Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: http://www.mmds.org

Community Detection in Graphs

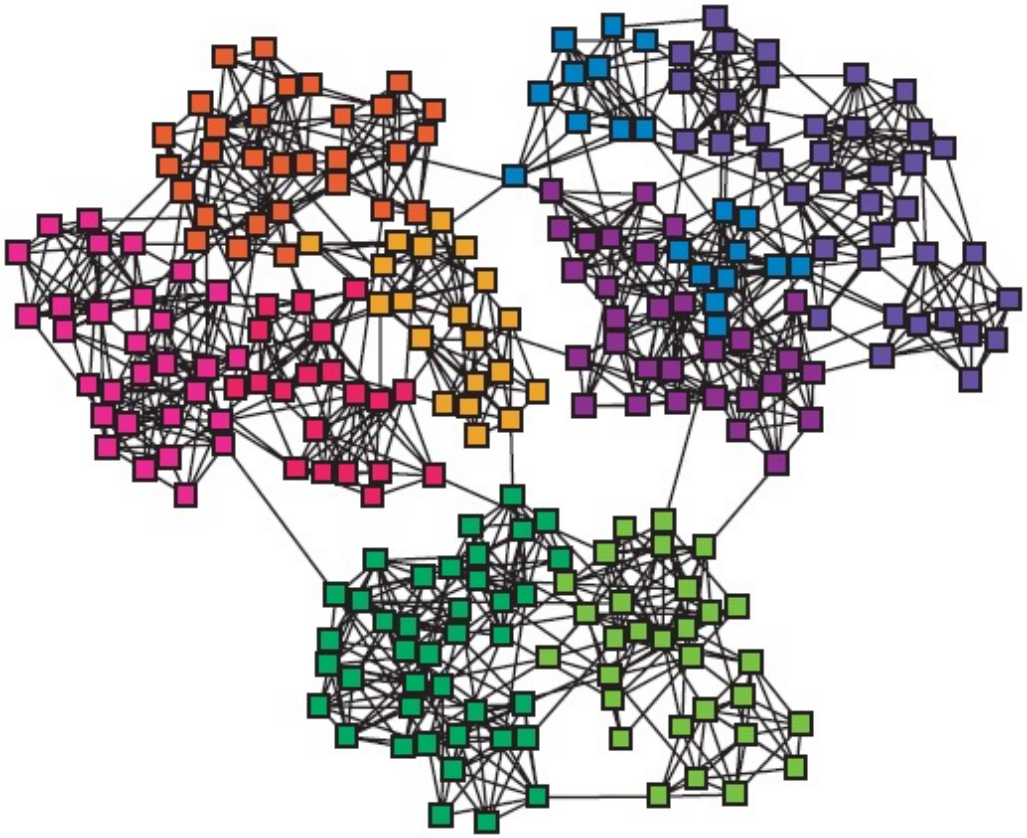
CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
Mina Ghashami, Amazon

http://cs246.stanford.edu

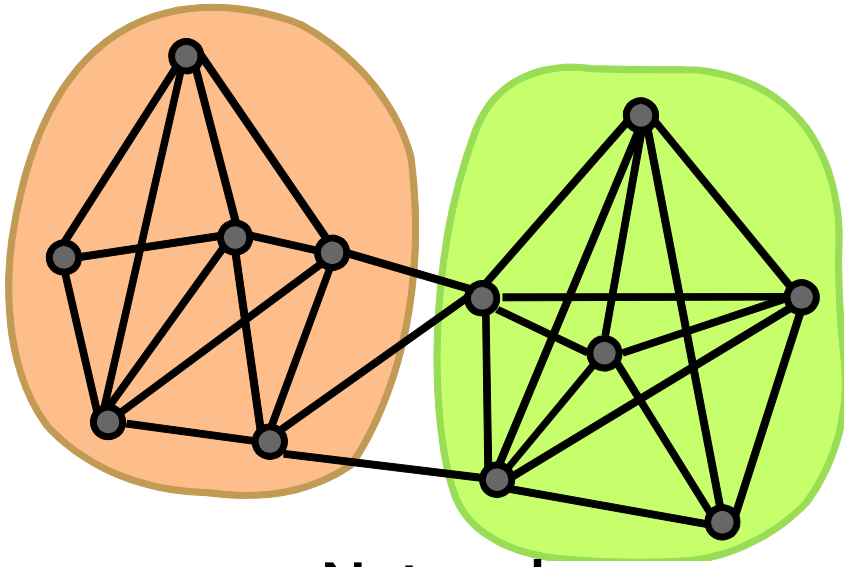


Networks & Communities

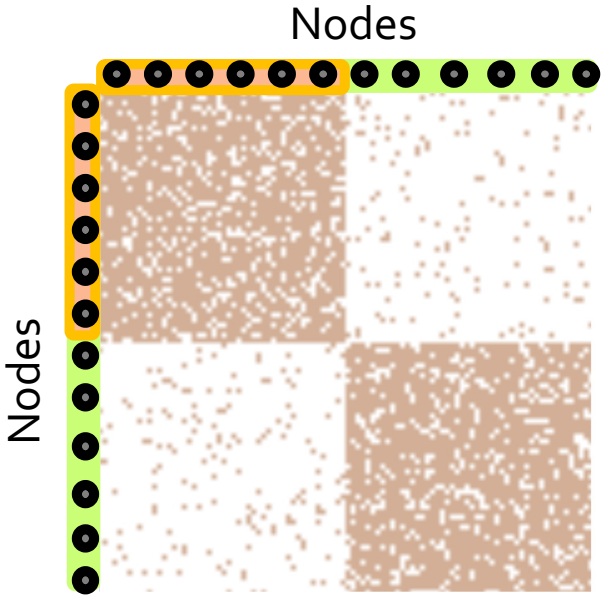
- **We often think of networks being organized into modules, clusters, communities:**



Non-overlapping Clusters

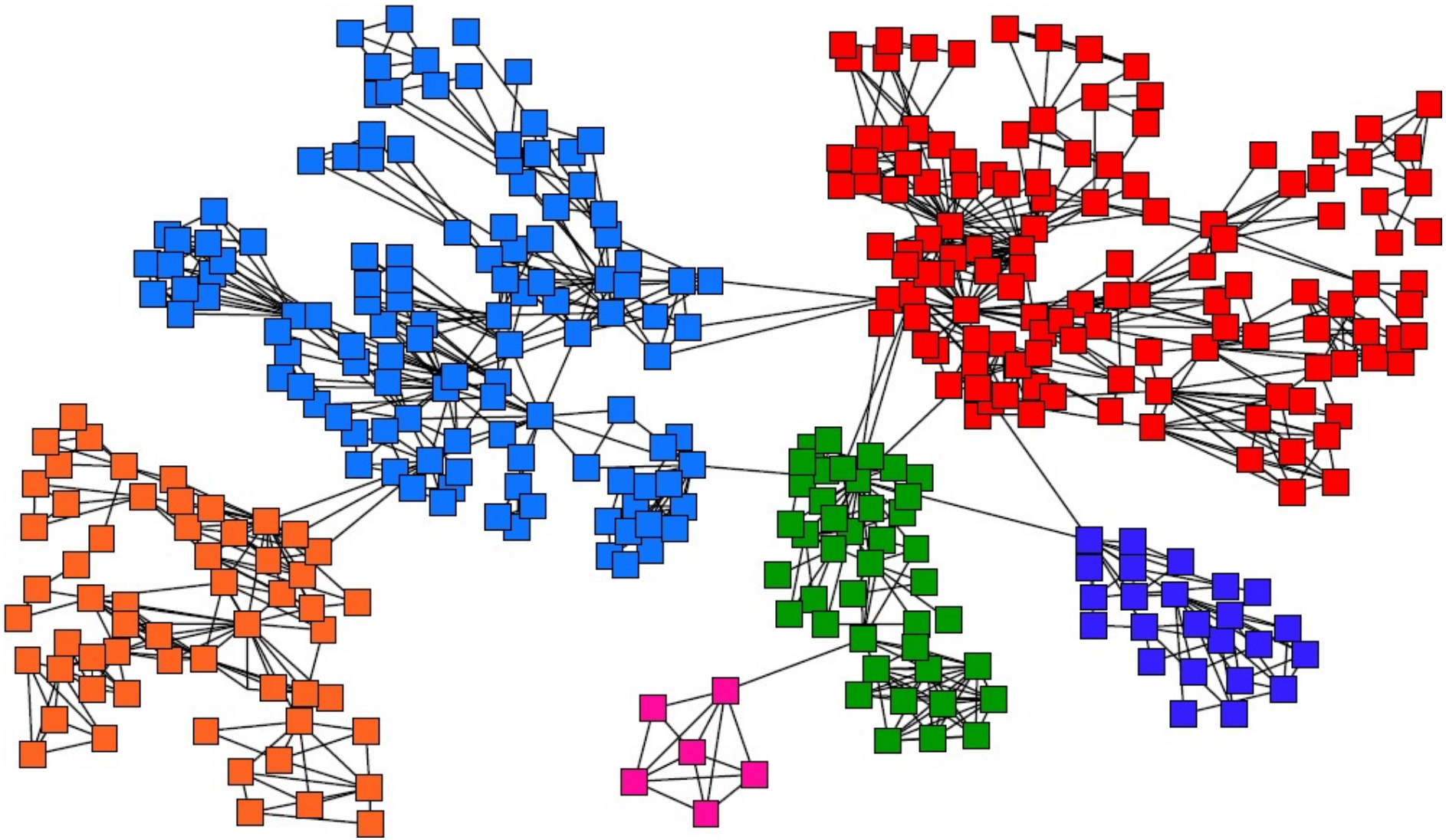


Network



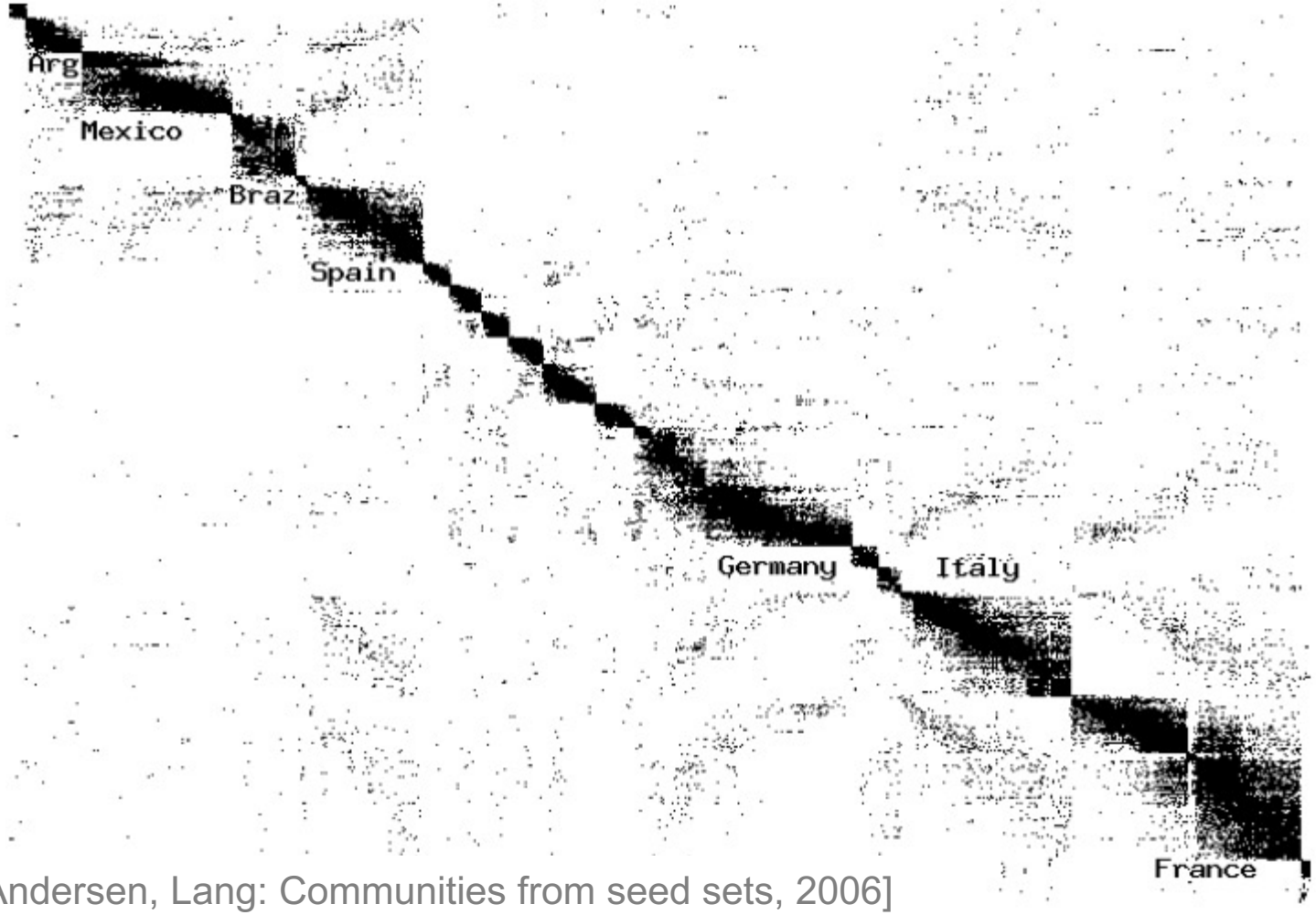
Adjacency matrix

Goal: Find Densely Linked Clusters



Movies and Actors

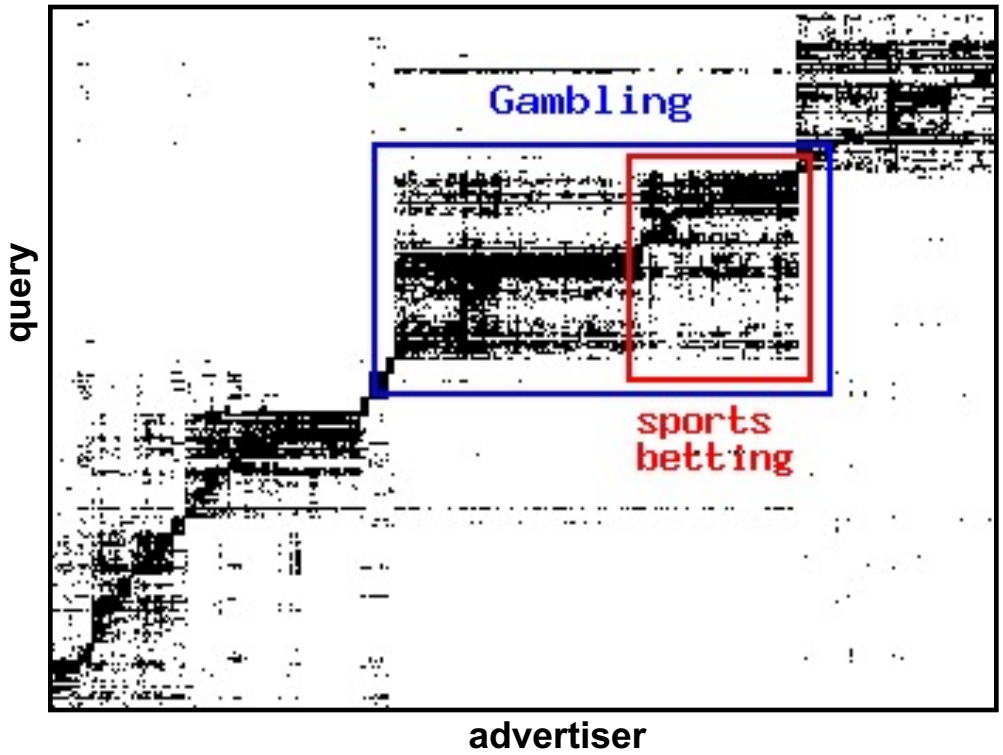
- **Clusters in Movies-to-Actors graph:**



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

Micro-Markets in Sponsored Search

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



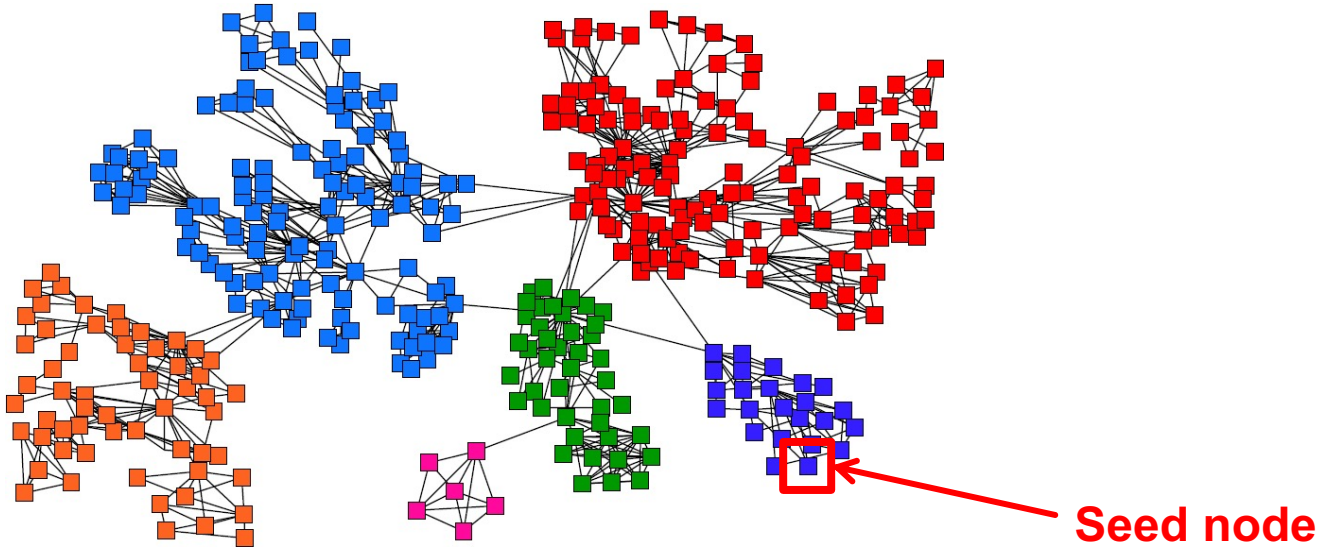
[Andersen, Lang: Communities from seed sets, 2006]

The Setting

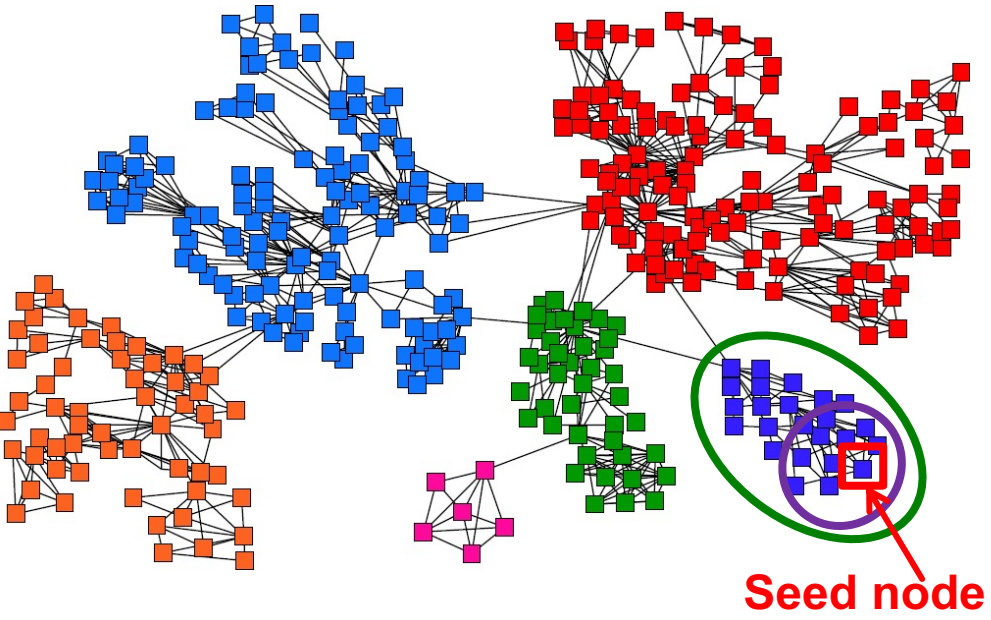
- **Graph is large**
  - **Assume the graph fits in main memory**
    - For example, to work with a 200M node and 2B edge graph one needs approx. 16GB RAM.
  - But the graph is too big for running anything more than linear time algorithms.
- **We will cover a PageRank based algorithm for finding dense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

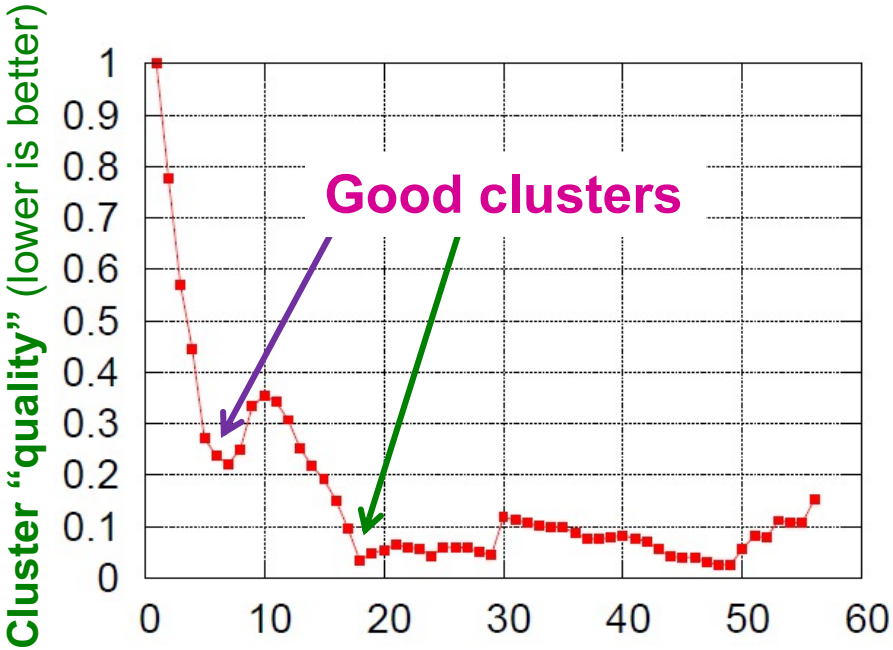
Idea: Seed Nodes

- **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

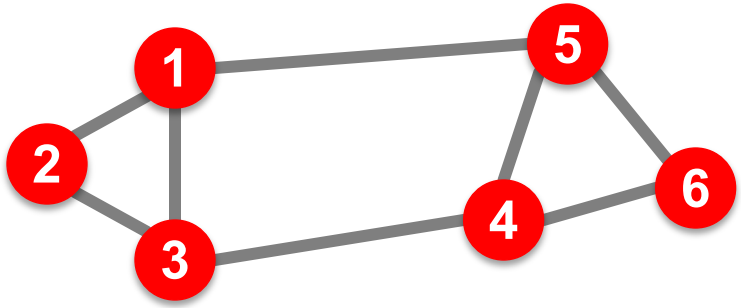




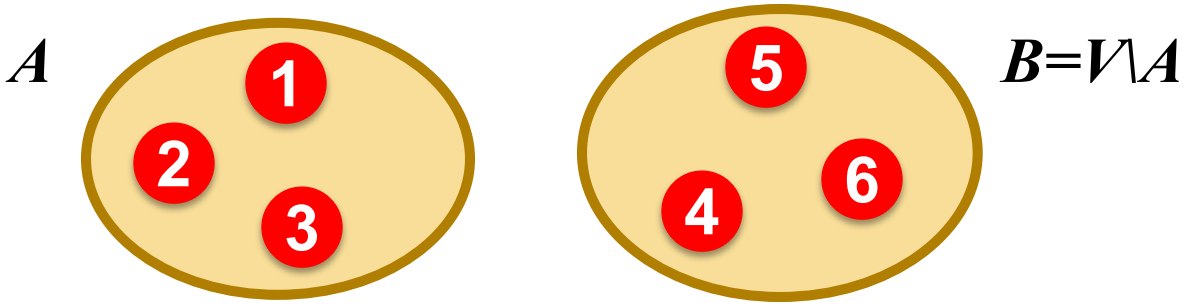
- **Algorithm outline:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** with teleport set = {***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find **good clusters**

What makes a good cluster?

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



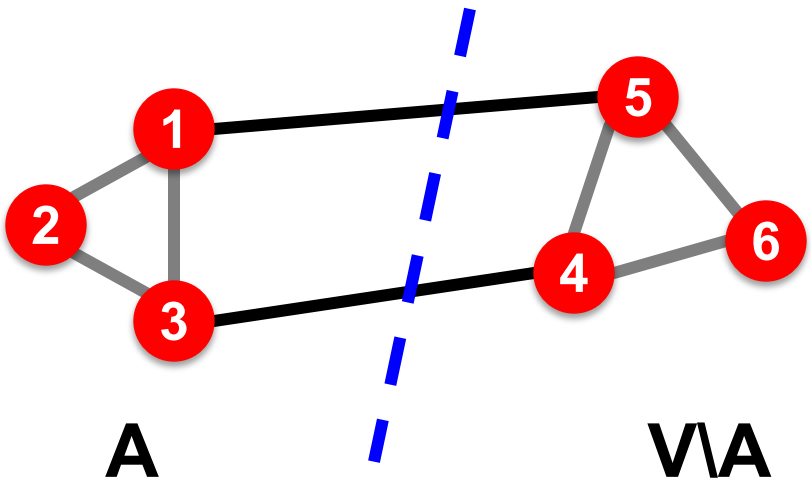
- **Partitioning task:**
  - Divide vertices into 2 disjoint groups $A, B = V \setminus A$



- **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**

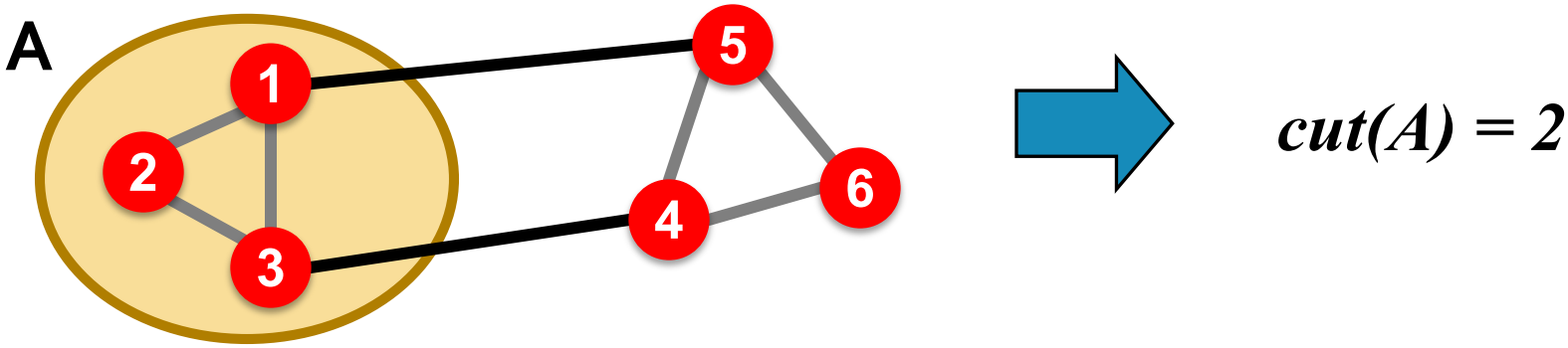


Graph Cuts

- **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:

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

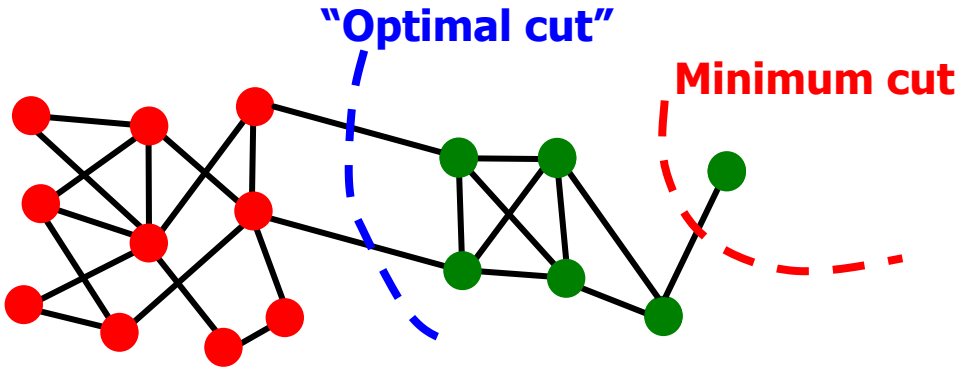
Note: This works for weighted and unweighted (set all $w_{ij}=1$) undirected graphs



$$cut(A) = 2$$

Cut Score

- **Partition quality: Cut score**
  - Quality of a cluster is the weight of connections pointing outside the cluster
- **Not so uncommon case:**



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

[Shi-Malik]

Graph Partitioning Criteria

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

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

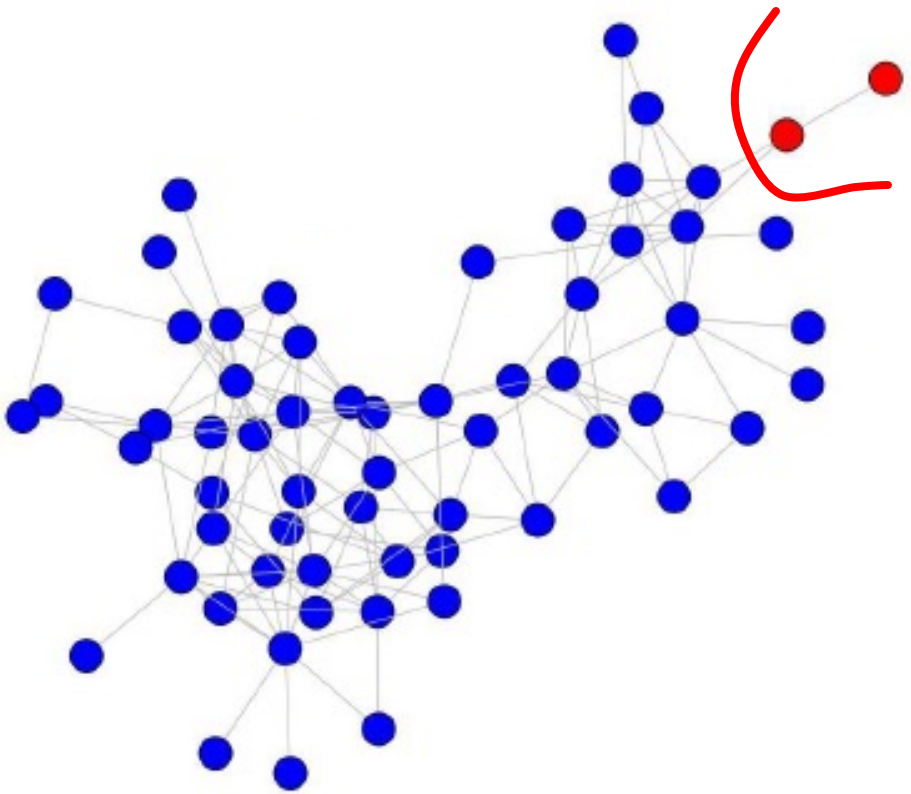
$\boldsymbol{vol(A)}$: total weight of the edges with at least one endpoint in $\boldsymbol{A}$: $\mathbf{vol}(\boldsymbol{A}) = \sum_{i \in A} \boldsymbol{d_i}$

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

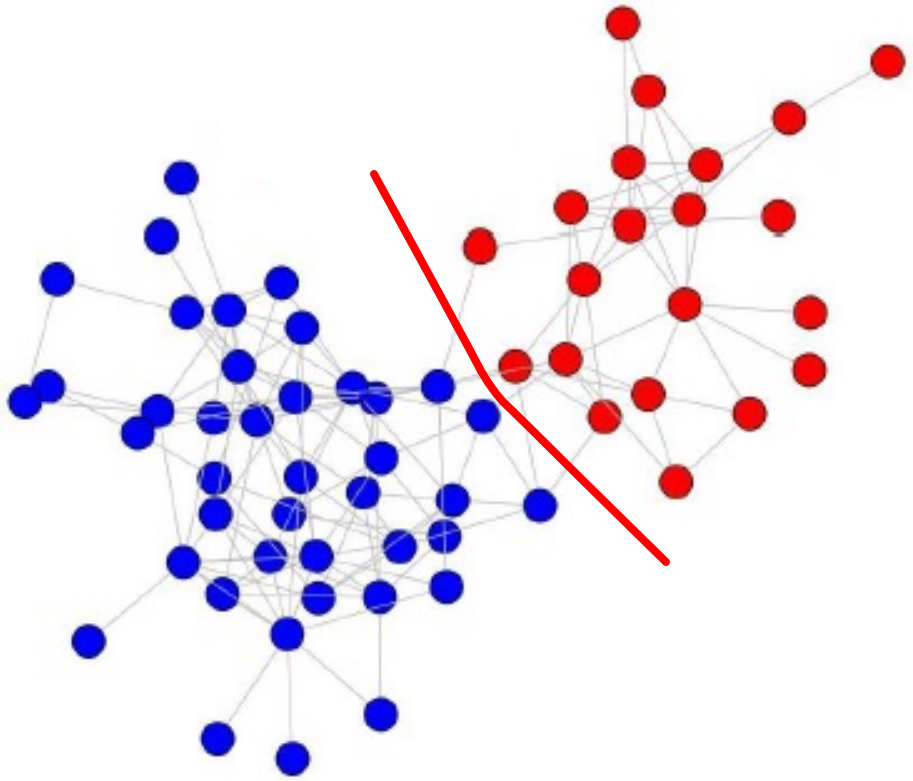
- **Why use conductance?**
  - Produces more balanced partitions

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

Example: Conductance Score



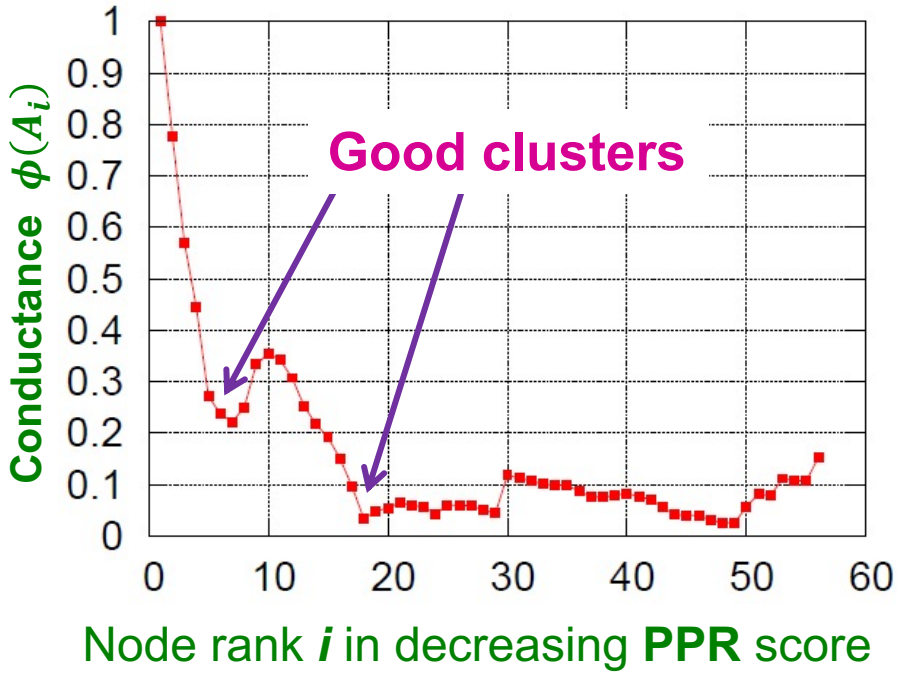
$\phi = 2/4 = 0.5$



$\phi = 6/92 = 0.065$

Algorithm Outline: Sweep

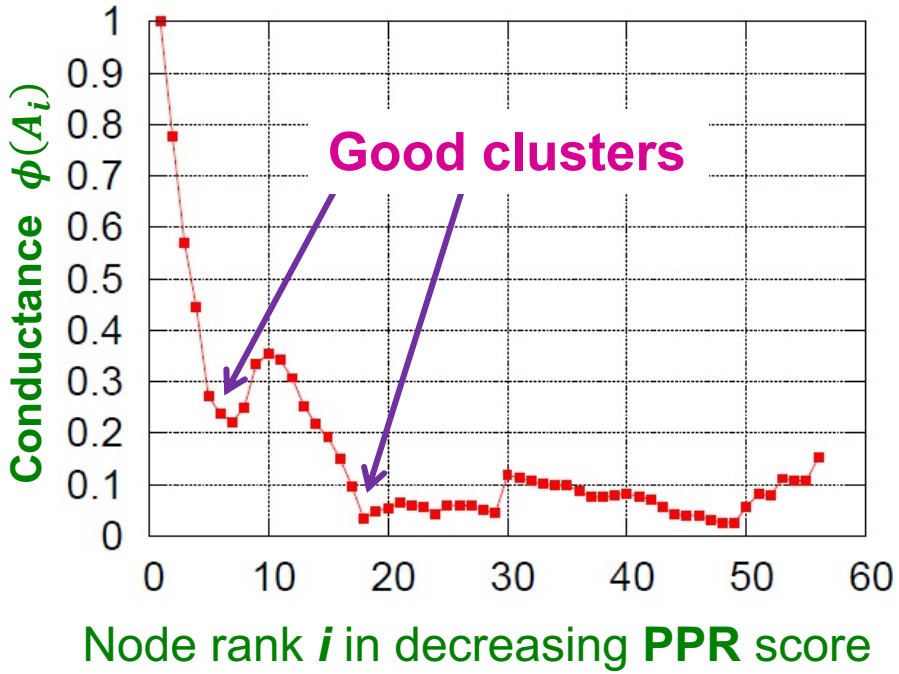
- **Algorithm outline:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** w/ teleport={***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find good clusters



- **Sweep:**
  - Sort nodes by decreasing PPR score $r_1 > r_2 > \cdots > r_n$
  - For each $\boldsymbol{i}$ compute $\boldsymbol{\phi(A_i = \{u_1, \ldots u_i\})}$
  - **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 (AKA PageRank-Nibble)** [Andersen, Chung, Lang, '07]
  - A fast method for computing approximate Personalized PageRank (**PPR**) with teleport set **S={s}**
  - **ApproxPageRank(s, β, ε)**
    - **s** … seed node
    - **β** … teleportation parameter
    - **ε** … approximation error parameter

Approximate PPR: Overview

- **Approximate PPR on undirected graph**
  - **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** q_u: how well is PPR score p_u of u is approximated
    - p_u… is the "true" PageRank of node u
    - $r_u^{(t)}$… is PageRank estimate of node u at around t

    If **residual** q_u of node 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 outgoing edges), else we don't touch the node

Towards approximate PPR

Why does Approximate PPR work?

- **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
- So, we can evolve our estimate of $p_\beta(a)$ step by step via the push operations
  - To understand why this is true, we will use the fact that $p_\beta(a)[i]$ is the probability of ending at node i after a single random walk from distribution a with length $\sim geometric(\beta)$.

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: a…teleport vector**
  - 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'

1-β…teleport prob

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 the excess PageRank

Approximate PPR

- **ApproxPageRank(S, β, ε):**

  Set $\boldsymbol{r} = \vec{0}, \boldsymbol{q} = [0\,..\,0\;1\;0\;...\;0]$

  While $\max_{u \in V} \frac{q_u}{d_u} \geq \varepsilon$: ← At index **S**

  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 $u \to v$:

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

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

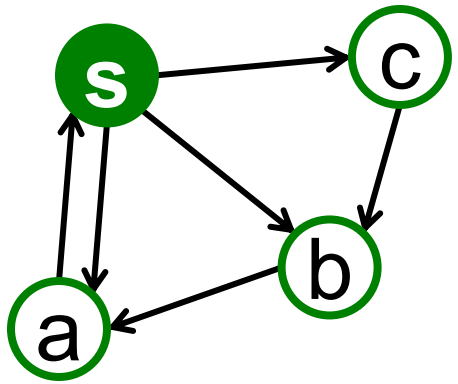
  Return r

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 **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**

Example: $\beta = 0.5$



ApproxPageRank(S, β, ε):

Set $\boldsymbol{r} = \vec{0}$, $\boldsymbol{q} = [0\,..\,0\;1\;0\;...\;0]$

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

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

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

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

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

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

For each $\boldsymbol{v}$ **such that** $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$:

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

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

return $\boldsymbol{r}$

s a b c

Init:

r = [0, 0, 0, 0]

q = [1, 0, 0, 0]

Push(s, r, q):

r = [.5, 0, 0, 0]

q = [.25, .08, .08, .08]

Push(s, r, q):

r = [.62, 0, 0, 0]

q = [.06, .10, .10, .10]

Push(a, r, q):

r = [0.62, .05, 0, 0]

q = [.09, .03, .10, .10]

Push(b, r, q):

r = [0.62, .05, .05, 0]

q = [.09, .05, .03, .10]

....

r = [0.57, 0.19, 0.14, 0.09]

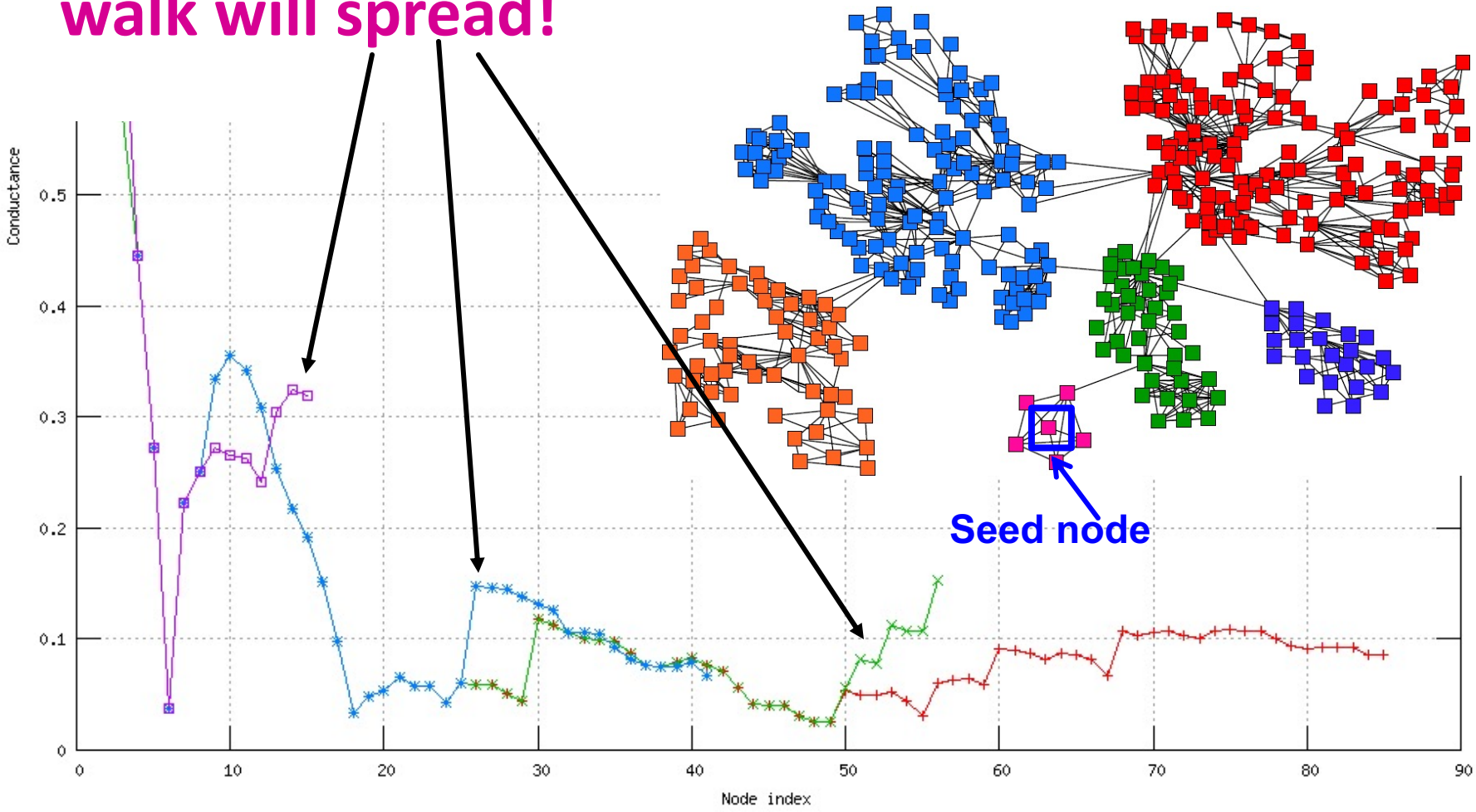
Observations (1)

- **Runtime:**
  - Approximate PageRank computes PPR in time $\left(\frac{1}{\varepsilon(1-\beta)}\right)$ 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]

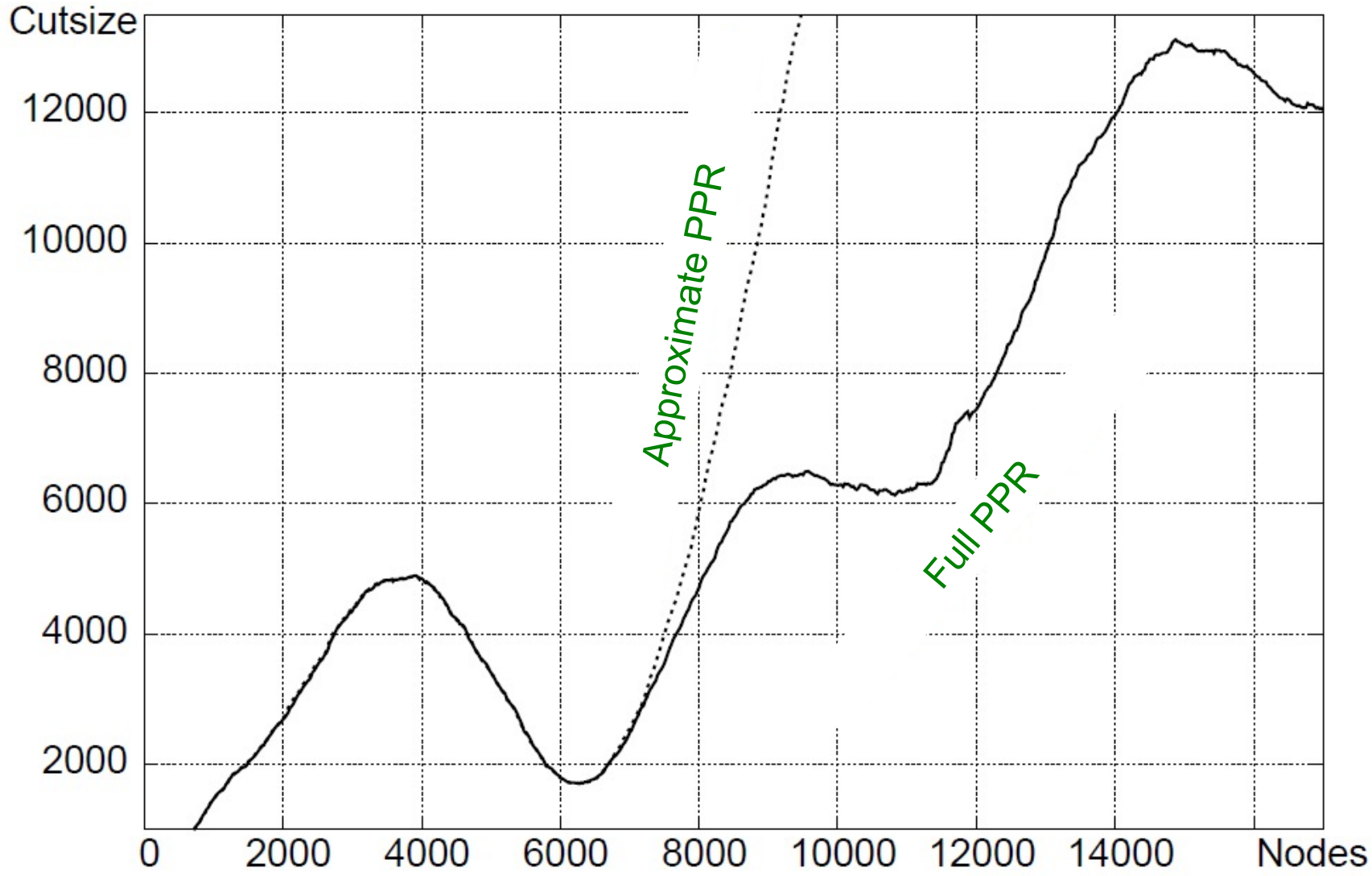
http://www.math.ucsd.edu/~fan/wp/localpartfull.pdf

Observations (2)

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

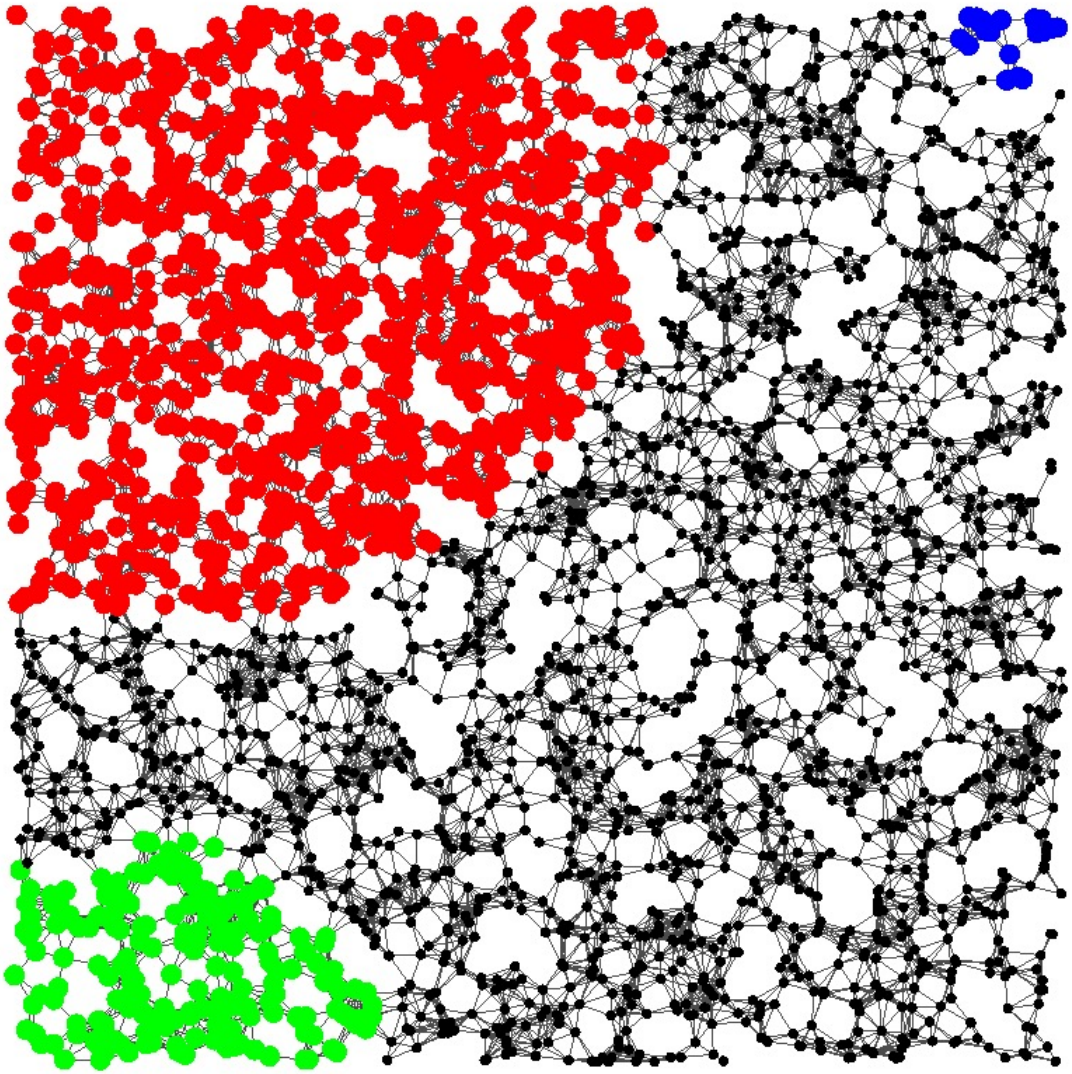


Observations (3)

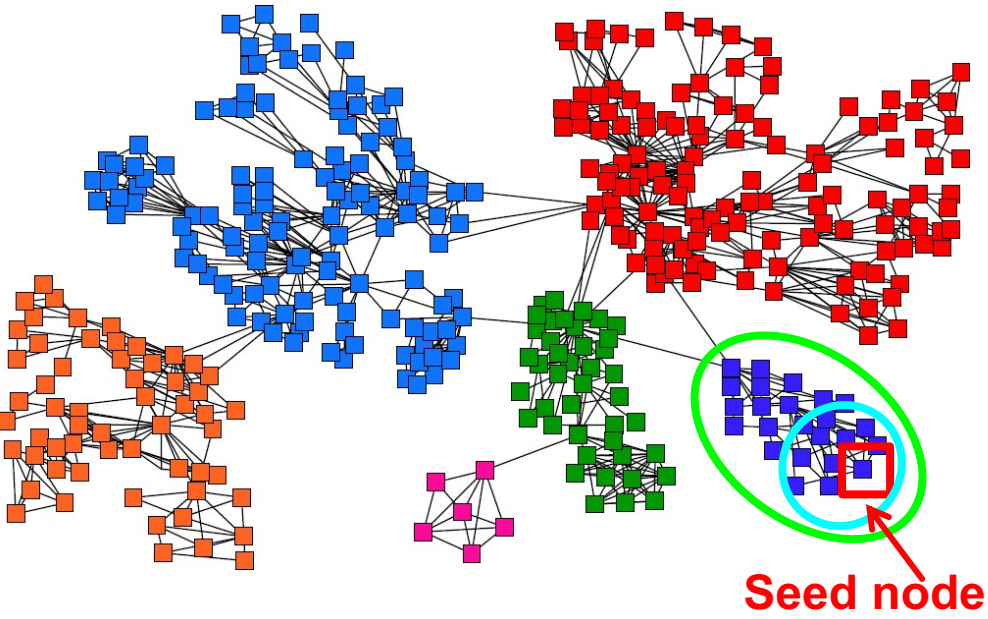


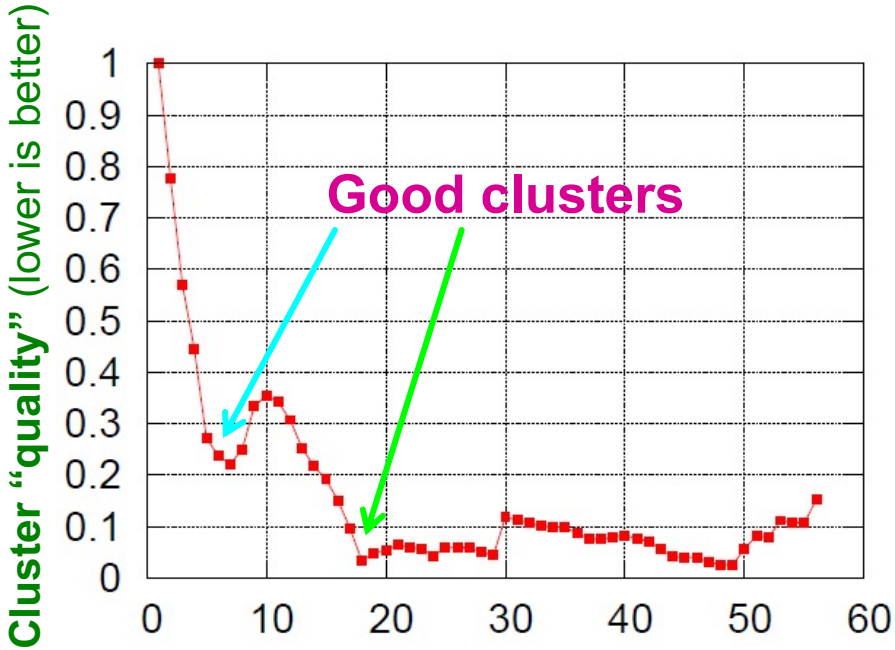
[Andersen, Lang: Communities from seed sets, 2006]

Example



Summary of Approx PPR Alg.



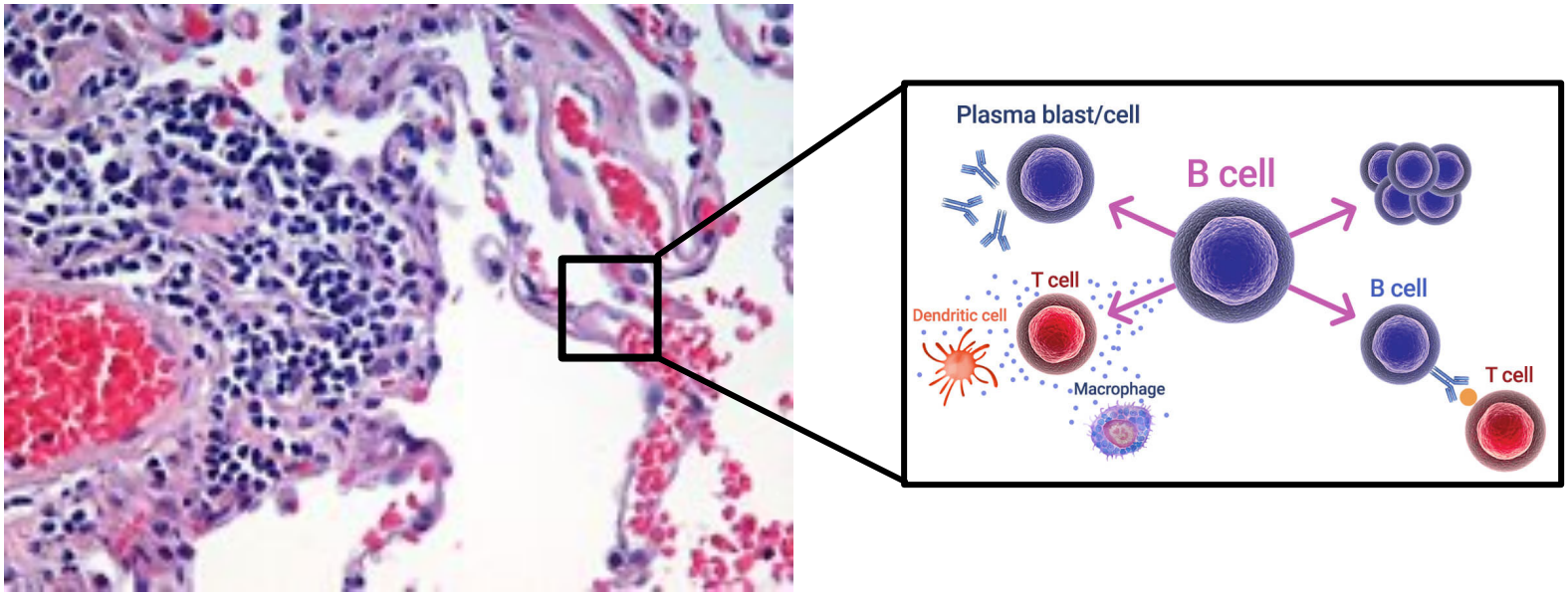


- **Algorithm summary:**
  - Pick a seed node ***s*** of interest
  - Run **PPR** with teleport set = {***s***}
  - Sort the nodes by the decreasing **PPR** score
  - **Sweep** over the nodes and find good clusters

Community Detection Motivation

Cells Are Heterogenous

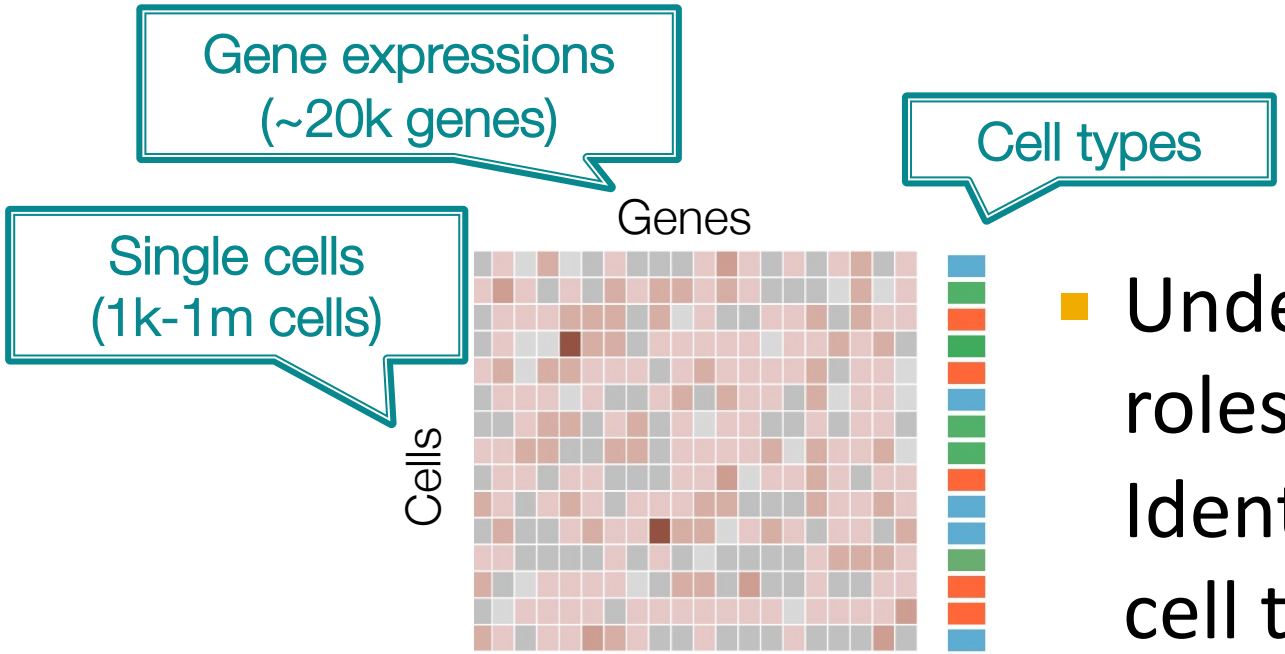
Every cell in a tissue has a specific role



Challenge:

How to determine roles of cells?

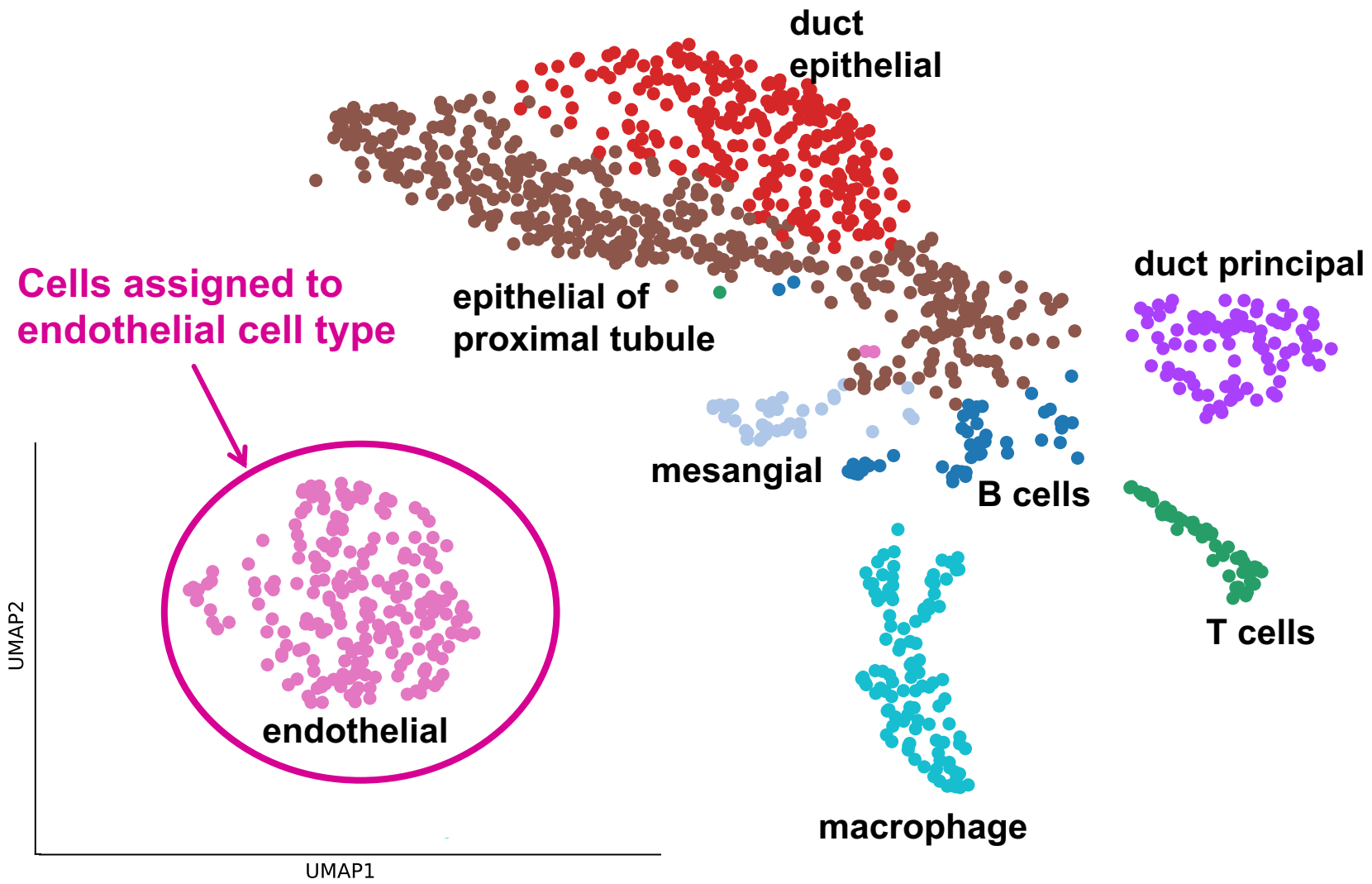
Cell Type Identification Task



- Understanding the roles of cells: Identification of cell types

- **Cell type identification task: Given gene expressions of cells, assign cells to cell types**
  - Boils down to a **clustering task:** group cells according to their gene expression similarities

Cell Type Identification Task



[UMAP. McInnes, Healy, Melville. '18]

Challenges with Standard Clustering

- **Can we use standard clustering methods such as K-means to solve this problem?**
- **Why standard cluster methods do not work well?**
  - Data is very high-dimensional (~20k genes per cell)
  - Data is noisy and sparse (most values will be zero)
  - Number of clusters (cell types) is unknown
  - Cell types are hierarchically organized
    - **Definition of cell type is provisional**
    - **One cell type can have multiple cell subtypes**
    - **Where to put a threshold on a definition of a cell type?**

Idea: Represent Cells as a Graph

- **Idea:** Construct a **graph** between data points (cells) and detect **hierarchical network communities** in a graph

Why is graph a good representation?

- **Natural representation:** models cell-cell **interaction**
- Cells with more similar gene expressions are more likely to interact
  - Construct a graph based on similarities between gene expressions of cells
- Hierarchical network communities model well cell type hierarchy

Up Next

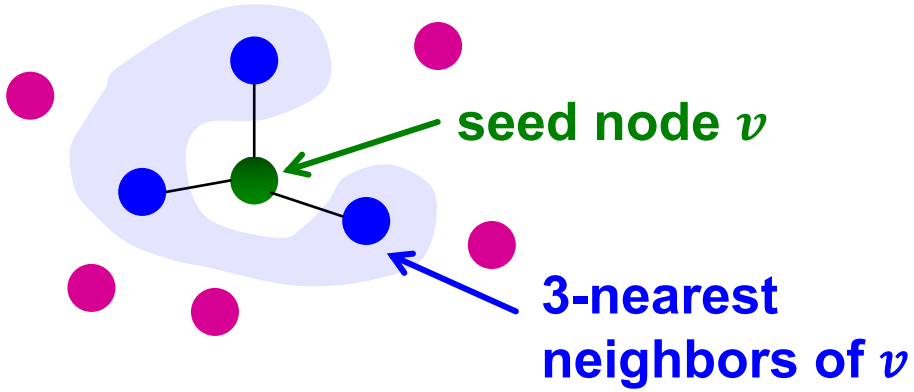
We will cover next:

1) How to construct a graph from high-dimensional data?
   - Efficient k-NN graph construction
2) How to define network communities?
   - Modularity
3) How to detect communities?
   - Louvain algorithm

Efficient K-NN Graph Construction

K-NN Graph

- **K nearest neigbor (K-NN) graph:** Directed graph with vertex set V and an edge from each $v \in V$ to its K **most similar objects** in V under a given similarity measure
  - *E.g.*, cosine similarity, l_2 distance, l_1 distance



Computing K-NN Graph

- **Brute force algorithm:**
  - Takes $\boldsymbol{O(n^2)}$ time
  - Only practical for small datasets!

How to efficiently compute K-NN graph?

- **NN-Descent** [Dong, Charikar, Li. '11]
  - Scalable method for creating approximate K-NN graph
  - Suitable for large-scale datasets
  - Empirical cost is around $\boldsymbol{O(n^{1.14})}$
  - Suitable for distributed implementation (*e.g.*, Map Reduce)

NN-Descent Heuristic

- **Idea: a neighbor of a neighbor is also likely to be a neighbor**
  - **Δ diameter of the whole dataset**
    - **Diameter = 2*radius**
  - **Heuristic argument:** if K is large enough then even if we start from a random K-NN approximation we are likely to find for each object K items with a **radius of Δ/2** by exploring its neighbors' neighbors.
  - Conceptually we iteratively **shrink** the radius until the nearest neighbors are found.

NN-Descent Heuristic

NN-Descent is an iterative refinement algorithm:

- Start with a random KNN graph
  - Each node picks K random other nodes as its nearest neighbors.
- Iteratively refine the list of nearest neighbors of each node:
  - A neighbor of a neighbor could also be my neighbor.
- Keep doing this until convergence.

NN-Descent Algorithm

- Start with a random K-NN list by sampling K items for every node $v \in V$
- Then iteratively for every node $v \in V$:
  - $\boldsymbol{B[v]}$ … is the current/approximate K-NN of v
  - $\boldsymbol{R[v]}$ … is the current/approximate reverse K-NN of v
    - **Reverse K-NN**: $\boldsymbol{R[v] = \{u \in V | v \in B[u]\}}$
  - Get **general neighbors** $\boldsymbol{B^*[v] = B[v] \cup R[v]}$
  - For each **general neighbor** $\boldsymbol{u \in B^*[v]}$, check the similarity between $\boldsymbol{v}$ and $\boldsymbol{B^*[u]}$ (general neighbors of u are candidates for new neighbors of v)
  - Update nearest neighbors list if similarity is higher compared to the set of current approximate neighbors

Efficient KNN Graph Construction

- **NNDescent(V,σ, K):**

  $B[v] = Random\ sample\ of\ K\ items\ V,\ \forall v \in V$

  Loop:

  $R = reverse\ (B)$

  $B^*[v] = B[v] \cup R[v],\ \forall v \in V$

  $c = 0$

  for $v \in V$**:**

  for $u_1 \in B^*[v], u_2 \in B^*[u_1]$**:**

  $l = \sigma(v, u_2)$

  $c = c + updateNN\ (B[v], \langle u_2, l \rangle)$

  return B **if** $c = 0$

V … dataset∂
σ …similarity oracle
K …number of neighbors
$B[v]$ … approximate neighbors of v
$R[v]$ … approximate reverse neighbors of v
$B^*[v]$ … approximate general neighbors of v
c … counter

$B[v]$ is organized as a heap
→ updates cost $O(logK)$

- $reverse(B)$:

  $R[v] = \{u \mid \langle v, \dots \rangle \in B[u]\},\ \forall v \in V$

  return R

- $updateNN(H, \langle u, l, \dots \rangle)$:

  Update KNN heap H

  return 1 if changed, 0 if not

Example: $K = 2$

Neighbors:

$$B[s] = \{c, d\}$$

Reverse neighbors:

$$R[s] = \{b, c, e\}$$

General neighbors:

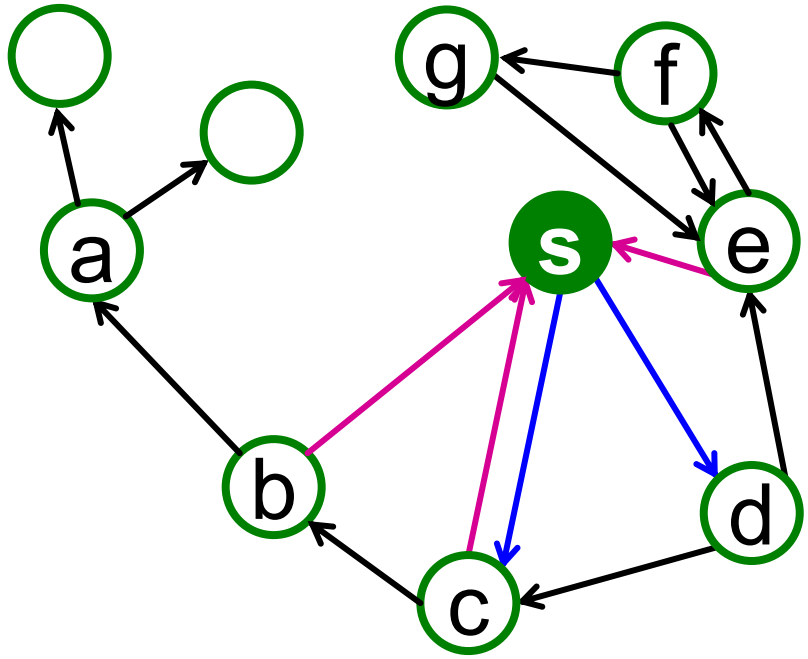
$$B^*[s] = \{b, c, d, e\}$$

$$B^*[b] = \{a, c, s\}$$

$$B^*[c] = \{b, d, s\}$$

$$B^*[d] = \{c, e, s\}$$

$$B^*[e] = \{d, f, g, s\}$$



General neighbors of $B^*[s]$ are new candidates for $B[s]$

We will check $\{a, f, g\}$ as next candidates for $B[s]$: Compute $\sigma(s, a)$, $\sigma(s, f)$, $\sigma(s, g)$ and update NNs of s.

Arrows denote neighbors of a particular node. For example, arrow from b to s means that b selected s as its neighbor (but the opposite does not need to be true).

Further Improvements

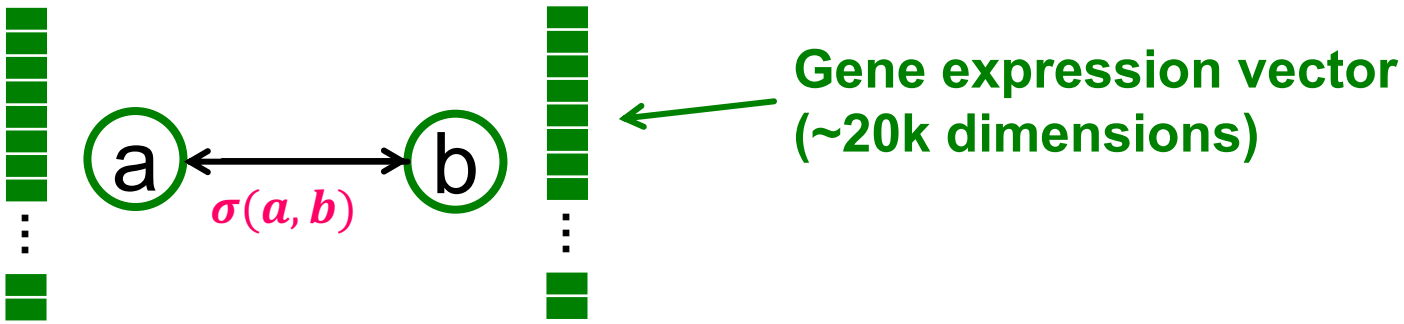
- **Basic algorithm can be further optimized:**
  - **Local join:** Given node v and its neighbors $\bar{B}[v]$, compute similarity between each pair of different $p, q \in B^*[v]$ and update $B[p]$ and $B[q]$ with the similarity (reduces number of comparisons)
  - **Incremental search:** Attach a Boolean flag to remember if the objects have already been compared
  - **Early termination:** Count the number of K-NN list updates in each iteration, and stop when it becomes less than δKN where δ is a precision parameter
  - **MapReduce** implementation

Details in [Dong, Charikar, Li. *Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures*, 2011]

https://dl.acm.org/doi/pdf/10.1145/1963405.1963487

Cell Type Identification Example

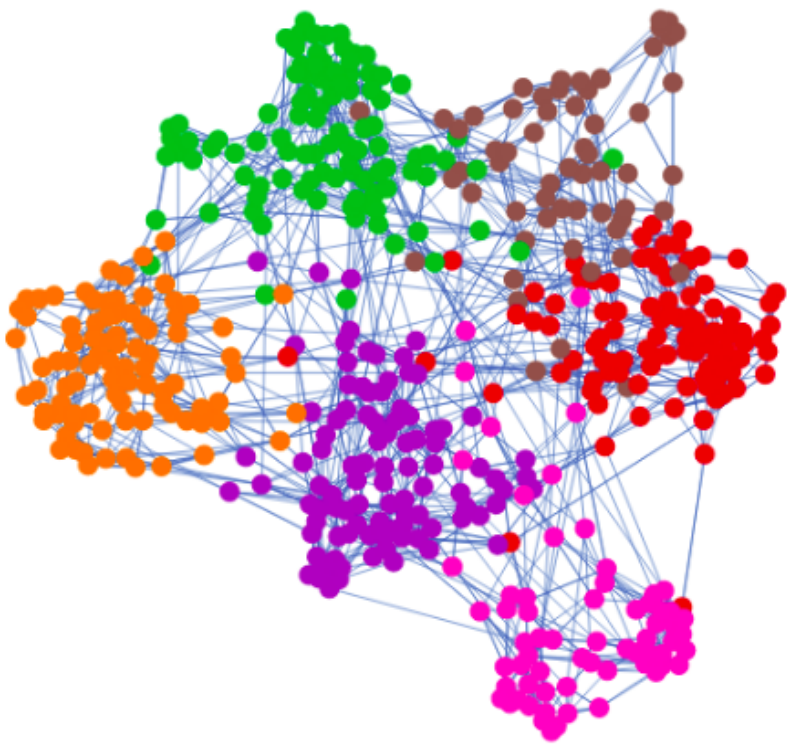
- **Which similarity measure σ to use?**
  - Cells are compared based on their gene expression profiles
  - **Challenge:** Number of genes is very high-dimensional



  - **Approach:** First **apply SVD** (around 50 dimensions) and then compute l_2 distance in the low-dimensional space

After KNN Graph Construction

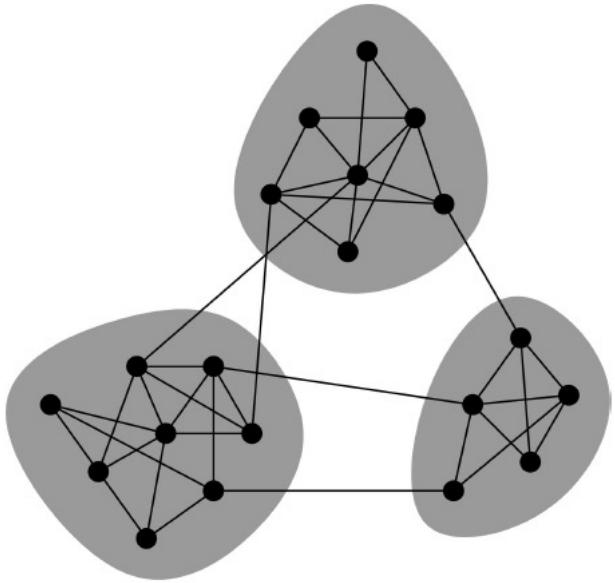
- Once we created K-NN graph of cells, how do we **define and detect network communities**?



Step 2:
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 $s \in S$:

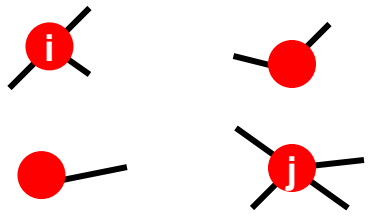


$$Q \propto \sum_{s \in S} [\text{(\# edges within group } s) - \text{(expected \# edges within group } s)]$$

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

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:

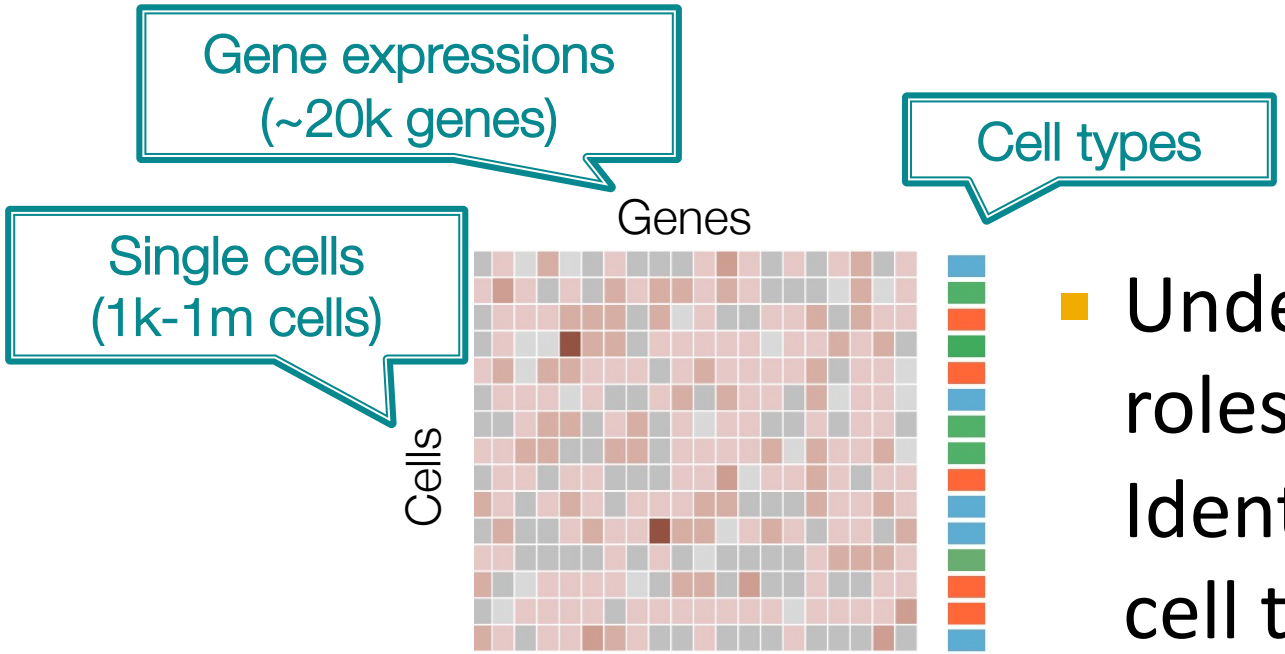
$$Q = \frac{1}{2m}\sum_{ij}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$$

- 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

RECAP: Community Detection

Cell Type Identification Task

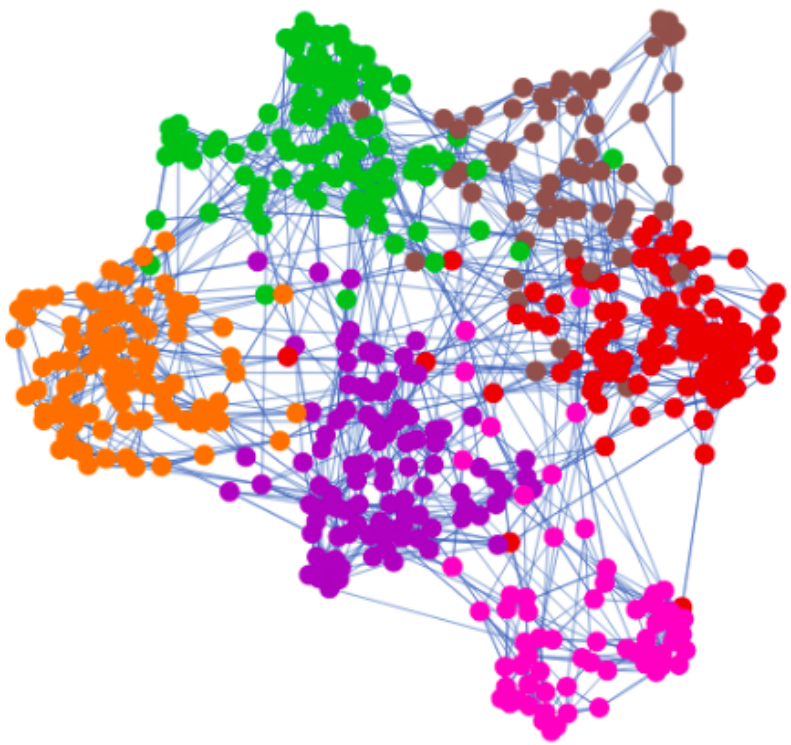


- Understanding the roles of cells: Identification of cell types

- **Cell type identification task: Given gene expressions of cells, assign cells to cell types**
  - Boils down to a **clustering task:** group cells according to their gene expression similarities

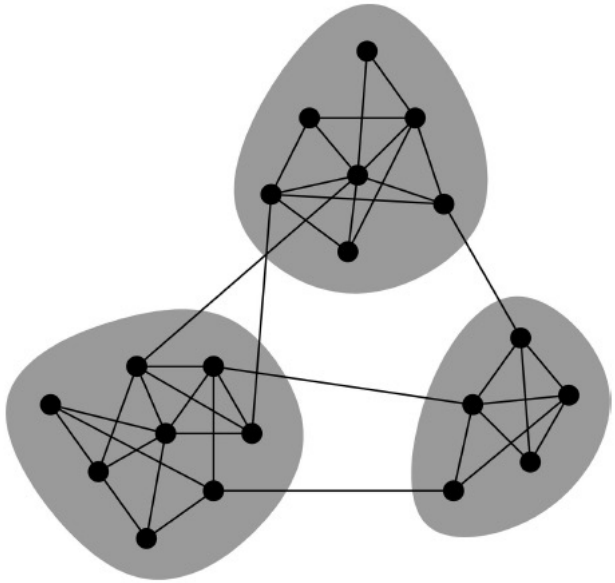
After KNN Graph Construction

- Once we created K-NN graph of cells, how do we **define and detect network communities**?



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 $s \in S$:

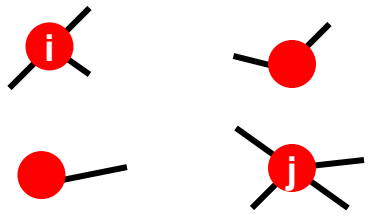


$$Q \propto \sum_{s \in S} [\text{(\# edges within group } s) - \text{(expected \# edges within group } s)]$$

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

Normalizing const.: -1<Q<1

A_{ij} = 1 if i→j,
0 else

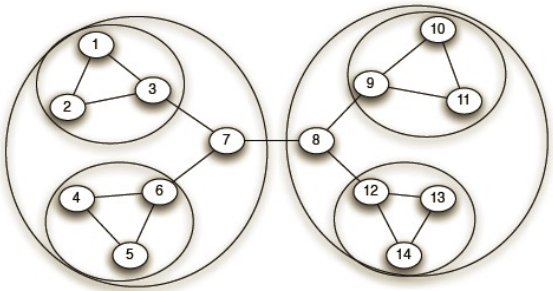
- **Modularity values take range [–1,1]**
  - It is positive if the number of edges within groups exceeds the expected number
  - **Q** greater than **0.3-0.7** means **significant community structure**

Louvain Algorithm

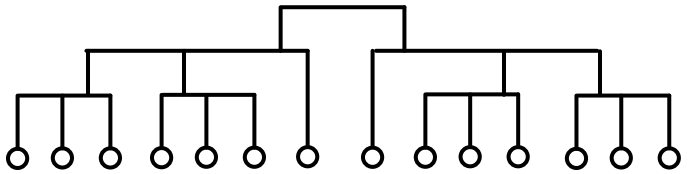
Louvain Algorithm

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

Network and communities:



Dendrogram:

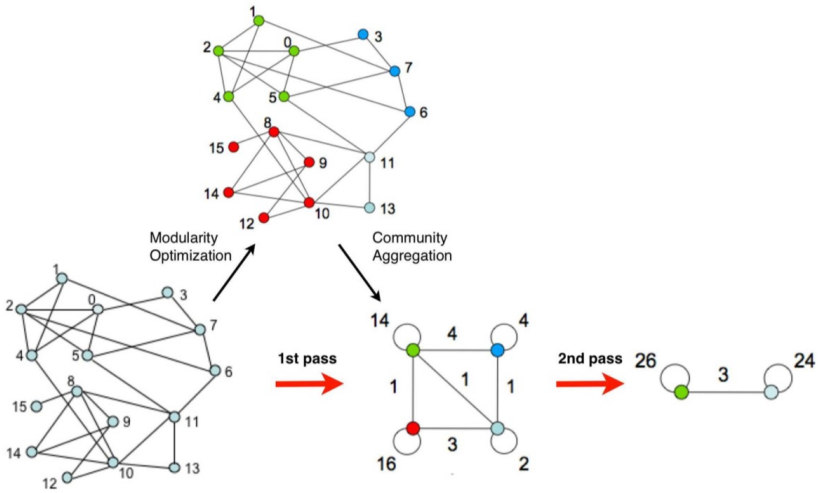


"Fast unfolding of communities in large networks" Blondel et al. (2008)

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 to node-communities memberships
  - **Phase 2:** The identified communities are aggregated into super-nodes to build a new network
  - **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 delta (ΔQ) when putting node i into the community of some neighbor j
  - Move i to a community of node j that yields the largest gain in ΔQ
- **Phase 1 runs until no movement yields a gain**

This first phase stops when a local maxima of the modularity is attained, i.e., when no individual node move can improve the modularity.
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 overall 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}:



Louvain: Modularity Gain

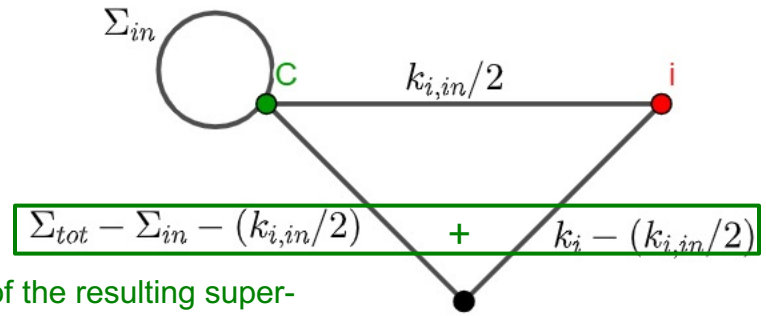
More in detail:

Modularity contribution **after merging** node *i*

Modularity contribution **before merging** node *i*

$$\Delta Q(i \to C) = \left[\frac{\Sigma_{in}+k_{i,in}}{2m} - \left(\frac{\Sigma_{tot}+k_i}{2m}\right)^2\right] - \left[\underbrace{\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2}_{\text{Modularity of C}} - \underbrace{\left(\frac{k_i}{2m}\right)^2}_{\text{Modularity of } i}\right]$$

Self-edge weight



Edge weight of the resulting super-node from merging C and *i*

rest of the graph (modeled as a single node)

By applying the Modularity definition:

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

Louvain: Modularity Gain

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

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

- 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 communities obtained in the first phase are contracted into **super-nodes**, and the network is created accordingly:
  - Super-nodes are connected if there is at least one edge between the 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 communities
- **Phase 1 is then run on the super-node network**

Louvain Algorithm

Algorithm 1: Sequential Louvain Algorithm

Input: $G=(V,E)$: graph representation.
Output: C: community sets at each level;
Q: modularity at each level.
Var: $\hat{c}$: vertex u's best candidate community set.

1 **Loop** ***outer***
2 $C \leftarrow \{\{u\}\}, \forall u \in V$;
3 $\Sigma_{in}^{c} \leftarrow \sum w_{u,v},\ e(u,v) \in E,\ u \in c$ and $v \in c$;
4 $\Sigma_{tot}^{c} \leftarrow \sum w_{u,v},\ e(u,v) \in E,\ u \in c$ or $v \in c$;
5 *// Phase 1.*
6 **Loop** ***inner***
7 **for** $u \in V$ *and* $u \in c$ **do**
8 *// Find the best community for vertex u.*
9 $\hat{c} \leftarrow \underset{\forall c',\ \exists e(u,v) \in E,\ v \in c'}{\text{argmax}} \Delta Q_{u \to c'}$; Modularity gain
10 **if** $\Delta Q_{u \to \hat{c}} > 0$ **then**
11 *// Update* Σ_{tot} *and* Σ_{in}.
12 $\Sigma_{tot}^{\hat{c}} \leftarrow \Sigma_{tot}^{\hat{c}} + w(u)$; $\Sigma_{in}^{\hat{c}} \leftarrow \Sigma_{in}^{\hat{c}} + w_{u \to \hat{c}}$;
13 $\Sigma_{tot}^{c} \leftarrow \Sigma_{tot}^{c} - w(u)$; $\Sigma_{in}^{c} \leftarrow \Sigma_{in}^{c} - w_{u \to c}$;
14 *// Update the community information.*
15 $\hat{c} \leftarrow \hat{c} \cup \{u\}$; $c \leftarrow c - \{u\}$;
16 **if** *No vertex moves to a new community* **then**
17 **exit** ***inner*** **Loop**;

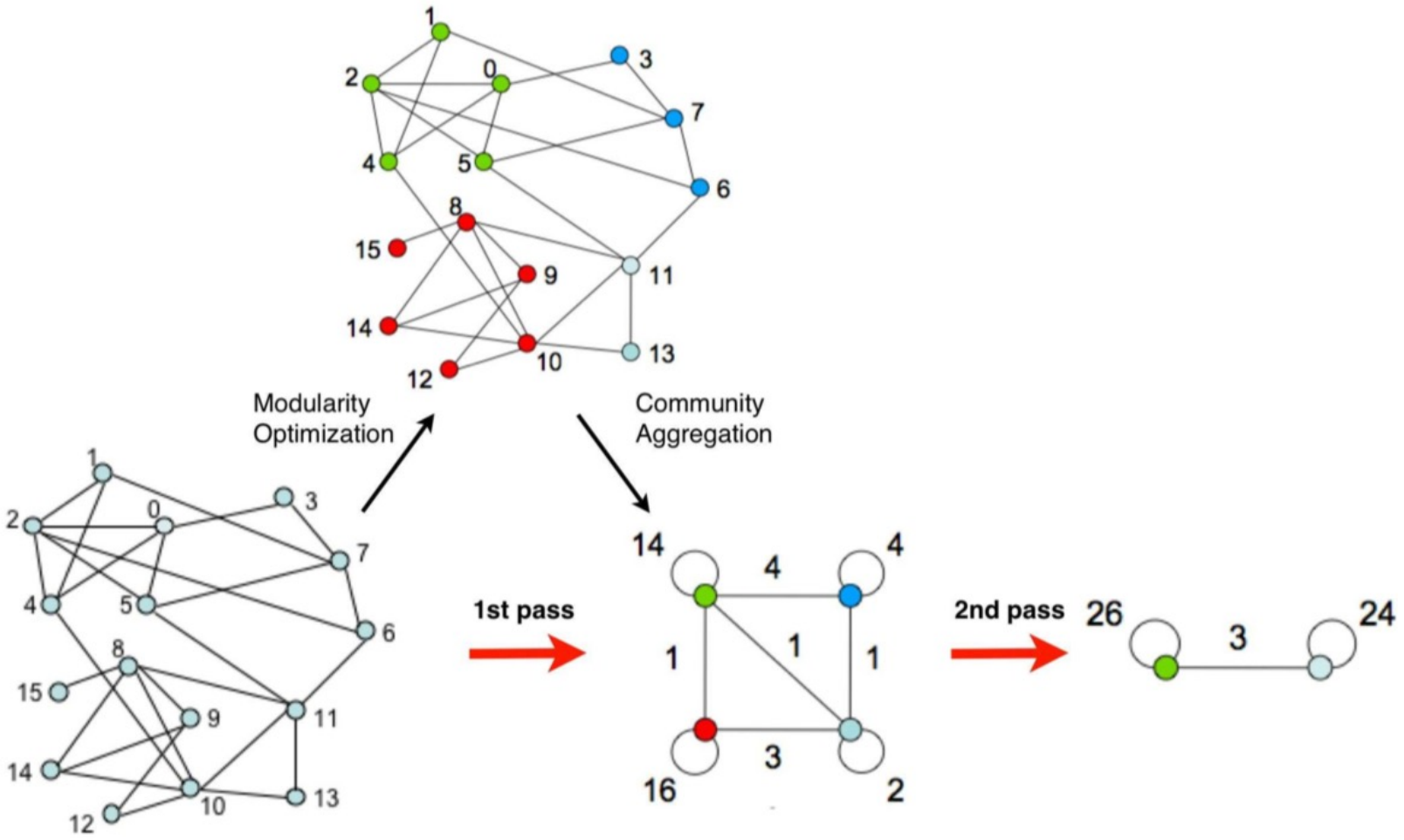
Halting criterion for 1st Phase

18 *// Calculate community set and modularity.*
19 $Q \leftarrow 0$;
20 **for** $c \in C$ **do**
21 $Q \leftarrow Q + \frac{\Sigma_{in}^{c}}{2m} - (\frac{\Sigma_{tot}^{c}}{2m})^2$;
22 $C' \leftarrow \{c\}, \forall c \in C$; print C' and Q ;
23 *// Phase 2: Rebuild Graph.*
24 $V' \leftarrow C'$; Communities contracted into super-nodes
25 $E' \leftarrow \{e(c,c')\},\ \exists e(u,v) \in E,\ u \in c,\ v \in c'$;
26 $w_{c,c'} \leftarrow \sum w_{u,v},\ \forall e(u,v) \in E,\ u \in c,\ v \in c'$;
27 **if** *No community changes* **then**
28 **exit** ***outer*** **Loop**;
29 $V \leftarrow V'$; $E \leftarrow E'$;

Halting criterion for 2nd Phase

the weights of the edges between the new super-nodes are given by the **sum of the weights of the edges** between vertices in the corresponding two communities

Louvain Algorithm



Back to Detecting Cell Types

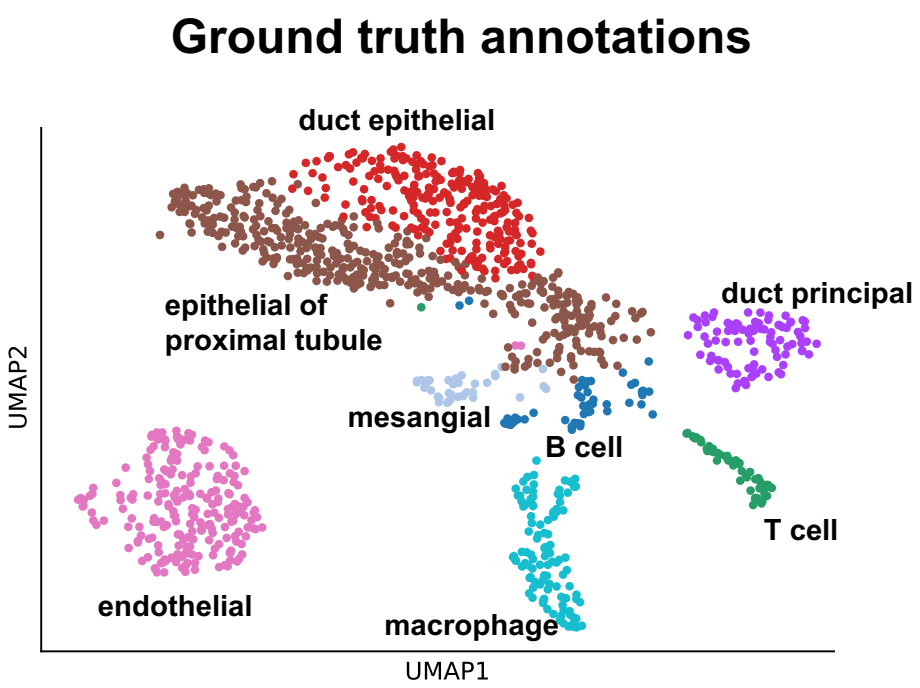
Input:

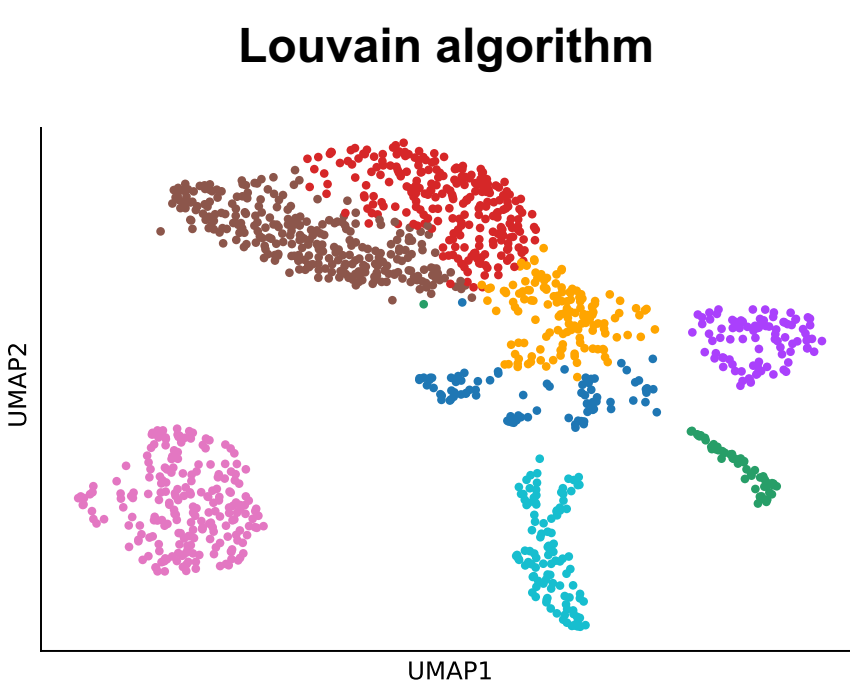
Single-cell gene expression data

Steps:

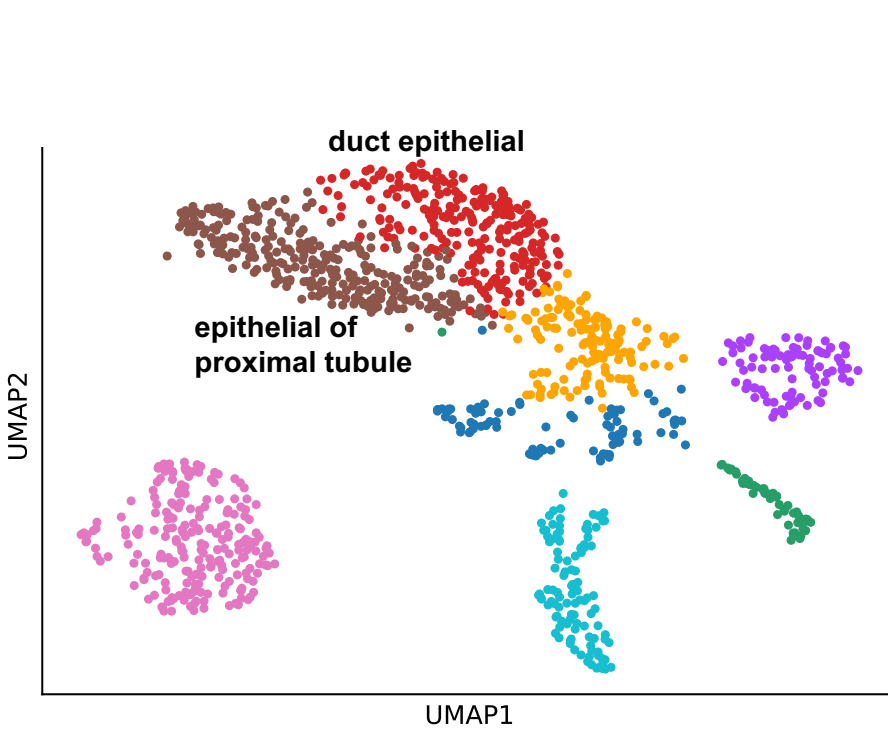
- 1) Apply SVD to cell gene expression data (~50 dim)
- 2) Create K-NN (K=15) graph between the low-dim cell gene expressions
- 2) Apply the Louvain algorithm to identify the clusters

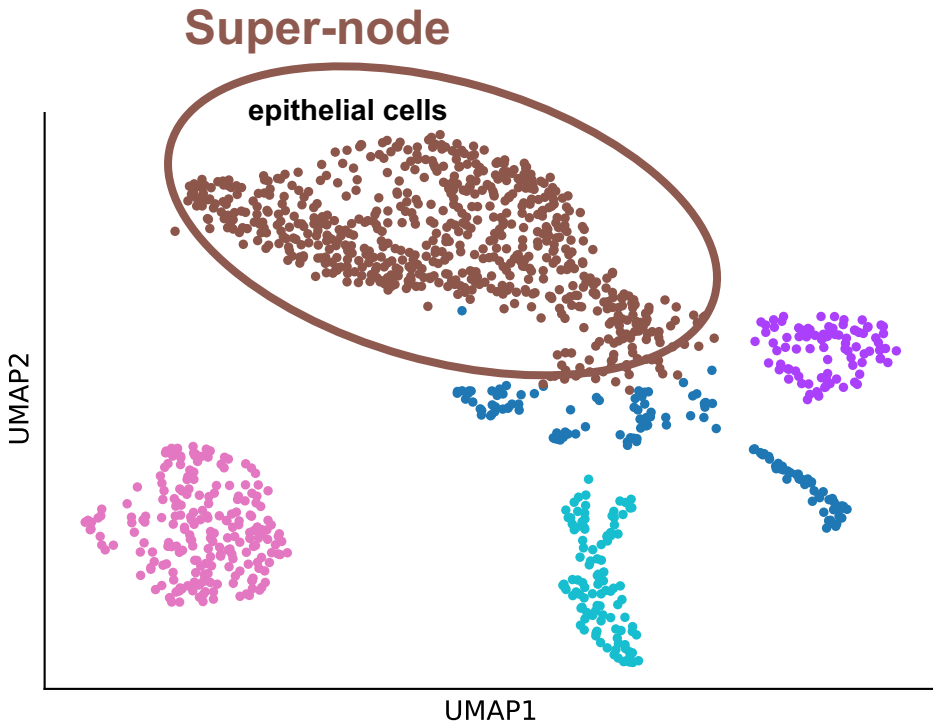
Cell Type Identification Task





Louvain Hierarchical Groups





Summary: Modularity

- **Modularity:**
  - Overall quality of the partitioning of a graph into communities
  - Used to determine the number of communities
- **Louvain modularity maximization:**
  - Greedy strategy
  - Great performance, scales to large networks