Community detection based on partial information

2020

Artur Myszkowski
University of Warsaw

Community detection algorithms and partial information - intuition behind

- Many algorithms solving community detection which use no external information except graph structure exist
- Could we perform better if some prior information about communities structure in the network given?
- Idea: Modify well known algorithms using provided seed communities and see what happens!
- Possible options:
  - [Trivial] Merge nodes of seed communities and run original algorithm
  - Consciously modify known algorithm and make it perform better

Merging nodes affects graph structure, thus we potentially lose valuable input for the community detection algorithm

Community detection algorithms and partial information - intuition behind

- What does it mean to perform better in regard with seed communities approach?
- Let consider 2 cases:
  - Given graph and a little information about communities structure (let denote them as seed communities)
  - Same as above but we presumably some links are missing and some are fake
- And 2 possible objectives:
  - Improve some measure of network division (e.g. modularity score)
  - Find division fitting better to reality (not necessarily maximize score of any measure globally)

Algorithms modification - assumptions

- Stress on non-trivial modifications
- Allow joining seed communities with the communities found during a run of the algorithm or other prior disjoint seed communities if it makes the division better

Network-Centrality

useful tool for analytics

Algorithms implementation - requirements

- **Many existing libraries, but...**
  - **SciPy, scikit-learn, NetworkX, cdlib (Python)**
  - **NetworKit (Python, C++)**
  - **igraph (C/C++, R, Python, Mathematica)**
  - **(…)**
- **Have to relate to original papers with “the more algorithms implemented the better”**
- **Written by one team/person (to considers algorithms comparable, e.g. to check running time, etc.)**
- **Have to work with JS (to embed within network-centrality)**
- **Individual JS libraries found, yet no single library satisfying *all* above points**

Algorithms implementation - solution

- igraph's C implementation chosen to modify on
- Compile igraph's code into WebAssembly with Emscripten, manually expose API, create npm package and run it using JS
- Pro - mature, tested and still maintained library used as core
- Pro - one source code, many applications:
  - Client-side computations ability in browser
  - Server-side computations ability using node.js
  - Embeddable in notebooks for additional analytics (e.g. such as observablehq.com or jupyter notebook with extensions)
- Con - harder to write and debug than using single language

Algorithms overview - selection

No.	**Algorithm name**	**Authors**	**igraph C**	**Paper[*]**	**Modification**
1.	EdgeBetweenness	*Girvan-Newman*	✓	✓	✓
2.	FastGreedy	*Clauset-Newman-Moore*	✓	✓	✓
3.	Infomap	*Rosvall-Bergstrom*	✓	✓	✓ (partially)
4.	LabelProp	*Raghavan-Albert-Kumara*	✓	✗	✓ (yes, but (almost) trivial)
5.	LeadingEigen	*Newman*	✓	✓	**?**
6.	Louvain	*Blondel-Guillaume-Lambiotte-Lefebvre*	✓	✓	✓
7.	Optimal	*Brandes-Delling-Gaertler-Gorke-Hoefer-Nikoloski-Wagner*	✓	✗	✗
8.	Spinglass	*Reichardt-Bornholdt*	✓	✓	**?**
9.	Walktrap	*Pons-Latapy*	✓	✓	**?**

✓ - yes; ✗ - no; ✓ - partially; ✓ - yes, but (almost) trivial

*Marcin Waniek, Tomasz Michalak, Talal Rahwan, Michael Wooldridge, Hiding Individuals and Communities in a Social Network
Nature Human Behaviour 2, 139-147 (2018)

Algorithms overview - selection

No.	Algorithm name	Authors	igraph C	Paper*	Modification
1.	**EdgeBetweenness**	***Girvan-Newman***	✓	✓	✓
2.	**FastGreedy**	***Clauset-Newman-Moore***	✓	✓	✓
3.	Infomap	*Rosvall-Bergstrom*	✓	✓	✔
4.	LabelProp	*Raghavan-Albert-Kumara*	✓	✗	☑
5.	LeadingEigen	*Newman*	✓	✓	?
6.	**Louvain**	***Blondel-Guillaume-Lambiotte-Lefebvre***	✓	✓	✓
7.	Optimal	*Brandes-Delling-Gaertler-Gorke-Hoefer-Nikoloski-Wagner*	✓	✗	✗
8.	Spinglass	*Reichardt-Bornholdt*	✓	✓	?
9.	Walktrap	*Pons-Latapy*	✓	✓	?

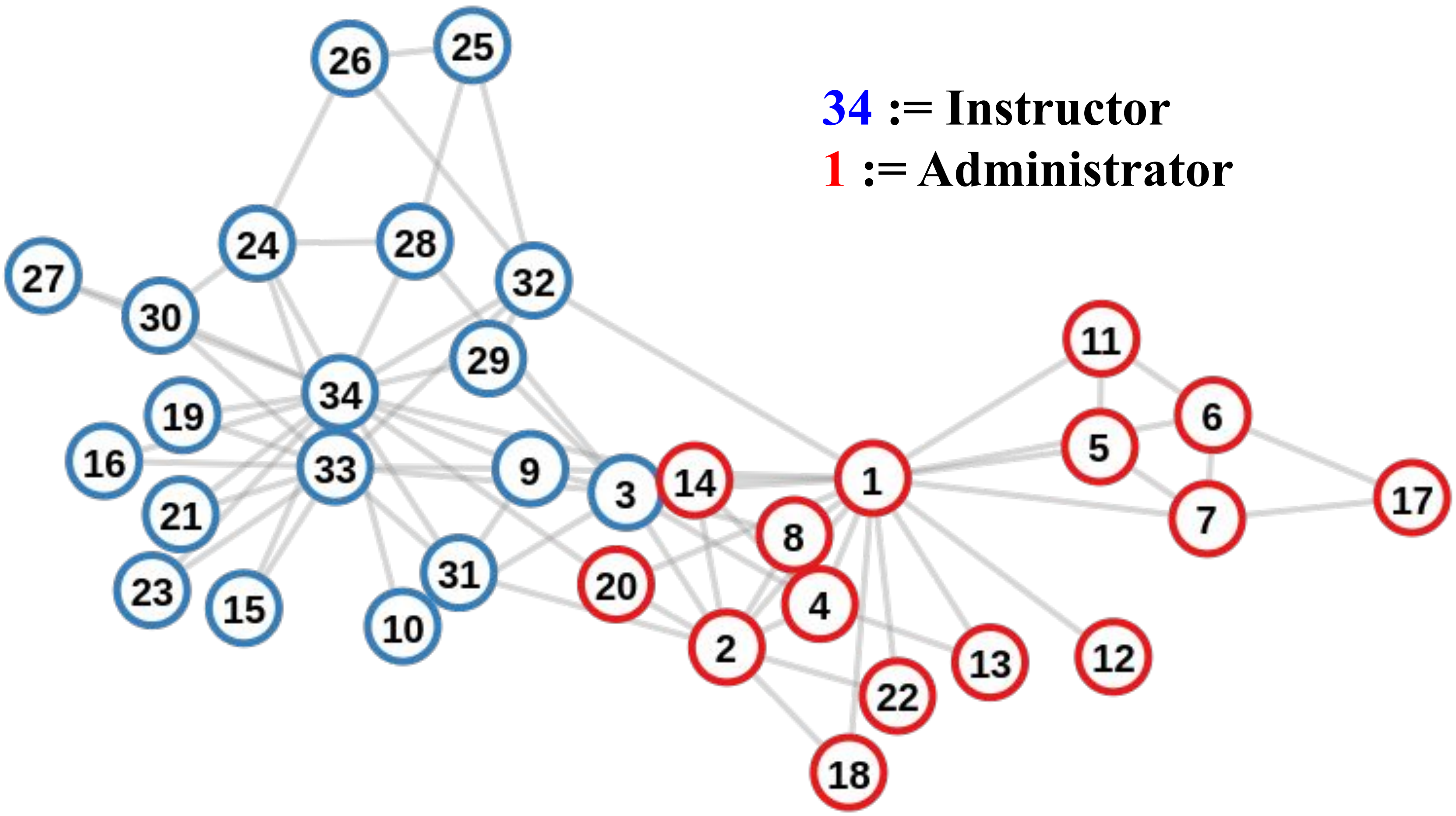
✓ **- yes;** ✗ **- no;** ✔ **- partially;** ☑ **- yes, but (almost) trivial**

EdgeBetweenness [EB] modification

- ❑ Original:
  1. Calculate betweenness score for each edge in the graph
  2. Remove edge with the highest score
  3. Repeat 1. and 2. until no edges remain
  4. Return division with the highest modularity score

- ❑ Modification:
  1. Calculate betweenness score for each edge in the graph
  2. Remove first edge with the the highest score which links no two nodes from the same seed community
  3. Repeat 1. and 2. until only edges linking nodes from the same seed community remain
  4. Return division with the highest modularity score

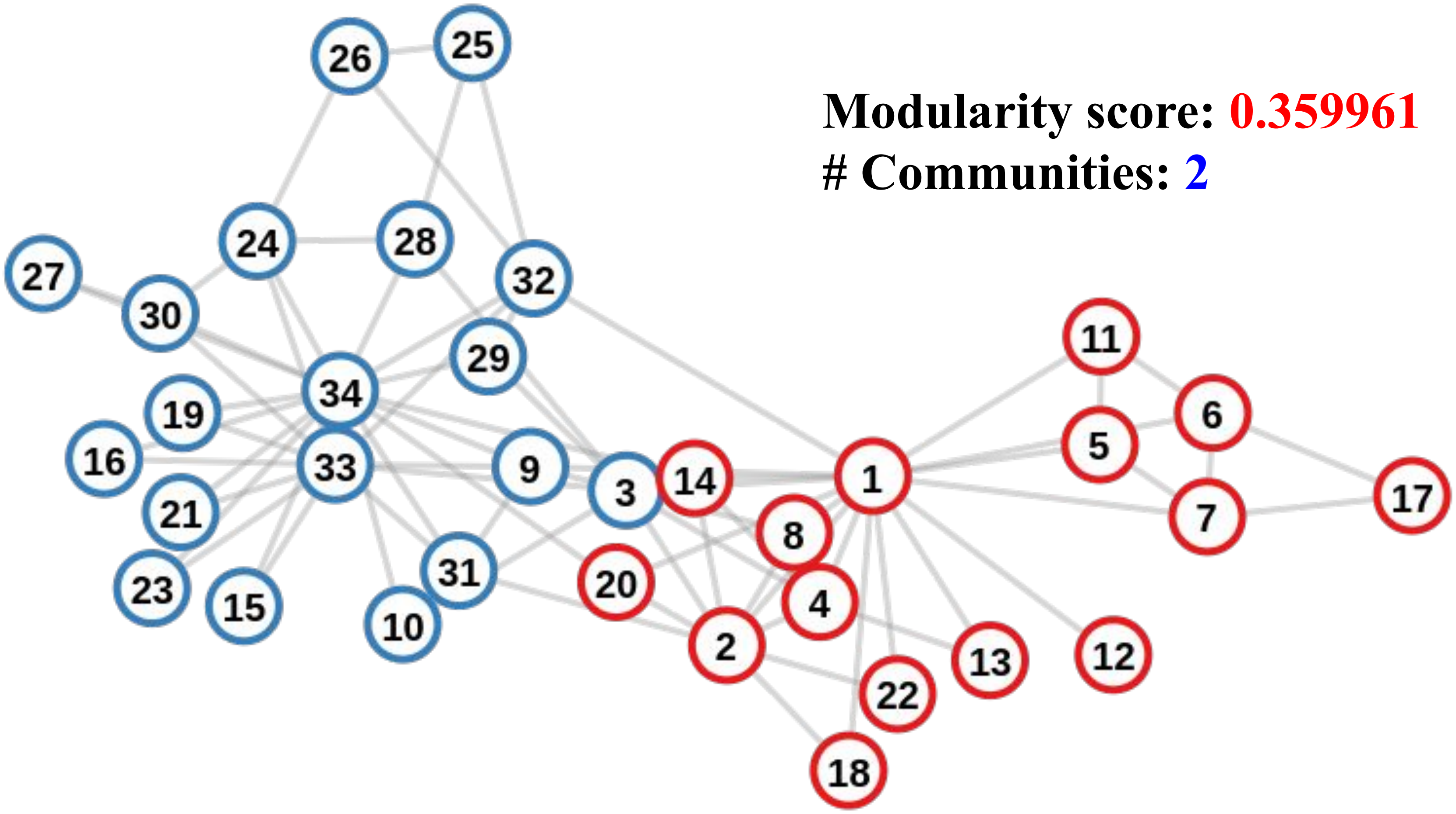
Zachary's karate club [ZKC]



34 := Instructor

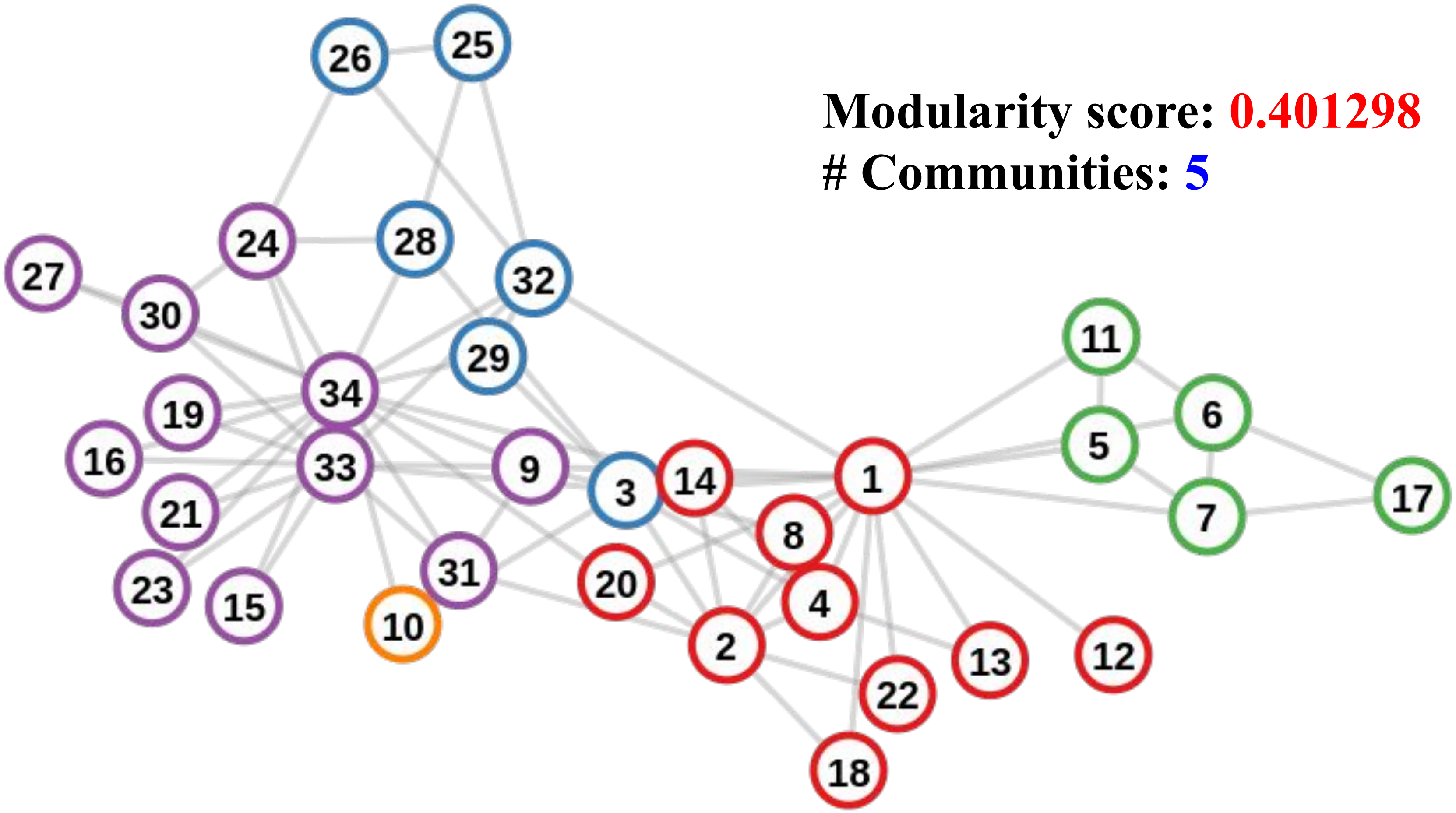
1 := Administrator

Zachary's karate club [ZKC]



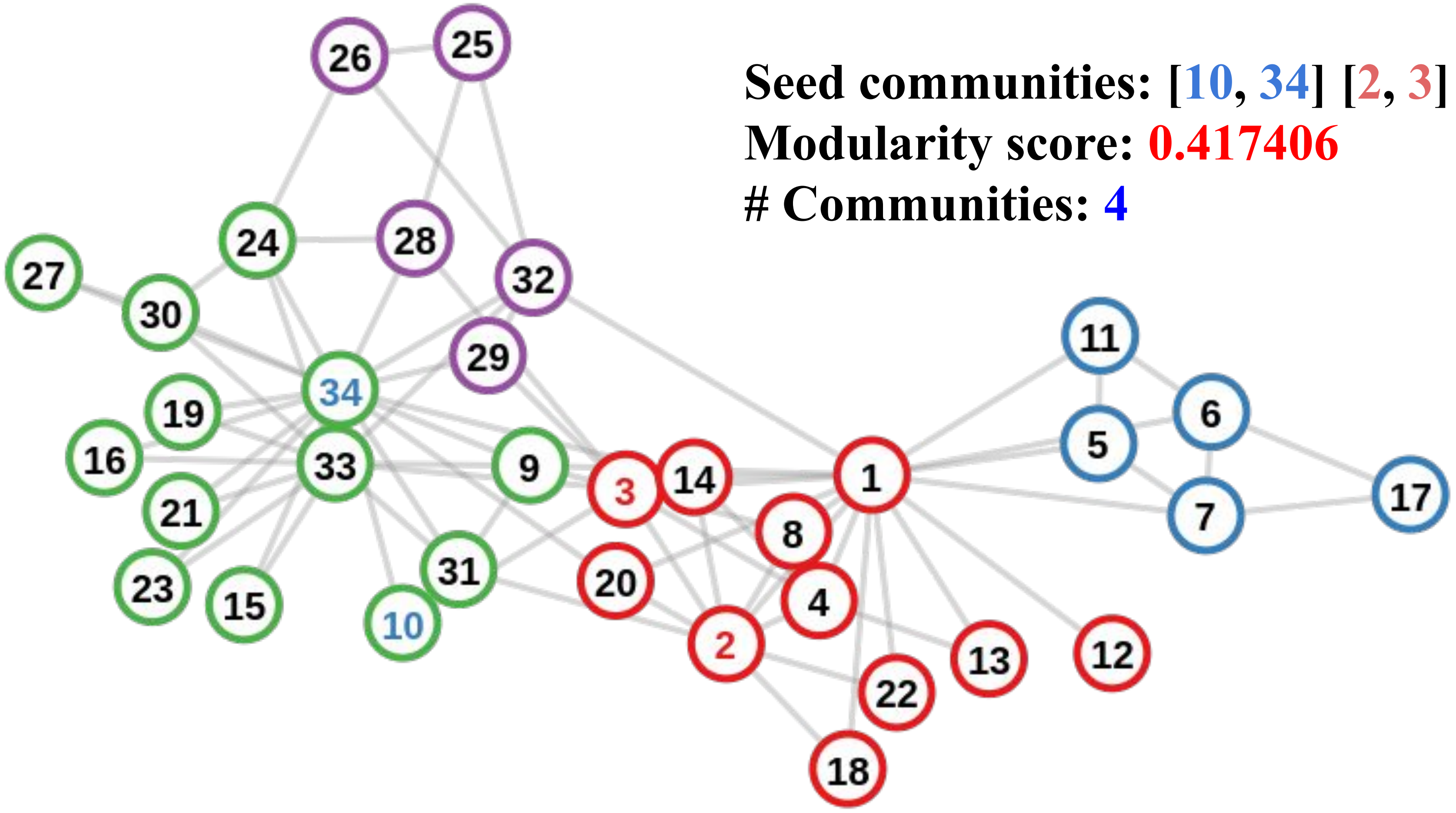
Modularity score: 0.359961
# Communities: 2

EdgeBetweenness on ZKC



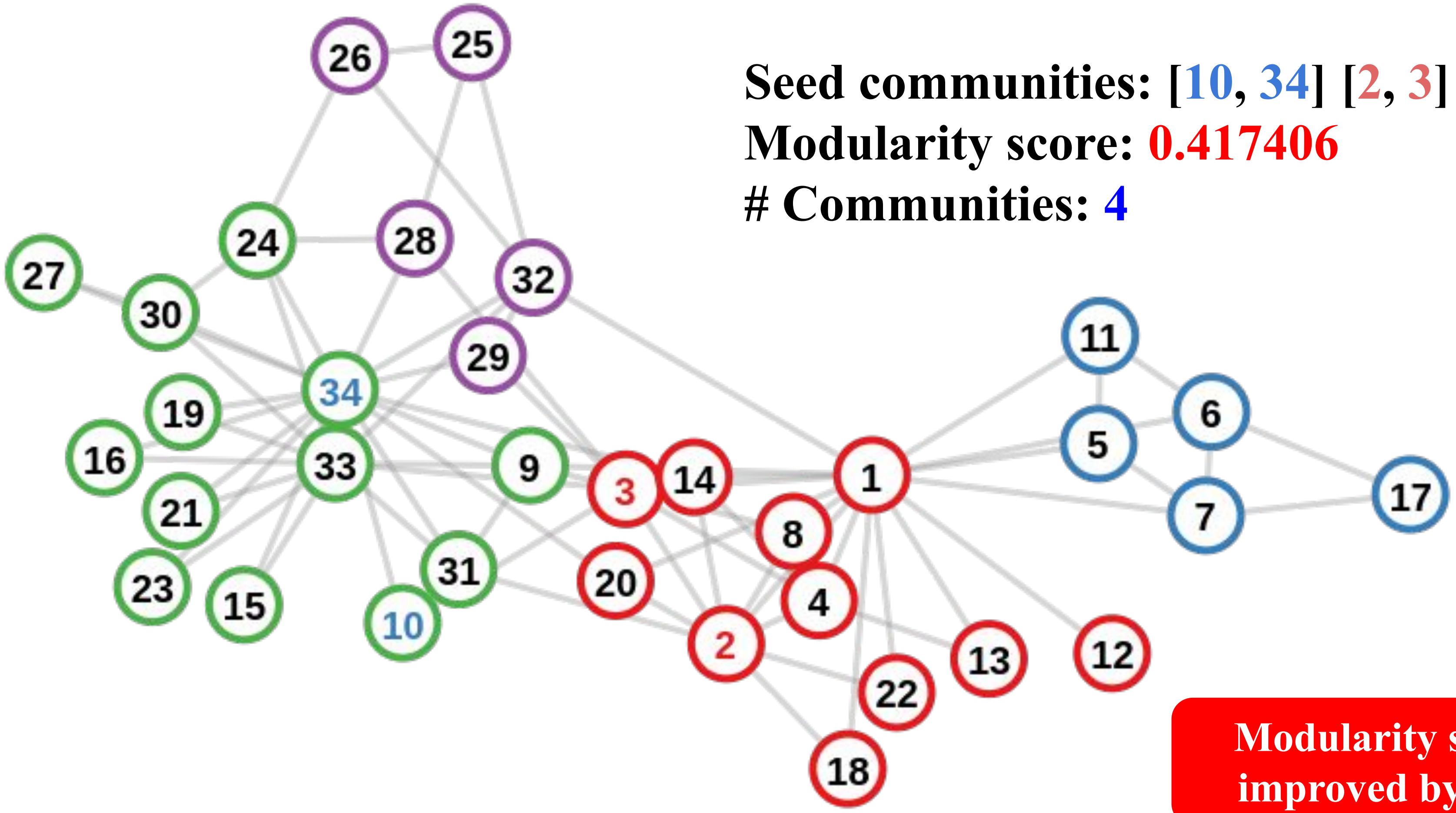
Modularity score: 0.401298
# Communities: 5

EB modification on ZKC (seed set 1)



Seed communities: [10, 34] [2, 3]
Modularity score: 0.417406
# Communities: 4

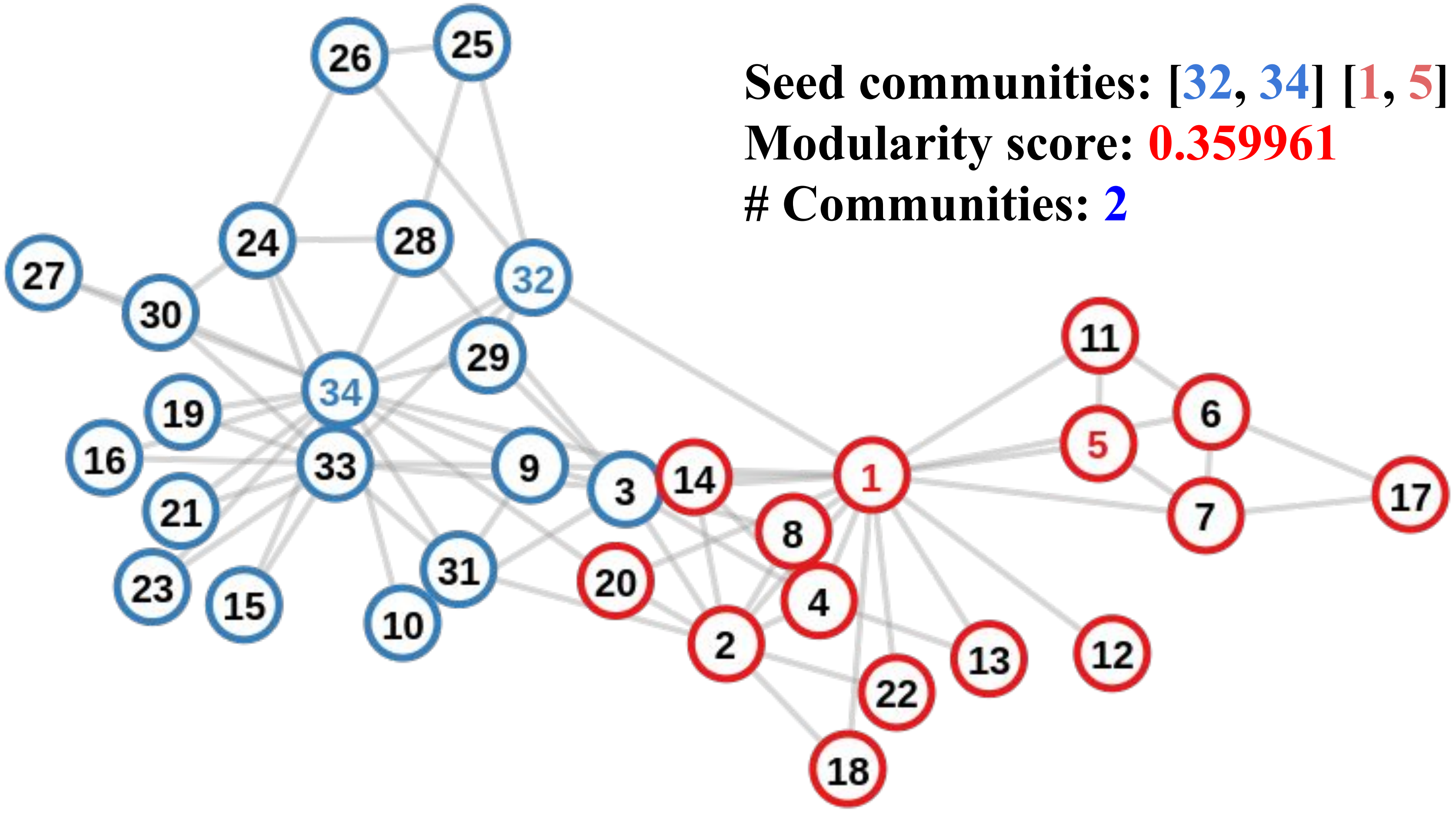
EB modification on ZKC (seed set 1)



Seed communities: [10, 34] [2, 3]
Modularity score: 0.417406
# Communities: 4

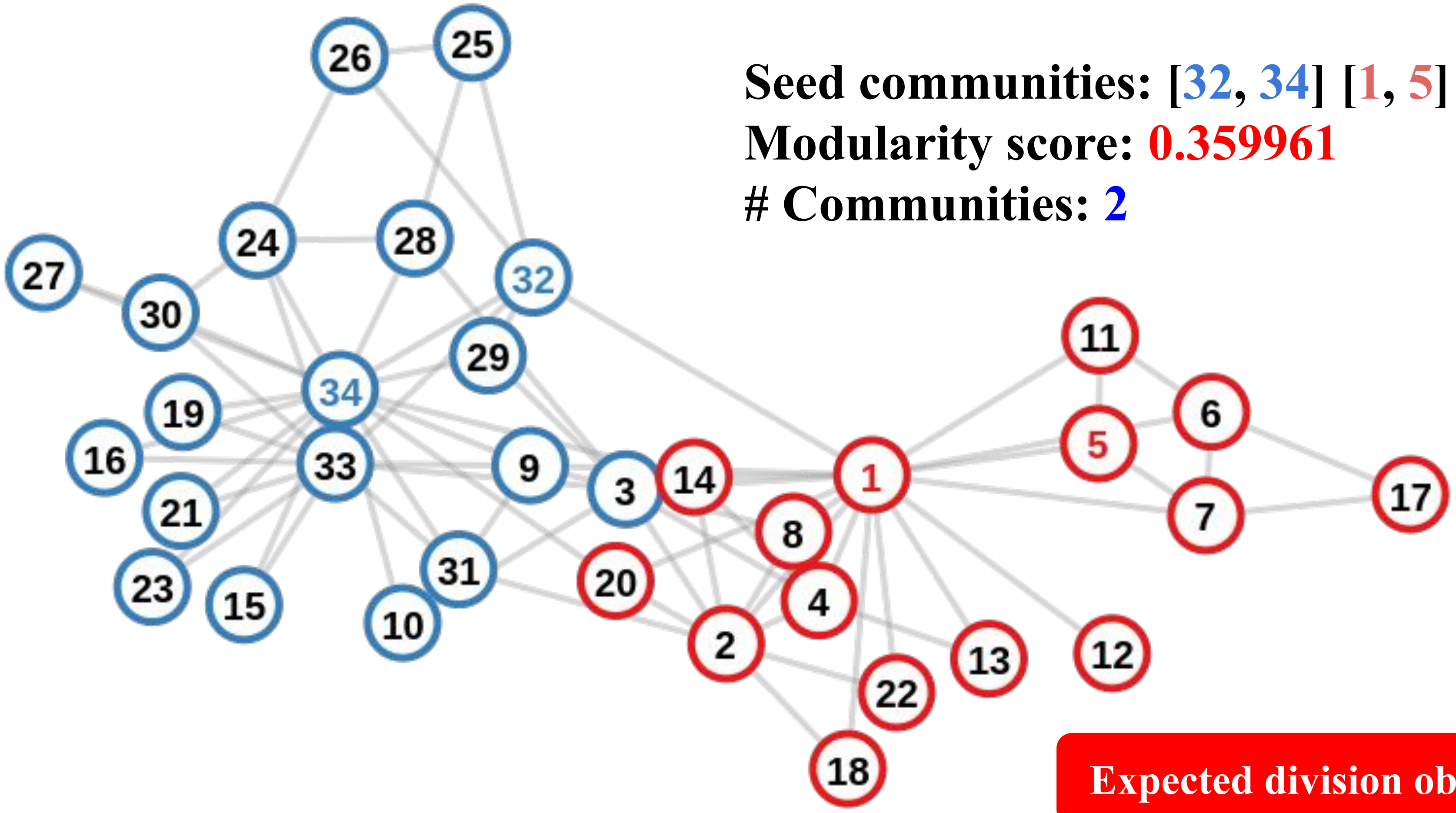
Modularity score improved by 4%

EB modification on ZKC (seed set 2)



Seed communities: [32, 34] [1, 5]
Modularity score: 0.359961
# Communities: 2

EB modification on ZKC (seed set 2)



Seed communities: [32, 34] [1, 5]
Modularity score: 0.359961
# Communities: 2

Expected division obtained

FastGreedy [FG] algorithm

- The algorithm uses a greedy optimization in which, starting with each vertex being the sole member of a community of one, we repeatedly join together the two communities whose amalgamation produces the largest increase in modularity Q

Probability a random edge would fall into community i

$$Q = \sum_i (e_{ii} - a_i^2)$$

Probability edge is in community i

High modularity = more edges within community than you expect by chance

$$\underbrace{e_{ij} = \frac{1}{2m} \sum_{vw} A_{vw} \delta(c_v, i) \delta(c_w, j)}_{} \qquad \underbrace{a_i = \frac{1}{2m} \sum_v k_v \delta(c_v, i)}_{}$$

Fraction of edges that join vertices in community i to vertices in community j

Fraction of ends of edges that are attached to vertices in community

FastGreedy [FG] algorithm

- We maintain 3 data structures:
  - A sparse matrix containing ΔQ_{ij} for each pair i, j of communities with at least one edge between them.
    - We store each row of the matrix both as a balanced binary tree (so that elements can be found or inserted in O(log n) time) and as a maxheap (so that the largest element can be found in constant time)
  - A max-heap H containing the largest element of each row of the matrix ΔQ_{ij} along with the labels i, j of the corresponding pair of communities.
  - An ordinary vector array with elements a_i

FastGreedy [FG] algorithm

1. Calculate the initial values of ΔQ_{ij} and a_i as below and populate the max-heap with the largest element of each row of the matrix ΔQ

   $\Delta Q_{ij} = 2\,[\,1/2m - k_i k_j / (2m)^2\,]$ if i and j are connected, 0 otherwise

   $a_i = k_i / 2m$

2. Select the largest ΔQ_{ij} from H, join the corresponding communities, update the matrix ΔQ, the heap H and a_i and increment Q by ΔQ_{ij}
3. Repeat step 2 until only one community remains.

Let's skip updating rules...

FG modification

1. Assign seed communities to given nodes and rest of the nodes assign to different communities as in the original
2. Calculate Q (modularity) for a state from 1.
3. Calculate a_i according to the original equation for a state from 1.
4. Calculate e'_{ij} according to algorithm $^{(*)}$ [more on the next slide]
5. Calculate initial ΔQ_{ij} according to equation:

$$\begin{cases} (e'_{ij} / m) - 2a_i a_j & \text{if } C_i\ C_j \text{ are connected} \\ 0 & \text{otherwise} \end{cases}$$

6. Continue as in the original algorithm

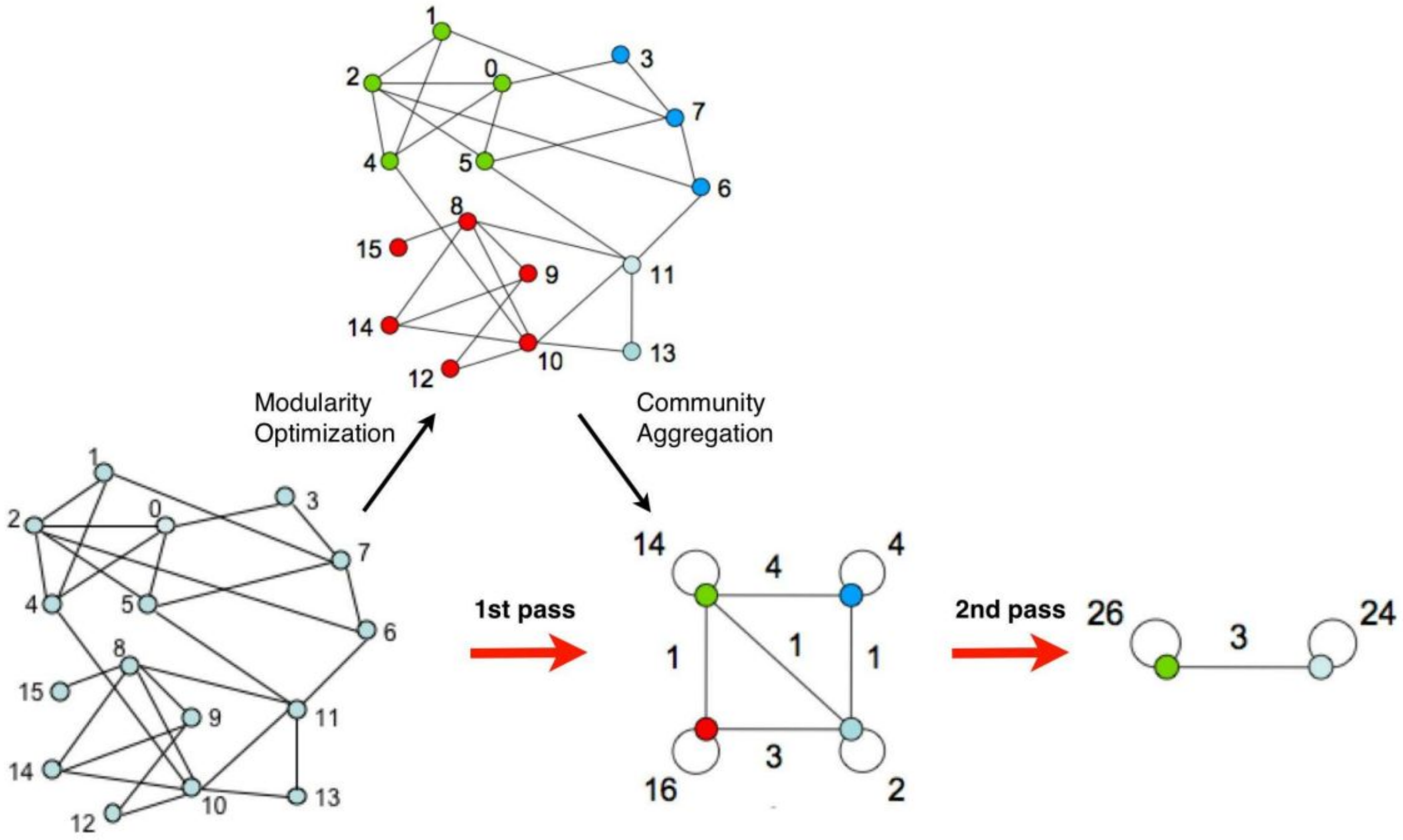
FG modification - algorithm$^{(*)}$ details

```
1. CT = AVLtree(key: node, value: community of node);
2. Initialize CT according to seed communities;
3. EW = empty HashMap; // values of $e'_{ij}$
4. For e = (v, w) in E:
       EW[ CT[v], CT[w] ]++ or init to 1 if not exists
5. Return EW;
```

Louvain algorithm

- First, assign a different community to each node of the network (so, in this initial partition there are as many communities as there are nodes)
- Then, the algorithm is divided in two phases (let denote by "pass" combination of these two phases):
  - 1st phase: modularity is optimized by allowing only local changes of communities
  - 2nd phase: found communities are aggregated in order to build a new network of communities
- The passes are repeated iteratively until no increase of modularity is possible

Louvain algorithm



Louvain algorithm - 1st pass details

- ❑ For each node X we consider the neighbours Y of X and we evaluate the gain of modularity that would take place by removing X from its community and by placing it in the community of Y
- ❑ The node X is then placed in the community for which this gain is maximum, but only if this gain is positive. If no positive gain is possible, X stays in its original community.
- ❑ This process is applied repeatedly and sequentially for all nodes until no further improvement can be achieved and the first phase is then complete
- ❑ Let insist on the fact that a node may be, and often is, considered several times
- ❑ Note that the output of the algorithm depends on the order in which the nodes are considered

Louvain algorithm - 1st pass details

- Gain in modularity ΔQ obtained by moving an isolated node i into a community C:

$$\Delta Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

- Σ_{in} := sum of the weights of the links inside C
- Σ_{tot} := sum of the weights of the links incident to nodes in C
- k_i := sum of the weights of the links incident to node i
- $k_{i,in}$:= sum of the weights of the links from i to nodes in C
- m := sum of the weights of all the links in the network

Louvain algorithm - 1st pass details

- Gain in modularity ΔQ obtained by moving an isolated node i into a community C:

$$\Delta Q = \underbrace{\left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right]}_{\text{Contribution to modularity of the new community (node i joined with C)}} - \left[\underbrace{\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2}_{\text{Contribution of community C}} - \underbrace{\left(\frac{k_i}{2m}\right)^2}_{\text{Contribution of node i's community}}\right]$$

Reminder

$$Q = \sum_i (e_{ii} - a_i^2) \qquad e_{ij} = \frac{1}{2m}\sum_{vw} A_{vw}\delta(c_v, i)\delta(c_w, j) \qquad a_i = \frac{1}{2m}\sum_v k_v \delta(c_v, i)$$

Louvain modification

- Assign seed communities to given nodes and rest of the nodes assign to different communities as in the original
- 1st pass: let create iterator consisting of seed communities and rest of the nodes
  - For each element X of this iterator let consider its neighbours Y (i.e. depending on type of X those are neighbours of a single node or all nodes linked to the community)
  - If X is a node, then act as normally
  - Otherwise X is a seed community and place it in the community for which the gain is maximum, but only if this gain is positive. If no positive gain is possible, it stays in its original community.
- 2nd pass remains unchanged

Many thanks!

Contact me: a.myszkowski@students.mimuw.edu.pl