Supplementary Materials for FARZ Benchmarks

Justin Fagnan, Afra Abnar, Reihaneh Rabbany, and Osmar R. Zaïane
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada
{rabbanyk,aabnar,fagnan,zaiane}@ualberta.ca

ABSTRACT

This document presents the supplementary materials for the FARZ benchmark paper. It includes detailed algorithms for the approaches presented in the paper, as well as extended results from our experiments.

1. GENERALIZED 3-PASS BENCHMARK

1.1 Phase 2: Assign

Algorithm 1 summarizes the assignment of nodes to communities, which has two main phases,
1) Determine the community sizes: By sampling from the community size distribution. Sample communities until the sum of all community sizes equals the total number of nodes.
2) Assign nodes to the communities: First select a random node in the network and add it, if its degree is less than the size of the community. Any vertex that is too big for the current community gets put aside. At the end of the community assignment, any nodes that left over are randomly added to a community that is big enough to contain their degree. If that community is already full, then it ejects a smaller-degree node and adds in the left over node. The ejected node then gets added into the leftover node queue. The process continues until there is no left over nodes.

ALGORITHM 1: Assign(G, θ^C)

1: $\{s_1, s_2 \dots s_m\}$ based on θ^C and G *// determine capacity of communities*
2: $\{c_1 = \emptyset, c_2 = \emptyset \dots c_m = \emptyset\}$ *// initialize the communities*
3: $H \leftarrow \emptyset$ *// initialize the homeless queue*
4: **for** $v \in G$ **do** *// assign nodes to communities*
5: randomly choose c_i from C where $s_i > |c_i|$
6: **if** $s_i \geq (1-\mu) \times G.degree(v)$ **then** $c_i.add(v)$
7: **else** $H.add(v)$
8: **while** H is not empty **do** *// assign homeless nodes*
9: randomly choose v from H
10: $i \leftarrow rand(1, m)$ *// pick a community at random*
11: **if** $s_i \geq (1-\mu) \times G.degree(v)$ **then**
12: $c_i.add(v)$
13: $H.remove(v)$
14: **if** $|c_i| > s_i$ **then** *// kick out a random node*
15: randomly choose u form c_i
16: $c_i.remove(u)$
17: $H.add(u)$

1.1.1 CN Modification

For the CN assignment, line 5 of the Algorithm 1 is changed so that the community c_i is chosen from C, for node v, with probabilities proportional to the current neighbours of v in c_i; see Algorithm 2 for the details.

ALGORITHM 2: CN Assign

4: ...
5: $P \leftarrow 1\ \forall c_i \in C$
5: **for** $i \in G.neigh(v)$ **do**
5: $p_{c_i} \leftarrow p_{c_i} + 1$
5: $P \leftarrow P/sum(P)$
5: randomly choose c_i from C where $s_i > |c_i|$, and probabilities are proportional to P
6: ...

1.1.2 NE Modification

For NE assignment, we change line 6-7 of the Algorithm 1 so that after community c_i is chosen for node v, we expand from node v using a BFS(Breadth-first search), to also add its neighbours to c_i, until c_i is full or the expansion is ended; see Algorithm 3 for details.

Note that this way of assigning nodes to communities results in far fewer re-wirings because all of the internal edges are already there. Fewer re-wirings yields more realistic network structure.

ALGORITHM 3: NE Assign

5: ...
6: $M \leftarrow [0] \times G.n$
7: $Q \leftarrow v$
7: $M_v \leftarrow 1$
7: **while** $len(Q) > 0$ and $c_i < s_i$ **do**
7: $v \leftarrow Q.pop$
7: $c_i.add(v)$
7: **if** $s_i \geq (1-\mu) \times G.degree(v)$ **then**
7: $Q.add(v)$
7: **for** $u \in G.neigh(v)$ **do**
7: **if** $M_u == 0$ **then**
7: $Q.add(u)$
7: $M_u \leftarrow 1$
8: ...

1.2 Phase 3: Overlay-Rewire

Algorithm 4 summarizes overlay of communities on the network through rewirings. This procedure is not described in details in the authors' original paper [3], and the available code for it reflects the modified LFR version[2]; hence the procedure presented here, is our best guess of the original method.

1) Determine rewirings: Each vertex should have μ% edges leading out of the community (between), and $(1-\mu)$% edges leading to other nodes in the community (within). So we start by computing how many edges need to rewired for each vertex. For example, Vertex v might need 2 external edges swapped to internal edges; thus has a desired internal change of $\delta^w = +2$, and a desired external change of $\delta^b = -2$.

2) Rewire edges within communities: After computing all the desired changes, iterate through all of the vertices. For each vertex, while it requires more internal edges (i.e. internal change > 0), we loop through all other vertices that require more internal edges. If there is another vertex in the same community and it requires more internal edges, then we pair them up and add an edge between them (i.e. internal change decreases by one for both). Moreover, we remove the excess internal edges, with a similar process; see line 14-21 of procedure 4 for more details.

3) Rewire edges between communities: The same process occurs for external change. In this case we look for other vertices that need an external edge and are out not in the current community of the target vertex. We add an external edge between these two vertices.

1.3 LFR Generator

Here we describe LFR in terms of this generalization. The network model used in the LFR benchmark is the CF model, described in details in the related work section. More specifically, for the LFR, $\theta^G = \{N, k_{avg}, k_{max}, \gamma\}$, which are respectively: number of nodes in the graph, average G.degree, maximum G.degree and exponent of power law G.degree distribution. These parameters are basically used to determine the G.degree sequence of G; from which the graph is then synthesized using the CF model.

On the other hand, the parameters for communities are $\theta^C = \{\mu, \beta, c_{min}, c_{max}\}$, which are respectively: mixing parameter, exponent of power law distribution for community sizes, minimum size and maximum size for communities. The latter three are used to determine the capacity of communities; whereas the mixing parameter μ controls the difficulty of problem, which is used in the rewiring phase.

Key issues of LFR benchmarks are discussed in the paper. From a technical point of view, the implementation of these benchmarks is heuristic and complex, which rules out modifications as well as analytical analysis. When in fact this sampling process of structured networks, does not need to be complicated. An example of unnecessary effort in the original LFR implementation is many rewirings in order to strictly stick to the exact degree distribution of the nodes, when this degree distribution is itself generated/sampled randomly in the first place. In our implementation of the LFR, we relaxed these restrictions to reach a simpler approach, which is as effective as the original one if not better, since it reduces the amount of rewirings.

ALGORITHM 4: Overlay(G, C, θ^C)

1: $\mu \leftarrow \theta^C$
2: **for** $v \in G$ **do** *// determine rewirings per node*
3: $\quad \delta^b[v] \leftarrow \lfloor \mu \times deg(v) - deg^b(v, c) \rfloor$
4: $\quad \delta^w[v] \leftarrow -\delta^b[v]$ *// desired within changes*
5: **for** $c \in C$ **do** *// rewire edges within communities*
6: $\quad I \leftarrow \{v \mid v \in c \wedge \delta^w[v] > 0\}$ *// add internal edges*
7: $\quad$ **while** $|I| \geq 2$ **do**
8: $\quad\quad$ randomly choose $v \neq u$ from I
9: $\quad\quad$ add $edge(u, v)$ to G
10: $\quad\quad \delta^w[v] \leftarrow \delta^w[v] - 1$
11: $\quad\quad \delta^w[u] \leftarrow \delta^w[u] - 1$
12: $\quad\quad$ **if** $\delta^w[v] = 0$ **then** $I.remove(v)$
13: $\quad\quad$ **if** $\delta^w[u] = 0$ **then** $I.remove(u)$
14: $\quad$ **for** $\{v \mid v \in c \wedge \delta^w[v] < 0\}$ **do** *// remove excess edges*
15: $\quad\quad I \leftarrow \{u \mid edge(u, v) \in G \wedge u \in c \wedge \delta^w[u] < 0\}$
16: $\quad\quad$ **while** $|I| \geq 1 \wedge \delta^w[v] < 0$ **do**
17: $\quad\quad\quad$ randomly choose u from I
18: $\quad\quad\quad$ remove $edge(u, v)$ from G
19: $\quad\quad\quad \delta^w[v] \leftarrow \delta^w[v] + 1$
20: $\quad\quad\quad \delta^w[u] \leftarrow \delta^w[u] + 1$
21: $\quad\quad\quad I.remove(u)$
22: **for** $c \in C$ **do** *// rewire edges between communities*
23: $\quad I \leftarrow \{v \mid v \in c \wedge \delta^b[v] > 0\}$ *// add between edges*
24: $\quad O \leftarrow \{v \mid v \notin c \wedge \delta^b[v] > 0\}$
25: $\quad$ **while** $|I| \geq 1 \wedge |O| \geq 1$ **do**
26: $\quad\quad$ randomly choose v from I
27: $\quad\quad$ randomly choose u from O
28: $\quad\quad$ add $edge(u, v)$ to G
29: $\quad\quad \delta^b[v] \leftarrow \delta^b[v] - 1$
30: $\quad\quad \delta^b[u] \leftarrow \delta^b[u] - 1$
31: $\quad\quad$ **if** $\delta^b[v] = 0$ **then** $I.remove(v)$
32: $\quad\quad$ **if** $\delta^b[u] = 0$ **then** $O.remove(u)$
33: $\quad$ **for** $\{v \mid v \in c \wedge \delta^b[v] < 0\}$ **do** *// remove excess edges*
34: $\quad\quad O \leftarrow \{u \mid edge(u, v) \in G \wedge u \notin c \wedge \delta^b[u] < 0\}$
35: $\quad\quad$ **while** $|O| \geq 1 \wedge \delta^b[v] < 0$ **do**
36: $\quad\quad\quad$ randomly choose u from O
37: $\quad\quad\quad$ remove $edge(u, v)$ from G
38: $\quad\quad\quad \delta^b[v] \leftarrow \delta^b[v] + 1$
39: $\quad\quad\quad \delta^b[u] \leftarrow \delta^b[u] + 1$
40: $\quad\quad\quad O.remove(u)$

2. FARZ

The procedures for FARZ generator is described in the main paper. The assign algorithm is left out due to its triviality which is described here.

2.1 FARZ Assign

ALGORITHM 5: FARZ Assign (i, C)

1: **for** $[1 \dots r]$ **do**
2: $\quad cid \leftarrow select(\{1 \dots k\}, p_u = \frac{|u|+\phi}{\sum_v (|v|+\phi)})$
3: $\quad$ **if** $i \notin C[cid]$ **then** $C[cid] \leftarrow i$

3. EXTENDED RESULTS

Here we report extended results for the experiments in the original paper, i.e. distribution plots for all the properties of networks generated by the generalized 3-pass benchmark (in Section 3.1), and rankings of algorithms in a wider set of settings for the FARZ (in Section 3.2).

3.1 Generalized 3-Pass Benchmark

Complete plots for the properties of the synthesized networks from the different variations derived form our generalized 3-pass model, described in the paper. The probability density estimation for parameters, as they change by μ. In each plot, rows are different models: CF(top), BA(middle), FF(last). Initial network is marked by blue. This corresponds to Figure 3 in the paper.

Figure 1: Original LFR Assignment

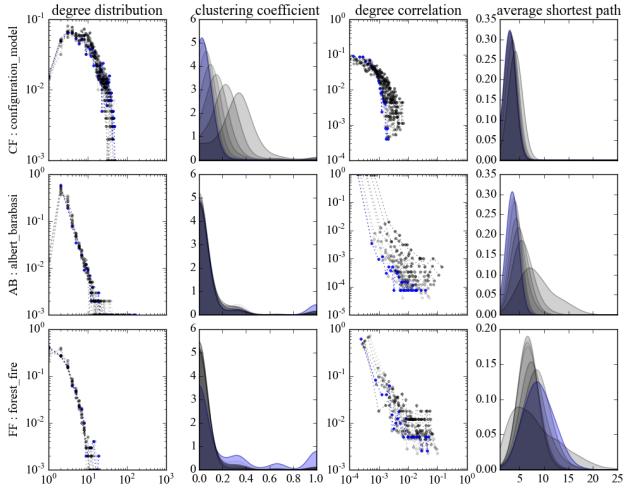


Figure 2: CN Assignment

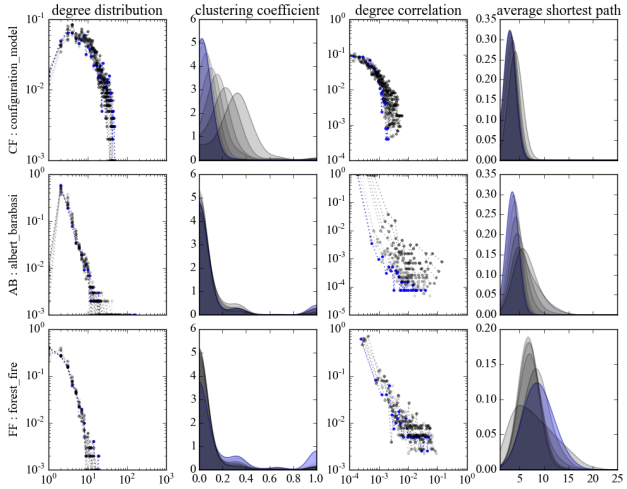
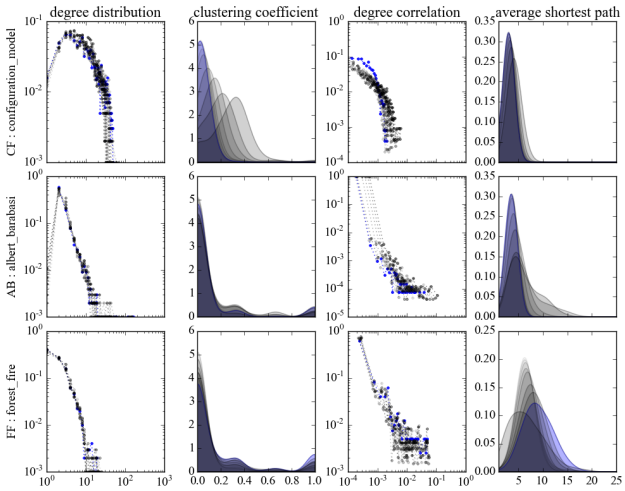


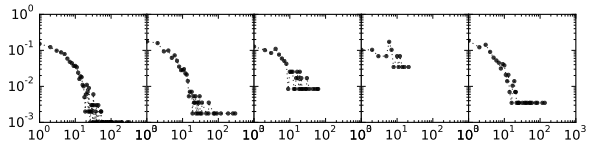
Figure 3: NE Assignment



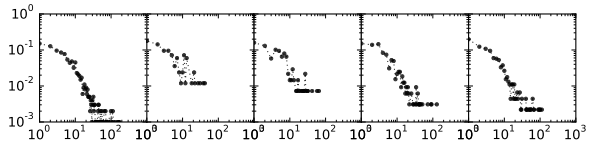
3.2 FARZ Extended Results

Figure 4: Degree distributions per community for synthetic networks generated by FARZ for 4 different settings of Figure 7. The first plot reports the degree distribution for the overall network, and the subsequent subplots show the degree distribution per community.

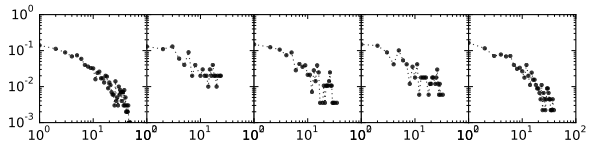
(a) $\alpha = 0.2,\ \gamma = -0.8,\ m = 5,\ k = 4$



(b) $\alpha = 0.5,\ \gamma = -0.5,\ m = 5,\ k = 4$



(c) $\alpha = 0.5,\ \gamma = 0.5,\ m = 5,\ k = 4$



(d) $\alpha = 0.8,\ \gamma = -0.2,\ m = 5,\ k = 4$

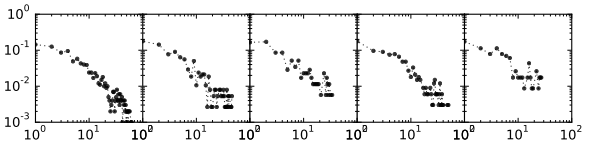


Figure 5: Degree distributions per community for synthetic networks generated by LFR of Figure 6 in the paper.

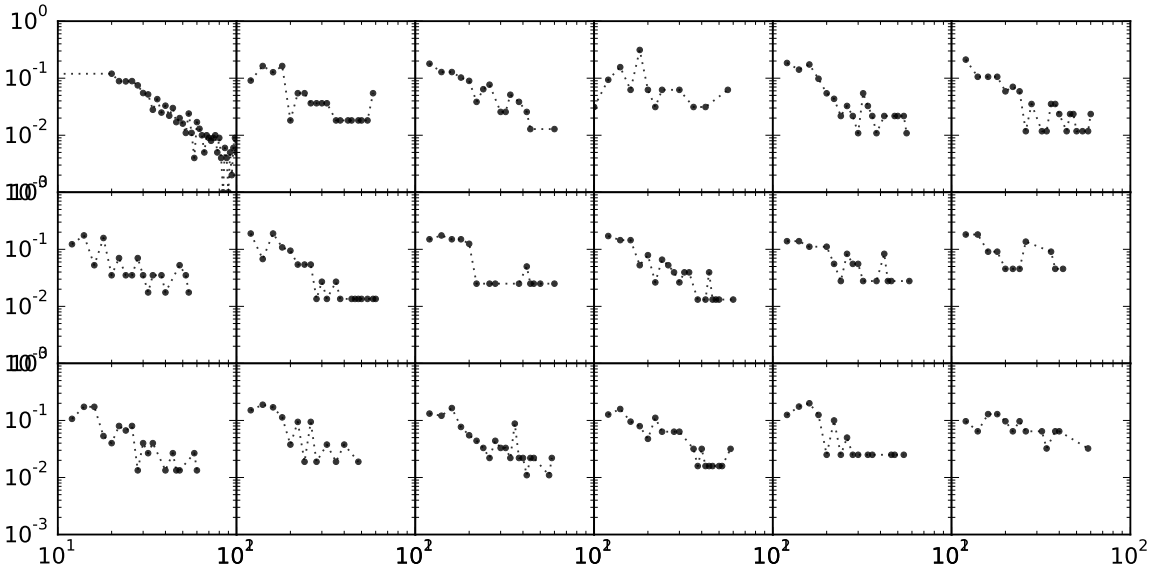
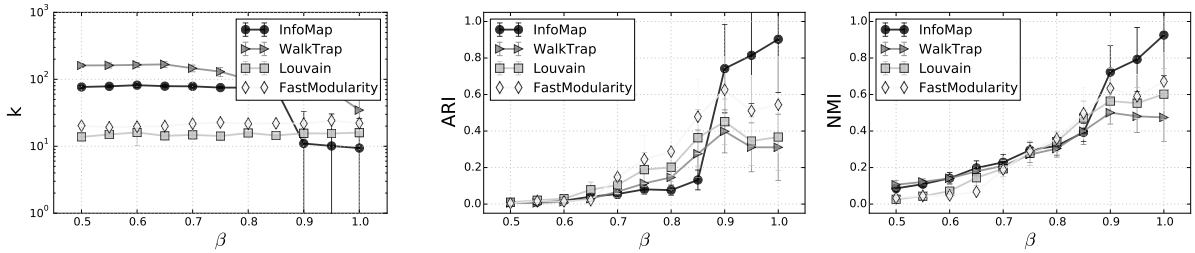
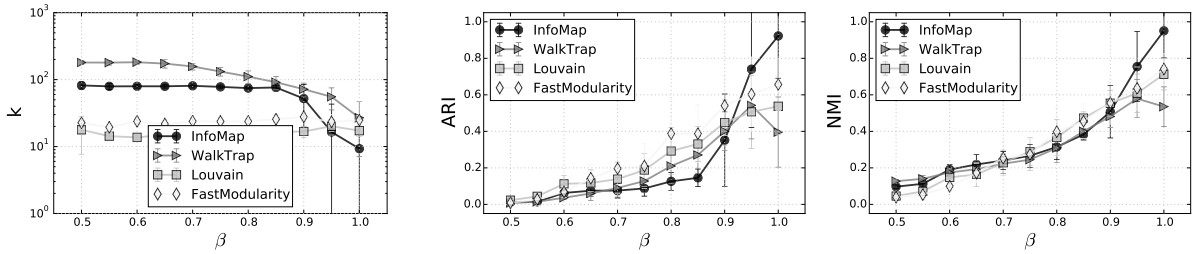


Figure 6: Comparing performance of community mining algorithms on benchmarks with **positive and negative degree correlation**, all the four settings. Also reporting the number of clusters found by each method.

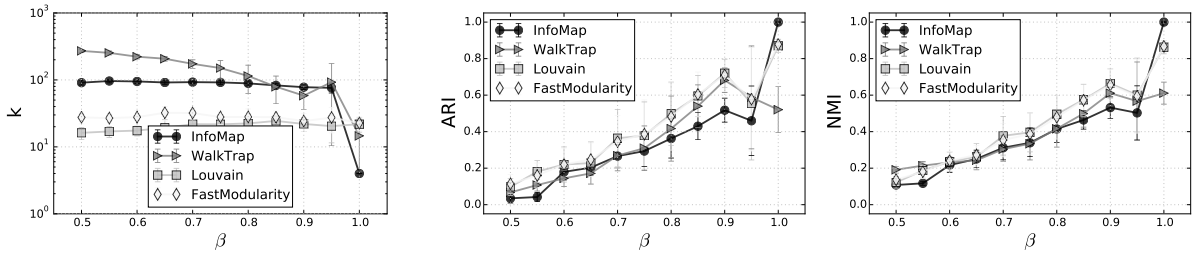
(a) $\alpha = 0.5,\ \gamma = 0.5,\ m = 5,\ k = 4$



(b) $\alpha = 0.8,\ \gamma = 0.2,\ m = 5,\ k = 4$



(c) $\alpha = 0.2,\ \gamma = -0.8,\ m = 5,\ k = 4$



(d) $\alpha = 0.5,\ \gamma = -0.5,\ m = 5,\ k = 4$

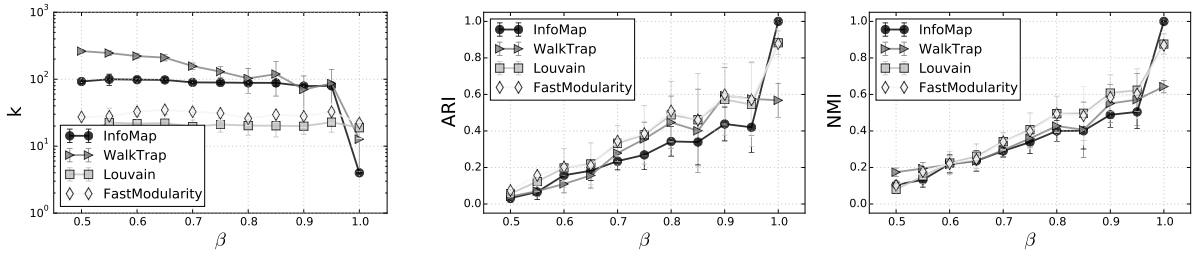


Figure 7: Same algorithms compared on LFR, setting is the 1000B used in [1], i.e. -N 1000 -k 20 -maxk 50 -t1 2 -t2 1 -minc 20 -maxc 100.

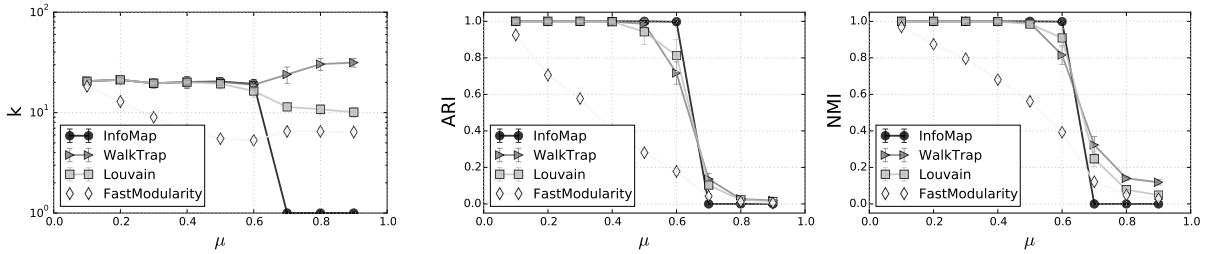
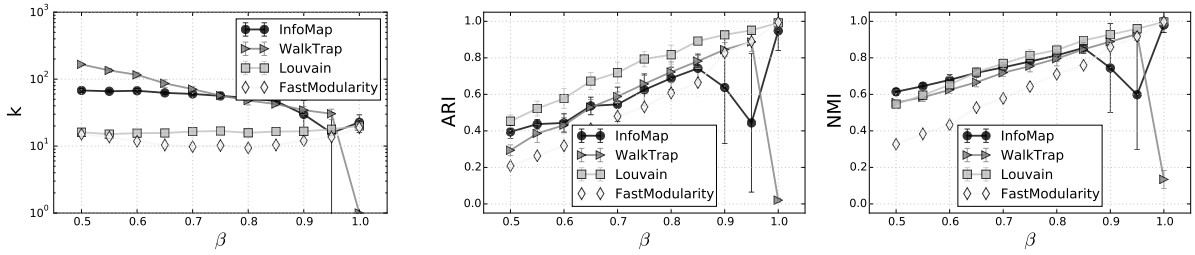
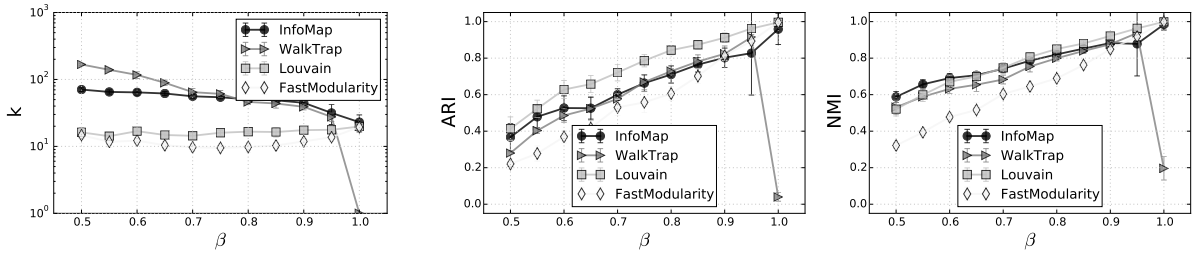


Figure 8: Comparing performance of community mining algorithms similar to Figure 6 but on **denser benchmarks ($m = 6$) with more communities ($k = 20$)**. In this setting Louvain clearly performs the best in particular in networks with positive degree correlation. The drop in $\beta = 1$ is due to the communities not linked together which makes the network disconnected and causes problem for the WalkTrap algorithm. The other random walk based method, InfoMap, also seems to have difficulty when communities are well separated, i.e. when $\beta \in [.85, .95]$ and $\gamma > 0$.

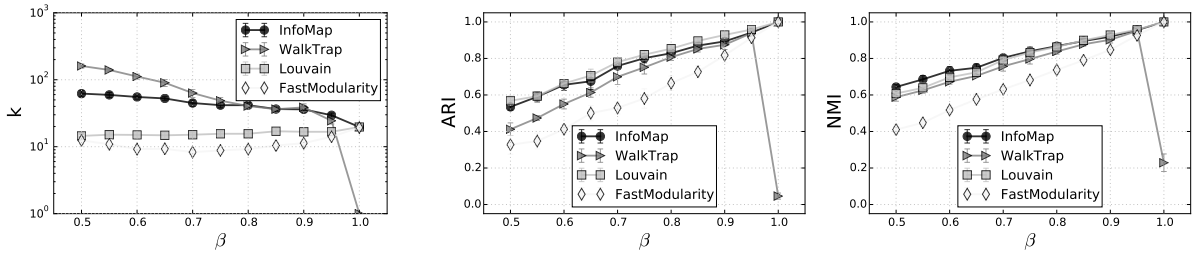
(a) $\alpha = 0.5,\ \gamma = 0.5,\ m = 7,\ k = 20$



(b) $\alpha = 0.8,\ \gamma = 0.2,\ m = 7,\ k = 20$



(c) $\alpha = 0.2,\ \gamma = -0.8,\ m = 7,\ k = 20$



(d) $\alpha = 0.5,\ \gamma = -0.5,\ m = 7,\ k = 20$

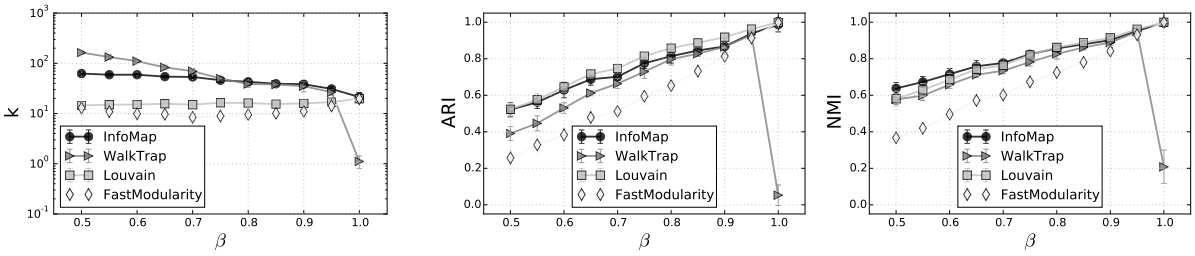
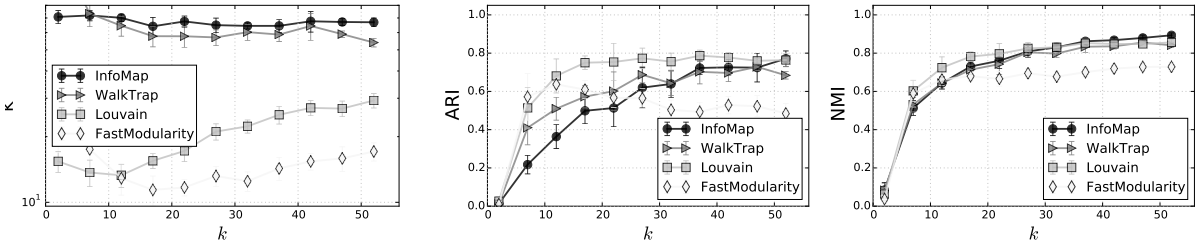
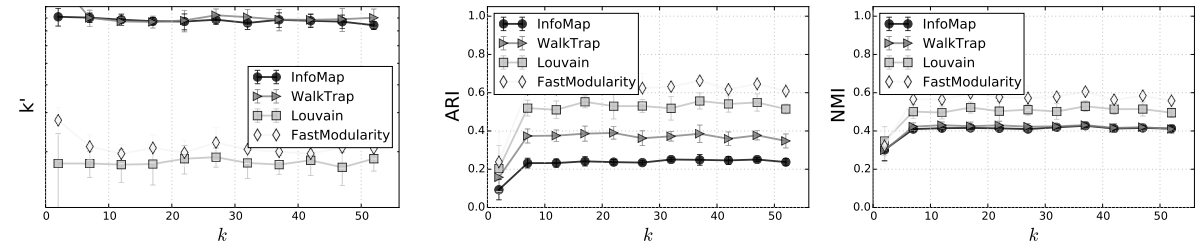


Figure 9: Performance of community mining algorithms on benchmarks with different **number of built-in communities**, for all the four settings. Also reporting the number of clusters found by each method.

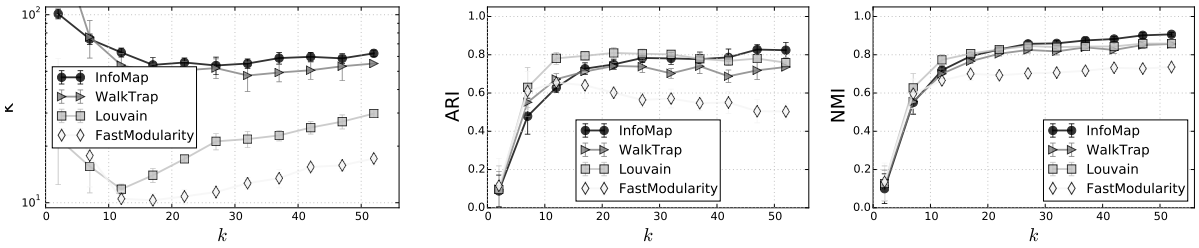
(a) $\alpha = 0.5,\ \gamma = 0.5,\ m = 5,\ k = 4$



(b) $\alpha = 0.8,\ \gamma = 0.2,\ m = 5,\ k = 4$



(c) $\alpha = 0.2,\ \gamma = -0.8,\ m = 5,\ k = 4$



(d) $\alpha = 0.5,\ \gamma = -0.5,\ m = 5,\ k = 4$

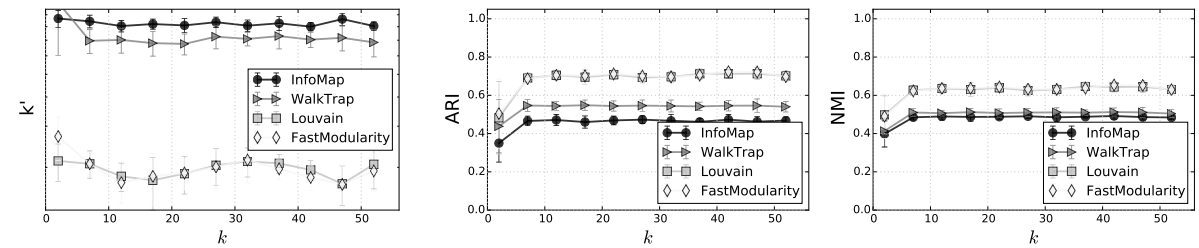
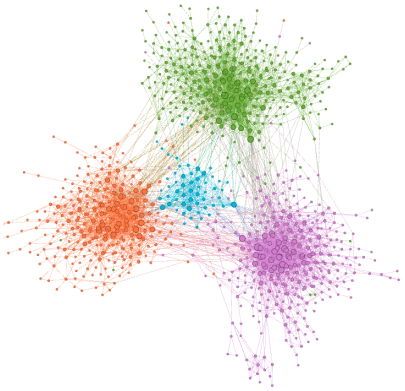


Figure 10: Example of actual graphs generated by FARZ, used in the previous plots, $\alpha = 0.5$, $\gamma = 0.5$, $m = 5$, $k = 4$. Plots visualized with Gephi toolbox using ForceAtlas2 layout, where node sizes corresponds to the degree of the nodes, and the colours of nodes to their assigned communities.

(a) $\beta = 1$

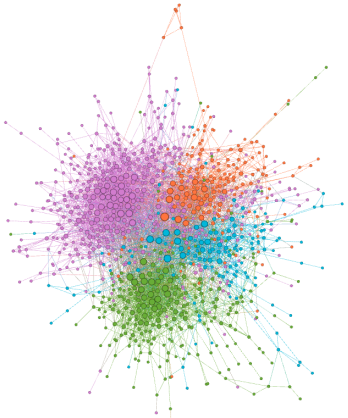
(b) $\beta = 0.95$



(c) $\beta = 0.9$

(d) $\beta = 0.85$

(e) $\beta = 0.8$



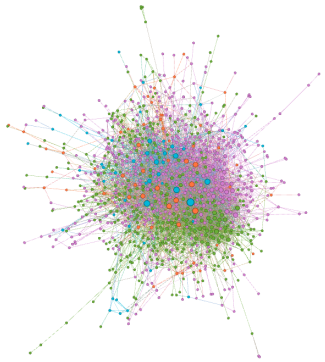
(f) $\beta = 0.75$

(g) $\beta = 0.7$

(h) $\beta = 0.65$

(i) $\beta = 0.6$

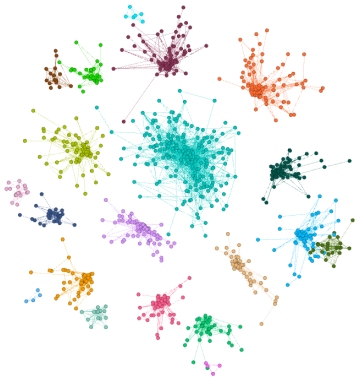
(j) $\beta = 0.55$



(k) $\beta = 0.5$

Figure 11: Example of actual graphs generated by FARZ, used in the previous plots, $\alpha = 0.5,\ \gamma = 0.5,\ m = 7,\ k = 20$.

(a) $\beta = 1$



(b) $\beta = 0.95$

(c) $\beta = 0.9$

(d) $\beta = 0.85$

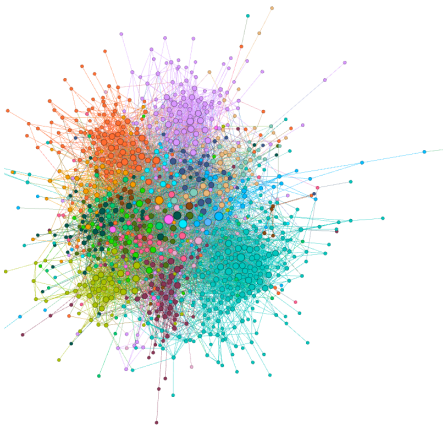
(e) $\beta = 0.8$

(f) $\beta = 0.75$

(g) $\beta = 0.7$

(h) $\beta = 0.65$

(i) $\beta = 0.6$

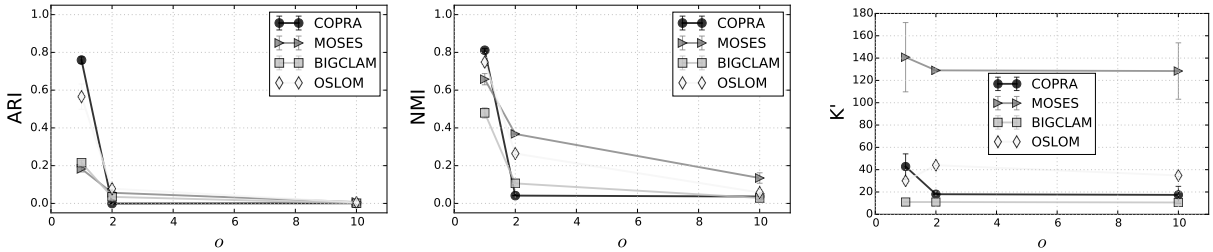


(j) $\beta = 0.55$

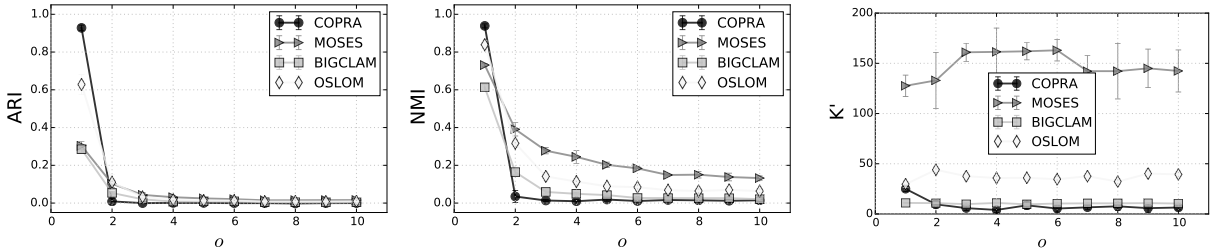
(k) $\beta = 0.5$

Figure 12: Comparing performance of community mining algorithms on benchmarks with **overlapping communities**, plotted as a function of the number of communities each node can belong to. All methods perform poorly, for when nodes are all overlapping.

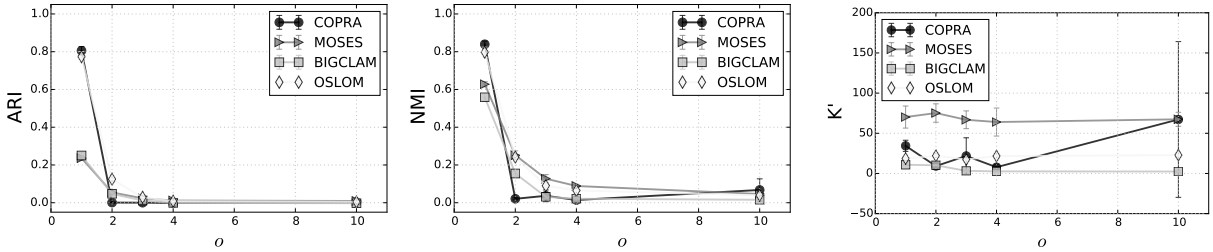
(a) $\alpha = 0.5,\ \gamma = 0.5,\ \beta = 0.8$



(b) $\alpha = 0.5,\ \gamma = 0.5,\ \beta = 0.9$



(c) $\alpha = 0.2,\ \gamma = -0.8,\ \beta = 0.8$



(d) $\alpha = 0.2,\ \gamma = -0.8,\ \beta = 0.9$

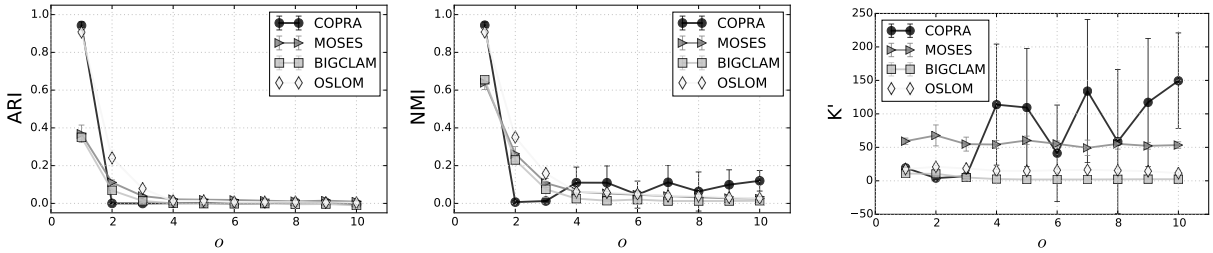


Figure 13: **Overlapping communities**, for setting (a) in Figure 11, where the number of communities that each node can belong to is fixed to 3, and the portion of overlapping nodes (q) is varied from 0.0 (no overlap), to 0.5 (half of the nodes are overlapping). Again all methods perform poorly, except COPRA, which is able to detect communities when the portion of overlapping nodes is small enough, i.e. $q < 0.2$.

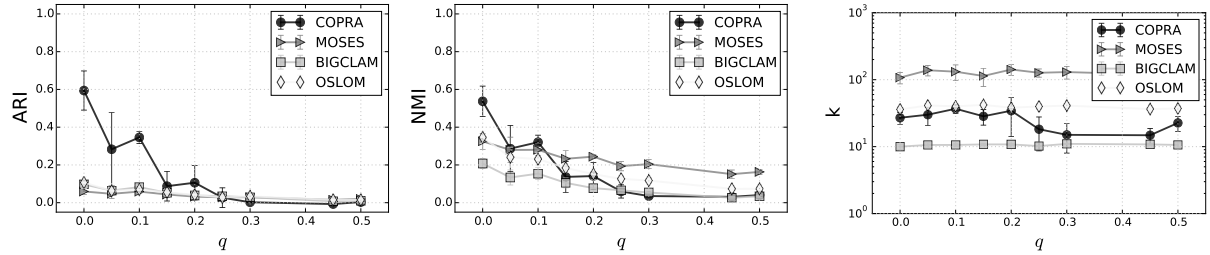
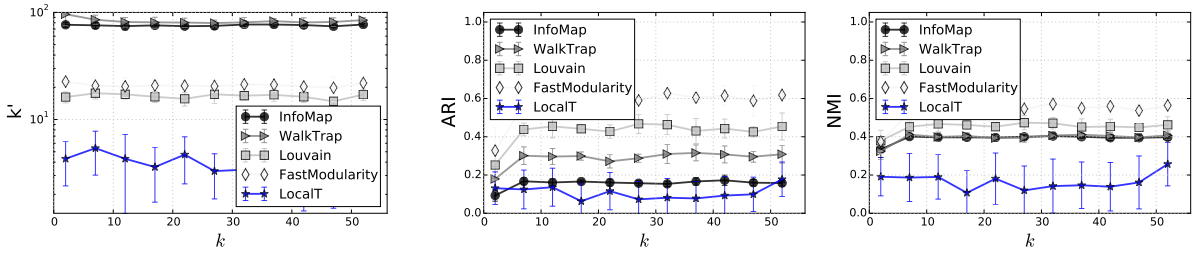
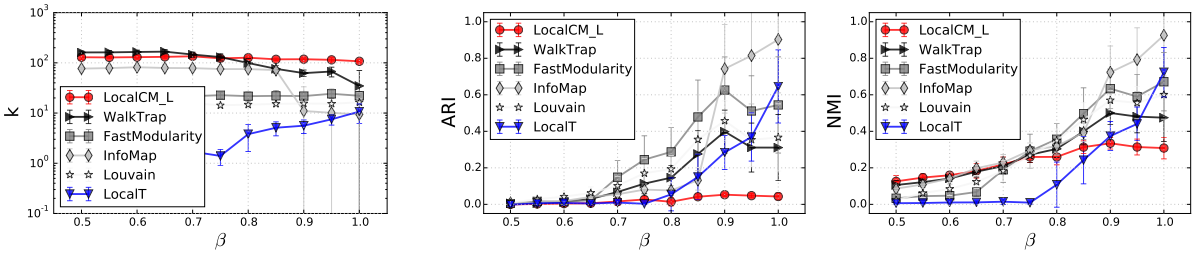


Figure 14: Comparing **LocalT**, **LocalCM**, and **TopLeaders** methods on FARZ benchmarks.

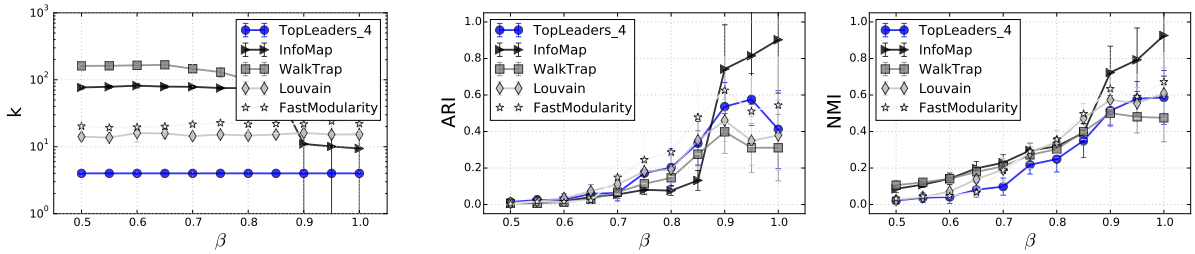
(a) $\alpha = 0.5,\ \gamma = 0.5,\ \beta = 0.8,\ m = 5$



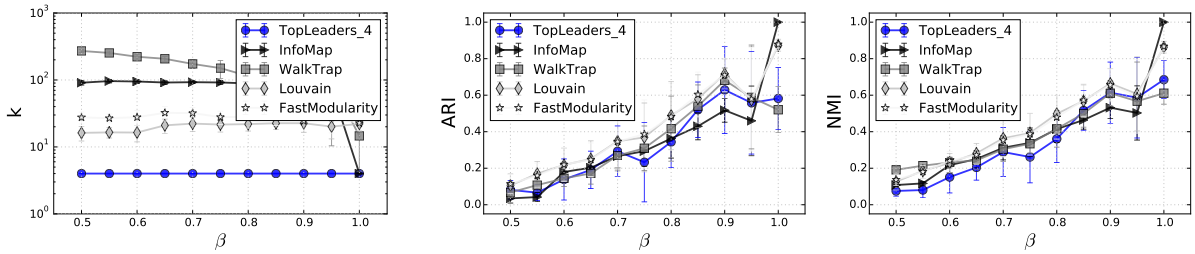
(b) $\alpha = 0.5,\ \gamma = 0.5,\ m = 5,\ k = 4$



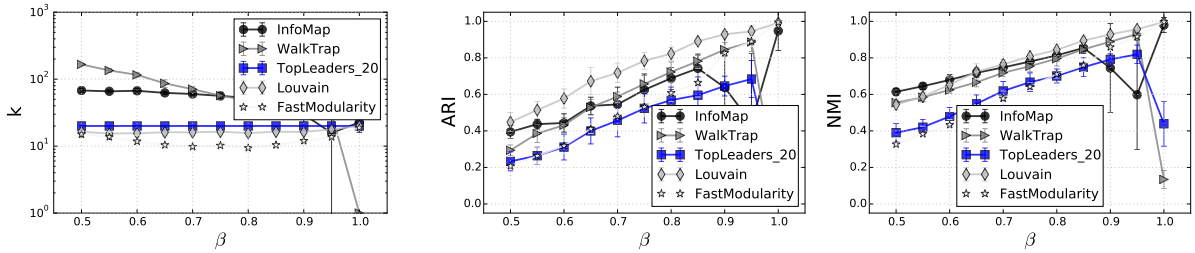
(c) $\alpha = 0.5,\ \gamma = 0.5,\ m = 5,\ k = 4$



(d) $\alpha = 0.2,\ \gamma = -0.8,\ m = 5,\ k = 4$



(e) $\alpha = 0.5,\ \gamma = 0.5,\ m = 7,\ k = 20$



Naming

FARZ, based on different transliteration, means sorting, division, or assess in Arabic; and assumption, or agile in Persian.

References

[1] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), 2009.

[2] A. Lancichinetti, S. Fortunato, and J. Kertesz. Detecting the overlapping and hierarchical community structure of complex networks. *New Journal of Physics*, 11(3):20, 2008.

[3] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4), 2008.