



УНИВЕРЗИТЕТ У НОВОМ САДУ  
ПРИРОДНО-МАТЕМАТИЧКИ  
ФАКУЛТЕТ  
ДЕПАРТМАН ЗА МАТЕМАТИКУ И  
ИНФОРМАТИКУ



# Извештај

-Социјалне мреже-

Главоњић Јован

552/17

## Садржај

1. Окружење: .....	2
2. Учитавање графова .....	3
*Конвертовање графа приликом учитавања: .....	4
3. Кластерабилност графова.....	6
*Проналажење компоненти: .....	6
*Креирање скупа графова од скупа кластера: .....	8
*Мрежа кластера: .....	8
*Провера кластерабилности: .....	9
4. Тестирање на реалној мрежи .....	11









## 1. Окружење:

Тестирање мрежа је рађено у програмском језику Пајтон 3.7.

За рад је кориштена библиотека: networkx.

Главни модул који покреће цео програм се назива maintTesting().

Сви модули као и фајлови за тестирање се налазе у пакету: socialnetworking (слика1.1).

	__pycache__	8/19/2019 11:30 PM	File folder	
	detectingClusters	8/19/2019 11:09 PM	Python File	5 KB
	loadingGraph	8/19/2019 4:54 PM	Python File	4 KB
	mainTesting	8/19/2019 11:30 PM	Python File	1 KB
	soc-sign-bitcoinalpha	10/6/2017 5:42 AM	Microsoft Excel C...	492 KB
	soc-sign-epinions	11/3/2009 11:45 PM	Text Document	10,980 KB
	soc-sign-Slashdot090221	11/3/2009 9:36 PM	Text Document	6,917 KB
	wiki-RfA	7/10/2018 11:25 PM	Text Document	48,937 KB

слика1.1

## 2. Учитавање графова

Након покретања програма унутар главног модула, кориснику се нуди могућност да одабере који граф жели да учита и тестира. На основу одабира корисника функција `loadGraph` (слика2.1) позива једну од помоћних функција за читање одређеног графа.

```
def loadGraph():
    try:
        gr = nx.DiGraph()
        options = [1, 2, 3, 4]
        choice = int(input("Enter which Graph do you want to load:\n"
                           "\t1 ---- Epinions\n"
                           "\t2 ---- Wikipedia\n"
                           "\t3 ---- Slashdot\n"
                           "\t4 ---- Bitcoin\n"))

        if choice in options:
            print("Loading..... ")
            if choice is 4:
                gr = loadBCoin()
                print("Graph is ready!")
            if choice is 1:
                gr = loadEpinionsandSlash(1)
                print("Graph is ready!")
            if choice is 3:
                gr = loadEpinionsandSlash(3)
                print("Graph is ready!")
            if choice is 2:
                gr = loadWikipedia()
                print("Graph is ready!")

        else:
            print("Please enter a valid number (1, 2, 3 or 4)")
            loadGraph()

        return gr
    except ValueError:
        print("Please enter valid number (1, 2, 3 or 4)")
        loadGraph()
```

слика2.1

С' обзиром на то да су фајлови за читање Epinios и Slash мрежа једнаки по конструкцији, користимо једну помоћну функцију за оба фајла (*слика2.2*).

```
def loadEpinionsandSlash(choice):
    gr = nx.DiGraph()
    txt1 = 'soc-sign-epinions.txt'
    txt2 = 'soc-sign-Slashdot090221.txt'
    final = ""
    if choice is 1:
        final = txt1
    else:
        final = txt2
    with open(final, 'r') as file:
        for _ in range(4):
            next(file)
        for line in file:
            data = line.split()
            af1 = data[2].strip()
            af = ""
            if af1 == "-1":
                af = "negative"
            else:
                af = "positive"
            gr.add_edge(data[0].strip(), data[1].strip(), affinity=af)
    return transformDirectedToUndirected(gr)
```

*слика2.2*

**\*Конвертовање графа приликом читавања:**

Свака помоћна функција за читање графова враћа конвертован неусмерен граф, позивајући функцију `transformDirectedToUndirected()` (слика2.3). Принцип је једноставан, пролазимо кроз учитани граф, с' обзиром на то да је учитани граф директан, може да се деси да чвор **a** показује на чвор **b** и обрнуто, тј да су два чвора повезана линком два пута. Дата функција провера да ли се тако нешто налази у графу и узима негативан афинитет као примарни и њега поставља на линк. У случају да су обе дирекције позитивне, линк добија позитивни афинитет.

```
def transformDirectedToUndirected(gr):
    undirected = nx.Graph()
    undirected.add_edges_from(gr.edges(), affinity="")
    for u, v, d in gr.edges(data=True):
        af = ""
        if (v, u) in gr.edges:
            af = gr[v][u]['affinity']
        if gr[u][v]['affinity'] == "negative" or af == "negative":
            undirected[u][v]['affinity'] = "negative"
        else:
            undirected[u][v]['affinity'] = "positive"
    return undirected
```

слика2.3

### 3. Кластерабилност графова

#### \*Проналажење компоненти:

Проналажење компоненти у графу се врши помоћу мало измењеног оригиналног **BFS** алгоритма (*слика3.1*). Разлика је у томе што радимо са означеним неусмереним графовима, тако да ће једну компоненту у графу представљати чворови конструисани тако да комшије чвора морају имати позитиван афинитет, у супротном не припадају компоненти. После ћемо другом функцијом поредити оригиналан граф и компоненте тражећи негативан афинитет у истим.

Напомена: пошто пролазимо кроз листу комшија и бришемо елементе из исте, како не би дошло до грешке и погрешних података јер мењамо листу кроз коју пролазимо, користимо погодност пајтона, тако да не пролазимо кроз листу на класичан начин. На пример: (*for i in list*), него користимо (*for i in list[:]*), чиме конструишемо копију листе са којом радимо, тако избегавамо могући проблем.

```

def components(gr):
    components = indetifyCmp(gr)
    return components

def indetifyCmp(gr):
    visited = set()
    components = set()
    print("Adding components ...")
    for v in gr.nodes:
        if v not in visited:
            components.add(bfs(v, visited, gr))
    number = str(len(components))
    print("Graph is made out of "+ number + " components")
    return frozenset(components)

def bfs(v, visited, gr):
    compSet = set()
    queue = list()
    compSet.add(v)
    visited.add(v)
    queue.append(v)
    while len(queue) != 0:
        current = queue.pop(0)
        neighbours = list(nx.neighbors(gr, current))
        for neighbour in neighbours[:]:
            afn1 = gr.get_edge_data(current, neighbour)
            afn2 = gr.get_edge_data(neighbour, current)
            if afn1['affinity'] == 'negative' or afn2['affinity'] == 'negative':
                neighbours.remove(neighbour)
        for neighbour in neighbours:
            if neighbour not in visited:
                compSet.add(neighbour)
                visited.add(neighbour)
                queue.append(neighbour)
    return frozenset(compSet)

```

слика 3.1



### \*Креирање скупа графова од скупа кластера:

Функција која се бави овим проблемом, прима као параметре оригиналан граф и скуп кластера (*слика3.2*). Једноставним алгоритмом прави граф од сваког кластера и враћа скуп кластера. Принцип је заснован на томе да у празан скуп додамо све чворове кластера (пролазимо фор петљом кроз све кластере и кораке радимо засебно за сваки кластер), направимо копију оригиналног учитаног графа и након тога све чворове који нису у кластеру а јесу у графу изbacимо из копије. Тиме добијамо граф који је кластер.

```
def clusterNetworkCreate(clusters, gr):
    print("Making networks from clusters....")
    networks = set()
    for cl in clusters:
        allnodes = set()
        for v in cl:
            allnodes.add(v)
        newGraph = gr.copy()
        for v in list(newGraph.nodes[:]):
            if v not in allnodes:
                newGraph.remove_node(v)
        networks.add(newGraph)
    return networks
```

слика3.2

### \*Мрежа кластера:

Након претварања кластера у графове, креирамо мрежу чији ће чворови у ствари бити кластери (*слика3.3*). Сваком кластеру дајемо име и смештамо га у граф као чвор. На основу оригиналног графа проверавамо да ли су кластери повезани међусобно. Ако јесу конструишемо линк између њих. Сви линкови (ако постоје) између кластера су негативни. То је логичан закључак јер се у кластеру налазе чворови такви да суседи имају позитиван афинитет, ако су **а** и **б** суседи а имају негативан афинитет долазе из различитих кластера. У случају да поредимо један те исти кластер, прескачемо проверу суседа.

```

def giantNetworkClusters(g, clusters):
    network = nx.Graph()
    i = 1
    for c in clusters:
        c.graph['name'] = "Cluster " + str(i)
        i = i + 1
        network.add_node(c)

    for c1 in clusters:
        for c2 in clusters:
            if c1 == c2:
                continue
            connected = False
            j = 0
            while not connected and j < len(c1.nodes):
                node = list(c1.nodes)[j]
                j = j + 1
                for node2 in c2:
                    if node in nx.neighbors(g, node2):
                        connected = True
            if connected:
                network.add_edge(c1, c2)
    nx.set_edge_attributes(network, "negative", "affinity")

```

слика3.3

### \*Провера кластерабилности:

Функција која проверава кластерабилност и приказује антикоалиције, као параметар прима скуп кластера који су представљени као графови (слика3.4). Проверавамо за сваки кластер да ли садржи линк негативног афинитета. Ако садржи граф није кластерабилан, а сваки негативни линк, тј. грана је линк који припада антикоалицији и треба га обрисати како би наш граф постао кластерабилан.

```

def isMyGraphClusterable(clusters):
    antiCoalitions = list()
    negativeEdges = list()
    coalitions = list()
    for cluster in clusters:
        for (u, v, d) in cluster.edges(data=True):
            if d['affinity'] == "negative":
                negativeEdges.append((u, v))
                antiCoalitions.append(cluster.nodes)
            else:
                coalitions.append((u, v))
    if len(antiCoalitions) == 0:
        print("Given graph is clusterable")
    else:
        print("Given graph is not clusterable")
        print_("Links for removal: ")
        for link in negativeEdges:
            print(link)

    return coalitions

```

слика3.4

## **4. Тестирање на реалној мрежи**

Приказ тестирања на мрежи биткоина се налази у фајлу: test.txt