Spring 2020 - Social Media Analytics
Community Detection in an Email network

7. April 2020

1 Dataset

Given data set: **https://snap.stanford.edu/data/email-Eu-core.html**

The network was generated using email data from a large European research institution. There is an edge **(u,v)** in the network if person **u** sent person **v** at least on email [see description on website]. Therefore, the graph representing the network is a *directed* graph.

However, since we had the definitions of the algorithms (CPM for example) only for *undirected* graphs and with consultation of our teaching assistant, we handle these questions as if our graph was *undirected*. The task about the Pagerank centrality remains with the directed graph.

1.1 Properties

With the use of the `info(G,n)` function of the networkx library one can extract the following information about the given data set (as an undirected and directed graph):

	# nodes	# edges	avg in degree	avg out degree	avg degree
undirected:	1’005	16’706	25.4438	25.4438	-
directed:	1’005	25’571	-	-	33.2458

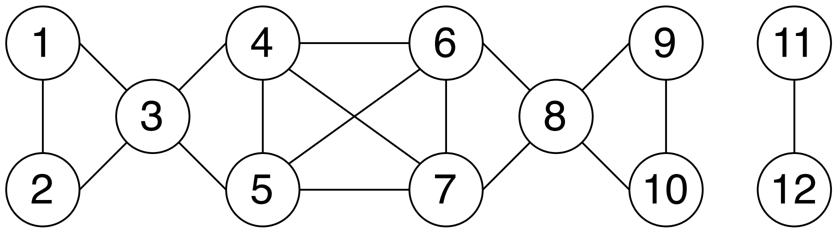
2 Implementation

2.1 Clique Percolation Method (CPM)

Our implementation of the Clique Percolation Method can be seen in the `CPM_Algorithm.py` file with the function `cpm_algorithm_selfmade()`.

When tested on an example of a toy-sized graph and analyzing with the debugger one can see that the algorithm performs as it should. Unfortunately the algorithm lacks in efficiency and therefore we weren't able to run it with the given Email data set or a subgraph of it. We let it run for 8 hours and still didn't have a result.

Our implementation was tested on the toy-sized graph which is illustrated in the following image and its corresponding edge list is saved in *toysize.txt*.



Following is the output for *k =3* of our implemented CPM algorithm on this example:

```
k = 3
There are 3 communities in the graph:
{3, 4, 5, 6, 7, 8}
{8, 9, 10}
{1, 2, 3}
Used time was:  0.0005359649658203125 sec
```

2.2 Vertex Similarity Method

Vertex similarity is a method which computes the number of neighbours (the common nodes) that two nodes **i** and **j** have in common.
First we need to find the similarity between each two nodes by calculating the variable: `common_neighbours = len(set(G[node_i]) & set(G[node_j]))`
The next code takes the value of the similarities for each pair from the previous code and assign it as a new similarity:
`"new_sim = calculate_vertex_similarity(G, i, j)"`
Then it compares each pair's similarity and returning the pair the highest number of common nodes.

```
if max_sim < new_sim:
max_pair = (i, j)
max_sim = new_sim
```

It repeats the same process till it find the pair with the highest similarity and returns this pair, followed by the number of common nodes.

This algorithm can be found in `Vertex_Sim_Pagerank_Cen.py`

3 Analysis

3.1 Identifying users' communities with CPM

Since our implementation didn't finish on the data set and even with the build-in function of the CPM algorithm from *networkx* the program didn't finish within 7 hours, we chose to reduce our data set for the analysis part to a subgraph of it.
In order to create such a subgraph we used the `Graph.subgraph(nodes)` function of *networkx*. For choosing the nodes we had the following thoughts:

- We took 335 nodes to create the subgraph (a third of the original amount). We tested various sizes and took this one since its computation time is reasonable and the resulting average degree of the subgraph is almost the same as in the original graph.
- We renounced taking random nodes. We didn't want our program to be based on random effects.
- We assumed that the order of the nodes in the data set doesn't come from a rank of importance. They all have the same importance.

This is why we chose simply the first 335 nodes of the data set to be included in the created subgraph. Following are values for this generated (undirected) subgraph:

	# nodes	# edges	avg degree
Email subgraph	335	6'039	36.0537

For choosing the optimal k value we consider that we don't want a giant component in our large network. Small k will often result in the emergence of a giant component to which most of the nodes belong. Bigger k tend to lead to a large number of smaller communities. We are aiming for a solution in between and so we tested for $k \in \{1, \dots, 11, 15, 20\}$. We counted the amount of received communities and got:

CPM parameter k:	3	4	5	6	7	8	**9**	**10**	**11**	15	20
# communities:	1	1	2	5	7	7	9	9	6	1	0

We think the most appropriate one is the one from *k=11* since the amount of communities is not the highest one but also not the lowest one. For the visualization we chose to precede with the values *k=9, 10, 11* for comparison reasons.

3.2 Identifying the top k users with Pagerank centrality

In PageRank the central node is the one which is highly linked with the others or linked from other important and link parsimonious nodes. It computes a ranking of the nodes in the graph G based on the structure of the incoming links. Since matrices can perform multiple calculations at the same time we chose this structure for implementation rather than a for loop. The code works as follows:

1. We create two dictionaries where we will save the values of the first iterations for each page and then the new iteration (`pr_t_plus_1`)

2. We then compute the transverse matrix of the graph: `A = nx.to_numpy_matrix(G)`
3. We then compute the stochastic matrix, which basically is the matrix which calculates the weight of each node in the first iteration. It is calculated by (1/degree of the node)
4. To find the pagerank we need to multiply the stochastic matrix by each value of the previous iteration. As there will be pages that are plain descriptive and contain no outgoing links we need to take a constant alpha (alpha=0.85) in order to fix the problem of pages which are not linked to other ones. So the final formula to calculate our pagerank will be:
`mat_pr_plus_1 = alpha * np.matmul(mat_mult, mat_pr) + ones * (1 - alpha) / float(n)`
5. The result we get at the end will give the pagerank value for each node.
`result = dict(zip(G.nodes, pr_List))`

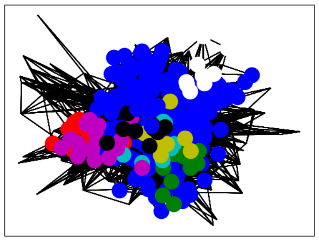
In order to find the best k, we tried different values and for $k \in \{7, 8, ..., 20\}$. The results of the pair with the best vertex similarity was {160,107}, with **154** common nodes. However, if we try with $k = 20+$, we obtain the pair {121,82} with 170 nodes in common, which leads us thinking that the best k for our case is $k = 21$.
`sorted(GetPageRankMatrix(G).items(), key=operator.itemgetter(1), reverse=True)[:21]`

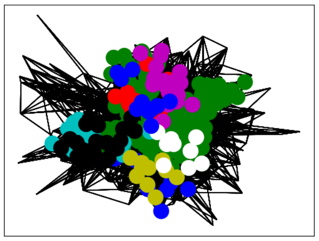
Using Vertex Similarity Algorithm and PageRank Algorithm with a k=21, we achieved the following results: `Top Vertex Pair:  ((121, 82), 170)`
The best PageRank values vary from 0.00816087676478221 to 0.00284138512870143 for 21st top nodes.

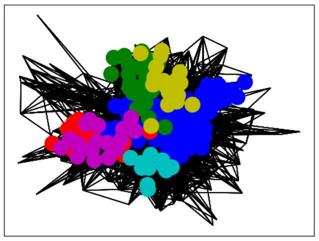
4 Visualization

4.1 Visualized output of CPM method

For the visualization of this algorithm we chose three values of k: *9, 10, 11.* Thus, to see the differences between them. The reason why we picked k = *9* and k = *10* although they have the same amount of communities is that we wanted to see if the size of the communities may change. The communities are colored in the graph and can be seen in the following images:


k = 9


k = 10

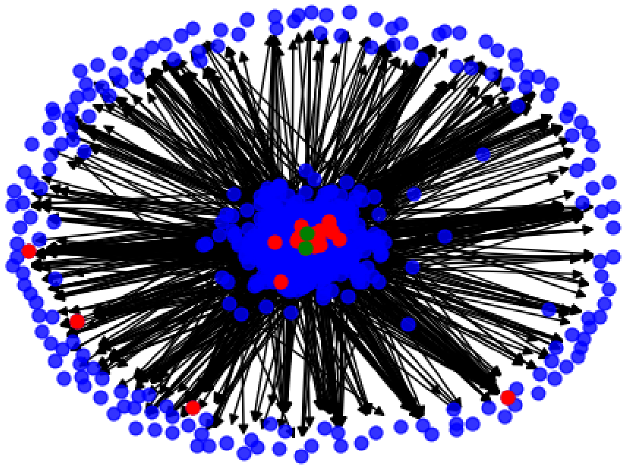

k = 11

Looking at those images we see a difference between *k=9* and *k=10*: The biggest community in the left one is bigger than the one in the middle. This is an interesting effect, that even with the same amount of communities, their size may vary regarding the choice of k.
The result of our chosen "most appropriate" k is visualized in the image on the right and we think by comparing this one with the others we conclude that this one is indeed a better choice since there is no too big component.

4.2 Visualized output of the top k users

1. Top 21 nodes with the highest PageRank are highlighted in **red**
2. Top vertex pair are highlighted in **green**
3. All nodes are colored in **blue**

We notice that the pair {121,82}, which is common for both sets, will remain colored in green.



5 Conclusion

Looking back at what we were able to do and where we have been refused to go further because of the limits, we conclude that the results would be more meaningful if we could have worked with a cluster/more computation power. In such a case we would have enough capacity to run the algorithms because our own equipment consisted of only two laptops which overheated all the time, refusing us to continue.

For future work we would run the computations on a cluster, apply the CPM algorithm on the whole data set and then probably vary with more k values since the outcome may be different with the whole amount of nodes and edges.