matrix, i, j, namelistsize, jart);
210
211             if (lepes >= 0) {
212                 siker++;
213                 if (lepes > seged1) {
214                     seged1 = lepes;
215                 }
216                 seged2 = seged2 + lepes;
217             }
218         }
219     }
220     cout << endl;
221     if (siker == namelistsize-20) {
222         pontok[i].maxlepes = seged1;
223         pontok[i].osszlepes = seged2;
224         cout << "mindent bejart a(z) " << i + 1 << " maxlepese: " << pontok[i].maxlepes << endl;
225     }
226     else {
227         pontok[i].maxlepes = 0;
228         pontok[i].osszlepes = 0;
229         cout << "nem mindent jart be a(z) " << i + 1 << " sikeresbejarasa:" << siker << endl;
230     }
231 }
232 auto end = chrono::high_resolution_clock::now();

```

3.3. ábra. két pont közötti út távolság meghatározása

Az első képen látható egy függvény, amely, ahogy említésre került, az első lépésre íródott. Tehát meghatározza két pont között, hogy van-e út, és ha van, akkor az hány él hosszúsággal rendelkezik. Ennek a függvénynek összesen 5 paramétere van. Sorrendben egy két dimenziós tömb, tehát egy mátrix, majd az 'st' változó, amely egy valós számot tartalmaz, és a kezdő pontot határozza meg, vagyis hogy honnan kell indulni az út meghatározásához. A harmadik paraméter a végső pont indexét határozza meg, majd ezek után következik a méret, amely segíti a mátrix bejárását, és végül egy tömb, amelyben rögzítem, hogy az adott ponton már jártam-e. Ezzel a tömbbel ki tudom küszöbölni a gráfban keletkező körök létrejöttékor felmerülő végtelen ciklust. A 't' és 'mint' változók a jelenlegi út hosszát tárolják, majd ezek közül a minimumot választják.

Kezdeként ellenőrzöm, hogy ha van kapcsolat az indulási pont és az úticél között, akkor megkapom, hogy az út hossza 1, és visszatérek vele. Ha nincs kapcsolat, akkor beállítom, hogy jártam a kezdő pontban, majd áttérek azokra az esetekre, amikor minden olyan útvonalat ellenőrzök, amely a kezdőpontból indul és valamerre halad. Ha nincs ilyen útvonal, akkor visszatérek -1 értékkel, amivel jelezem, hogy nincs út a két pont között. Ha van olyan eset, hogy a program tovább tud lépni a kezdőpontból, vagyis van kimenő éle a pontnak, akkor ebben az esetben hívom meg újra a függvényt, amikor már eljutottam a pozícióhoz. Ezzel a meghívással hozom létre a rekurziót.

A minimális út kiválasztása az alábbi módon történik: Ha már van értéke a 'mint' változónak, és nagyobb az eddig meghatározott útnál ('t'-nél), akkor megváltoztatom annak értékét. Ugyanakkor, ha a 'mint' változó továbbra is -1, és kapok egy helyes útvonalat, vagyis változott a 't' értéke, akkor visszaadom újra a 't' értékét. Abban az esetben, ha a 't' soha nem változik, akkor a 'mint' értéke változatlan marad, és a program -1 értéket térít vissza. De ha a 'mint' értéke változott, akkor az adott értékhez hozzáadok 1-et, és ezt a függvény visszatéríti, mivel ebben az esetben a következő ponttól volt számolva a távolság.

```

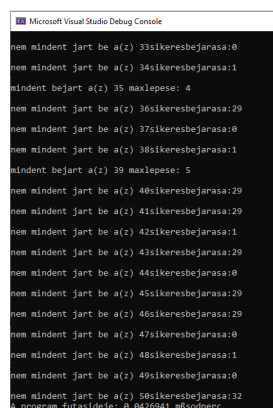
200 auto start = chrono::high_resolution_clock::now();
201 Pontok pontok[50];
202 int lepes = 0;
203 int siker = 0;
204 for (int i = 0; i < namelistsize; i++) {
205     siker = 0; seged1 = 0; seged2 = 0;
206
207     for (int j = 0; j < namelistsize; j++) {
208         int jart[50] = {0};
209         if (i != j) {
210             lepes = utakmeghatarozasa(matrix, i, j, namelistsize, jart);
211
212             if (lepes >= 0) {
213                 siker++;
214                 if (lepes > seged1) {
215                     seged1 = lepes;
216                 }
217                 seged2 = seged2 + lepes;
218             }
219         }
220     }
221
222     cout << endl;
223     if (siker == namelistsize-1) {
224         pontok[i].maxlepes = seged1;
225         pontok[i].osszlepes = seged2;
226         cout << "mindent bejart a(z) " << i + 1 << " maxlepes: " << pontok[i].maxlepes << endl;
227     }
228     else {
229         pontok[i].maxlepes = 0;
230         pontok[i].osszlepes = 0;
231         cout << "nem mindent jart be a(z) " << i + 1 << " sikeresbejarasa:" << siker << endl;
232     }
233 }
234
235 auto end = chrono::high_resolution_clock::now();
236
237 // Futásidő kiszámítása
238 chrono::duration<double> duration = end - start;
239 double seconds = duration.count();
240 cout << "A program futásideje: " << seconds << " másodperc." << endl;
241
242
243

```

3.4. ábra. Utak meghatározása és középső pont kiderítése

A fenti képen látható az algoritmus főbb része, amely során beolvasott adatokat feldolgozza a program, meghatározza minden csúcstól bejárható utak összegét, valamint ezek közül a legnagyobb út hosszát. Kivételt képeznek azok a csúcsok, amelyek nem tudnak üzenetet küldeni mindenkinek. A program minden érték esetén kiírja, hogy az adott személy elérte-e mindenkit, és ha igen, akkor mekkora a maximális út, amit egy személy elérése során megtett.

A fenti képen látható, hogy az algoritmus futási ideje mérve van (200., 232. sor), és az eredményben is megjelenik ennek kiírása.



```

nem mindent jart be a(z) 33 sikeresbejarasa:0
nem mindent jart be a(z) 34 sikeresbejarasa:1
mindent bejart a(z) 35 maxlepes: 4
nem mindent jart be a(z) 36 sikeresbejarasa:29
nem mindent jart be a(z) 37 sikeresbejarasa:0
nem mindent jart be a(z) 38 sikeresbejarasa:1
mindent bejart a(z) 39 maxlepes: 5
nem mindent jart be a(z) 40 sikeresbejarasa:29
nem mindent jart be a(z) 41 sikeresbejarasa:29
nem mindent jart be a(z) 42 sikeresbejarasa:1
nem mindent jart be a(z) 43 sikeresbejarasa:29
nem mindent jart be a(z) 44 sikeresbejarasa:0
nem mindent jart be a(z) 45 sikeresbejarasa:29
nem mindent jart be a(z) 46 sikeresbejarasa:29
nem mindent jart be a(z) 47 sikeresbejarasa:0
nem mindent jart be a(z) 48 sikeresbejarasa:1
nem mindent jart be a(z) 49 sikeresbejarasa:0
nem mindent jart be a(z) 50 sikeresbejarasa:32
A program futásideje: 0.0426941 másodperc.

```

3.5. ábra. Eredmény kiírása

3.2. A csomósodási együttható kiszámítása

Mi az a csomósodási együttható? A csomósodási együttható egy érték, amely a gráfokban a csúcsokhoz kapcsolódik, és az adott pont szomszédok közötti kapcsolatot vizsgálja. Ez az érték egy aránynak tekinthető, amely maximum 1 lehet. Az arány két összetevőből áll: a szomszédok között lehetséges kapcsolatok száma és a létrejött kapcsolatok száma. A csomósodási együttható segítségével meg lehet határozni, hogy mennyi valószínűsége van annak, hogy egy pont barátai egymással is kapcsolatban állnak. Minél kevesebb szomszédal rendelkezik egy csúcs, annál nagyobb az esélye annak, hogy közelíti az 1-hez tartó értéket.

Mi a csomósodási együttható célja? A csomósodási együttható mérése segít megérteni, hogyan változik a hálózat dinamikája az új élek hozzáadásával. Ha új élek kerülnek a gráfba, ez az érték növekedhet, és megfelelő mennyiségű él hozzáadásával általában eléri az 1-es értéket. Ha minden pont csomósodási együtthatója 1, akkor bármely pont kiválasztása esetén a szomszédok egy teljes gráfot alkotnak.

Összefoglalva, a csomósodási együttható az hálózat dinamikájának figyelésére szolgáló érték, amely rámutat a pontok közötti kapcsolatok kialakulására.

A használt hálózatot szép és látványos eredmények érdekében átalakítottam az irányított gráfból egy irányítatlan hálózattá. Ennek az átalakításnak az alapja az volt, hogy ha az x pont küld egy e-mailt az y pontnak, akkor az y pont ismeri az x pontot. Az alábbi ábrán bemutatam a szoftver lényeges részeit:

- Kezdeném az elsődleges függvénnyel, amelyet a második lépésben felhasználok. Ez a függvény segít meghatározni az adott pont szomszédai között kialakult kapcsolatok, azaz élek számát. A függvény egy számot ad vissza, és a paramétereik között szerepel egy mátrix, amely tárolja a pontok közötti éleket. Emellett megkap egy tömböt is, amelyben szerepel a szomszédok listája, valamint a szomszédsági lista hossza is felhasználható harmadik paraméterként.

A függvény a mátrix bejárása során meghatározza, hány él található a szomszédok között, és ezt az értéket az 'eredmény' változóban tárolja. Ezután az eredményt visszatéríti.

```
33 int egyutthatoSzamitas(int matrix[][50], int szom[], int sz) {
34     int eredmeny = 0;
35     for (int i = 0; i < sz - 1; i++) {
36         for (int j = i + 1; j < sz; j++) {
37             if (matrix[szom[i]][szom[j]] != 0) {
38                 eredmeny++;
39             }
40         }
41     }
42     return eredmeny;
43 }
```

3.6. ábra. Együttható kiszámításához használt függvény

- A második kódrészletben látható az összes csomósodási együttható kiszámítása és tárolása.

Ebben a részben a program használója beolvashat egy számot, amely a csomópontok között szerepel, és válaszként megkapja a félévente lekérdezett csomósodási együttható értékét. A program az adott személyek közötti kapcsolatokat rendezett elküldési időpont szerint növekvő sorrendbe, így jobban megfigyelhetővé válik a hálózat dinamikája.

```

207 int valasztott;
208 cout << "Olvasza be melyik pont csomosodasi egyutthatojanak a fejlodeset mutassuk ki a" << kezdet << "-" << kezdet + enronNum - 1 << "intervallumbol" << endl;
209 cin >> valasztott;
210 int* szomszedok = new int[enronNum];
211 int maximalisbaratsag = 0;
212 bool ebbeahonapba = false;
213 //rendezve megvannak a kapcsolatok
214 for (int c = 0; c < db; c++) {
215     //felbontom a elek betolteset kulonbozo idokre
216     segedId1 = nevsorban(nevsor, enronNum, kapcsolatok[c].nev1);
217     segedId2 = nevsorban(nevsor, enronNum, kapcsolatok[c].nev2);
218     iranyitatlan[segedId1][segedId2]++;
219     iranyitatlan[segedId2][segedId1]++;
220 }
221 //csomosodasi egyutthato meghatarozasa
222 if (kapcsolatok[c].honap % 6 == 0 && ebbeahonapba == false) {
223     ebbeahonapba = true;
224     maximalisbaratsag = 0;
225     for (int i = 0; i < enronNum; i++) {
226         for (int j = 0; j < enronNum; j++) {
227             if (iranyitatlan[i][j] != 0) {
228                 szomszedok[pontok[i].szomszedokszama] = j;
229                 pontok[i].szomszedokszama++;
230             }
231         }
232         if (pontok[i].szomszedokszama < 2) {
233             pontok[i].csomosodasi egyutthato = -1;
234         }
235         else {
236             maximalisbaratsag = (pontok[i].szomszedokszama * (pontok[i].szomszedokszama - 1)) / 2;
237             pontok[i].csomosodasi egyutthato = (double)egyutthatoSzamitas(iranyitatlan, szomszedok, pontok[i].szomszedokszama) / maximalisbaratsag;
238             // cout << egyutthatoSzamitas(iranyitatlan, szomszedok, pontok[i].szomszedokszama) << " / " << maximalisbaratsag << endl;
239         }
240     }
241     if (i == valasztott) {
242         cout << "A valasztott pont csomosodasi egyutthatoja " << pontok[i].csomosodasi egyutthato << " valamint szomszedjainak szama " << pontok[i].szomszedokszama << endl;
243     }
244 }
245 }
246 if (kapcsolatok[c].honap % 6 == 1) {
247     ebbeahonapba = false;
248 }
249 }
250 }

```

3.7. ábra. Csomósodási együttható meghatározása időközönként

A fenti kódrészletben a szűrésen és rendezésen áteső kapcsolatokat bejárva feltöltődik az irányítatlan gráf. Ez a feltöltés megszakad minden 6 hónapban, amikor a csomósodási pontok kiszámításai történnek. Ahhoz, hogy egy hónapban csak egyszer mérjünk, bevezettem az 'ebbeahonapba' nevű bool típusú változót, amely akkor vált igazra, ha már történt mérés az adott hónapban. Minden első és hetedik hónapban visszaállítom a változót hamisra (false). Az újbóli bejutás a mérésbe csak a 6-tal osztható hónapokban lehetséges.

Hogyan zajlik egy mérés? Egy mérés során megvizsgálom, hogy minden pontnak hány szomszédja van, és ezt beállítom az adott pontnak. Ezután két lehetőséget zárok ki: ha egy pontnak 1 vagy 0 szomszédja van. Ha egy pontnak 1 szomszédja van, akkor annak nem lehet más kapcsolata, és ha 0 szomszédja van, akkor sem lehet kiszámolni a pont csomósodási együtthatóját. Ha a szomszédok száma meghaladja az 1-et, akkor már kiszámítható az együttható.

Az együttható meghatározásának képlete :

$$\text{Együttható} = \frac{\text{SzomszédokKözöttiKapcsolatokSzama}}{\text{SzomszédokSzama} * (\text{SzomszédokSzama} - 1) / 2}$$

Ugyanakkor beszéljek néhány használt függvényről és változóról. A program kezdetén bekérek egy 1 és 50 közötti számot, amely meghatározza a gráf méretét. Ez a változó neve 'enronNum' (210. sor). A szomszédok dinamikus tömböt használok, amelyben tárolom a pontok jelenlegi szomszédjait a kapcsolatok meghatározásához.

A maximális barátság változó a fenti képletben látható osztó értékét fogja tárolni az eredmény számításához. A 'segedId1' és 'segedId2' változók a nevsorban függvény visszatérési értékét tárolják, amely egy int típusú elem lesz, mert a függvény visszaadja a nevsorban szereplő email indexét. Majd a meghatározott indexekből létrehozott éleket bevezetem az irányítatlan gráfot tartalmazó mátrixba. Ezzel már bemutattam az összes változót, amely a képen látható. Emellett említeném még a Kapcsolat típusú változót, amelyből egy tömb van létrehozva a kapcsolatok tárolására. Ennek az elemnek van két karakterláncot tartalmazó név1 és név2 változója, valamint egy három számból álló értéke a dátum kezelhetősége miatt.

```

15 struct Kapcsolat {
16     string nev1;
17     string nev2;
18     int nap;
19     int honap;
20     int ev;
21 };

```

3.8. ábra. Kapcsolat struktúra

- Utolsó megjegyzésként szeretném megmutatni egy algoritmus eredményét, amely egy 50 elemű gráf esetén, 3923 éllel rendelkezve az 5 pont csomósodási együtthatóját jeleníti meg 6 hónapos lépésekkel. Emellett a program végén meghatározom, melyik csúcs, vagyis személy rendelkezik a legnagyobb együtthatóval, feltéve, hogy a pontok számának legalább az ötödével van kapcsolatban. Végül, az utolsó eredményként meghatározom a legtöbb szomszédal rendelkező személyt a gráfból.

```

Kerem adja meg hany személybol alljon a graf
50
3923
Olvassa be melyik pont csomosodasi egyutthatojanak a fejlodeset mutassuk ki a0-49intervallumbol
5
A valasztott pont csomosodasi egyutthatoja -1 valamint szomszedjainak szama 1
A valasztott pont csomosodasi egyutthatoja 0 valamint szomszedjainak szama 2
A valasztott pont csomosodasi egyutthatoja 0.4 valamint szomszedjainak szama 5
A valasztott pont csomosodasi egyutthatoja 0.345588 valamint szomszedjainak szama 17
A valasztott pont csomosodasi egyutthatoja 0.441532 valamint szomszedjainak szama 32
Legnagyobb egyutthatoval rendelkezo csucs , amelynek legalabb 10 szomszedja van: richard.shapiro@enron.com erteke: 0.654412
Legtobb szomszeddal rendelkezo személy email címe robert.badeer@enron.com es szomszedjainak szama 40

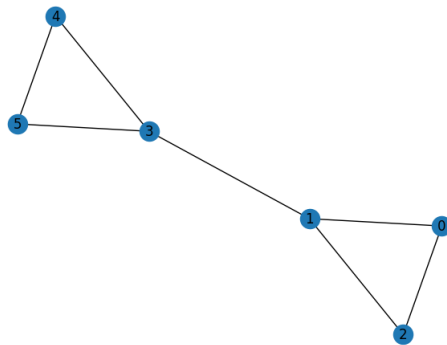
```

3.9. ábra. Változás az együtthatókban

3.3. Az hálózat kapcsolatainak vizsgálata

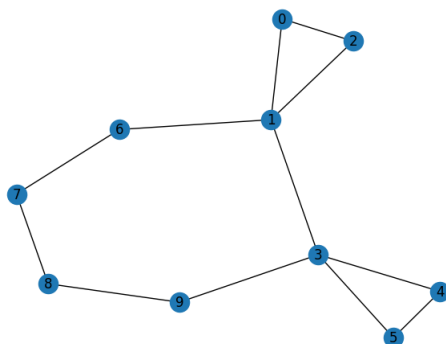
Ebben a részben a kapcsolatok fontosságával foglalkoztam, és azt vizsgáltam, hogy egy kapcsolat, üzenetváltás mennyire fontos a hálózatban. Azért fontos egy ilyen nagy hálózatban, mert akár egyetlen egy kapcsolat, kötődés két egyén között is jelentős hatással lehet a hálózatra. Ha töröljük vagy megszüntetjük az adott kapcsolatot, az egész hálózat több részre szakadhat. Ezt a szétesést a gráfoknál úgy érhetjük el, hogy töröljük az egyik élt, amely egy hídát alkotott két komponens között.

Az alábbi gráfon is látható, hogy a 3-1 él hídát képez a 2 háromszöget kirajzoló pontthalmazok között.



3.10. ábra. Gráf a híd magyarázatához

A második ábrán megfigyelhető különbség segít megérteni a lokális híd kialakulásának könnyed magyarázatát. Amikor a 1-3 él törlésre kerül, még mindig képesek kommunikálni egymással az üzenetek, de a bejárás során megfigyeljük, hogy a két pont közötti távolság megnövekszik. Ha ez a növekedés meghaladja a 2 egységet, akkor a törölt élet lokális hídnak nevezzük.



3.11. ábra. Gráf a lokálishíd magyarázatához

És mire is jó, ha tudjuk, hogy egy él híd vagy sem? A hídek lehetővé teszik két vagy több komponens összekapcsolását, így összefüggő gráfokkal tudunk dolgozni. Ugyanakkor a hídek nélkül is lehet az hálózat összefüggő, de ekkor lehetnek benne lokális hídek. Ha egy összefüggő gráfban nincs híd, és nincsenek lokális hídek sem, akkor nagy valószínűséggel teljes gráfról beszélünk.

Ezeket a jelenségeket vizsgáltam az általam használt adatokon, és a következő kis kódrészletekben bemutatom a kutatás lépéseit.

Az algoritmus legfontosabb eleme egy mélységi bejárás, amelyet a legrövidebb út meghatározásához is használtam a dolgozathoz szükséges szoftverek megírásában. A bejárás segítségével el tudom dönteni, hogy két pont között, ha törlöm az élt, akkor marad-e út a két csúcs között. Ennek egyik alapvető része az, hogy az irányított gráfot átalakítottam irányítatlan gráffá, mivel egy irányított gráfban nagyon nehéz két pont között több

utat keresni, mivel kevesen válaszolnak az üzenetekre. Ezért szemléletesebbnek találtam, ha irányítatlan gráfon tesztelem a csúcsokat.

```

146     Elek* elek = new Elek[t];
147     t = 0;
148     int p = 0;
149     int uthossz=0;
150
151     for (int i = 0; i < enronNum-1; i++) {
152         for (int j = i + 1; j < enronNum; j++) {
153
154             if (matrix[i][j] != 0) {
155                 int jart[50] = { 0 };
156                 p = matrix[i][j];
157                 matrix[i][j] = 0;
158                 matrix[j][i] = 0;
159                 elek[t].nev1Id = i;
160                 elek[t].nev2Id = j;
161                 uthossz = utakmeghatarozasa(matrix, i, j, enronNum, jart);
162                 if (uthossz < 0) {
163                     elek[t].tipus = (string)"hid";
164                     elek[t].atfedesifoka = 0;
165                 }
166                 else {
167                     if (uthossz > 2) {
168                         elek[t].tipus = (string)"lokalishid";
169                         elek[t].atfedesifoka = atfedes(matrix, i, j, enronNum);
170                     }
171                     else {
172                         elek[t].tipus = (string)"el";
173                         elek[t].atfedesifoka = atfedes(matrix, i, j, enronNum);
174                     }
175                 }
176
177                 cout << "A(z) " << i << " -" << j << " tipusa: " << elek[t].tipus << " es szomszedsag atfedesi foka: " << elek[t].atfedesifoka << endl;
178                 matrix[i][j] = p;
179                 matrix[j][i] = p;
180                 t++;
181             }
182         }
183     }

```

3.12. ábra. Híd meghatározása

A fenti program részben megfigyelhető egy egymásba ágyazott 'for' ciklus, amely az éleket keresi meg és velük végzi el a műveleteket. Először megvizsgálja, hogy egy adott él létezik-e két pont között. Ha igen, akkor átírja annak értékét egy változóba, majd törli az élt (az értékét 0-ra állítja). A törölt éllel rendelkező mátrixszal meghívja a két pont közötti útvonalat kereső függvényt, amely egy számot ad vissza. Ha a visszatérített érték kisebb mint 0, akkor nincs kapcsolat a két pont között, tehát a törölt él híd volt. Ha az útvonal hossza meghaladja a 2 egységet, akkor a törölt élnek lokális híd jellegzetességet adunk. Ha az előző két eset nem teljesül, akkor az él átlagosnak tekintett jellegű marad.

Az él tulajdonságainak tárolására egy Elek struktúrát hoztam létre, amely tartalmazza a küldő és címzett információkat, a szomszédsági átfedést, az él jellegét tároló lehetőséget, valamint a később említett közteségi foksámát.

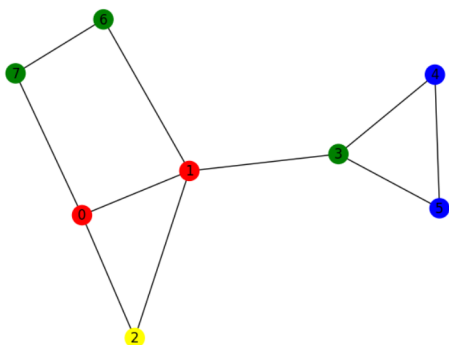
```

struct Elek {
    string tipus;
    int nev1Id=0;
    int nev2Id=0;
    double atfedesifoka = 0;
    int koztessegiFok = 0;
};

```

3.13. ábra. Él struktúra

Ahogy az előző gondolatban is említésre került a következő tulajdonság amely az élekhez tartozik a szomszédsági átfedés. Ezen változó kiszámítása hasonlóan a pontok csomósodási együtthatójához, egy olyan arány határoz meg, amely az él két végpontjának a közös szomszédjait és az mindkét pont összes szomszédjának számát használja fel. A kódrészlet magyarázata előtt kis ábrával pontosítanám ezen átfedetségi változó értékének fogalmát, melyben a pontok színbeli eltéréseivel szemléletesebb a különbség és könnyen észrevehető, hogy miből is áll ez az érték.



3.14. ábra. Átfedéshez használt példa gráf

Ahogy az előző részben is említésre került, az élekhez tartozik a szomszédsági átfedés tulajdonság. Ennek a változónak a kiszámítása hasonló módon történik, mint a pontok csomósodási együtthatójának meghatározása. Ez az arány meghatározza az él két végpontjának közös szomszédjainak számát és az összes szomszédjának számát mindkét pontnál. Az ábrával kísérve pontosítom ezen átfedési változó értékének fogalmát, ahol a pontok különböző színekkel vannak jelölve, hogy szemléletesebb legyen a különbség és könnyen észrevehető legyen, hogy ez az érték miből áll.

```
double atfedes(int matrix[][50], int elso, int masodik, int mer) {
    double osszes = 0, kozos=0;
    bool volt = false;
    for (int i = 0; i < mer; i++) {
        volt = false;
        if (matrix[elso][i] != 0 && i != masodik) {
            osszes++;
            volt == true;
        }
        if (matrix[masodik][i] != 0 && i != elso ) {
            osszes++;
            if (volt) {
                kozos++;
                osszes--;
            }
        }
    }
    if (kozos == 0) { return 0;}
    else { return kozos/osszes ;}
}
```

3.15. ábra. Átfedés számításához használt függvény

A függvény paraméterlistája egy mátrixot tartalmaz, amelyben az élek vannak tárolva. A második paraméter az él indulási pontját határozza meg, és a függvény megvizsgálja ennek a pontnak a szomszédjait. Ezután a harmadik paraméterrel ismét ugyanezt a műveletet végzi el. Az utolsó paraméter meghatározza, hogy hány csúcsból áll a gráf. Ha ez a szám kevesebb, mint 50, akkor az "i" változónak nem kell 50-ig iterálnia a "for" ciklusban.

A közös szomszédok kiválasztása egy segéd "bool" típusú változóval van megoldva. Ez a változó csak akkor vesz fel igaz értéket, ha az "i"-dik csúcsnak van közös szomszédja az első elemmel. Ha a második elem is kapcsolatban van az "i"-dik elemmel, akkor azt közös szomszédnak számolom, és visszavonom az összes szomszéd számának növelését. A bejárás után, ha a közös változó értéke 0 maradt, akkor a függvény 0-t ad vissza, különben pedig egy 0 és 1 közötti értéket.

4. fejezet

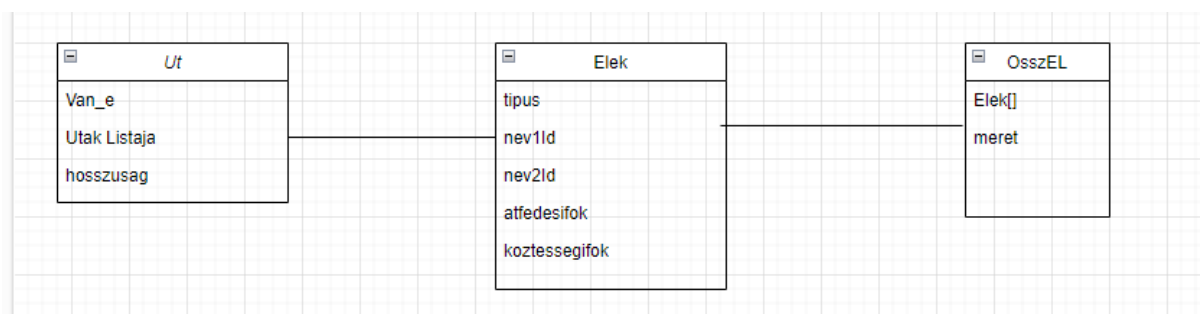
Girvan-Newman módszer

4.1. Általános tudnivaló a módszerről

Az algoritmus elnevezése Michelle Girvan, egy amerikai fizikus, és Mark Newman, szintén egy amerikai fizikus nevéből származik. A két fizikus egy hierarchikus módszert fejlesztett ki a közösségek kimutatására komplex rendszerekben.

Az algoritmus a hálózat felépítését vizsgálja, és az élek közötti köztességi fokra épül. A köztességi fok egy érték, amely meghatározza, hogy az adott él hány legkisebb út megy át rajta. Ez az érték a forgalmat tükrözi, vagyis azt, hogy a legkisebb utak gyakori átjárhatósága mekkora. Az él közösségi fokának értéke az áthaladó összes legrövidebb út forgalmának az összege. Ez az érték segít több részre bontani a hálózatot a legforgalmasabb élek törlésével. Az így kapott részgráfok alkotják az első szintű góccokat. Ezután ezeket a részalmazokat tovább lehet bontani, amíg minden élet törölünk.

Az általam megvalósított algoritmus három modulra épül. Az első modul az útmodul, amely a legrövidebb utak tárolásában használható, és képes az élek tárolására. A második modul az élekről tárol információkat, mint az él típusa, kezdő- és végpontja, köztességi foka és átfedési foka. Az utolsó modul azért jött létre, hogy egyik függvény visszatérési értékét tovább tudjuk adni, és ezen keresztül az éleket kezelni tudjuk.



4.1. ábra. Modulok

Modulok bemutatása után következzen a legkisebb út meghatározása :

```
81 Ut pontosut(int matrix[][50], int m, int st, int fin, int jart[]) {
82     Ut ut, segedUt;
83
84     if (matrix[st][fin] > 0) {
85         ut.utak[ut.hossz][0] = st;
86         ut.utak[ut.hossz][1] = fin;
87         ut.van_e = true;
88         ut.hossz = 1;
89         return ut;
90     }
91     else {
92         jart[st] = 1;
93         int aktualis = 0;
94         for (int i = 0; i < m; i++) {
95             if (matrix[st][i] > 0 && jart[i] < 1) {
96                 ut = pontosut(matrix, m, i, fin, jart);
97                 if (segedUt.van_e == false && ut.van_e == true) {
98                     segedUt.van_e = true;
99                     segedUt.hossz = ut.hossz;
100                     for (int i = 0; i < ut.hossz; i++) {
101                         segedUt.utak[i][0] = ut.utak[i][0];
102                         segedUt.utak[i][1] = ut.utak[i][1];
103                         aktualis = i;
104                     }
105                 }
106                 if (segedUt.van_e == true && ut.van_e == true && segedUt.hossz > ut.hossz) {
107                     segedUt.hossz = ut.hossz;
108                     for (int i = 0; i < ut.hossz; i++) {
109                         segedUt.utak[i][0] = ut.utak[i][0];
110                         segedUt.utak[i][1] = ut.utak[i][1];
111                         aktualis = i;
112                     }
113                 }
114             }
115         }
116         if (segedUt.hossz > 0) {
117             segedUt.utak[segedUt.hossz][0] = st;
118             segedUt.utak[segedUt.hossz][1] = aktualis;
119             segedUt.hossz++;
120             return segedUt;
121         }
122         else {
123             Ut s;
124             return s;
125         }
126     }
127 }
```

4.2. ábra. Út meghatározására használt függvény

Az algoritmusodban a legrövidebb utak meghatározásához szükséges az optimális útvonalak ismerete. Az élek közötti köztességi fok növelésével könnyen megtudhatod a keresett eredményt, ha pontosan ismered az irányt.

A fenti függvény meghatározza az adott út pontos éleit, és ehhez öt paramétert használ. Az első paraméter a gráf éleit tartalmazó mátrix, majd három 'int' típusú változó következik (a csúcsok száma, az indulási pont és a cél), végül pedig egy tömb, amelyben nyomon követed, hogy mely csúcsokon jártál már. A függvény első sorában létrehozol két 'Ut' típusú változót, amelyek a legjobb utat és a kapott értékeket tárolják a rekurzió során.

A függvény első ellenőrzése azt vizsgálja, hogy van-e kapcsolat az indulási pont és a cél között. Ha van kapcsolat, akkor hozzáadod az élek listáját a tárolt tömbhöz, beállítod a tömb méretét 1-re, és az "van-e ut" logikai változót igazra állítod. Ez az eset jelenti azt, amikor nem kell a függvényt újra meghívni, tehát itt véget ér a rekurzió, ha előzőleg elindult a folyamat.

A teljes függvény a rekurzióra épül, amely akkor hívja meg önmagát, ha nem kap rögtön választ, hanem tovább kell vizsgálnia. Ehhez megvizsgálod az indulási pont összes élet, hogy merre lehet továbbhaladni. Ezek az élek elvezetnek más pontokhoz, amelyekről szeretnéd megtudni, milyen messze vannak a céltól. Ez a folyamat addig ismétlődik, amíg egy vagy több megoldást találsz. Ha csak egy megoldás van, az lesz a legrövidebb út, de ha több van, akkor ki kell választanod a legjobbat. Ehhez szükséges a két változó, amelyeket az első sorban hoztál létre. Az élek listájának mérete kulcsfontosságú, mert ez döntő jelleggel befolyásolja a választ. Minél kisebb a legrövidebb út hossza, és ha valóban elértél a célhoz, annál jobb a megoldás. A ciklusban megkapod a legjobb utat, majd visszatérsz a függvényből egy kis bővítéssel. A bővítés azt jelenti, hogy hozzáadod az élt, amelyen keresztül eljutottál a legoptimálisabb szomszédhoz. Ebben az esetben növeled az élek listájának méretét, majd visszatérsz az így kapott úttal. Az így visszaadott értékek alapján beállítod az élek közötti köztessegi fokot a következő függvényben.

```

165  osszEl koztessegifokszamolas(int enronNum, int matrix[][50], Elek elek[], int elekszama) {
166      Elek* el = new Elek[elekszama];
167      for (int i = 0; i < elekszama; i++) {
168          el[i].atfedesifoka = elek[i].atfedesifoka;
169          el[i].nev1Id = elek[i].nev1Id;
170          el[i].nev2Id = elek[i].nev2Id;
171          el[i].koztessegiFok = elek[i].koztessegiFok;
172          el[i].tipus = elek[i].tipus;
173      }
174      int s = 0;
175      for (int i = 0; i < enronNum - 1; i++) {
176          for (int j = i + 1; j < enronNum; j++) {
177              int jart[50] = { 0 };
178              Ut ut = pontosut(matrix, enronNum, i, j, jart);
179              if (ut.van_e) {
180                  for (int k = 0; k < ut.hossz; k++) {
181                      s = ELIndex(el, elekszama, ut.utak[k][0], ut.utak[k][1]);
182                      el[s].koztessegiFok++;
183                  }
184              }
185          }
186      }
187      osszEl eredmeny;
188      for (int i = 0; i < elekszama; i++) {
189          eredmeny.elek[i].atfedesifoka = el[i].atfedesifoka;
190          eredmeny.elek[i].nev1Id = el[i].nev1Id;
191          eredmeny.elek[i].nev2Id = el[i].nev2Id;
192          eredmeny.elek[i].koztessegiFok = el[i].koztessegiFok;
193          eredmeny.elek[i].tipus = el[i].tipus;
194      }
195      eredmeny.hossz = elekszama;
196
197      return eredmeny;
198  }

```

4.3. ábra. Köztessegi fok számítására alkalmazott függvény

Ahogy látható, bejárom az összes lehetséges utat az egymásba ágyazott for ciklusok segítségével, és így meghatározom minden út pontos lépéseit. Ezeket az utakat aztán egy segédfüggvény (ELIndex) által feldolgozom, hogy növeljem az utakban előforduló élek köztessegi fokának számát. Ez a növelés akkor történik, ha az aktuális pontok között van út, mert a törlések és a komponensekre való szétesés megakadályozza, hogy minden pont között út létezzen. A köztessegi fok kiszámítása után az eredeti él listát visszatérítem, de a köztessegi fokokat átírom.

A következőkben bemutatom a Girvan-Newman módszer megvalósítását, amely segítségével meghatározom, hogy a legnagyobb azonos köztessegi fokkal rendelkező élek törlése során milyen gyorsan esik szét a gráf izolált pontokra.

```

467 //Girvan-Newman módszer alapján
468 szeteseett = false;
469 szamlalo = 0;
470 int meglevoElekSzama = elekszama;
471 int* komponensek = new int[enronNum];
472 while (meglevoElekSzama > 0) {
473     osszel osszesel;
474     osszesel = koztessegifokszamolas(enronNum, girvan, elek, elekszama);
475     for (int i = 0; i < elekszama; i++) {
476         elek[i].koztessegiFok = osszesel.elek[i].koztessegiFok;
477     }
478     szeteseett = false;
479
480     while (!szeteseett) {
481         int jart[50] = { 0 };
482         sorszam = legnagyobbEl(girvan, elek, elekszama);
483         for (int i = 0; i < elekszama; i++) {
484             if (elek[i].koztessegiFok == elek[sorszam].koztessegiFok) {
485                 meglevoElekSzama--;
486                 girvan[elek[i].nev1Id][elek[i].nev2Id] = 0;
487                 girvan[elek[i].nev2Id][elek[i].nev1Id] = 0;
488                 if (utakmeghatarozasa(girvan, elek[sorszam].nev1Id, elek[sorszam].nev2Id, enronNum,
489                     szeteseett = true;
490                     int jart2[50] = { 0 };
491                     int kompszama = komponensekSzama(girvan, enronNum, jart2);
492                     if (komponensek[szamlalo - 1] != kompszama) {
493                         komponensek[szamlalo] = kompszama;
494                         szamlalo++;
495                     }
496                 }
497             }
498         }
499     }
500 }

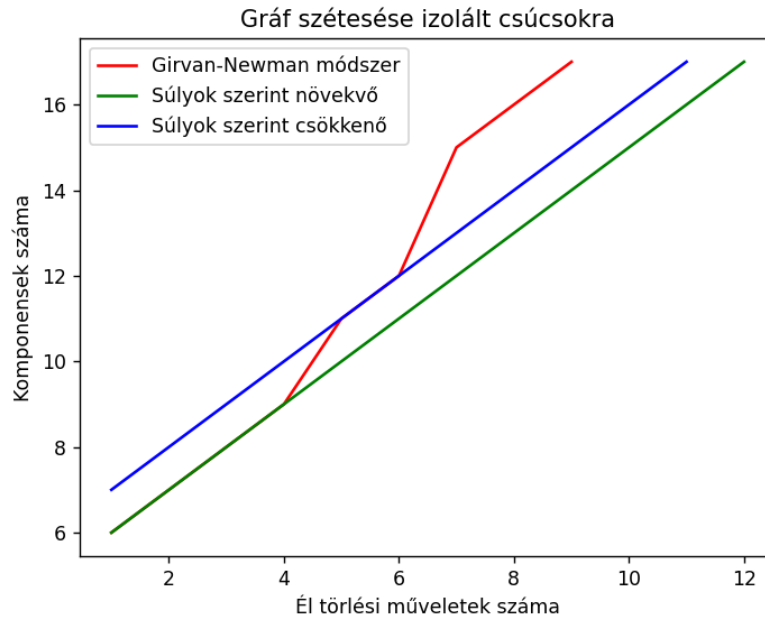
```

4.4. ábra. Girvan-Newman módszer

A fenti kódrészletben több egymásba ágyazott ciklus segítségével valósítom meg a műveleteket. Számolom a megmaradt éleket, amelyeket kitörlés esetén nullázok a girvan nevű mátrixon keresztül. A 488. sorban ellenőrzöm, hogy a törlés után van-e még út a törölt él két pontja között. Ha nincs, akkor biztosan változik a komponensek száma, amit később egy bejárás segítségével megszámlolok és visszatérítek. Ha változik a komponensek száma, akkor visszalépek a kódrészlet elejére, újra számolom a köztessegi fokokat, majd ezt ismétlem addig, amíg elfogynak az élek.

Ezután felmerült bennem a kérdés, hogy vajon a súlyok szerinti él törlése során nem szétesik-e hamarabb a gráf izolált csúcsokra. Ennek vizsgálatára elvégeztem a súlyok szerinti törlést csökkenő és növekvő sorrendben is. Minden művelet hasonlóan működött a Girvan-Newman algoritmushoz, hasonló ciklusokkal dolgoztam, és az eredmények is hasonlóak voltak, kis eltéréssel. A következő ábrákon látható eredmények rámutatnak arra, hogy a Girvan-Newman algoritmus a leggyorsabb módszer az összes él törlésére. Ugyanakkor, ha a súlyok kisebb szórásúak lennének, akkor véleményem szerint a gráf hamarabb szétesne a súly szerinti törlés esetén.

A következő lépésben a szoftver megjelenít egy kis adatvizualizációt, amely szemlélteti a hálózat kapcsolatainak törlés általi szétesését.



4.5. ábra. Összehasonlítás eredményének ábrázolása

Módszer	ciklusok száma	első éltörlés után
Girvan-Newman	9	6
Súlyokkal növekvő	12	6
Súlyokkal csökkenő	11	7

4.1. táblázat. Mérési eredmények $n = 17$ csúccsal való tesztelésre.

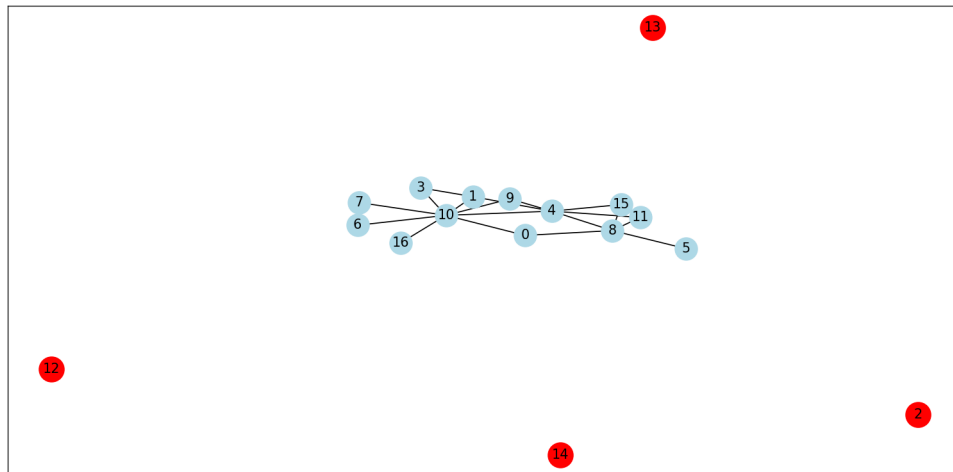
A Girvan-Newman algoritmus a leggyorsabbnak bizonyult ebben az esetben, ahogy az összehasonlító ábrán is látható. Emellett bemutattam az egyes törlési események során készült állapotokat is.

A vizualizációs program lehetőséget nyújt arra, hogy válasszunk az 3 algoritmus által generált törlési események között, így kirajzolhatjuk a súlyok szerinti csökkenő vagy növekvő sorrendben bekövetkező változásokat.

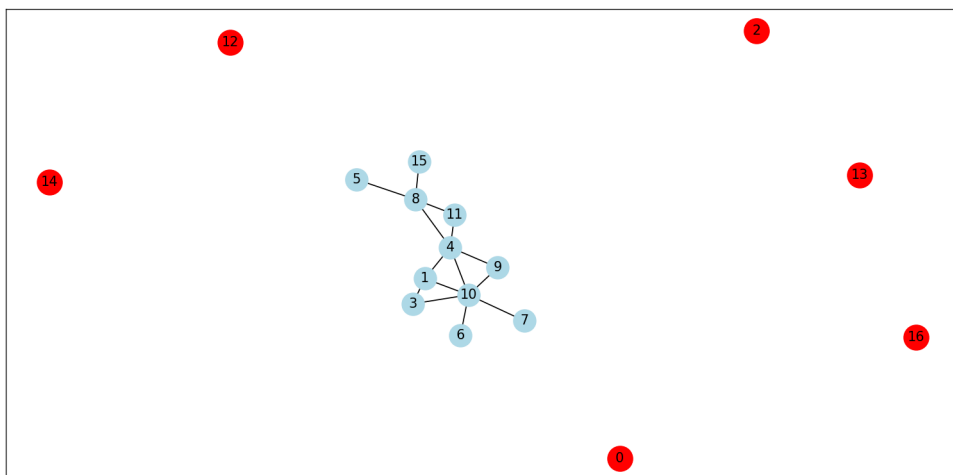
Az alábbiakban megjeleníték egy állapotot, amelyet a súlyokkal való törlési művelet során generáltam:

- Az első három ábrán látható a Girvan-Newman módszer lépései által generált aktuális gráf. Ezek közül három ciklusos törlést mutatok be: az kezdeti állapotot, majd egy törlést és végül hét törlést követő állapotot. A fenti táblázatban is látható, hogy kilenc törlés után a gráf teljesen izolált csúcsokra bomlik. A két törlés azt jelzi, hogy a komponensek száma kettővel növekedett.
- A második két ábrán a súlyok szerint növekvő sorrendben történik az élek kiválasztása és törlése.
- A táblázatban is látható, hogy a súly szerinti csökkenő és növekvő módszer nem dolgozik egyformán. Ha nem vesszük figyelembe a törlési módszer legfontosabb tulajdonságát, akkor azt gondolhatnánk, hogy a gráfban ugyanannyi különböző értékű

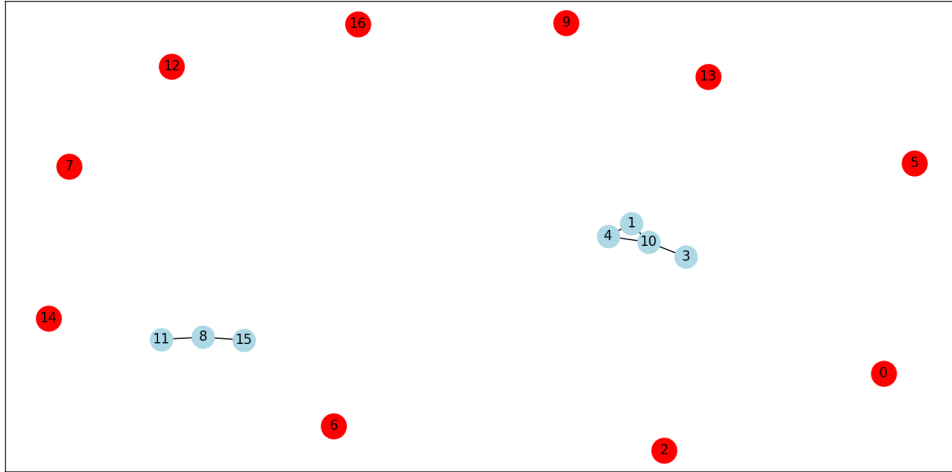
él van, tehát mindegy, hogy lentről felfelé vagy fentről lefelé haladunk a súlyokkal, mindig ugyanannyi különböző súlyú él lesz. Úgy tűnhet, hogy nincs különbség a két módszer között. Azonban itt jön a fordulat, mert csak akkor lépünk tovább egy új lépésre, ha a törlések során a komponensek száma növekszik. Ezért a csökkenő és növekvő sorrendben történő módszer eredménye nem lesz azonos. Csak bizonyos speciális esetekben lesznek az eredmények azonosak, például ha az élek súlyai azonosak, vagy ha minden különböző súlyú él törlése több komponenst eredményez.



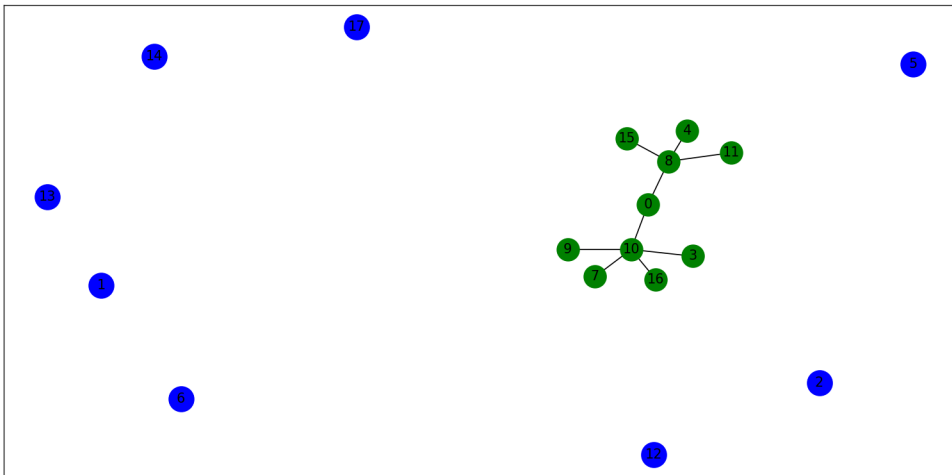
4.6. ábra. A gráf törlések előtt



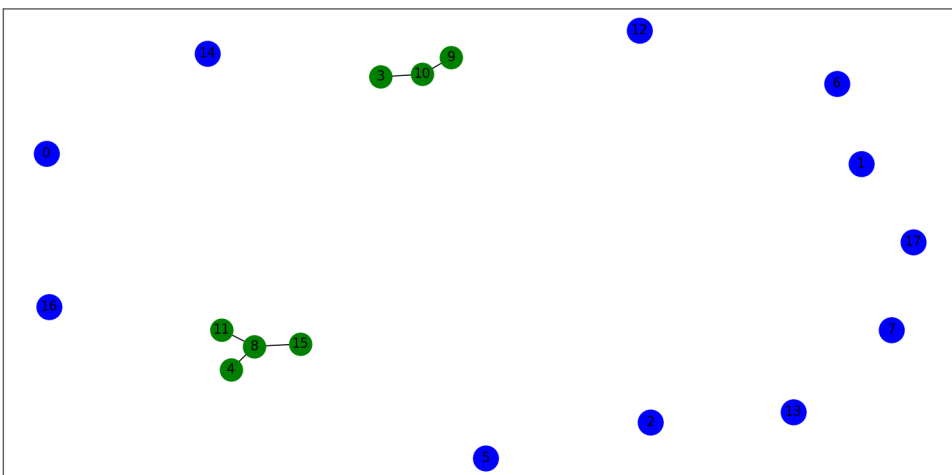
4.7. ábra. A gráf 2. Girvan-Newman módszerrel való törlés után



4.8. ábra. A gráf 7. Girvan-Newman módszerrel való törlés után



4.9. ábra. A gráf 4 törlés után



4.10. ábra. A gráf 8 törlés után

Összefoglaló

Dolgozatomban több gráfelméleti algoritmust megírtam és teszteltem. Valós adatokkal feltöltött adatbázist használtam, amit kis adatbányászattal és adatszűréssel alakítottam könnyen használható formába. Az adatokat egy egyszerű szöveges állományban tároltam, ami segítségemre lesz a gráfelméleti algoritmusok gyakorlásában az egyetemi tananyag részére.

Az adatokat szűrtem és rendeztem, majd külön figyelmet fordítottam az adatvizualizációra is. Úgy gondolom, hogy ha látom a gráfokat, könnyebben tudom megérteni azokat, és könnyebb meghatározni egy adott algoritmus eredményét vagy a valós értékét. A Python programozási nyelvet használtam az adatbányászathoz, szűrésekhez és a vizualizációhoz. Fájlkezeléseket, néhány reguláris kifejezést és a matplotlib osztályt használtam az ábrák megjelenítéséhez.

Ezután áttértem egy másik programozási nyelvre, mert úgy gondolom, hogy ha a gráfelméletet C/C++ nyelven tanuljuk, könnyebben megérthetjük és átláthatjuk hasonló nyelven megírt kódokat, mint egy idegen nyelven írt algoritmust. C++ nyelven megvalósítottam az algoritmusaimat és szoftvereimet, amik a szociális jelenségekhez kapcsolódnak. Például, a pletyka terjedését vizsgáló algoritmust vagy a csomósodási együtttható kiszámítását, amivel meghatározhatjuk, hogy mennyi esély van a barátok összekapcsolódására. Az irányított és irányítatlan gráfokat is figyelembe vettem, és bemutattam a változásokat az idő múlásával.

Az egyetemi órákon szereztem meg a gráfokkal kapcsolatos információkat és ismereteket.[Zol08]

Ezen kutatást, amelynek során vizsgálatokat végeztem, szeretném a jövőben továbbfejleszteni és magasabb szintre emelni. Az egyik célom ennek a fejlesztésnek a mesterséges intelligencia bevonása és az üzenetek hangulatának elemzése. Ezen túlmenően hosszabb távú terveim között szerepel a mesterséges intelligencia alkalmazása után az akkori üzenetek összehasonlítása a jelenlegi cégekben vagy akár kisebb szervezetekben elküldött e-mailekkel.

Github link: <https://github.com/tankotamas11/Allamvizsga2023>

Köszönetnyilvánítás

A szakdolgozatom elkészítéséhez nyújtott segítségért, a gráfelméleti gondolkodásom fejlesztéséhez nyújtott útbaigazításokért valamint a dolgozat felépítéséhez átadott tanácsokért, ötletekért szeretnék köszönetet mondani a vezetőtanárainak, Dr. Kátai Zoltánnak és Oltean-Péter Borókának akik segítségével sok újat tanulhattam az elmúlt évben. Nagyon hálás vagyok tanárainak a téma ajánlához valamint a kezdeti adatbázis beszerzése is az ő érdemük, mert ezáltal megteremtettek egy alapanyagot amin elvégezhettem számos műveletet, ennek köszönhetően fejleszthettem az algoritmikai tudásomat.

Utolsó gondolatként remélem sikeresen felhasználhatóak lesznek ezen algoritmusok és vizualizációk a jövő generációk tanításához, melynek ötletadója Dr. Kátai Zoltán tanárúr volt. Örömmel töltött el, hogy segíthetek a következő hallgatóknak a gráfelméleti tananyag vizualizációkkal való fejlesztésemmel, hisz fontos célként tűzhettem ki magam előtt ezen gondolatot ami egy plusz motivációt adott a dolgozat elkészítéséhez.

Ábrák jegyzéke

2.1. Folyamat ábra	12
2.2. Adatok számlálásának eredménye	13
2.3. Adatbányászat	14
2.4. Szűrési művelet	15
2.5. Rendezés kronológiailag	16
2.6. 1000 üzenet utáni hálózat	17
2.7. Összes üzenet utáni hálózat	17
2.8. Csucspontok méretének beállítása és kijarzolása	18
2.9. Csucspontok kijarzolása irányított gráfként	18
2.10. Kis nagyítás a gráf középpontjára	19
3.1. Szoftver lépései	20
3.2. Gráf a szóbeszédhez	21
3.3. két pont közötti út távolság meghatározása	22
3.4. Utak meghatározása és középső pont kiderítése	23
3.5. Eredmény kiírása	23
3.6. Együttható kiszámításához használt függvény	24
3.7. Csomósodási együttható meghatározása időközönként	25
3.8. Kapcsolat struktúra	26
3.9. Változás az együtthatókban	26
3.10. Gráf a híd magyarázatához	27
3.11. Gráf a lokálishíd magyarázatához	27
3.12. Híd meghatározása	28
3.13. Él struktúra	28
3.14. Átfedéshez használt példa gráf	29
3.15. Átfedés számításához használt függvény	29
4.1. Modulok	30
4.2. Út meghatározására használt függvény	31
4.3. Köztességifok számítására alkalmazott függvény	32
4.4. Girvan-Newman módszer	33
4.5. Összehasonlítás eredményének ábrázolása	34
4.6. A gráf törlések előtt	35
4.7. A gráf 2. Girvan-Newman módszerrel való törlés után	35
4.8. A gráf 7. Girvan-Newman módszerrel való törlés után	36
4.9. A gráf 4 törlés után	36
4.10. A gráf 8 törlés után	36

Táblázatok jegyzéke

4.1. Mérési eredmények $n = 17$ csúccsal való tesztelésre.	34
--	----

Irodalomjegyzék

- [Adi05] Jitesh Shetty Jafar Adibi. Discovering important nodes through graph entropy the case of enron email database. *LinkKDD '05: Proceedings of the 3rd international workshop on Link discovery*, pages 74–81, aug 2005.
- [KPX11] Matt Hohensee Kelly Peterson and Fei Xia. Email formality in the workplace: A case study on the enron corpus. *Proceedings of the Workshop on Language in Social Media (LSM 2011)*:86–95, jun 2011.
- [Zol08] Káta Zoltán. *Gráfelméleti algoritmusok*. Scientia Kiadó, 2008.