Graph clustering

What is Graph Clustering?

Types of Graph Clustering:

– Between-graph

Clustering a set of graphs: Between-graph clustering methods divide a set of graphs into different clusters

– Within-graph

Clustering the nodes/edges of a single graph: Within-graph clustering methods divides the nodes of a graph into clusters

Note: ***In this lecture we will look at different algorithms to perform within-graph clustering***

Network Partition

- Network Partition
  - Finding modules of the network.
- Graph Clustering
  - Partition graphs according to the connectivity.
  - Nodes within a cluster is highly connected
  - Nodes in different clusters are poorly connected.

Applications

- It can be applied to regular clustering
  - Each object is represented as a node
  - Edges represent the similarity between objects
- Bioinformatics
  - Partition genes, proteins
- Web pages
  - Communities discoveries

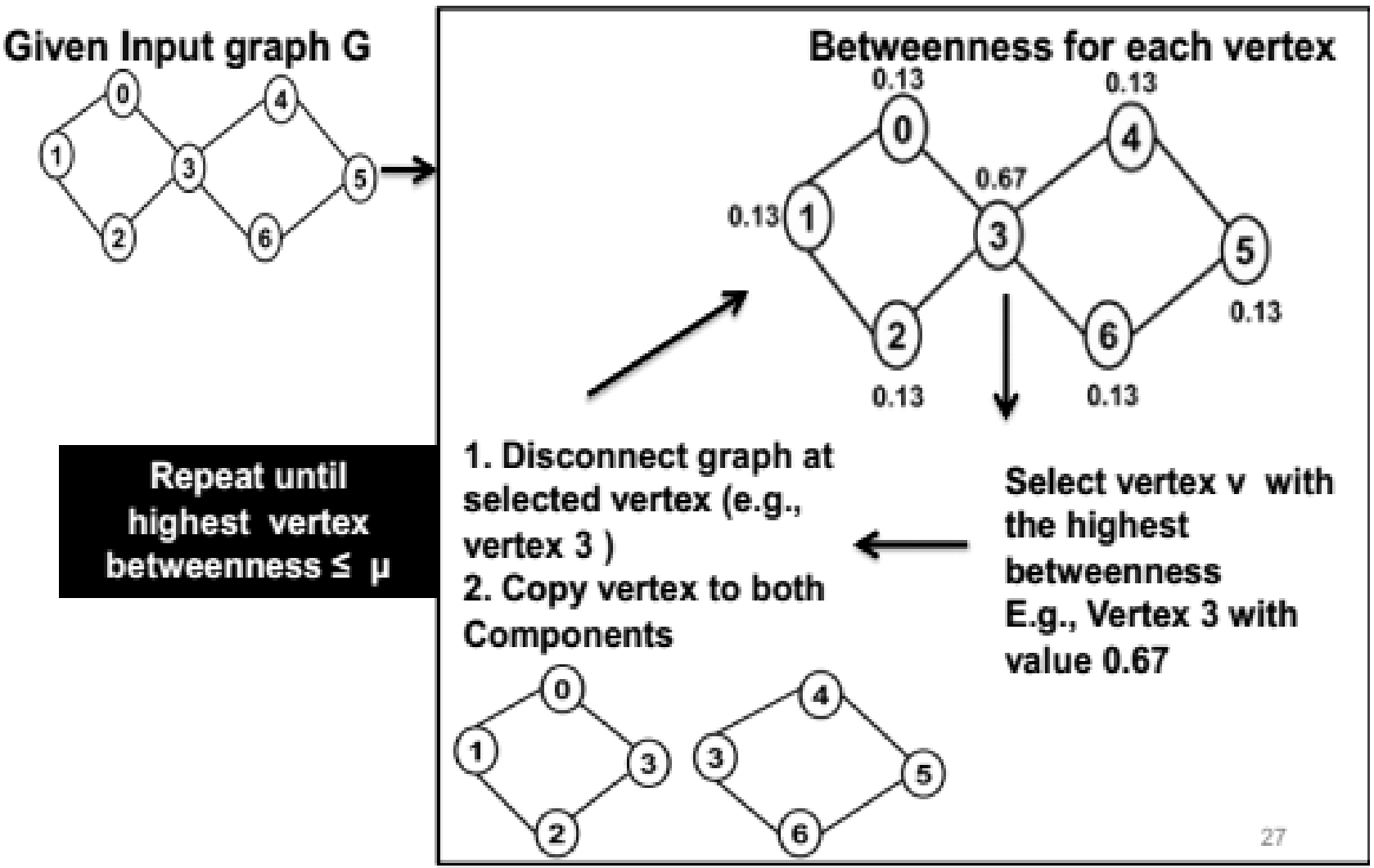
Pervious Work

- Graph Partitioning
  - Minimum Cuts
  - Spectral Partitioning
- Applications:
  - Parallel computing
  - VLSI design and other CAD applications

Pervious Work

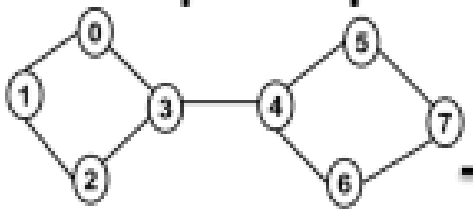
- Community detection
  - Block Modeling: Best fits to stochastic models
  - Hierarchical Clustering based on single or average linkage clustering
  - Community Structure Detection with Betweenness-based Methods

Vertex Betweenness Clustering

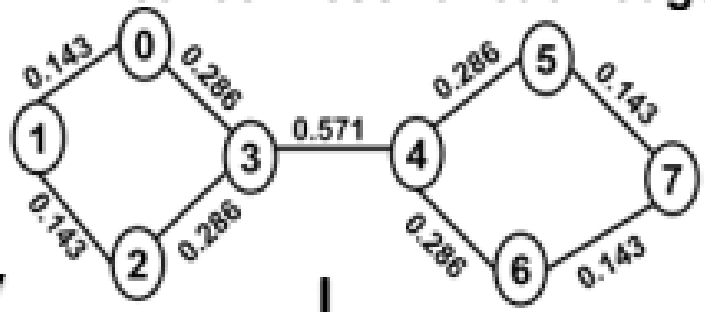


Edge-Betweenness Clustering
Girvan and Newman Algorithm

Given Input Graph G

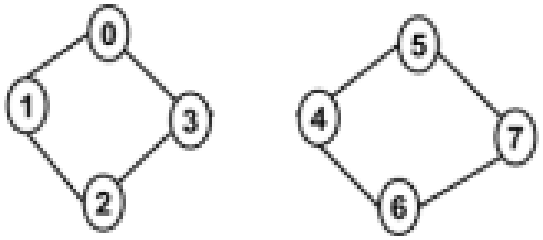


Betweenness for each edge



Select edge with Highest Betweennes:
E.g., edge (3,4) with value 0.571

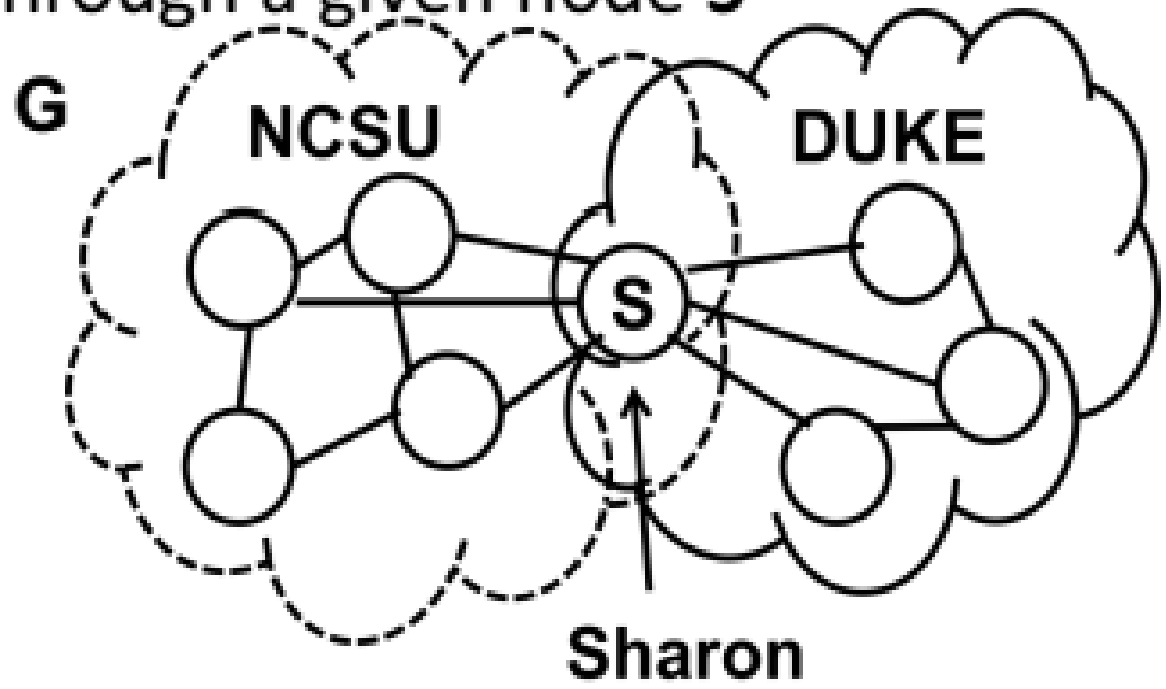
Disconnect graph at selected edge (E.g., (3,4))



Repeat until highest edge betweenness ≤ μ

Vertex Betweenness

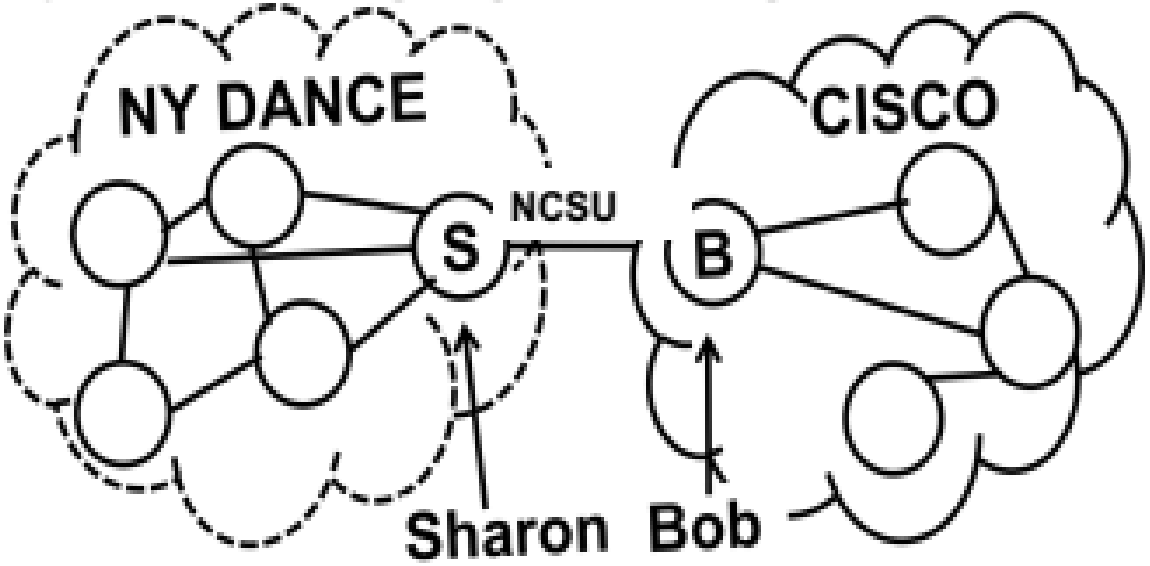
The number of **shortest paths in the graph G** that pass through a given node **S**



E.g., Sharon is likely a liaison between NCSU and DUKE and hence many connections between DUKE and NCSU pass through Sharon

Edge Betweenness

The number of **shortest paths in the graph G** that pass through given edge (S, B)

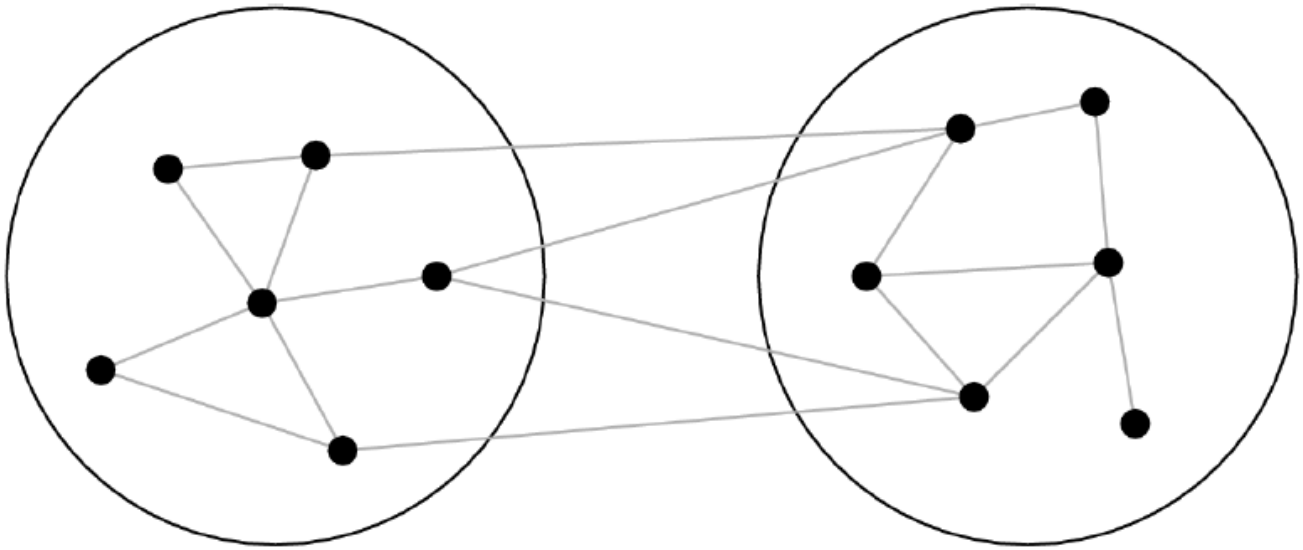


E.g., Sharon and Bob both study at NCSU and they are the only link between NY DANCE and CISCO groups

Vertices and Edges with high Betweenness form good starting points to identify clusters

Graph Partitioning

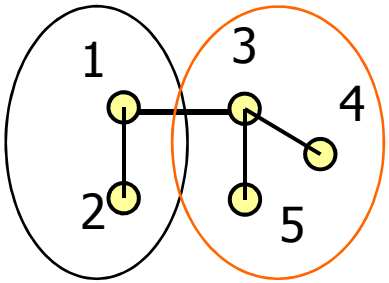
- Graph partitioning algorithms are typically based on *minimum cut approaches or spectral* partitioning *:*



Spectral bisection

- Eigen-vectors of the graph Laplacian.
- L = D-A
- A is the adjacency matrix
- D is a diagonal Matrix of vertex degrees

Bisect !



$$L = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 3 & -1 & -1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.45 & 0.34 & 0 & -0.70 & 0.44 \\ 0.45 & 0.70 & 0 & 0.54 & -0.14 \\ 0.45 & -0.20 & 0 & -0.32 & -0.81 \\ 0.45 & -0.42 & -0.71 & 0.24 & 0.26 \\ 0.45 & -0.42 & 0.71 & 0.24 & 0.26 \end{bmatrix}$$

$$E = [0.00, 0.52, 1.00, 2.31, 4.17]$$

The eigenvector corresponding to the lowest eigenvalue must have both positive and negative elements.

Spectral Bisection (Cont.)

- It only bisects graphs into only 2 communities.
- Division into a larger number of communities is usually achieved by repeated bisection, but this does not always give satisfactory results.
- We do not in general know ahead of time how many communities we want to divide the graph into.

Graph Partitioning

- Minimum cut partitioning breaks down when we don't know the sizes of the groups

  - Optimizing the cut size with the **groups sizes free** puts all vertices in the same group

- *Cut size is the **wrong** thing to optimize*

  - A good division into communities is not just one where there are a small number of edges between groups

- There must be a *smaller than* **expected number** edges between communities

Modularity

Other Approaches:-

- Greedy Algorithm: Start with all the vertices in separate communities.

  - Find the two communities whose amalgamation gives the greatest increase in the modularity
- Simulated annealing (Guimera & Amaral 2005)
- External Optimization(Dutch & Arenas 2005)

Modularity (Newman and Girvan 2004)

Define modularity to be

Q = (number of edges within groups) – (expected number within groups).

Actual Number of Edges between i and j is

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge } (i,j), \\ 0 & \text{otherwise.} \end{cases}$$

Expected Number of Edges between i and j is

$$\text{Expected number} = \frac{k_i k_j}{2m}. \qquad m = \frac{1}{2}\sum_i k_i$$

Modularity Matrix

- So Q is a sum of $A_{ij} - \frac{k_i k_j}{2m}$ (s_i, s_j)

over pairs (i, j) that are in the same group

- Or we can write in matrix form as

$$Q = \mathbf{s}^T \mathbf{B} \mathbf{s},$$

Where **s** is a the vector whose elements are s_i

Where B is a new characteristic matrix, the *modularity marix*,

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m}.$$

Modularity Matrix

s is the linear combination of the normalized eigenvectors u_i of B

Write $\mathbf{s} = \sum_i a_i \mathbf{u}_i$ where $a_i = \mathbf{u}_i^T \mathbf{s}$ and then

$$Q = \mathbf{s}^T \mathbf{B}\mathbf{s} = \sum_i a_i^2 \beta_i .$$

β_i is the eigenvalue of B corresponding to eigenvector u_i

To maximize Q we want to place as much weight as possible in the terms corresponding to the *largest* eigenvalues.

- We maximize the coefficient on the largest eigenvalue by choosing

$$s_i = \begin{cases} -1 & \text{if } i\text{th element of } \mathbf{u}_1 \geq 0, \\ -1 & \text{if } i\text{th element of } \mathbf{u}_1 < 0. \end{cases}$$

Modularity Matrix

Algorithm

- Calculate the leading eigenvector of the modularity matrix
- Divide the vertices according to the signs of the elements

Note that there is no need to forbid the solution with all the vertices in a single group.

Example

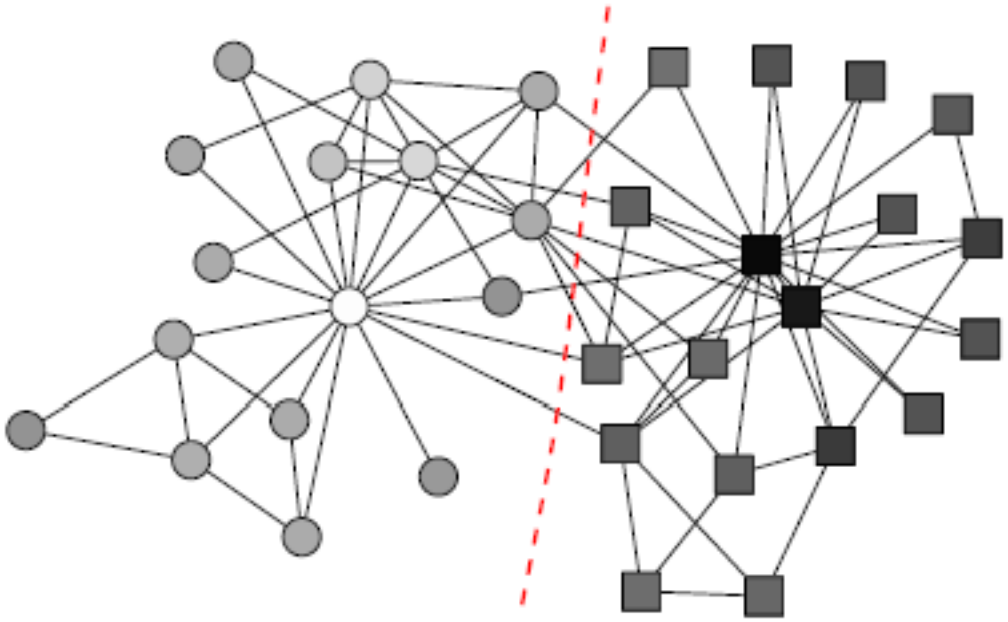


FIG. 2: Application of our eigenvector-based method to the "karate club" network of Ref. [23]. Shapes of vertices indicate the membership of the corresponding individuals in the two known factions of the network while the dotted line indicates the split found by the algorithm, which matches the factions exactly. The shades of the vertices indicate the strength of their membership, as measured by the value of the corresponding element of the eigenvector.

Spectral properties of modularity matrix

- Vector(1,1,1,…) is always an eigenvector of B with eigenvalue zero
- Eigenvalues can either be positive or negative

- So long as there is *any* positive eigenvalue we will never put all vertices in the same group

- But there may be no positive eigenvalues

- All vertices in same group gives highest modularity

- Such networks are *indivisible*

Dividing into more than two groups

- Repeated division into two groups

  - Divide into two, then divide those parts into two,etc
- Stop when there is no division that will increase the modularity

  - This is precisely when the subgraph is indivisible

  - Stop when there are no positive eigenvalues of the modularity matrix

Modularity Matrix

- Time Complexity

  $O(n^2 \log n)$
- Better than

  Betweenness Algorithm $O(n^3)$

  External Optimization $O(n^2 \log^2 n)$
- Not as good as

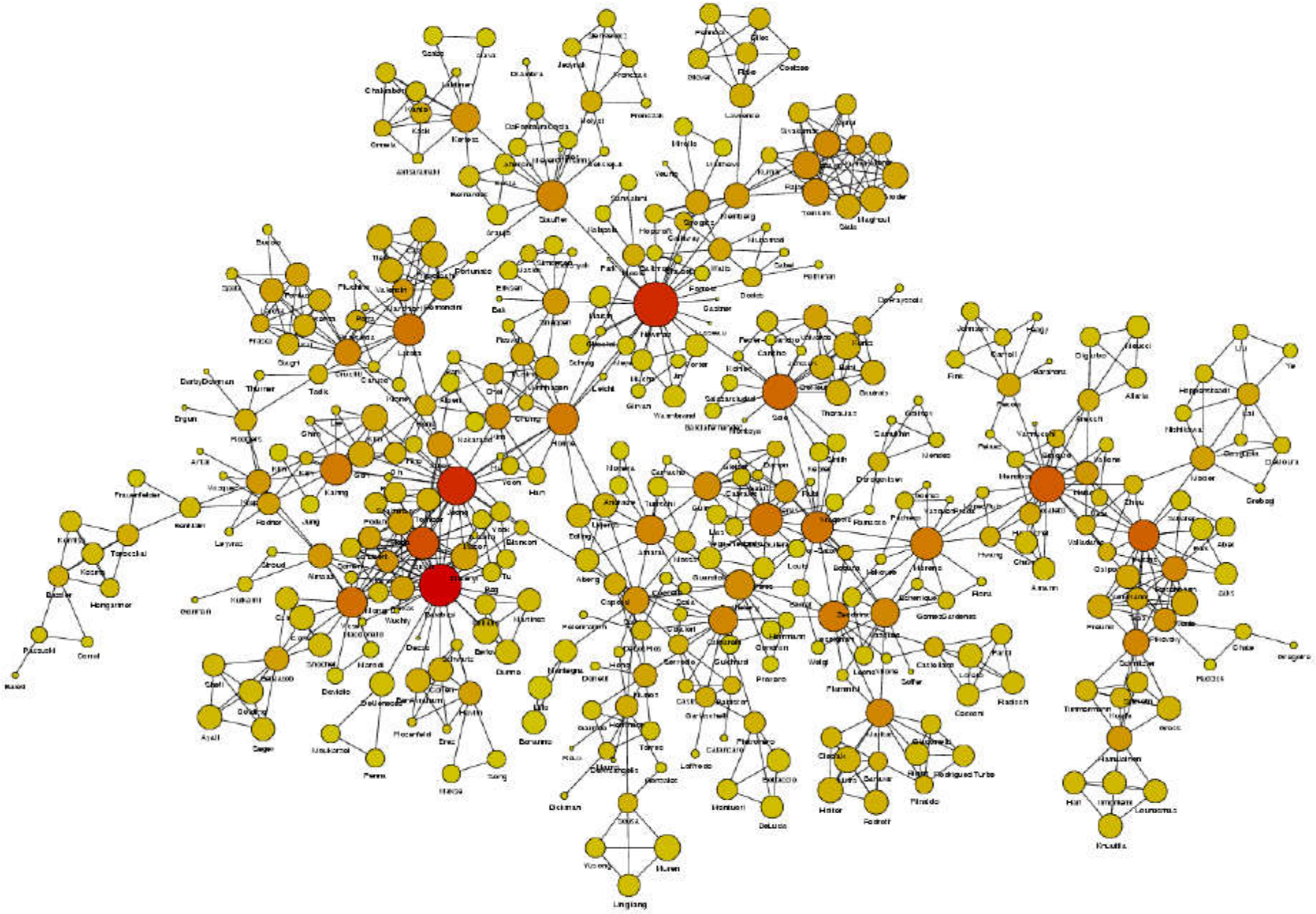
  Greedy Algorithm $O(n \log^2 n)$ but better quality results

Modularity Matrix

- Actual Running Time

  Collaboration network of about 27000 vertices, the algorithm takes around 20 minutes to run on a standard personal computer.

Example: Collaboration network



Example Applications

- Books on politics

  The vertices represent 105 recent books sold from Amazon.com

  Divide the books according to their political alignment

  Liberal / Conservative / Centrist

Example

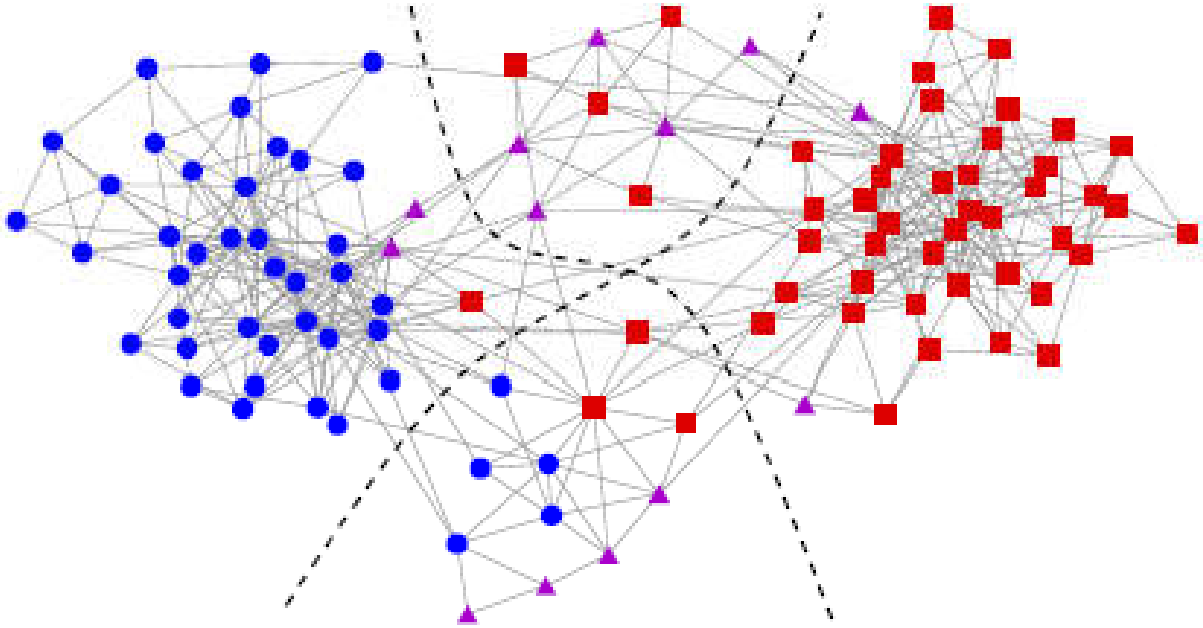


FIG. 3: Krebs' network of books on American politics. Vertices represent books and edges join books frequently purchased by the same readers. Dotted lines divide the four communities found by our algorithm and shapes represent the political alignment of the books: circles (blue) are liberal, squares (red) are conservative, triangles (purple) are centrist or unaligned.

Comparison to other methods

network	size n	modularity Q GN	CNM	DA	this paper
karate	34	0.401	0.381	0.419	0.419
jazz musicians	198	0.405	0.439	0.445	0.442
metabolic	453	0.403	0.402	0.434	0.435
email	1133	0.532	0.494	0.574	0.572
key signing	10 680	0.816	0.733	0.846	0.855
physicists	27 519	–	0.668	0.679	0.723

TABLE I: Comparison of modularities for the network divisions found by the algorithm described here and three other previously published methods as described in the text, for six networks of varying sizes. The networks are, in order, the

CN = Betweenness
CNM = Greedy
DA = External Optimization

Summary

- Modularity maximization appears to be a highly competitive approach to community detection in networks
- It can be formulated as a spectral optimization problem, which leads to fast and accurate algorithms
- There are close connections between the spectrum of the modularity matrix and the community structure

References

- Modularity and Community Structure in Networks – MEJ Newman
- Detecting community structure in network, M. E. J. Newman.
- Finding community structure in very large networks, Aaron Clauset, M. E. J. Newman, and Cristopher Moore.

Challenges of Graph Partition

- Graph may be large
  - Large number of nodes
  - Large number of edges
  - Unknown number of clusters
  - Unknown cut-off threshold
- Intuition:
  - High connected nodes could be in one cluster
  - Low connected nodes could be in different clusters.

A Partition Method based on Connectivities

- Cluster analysis seeks grouping of elements into subsets based on similarity between pairs of elements.
- The goal is to find disjoint subsets, called clusters.
- Clusters should satisfy two criteria:
  - Homogeneity
  - Separation

Introduction

- In **similarity** graph data vertices correspond to elements and edges connect elements with similarity values above some threshold.
- Clusters in a graph are highly connected subgraphs.
- Main challenges in finding the clusters are:
  - Large sets of data
  - Inaccurate and noisy measurements

Important Definitions in Graphs

Edge Connectivity:

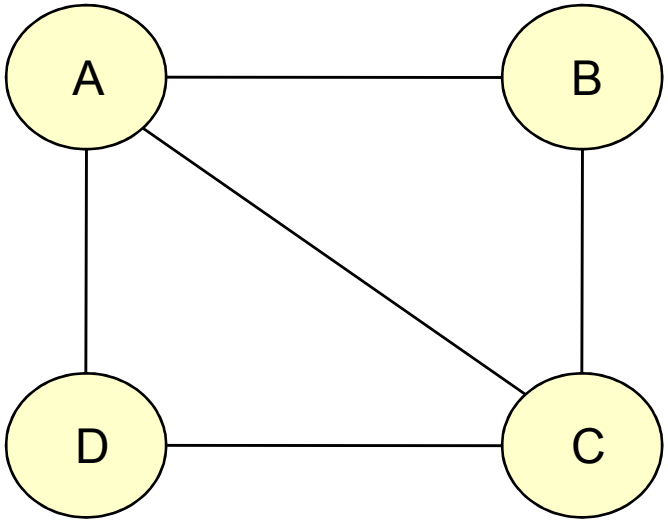
- It is the minimum number of edges whose removal results in a disconnected graph. It is denoted by k(G).

- For a graph G, if k(G) = l then G is called an l-connected graph.

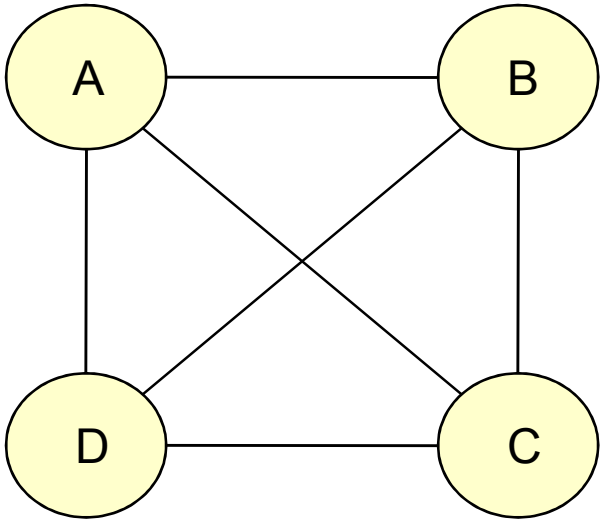
Important Definitions in Graphs

Example:

GRAPH 1



GRAPH 2



The edge connectivity for the GRAPH 1 is 2.

The edge connectivity for the GRAPH 2 is 3.

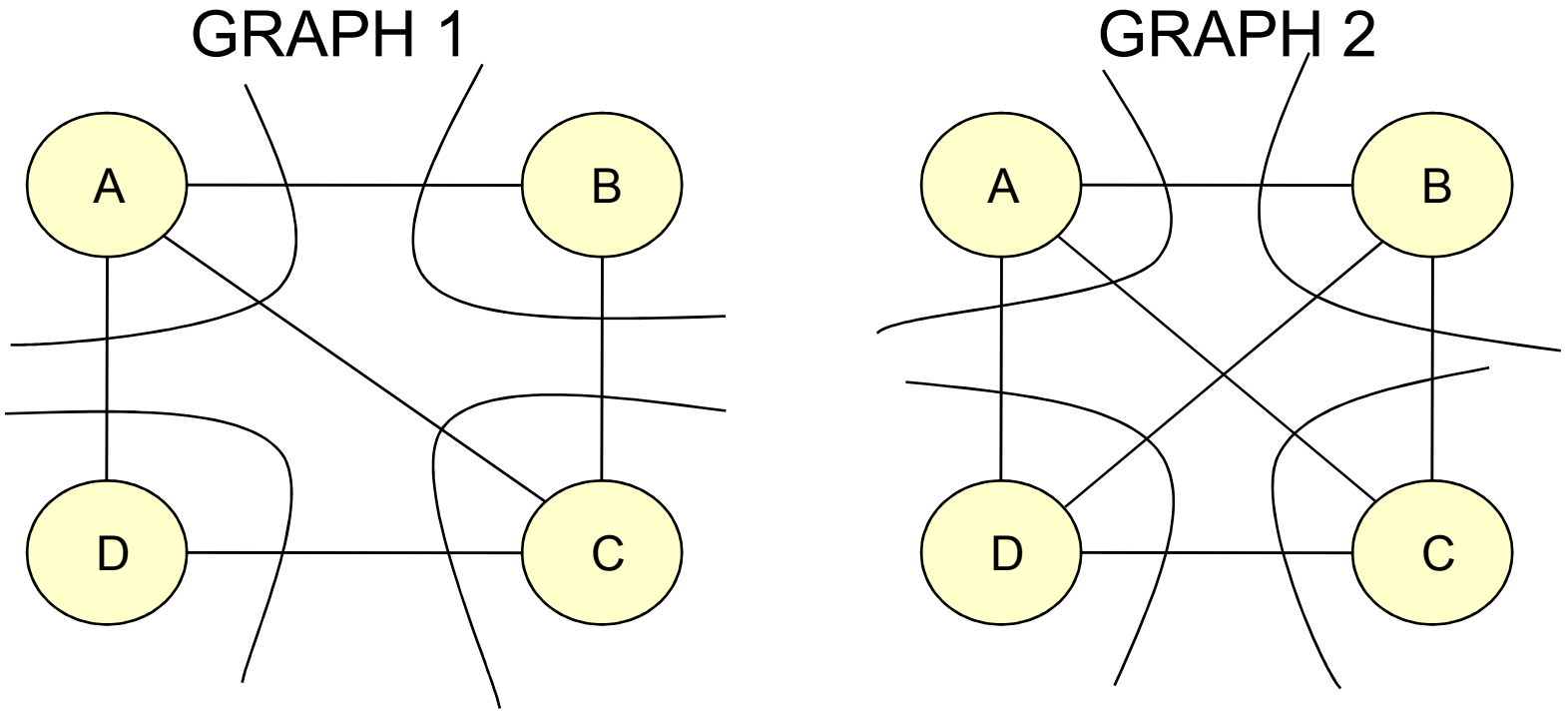
Important Definitions in Graphs

Cut:

- A cut in a graph is a set of edges whose removal disconnects the graph.
- A minimum cut is a cut with a minimum number of edges. It is denoted by S.
- For a non-trivial graph G iff |S| = k(G).

Important Definitions in Graphs

Example:



The min-cut for GRAPH 1 is across the vertex B or D.

The min-cut for GRAPH 2 is across the vertex A,B,C or D.

Important Definitions in Graphs

Distance d(u,v):

- The distance d(u,v) between vertices u and v in G is the minimum length of a path joining u and v.

- The length of a path is the number of edges in it.

Important Definitions in Graphs

Diameter of a connected graph:

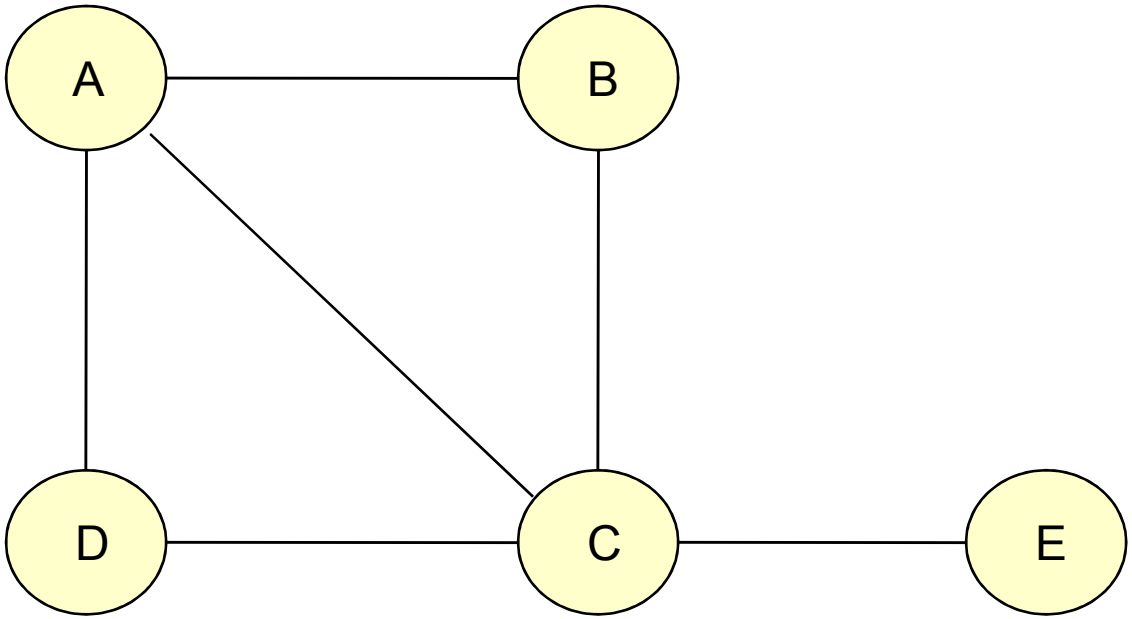
- It is the longest distance between any two vertices in G. It is denoted by diam(G).

Degree of vertex:

- Its is the number of edges incident with the vertex v. It is denoted by deg(v).
- The minimum degree of a vertex in G is denoted by delta(G).

Important Definitions in Graphs

Example:



d(A,D) = 1 d(B,D) = 2 d(A,E) = 2

Diameter of the above graph = 2

deg(A) = 3 deg(B) = 2 deg(E) = 1

Minimum degree of a vertex in G = 1

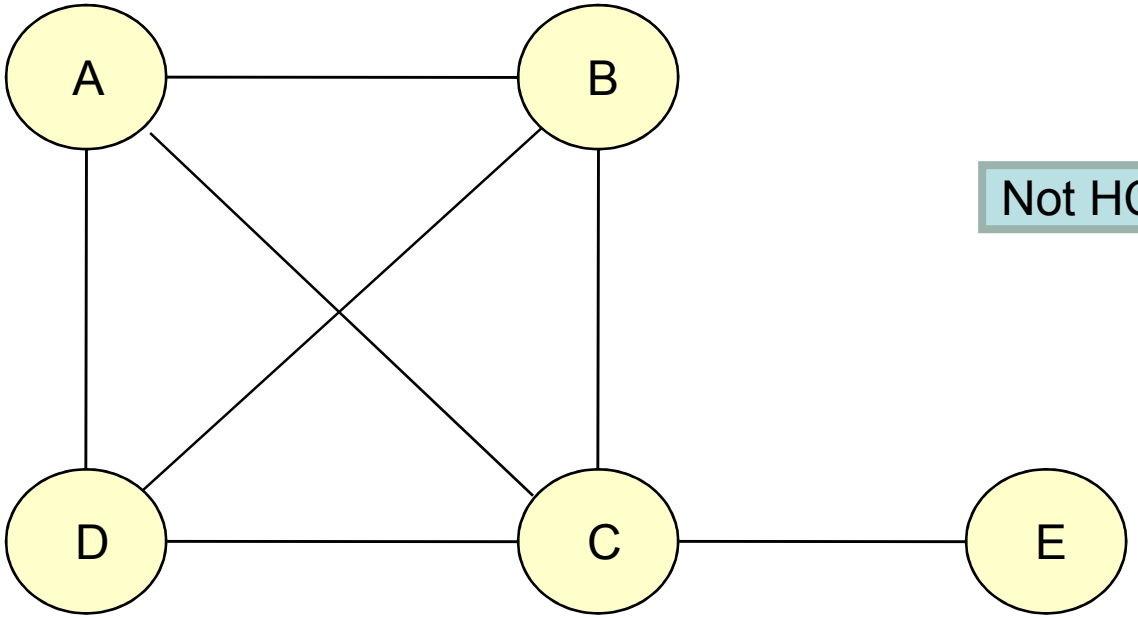
Important Definitions in Graphs

Highly connected graph:

- For a graph with vertices $n > 1$ to be highly connected if its edge-connectivity $k(G) > n/2$.
- A highly connected subgraph (HCS) is an induced subgraph H in G such that H is highly connected.
- HCS algorithm identifies highly connected subgraphs as clusters.

Important Definitions in Graphs

Example:



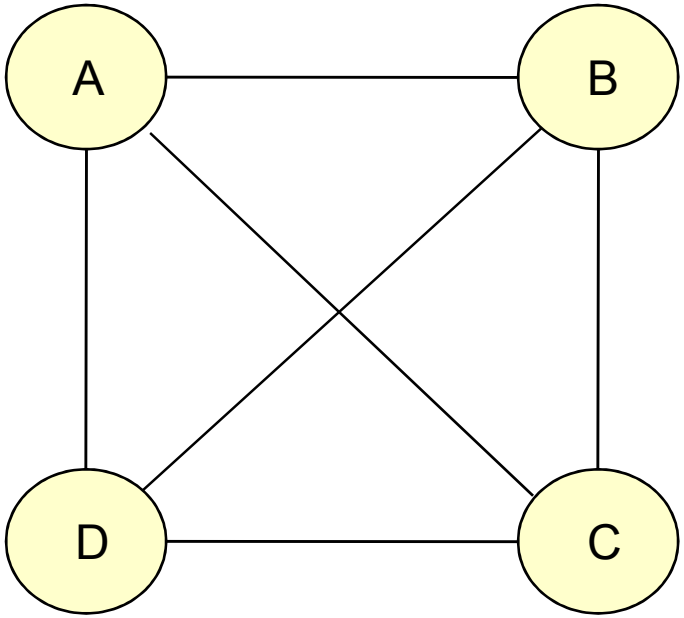
Not HCS!

No. of nodes = 5

Edge Connectivity = 1

Important Definitions in Graphs

Example continued:



HCS!

No. of nodes = 4

Edge Connectivity = 3

HCS Algorithm

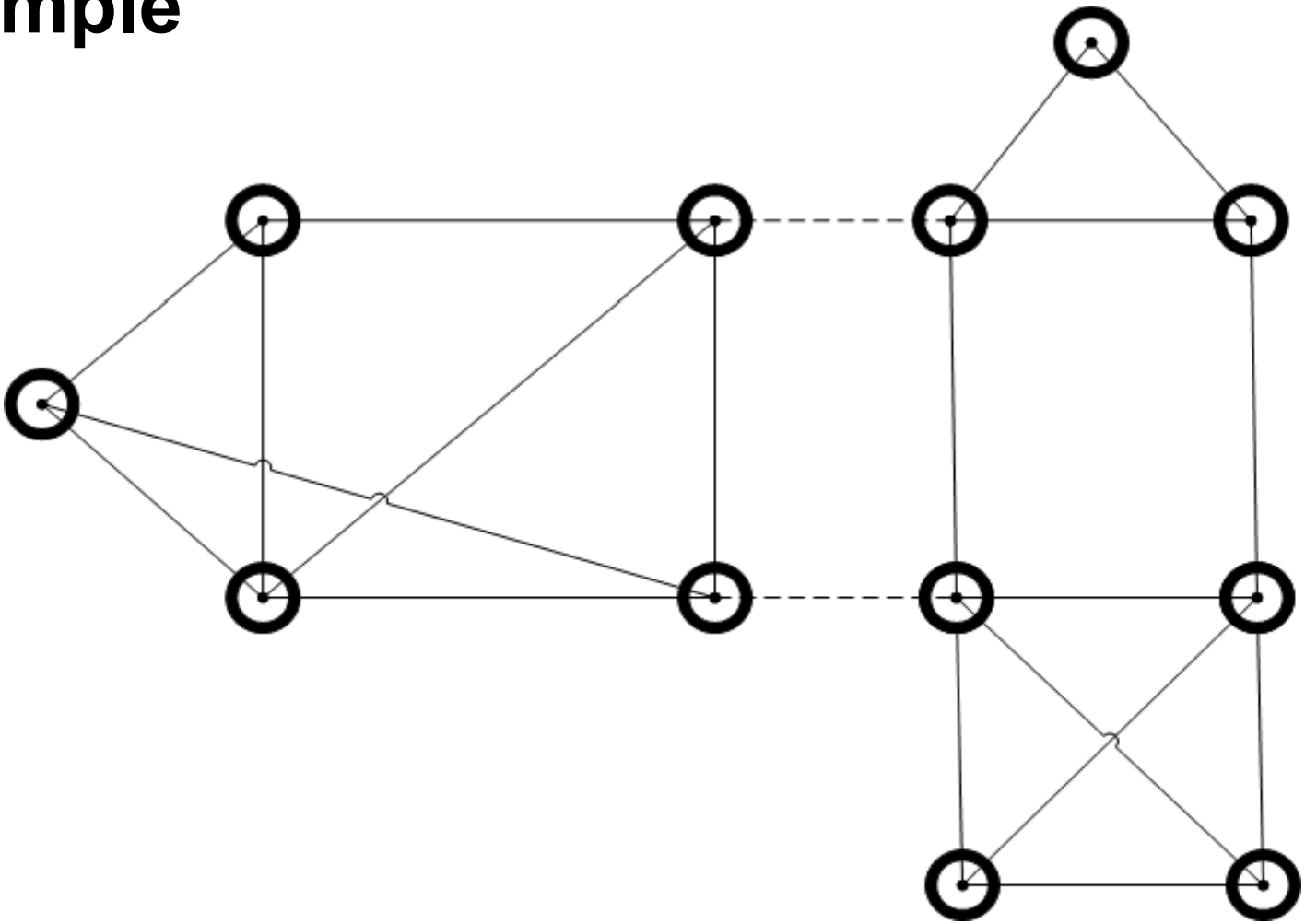
```
HCS(G(V,E))
begin
   (H, H',C) ← MINCUT(G)
   if G is highly connected
        then return (G)
   else
        HCS(H)
        HCS(H')
   end if
end
```

HCS Algorithm

- The procedure MINCUT(G) returns H, H’ and C where C is the minimum cut which separates G into the subgraphs H and H’.
- Procedure HCS returns a graph in case it identifies it as a cluster.
- Single vertices are not considered clusters and are grouped into singletons set S.

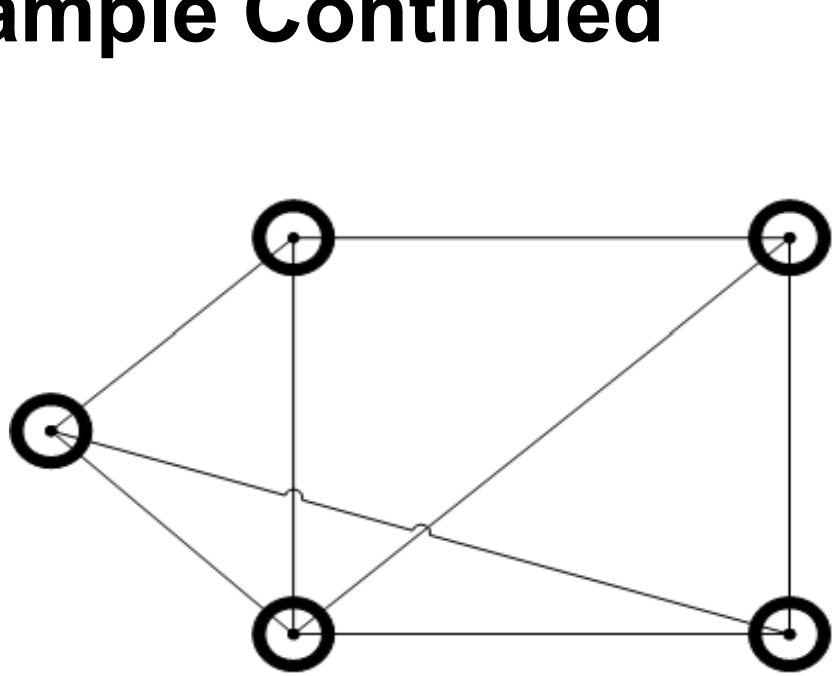
HCS Algorithm

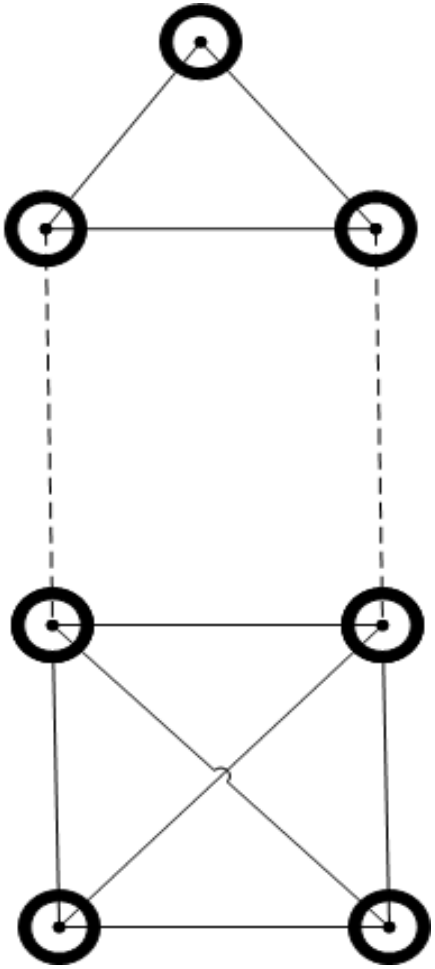
Example



HCS Algorithm

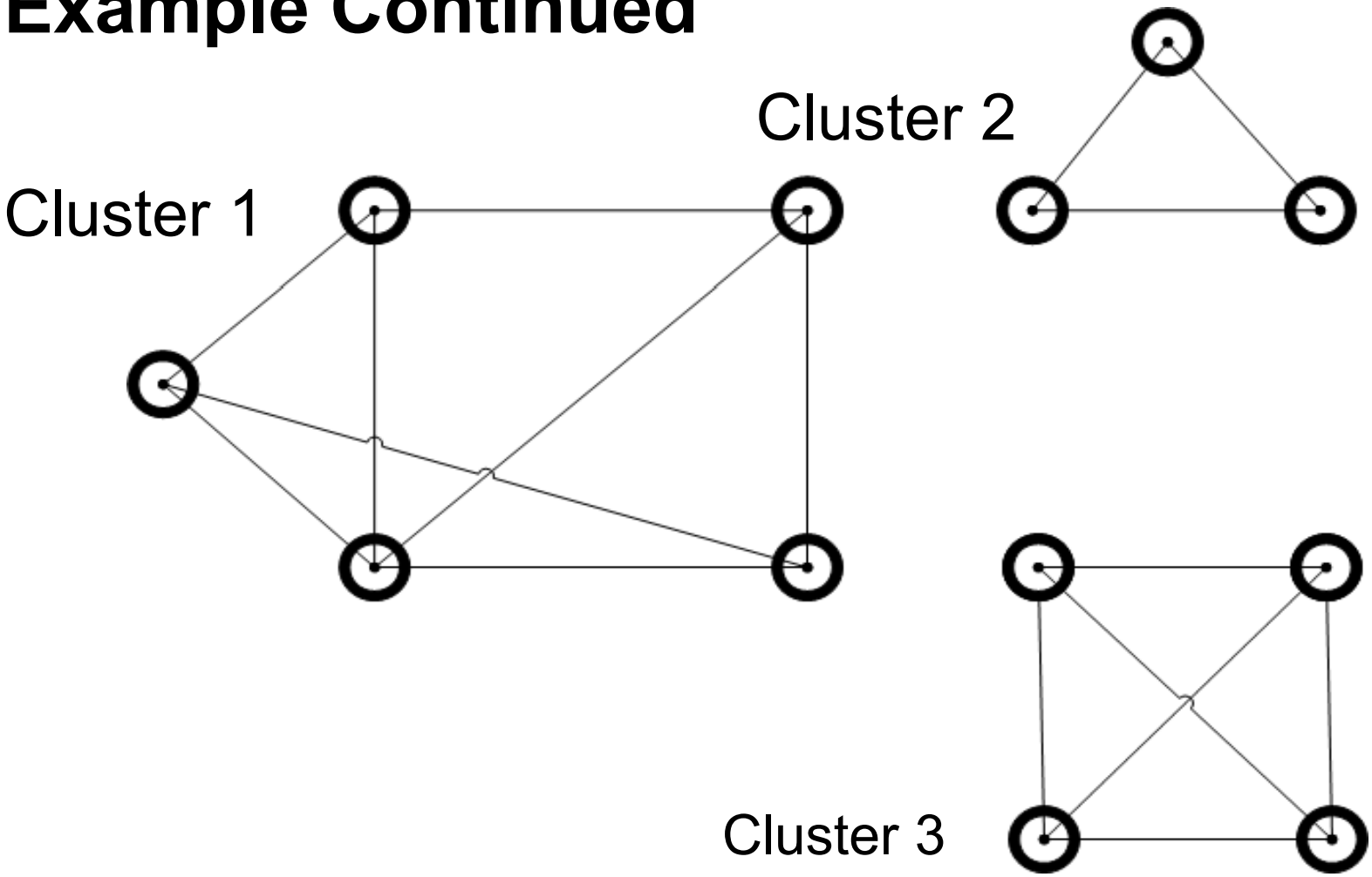
Example Continued





HCS Algorithm

Example Continued



HCS Algorithm

- The running time of the algorithm is bounded by 2N*f(n,m).

  N - number of clusters found

  f(n,m) – time complexity of computing a minimum cut in a graph with n vertices and m edges
- Current fastest deterministic algorithms for finding a minimum cut in an unweighted graph require O(nm) steps.

Properties of HCS Clustering

- Diameter of every highly connected graph is at most two.
- That is any two vertices are either adjacent or share one or more common neighbors.

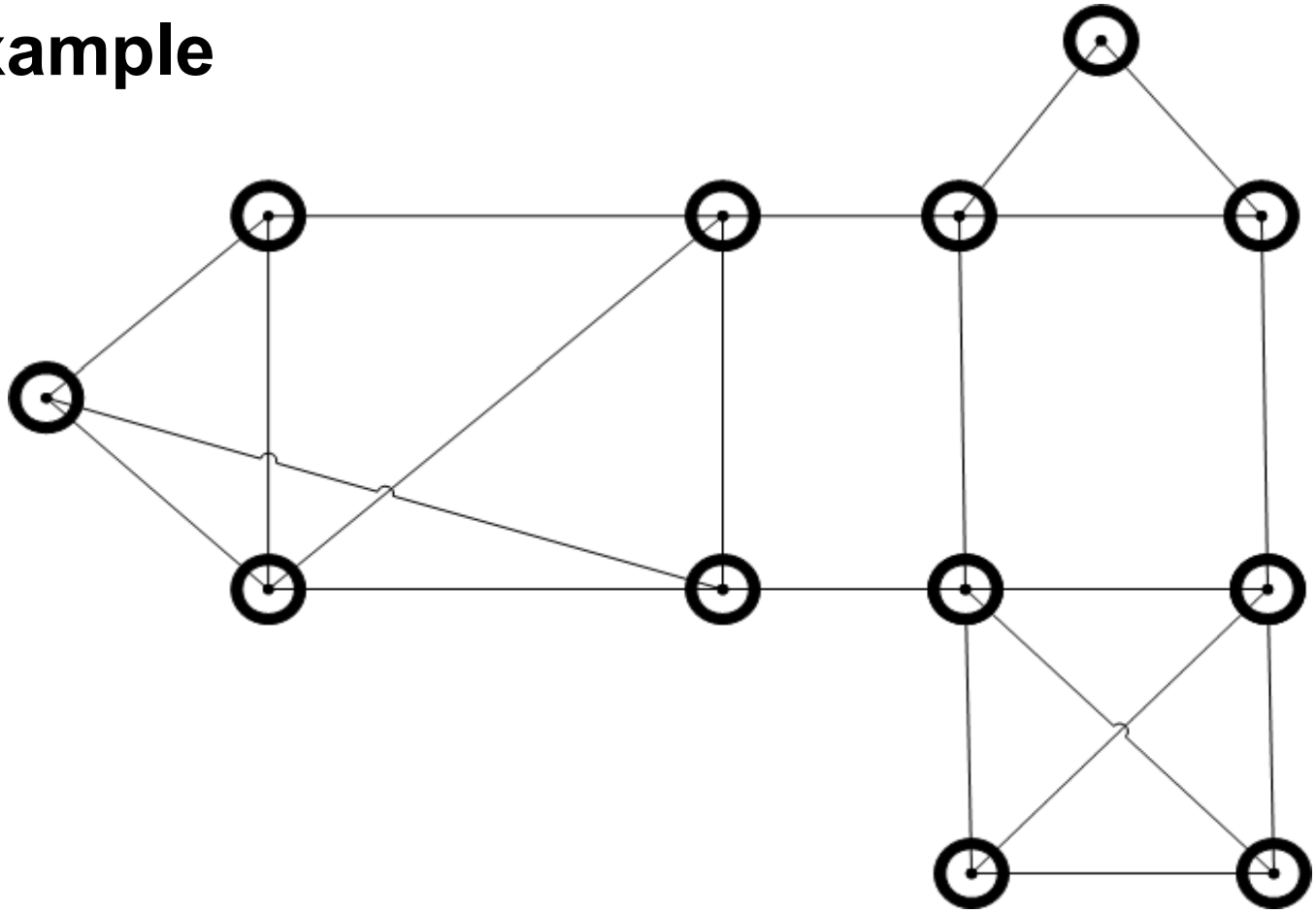
- This is a strong indication of homogeneity.

Properties of HCS Clustering

- Each cluster is at least half as dense as a clique which is another strong indication of homogeneity.
- Any non-trivial set split by the algorithm has diameter at least three.
- This is a strong indication of the separation property of the solution provided by the HCS algorithm.

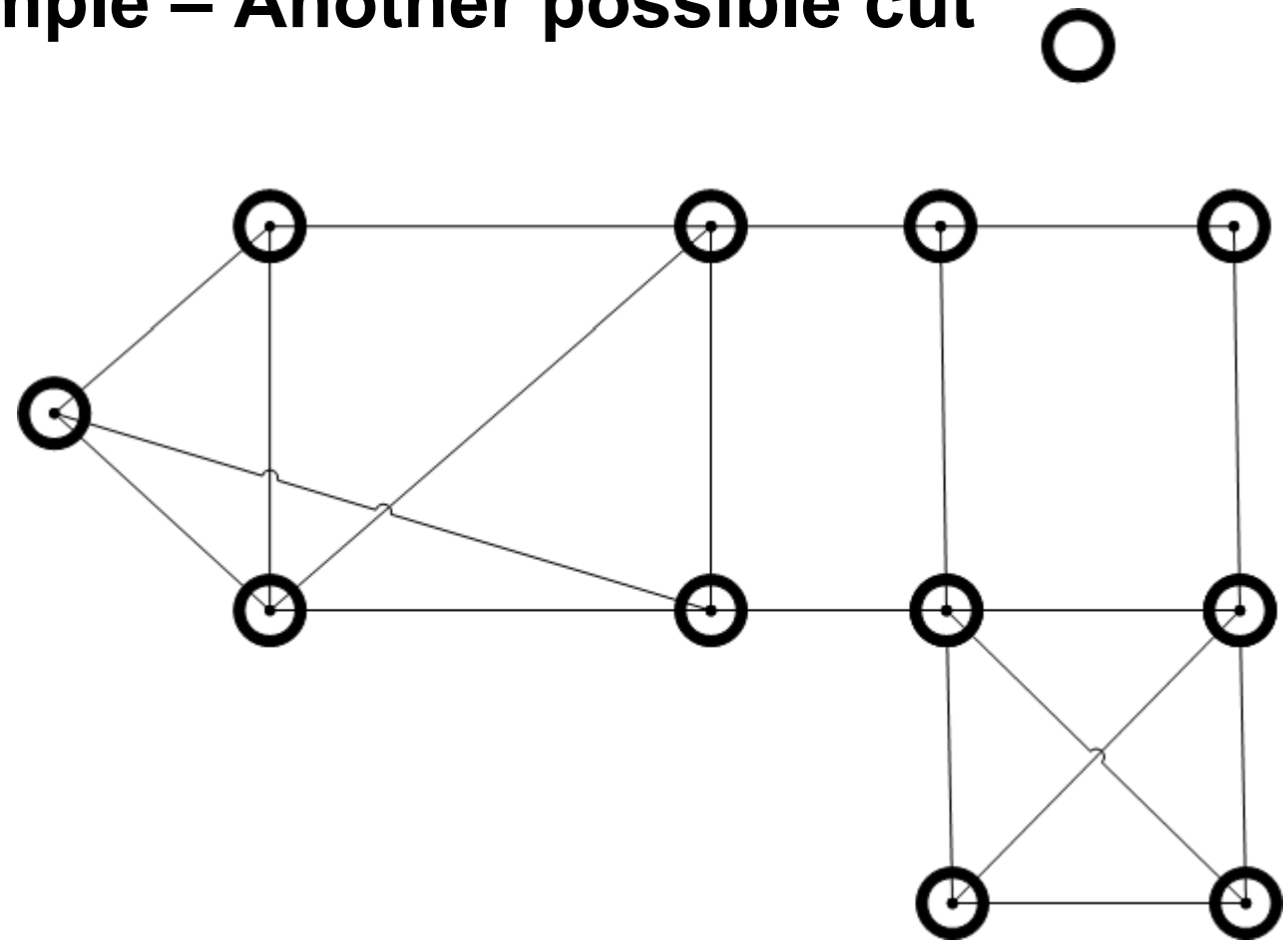
Modified HCS Algorithm

Example



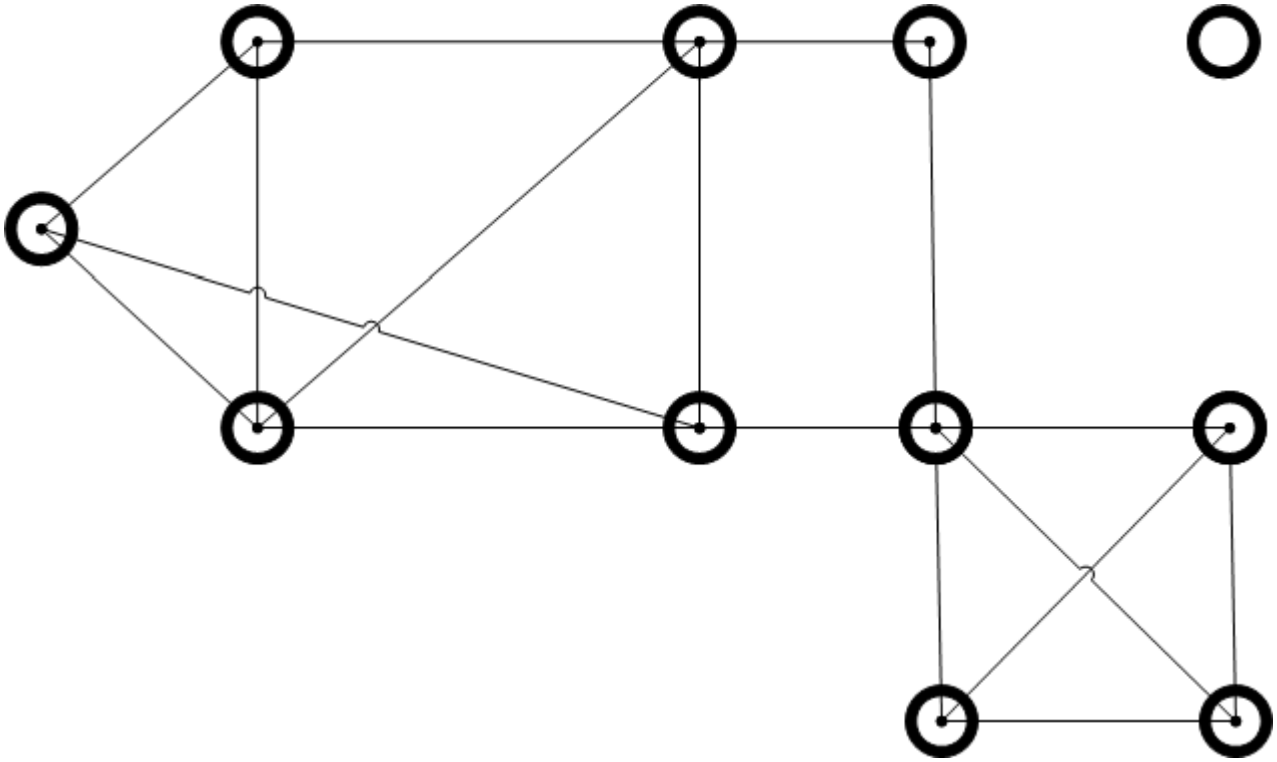
Modified HCS Algorithm

Example – Another possible cut



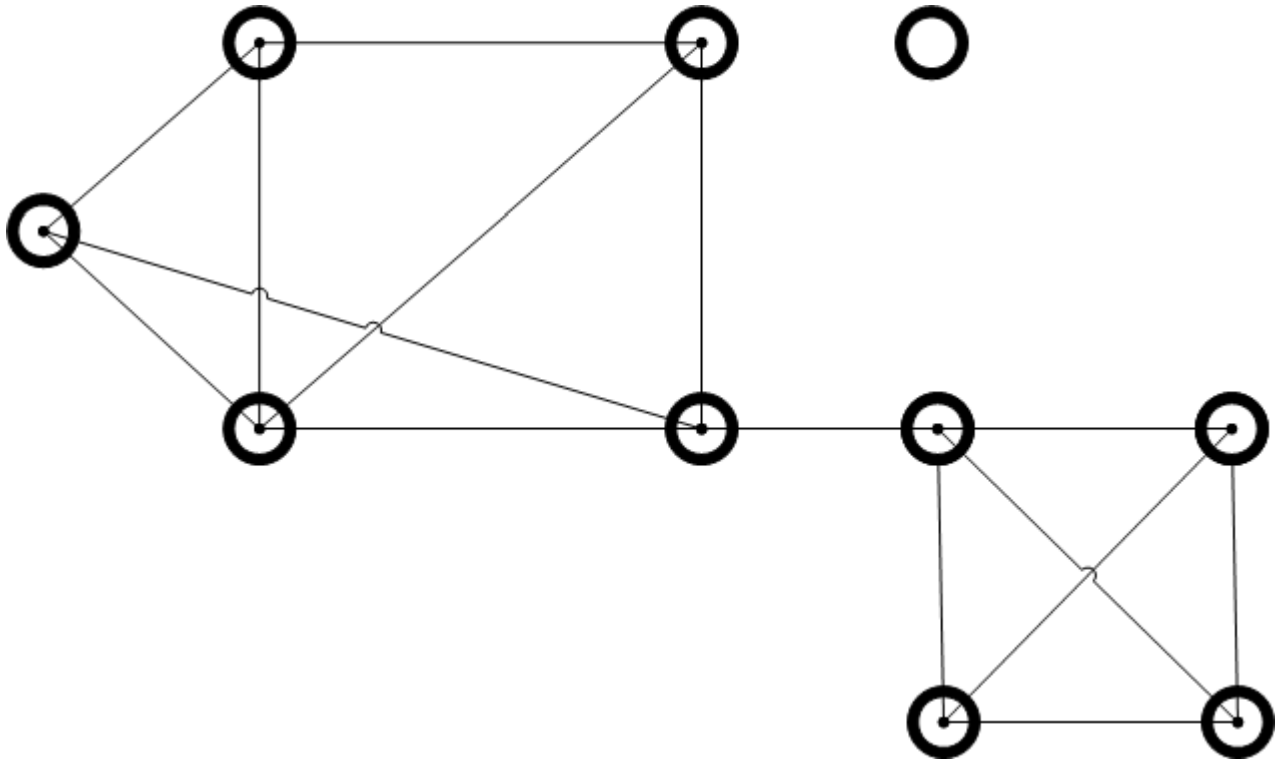
Modified HCS Algorithm

Example – Another possible cut



Modified HCS Algorithm

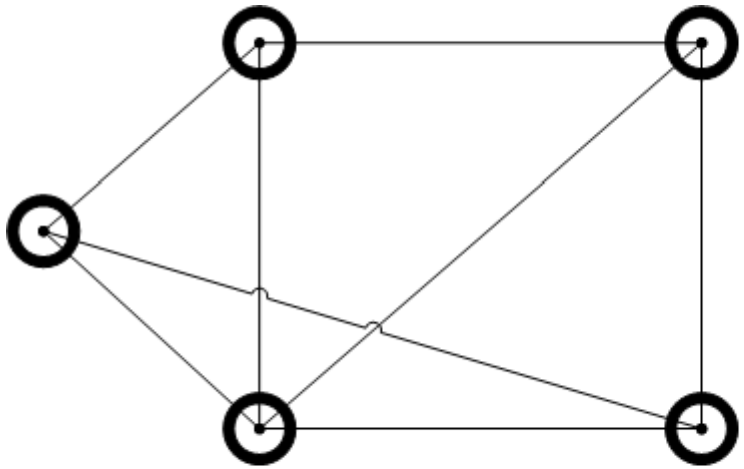
Example – Another possible cut



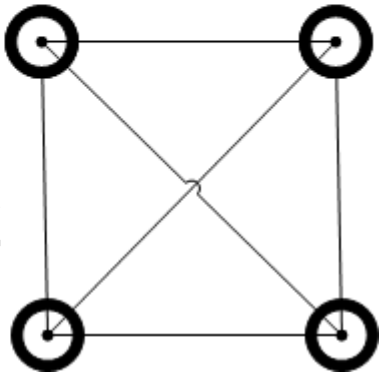
Modified HCS Algorithm

Example – Another possible cut

Cluster 1



Cluster 2



Modified HCS Algorithm

Iterated HCS:

- Choosing different minimum cuts in a graph may result in different number of clusters.
- A possible solution is to perform several iterations of the HCS algorithm until no new cluster is found.
- The iterated HCS adds another O(n) factor to running time.

Modified HCS Algorithm

Singletons adoption:

- Elements left as singletons can be adopted by clusters based on similarity to the cluster.
- For each singleton element, we compute the number of neighbors it has in each cluster and in the singletons set S.
- If the maximum number of neighbors is sufficiently large than by the singletons set S, then the element is adopted by one of the clusters.

Modified HCS Algorithm

Removing Low Degree Vertices:

- Some iterations of the min-cut algorithm may simply separate a low degree vertex from the rest of the graph.
- This is computationally very expensive.
- Removing low degree vertices from graph G eliminates such iteration and significantly reduces the running time.

Modified HCS Algorithm

HCS_LOOP(G(V,E))

begin

for (i = 1 to p) do

remove clustered vertices from G

H ← G

repeatedly remove all vertices of degree < d(i) from H

Modified HCS Algorithm

until(no new cluster is found by the HCS call) do

HCS(H)

perform singletons adoption

remove clustered vertices from H

end until

end for

end

Key features of HCS Algorithm

- HCS algorithm was implemented and tested on both simulated and real data and it has given good results.
- The algorithm was applied to gene expression data.
- On ten different datasets, varying in sizes from 60 to 980 elements with 3-13 clusters and high noise rate, HCS achieved average Minkowski score below 0.2.

Key features of HCS Algorithm

- In comparison greedy algorithm had an average Minkowski score of 0.4.

Minkowski score:

- A clustering solution for a set of n elements can be represented by n x n matrix M.
- M(i,j) = 1 if i and j are in the same cluster according to the solution and M(i,j) = 0 otherwise.
- If T denotes the matrix of true solution, then Minkowski score of M = ||T-M|| / ||T||

Key features of HCS Algorithm

- HCS manifested robustness with respect to higher noise levels.
- Next, the algorithm were applied in a blind test to real gene expression data.
- It consisted of 2329 elements partitioned into 18 clusters. HCS identified 16 clusters with a score of 0.71 whereas Greedy got a score of 0.77.

Summary

- Clusters are defined as subgraphs with connectivity above half the number of vertices
- Elements in the clusters generated by HCS algorithm are homogeneous and elements in different clusters have low similarity values
- Possible future improvement includes finding maximal highly connected subgraphs and finding a weighted minimum cut in an edge-weighted graph.

Graph clustering

- Spectral methods that target weighted cuts form an important class of such algorithms and are shown to be very effective for problems such as image segmentation

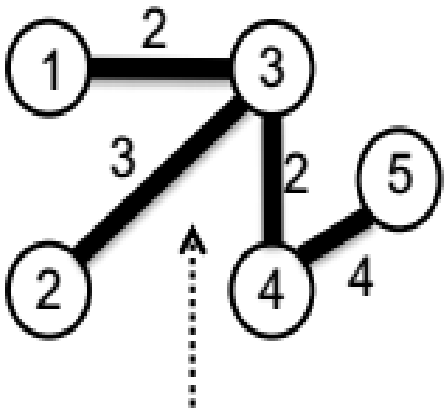
- Multi-level graph partitioning algorithms with biggest graph datasets
  - **Metis:** . A fast and high quality multilevel scheme for partitioning irregular graphs
  - **Graclus:** Weighted Graph Cuts without Eigenvectors A Multilevel Approach .

    It optimizes weighted cuts by optimizing an equivalent weighted kernel K-means loss function. The avoidance of expensive eigenvector computation gives it a big boost in speed while retaining or improving upon the quality of spectral approaches

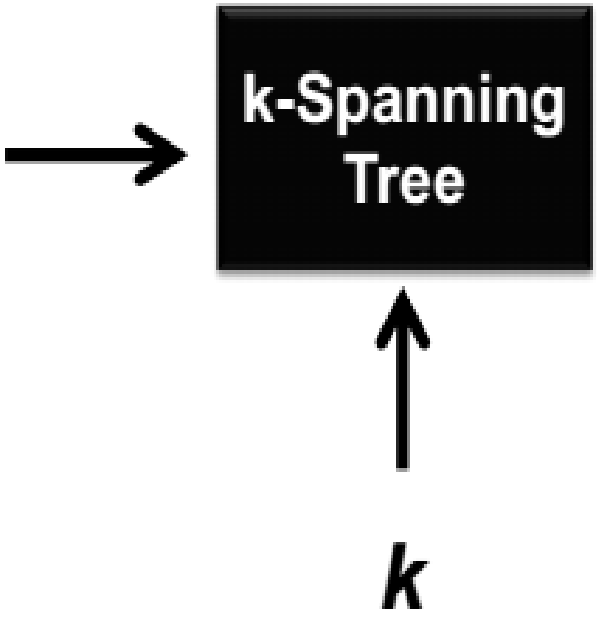
- Divisive/agglomerative approaches have been popular in network analysis, but they are expensive and do not scale well
- Minimum Spanning Tree based clustering

- Markov Clustering (**MCL**), a graph clustering algorithm based on stochastic flow simulation, has proved to be highly effective at clustering biological networks

Minimum Spanning Tree based Clustering



Minimum Spanning Tree



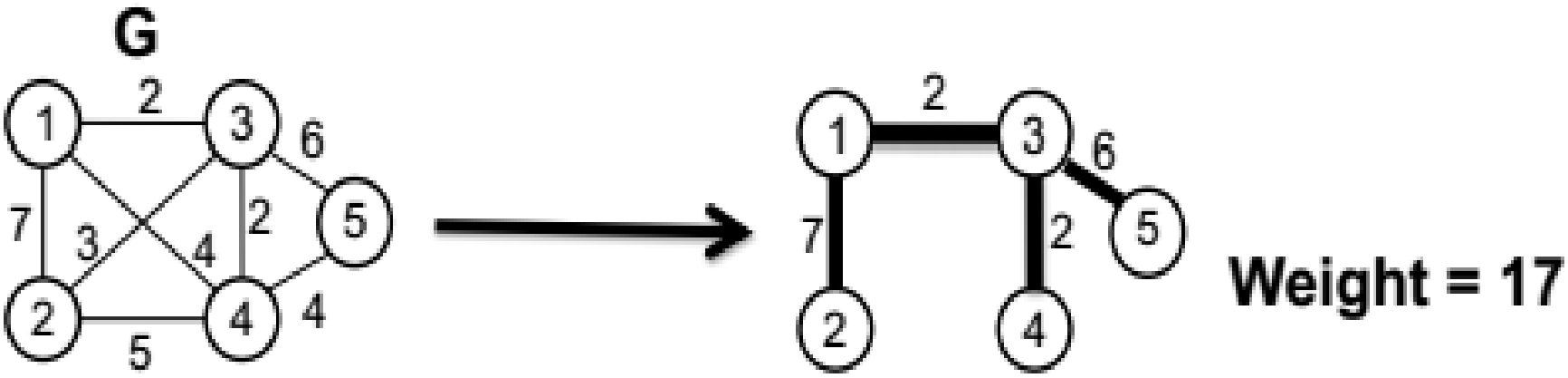
k groups of non-overlapping vertices

STEPS:

- **Obtains the Minimum Spanning Tree (MST) of input graph G**
- **Removes k-1 heaviest edges from the MST**
- **Results in k clusters**

What is a Spanning Tree?

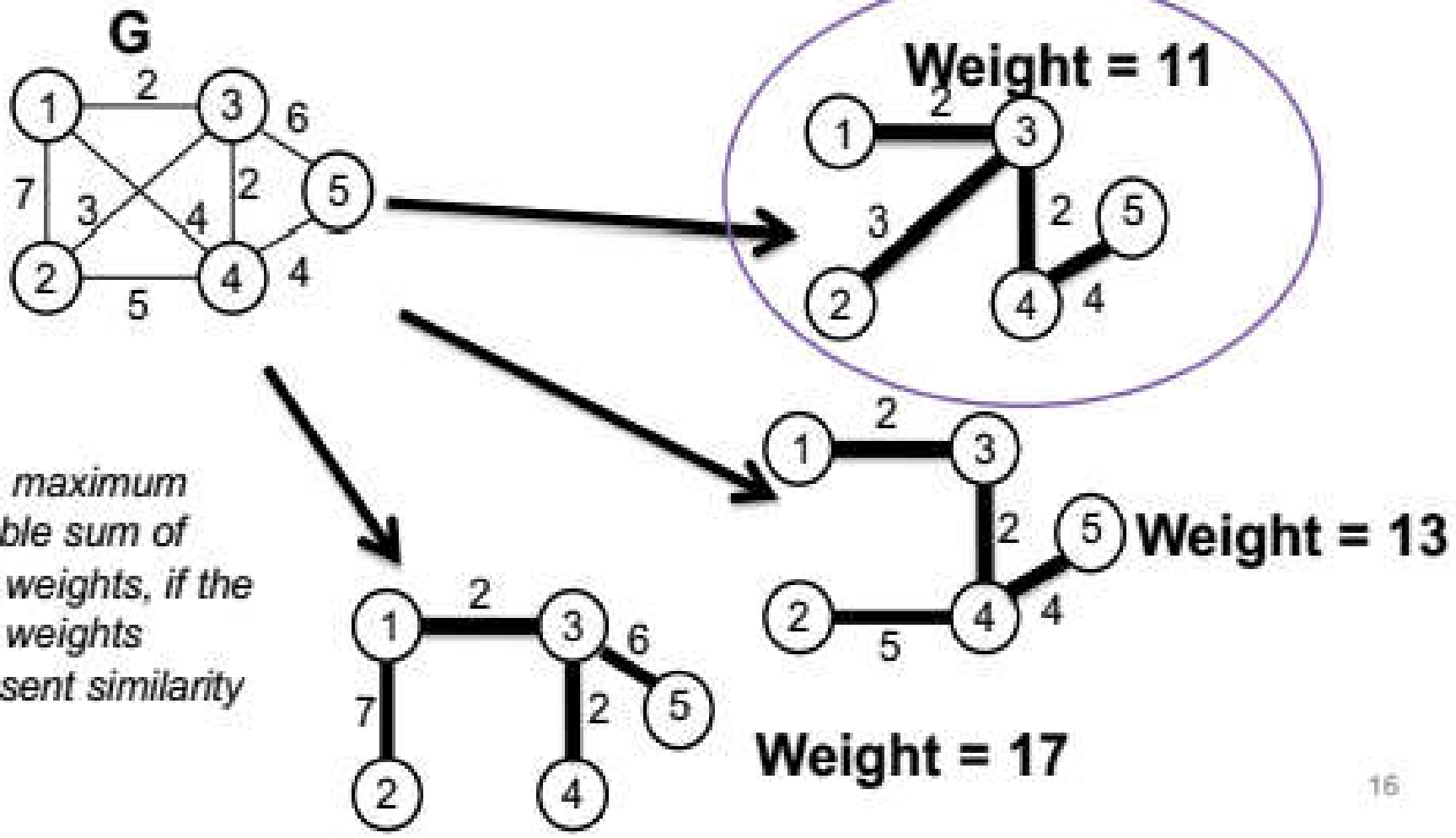
A connected subgraph with no cycles that includes all vertices in the graph



Note: *Weight can represent either distance or similarity between two vertices or similarity of the two vertices*

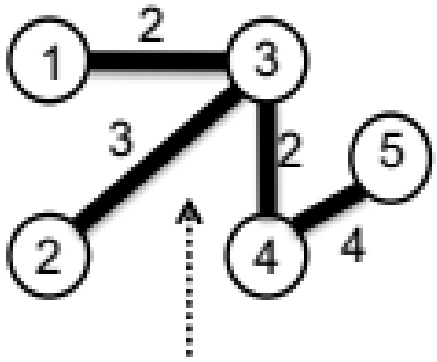
What is a Minimum Spanning Tree (MST)?

The spanning tree of a graph with the minimum possible sum of edge weights, if the edge weights represent distance



Note: *maximum possible sum of edge weights, if the edge weights represent similarity*

k-Spanning Tree

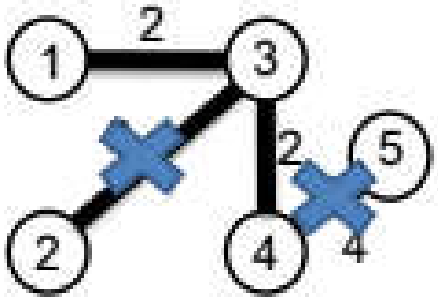


Minimum Spanning Tree

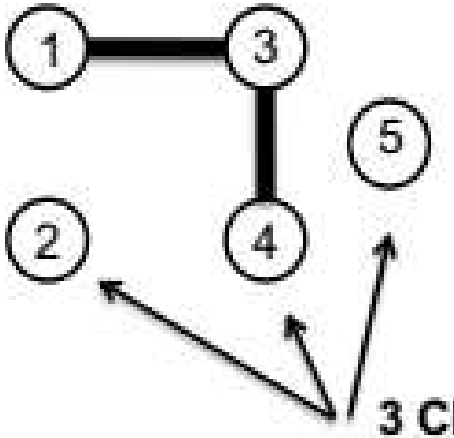
Remove k-1 edges with highest weight

Note: *k – is the number of clusters*

E.g., k=3



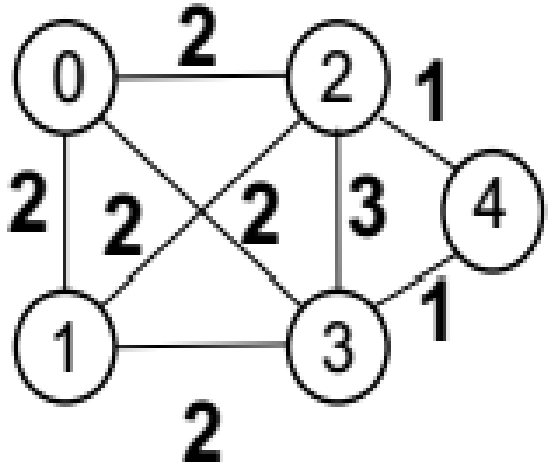
E.g., k=3



3 Clusters

Shared Nearest Neighbor Clustering

Shared Nearest Neighbor Graph (SNN)



Groups
of
non-overlapping
vertices

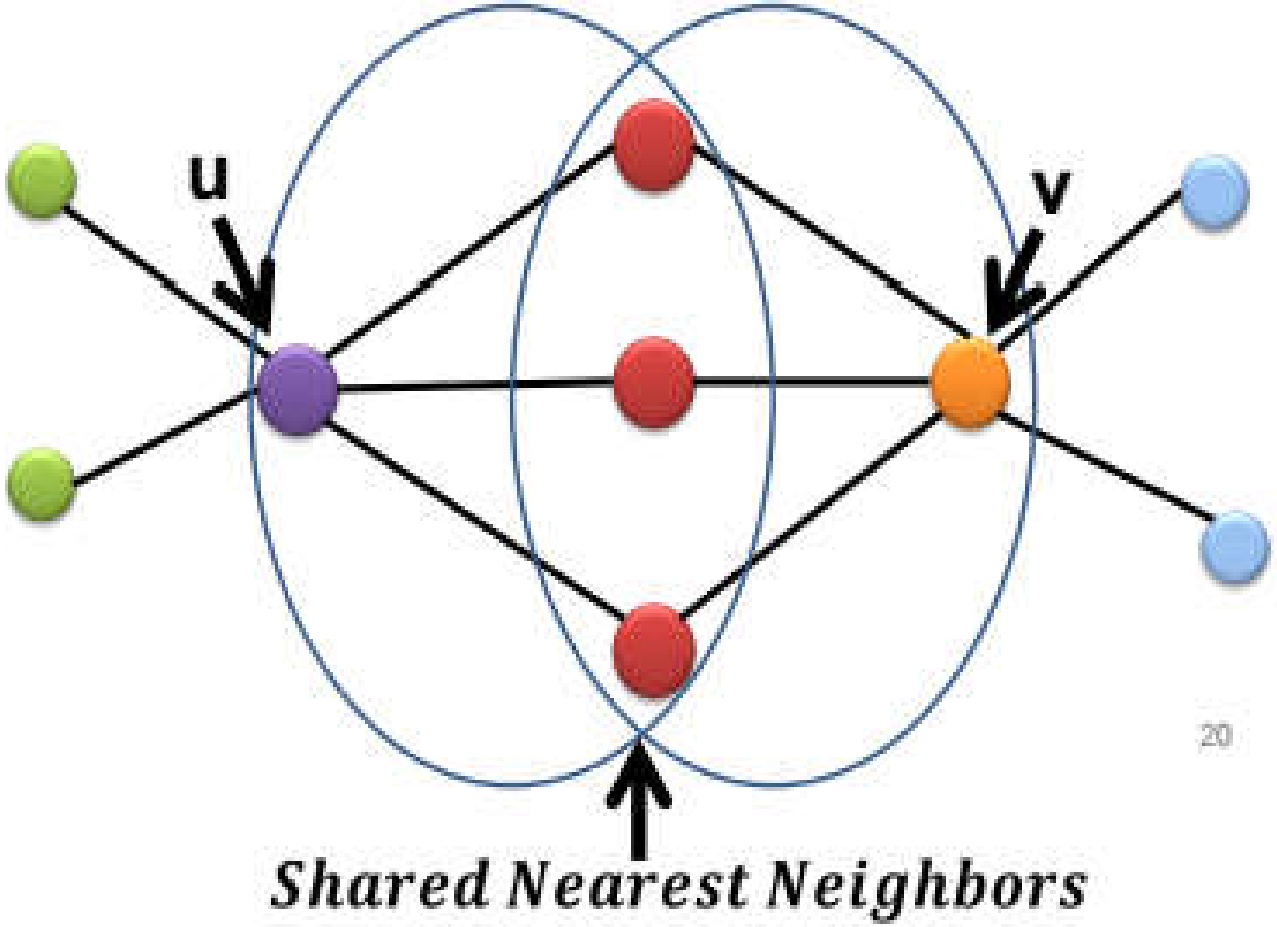
T

STEPS:

- **Obtains the Shared Nearest Neighbor Graph (SNN) of input graph G**
- **Removes edges from the SNN with weight less than τ**

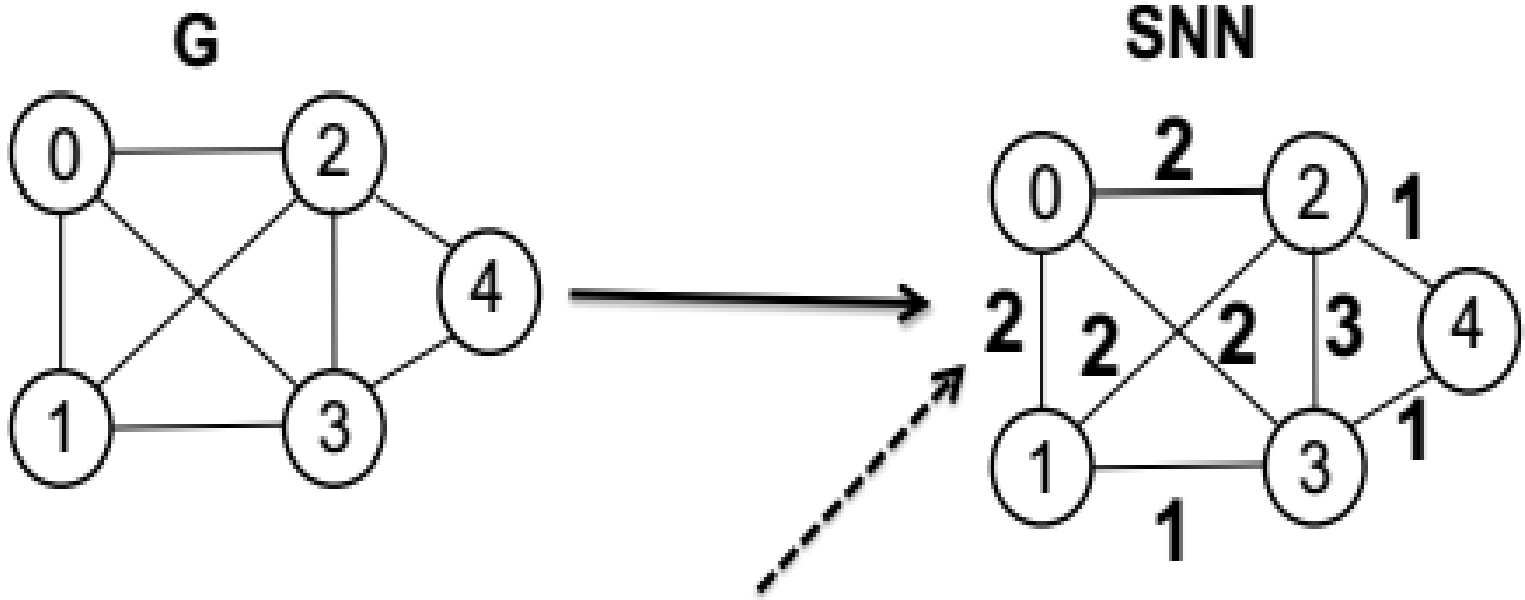
What is Shared Nearest Neighbor?

Shared Nearest Neighbor is a proximity measure and denotes the number of neighbor nodes common between any given pair of nodes



Shared Nearest Neighbor (SNN) Graph

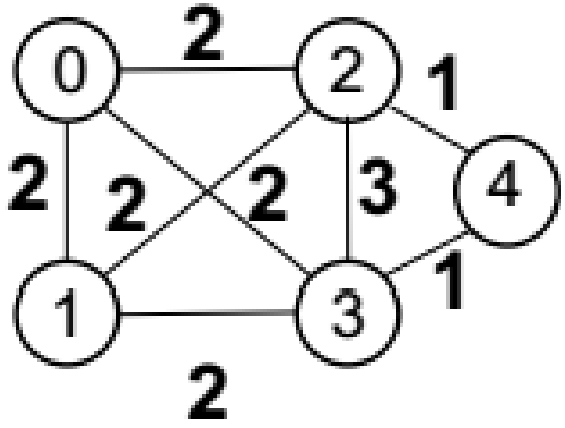
Given input graph G, weight each edge (u,v) with the number of shared nearest neighbors between u and v



Node 0 and Node 1 have 2 neighbors in common: Node 2 and Node 3

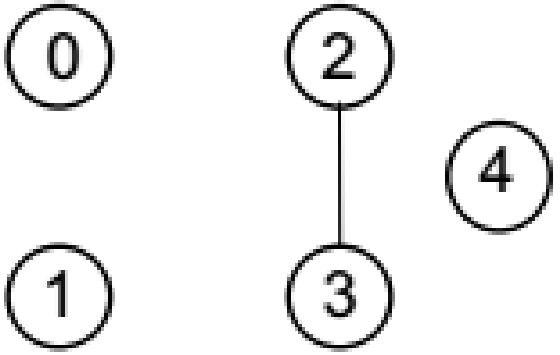
Shared Nearest Neighbor Clustering
Jarvis-Patrick Algorithm

SNN graph of input graph G



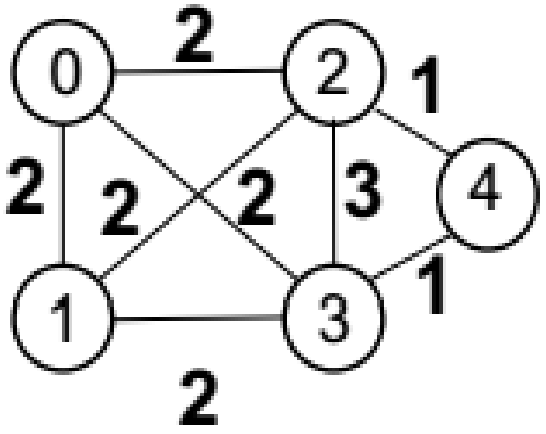
If u and v share more than τ neighbors
Place them in the same cluster

E.g., τ =3



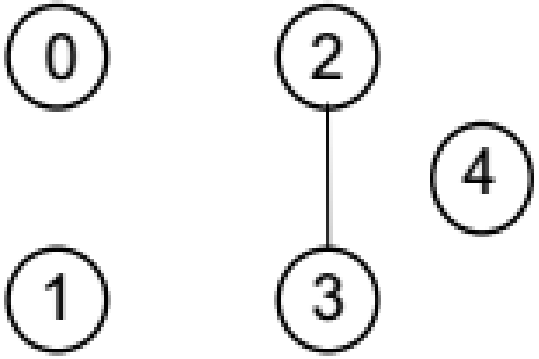
Shared Nearest Neighbor Clustering
Jarvis-Patrick Algorithm

SNN graph of input graph G



If u and v share more than τ neighbors
Place them in the same cluster

E.g., τ =3



Graph Clustering

- Intuition:
  - High connected nodes could be in one cluster
  - Low connected nodes could be in different clusters.
- Model:
  - A random walk may start at any node
  - Starting at node r, if a random walk will reach node t with high probability, then r and t should be clustered together.

Markov Clustering (MCL)

- Markov process
  - The probability that a random will take an edge at node u only depends on u and the given edge.
  - It does not depend on its previous route.
  - This assumption simplifies the computation.

MCL

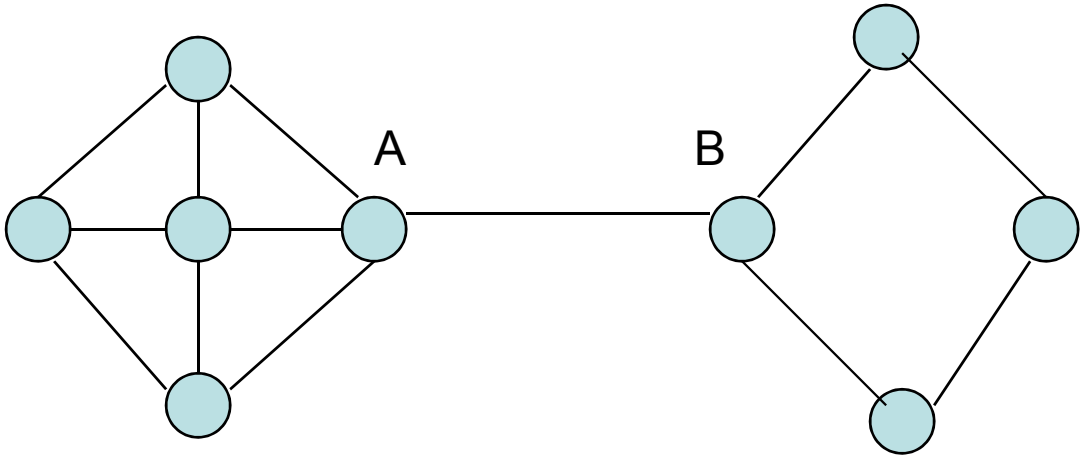
- Flow network is used to approximate the partition
- There is an initial amount of flow injected into each node.
- At each step, a percentage of flow will goes from a node to its neighbors via the outgoing edges.

MCL

- **Edge Weight**
  - Similarity between two nodes
  - Considered as the **bandwidth or connectivity**.
  - If an edge has higher weight than the other, then more flow will be flown over the edge.
  - The amount of flow is proportional to the edge weight.
  - If there is **no edge weight**, then we can assign the **same** weight to all edges.

Intuition of MCL

- Two natural clusters

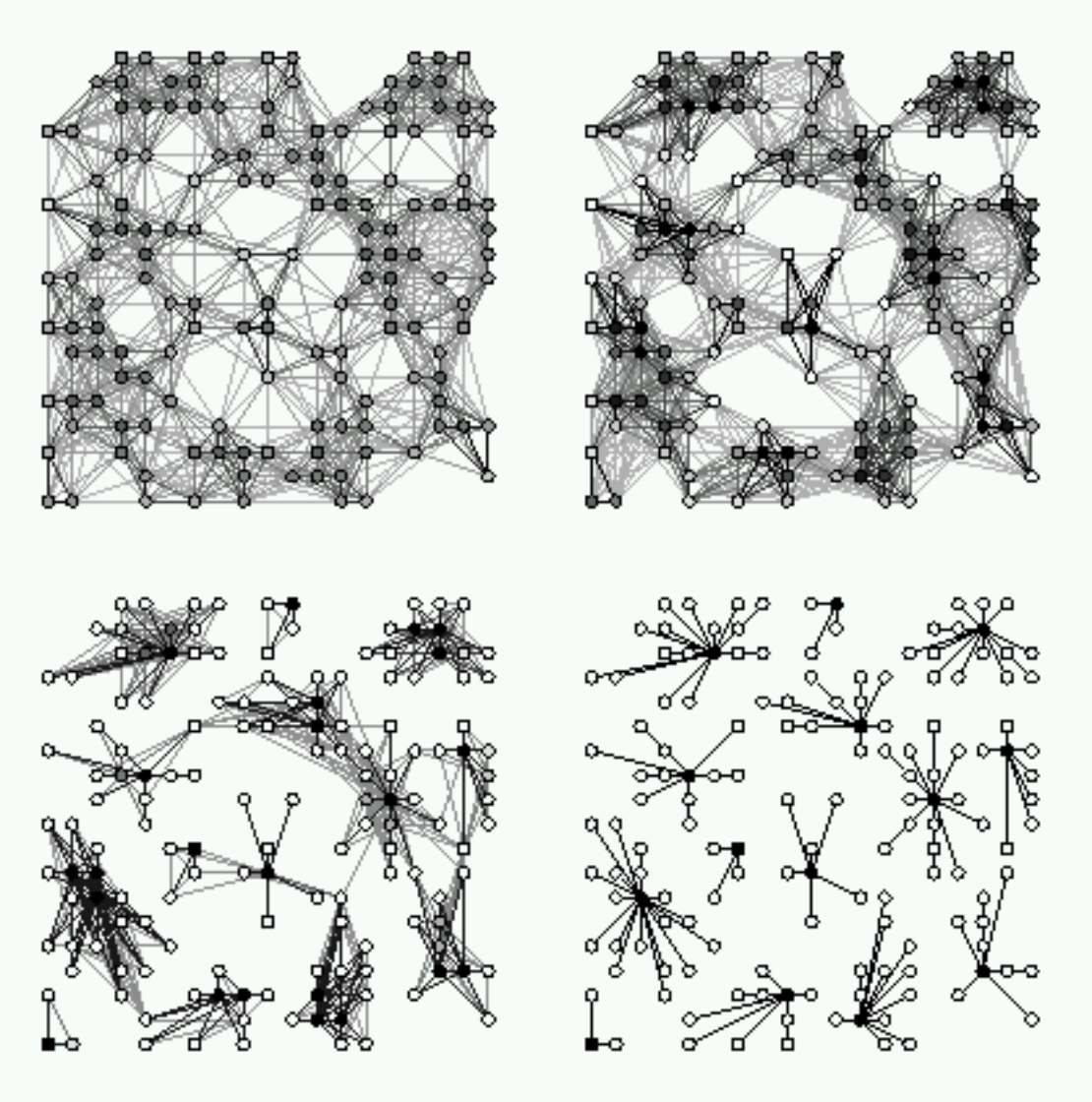


- When the flow reaches the border points, it is likely to return back, then cross the border.

MCL

- When the flow reaches A, it has four possible outcomes.
  - Three back into the cluster, one leak out.
  - ¾ of flow will return, only ¼ leaks.
- Flow will accumulate in the center of a cluster (island).
- The border nodes will starve.

Example



Expand: Input M , output M_{exp}.

$$M_{exp} = Expand(M) \stackrel{def}{=} M * M$$

The i^{th} column of M_{exp} can be interpreted as the final distribution of a random walk of length 2 starting from vertex v_i, with the transition probabilities of the random walk given by M. One can take higher powers of M instead of a square (corresponding to longer random walks), but this gets computationally prohibitive very quickly.

Inflate: Input M and inflation parameter r, output M_{inf}.

$$M_{inf}(i,j) \stackrel{def}{=} \frac{M(i,j)^r}{\sum_{k=1}^{n} M(k,j)^r}$$

M_{inf} corresponds to raising each entry in the matrix M to the power r and then normalizing the columns to sum to 1. By default $r = 2$. Because the entries in the matrix are all guaranteed to be less than or equal to 1, this operator has the effect of exaggerating the inhomogeneity in each column (as long as $r > 1$). In other words, flow is strengthened where it is already strong and weakened where it is weak.

Prune: In each column, we remove those entries which have very small values (where "small" is defined in relation to the rest of the entries in the column), and the retained entries are rescaled to have the column sum to 1. This step is primarily meant to reduce the number of non-zero entries in the matrix and hence save memory.We use the *threshold pruning* heuristic, where we compute a threshold based on the average and maximum values within a column, and prune entries below the threshold. [5]

MCL

R-MCL

MLR-MCL

MCL

- MCL offers several advantages in that it is an elegant approach based on the natural phenomenon of flow, or transition probability, in graphs

- It has been shown to be robust to topological noise effects, and while not completely non-parametric, varying a simple parameter can result in clusterings of different granularities

- However, MCL has drawn limited attention from the data mining community primarily, because
  - It does not scale very well even to moderate sized graphs
  - MCL tends to fragment communities, a less than desirable feature in many situations

S. V. Dongen. Graph Clustering by Flow Simulation. PhD thesis, University of Utrecht, 2000.

R-MCL & MLR-MCL

- *Can we retain the strengths of MCL while redressing its weaknesses?*

- **R-MCL**: first analyze the basic MCL algorithm carefully to understand the cause for these two limitations. Then identify a simple regularization step that can help alleviate the fragmentation problem

- **MLR-MCL**: develop a scalable multi-level variant of R-MCL. The basic idea is to coarsen the graph, run R-MCL on the coarsened graph, and then refine the graph in incremental steps

- The central intuition of MLR-MCL is that using flow values derived from simulation on coarser graphs can lead to good initializations of flow in the refined graphs. The multi-level approach also allows us to obtain large gains in speed

Definition

- A natural notion of graph clustering is the separation of sparsely connected dense subgraphs from each other

- A column-stochastic matrix of graph *G* is simply a matrix where each column sums to 1. Such matrix *M* can be interpreted as the matrix of the *transition probabilities* of a random walk or a Markov chain
  - M(j,i) represents the probability of a transition from vertex v_i to v_j.
  - Given adjacency matrix *A,* the most common way of deriving a column-stochastic transition matrix M is:

$$M(i,j) = \frac{A(i,j)}{\sum_{k=1}^{n} A(k,j)}$$

Fragment communities with MCL

- *Why does MCL output so many clusters?*

- MCL allows the columns of many pairs of neighboring nodes in the flow matrix M to diverge significantly

- This happens because the MCL algorithm uses the adjacency structure of the input graph only at the start, to initialize the flow matrix M

- After that, the algorithm works only with the current flow matrix M, and there is nothing in the algorithm that prevents the columns of neighboring nodes to differ widely without any penalty

Solution

- Idea: regularize or smooth the flow distributions out of a node w.r.t. its neighbors

- Let q_i be the flow distributions of the k neighbors of a given node i in the graph ⇔ column i from the current flow matrix M. Let w_i be the respective normalized edge weights, *i.e.* summary of w_i is 1

- We ask the following question: how do we update the flow distribution out of a given node (q*) so that it is the least divergent from the flow distributions of its neighbors? q* can formalized as:

$$\mathbf{q}^*(j) = \sum_{i=1}^{k} w_i \mathbf{q}_i(j)$$

Solution

Algorithm 2 Regularized MCL

$A := A + I$ // Add self loops and transform weights
$M := M_G := AD^{-1}$ //Initialize M as the canonical transition matrix

repeat
  $M := M_{exp} = Expand(M) \stackrel{def}{=} M * M$
  $M := M_{inf} := Inflation(M, r)$
  $M := Prune(M)$
until M converges

Interpret M as a clustering as described in Section 2.2

Hence, we replace the *Expand operator in the MCL process* with a new operator which updates the flow distribution of each node. We call this the *Regularize* operator, and it can be conveniently expressed in matrix notation as right multiplication with the canonical transition matrix M_G *of the graph G*

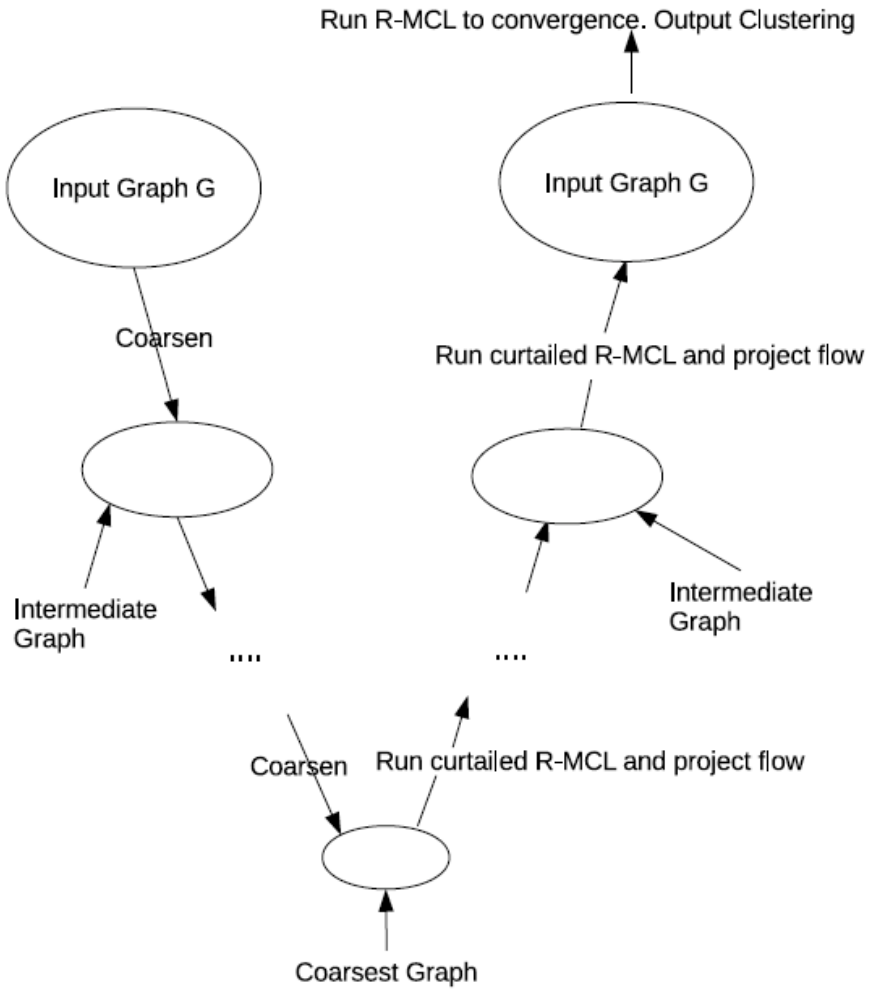
Results

Dataset	R-MCL				MCL			
	Clusters	N-Cut	Avg. N-Cut	Time	Clusters	N-Cut	Avg. N-Cut	Time
Hep-Ph	458	190.03	0.41	5.41	1464	827.31	0.56	85
Cora	880	367.62	0.41	3.58	2991	1888.6	0.63	82
Astro	343	124.40	**0.36**	8.84	1940	1301.5	0.67	515
Dblp	1750	575.19	0.32	1.57	1943	648.48	0.33	12.0
Epinions	4025	1863	0.46	32.3	15663	10041	0.64	4383
Hep-Th	735	266.4	**0.36**	1.05	1655	855	0.51	12

Multi-level Regularized MCL (MLR-MCL)

- The main intuition behind using a multi-level framework is that the flow values resulting from simulation on the coarser graphs can be effectively used to initialize the flows for simulation on the bigger graphs

- The algorithm also runs much faster because the initial few iterations, which are also the most time taking, are run on the smallest graphs first



Step 1. Coarsening

- The input graph G is coarsened successively into a series of smaller graphs G_1, G_2, . . . $G_{l,}$ where G_l is a graph of manageable size

- First, construct a matching on the graph, where a matching is defined as a set of edges no two of which are incident on the same vertex
- Second, the two vertices that are incident on each edge in the matching are collapsed to form a super-node, and the edges of the super-node are the union of the edges of its constituent nodes

- To keep track of the coarsening, Two arrays in each coarse graph, NodeMap1 and NodeMap2, were used
  - NodeMap1 maps a node in the coarse graph to its first constituent node
  - NodeMap2 maps a node to its second constituent node

Solution

- The original R-MCL and MLR-MCL use HEM (Heavy Edge Matching), which picks an unmatched neighbor connected to the heaviest edge for a given node, to coarse the graph [15]. In HEM, the node *v* to which a node *u* is merged is determined as follows:

$$v = \underset{v' \in N_{unmatched}(u)}{\arg\max} W(u, v'),$$

where $N_{unmatched}(u)$ is the set of unmatched neighbors of *u*, and $W(u,v')$ is the weight between *u* and v'. Note that HEM allows a node to be matched with at most one other node. MLR-MCL assigns all flows of a super node to one of its children for the flow projection. It is shown that a clustering result is invariant on the choice of the child to which all flows are assigned. For more details, refer to [15]. Note that MLR-MCL greatly reduces the overall computation of R-MCL since the flow update is done for the coarsened graph which is smaller than the original graph.

Solution

Limitation of HEM. HEM of MLR-MCL has two main limitations. First, the strategy of HEM that merges at most two single nodes can lead to undesirable coarsening where super nodes are not cohesive enough (see "Cohesive super node" section for details). Second, HEM is known to be unsuitable for real-world graphs [19] due to skewed degree distribution of the graphs which prevents the graph size from being effectively reduced (see "Quickly reduced graph" section for details). These shortages of HEM make MLR-MCL inefficient for real-world graphs. To overcome this, in the next section we propose PS-MCL that allows multiple nodes to be merged at a time.

Solution

- Next, we consider the method PS-MCL (Parallel Shotgun Coarsened MCL) which improves MLR-MCL in two perspectives:
  1. increasing efficiency of the graph coarsening and
- 2. parallelizing the operations of R-MCL.

Solution

Next, we consider the method PS-MCL (Parallel Shotgun Coarsened MCL) which improves MLR-MCL in two perspectives:

1. increasing efficiency of the graph coarsening and
2. parallelizing the operations of R-MCL.

Algorithm 2: Shotgun Coarsening

Input: Graph $G = (V, E)$,
a node weight map $Z : V \to \mathbb{N}$ for G,
an edge weight map $W : E \to \mathbb{N}$ for G.
Output: Coarsend Graph $G' = (V', E')$,
a node weight map $Z' : V \to \mathbb{N}$ for G',
an edge weight map $W' : E' \to \mathbb{N}$ for for G'.

$F \leftarrow \emptyset$.
foreach $u \in V$ **do**
  $N_1(u) \leftarrow \{v \in N(u) : W(u, v) = \max_{x \in N(u)} W(u, x)\}$.
  $N_2(u) \leftarrow \{v \in N_1(u) : Z(v) = \min_{x \in N_1(u)} Z(x)\}$.
  $F \leftarrow F \cup \{(u, v)\}$ for arbitrary picked $v \in N_2(u)$.
end
$V' \leftarrow connected\_components(V, F)$.
foreach $S \in V'$ **do**
  $Z'(S) \leftarrow \sum_{u \in S} Z(u)$.
  $E \leftarrow E \cup \{(S, S)\}$.
  $W'(S, S) \leftarrow \sum_{e \in E \cap (S \times S)} W(e)$.
end
$E' \leftarrow \emptyset$.
foreach $(S, T) \in V' \times V'$ *such that* $S \neq T$ **do**
  $H = \{e \in E : e \in S \times V \text{ or } e \in T \times S\}$.
  if $|H| > 0$ **then**
    $E' \leftarrow E' \cup \{(S, T)\}$.
    $W'(S, T) \leftarrow \sum_{e \in H} W(e)$.
  end
end

Solution

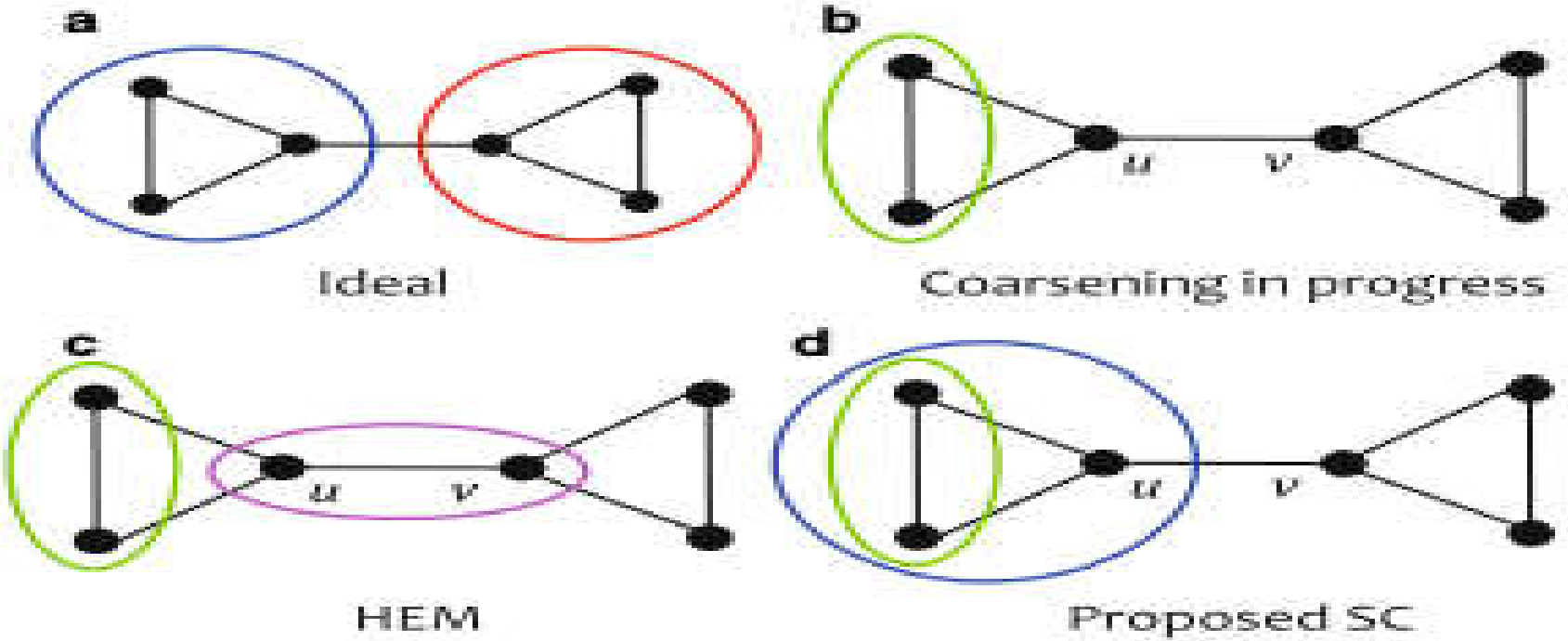


Fig. 1 Effectiveness of our proposed SC method compared with HEM. **a** Ideal coarsening for the graph. **b** Coarsening in progress. For the first merging, the leftmost two nodes are chosen. **c** For the second node to be merged, *u* is chosen. Since *v* is the only candidate for merging in HEM, *u* and *v* are merged to a super node. **d** In SC, the green super node is also a candidate for *u*. Since the weight of *u* to the green node is larger than that to *v*, *u* is merged to the green super node

Solution

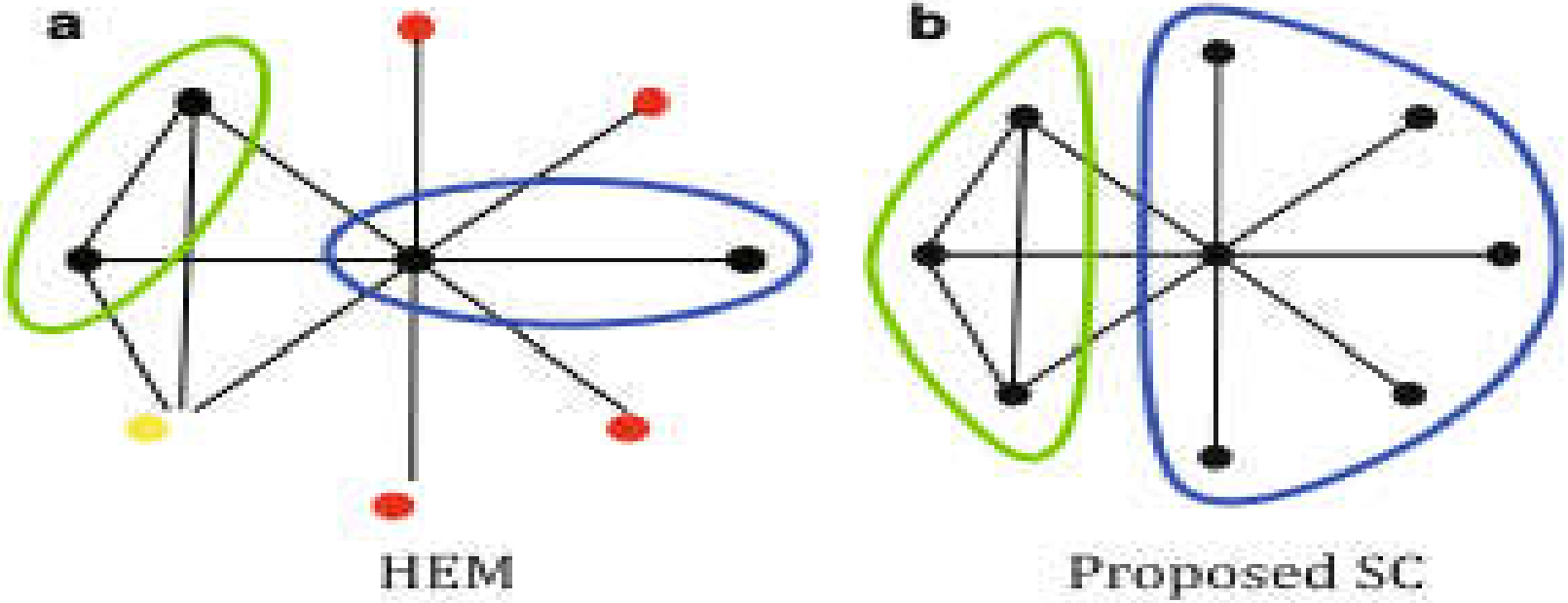


Fig. 2 Efficiency of our proposed coarsening method SC compared with HEM. **a** Result of one coarsening step by HEM. The red and yellow nodes are eventually merged to the blue and green nodes, respectively, but it takes 5 more coarsening steps. **b** Result of one coarsening step by SC. The result is the same as (**a**) after 5 more coarsening

Step 2. Curtailed R-MCL along with refinement

- Beginning with the coarsest graph, R-MCL is run for a few iterations (typically 4 to 5). This is referred as abbreviated version of R-MCL as Curtailed R-MCL

- The flow values at the end of this step run are then projected on to the refined graph of the current graph as per Algorithm 4, and R-MCL is run again for a few iterations on the refined graph, and this is repeated until we reach the original graph

Algorithm 4 ProjectFlow

Inputs: Coarse graph $\mathcal{G}_c$, Flow on the coarse graph M_c.
Output: Projected flow matrix on the refined graph M_r

NodeMap1 := $\mathcal{G}_c$.NodeMap1
NodeMap2 := $\mathcal{G}_c$.NodeMap2

for each non-zero entry (i,j) in M_c **do**
    M_r(NodeMap1(i),NodeMap1(j)) :=M_c(i,j)
    M_r(NodeMap1(i),NodeMap2(j)) :=M_c(i,j)
    M_r(NodeMap2(i),NodeMap1(j)) :=0
    M_r(NodeMap2(i),NodeMap2(j)) :=0
end for
return M_r

Problem of flow projection

- The remaining part of MLR-MCL is the algorithm for projecting flow from a coarse graph to a refined graph
- Projection of flow is concerned with using the flow values of a coarse graph to provide a good initialization of the flow values in the refined graph
- Since there are two nodes in the refined graph corresponding to each node in the coarse graph, a flow value between two nodes in the coarse graph must be used to derive the flow values between four pairs of nodes in the refined graph
- Let n_c be the size of the coarse graph, then n_c^2 entries of the flow matrix of the coarse graph may derive the $4 * n_c^2$ entries of the refined graph

Solution

- Naive strategy: assign the flow between two nodes in the refined graph as the flow between their respective parents in the coarse graph
- But this doubles the number of nodes that any node in the refined graph flows out to. This results in excessive smoothing of the out-flows of each node
- New strategy: choose only one child node for each parent node and project all the flow into the parent node to the chosen child node
- However, this raises the question: which child node do we pick in order to assign all the flow into? It turns out that it doesn't matter which child node to pick, as long as for each parent, the choice is consistent

Proof

PROPOSITION 4. *Let M_c be a flow matrix on one of the coarse graphs in MLR-MCL. Let M_r^1 be the result of flow projection from M_c by randomly choosing for each parent node which of its child nodes will be assigned all the in-flows of the parent node. Let M_r^2 be the result of another flow projection along the same lines, with a (possiby different) random choice at each parent node. Then M_r^1 and M_r^2 are row permutable.*

PROOF. Consider a fixed node i in the coarse graph. Assume that in M_r^1, the choice at i is to project the flow to the first child node NodeMap1(i),whereas in M_r^2 the flow is projected to NodeMap2(i) instead. Then the row corresponding to NodeMap1(i) in M_r^1 is the same as the row corresponding to NodeMap2(i) in M_r^2, since they are both projections of the in-flows of the same parent node i. Also, both the row corresponding to NodeMap2(i) in M_r^1 and the row corresponding to NodeMap1(i) in M_r^2 consist of zeroes. Hence, the two rows corresponding to the two child nodes of i in M_r^1 have been permuted in M_r^2. We can extend this argument for every node at which different choices are made in the two flow projections, and therefore conclude that M_r^1 can be converted to M_r^2 using a series of row permutations, one for each different choice. □

Step 3. R-MCL on original graph

- With flow values initialized from the previous phase, R-MCL is run on the final graph until convergence

- The flow matrix at the end is converted into a clustering in the usual way, with all the nodes that flow into the same “attractor” node being assigned into one cluster

Clustering evaluation criteria

- The *normalized cut* of a cluster C in the graph G is defined as

$$Ncut(\mathcal{C}) = \frac{\sum_{v_i \in \mathcal{C}, v_j \notin \mathcal{C}} A(i,j)}{\sum_{v_i \in \mathcal{C}} degree(v_i)}$$

- The normalized cut of a clustering $\{C_1, C_2, \ldots, C_k\}$ is the sum of the normalized cuts of the individual clusters

$$Ncut(\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}) = \sum_{i=1}^{k} Ncut(\mathcal{C}_i)$$

- The average normalized cut of a clustering is the average of the normalized cuts of each of the constituent clusters, and lies between 0 and 1

Data

Name	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	Avg. degree
Cora	17604	74180	8.42
Dblp	16196	45031	5.56
Astro-Ph	17903	196972	22.00
Hep-Ph	11204	117619	21.00
Hep-Th	8638	24806	5.78
Epinions	75877	405739	10.69
Yeast-PPI	4741	15148	6.39

Table 1: Details of real datasets

Name	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	No. of clusters
synth_1M	1000, 000	58, 256, 766	10000
synth_500K	500, 000	22, 682, 795	5000
synth_100K	100, 000	3, 447, 123	1000
synth_50K	50, 000	2, 272, 534	500
synth_10K	10, 000	319, 383	100

Table 2: Details of synthetic datasets

Results

Dataset	MLR-MCL				R-MCL				MCL			
	Clusters	N-Cut	Avg. N-Cut	Time	Clusters	N-Cut	Avg. N-Cut	Time	Clusters	N-Cut	Avg. N-Cut	Time
Hep-Ph	264	76.77	**0.29**	**0.91**	458	190.03	0.41	5.41	1464	827.31	0.56	85
Cora	670	238.19	**0.35**	**1.26**	880	367.62	0.41	3.58	2991	1888.6	0.63	82
Astro	411	153.43	0.37	**2.62**	343	124.40	**0.36**	8.84	1940	1301.5	0.67	515
Dblp	723	152.24	**0.21**	**0.51**	1750	575.19	0.32	1.57	1943	648.48	0.33	12.0
Epinions	1632	735.87	**0.45**	**26.86**	4025	1863	0.46	32.3	15663	10041	0.64	4383
Hep-Th	795	293	0.37	**0.37**	735	266.4	**0.36**	1.05	1655	855	0.51	12

Table 3: Comparison of MLR-MCL, R-MCL and MCL on 6 real datasets. The same inflation parameter value of $r = 2$ was used for all 3 methods. The times are in seconds. Best results are in bold.

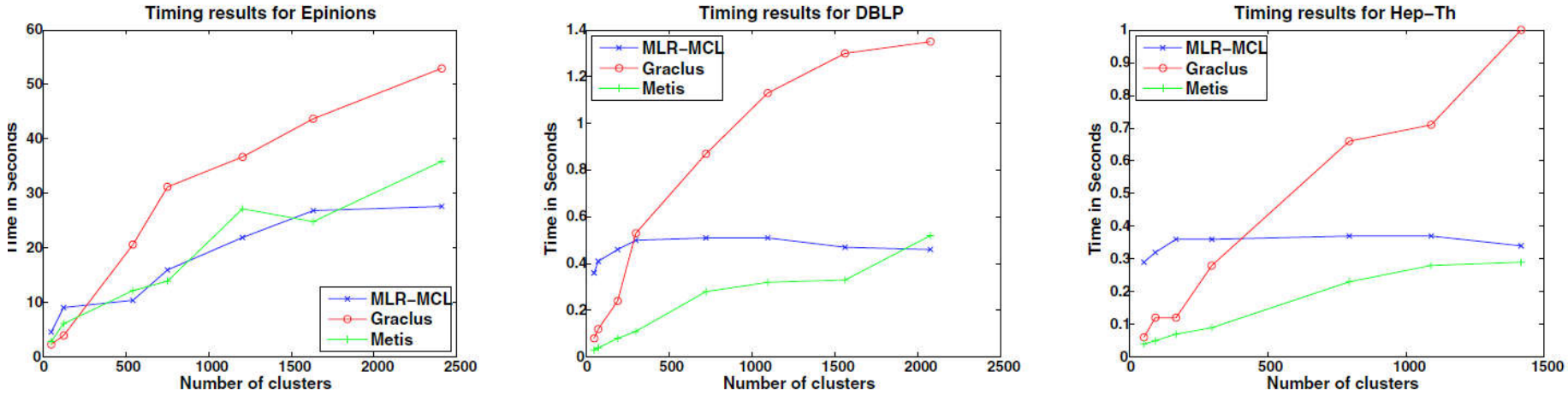


Figure 5: Comparison of timing for varying number of clusters across 3 real world datasets. (a) Epinions (b) Dblp (c) Hep-Th

Results

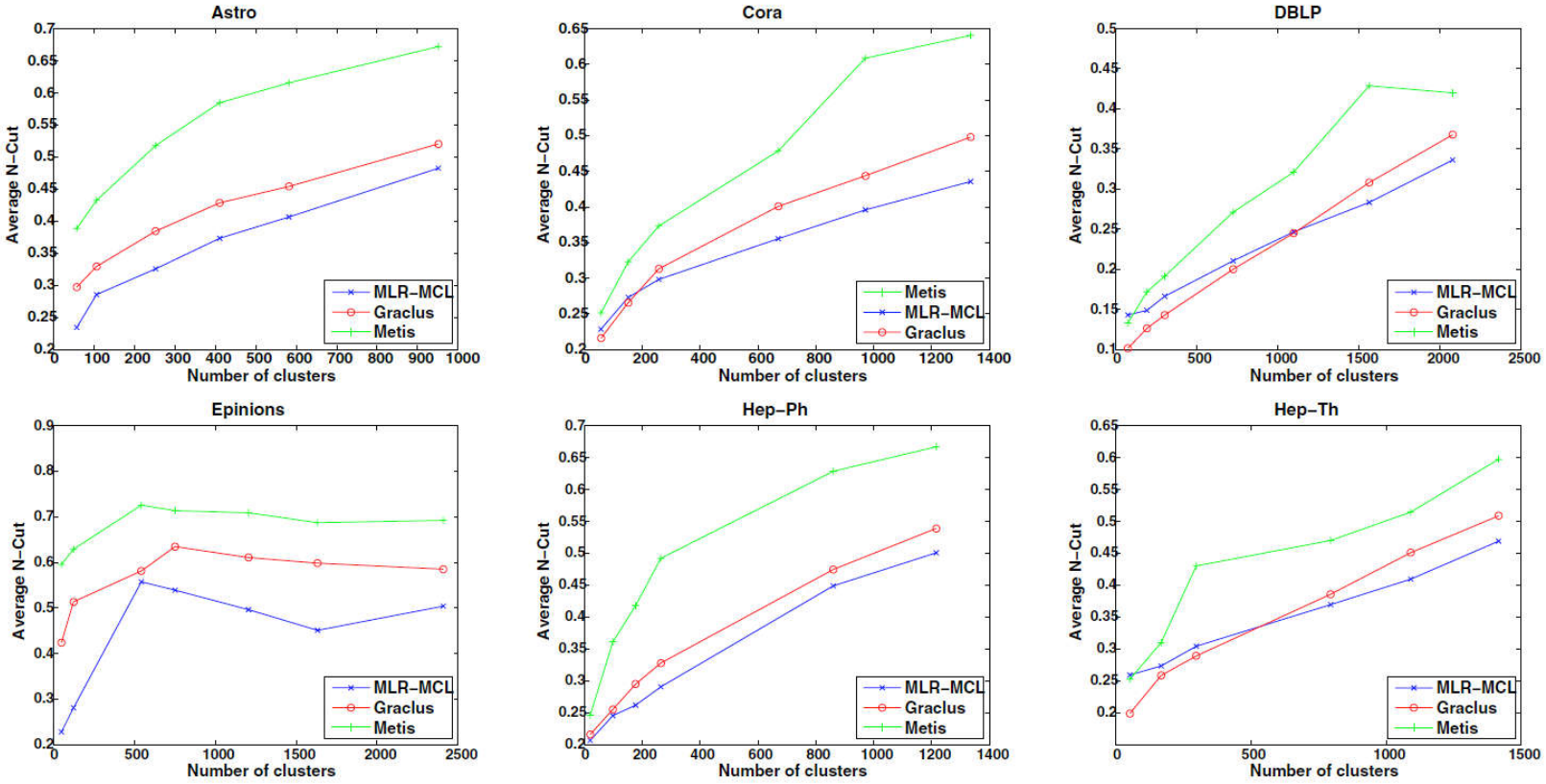
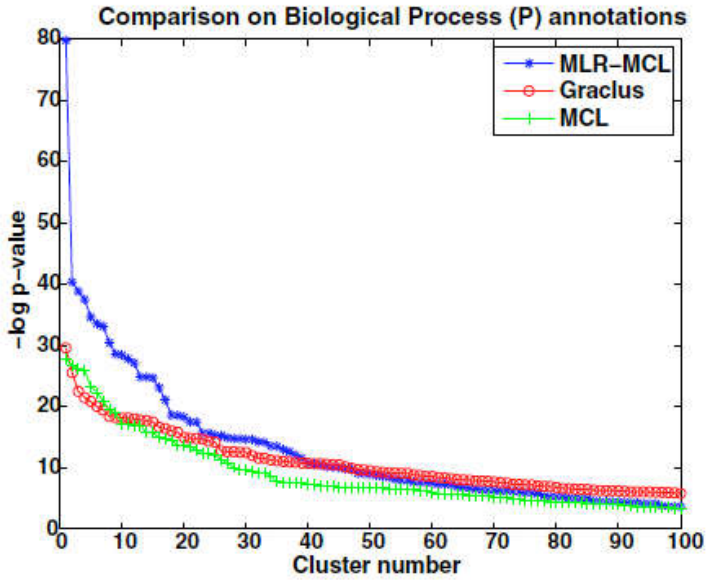


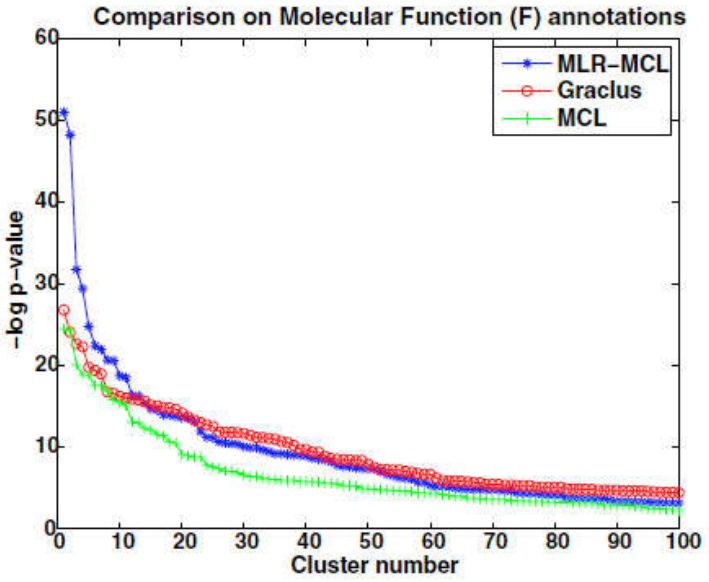
Figure 4: Comparison of Average Normalized Cut scores for varying number of clusters across 6 real world datasets. (a) Astro-Ph (b) Cora (c) Dblp (d) Epinions (e) Hep-Ph (f) Hep-Th. Lower is better.

Clustering PPI networks: A Case Study

- PPI network of *S. cervisiae*, which contains 4,741 proteins with 15,148 known interactions
- Perform a domain-specific evaluation using Gene Ontology using the Hyper-geometric test to construct domain specific qualitative metric
- Compare the performance of MCL, MLR-MCL and Graclus on this domain specific qualitative metric

Clustering PPI networks: A Case Study





Results

- **MLR-MCL vs. R-MCL and MCL:** MLR-MCL typically outperforms both R-MCL and MCL in terms of quality and delivers performance that is roughly 2-3orders of magnitude faster than MCL.

- **MLR-MCL vs. Graclus and Metis:** MLR-MCL delivers significant (10-15%) improvements in N-Cut values over Graclus in 4 of 6 datasets.

- **MLR-MCL vs. Graclus and MCL on PPI networks:** The top 8 clusters of proteins found by MLR-MCL are rated better than the best cluster returned by either Graclus and MCL.

Summary

- Algorithms based on simulating stochastic flows are a simple and natural solution for the problem of clustering graphs, but their widespread use has been hampered by their lack of scalability and fragmentation of output
- A multi-level algorithm for graph clustering was developed using flows that delivers significant improvements in both quality and speed
- The graph is first successively coarsened to a manageable size, and a small number of iterations of flow simulation is performed on the coarse graph. The graph is then successively refined, with flows from the previous graph used as initializations for brief flow simulations on each of the intermediate graphs. When we reach the final refined graph, the algorithm is run to convergence and the high-flow regions are clustered together, with regions without any flow forming the natural boundaries of the clusters
- Extensive experimental results on several real and synthetic datasets demonstrate the effectiveness of this approach when compared to state-of-the-art algorithms

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

A single-layer network is a graph $G = (V, E)$, where V is the set of nodes and E is the set of links that connect the objects of V. A multilayer network consists of a network at multiple levels, that is, with multiple types of edges. Thus, the notion of layer, along with nodes and edges, must be considered. Each layer represents a combination of different features of the network, called aspects or facets. More formally, a multilayer network $\mathcal{M}$ is defined as a quadruple:

$$M = (V_M, E_M, V, \mathbf{L})$$

where V is the set of nodes, $\mathbf{L} = \{\mathbf{L_a}\}_{\mathbf{a=1}}^{\mathbf{l}}$ is a sequence of sets of elementary layers L_a, $V_M \subseteq V \times L_1 \times \ldots \times L_l$ contains only the set of combinations of nodes and elementary layers effectively present in a layer, $E_M \subseteq V_M \times V_M$ is a set of couples of possible combinations (Kivelä et al. 2014). Nodes could be connected to any other both inside the same layer and across layers.

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

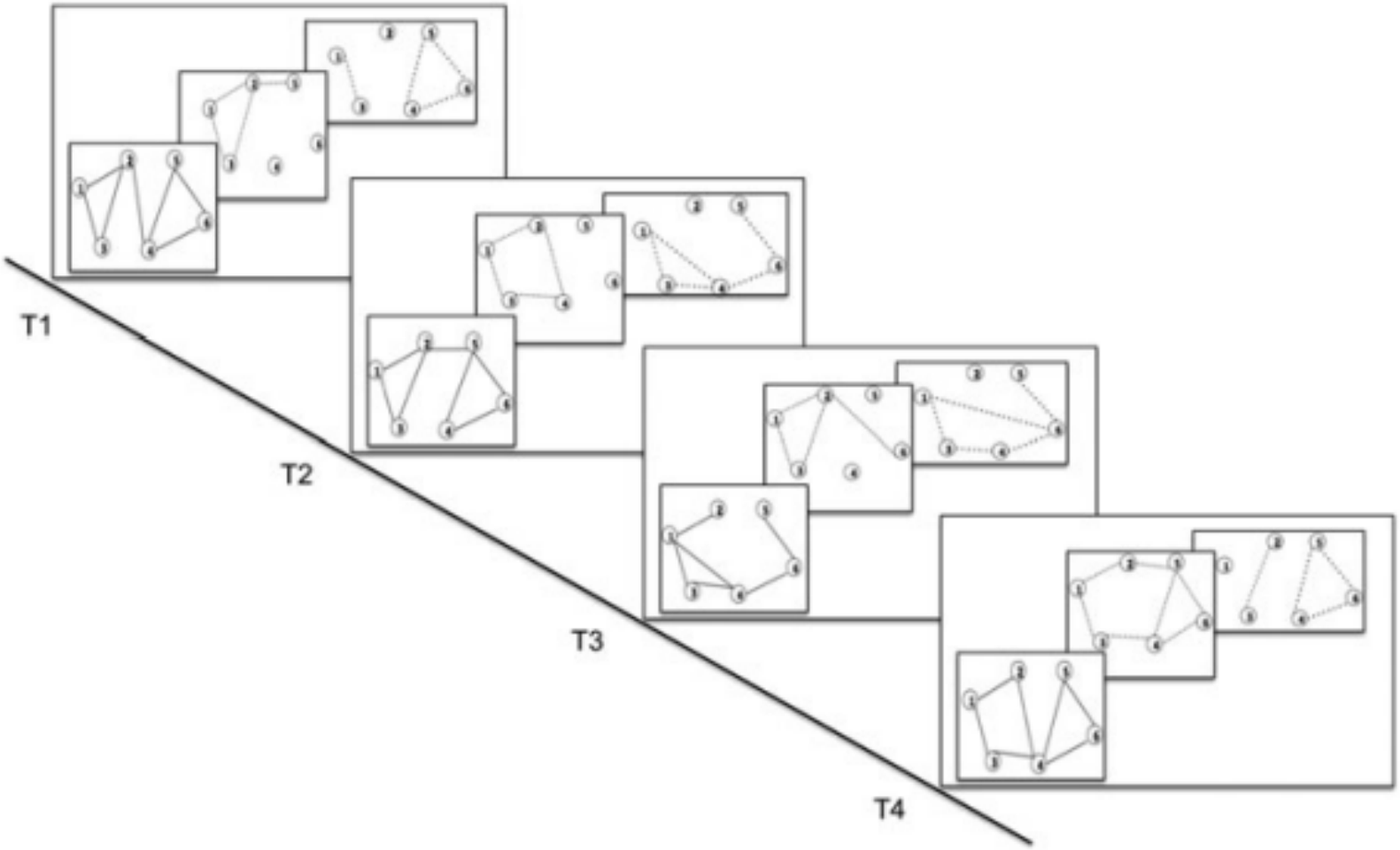


FIGURE 1. Example of dynamic multilayer network with four timestamps and three types of relationships.

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

An example of temporal multilayer network can be seen in Figure 1. The first aspect $L_1 = \{T_1, T_2, T_3, T_4\}$ is the temporal information about the network and consists of four timestamps. The second aspect $L_2 = \{D_1, D_2, D_3\}$ represents three different types of connections between nodes. Thus, there are 12 different layers $\{(T_1, D_1), (T_1, D_2), \ldots, (T_4, D_3)\}$. By grouping them with respect to the same timestamp, there are four multiplex networks $\mathcal{T}^t = \{\mathcal{N}_1^t, \mathcal{N}_2^t, \mathcal{N}_3^t\}, t = 1, \ldots, 4$. The temporal network $\mathcal{DM} = \{\mathcal{T}^1, \ldots, \mathcal{T}^4\}$ thus consists of four multiplex networks. At each timestamp, different kinds of links between nodes can be observed. For instance, at time $T1$, nodes 4 and 6 are connected with respect to the first and third relationships, that is, in $\mathcal{N}_1^1$ and $\mathcal{N}_3^1$, but not with respect to the second type of connection. In $\mathcal{N}_2^1$, they are isolated nodes. Along the time axis, it is possible to observe how layers evolve. For example, at time $T4$, nodes 4 and 6 are connected in all the $\mathcal{N}_1^4$, $\mathcal{N}_2^4$, and $\mathcal{N}_3^4$ networks.

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

Let $\mathcal{CS}_1^t, \ldots, \mathcal{CS}_d^t$ be the community structures obtained for each elementary layer of a multiplex network $\mathcal{T}^t = \{\mathcal{N}_1^t, \mathcal{N}_2^t, \ldots, \mathcal{N}_d^t\}$, at timestamp t. We say that $\mathcal{CS} = \{C_1, \ldots, C_k\}$ is a shared community structure of $\mathcal{T}^t$ if the two functions are maximized:

$$f_q(\mathcal{CS}, \mathcal{N}_i^t), i = 1, \ldots, d \tag{1}$$

$$f_s(\mathcal{CS}, \mathcal{CS}_i^t), i = 1, \ldots, d \tag{2}$$

This framework is then utilized between couples of consecutive timestamps $t - 1$ and t by resorting to the dynamic evolutionary approach where the temporal cost $\mathcal{TC}$ is guaranteed by considering the similarity between the community structure $\mathcal{CS}^{t-1}$ obtained for the previous timestamp and that found for the first elementary layer $\mathcal{CS}_1^t$ of the current timestamp.

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

DMultiMOGA optimizes the two competitive objectives $\mathcal{FQ}$ and $\mathcal{SQ}$ inside a fixed timestamp, and $\mathcal{FQ}$ and $\mathcal{TC}$ when a new timestamp starts. The first two objectives guarantee dimensional smoothing among the layers of a multidimensional network at a current timestamp, while the substitution of $\mathcal{SQ}$ with $\mathcal{TC}$, when a new timestamp begins, ensures temporal smoothness between consecutive timestamps. Thus, the *DMultiMOGA* method, at each timestamp t, tries to maximize the quality of the clustering obtained for the multidimensional network $\mathcal{T}^t = \{\mathcal{N}_1^t, \mathcal{N}_2^t, \ldots, \mathcal{N}_d^t\}$ at time t and to minimize the differences with respect to that obtained at time $t-1$.

EVOLUTIONARY CLUSTERING FOR MINING AND TRACKING DYNAMIC MULTILAYER NETWORKS

Input: Given a dynamic multilayer network $\mathcal{DM} = \{\mathcal{T}^1, \ldots, \mathcal{T}^T\}$, and the number T of timestamps.

Output: A clustering for each multilayer network $\mathcal{T}^i = \{\mathcal{N}_1^i, \ldots, \mathcal{N}_d^i\}$ of $\mathcal{DM}$ and the evolving community tracking.

Method: Perform the following steps:

1 **Let** $\mathcal{G}^1 = \{G_1^1, \ldots, G_d^1\}$ be the set of graphs modeling $\mathcal{T}^1$
2 $Track = \emptyset$
3 $\mathcal{CS}^0 = \emptyset$
4 **Perform** *MultiMOGA*($\mathcal{G}^1$,1, $\mathcal{CS}^0$)
5 **for** $t = 2$ to T
6 **Let** $\mathcal{G}^t = \{G_1^t, \ldots, G_d^t\}$ be the set of graphs modeling $\mathcal{T}^t$, and $\mathcal{CS}^{t-1}$ the clustering obtained at the previous timestamp
7 **Perform** *MultiMOGA*($\mathcal{G}^t$, t, $\mathcal{CS}^{t-1}$)
8 **Let** L^{t-1} and L^t be the cluster labels of nodes at timestamp $t-1$ and t, respectively
9 **Perform** $Track^t = RelabelHungarian(L^t, L^{t-1})$
10 $Track = Track \cup Track^t$
11 **end for** t % t-th timestamp

FIGURE 3. The pseudo-code of the *DMultiMOGA* algorithm.

MultiMOGA($\mathcal{G}^t$, t, $\mathcal{CS}^{t-1}$) **Method:**

Input: A multidimensional network $\mathcal{T}^t = \{\mathcal{N}_1^t, \mathcal{N}_2^t, \ldots, \mathcal{N}_d^t\}$ at timestamp t, modeled as a sequence of graphs $\mathcal{G}^t = \{G_1^t, \ldots, G_d^t\}$, and the clustering $\mathcal{CS}^{t-1}$ obtained at the $t-1$ timestamp

Output: A node cluster labeling that partitions $\mathcal{T}^t$ in the optimal shared community structure

```
1   if (t==1)
2     Perform ClustGA on the G_1^t graph
    else
3     Perform MultiDim(G_1^t, CS^{t-1}) on the G_1^t graph
4   for each node v_i of G^t not appearing in G_1^t
5     Perform LabelAssignment
6   for i = 2 to d
7     Create a population of random individuals whose
      length equals the number n_i =| V_i | of nodes of G_i^t;
8     Perform a multiobjective GA with objectives
        8.1 FQ = f_q(CS_i^t, G_i^t)
        8.2 SQ = f_s(CS_i^t, CS_{i-1}^t)
9     choose the solution CS_i^t = {C_{i_1}^t, ... C_{i_{k_i}}^t} of the
      Pareto front having the maximum f_q value;
10    for each node v_j of G^t not appearing in G_i^t
11      Perform LabelAssignment
12  end for
```

FIGURE 4. The pseudo-code of the *MultiMOGA* algorithm for a single timestamp.

***ClustGA* Method:**

Input: The graph G_1^1 modeling the first dimension of a multiplex network

Output: The clustering of G_1^1 having the best fitness value,
the node cluster labeling $L^1 = \{l_{v_1}, \ldots, l_{v_N}\}$

1 **create** an initial population of random individuals
whose length equals the number N of nodes
2 **while** not maxGen
3 **decode** each individual $I = \{g_1, \ldots, g_N\}$ to obtain
the partitioning $C = \{C_1, \ldots, C_k\}$
of the graph G in k connected components.
4 **evaluate** the fitness of the translated individuals
5 **create** a new population by applying the variation operators
6 **end while**

FIGURE 5. The pseudo-code of the *ClustGA* algorithm for the first dimension of multiplex network at timestamp $t = 1$.

***MultiDim* Method:**

Input: the first graph $\mathcal{G}_1^t$ at timestamp t, and the clustering $\mathcal{CS}^{t-1}$ at timestamp $t-1$

Output: A clustering of $\mathcal{G}_1^t$

1 **Perform** a multiobjective GA on G_1^t with objectives
  1.1 $\mathcal{FQ} = f_q(\mathcal{CS}_1^t, G_1^t)$
  1.2 $\mathcal{SC} = f_s(\mathcal{CS}_1^t, \mathcal{CS}^{t-1})$
2 **choose** the solution $\mathcal{CS}_1^t$ of the Pareto front
having the maximum modularity value;

FIGURE 6. The pseudo-code of the $MultiDim$ algorithm for the first dimension of multiplex network at a timestamp $t > 1$.

***LabelAssignment* Method:**

Input: the set of graphs $\mathcal{G} = \{G_1, \ldots, G_d\}$ modeling a multiplex network
the current node cluster labeling $L = \{l_{v_1}, \ldots, l_{v_N}\}$ of nodes of $\mathcal{G}$

Output: Updated L

2 **for each** node v_j with no cluster label
3 **let** $v_{n_1}, \ldots, v_{n_u}$ be the neighbors of v_j in $\mathcal{G}$
  and $l_{v_{n_1}}, \ldots, l_{v_{n_u}}$ be their cluster labels
4 assign to v_j $argmax_l$ $\{l_{v_{n_1}}, \ldots, l_{v_{n_u}})\}$

FIGURE 7. The pseudo-code of the $LabelAssignment$ algorithm.

RelabelHungarian **Method:**

Input: the node cluster labeling L^t at timestamp t and the node cluster labeling L^{t-1} at timestamp $t-1$

Output: the updated node cluster labeling L^t where each node label has been changed on the base of Hungarian alignment

1 **Let** CM be the $|\mathcal{CS}^{t-1}| \times |\mathcal{CS}^t|$ confusion matrix between L^{t-1} and L^t
2 **Execute** the Hungarian algorithm to obtain the match matrix MM from CM
3 $Corr = \emptyset$
4 **for** $j = 1, \ldots, |\mathcal{CS}^t|$
5 Let i be the class label at timestamp $t-1$ corresponding to j in MM, i.e. such that $MM_{ij} = 1$
6 Update L^t by substituting all the class labels j with i
7 $Corr = Corr \cup < i, j >$
8 **end for j**
9 **Renumber** L^t in progressive order