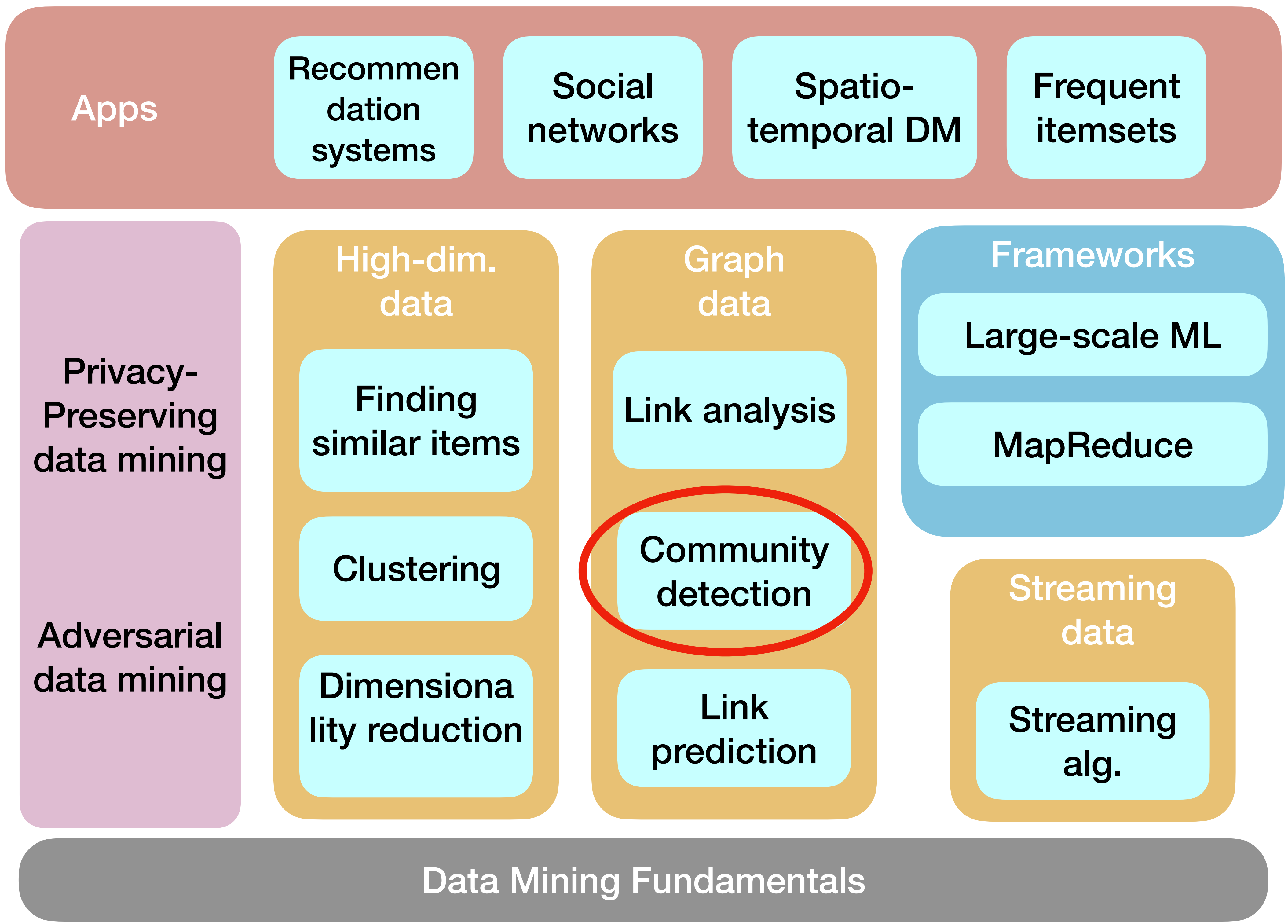
# Community Detection

Liyao Xiang
http://xiangliyao.cn/
Shanghai Jiao Tong University

# Course Landscape

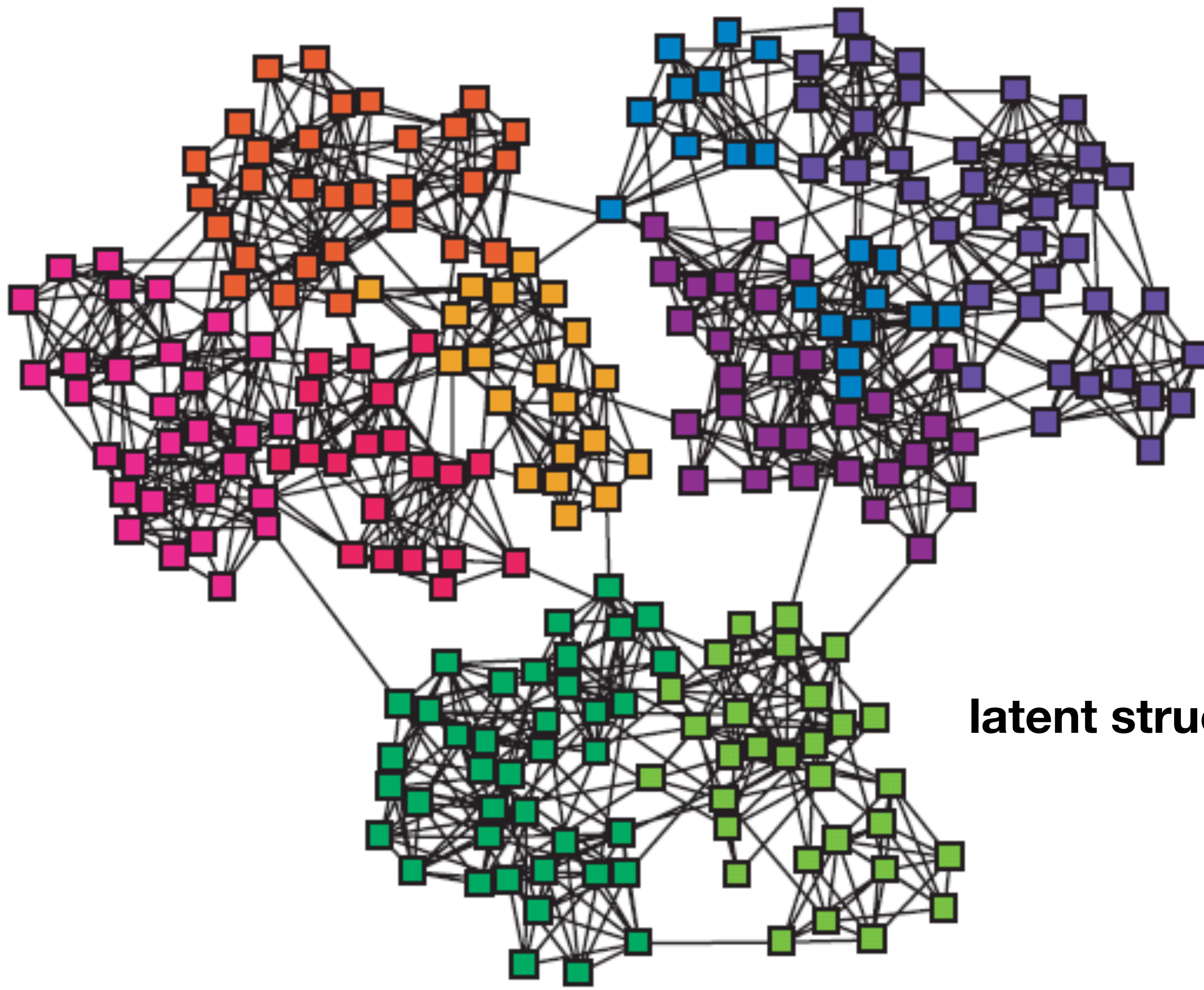


# Outline

- Motivation
- PageRank based Clustering
- Modularity Maximization

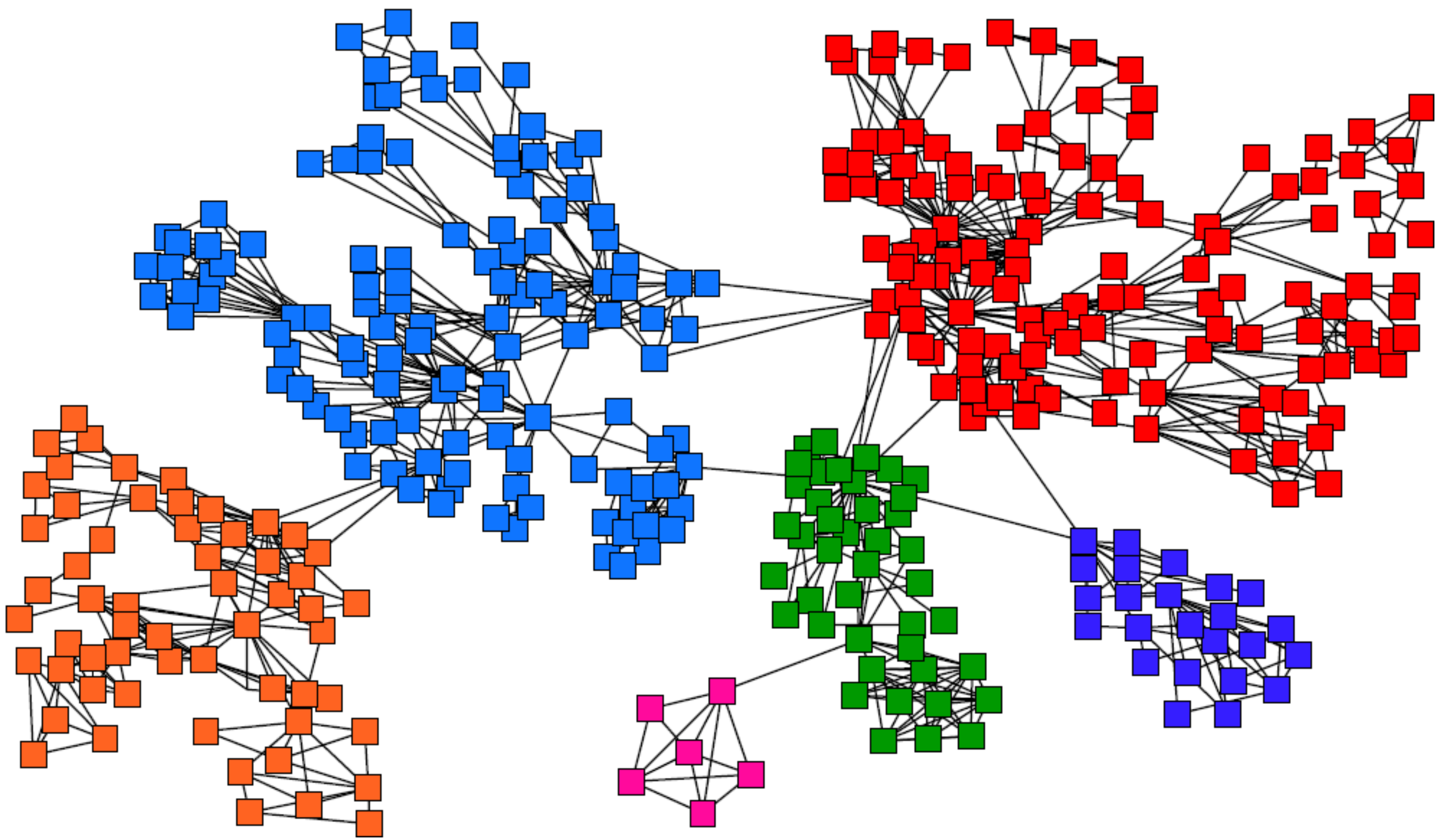
# Networks & Communities

- We often think of networks being organized into **modules, clusters, communities**:

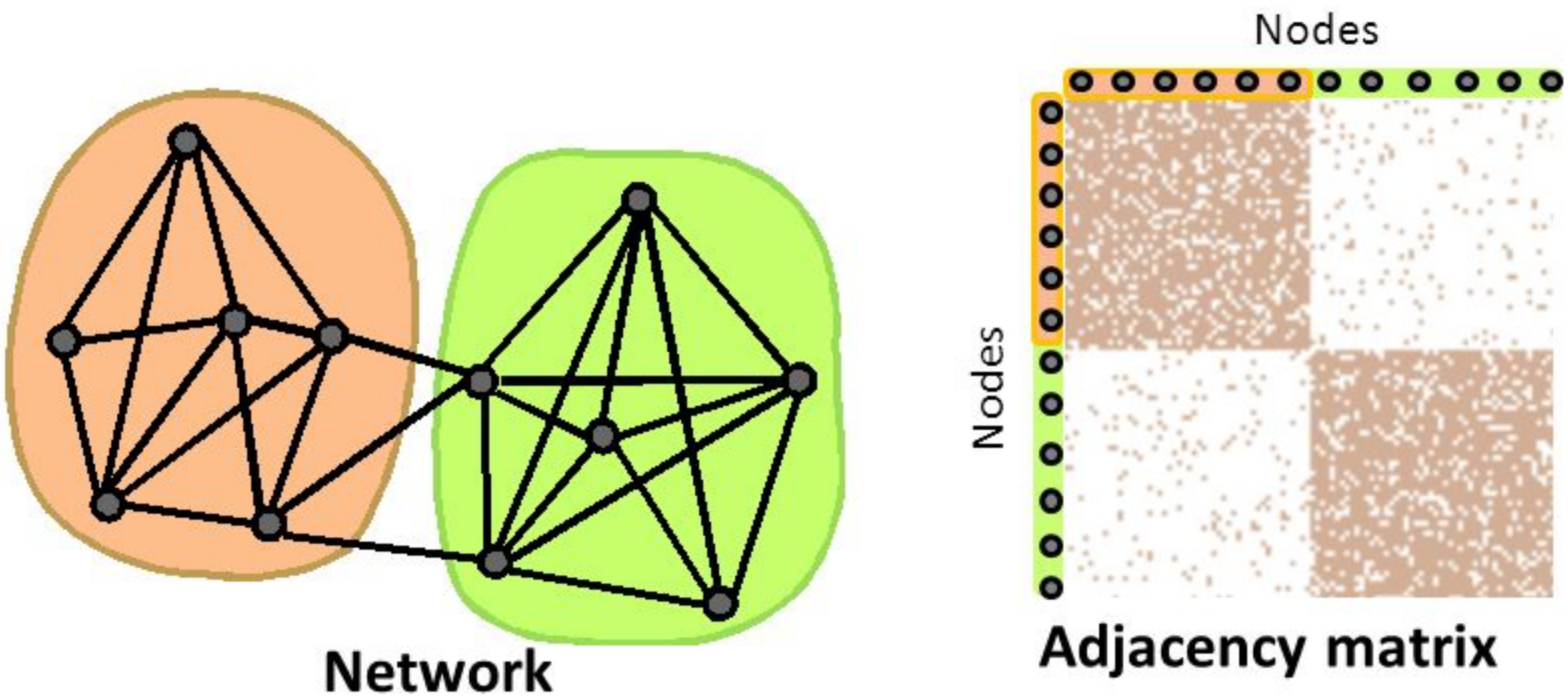


**latent structure!**

# Goal: Find Densely Linked Clusters

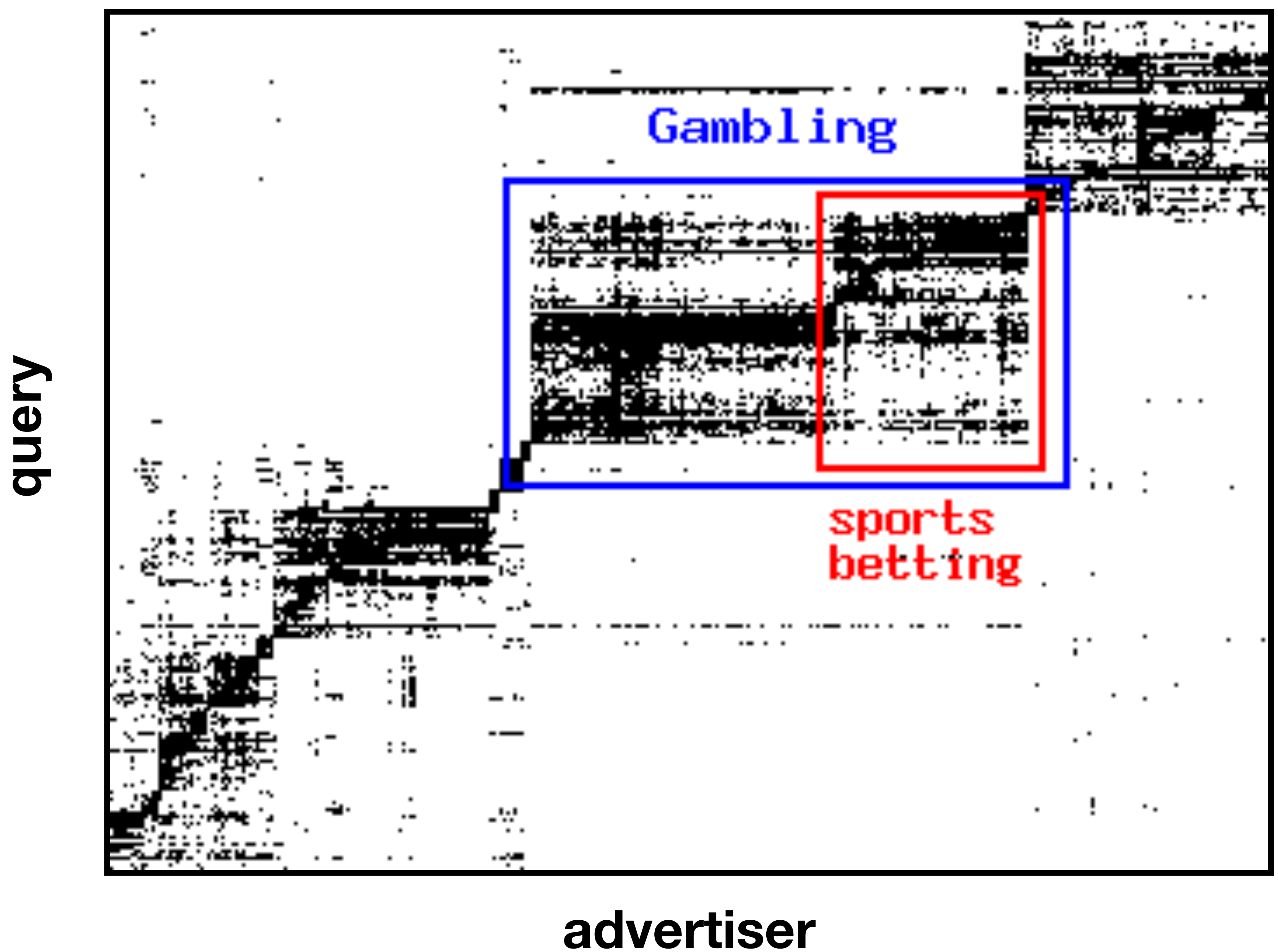


# Non-overlapping Communities



# Micro-Markets in Sponsored Search

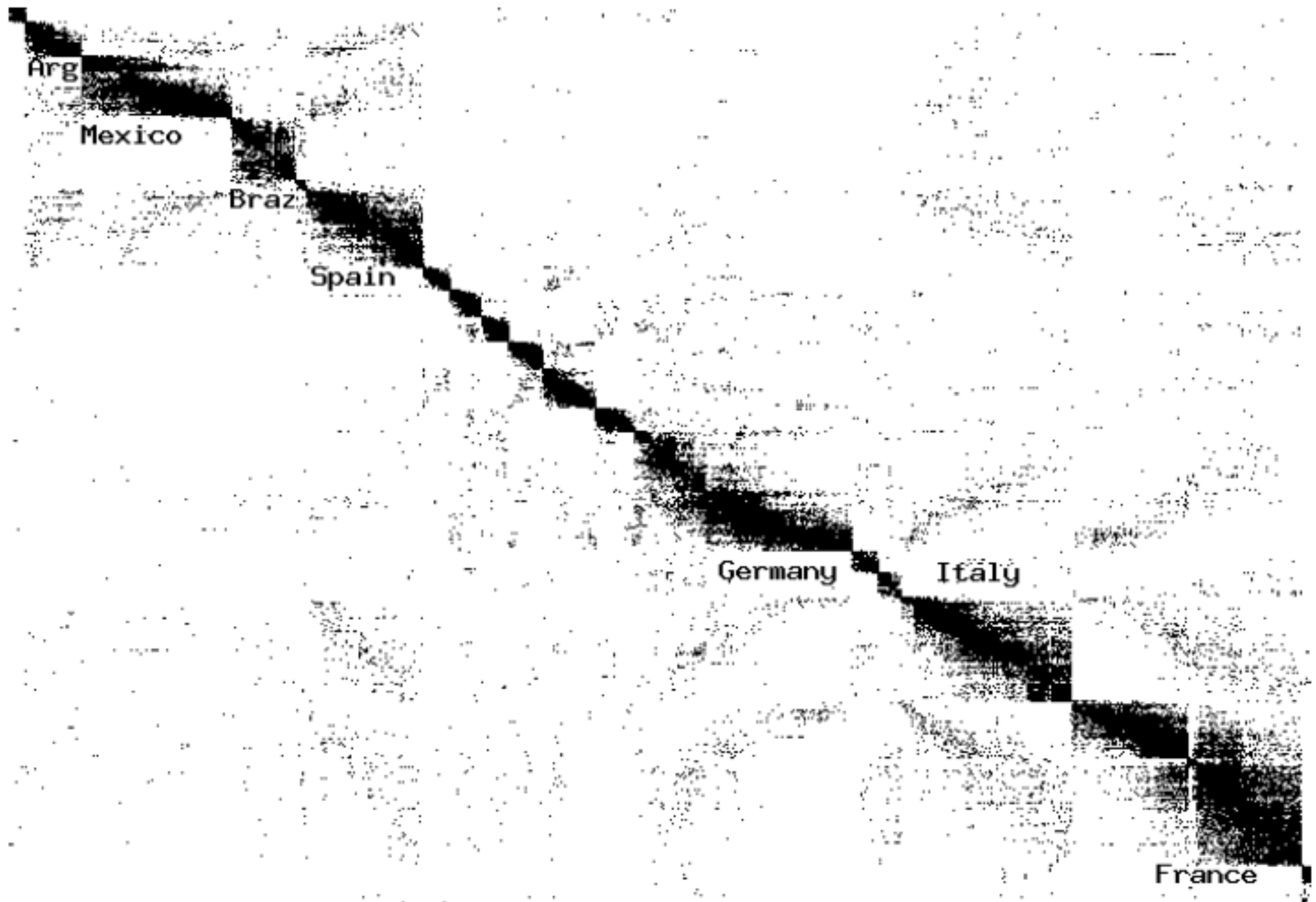
- Find micro-markets by partitioning the query-to-advertiser graph:




[Andersen, Lang: Communities from seed sets, 2006]

# Movies and Actors

- Clusters in Movies-to-Actors graph:

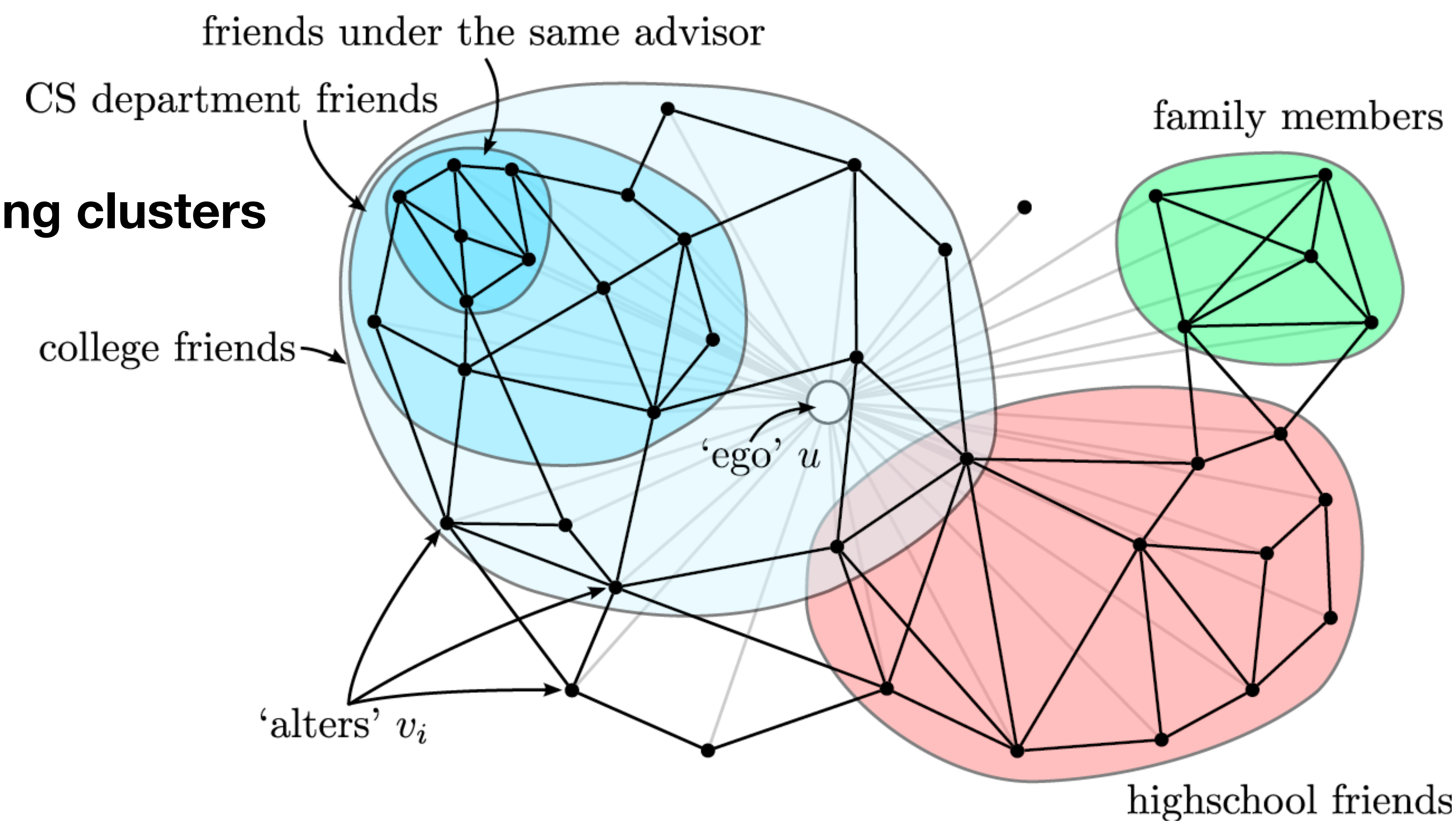



[Andersen, Lang: Communities from seed sets, 2006]

# Twitter & Facebook

- Discovering social circles, circles of trust:

**Overlapping clusters**




[McAuley, Leskovec: Discovering social circles in ego networks, 2012]

# Outline

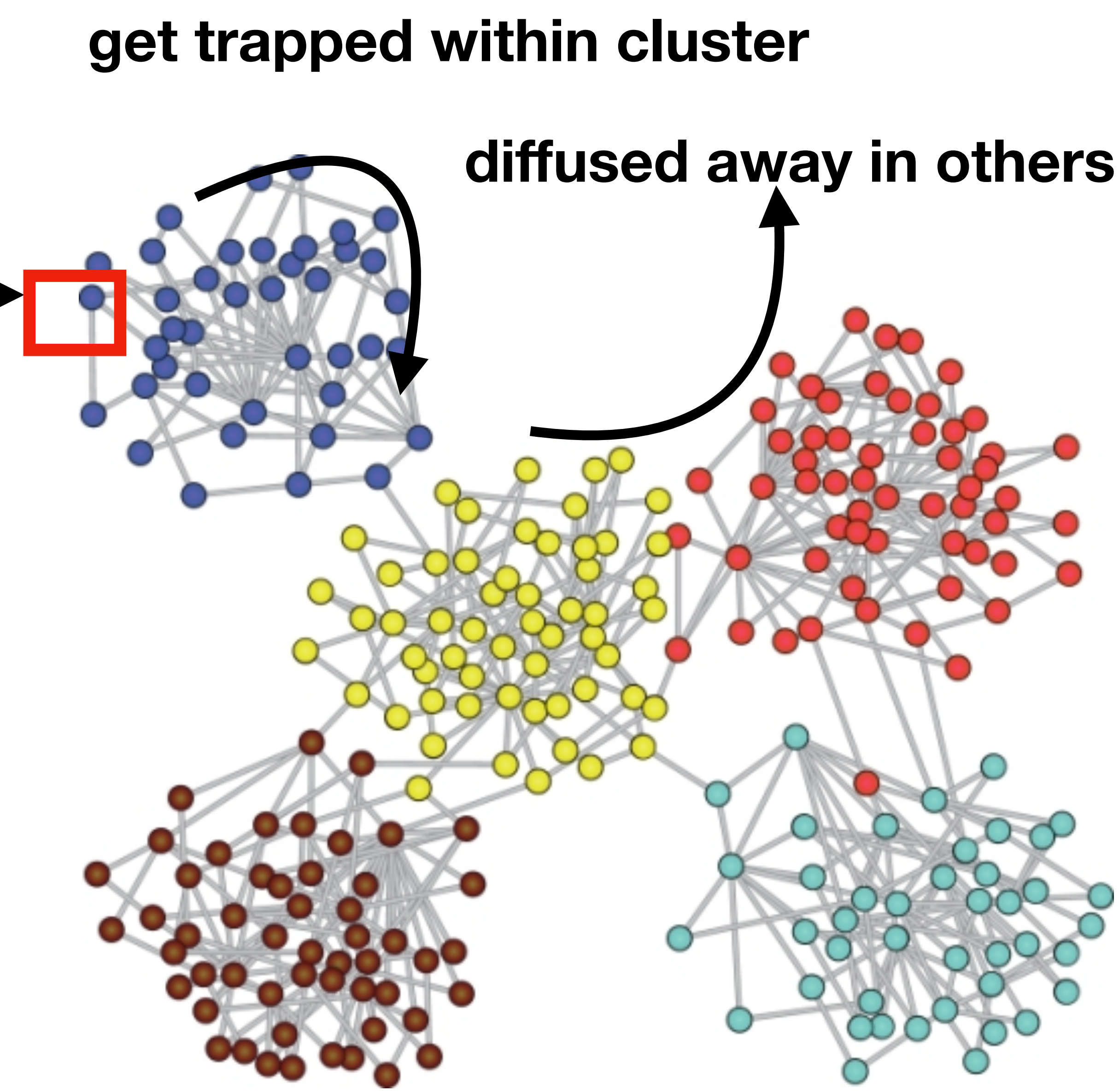
- Motivation
- PageRank based Clustering
- Modularity Maximization

# Setting

- Graph is large: assume the graph fits in main memory
    - e.g., to work with a 200M node and 2B edge graph, one needs approximately 16GB RAM
    - But the graph is **too big** for running anything more than linear time alg.
- **PageRank based alg. for finding dense clusters**
    - The runtime of the alg. will be proportional to the **cluster size** (not the graph size!)

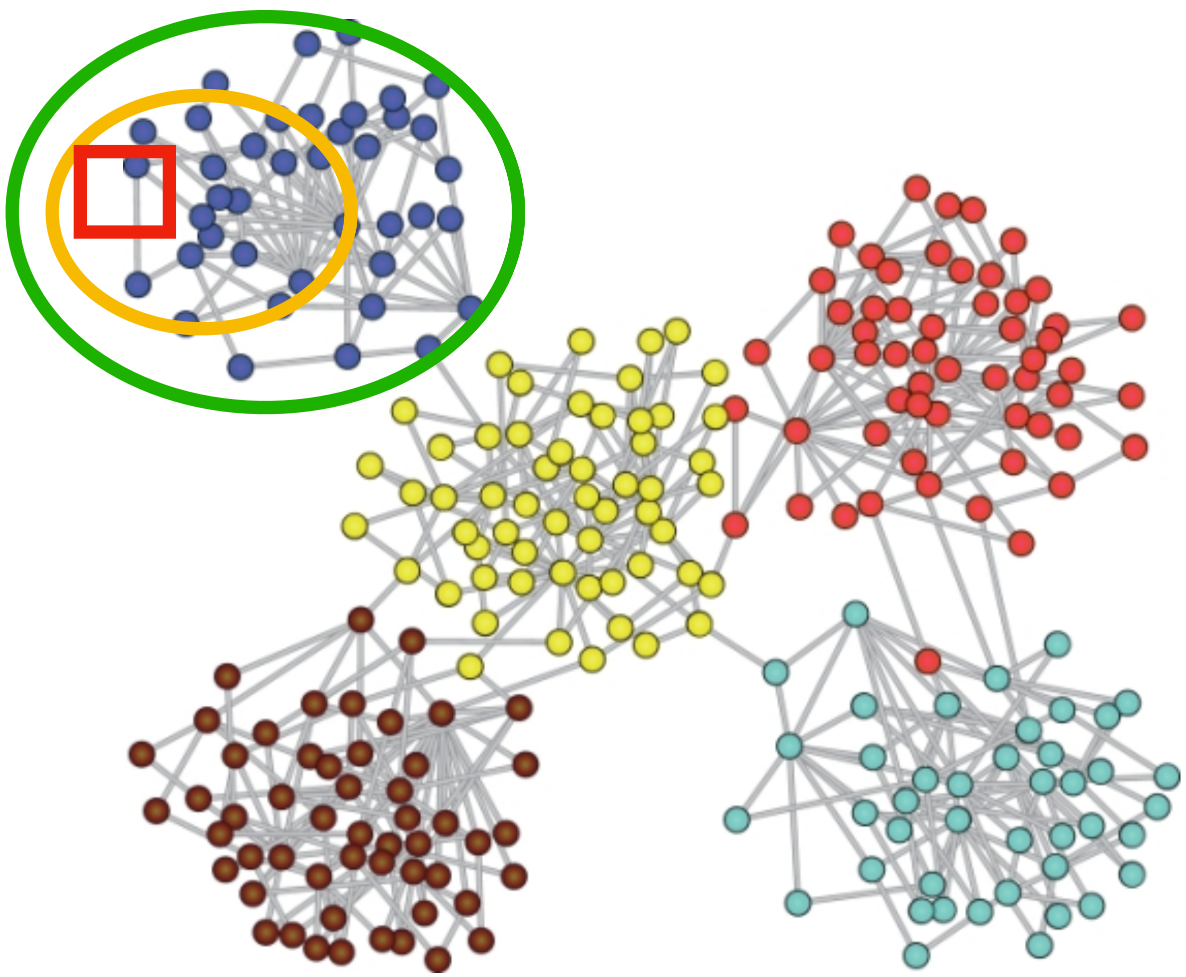
# Idea: Seed Nodes

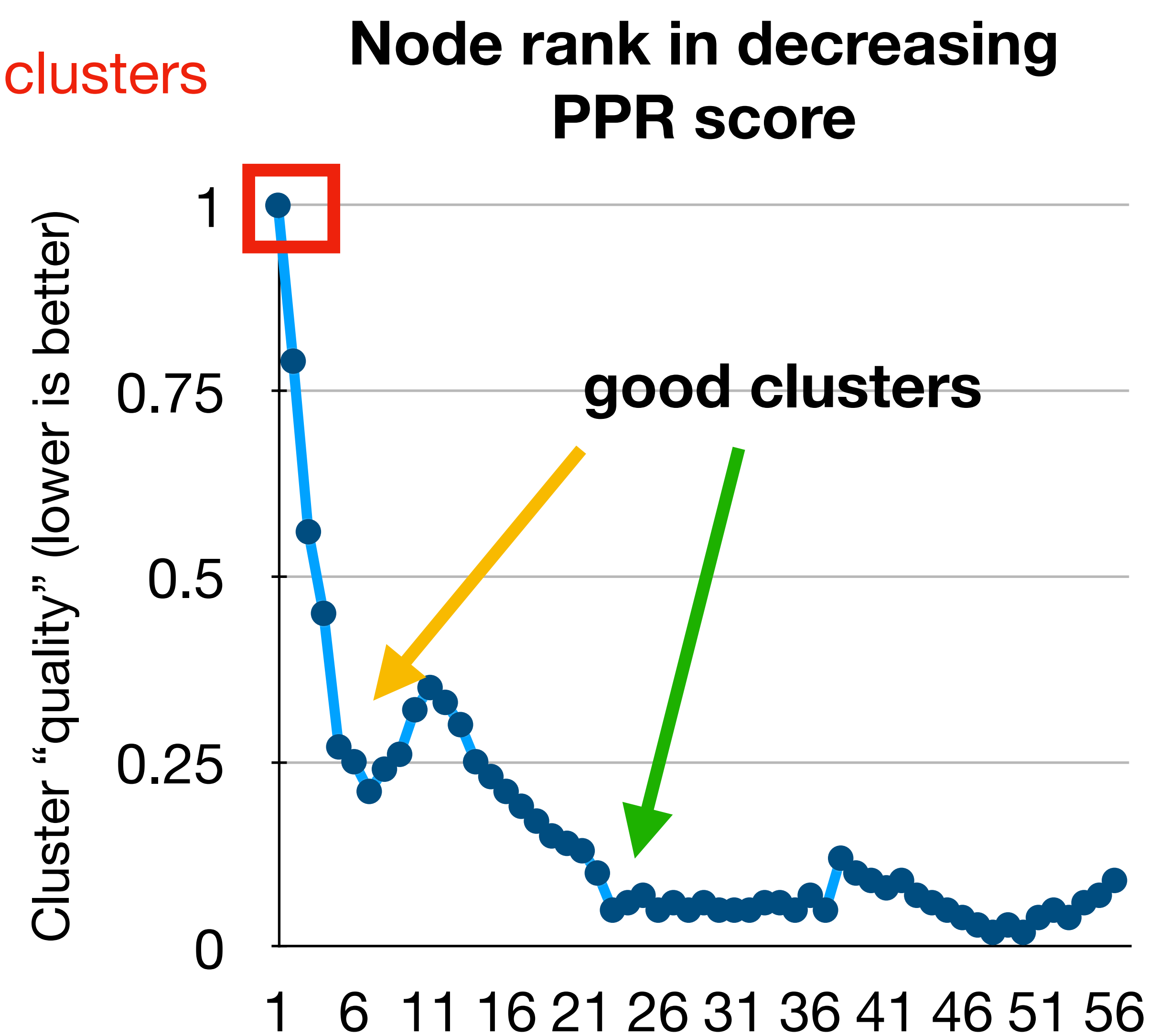
- Discovering clusters based on seed nodes
  - Given: Seed node **s**
  - Compute (approximate) Personalized PageRank (PPR) around node **s** (teleport set = {**s**})
  - Idea is that if **s** belongs to a nice cluster, the random walk will get **trapped** inside the cluster



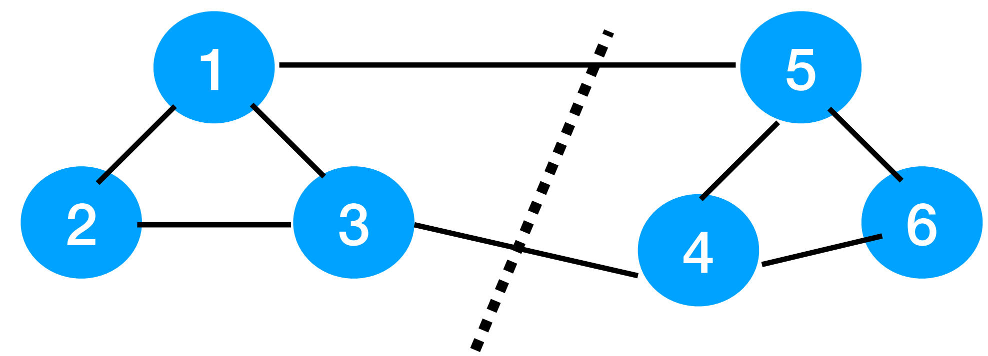
# Intuition

- Alg. outline:
  - Pick a seed node **s** of interest
  - Run PPR with teleport set = {**s**}
  - Sort the nodes by the decreasing PPR score
  - **Sweep** over the nodes and find good clusters

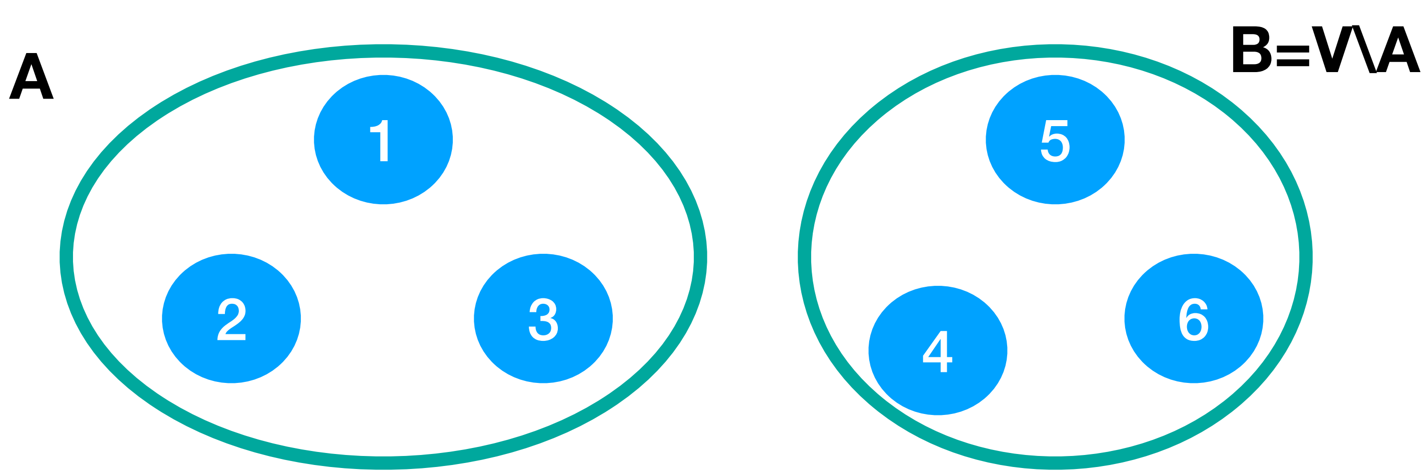




# What makes a good cluster?




- Undirected graph G(V, E)
- Partitioning task:
    - Divide vertices into 2 disjoint groups A, B=V \ A
- Question:
    - How can we define a “good” cluster in G?




**Surface vs. Volume**

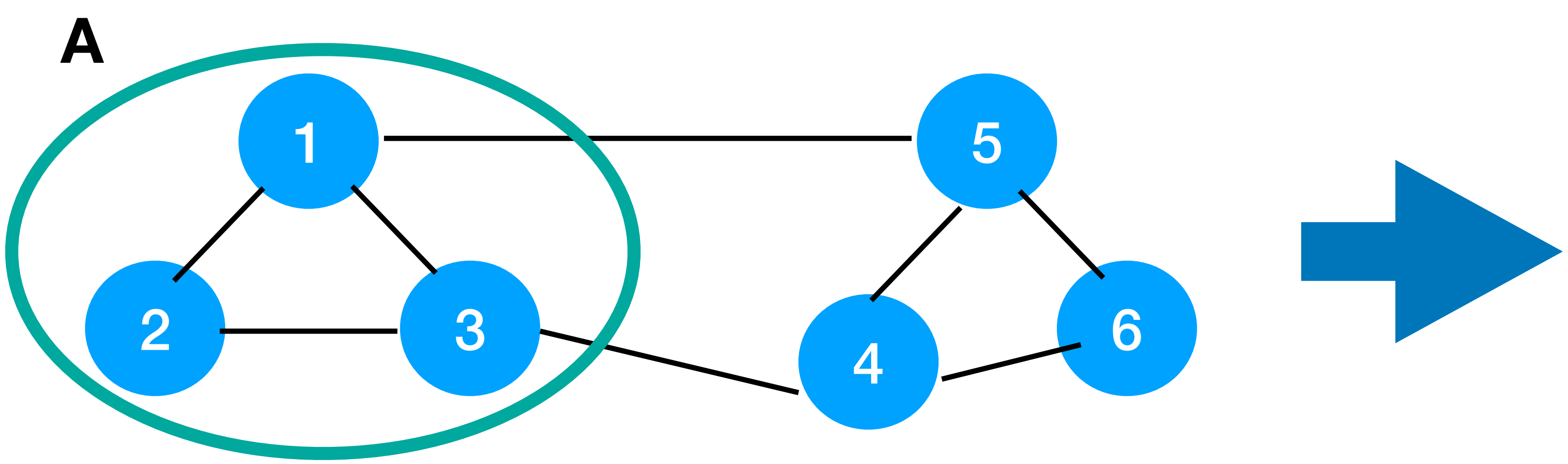
- Maximize the number of within-cluster connections
- Minimize the number of between-cluster connections

# Graph Cuts

- Express **cluster quality** as a function of the "edge cut" of the cluster
- **Cut**: set of edges (edge weights) with only one node in the cluster:

$$cut(A) = \sum_{i \in A, j \notin A} w_{ij}$$

Note: this works for weighted and unweighted (set all $w_{ij}$=1) graphs

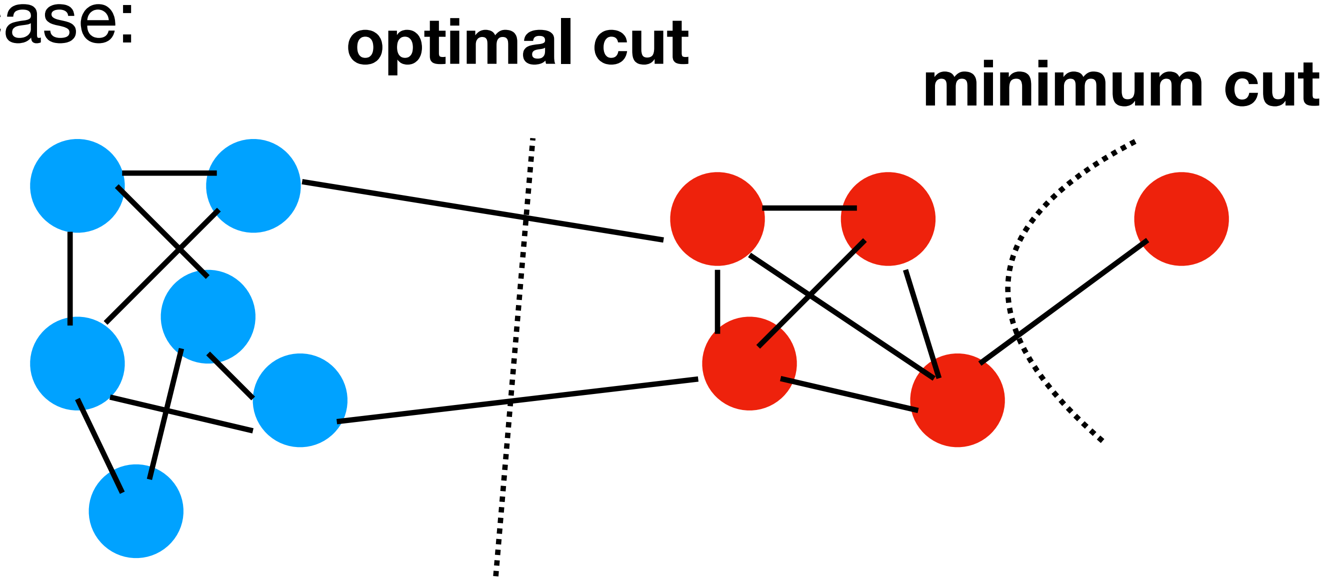



**cut(A) = 2**

**minimize the cut?**

# Cut Score

- Partition quality: **cut score**
  - Quality of a cluster is the weight of connections pointing outside the cluster
  - Degenerate case:




- Problem:
  - Only considers **external** cluster connections
  - Does not consider **internal** cluster connectivity

# Graph Partitioning Criteria

- Criterion: **Conductance** — connectivity of the group to the rest of the network relative to the density of the group

$$\phi(A) = \frac{|\{(i,j) \in E; i \in A, j \notin A\}|}{\min(vol(A), 2m - vol(A))}$$

  - m: number of edges of the graph
  - E: edge set of the graph
  - $d_i$: degree of node i
  - vol(A): total weight of the edges with at least one endpoint in A:

$$vol(A) = \sum_{i \in A} d_i$$

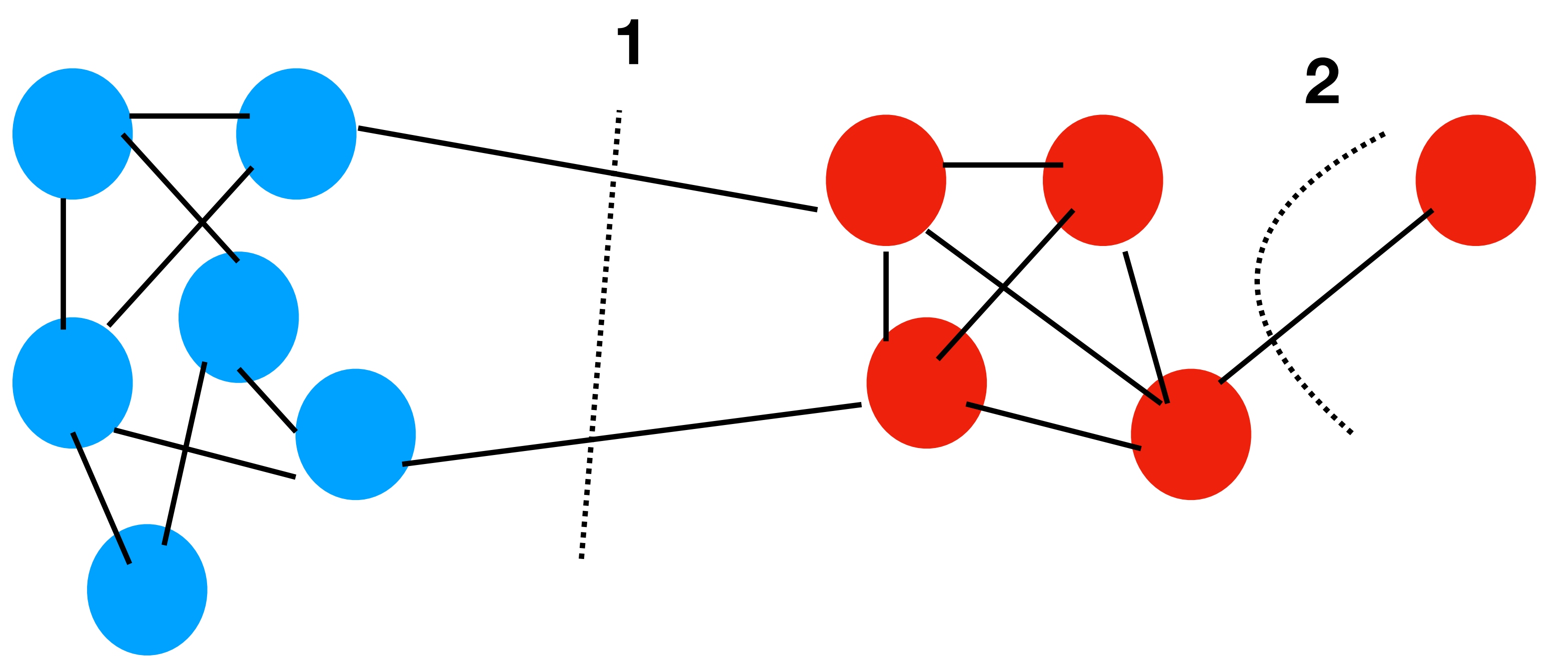
**vol(A) = 2 * #edges inside A + #edges pointing out of A**

- Why use this criteria? Produces more balanced partitions

# Example

$$\phi(A) = \frac{|\{(i,j) \in E; i \in A, j \notin A\}|}{\min(vol(A), 2m - vol(A))}$$

- What are the conductance of the following cut?



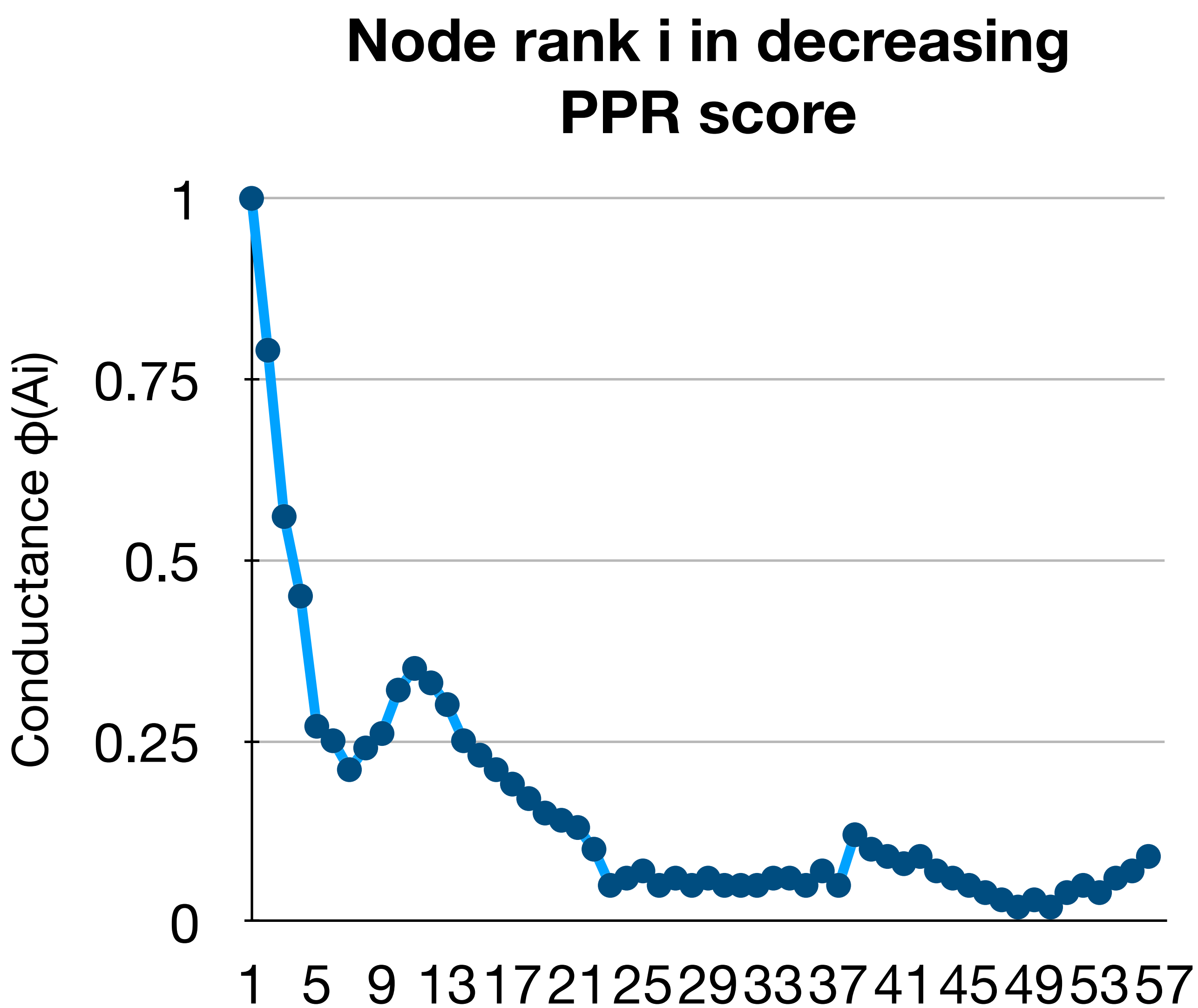

$\phi_1 = 2/16 = 0.125$

$\phi_2 = 1/1 = 1.0$

**Why do we take minimum?**

# Algorithm Outline: Sweep

- Alg. outline:
  - Pick a seed node **s** of interest
  - Run PPR w/ teleport={**s**}
  - Sort the nodes by the decreasing **PPR** score
  - Sweep over the nodes and find good clusters

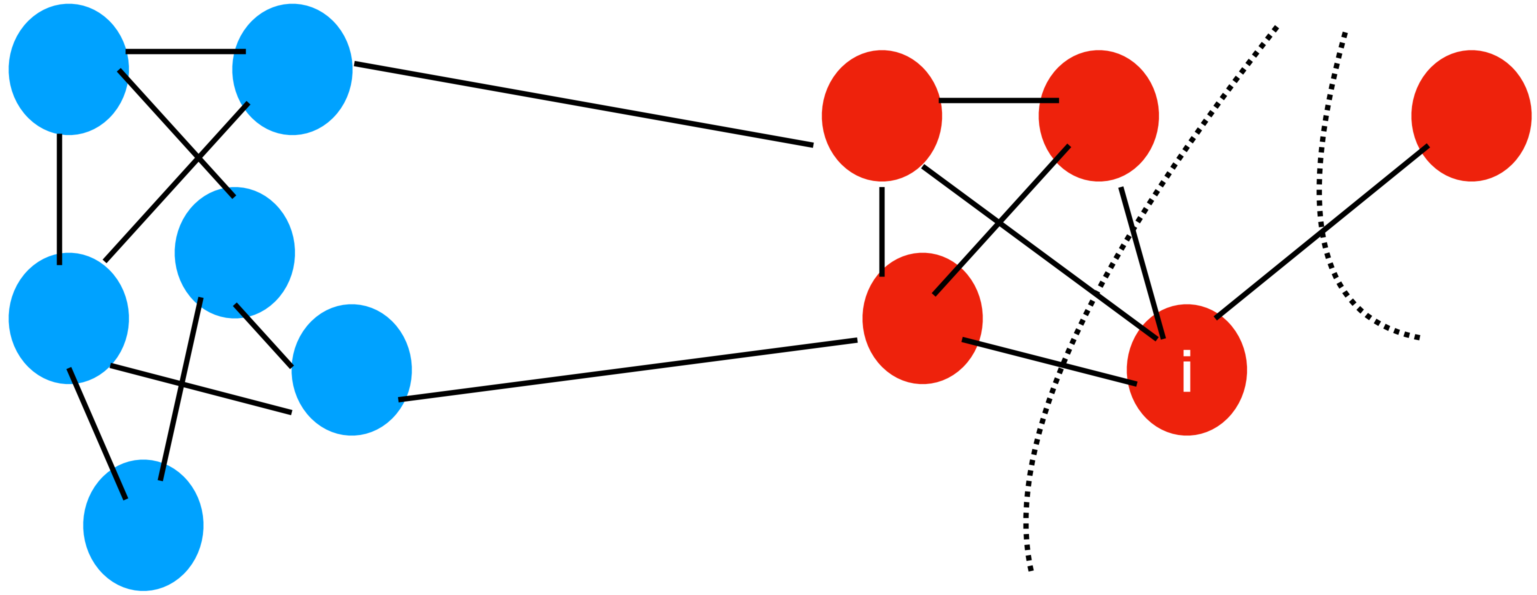



- Sweep:
  - Sort nodes in decreasing PPR score $r_1 > r_2 > \ldots > r_n$
  - For each i compute $\phi(A_i = \{r_1, \ldots r_i\})$
  - **Local minima** of $\phi(A_i)$ correspond to good clusters

# Computing the Sweep

- The whole Sweep curve can be computed in **linear** time:
  - For loop over the nodes
  - Keep hash-table of nodes in a set Ai
  - To compute $\phi(A_{i+1}) = \mathrm{Cut}(A_{i+1})/\mathrm{Vol}(A_{i+1})$
    - $\mathrm{Vol}(A_{i+1}) = \mathrm{Vol}(A_i) + d_{i+1}$
    - $\mathrm{Cut}(A_{i+1}) = \mathrm{Cut}(A_i) + d_{i+1} - 2\#(\text{edges of } u_{i+1} \text{ to } A_i)$

**How Vol and Cut change when we include node i?**




$\mathrm{Vol}(A_i) = 1$

$d_{i+1} = 4$

edges of $u_{i+1}$ to $A_i = 1$

# Computing PPR

- How to compute Personalized PageRank (PPR) without touching the whole graph?
  - Power method won't work since each single iteration accesses all nodes of the graph:

    $r^{(t+1)} = \beta M \cdot r^{(t)} + (1 - \beta)a$

    **at index s**

    - a is a teleport vector: $a = [0 \ldots 0\ 1\ 0 \ldots\ 0]^T$
    - r is the personalized PageRank vector
- Approximate PageRank [Andersen, Chung, Lang,' 07]
  - A fast method for computing approximate Personalized PageRank (PPR) with teleport set = {**s**}
  - ApproxPageRank (**s**, β, **ε**): (seed node, teleportation param., approx. error param.)

# Approximate PPR: Overview

- Overview of the approximate PPR
  - **Lazy random walk**: a variant of a random walk that stays put with prob. 1/2 at each time step, and walks to a random neighbour the other half of the time:

$$r_u^{(t+1)} = \frac{1}{2} r_u^{(t)} + \frac{1}{2} \sum_{i \rightarrow u} \frac{1}{d_i} r_i^{(t)}$$

  - Keep track of **residual PPR score**:

**“true” PageRank of node u**

$q_u = p_u - r_u^{(t)}$

**PageRank estimate of node u at round t**

- Residual tells us how well PPR score $p_u$ of u is approximated
- If residual $q_u$ of node u is too big: $q_u / d_u >= \varepsilon$ then **push the walk further** (distribute some residual $q_u$ to all u’s neighbours along out-coming edges), else don’t touch the node

# Towards Approx. PPR

- A different way to look at PageRank: [Jeh&Widom. Scaling Personalized Web Search, ’02]

$$p_\beta(a) = (1-\beta)a + \beta p_\beta(M \cdot a)$$

  - $p_\beta(a)$ is the true PageRank vector with teleport param. β, and teleport vector a
  - $p_\beta(M \cdot a)$ is the PageRank vector with teleportation vector M·a, and teleportation param. β
    - where M is the stochastic PageRank transition matrix
    - Notice: M·a is one step of a random walk

# Towards Approx. PPR

- Proving $p_\beta(a) = (1-\beta)a + \beta p_\beta(M \cdot a)$
- Break the prob. into two cases:
  - Walks of length 0
  - Walks of length longer than 0
- The prob. of length 0 walk is **1-β**, and the walk ends where it started, with walker distribution **a**
- The prob. of walk length >0 is **β**, and the walk starts at distribution **a**, takes a step, (ends in distribution **Ma**), then takes the rest of the random walk to with distribution $p_\beta(M \cdot a)$
  - By memoryless nature of the walk: after we know the location of the 2nd step of the walk has distribution **Ma**, the rest of the walk can forget where it started and behave as if it started at **Ma**

# “Push” Operation

- Idea:
  - **r**: approx. PageRank, **q**: residual PageRank
  - Start with trivial approximation **r**=0 and **q**=a
  - Iteratively push PageRank from **q** to **r** until **q** is small
- Push: 1 step of a lazy random walk from node **u**:

**Push** (u, r, q):
    $r' = r$, $q' = q$
    $r'_u = r_u + (1-\beta)q_u$
    $q'_u = (1/2)\beta q_u$
    for each v such that u -> v:
        $q'_v = q_v + (1/2)\beta(q_u/d_u)$
    return r’, q’

push from q to r

update **r**
**Do 1 step of a walk**:
**stay** at **u** with prob. 1/2
**spread** remaining 1/2 fraction of $\mathbf{q_u}$ as if a single step of random walk were applied to **u**

# Intuition Behind Push Operation

- If $\mathbf{q_u}$ is large, this means that we have **underestimated** the importance of node **u**

- Then we want to take some of that residual ($\mathbf{q_u}$) and give it away, since we have too much of it

**Push** (u, r, q):
  $r' = r$, $q' = q$
  $r'_u = r_u + (1-\beta)q_u$
  $q'_u = (1/2)\beta q_u$
  for each v such that u -> v:
    $q'_v = q_v + (1/2)\beta(q_u/d_u)$
  return $r'$, $q'$

- So we keep $\mathbf{(1/2)\beta q_u}$ and then give away the rest to our neighbors to get rid of it corresponding to the spreading of $\mathbf{(1/2)\beta(q_u/d_u)}$ term

- Each node wants to keep giving away this excess PageRank until all nodes have no or a very small gap in excess PageRank

# Approx. PPR

r: PPR vector
$r_u$: PPR score of u
q: residual PPR vector
$q_u$: residual of node u
$d_u$: degree of node u

- ApproxPageRank(s, β, **ε**):

Set r = [0 .. 0], q = [0 .. 0 1 0 .. 0]

While $\max_{u \in V} q_u/d_u >= \boldsymbol{\varepsilon}$:

  Choose any vertex u where $q_u/d_u >= \boldsymbol{\varepsilon}$

  Push (u, r, q):

    $r' = r, q' = q$
    $r'_u = r_u + (1-\beta)q_u$
    $q'_u = (1/2)\beta q_u$
    for each v such that u -> v:
      $q'_v = q_v + (1/2)\beta(q_u/d_u)$
    $r = r', q = q'$

Return r

**ε** is small, compute PageRank of all nodes

**ε** is large, push operator only pushes locally

Update r: move (1-β) of the prob. from $q_u$ to $r_u$
1 step of a lazy random walk:
-Stay at $q_u$ with prob. 1/2
-Spread remaining 1/2β fraction of $q_u$ as if a single step of random walk were applied to u

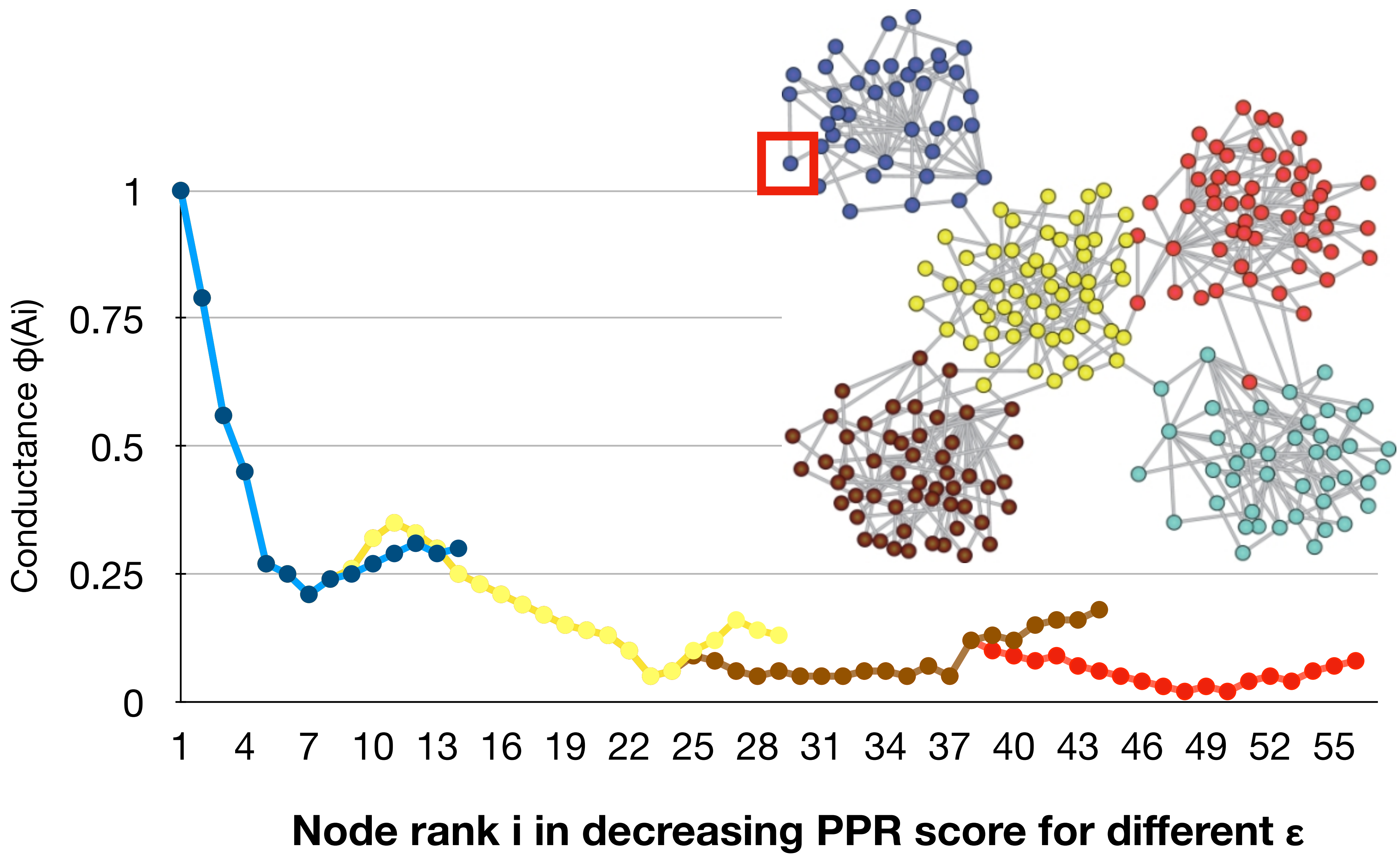
# Observations (1)

- Runtime:
  - PageRank-Nibble computes PPR in time $\left(\frac{1}{\epsilon(1-\beta)}\right)$ with residual error <= **ε**
    - Power method would take time $O\left(\frac{\log n}{\epsilon(1-\beta)}\right)$
- Graph cut approx. guarantee:
  - If there exists a cut of conductance ϕ and volume k then the method finds a cut of conductance $O(\sqrt{\phi \log k})$

Details in [Andersen, Chung, Lang, Local graph partitioning using PageRank vectors, ' 07]

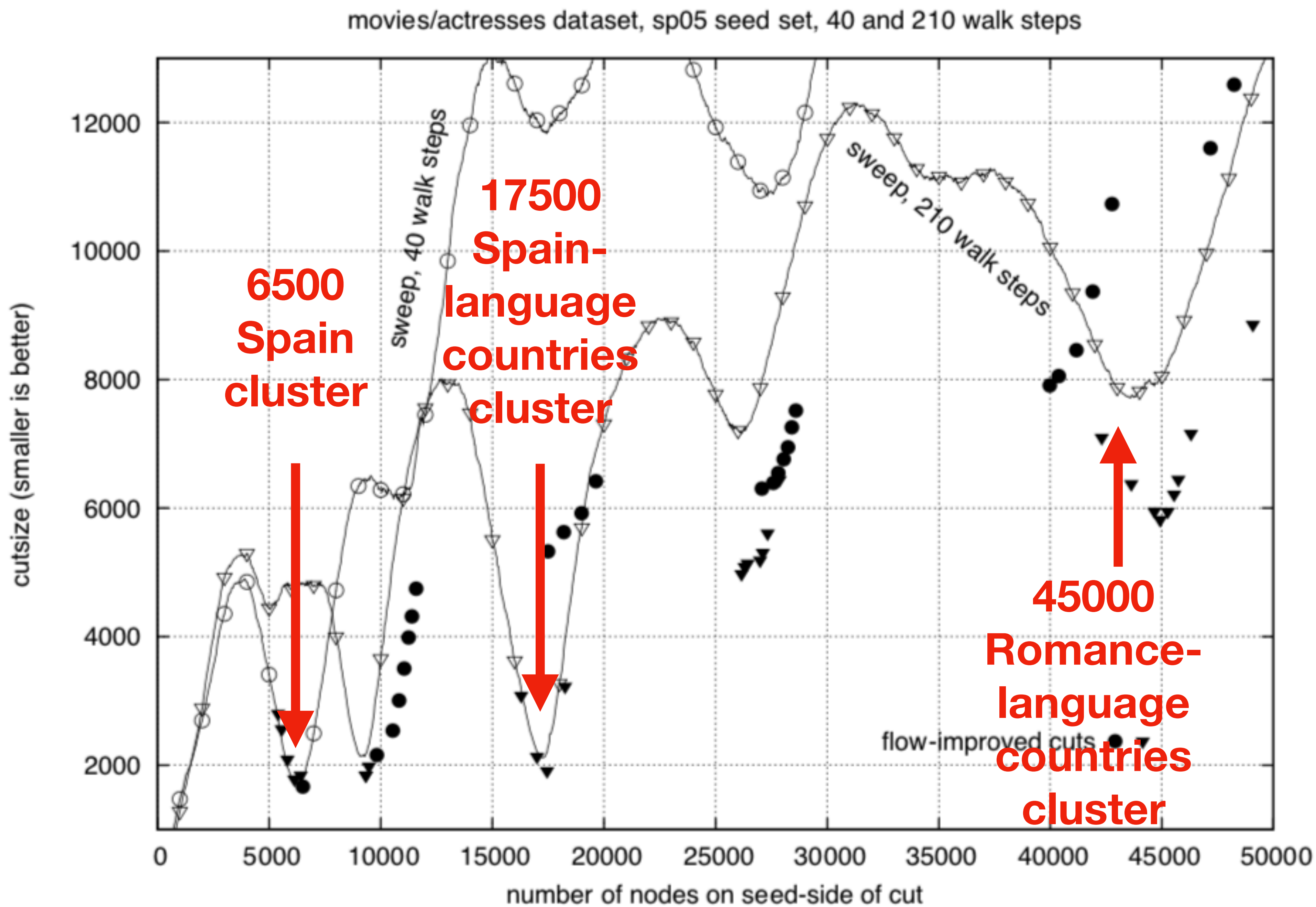
# Observations (2)

- The smaller the **ε** the farther the random walk will spread!

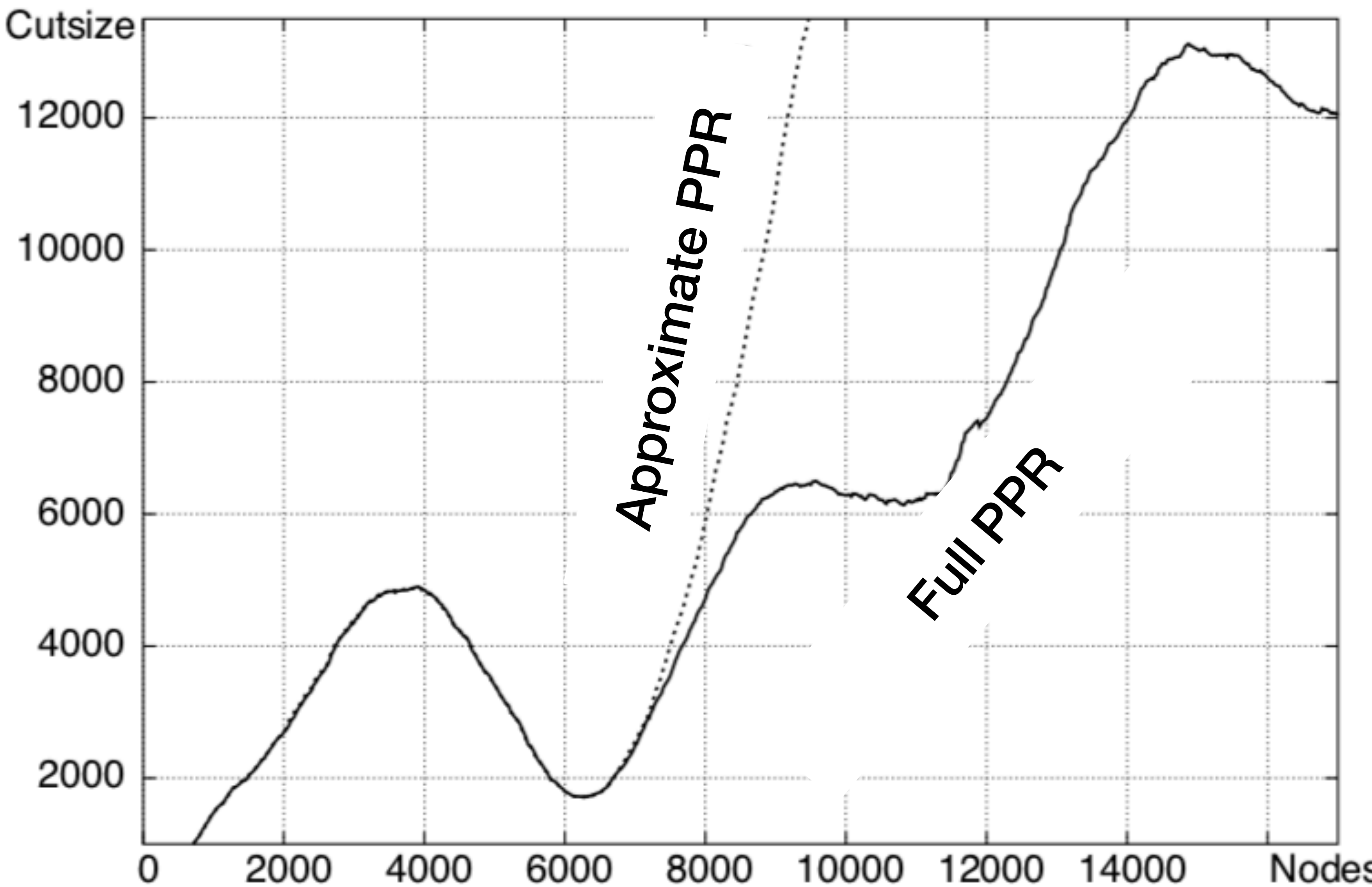


# Observations (2)

- Movie-actor example: Seed set contains 179 Spanish movie nodes, 0 actor nodes



# Observations (3)




It might seem confusing that our sweep plots show cutsize while the random walks algorithm optimizes conductance. However, conductance is a particular way of combining our twin goals of 1) growing the seed set and 2) finding a small boundary. These plots of cutsize as a function of node count display a range of possible tradeoffs between these two goals.

[Andersen, Lang: Communities from seed sets, 2006]

# Summary of Approx. PPR

- Alg. summary:
  - Pick a seed node **s** of interest
  - Run PPR with teleport set = {**s**}
  - Sort the nodes by the decreasing PPR score
  - **Sweep** over the nodes and find good clusters

# Outline

- Motivation
- PageRank based Clustering
- Modularity Maximization

# Network Communities

- Communities: sets of tightly connected nodes
- Define: **Modularity** Q
  - A measure of how well a network is partitioned into communities
  - Given a partitioning of the network into groups **s** ∈ **S**:

$$Q \propto \Sigma_{\mathbf{s} \in \mathbf{S}} \, [\text{(\# edges within group s)} - \text{(expected \# edges within groups s)}]$$

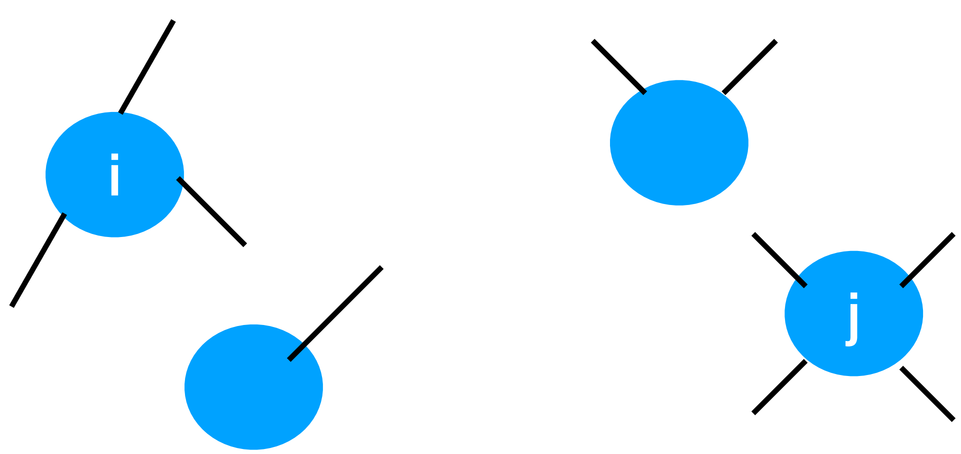
**need a null model**

# Null Model: Configuration Model

- Given real G on n nodes and m edges, construct rewired network G’
  - Same degree distribution but random connections
  - Consider G’ as a **multigraph** (allow multiple edges between a pair of nodes)
  - The expected number of edges between nodes i and j of degrees $k_i$ and $k_j$ equals to: $\mathbf{k_i \cdot k_j / (2m)}$

node i has $k_i$ spokes with each connected to a random spoke

total number of edges in the graph




**each node keeps the same number of edges, but now edges are randomly connected**

# Modularity

- Modularity of partitioning S of Graph G:
  - Q ∝ $\Sigma_{s \in S}$ [(# edges within group s) - (expected # edges within groups s)

$A_{ij}$ = 1 if node i is connected with node j

$$Q(G,S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left( A_{ij} - \frac{k_i k_j}{2m} \right)$$

**normalizing const:**
**-1 < Q < 1**

**how many edges inside s**

- Modularity values take range [-1, 1]
  - It is positive if the number of edges within groups exceeds the expected number
  - Q greater than **0.3-0.7** means significant community structure

# Modularity: 2 Defs

$$Q(G,S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left( A_{ij} - \frac{k_i k_j}{2m} \right)$$

- Equivalently modularity can be written as:

**δ=1 if node i and j belong to the same community**

$$Q(G,S) = \frac{1}{2m} \sum_{ij} \left( A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

**community of node i**

- Idea: We identify communities by **maximizing modularity**

# Louvain Method

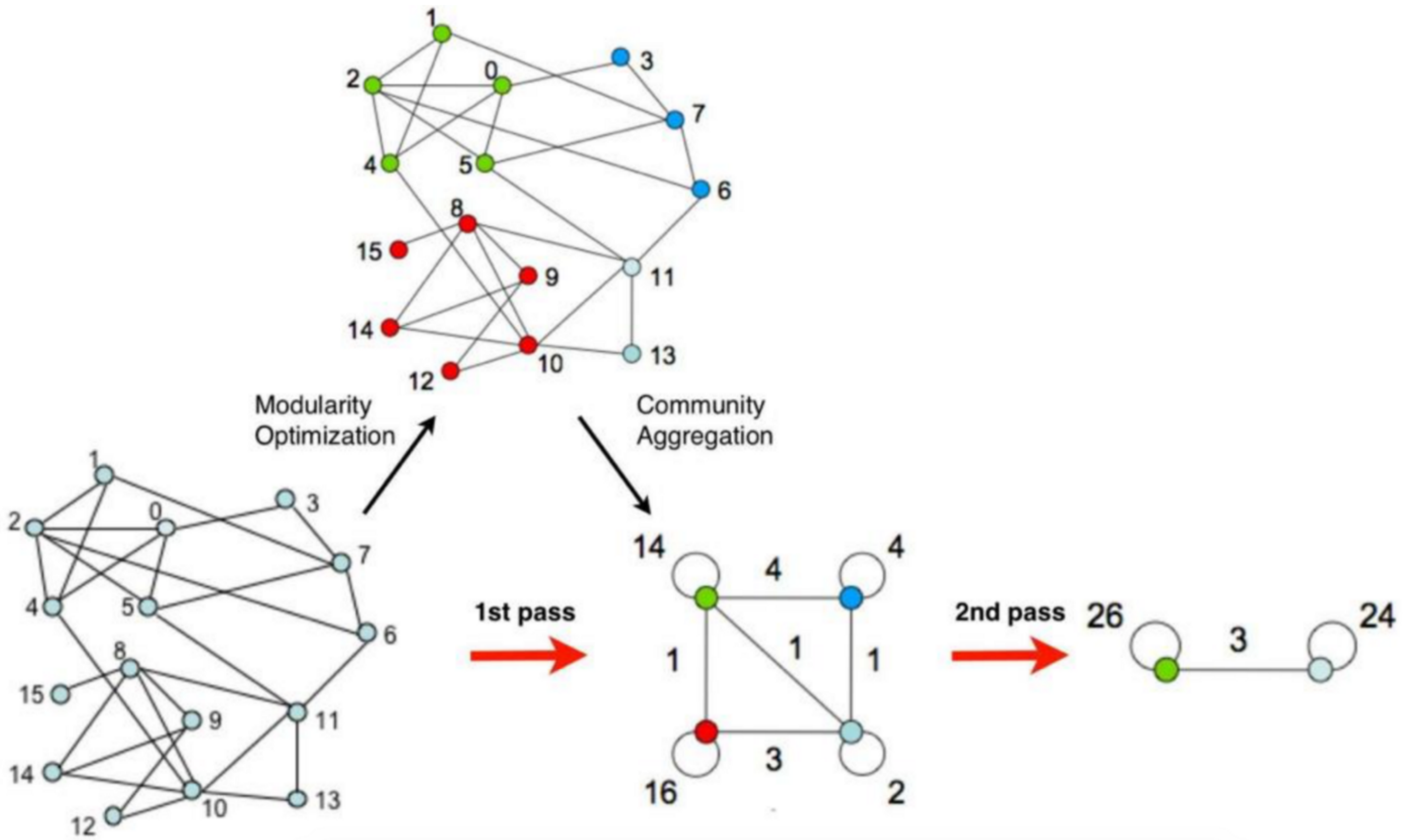
- **Greedy** alg. for community detection
  - $O(n \log n)$ run time
- Supports weighted graphs
- Provides hierarchical partitions
- Widely utilized to study **large** networks because:
  - Fast
  - Rapid convergence properties
  - High modularity output (i.e., “better communities”)

[Fast unfolding of communities in large networks, Blondel et al. 2008]

# Louvain Alg.: Outline

- Louvain alg. greedily maximizes modularity
- Each pass is made of 2 phases:
  - Phase 1: Modularity is optimized by allowing only local changes of communities
  - Phase 2: The identified communities are aggregated in order to build a new network of communities
  - Goto Phase 1

The passes are repeated iteratively until no increase of modularity is possible!



# Phase 1 Partitioning

- Put each node in a graph into a distinct community (one node per community)
- For each node i, the alg. performs two calculations:
  - Compute the modularity gain (ΔQ) when putting node i from its current community into the community of some neighbour j of i
  - Move i to a community that yields the **largest** modularity gain (ΔQ)
- The loop runs until no movement yields a gain

The 1st phase stops when a local maxima of the modularity is attained, i.e., when no individual move can improve the modularity.
The modularity depends on the order you visit the node, but does not matter.

# Modularity Gain

- What is ΔQ if we move node i to community C?

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

where:

$\Sigma in$ the sum of the weights of the links <u>inside</u> $C$

$\Sigma tot$ the sum of the weights of <u>all links to</u> nodes in $C$

$ki$ the sum of the weights (i.e., degree) of all links to node $i$

$ki,in$ the sum of the weights of links from node $i$ to nodes in $C$

$m$ is the sum of the weights of all edges in the graph

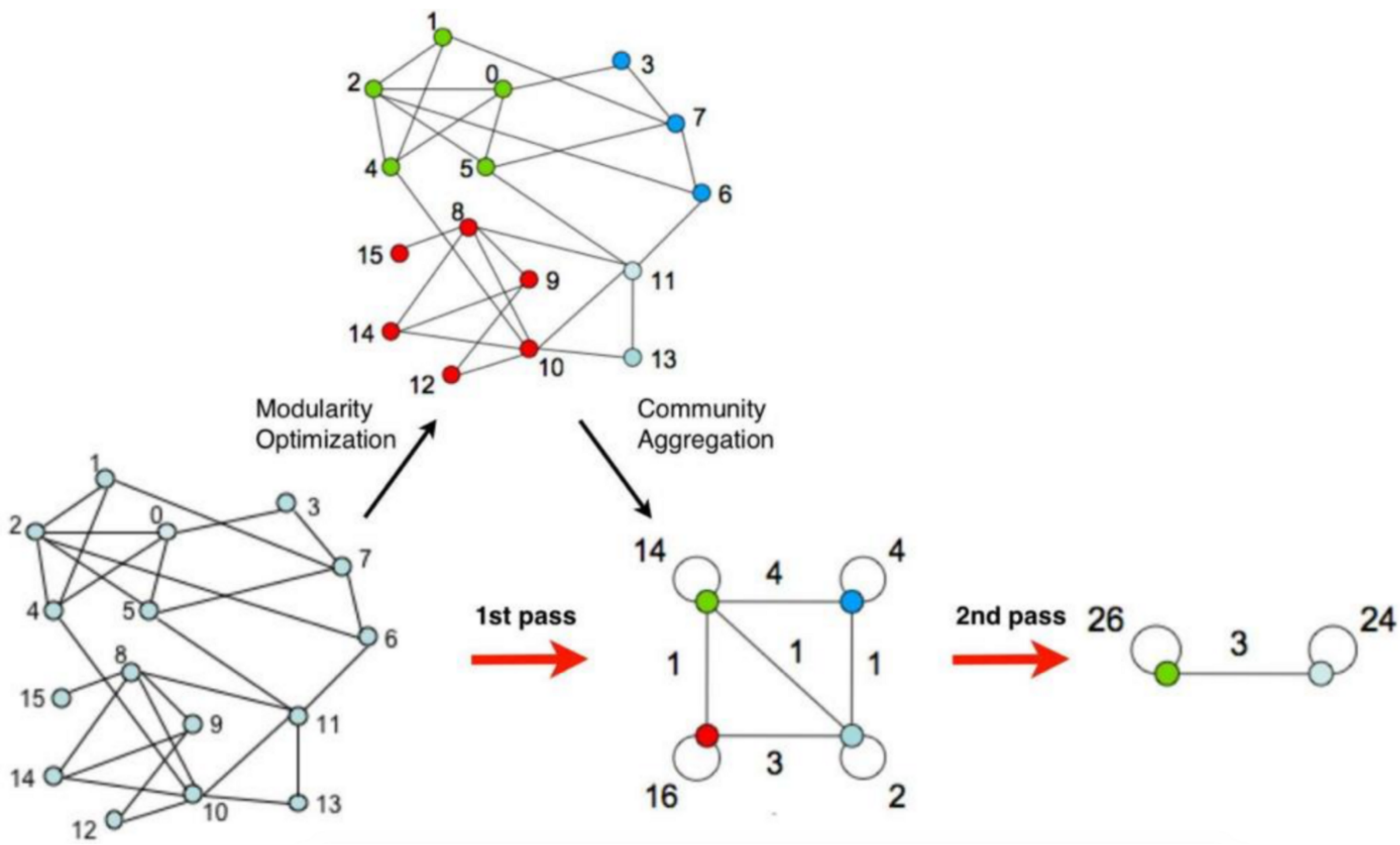
- Also need to compute ΔQ(D →i) of taking node i out of community D

- ΔQ = ΔQ(i →C) + ΔQ(D →i)

# Phase 2 Restructuring

- The partitions obtained in the 1st phase are contracted into **super-nodes**, and the weighted network is created as follows:
  - Super-nodes are connected if there is at least one edge between nodes of the corresponding communities
  - The **weight** of the edge between the two super-nodes is the sum of the weights from all edges between their corresponding partitions
- The loop runs until the community configuration does not change anymore

# Louvain Alg.



Modularity
Optimization
Community
Aggregation
1st pass
2nd pass
14
4
16
2
3
1
26
24

# Reading

- Papers cited in the slides.