Community Detection in the Russian social media VKontakte & Computational Complexity Analysis

Alexey Elkin

August 2020

1 Introduction

The purpose of this project is to create a program that detects discrete groups in large graphs and apply it to categorize users into social groups on the Russian social media VKontakte.

The program is based on the Girvan-Newman algorithm and implemented in Python[3]. The partitioned graphs are drawn using *NetworkX*[4].

Data from *Zachary's Karate Club*[2] was used to test the program, and the canonical result was produced, demonstrating that the program works correctly when applied to small graphs.

Data harvested from VKontakte was converted into a .CSV file with the rows representing "friendship status" and the two columns representing the friends. The data consisted of 407 users/nodes, and 16K connections between them. After 710 iterations, no new groups were detected. Further iterations removed outer nodes instead.

The output dendrogram demonstrates that the dataset can be split into 4 or 5 social groups. The results generated by my code were compared to a standard library program based on the Louvain method[1], which produced practically identical end results. This demonstrated the robustness of the my program and confirmed that it works appropriately even for very large graphs

Additionally, the algorithmic complexity of each type of operation was assessed. This showed that the overall time complexity is $T(n) = O(n^3)$.

The resulting data is very versatile; it can be used by marketing companies to target users based on their interests, group membership, and even political opinions.

1.1 Undirected Graphs

The dataset extracted from VKontakte can be represented by an undirected graph in which the users are represented by nodes and the connections between users - ”friendship status” - is represented by an edge connecting two nodes together. Partitioning the graph with the Girvan-Newman algorithm reveals the underlying structure and allows discrete social groups to be identified. An undirected graph, G, is a pair (V, E) where V is a non-empty set of nodes, and a set of undirected edges can be represented by

$$E \subseteq \{\{a, b\} | a, b \in V\} \tag{1}$$

Let $G = (V, E)$ be a graph. Two nodes, $u, v \in V$ are adjacent or neighbors if $\{u, v\} \in V$. If the nodes of the graph G are $v_1, v_2, ..., v_n$, the adjacency matrix of the graph, A_G, is an $n * n$ -1, 0 matrix such that

$$A = \begin{Bmatrix} -1 \text{ if } \{v_i, v_j\} \notin E \\ 0 \quad \text{if } \{v_i, v_j\} \in E \end{Bmatrix} \tag{2}$$

1.2 Graph Implementation in Python

```
df = pd.read_csv('vk_links.csv')
users = set(df['out_f']).union(set(df['in_f']))
users = list(users) #converting the users set to a list
index = np.array(range(len(users)))
graph = -np.ones((491,491))

for idx, row in df.iterrows():

    index1 = users.index(row['out_f'])
    index2 = users.index(row['in_f'])

    graph[index1][index2] = 0
    graph[index2][index1] = 0
```

2 The Girvan-Newman algorithm

The Girvan-Newman algorithm is a hierarchical method used to detect communities in complex systems. The method is as follows:

1. The betweenness of all existing edges in the network is calculated first.
2. The edge(s) with the highest betweenness are removed
3. The betweenness of all edges affected by the removal is recalculated.
4. Steps 2 and 3 are repeated until no edges remain.

This algorithm serves as the basis for graph categorization in my investigation.

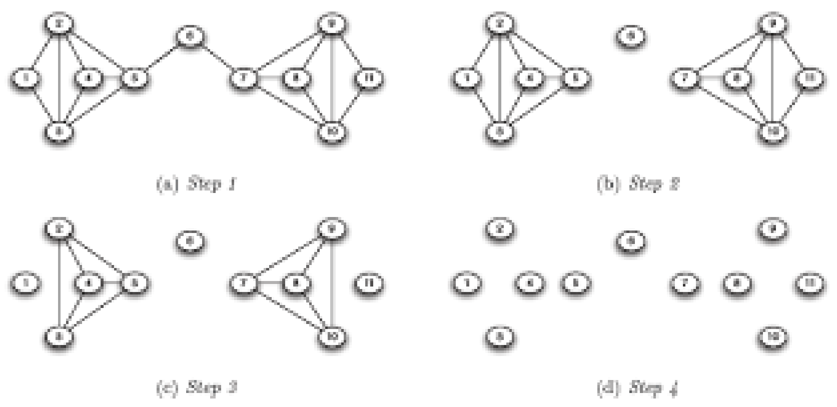

Figure 1: The Girvan-Newman algorithm applied to a graph[2]

3 Flow and Betweenness

Betweenness is a measure of centrality in a graph based on the number of shortest paths between nodes. The *betweenness* of an edge is defined as the total amount of *flow* it carries, taking into account the *flow* between all pairs of nodes using this edge.

The *betweenness* centrality of a node is give by the expression:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{s_t}(v)}{\sigma_{s_t}} \tag{3}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v
The concept of *flow* is defined as follows: For any pair of nodes A and B in the graph that have a connection between them, one unit of "fluid" flows along the edge between A and B. If there are no edges connecting A and B, zero units of "fluid" *flow* between them. The *flow* between A and B divides itself evenly along all possible shortest paths from A to B, so if there are k shortest paths between A and B, $1/k$ unit of *flow* passes along each path.

3.1 Computing Betweenness values

The method for computing betweenness values for each node in a network is as follows:

1. Perform a breadth-first search of the graph, starting at A
2. Determine the number of shortest paths from A to each other node
3. Based on these numbers, determine the amount of flow from A to all other nodes that use each edge

3.2 Modified Breadth-First Search

The breadth-first search algorithm is as follows: Start from a pre-specific source node, visit the node and its neighbors' then, for each neighbor, visit its neighbors' and so on until no more nodes can be explored. However, this modified version of the breadth-first search works slightly differently. The step at which a node is accessed is saved for each node, and the a new graph without any connections from the 1st step is created.

```
def breadth_first(graph, root):
    n = graph.shape[0]
    new_graph = -np.ones((n, n))
    level = -np.ones(n,int)
    nodes = np.array(range(n))
    current_level = 0
    level[root] = current_level
    while current_level in level:
        for node in nodes[level == current_level]:
            index = nodes[(graph[node]>=0) & (level == -1)]
            new_graph[node,index] = 0
            new_graph[index,node] = 0
            level[index] = current_level + 1
        current_level += 1
    return new_graph, level
```

3.3 Computing Shortest Paths in Python

This function calculates the values of the shortest paths to each node (shortest path = the sum of incoming paths). The input to this function consists of a graph which includes the root node. Every node has its own designated "level", which is its distance from the root node.

```
def pointer(new_graph, level, root):

    n = new_graph.shape[0]
    output = np.zeros(n)
    output[root] = 1
    nodes = np.array(range(n))

    for x in range(int(level.max()+1)):
        for y in nodes[level == x]:
            index = nodes[(new_graph[y] >= 0) & (level == x+1)]
            output[index] += output[y]
    return output
```

3.4 Computing Flow in Python

The flow values from a starting node A to all other nodes in the network are determined by working up from the lowest layers of the breadth-first search and dividing up the flow above a node in proportion to the number of shortest paths coming into it on each edge.

```
def flow(new_graph, level, root, output):

    n = new_graph.shape[0]
    nodes = np.array(range(n))
    flows = np.ones(n)

    for x in np.array(range(np.max(level)+1)[::-1]):
        for y in nodes[level == x]:
            for i in nodes[(new_graph[y] >= 0) & (level == x-1)]:
                new_graph[i][y] = flows[y] * output[i] / output[y]
                new_graph[y][i] = flows[y] * output[i] / output[y]
                flows[i] += flows[y] * output[i] / output[y]

    return new_graph
```

3.5 Removing edges with the highest betweenness values

The edge(s) with the highest betweenness values are removed using the following function:

```
def removeEdge(graph):
    max_edge = graph.max()
    graph[graph == max_edge] = -1
    return graph
```

4 Algorithmic Complexity Analysis

The arguments of the program consist of the graph **n** and the number of iterations **K**. The graph is stored as a matrix of size n^2. Since the number of operations performed is dependent on the structure of the graph, the worst-case time is considered.

1. `def breadth_first(graph, root):`

   $2n^2 + n$ comparison operators
   $n^2 + 2n$ assignments

2. `def pointer(new_graph, level, root):`

   $2n^2$ comparison operators
   n^2 assignments

3. `def flow(new_graph, level, root, output):`

   $3n^2$ comparison operators
   $3n^2$ multiplication / division
   n^2 addition

4. `def summary(graph, iterations):`

   $9n^3 + n^2$ comparison operators
   $3n^3 + 2n^2$ addition

Overall, the algorithmic complexity across all categories is $O(n) = n^3$. Therefore, the number of operations for each iteration is comparable to the number of nodes in the graph cubed. The reason why the complexity of different operators was considered individually for each function is because of the varying speeds at which they are performed. For example, multiplication and addition operators are faster than comparison operators.

5 Karate Club Test Graph Visualisation

"Zachary's Karate Club" is a popular placeholder dataset used to test graph partitioning algorithms. *Figure 2* is a visualisation of the unpartitioned graph. Upon inspection, it is evident that the social hierarchies in the club are split into two distinct groups connected by a narrow "bridge" which connects them. The fact that my program produces the canonical partitioning of the Karate Club graph demonstrates that it works correctly.

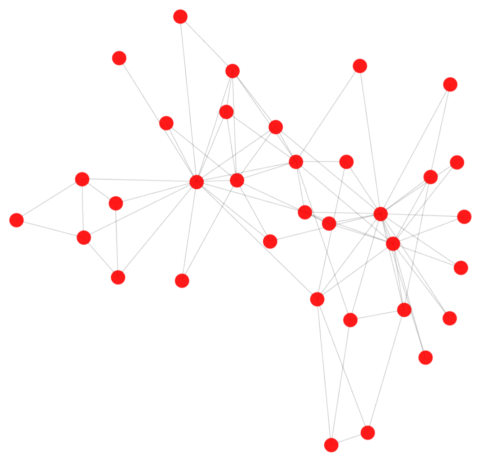

Figure 2: The unpartitioned Graph

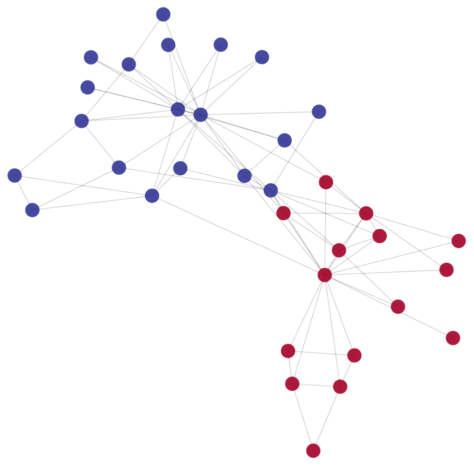

Figure 3: The partitioned Graph

6 VKontakte Graph Visualisations

The program was fed the data extracted from VKontakte (407 users/nodes, 16K connections). The program was run up to the 710th iteration, at which no more unique groups appeared and further iterations simply removed outer nodes. The images were generated using the *networkx* library [4].

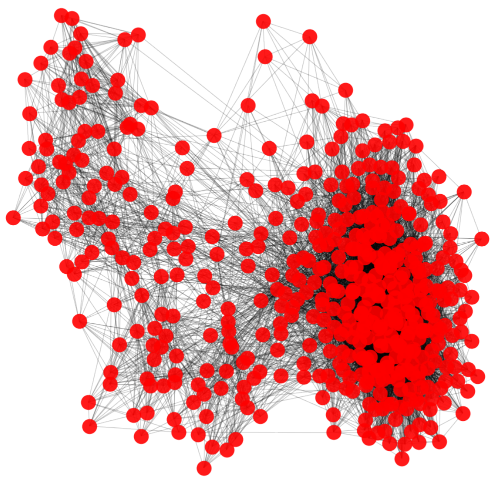

Figure 4: A visualisation of the original graph

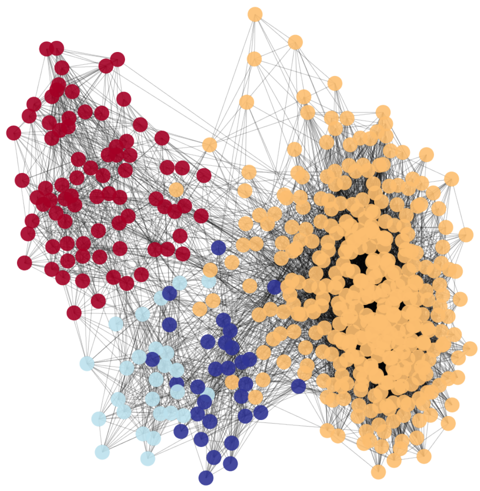

Figure 5: The graph partitioned into 4 sections (Girvan-Newman Algorithm)

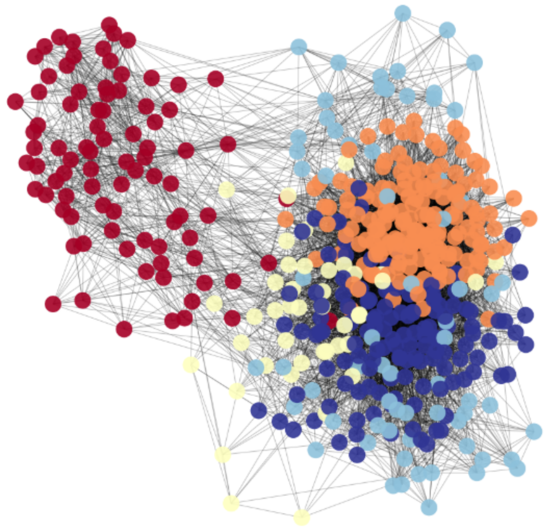

Figure 6: The graph partitioned into 5 sections (Girvan-Newman Algorithm)

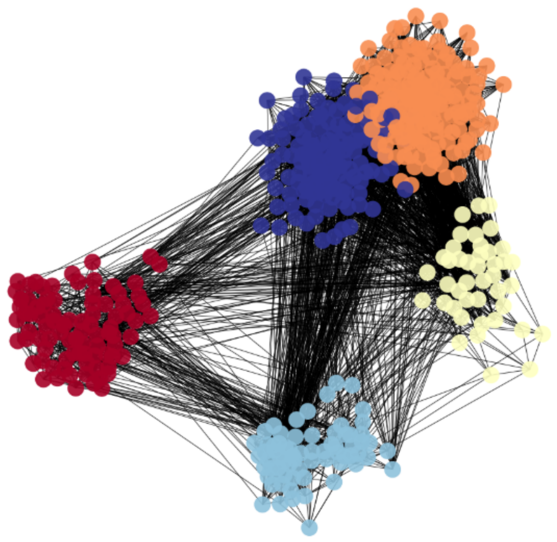

Figure 7: The graph partitioned into 5 sections (Louvain Method)

7 Conclusion

Description The images generated by the graph categorization program demonstrate that the social network used in the investigation can be separated into four or five discrete social groups, which are represented by different colors. Large discrete groups can be seen on both sides of the network, with a smaller central group that can be divided into two sub-groups. Overall, the examples demonstrate that my implementation of the Girvan-Newman algorithm works as intended on classical test graphs, and is able to categorize even very large graphs with thousands of nodes

Comparison to Louvain method The result produced by the Louvain method is almost identical to the graph produced by my program, with the exception of the fact that the largest group was halved by the Louvain method algorithm. The fact that my program functions very similarly suggests that it is also suitable to be used with large graphs.

Usefulness People categorized into the same groups often have something in common. For example, occupation, interests, or location. This makes categorized user data is very valuable, because it can be used for targeted advertisements, social media friend recommendations, or suggestions during user searches. Finally, categorized data can also be used to highlight statistical relationships between factors that people have in common. This gives insight into cause-and-effect relationships, which can be used to solve real-world problems.

Optimization The code has been optimized to minimize the number of operations performed and memory consumed.The program took approximately 1:49h to run on a 6-Core Intel i7-9750H processor with 32Gb RAM.The total number of iterations was 350K. The graph was stored as a 357kB .CSV file.

References

[1] Vincent Blondel. *The Louvain method for community detection in large networks.* URL: `https://perso.uclouvain.be/vincent.blondel/research/louvain.html`. (accessed: 18.09.2020).

[2] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World.* Cambridge University Press, 2010.

[3] Alexey Elkin. *Hierarchical clustering by deleting edges of high betweenness and dendrogram with Python.* URL: `https://github.com/elkinal/Social_media_analysis`. (accessed: 17.09.2020).

[4] Aric Hagberg; Pieter Swart; Dan Schult. *NetworkX.* URL: `https://networkx.github.io/documentation/stable/index.html`. (accessed: 01.08.2020).