Announcements:

- Course project –TAs will reach out with feedback (if not already)
  - Course grade expectations
- Project Milestone due Thu
- Watch out for homework feedback poll – Please participate ☺

Community Detection in Graphs

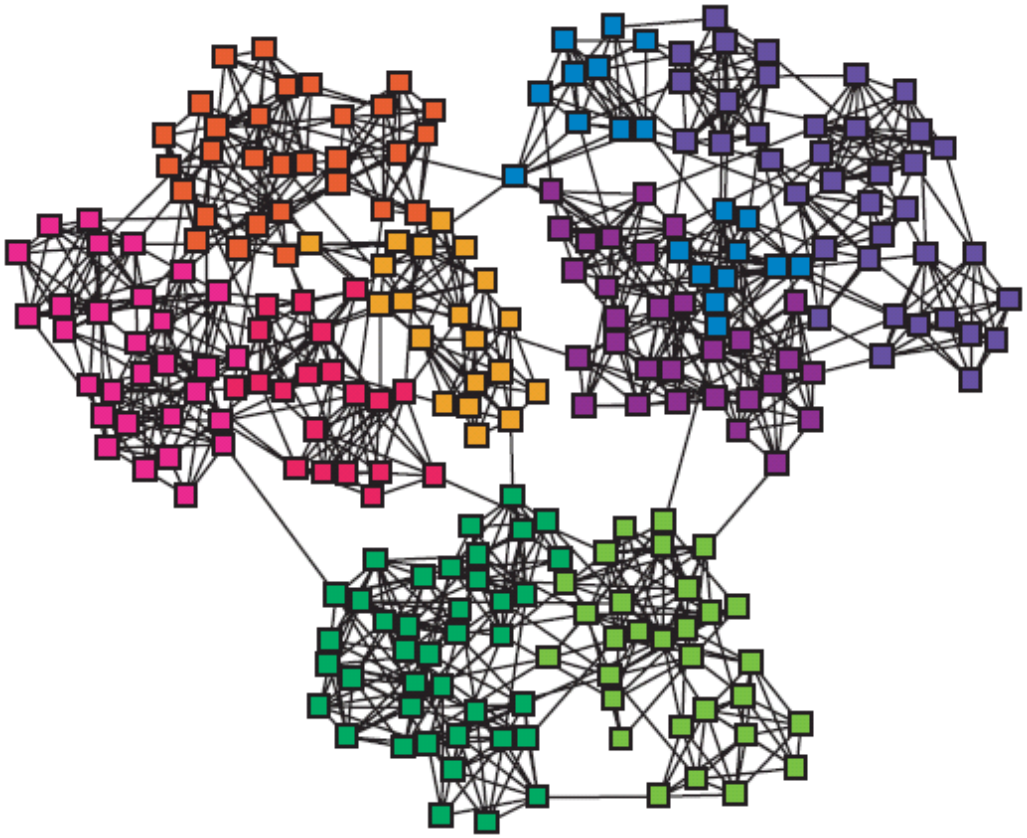
CS547 Machine Learning for Big Data

Tim Althoff

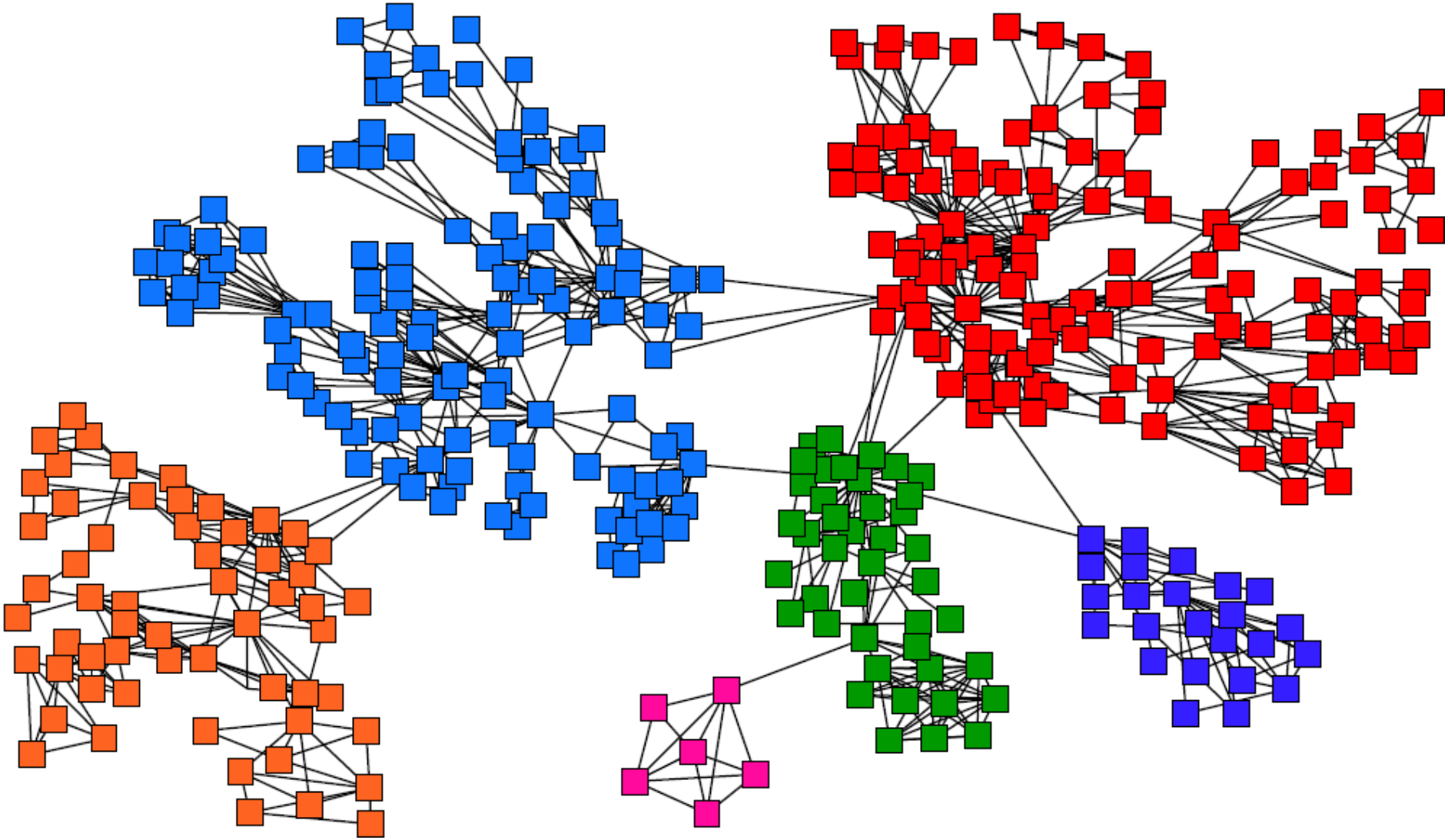
PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Networks & Communities

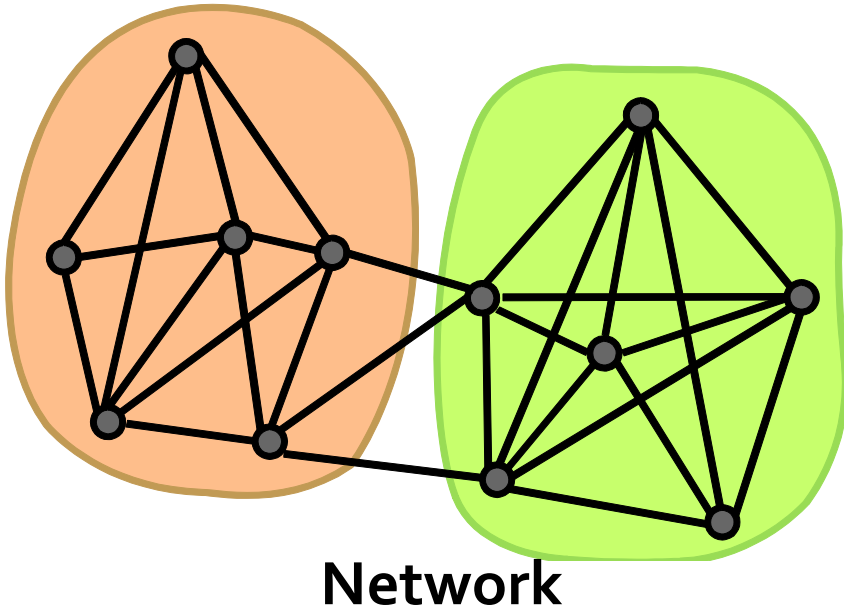
- **We often think of networks being organized into modules, clusters, communities:**



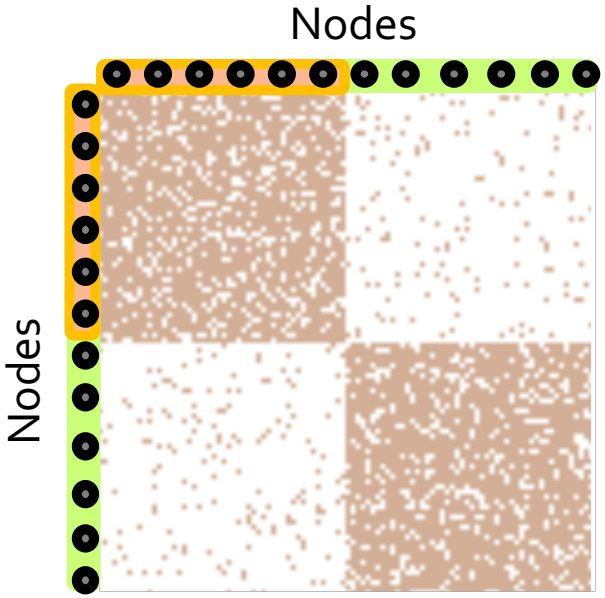
Goal: Find Densely Linked Clusters



Non-overlapping Clusters



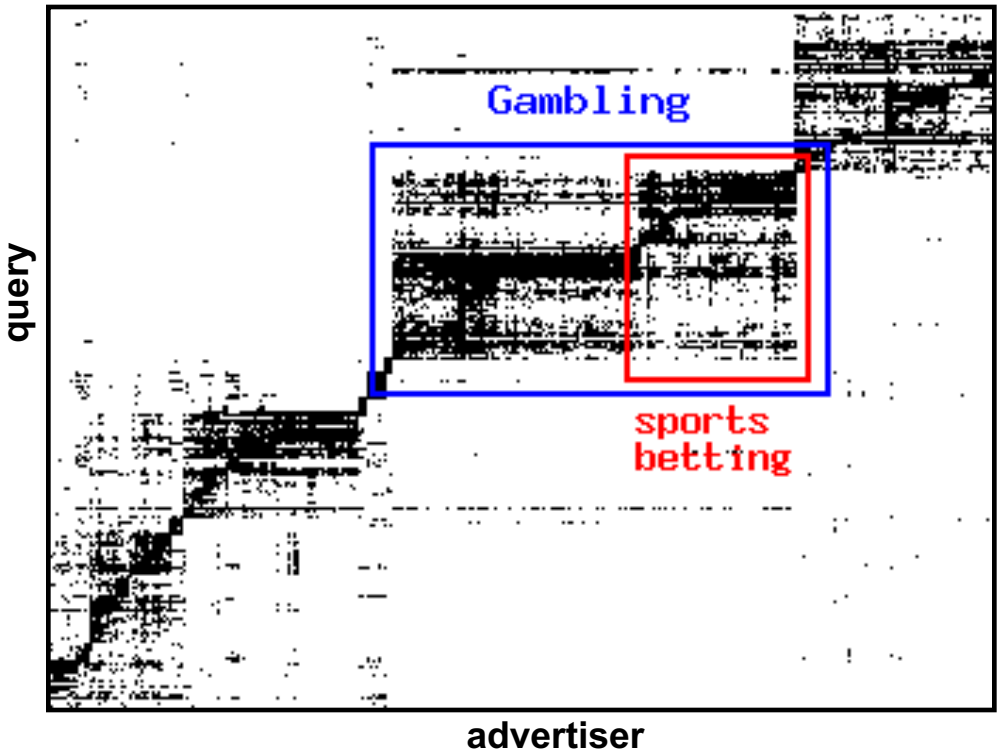
Network



Adjacency matrix

Micro-Markets in Sponsored Search

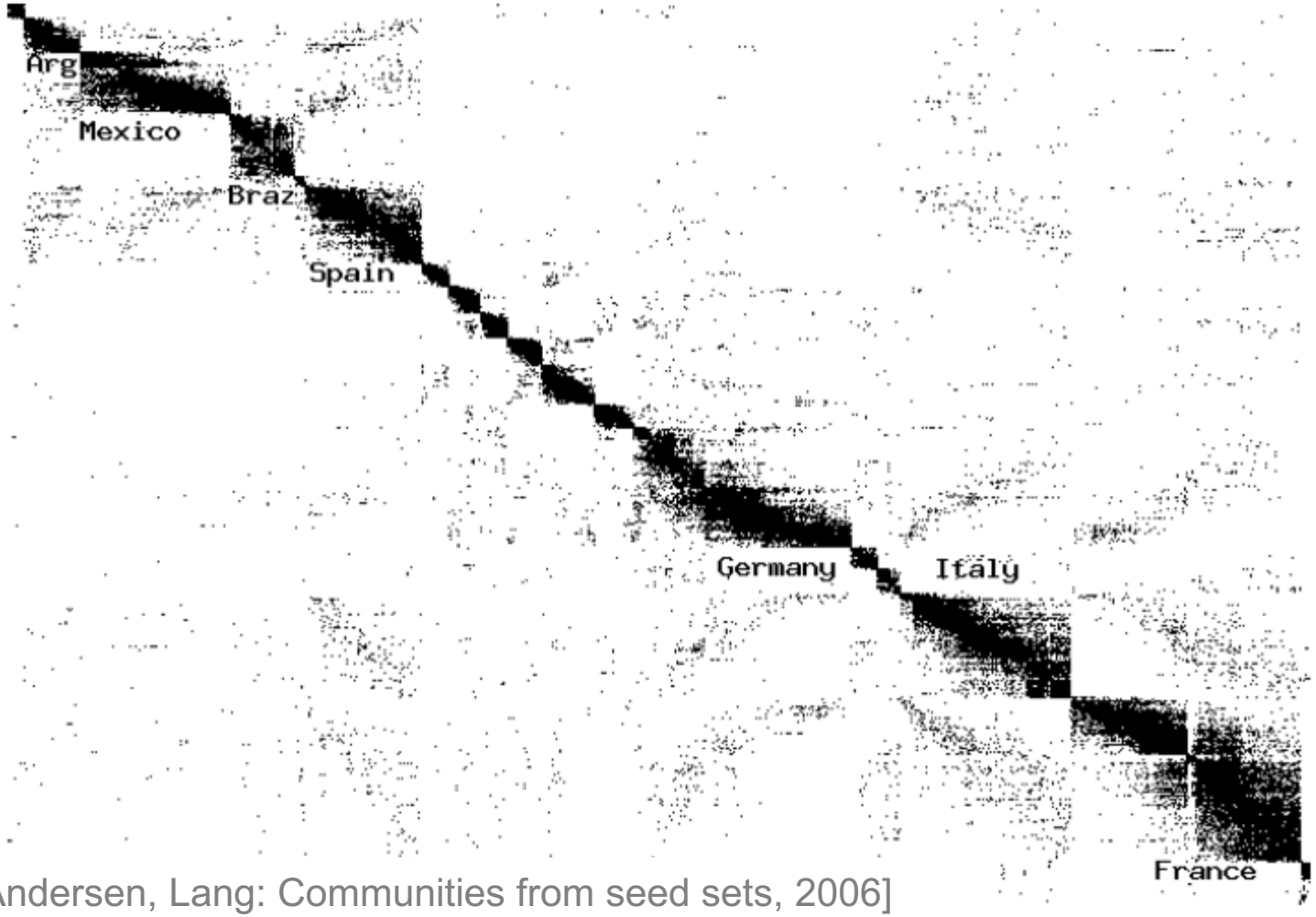
- **Find micro-markets by partitioning the query-to-advertiser graph:**



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

Movies and Actors

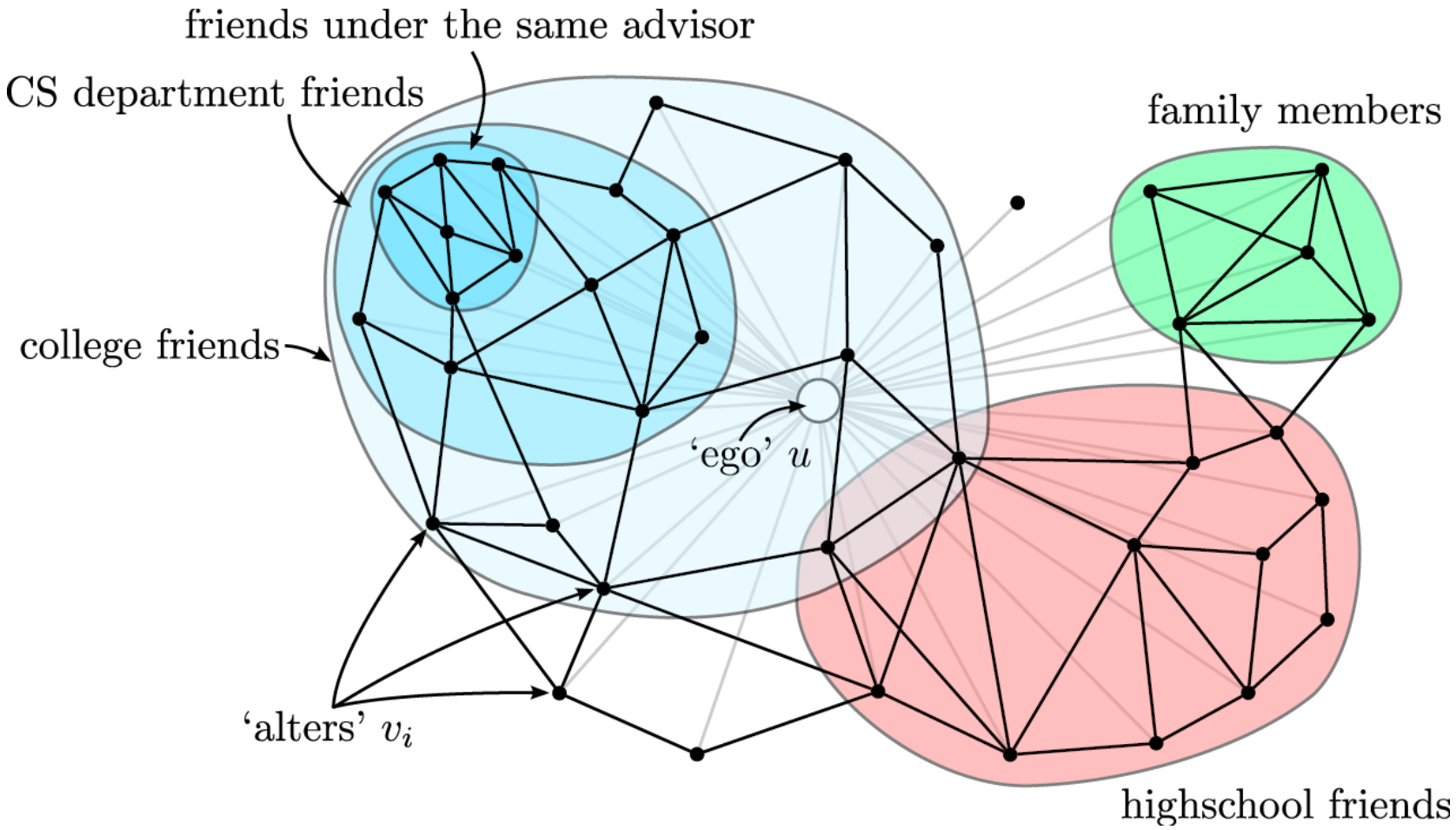
- **Clusters in Movies-to-Actors graph:**



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

Twitter & Facebook

- **Discovering social circles, circles of trust:**



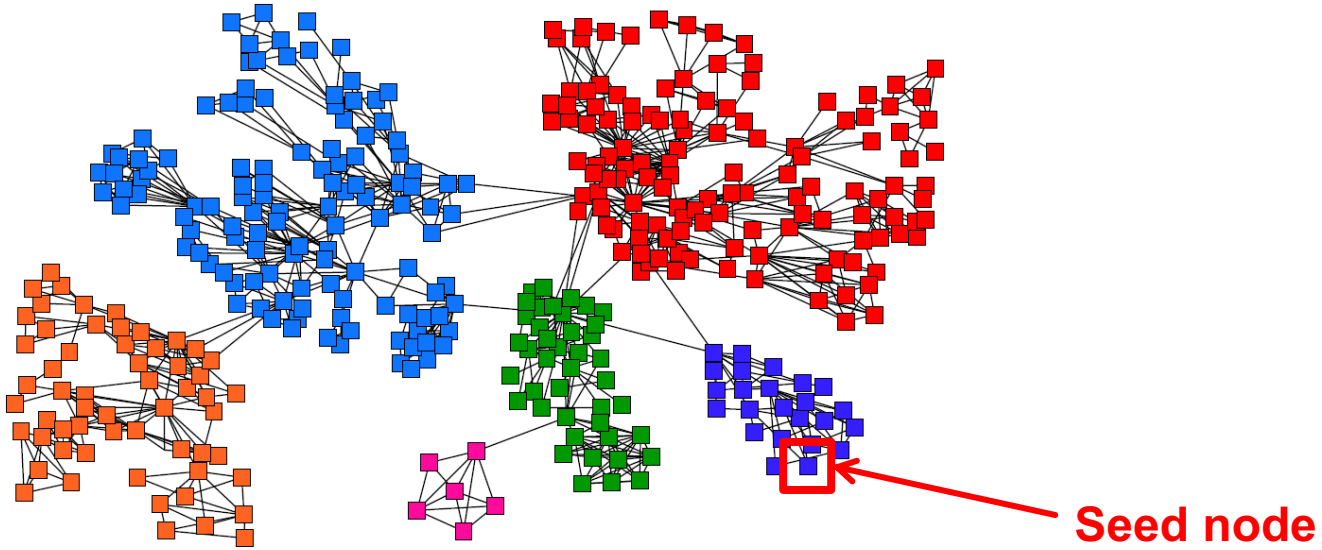
[McAuley, Leskovec: Discovering social circles in ego networks, 2012]

The Setting

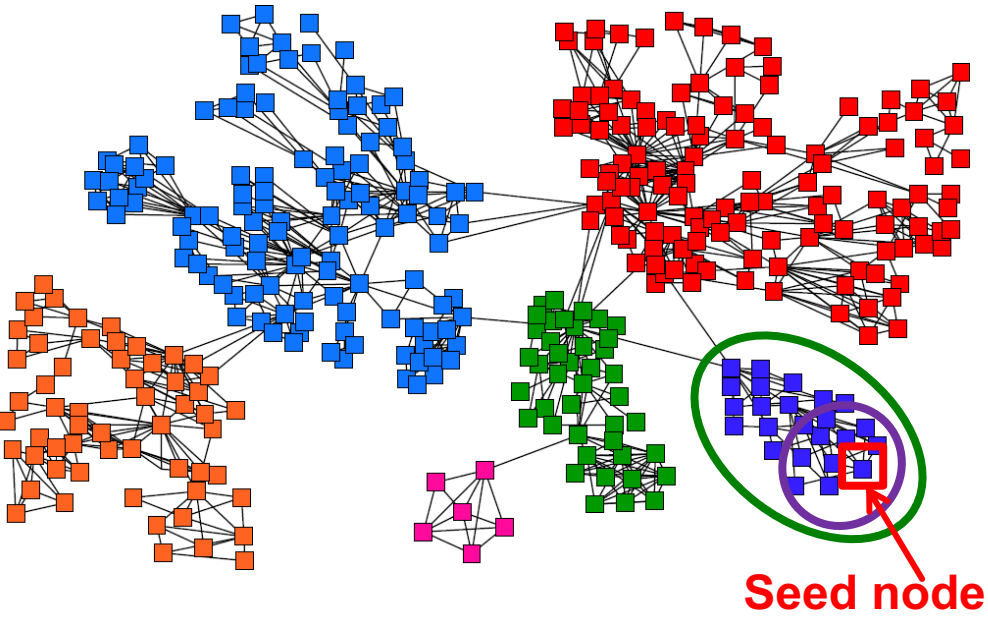
- **Graph is large**
  - **Assume the graph fits in main memory**
    - For example, to work with a 200M node and 2B edge graph one needs approx. 16GB RAM
  - But the graph is too big for running anything more than linear time algorithms
- **We will cover a PageRank based algorithm for finding dense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

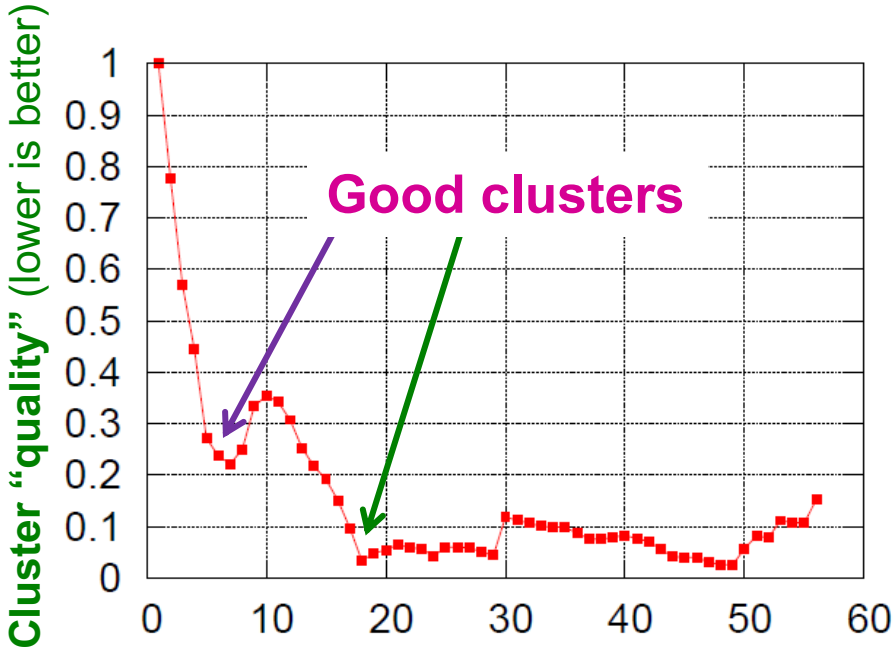
Idea: Seed Nodes

- **Discovering clusters based on seed nodes**
  - **Given:** Seed node **s**
  - Compute (approximate) **P**ersonalized **P**age**R**ank (**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



Seed Node: Intuition

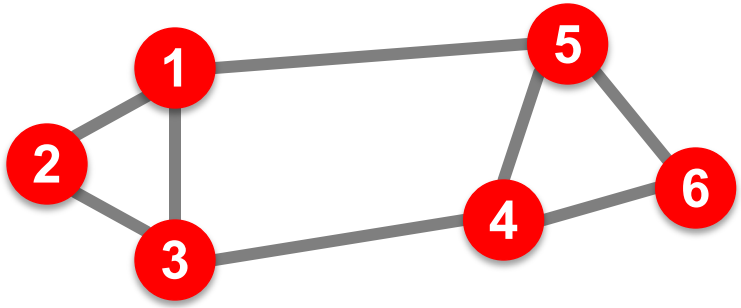




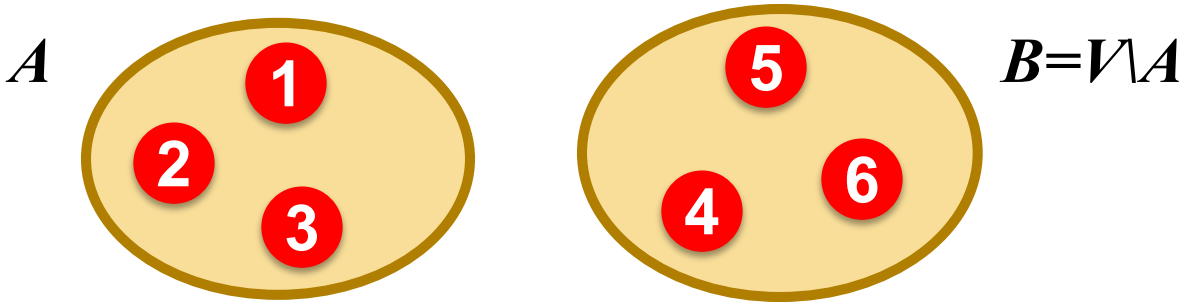
- **Algorithm outline:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** with teleport set = {***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find **good clusters**

What makes a good cluster?

- **Undirected graph $G(V, E)$:**



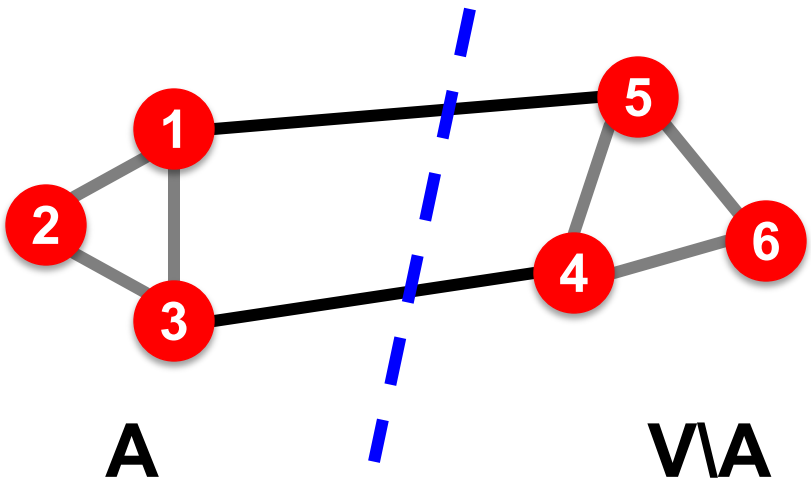
- **Partitioning task:**
  - Divide vertices into 2 disjoint groups $A, B = V \setminus A$



- **Question:**
  - **How can we define a "good" cluster in G?**

What makes a good cluster?

- **What makes a good cluster?**
  - **Maximize the number of within-cluster connections**
  - **Minimize the number of between-cluster connections**

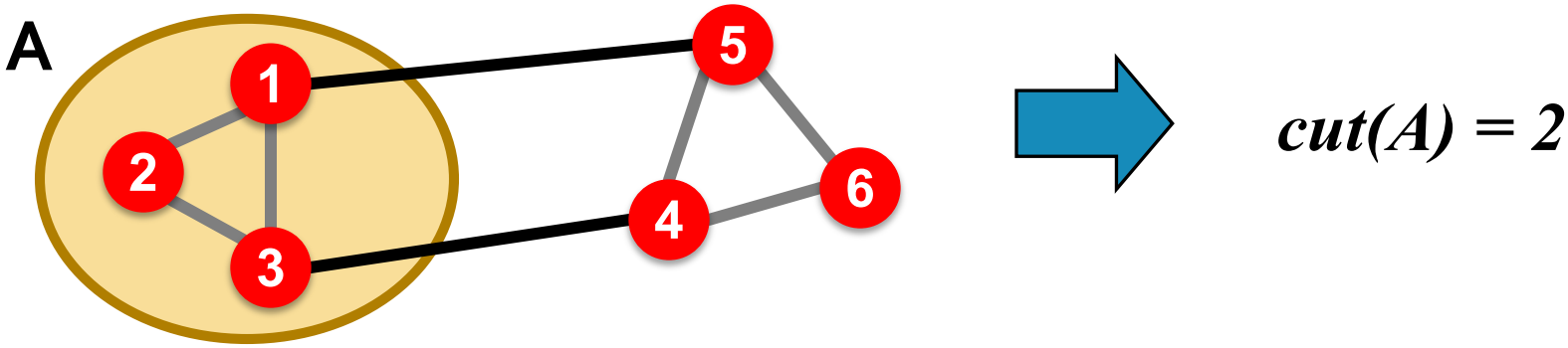


Graph Cuts

- **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:

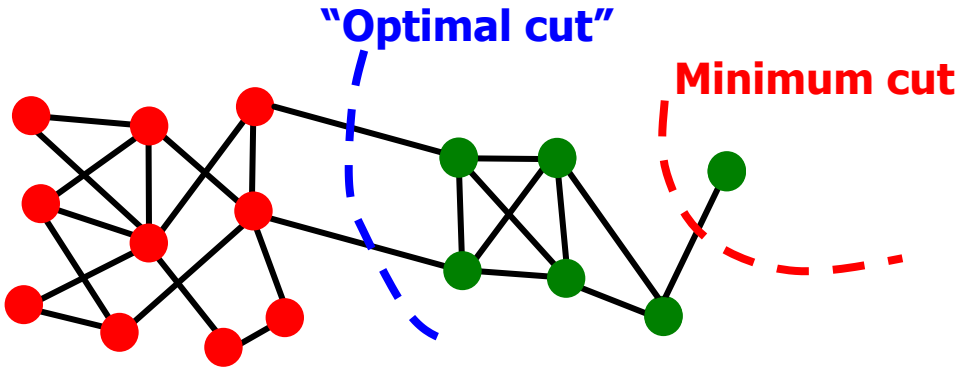
$$cut(A) = \sum_{i \in A, j \notin A} w_{ij}$$

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



Cut 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

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

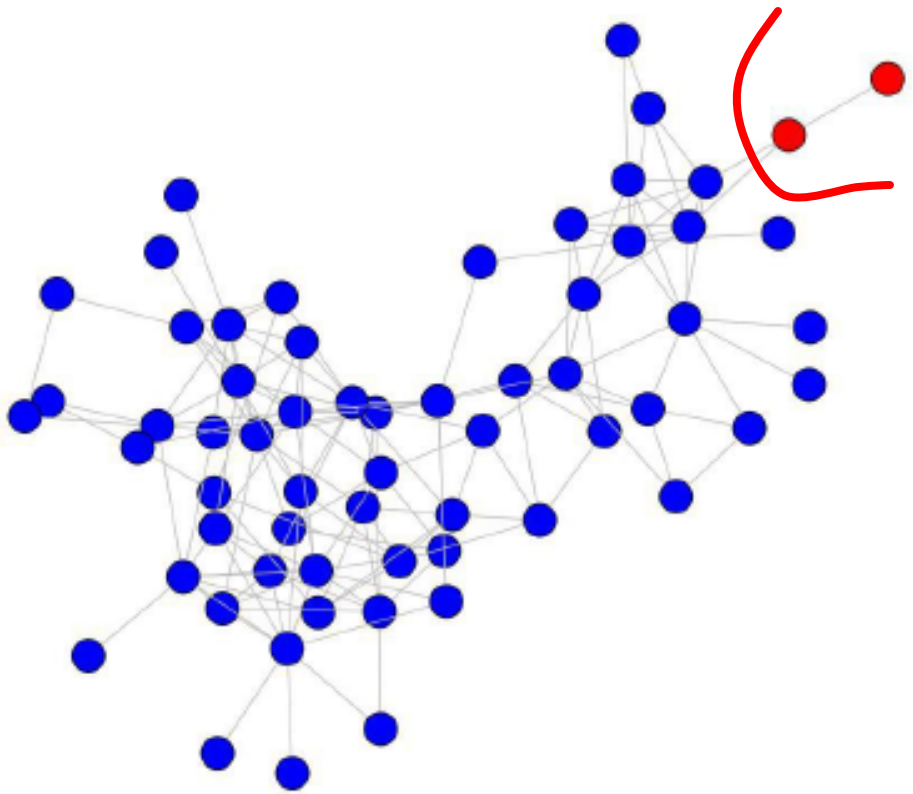
$\boldsymbol{vol(A)}$: total weight of the edges with at least one endpoint in $\boldsymbol{A}$: $\mathbf{vol}(\boldsymbol{A}) = \sum_{i \in A} \boldsymbol{d_i}$

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

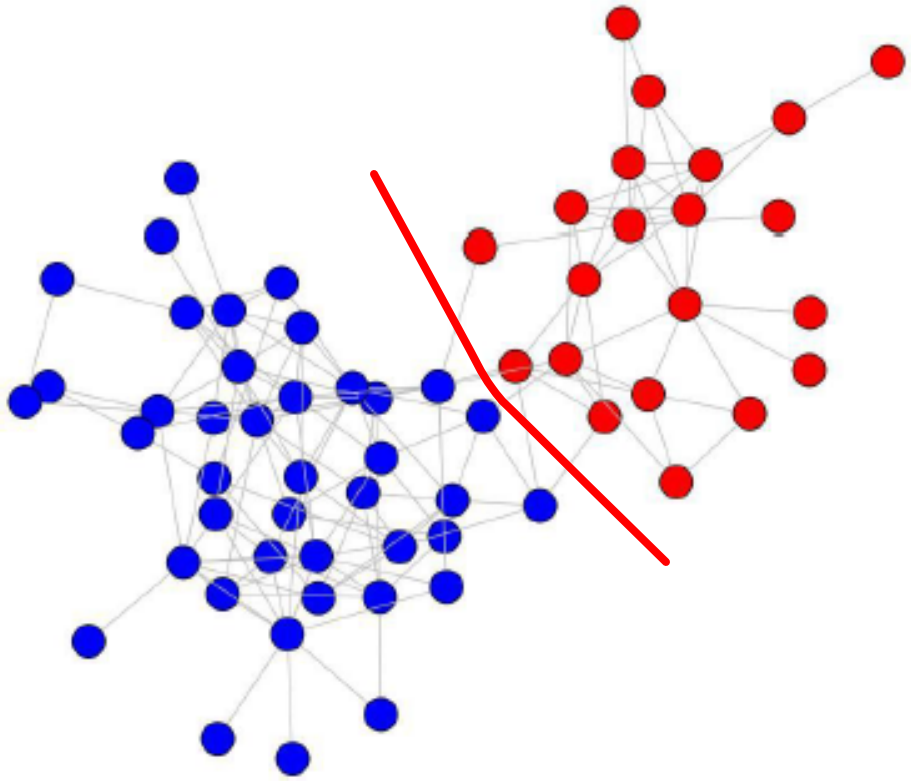
- **Why use this criterion?**
  - Produces more balanced partitions

m… number of edges of the graph
$\mathbf{d_i}$… degree of node ***I***
E…edge set of the graph

Example: Conductance Score



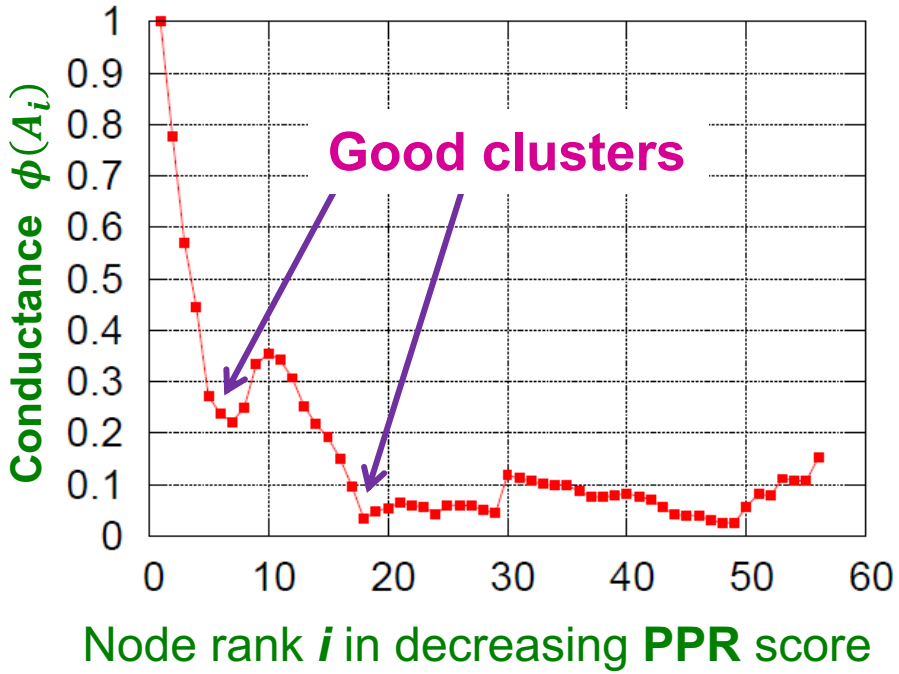
$\phi = 2/4 = 0.5$



$\phi = 6/92 = 0.065$

Algorithm Outline: Sweep

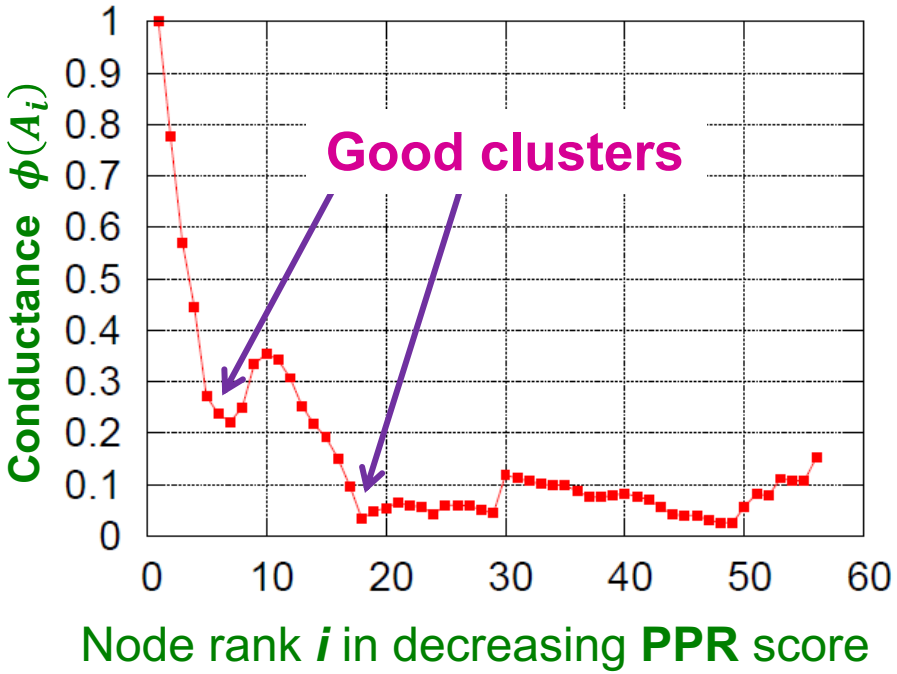
- **Algorithm outline:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** w/ teleport={***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find good clusters



- **Sweep:**
  - Sort nodes in decreasing PPR score $r_1 > r_2 > \cdots > r_n$
  - For each $\boldsymbol{i}$ compute $\boldsymbol{\phi(A_i = \{r_1, \ldots r_i\})}$
  - **Local minima** of $\boldsymbol{\phi(A_i)}$ 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 A_i
  - To compute $\boldsymbol{\phi(A_{i+1})} = Cut(A_{i+1})/Vol(A_{i+1})$
    - $Vol(A_{i+1}) = Vol(A_i) + d_{i+1}$
    - $Cut(A_{i+1}) = Cut(A_i) + d_{i+1} - 2\#(edges\ of\ u_{i+1}\ to\ A_i)$



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:
    $\mathbf{r}^{(t+1)} = \beta \mathbf{M} \cdot \mathbf{r}^{(t)} + (1-\beta)\boldsymbol{a}$
    At index **s**
    - $\boldsymbol{a}$ is a teleport vector: $\boldsymbol{a} = [0 \ \ldots 0 \ 1 \ 0 \ \ldots 0]^T$
    - $\boldsymbol{r}$ 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)**
    - **s** … seed node
    - **β** … teleportation parameter
    - **ε** … approximation error parameter

Approximate PPR: Overview

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

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

d_i… degree of i

  - Keep track of **residual PPR score $q_u = p_u - r_u^{(t)}$**
    - Residual tells us how well PPR score p_u of $\boldsymbol{u}$ is approximated
    - $\boldsymbol{p_u}$… is the "true" PageRank of node $\boldsymbol{u}$
    - $\boldsymbol{r_u^{(t)}}$… is PageRank estimate of node u at around $\boldsymbol{t}$

    If **residual q_u** of node $\boldsymbol{u}$ is too big $\frac{q_u}{d_u} \geq \varepsilon$ then **push the walk further** (distribute some of residual q_u to all u's neighbors along out-coming edges), else don't touch the node

Towards approximate PPR

- **A different way to look at PageRank:**
  [Jeh&Widom. *Scaling Personalized Web Search*, 2002]

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

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

Towards approximate PPR

- Proving: $\boldsymbol{p_\beta(a) = (1-\beta)a + \beta\, p_\beta(M \cdot a)}$
  - We can break this probability into two cases:
    - Walks of length 0, and
    - Walks of length longer than 0
  - The probability of length 0 walk is $\boldsymbol{1-\beta}$, and the walk ends where it started, with walker distribution $\boldsymbol{a}$
  - The probability of walk length >0 is $\boldsymbol{\beta}$, and then the walk starts at distribution $\boldsymbol{a}$, takes a step, (so it has distribution $\boldsymbol{Ma}$), then takes the rest of the random walk to with distribution $\boldsymbol{p_\beta(Ma)}$
    - Note that we used the memoryless nature of the walk: After we know the location of the second step of the walk has distribution $\boldsymbol{Ma}$, the rest of the walk can forget where it started and behave as if it started at $\boldsymbol{Ma}$. This proves the equation.

"Push" Operation

residual PPR score $q_u = p_u - r_u$

- **Idea:**
  - r… approx. PageRank, q… its 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:**

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

Update **r**
Do 1 step of a walk:
Stay at **u** with prob. ½
Spread remaining ½ fraction of $\mathbf{q_u}$ as if a single step of random walk were applied to **u**

Intuition Behind Push Operation

- If q_u is large, this means that we have underestimated the importance of node u
- Then we want to take some of that residual (q_u) and give it away, since we know that we have too much of it

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

- So, we keep $\frac{1}{2}\beta q_u$ and then give away the rest to our neighbors, so that we can get rid of it
  - This correspond to the spreading of $\frac{1}{2}\beta\ q_u/d_u$ term
- Each node wants to keep giving away this excess PageRank until all nodes have no or a very small gap in excess PageRank

Approximate PPR

- **ApproxPageRank(S, β, ε):**

  Set $\boldsymbol{r} = \vec{0}, \boldsymbol{q} = [0\,..\,0\;1\;0\;...\;0]$ ← At index **S**

  While $\max_{u \in V} \frac{q_u}{d_u} \geq \varepsilon$:

    Choose any vertex $\boldsymbol{u}$ where $\frac{q_u}{d_u} \geq \varepsilon$

    $\boldsymbol{Push(u, r, q)}$:

      $\boldsymbol{r' = r,\ q' = q}$

      $\boldsymbol{r'_u = r_u + (1-\beta) q_u}$

      $\boldsymbol{q'_u = \frac{1}{2}\beta q_u}$

      For each v such that $u \to v$:

        $\boldsymbol{q'_v = q_v + \frac{1}{2}\beta q_u / d_u}$

      $\boldsymbol{r = r',\ q = q'}$

  Return r

r … **PPR** vector
$\mathbf{r_u}$ …**PPR** score of **u**
q …residual **PPR** vector
$\mathbf{q_u}$ … residual of node **u**
$\mathbf{d_u}$ … degree of **u**

Update **r:** Move $(1-\beta)$ of the prob. from $\mathbf{q_u}$ to $\mathbf{r_u}$

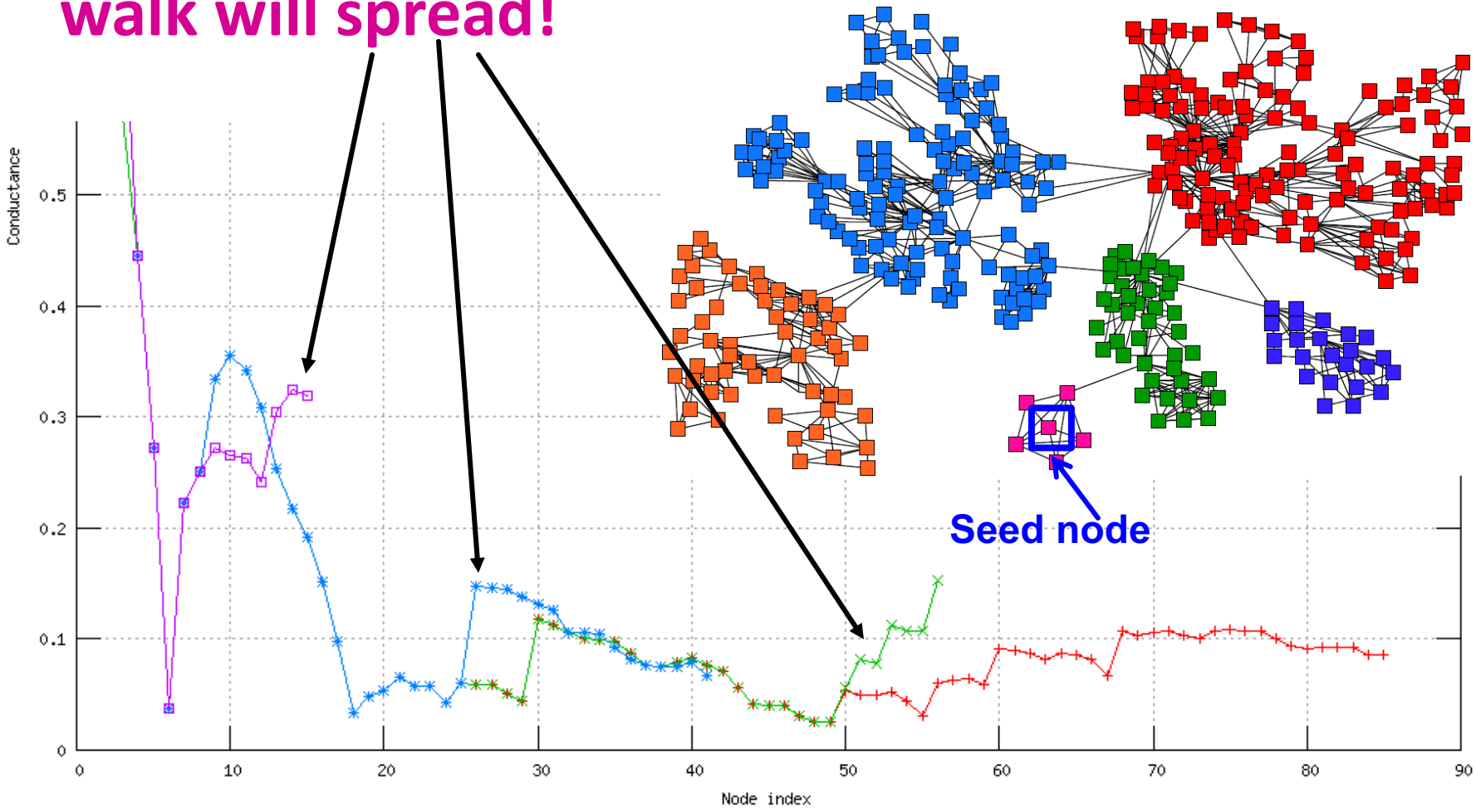
1 step of a lazy random walk:
- Stay at $\mathbf{q_u}$ with prob. ½
- Spread remaining ½ $\boldsymbol{\beta}$ fraction of $\mathbf{q_u}$ as if a single step of random walk were applied to **u**

Observations (1)

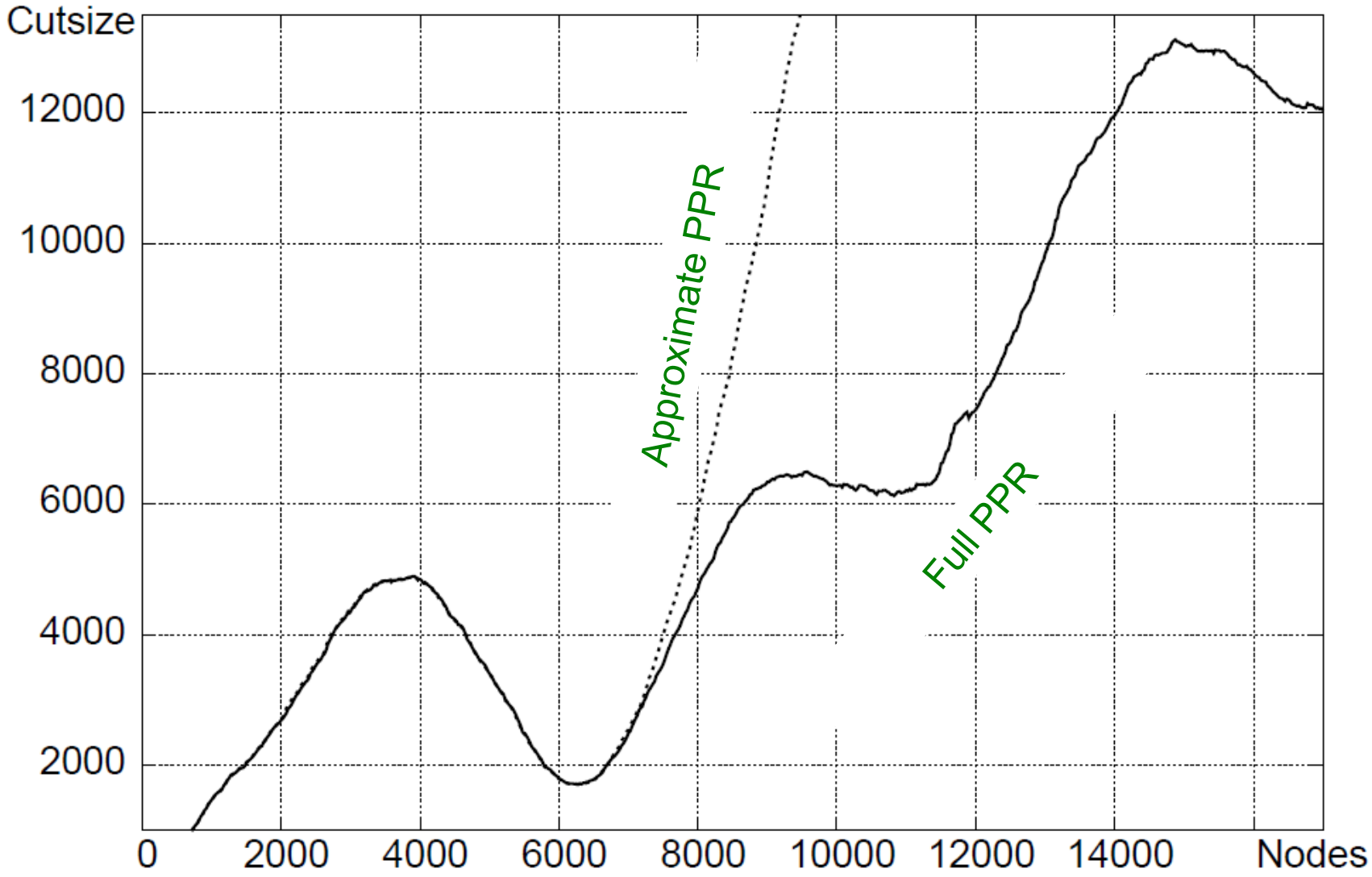
- **Runtime:**
  - PageRank-Nibble computes PPR in time $O\left(\frac{1}{\varepsilon(1-\beta)}\right)$ with residual error $\leq \varepsilon$
    - Power method would take time $O(\frac{\log n}{\varepsilon(1-\beta)})$
- **Graph cut approximation guarantee:**
  - If there exists a cut of conductance ϕ 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*, 2007]
    http://www.math.ucsd.edu/~fan/wp/localpartfull.pdf

Observations (2)

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

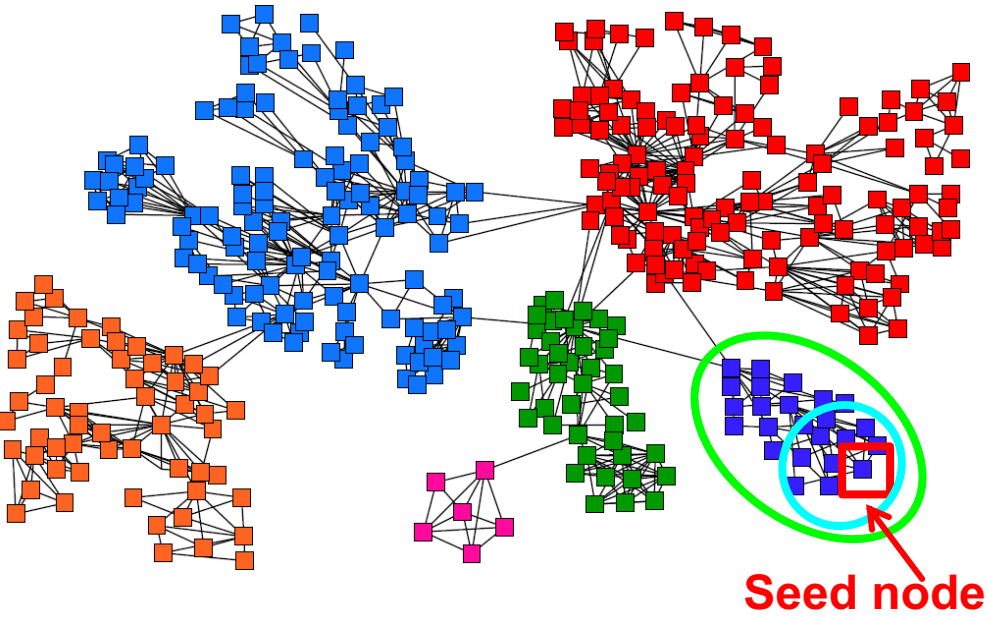


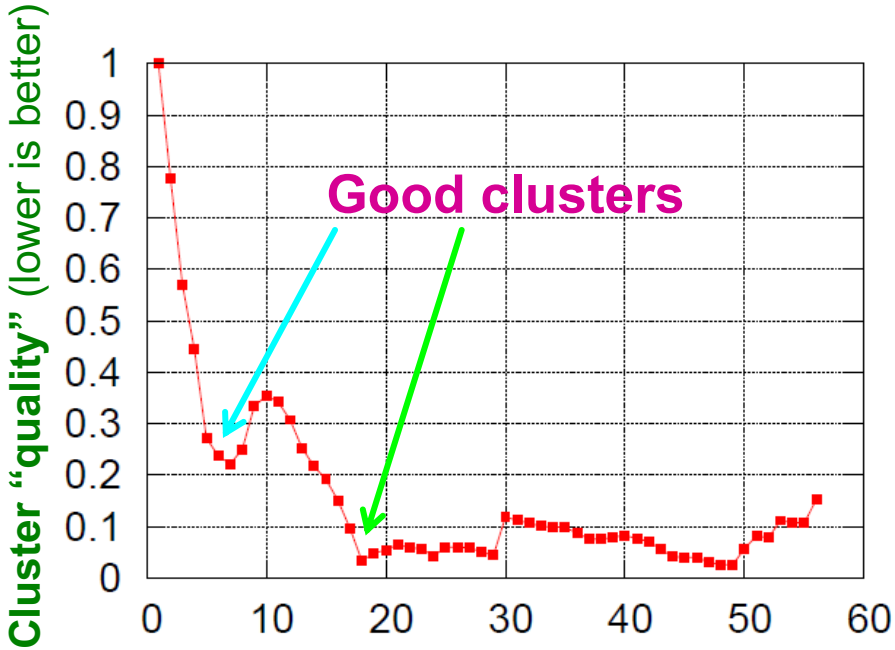
Observations (3)



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

Summary of Approx PPR Alg.



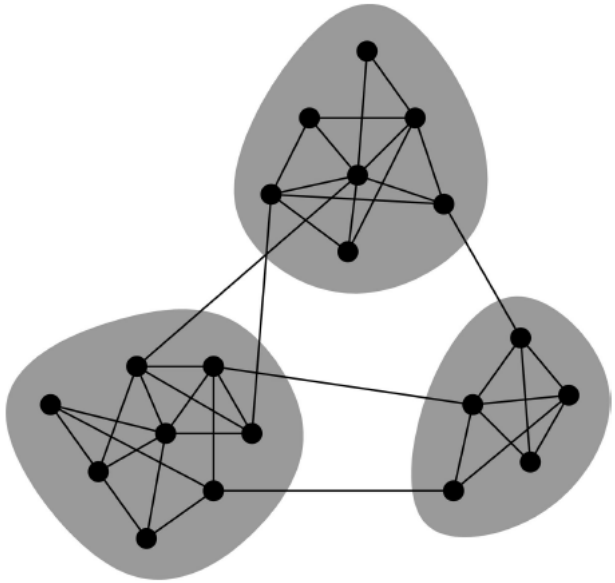


- **Algorithm summary:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** with teleport set = {***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find good clusters

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 $s \in S$:

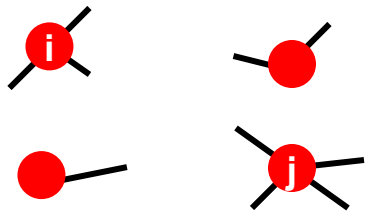


$$Q \propto \sum_{s \in S} [\text{(\# edges within group } s) - \text{(expected \# edges within group } s)]$$

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**



  - **The expected number of edges between nodes i and j** of degrees k_i and k_j equals to: $k_i \cdot \frac{k_j}{2m} = \frac{k_i k_j}{2m}$
    - The expected number of edges in (multigraph) **G'**:
      - $= \frac{1}{2}\sum_{i \in N}\sum_{j \in N}\frac{k_i k_j}{2m} = \frac{1}{2}\cdot\frac{1}{2m}\sum_{i \in N} k_i\left(\sum_{j \in N} k_j\right) =$
      - $= \frac{1}{4m} 2m \cdot 2m = m$

Note: $\sum_{u \in V} k_u = 2m$

Modularity

- **Modularity of partitioning S of graph G:**
  - **Q** $\propto \sum_{s \in S}$ **[(# edges within group** s**) –**
    (expected # edges within group s**)]**
  - $Q(G,S) = \underbrace{\frac{1}{2m}} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m} \right)$

Normalizing const.: -1<Q<1

A_{ij} = 1 if i→j,
0 else

- **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:

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

- A_{ij} represents the edge weight between nodes i and j;
- k_i and k_j are the sum of the weights of the edges attached to nodes i and j, respectively;
- $2m$ is the sum of all of the edge weights in the graph;
- c_i and c_j are the communities of the nodes; and
- δ is an indicator function

Idea: We can identify communities by maximizing modularity

Louvain Method

Louvain Algorithm

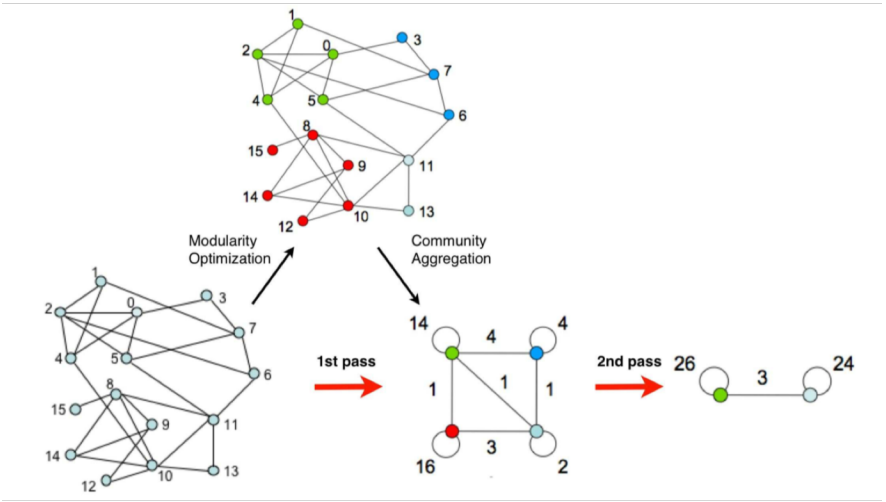
- **Greedy algorithm** for community detection
  - $O(n \log n)$ run time
- Supports weighted graphs
- Provides hierarchical partitions
- Widely utilized to **study large networks** because:
  - Fast
  - Rapid convergence properties
  - High modularity output (i.e., "better communities")

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

Louvain Algorithm: At High Level

- Louvain algorithm **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!



Louvain: 1st phase (partitioning)

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

This first phase stops when a local maximum of the modularity is attained, i.e., when no individual move can improve the modularity.
One should also note that the output of the algorithm depends on the order in which the nodes are considered. Research indicates that the ordering of the nodes does not have a significant influence on the modularity that is obtained.

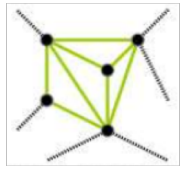
Louvain: Modularity Gain

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

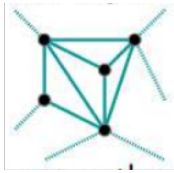
$$\Delta Q(i \to C) = \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

- where:
  - Σ_{in}… sum of link weights between nodes in C
  - Σ_{tot}… sum of all link weights of nodes in C
  - $k_{i,in}$… sum of link weights between node i and C
  - k_i… sum of all link weights (i.e., degree) of node i

Σ_{in}:



Σ_{tot}:

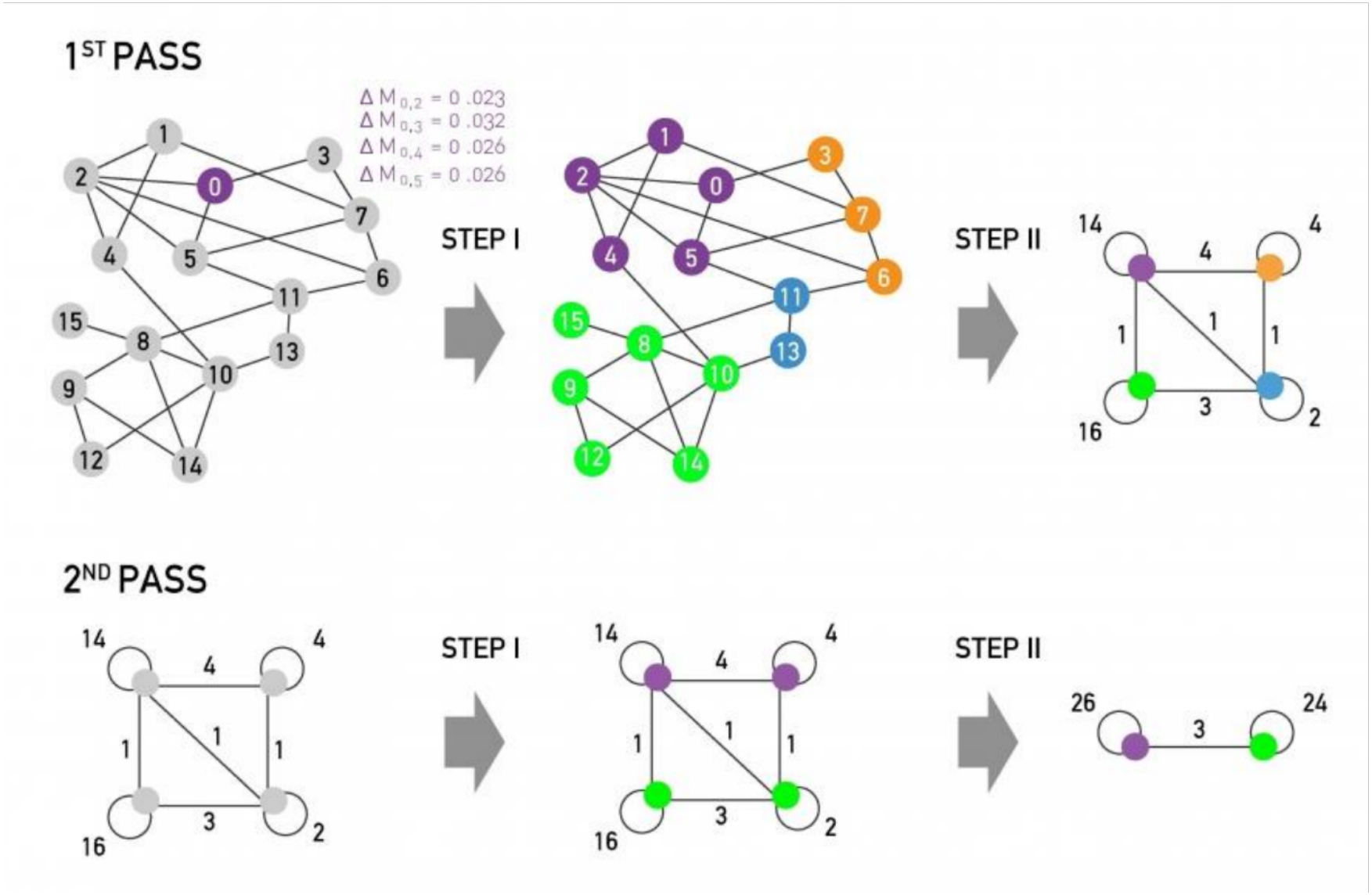


- Also need to derive $\Delta Q(D \to i)$ of taking node i out of community D.
- And then: $\Delta Q = \Delta Q(i \to C) + \Delta Q(D \to i)$

Louvain: 2nd phase (restructuring)

- The partitions obtained in the first 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 Algorithm



Next week: Two exciting guests

Guest Lectures

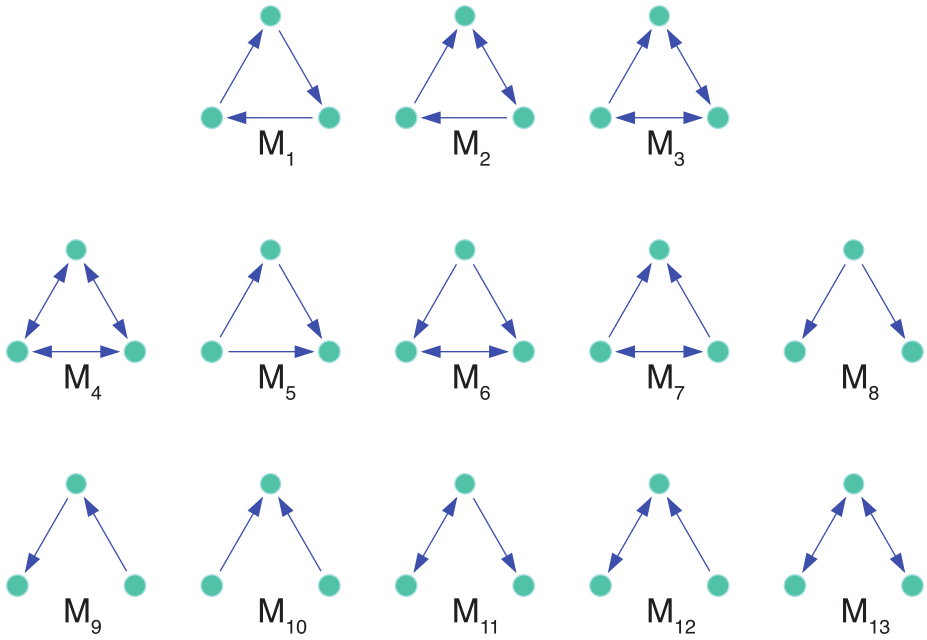
- Tuesday
  - Jevin West, UW iSchool
  - Memory in large networks
  - Mining the scientific literature

- Thursday
  - Su-In Lee, UW CSE
  - Explainable machine learning
  - Applications in Precision Medicine

Motif-Based Spectral Clustering

Motif-based Spectral Clustering

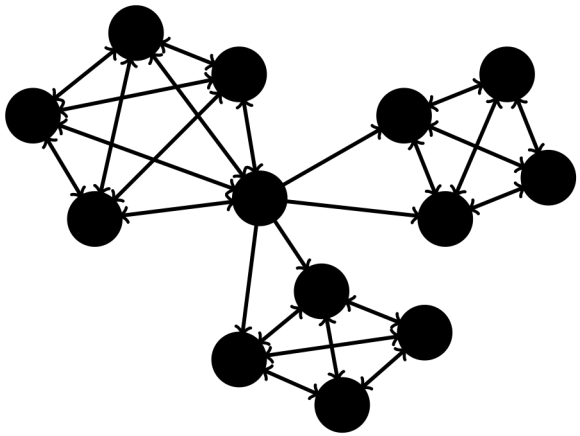
- **What if we want our clustering based on other patterns (not edges)?**



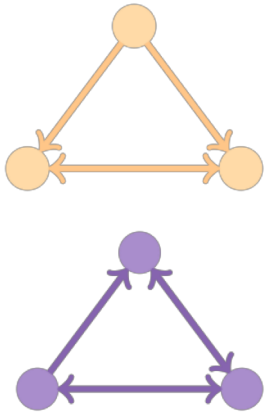
Small subgraphs (motifs, graphlets) are building blocks of networks [Milo et al., '02]

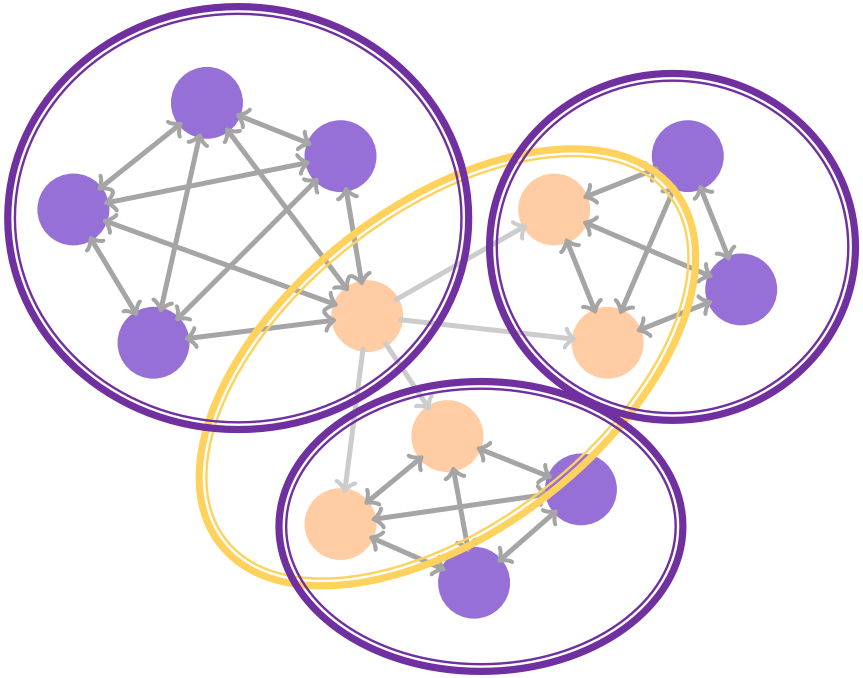
Motif-based spectral clustering

Network:



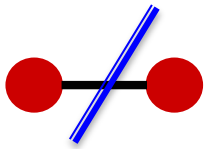
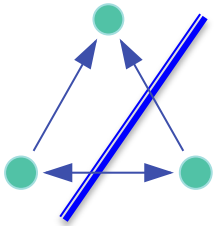
Motif:





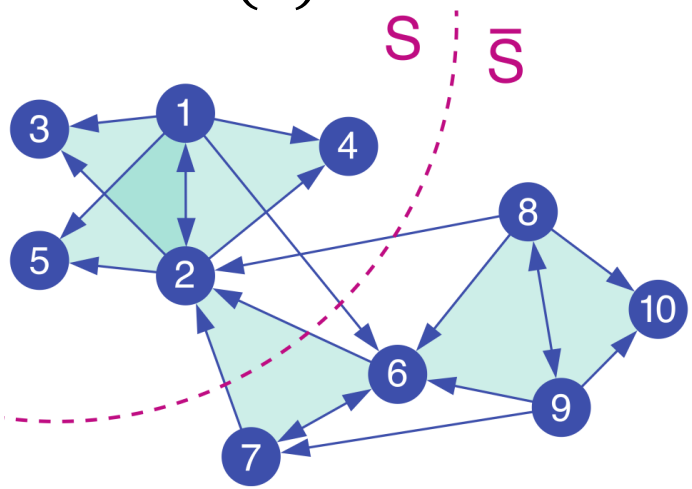
Re-define Conductance for Motifs

- **Generalize cuts and volumes to motifs**

$edges\ cut$  $\Rightarrow$ $motifs\ cut$ 

$vol(S)$ = #(edge end-points in S) $\Rightarrow$ $vol_M(S)$ = #(motif end-points in S)

$$\phi(S) = \frac{\#(edges\ cut)}{vol(S)} \Rightarrow \phi_M(S) = \frac{\#(motifs\ cut)}{vol_M(S)}$$

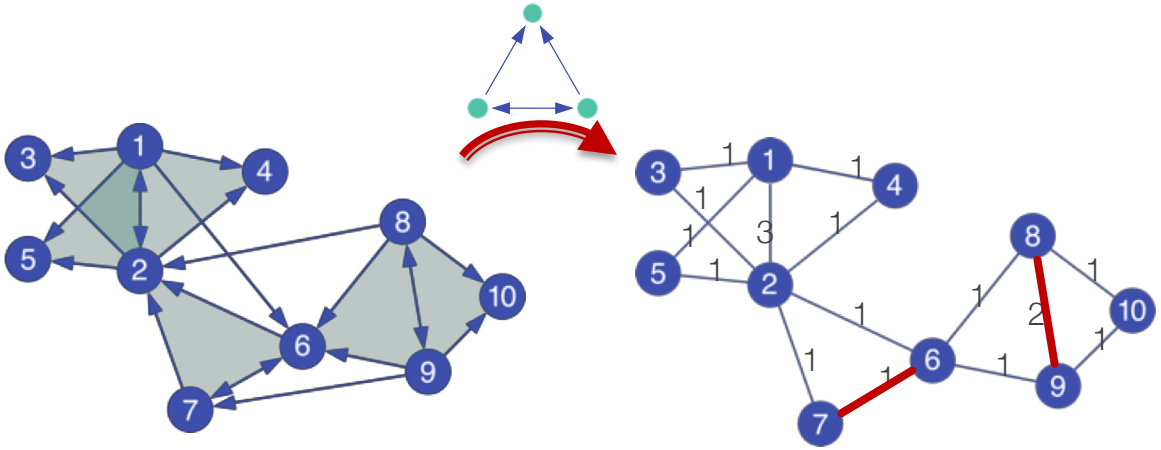


Optimize motif conductance

[Benson et al., '16]

Motif-based Clustering

- **Three basic stages:**
  - **1) Pre-processing**
    - $W_{ij}^{(M)}$ = # times (i, j) participates in the motif

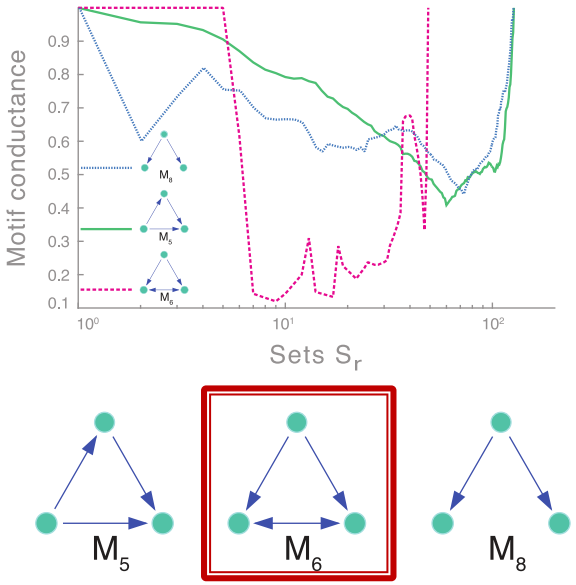


Graph G

Weighted graph $W^{(M)}$

  - **2) PageRank Nibble**
    - Same as before but on weighted $W^{(M)}$
  - **3) Sweep**
    - Same as before

Motif-based Clustering of a Food Web



Use multiple eigenvectors or recursive bi-partitioning to get multiple clusters