Federal Autonomous Educational Institution of Higher Professional Education
National Research University “Higher School of Economics”
Faculty of computer science
Educational program Data science and busyness analytics
Baccalaureate

01.03.02 Applied Mathematics and Information Science

Report
Hands-on training

Conducted by
Antonovsky Gregory
And
Vladimirov Leonid

Verified by:

_____
(occupation, full name, chief of organization/ NRU HSE)

_____ _____
(grade) *(signature)*

_____
(dates)

The Problem

For a given VK community, detect sub-communities inside and find dependencies.
Used technologies: Python, VK API, NetworkX and Pandas.

For this task we chose to investigate the following community:

https://vk.com/artprostranstvo57

Obtaining data and graph creation

We decided to collect all the subscriptions of the followers of the target page since this allows us to choose any dependency criterion and easily collect any additional data. The initial gathered data is organized as a CSV file where the first entry of each row is the User ID and the rest of the row are ID's of the pages he follows - a key-value pair.

To collect the data we used the VK API along with the requests library.
The VK API contains all of the methods used to gather this data.
groups.getMembers, groups.get
Are the main methods used to gather the needed data. "groups.getMembers" returns subscriber ID's for a page and "groups.get" returns ID's of pages followed by the user in question.

Obviously these methods have restrictions - both of these methods can only return 1000 objects per call and the VK API gives us only 3 requests per second.
So to collect all of the data we need to first find the amount of items we need to collect and then calculate how many requests we need to make to gather everything. And each new request should have an "offset" parameter - which is a part of the VK API to gather only new information about a page or a user.

To calculate the amount followed pages we use this function:

```
def _get_followed_pages_amount(user_id):
    params = (
        ('access_token', api_token),
        ('v', '5.65'),
        ('user_id', '{0}'.format(user_id)),
    )

    response = requests.get('https://api.vk.com/method/groups.get', params=params)
    response = response.json()
    return response[“response"]["count"]
```

To calculate the amount of members we use this function:

```
def _get_subscriber_amount(community_name):
    params = (
        ('access_token', api_token),
        ('v', '5.65'),
        ('group_id', '{0}'.format(community_name)),
        ('fields', 'members_count'),
    )

    response = requests.get('https://api.vk.com/method/groups.getById', params=params)
    response = response.json()

    return response["response"][0]["members_count"]
```

Once we have acquired the amount of group members or the amount of followed pages we can processed to the collection phase.
Here is the function that returns an array of pages that a given user follows:

```
def _get_user_subscriptions(user_id):
    try:
        # The VK api returns 1000 results a time so we need to make multiple requests
         #   to collect all data
        amount_of_followed_pages = _get_followed_pages_amount(user_id)

        if amount_of_followed_pages <= 1000:
            params = (
                ('access_token', api_token),
                ('v', '5.65'),
                ('count', '1000'),
                ('user_id', '{0}'.format(user_id)),
            )
            response = requests.get('https://api.vk.com/method/groups.get', params=params)

            response = response.json()
            subscriptions = response["response"]["items"]
```

```
            return subscriptions

        else:
            subscriptions = []
            # calculate the amount of required requests
            needed_requests = math.ceil(amount_of_followed_pages / 1000)
            offset = 0
            for i in range(0, needed_requests):


                # if this is the third request we need to sleep for one second to keep
                # getting responses from VK
                if i % 3 == 0:
                    time.sleep(1)

                params = (
                    ('access_token', api_token),
                    ('v', '5.65'),
                    ('count', '1000'),
                    ('offset', '{0}'.format(offset)),
                    ('user_id', '{0}'.format(user_id)),
                )
                response = requests.get('https://api.vk.com/method/groups.get', params=params)

                response = response.json()
                subscriptions += response["response"]["items"]

                offset += 1000

            return subscriptions

    # if a user has a private page or doesn't allow us to view his subscriptions
    # we raise a KeyError
    # Which is handled in _get_user_data()
    except KeyError:
        raise KeyError
```

This function raises a KeyError if it fails to collect the needed information. This may happen because of a user's page settings. This error is handled in the "_get_user_data(members)" function. Which returns a dictionary where each key is the User ID and the value is an array of followed pages. As we can see here we sleep each third request for a second to avoid getting an error from the VK servers which would tell us that we are making more than 3 requests per second.

```
def _get_user_data(members):
```

```
    data_dict = {

    }

    for i in range(0, len(members)):
        if i % 3 == 0:
            time.sleep(1)
        try:
            subscriptions = ','.join(map(str, _get_user_subscriptions(members[i])))
            # the key is the user id and the value are his subscriptions
            data_dict[members[i]] = subscriptions
        except KeyError:
            pass

    return data_dict
```

"_get_user_data(members)" is called from our "main_call(community_-name)" function:

```
# Main function. Gets all members for a community and finds all followed pages
# for each member
def main_call(community_name):
    members = _get_members_list(community_name)
    print("gathered members.")
    data = _get_user_data(members)
    print("finished gathering all data.")

    # after all data is collected we write this data to a CSV file
    with open('data.csv', 'w') as f:
        for key in data.keys():
            f.write("%s,%s\n" % (key, data[key]))
```

After running the this function we get the "data.csv" file from which we obtain our graph. We choose the dependency criterion to be the similarity of each user's subscription from [0, 1]. To accomplish this we use the inbuilt "difflib" library.

This is the function that compares each user with everyone else:

```
def DataToGraphDict(data, threshold): # compares users by comunities that they follow
    g = {}
    for i, (k1, v1) in enumerate(data.items(), 1):
        for k2, v2 in list(data.items())[i:]:
            if k1 != k2:
                sm = difflib.SequenceMatcher(None, v1, v2)
                if sm.ratio() > threshold:
```

```
            g[k1] = g.get(k1, []) + [k2]
            g[k2] = g.get(k2, []) + [k1]
    print("data processing: ", i, "/", len(data.keys()))
return g
```

Now we have obtained a file that contains everything we need to proceed to the analysis stage.

Karger's algorithm

The algorithm is used for cauterization in a network. It finds **N** sets of nodes in a graph such that number of edges between them is minimized. Therefore, it can be used for community detection.

This algorithm is randomized and computes the minimum cut of a connected graph. It continuously contracts randomly chosen edges, until only **N** nodes remain. By iterating this set of operations for a sufficient number of times, we can achieve a high probability of finding a minimum cut.

The pseudo code for the algorithm:

*While there are more than **N** groups: do*
Pick random edge $(u, v) \in E(G)$;
Merge u and v;
*Output nodes with **N** groups assigned to them*

Implementation:
Using python 3.6.5

```
# Karger's Algorithm
def kargerMinCut(graph, n):
    gn = { k : str(i) for (i, k) in enumerate(graph.keys(), 0)}
    while len(graph) > n:
        v = random.choice(list(graph.keys())) # the key
        w = random.choice(graph[v]) # the list of connections

        # assigning the same label to the nodes that are being grouped
        label = gn[w]
        for key in gn.keys():
            if(gn[key] == label):
                gn[key] = gn[v]

        contract(graph, v, w) # merge together

    mincut = len(graph[list(graph.keys())[0]]) # calculate mincut
    return (gn, mincut)

def contract(graph, v, w):
    for node in graph[w]:  # merge the nodes from w to v
        if node != v:  # we don't want to add self-loops
            graph[v].append(node)
        graph[node].remove(w)  # delete the edges to the absorbed
        if node != v:
            graph[node].append(v)
    del graph[w]  # delete the absorbed vertex 'w'
```

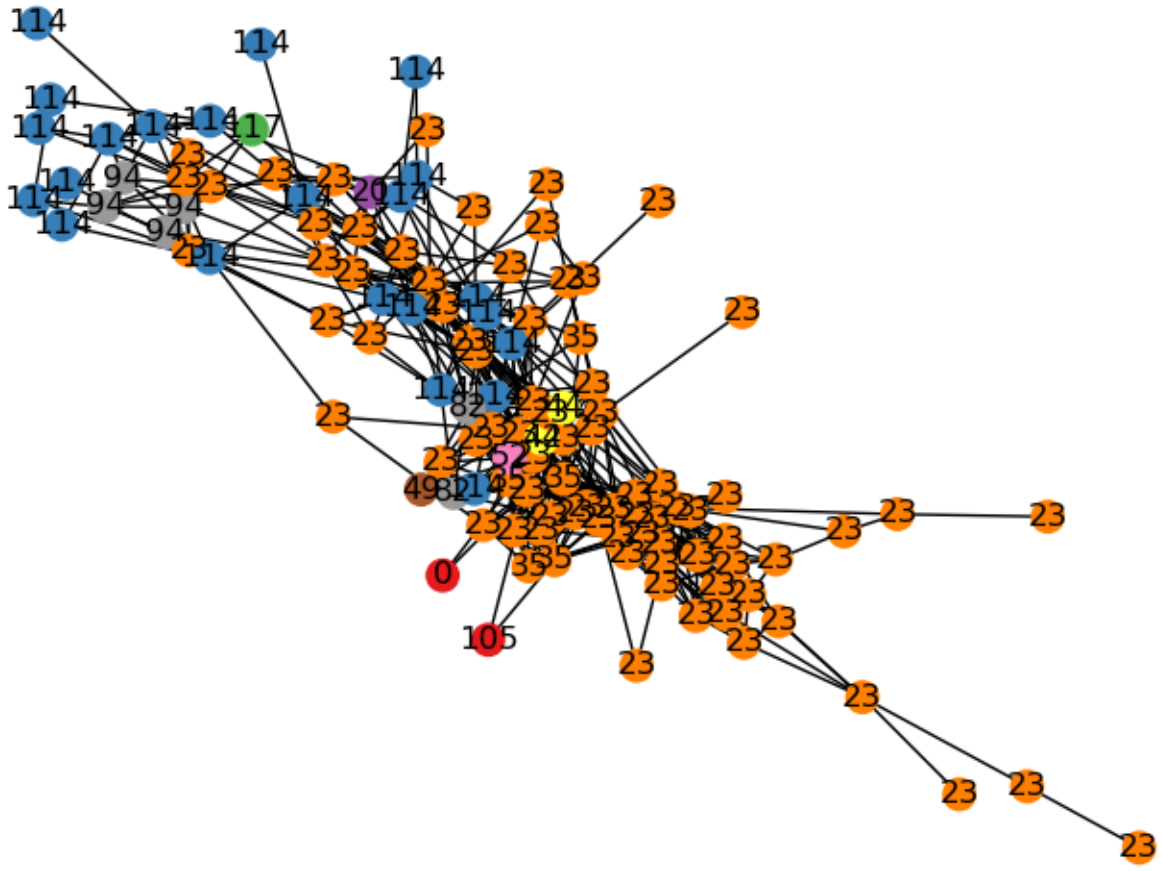
Results

We applied Karger's Algorithm to the dataset with N = 12 (number of groups) and visualized the results.
The minimum number of cuts was 2.
The distribution of users in dataset to the groups was the following

(group ID : number of users)

{'0': 1, '23': 79, '35': 5, '114': 23, '20': 1, '44': 2, '82': 2, '49': 1, '52': 1, '94': 4, '105': 1, '117': 1}

The image below shows the visualization of the dataset divided into groups. Each group is colored with a unique color. Each node is labeled with the group ID it was assigned to.



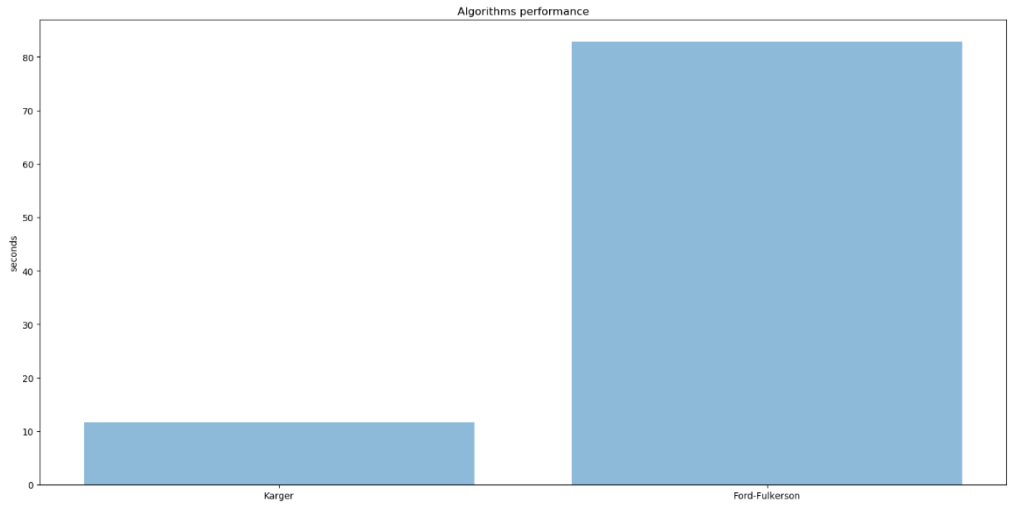
The run time of the algorithm was approximately 14 seconds for the set of 127 nodes.

In conclusion, Karger's algorithm managed to divide the given dataset into distinct subsets and many smaller ones that do not have a strong connection to others.

Ford-Fulkerson Algorithm

We also compared Karger's algorithm to a maximum flow-based clustering algorithm - the Ford-Fulkerson algorithm.
Our implementation uses BFS to find out if there is a path from two points in graph.
The Ford and Fulkerson theorem states that the maximum flow from the vertex **s** to the vertex **t** is equal to the value of the minimum cut separating **s** and **t**. We take the edges on the graph as channels of communication, the flow on every edge can be considered as the amount of information passed through the edge, with the vertex **s** as the information source and the vertex **t** as the information receiver. The maximum flow from the vertex **s** to the vertex **t** reflects the maximum amount of information passing through all paths between the vertices **s** and **t**, not only through one path (for example, the shortest path). This way, the maximum flow carries the "global relations" between vertex **s** and **t**.

Running time comparison between Ford-Fulkerson and Karger's algorithms on the same dataset:



Links

Github

https://github.com/onlinex/Community-detection-in-VK

Max-Flow-Based Similarity Measure for Spectral Clustering

https://pdfs.semanticscholar.org/4707/ef971b76f51a43ea358231606cee1c23a89d.pdf

Community for the dataset

https://vk.com/artprostranstvo57