A Survey on Overlapping Community Detection

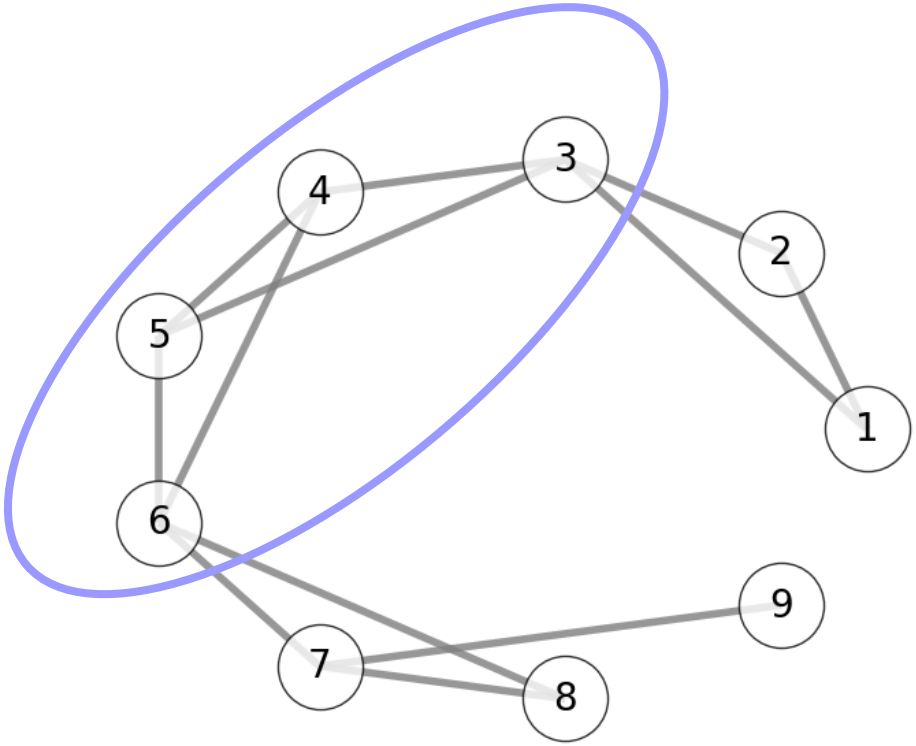
Algorithms and Evaluation Metrics

Presenter: Yulin Che
Supervisor: Qiong Luo

Content

Community

- **Community:** densely connected vertices



Community Definitions

- **Community:** No rigorous definition, mainly two alternative ways to define a community

- **Way1: Measure Maximization**
  - Local fitness function(measure the local structure density)
  - Global fitness function(global modularity to measure the community structure)

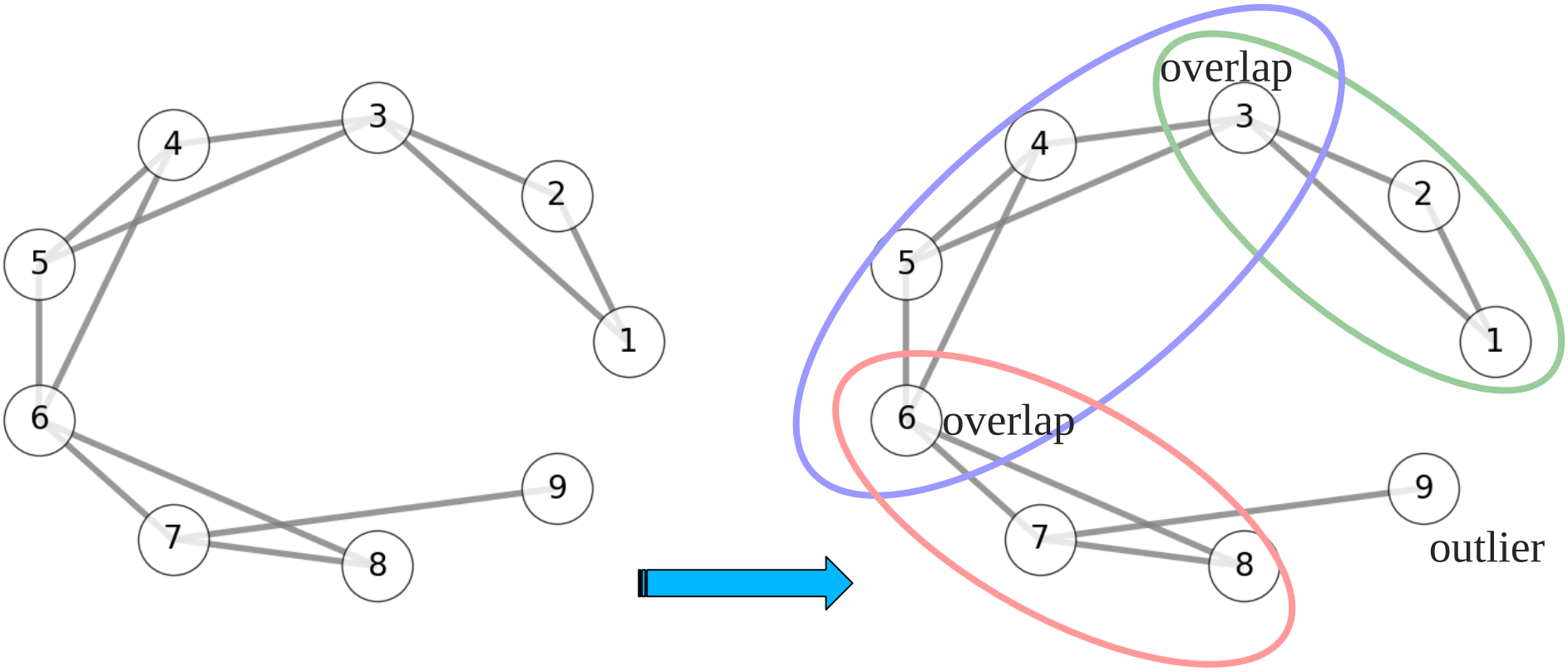
- **Way2: Heuristics**
  - Grow from a small group of densely connected vertices
  - Propagate community labels

Overlapping versus Disjoint Communities

- **Disjoint Communities :** a single vertex belongs to at most one community
- **Overlapping Communities:** a single vertex belongs to possibly multiple communities
- **Why Overlapping:** Sharing many things in common, e.g., regions, topics, hobbies

Community Detection

- **Input:** graph
- **Output:** a list of communities
- **Post-Processing**: get community labels, evaluate the quality



Input Graphs

- **Real-world graphs**
  - snap datasets: http://snap.stanford.edu/data/index.html
  - uci datasets: https://networkdata.ics.uci.edu/about.php

- **Synthetic graphs**
  - **GN benchmark:** generate the graph from communities by linking intra-community vertices with a higher probability than inter-community vertices.
  - **LFR benchmark:** generate the graph with social network features
    - small world, 6-hop theory
    - scale free, vertex degree follows power-law distribution.

Lancichinetti A, Fortunato S, Radicchi F. Benchmark graphs for testing community detection algorithms. Physical review E, 2008, 78(4): 046110.

Application Domains

- **Social Network**

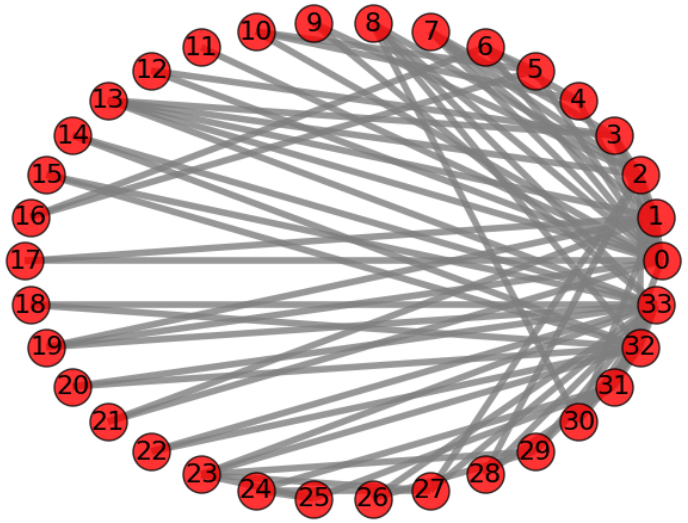
- **Content Sharing**

- **Blogs**

- **Wiki/Forum**

Example : Karate Club Friendship

- **Vertex**: Karate club member
- **Edge**: member friendship

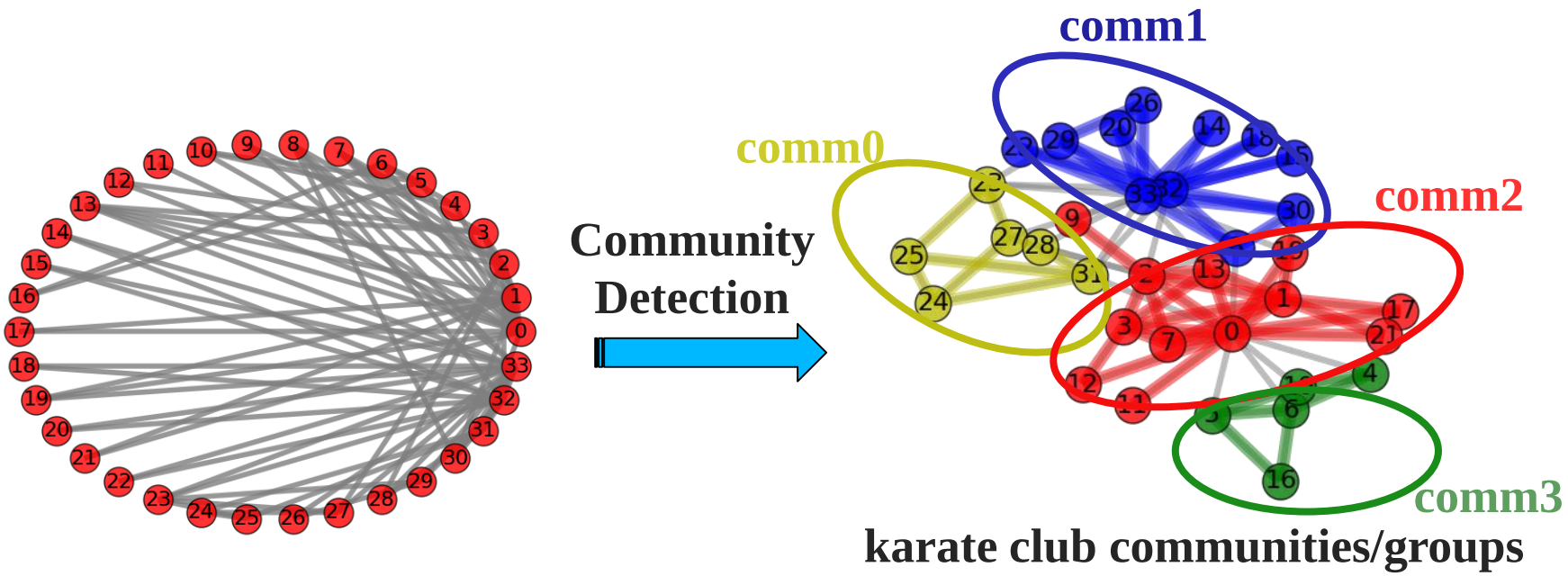


- **Applications:**
  - Understand the interactions among members
  - Help the network visualization

http://networkdata.ics.uci.edu/data/karate/

Example : Karate Club Friendship

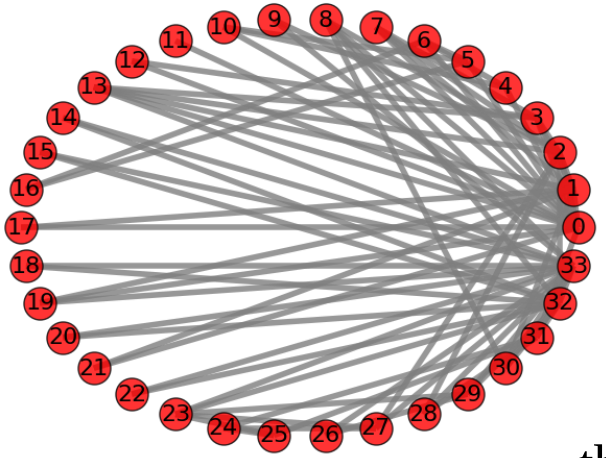
- **Applications:**
  - **Understand the interactions among members**
    - grey edges: weak **inter-community** interactions
    - other color edges: strong **intra-community** interactions
  - Help the network visualization

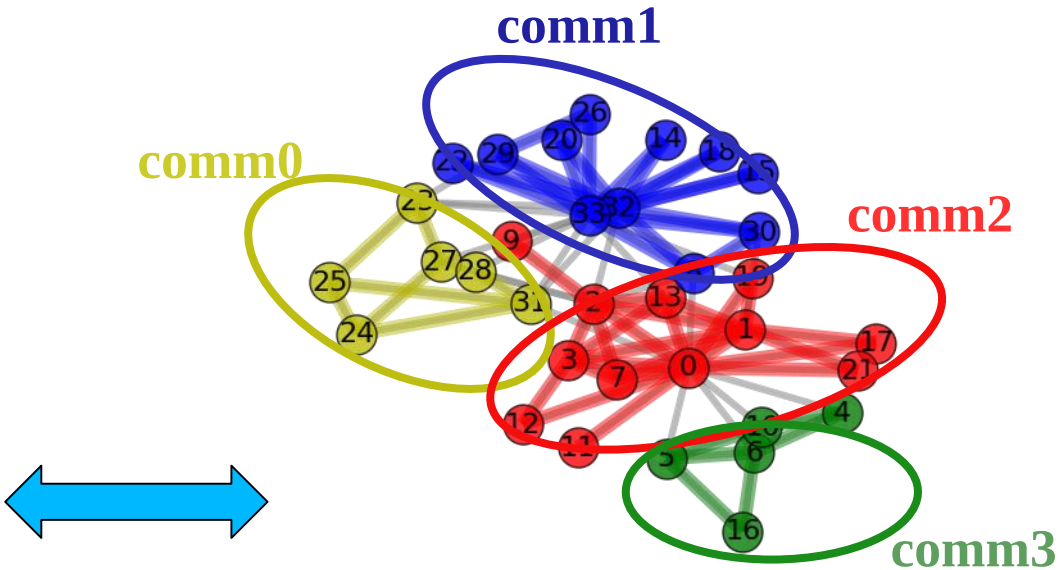


karate club communities/groups

Example : Karate Club Friendship

- **Applications:**
  - Understand the interactions among members
  - **Help the network visualization**
    - clustering vertices in the same community
      - grey edges(**inter-community**): sparse
      - other color edges(**intra-community**): dense





the same underlying graph

Content

1、Background

2、Algorithms

3、Evaluations Metrics

4、Summary & Discussion

Algorithm Category

- **Categorization**: based on algorithm design ideas**.**
  - Clique Percolation
  - Link Partition
  - Local Expansion
  - Dynamics
  - Statistical Inference(mainly by researchers in statistics, not included in this survey)

Xie J, Kelley S, Szymanski B K. Overlapping community detection in networks: The state-of-the-art and comparative study. Acm computing surveys (csur), 2013, 45(4): 43.

Category1: Clique Percolation

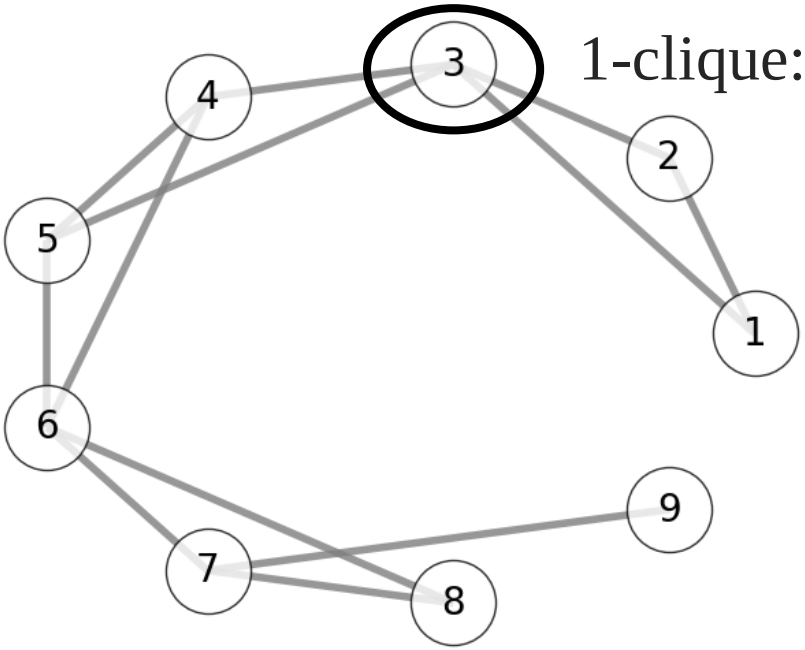
- **k-Clique**: a complete graph of k vertices
- **Percolation**: two k-cliques sharing enough common vertices will influence each other

- **Step1: Clique Percolation**
  - construct vertices of the k-clique graph
  - construct edges of the k-clique graph(percolate if two vertices in k-clique graph have strong connections)

- **Step2: Community Extraction**
  - find connected components
  - Set-union vertices within each connected component to get a new community

Category1: Clique Percolation

- **k-Clique**: a complete graph of k vertices

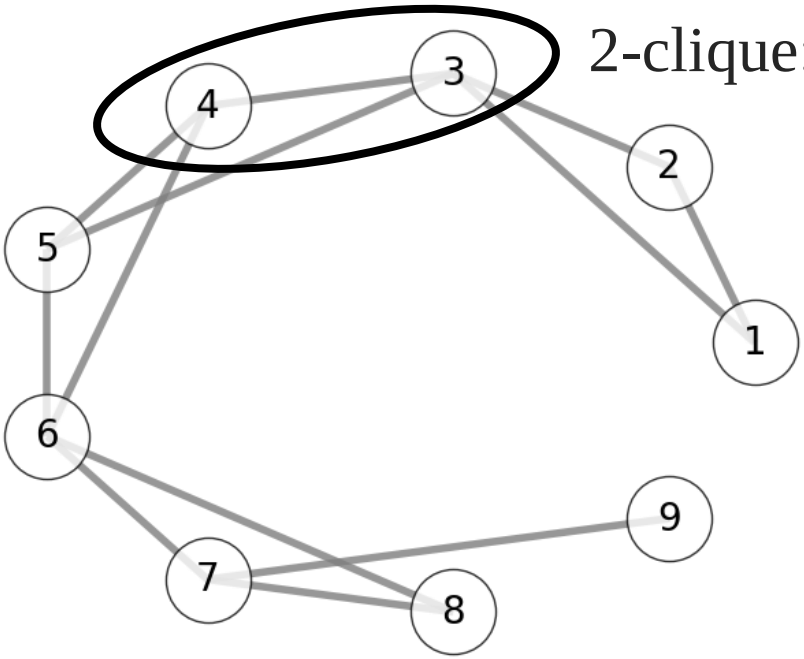


1-clique: a single vertex

input graph

Category1: Clique Percolation

- **k-Clique**: a complete graph of k vertices

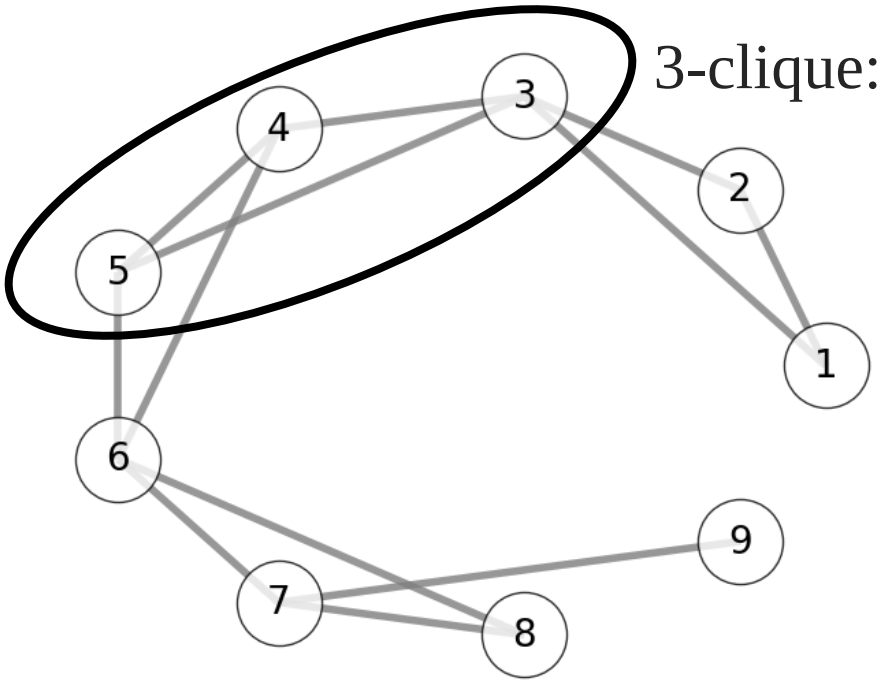


2-clique: two vertices and an edge

input graph

Category1: Clique Percolation

- **k-Clique**: a complete graph of k vertices

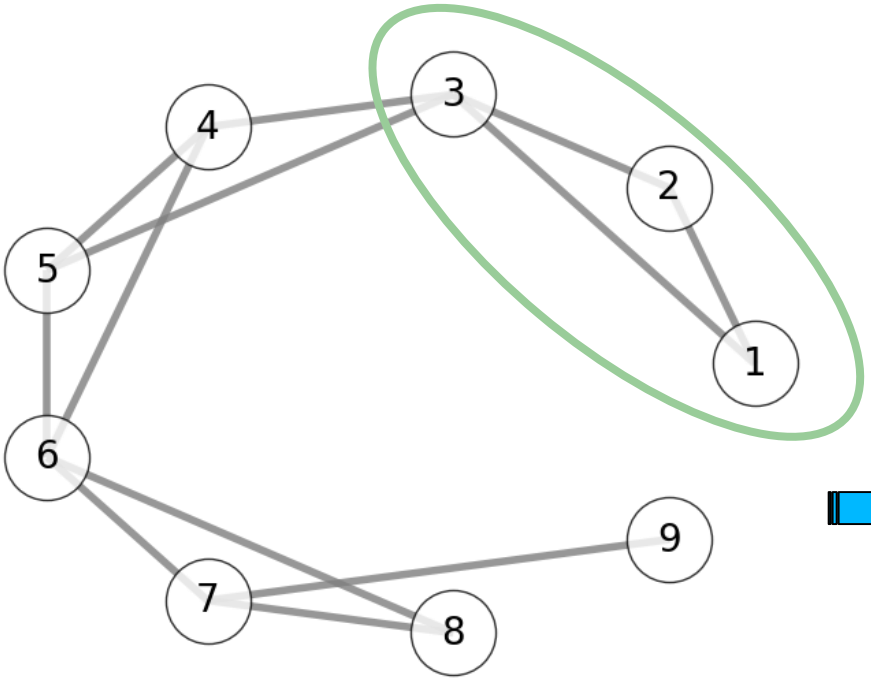


3-clique: three vertices and three edges

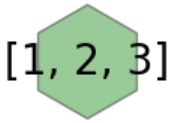
input graph

Category1: Clique Percolation

- **Clique Percolation** : construct the 3-clique graph
  - **construct vertices**
  - construct edges (percolation)



input graph

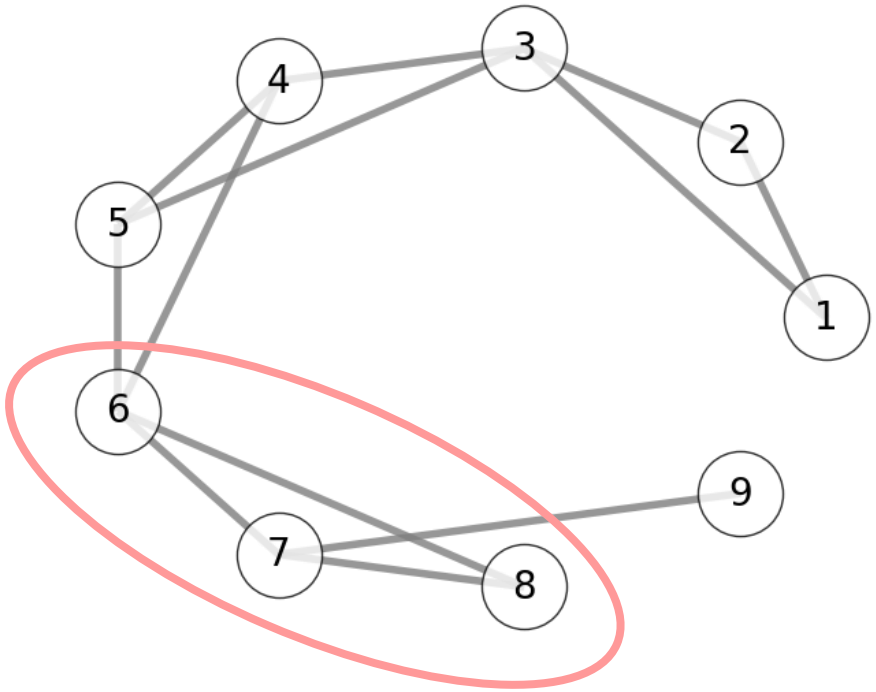


3-clique graph

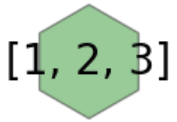
https://gist.github.com/conradlee/1341940/5011dd087c29e22bada85010e0dba6bbd459c5be

Category1: Clique Percolation

- **Clique Percolation** : construct the 3-clique graph
  - **construct vertices**
  - construct edges(percolation)



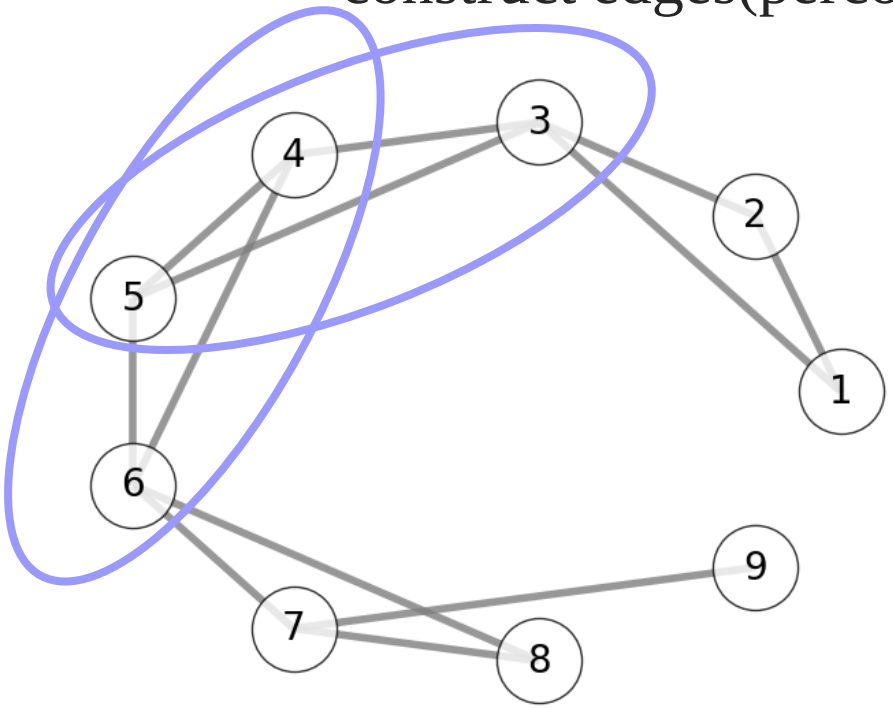
input graph



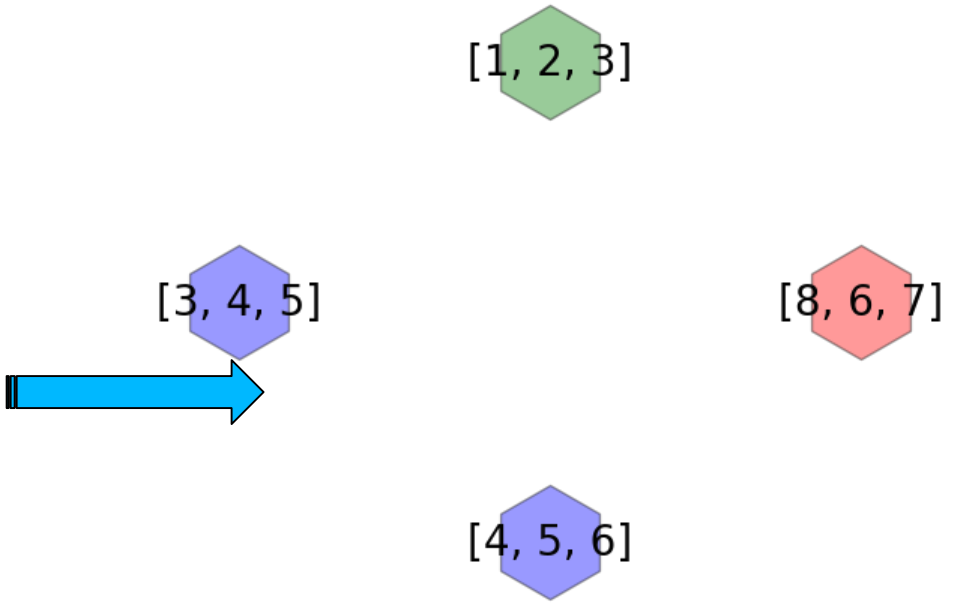
3-clique graph

Category1: Clique Percolation

- **Clique Percolation** : construct the 3-clique graph
  - **construct vertices**
  - construct edges(percolation)



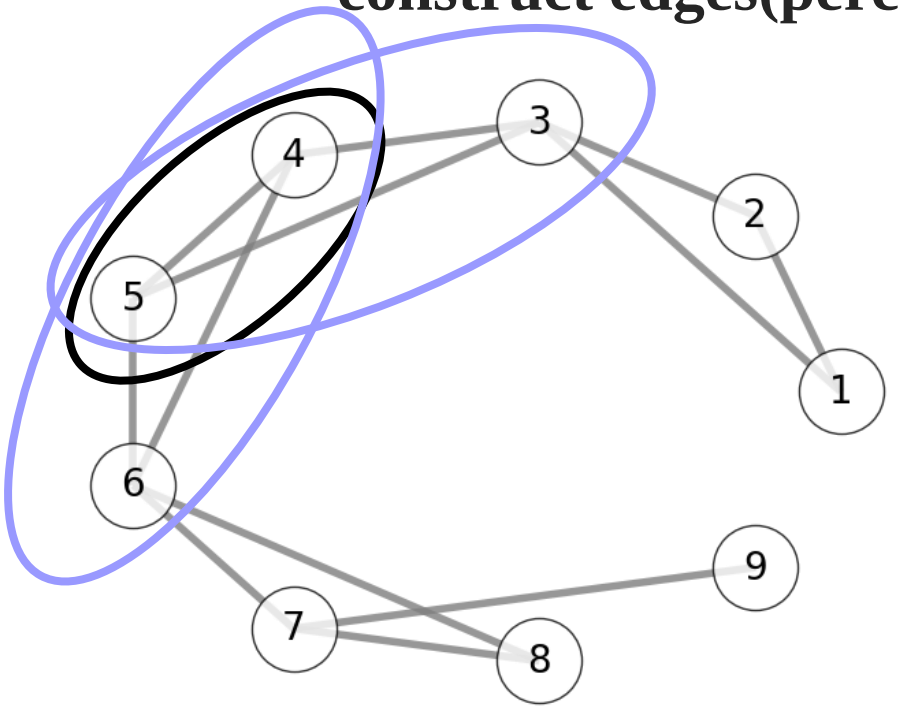
input graph



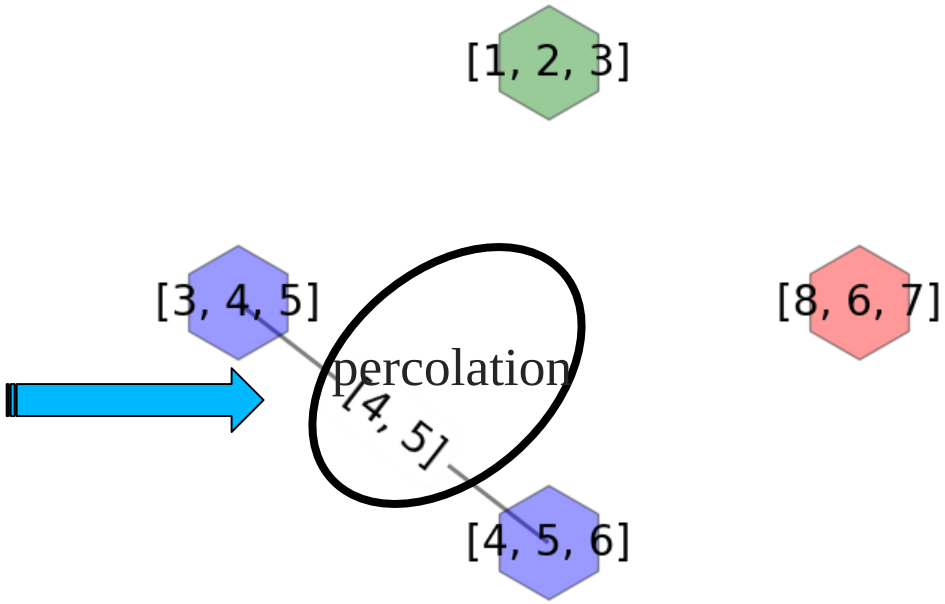
3-clique graph

Category1: Clique Percolation

- **Clique Percolation** : construct the 3-clique graph
  - construct vertices
  - **construct edges(percolation)**



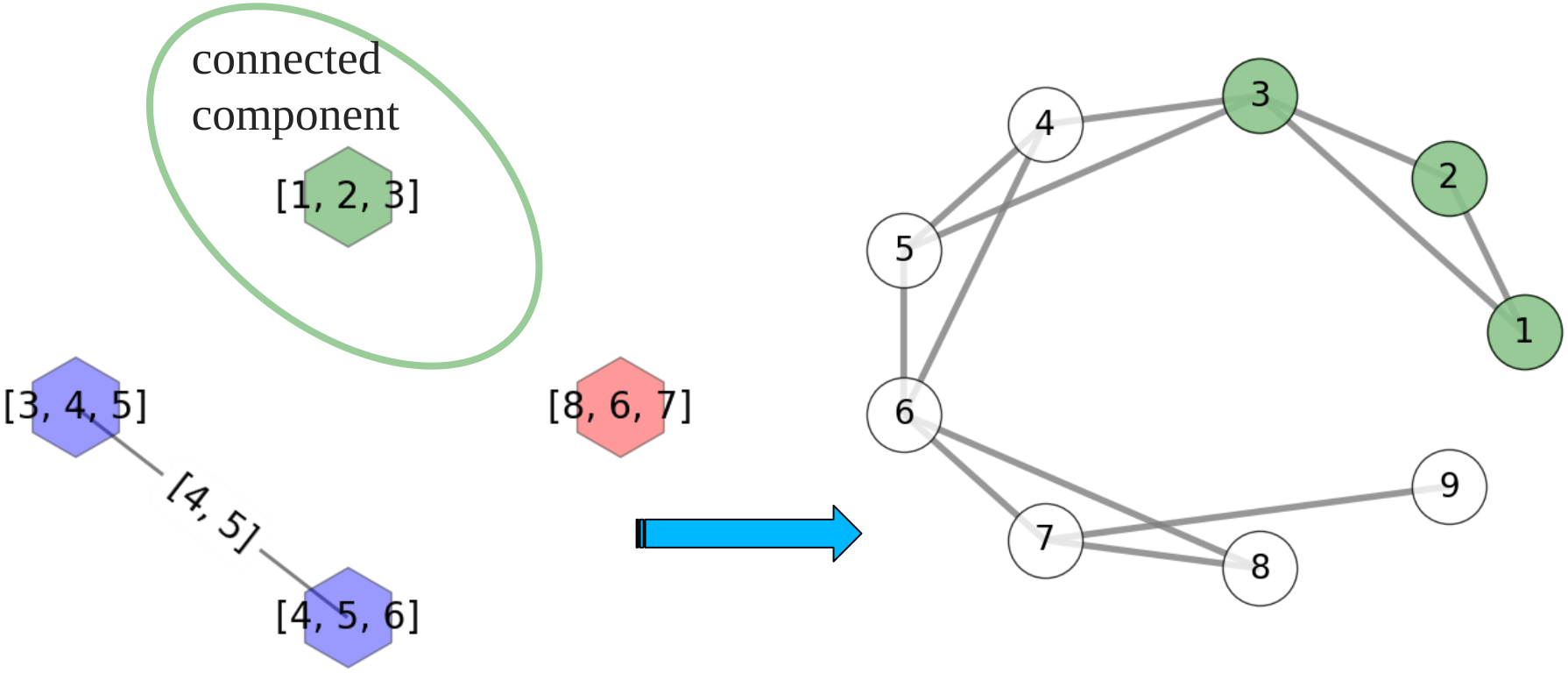
input graph



3-clique graph

Category1: Clique Percolation

- **Step2: Community Extraction**

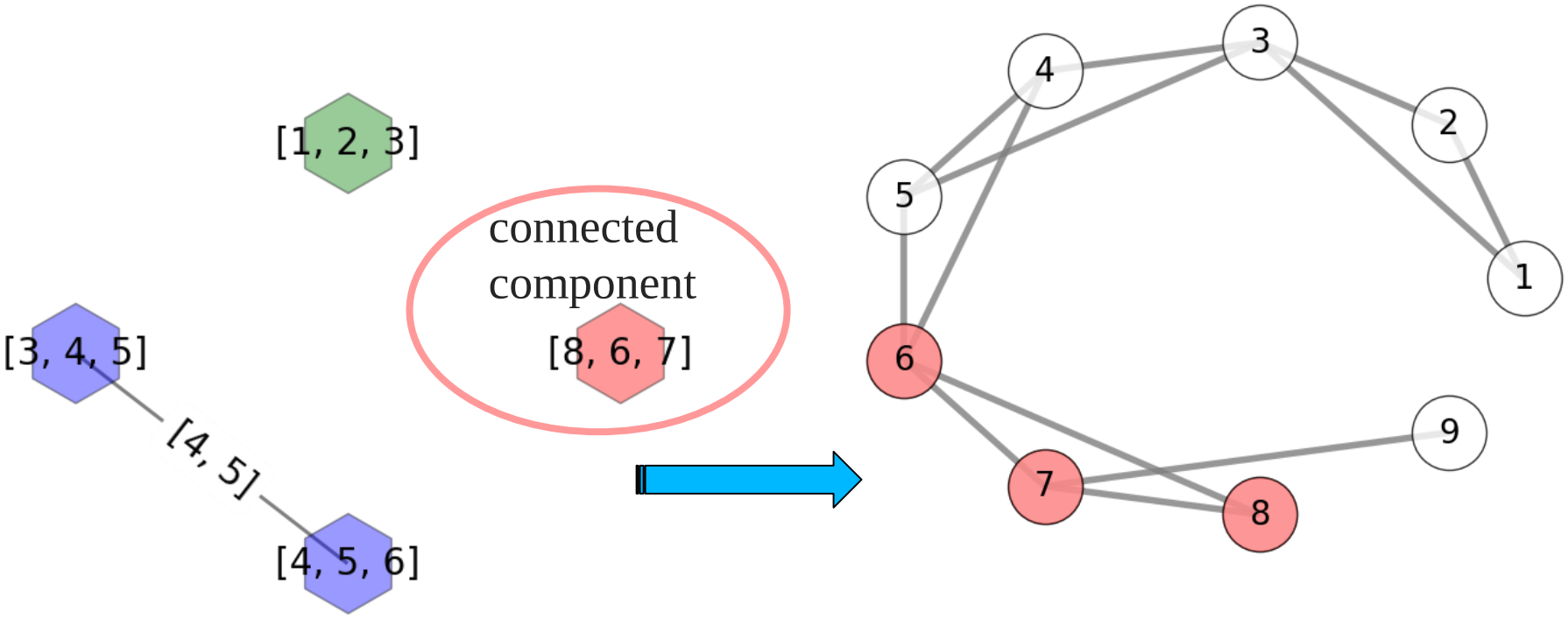


3-clique graph

extracted community

Category1: Clique Percolation

- **Community Extraction**



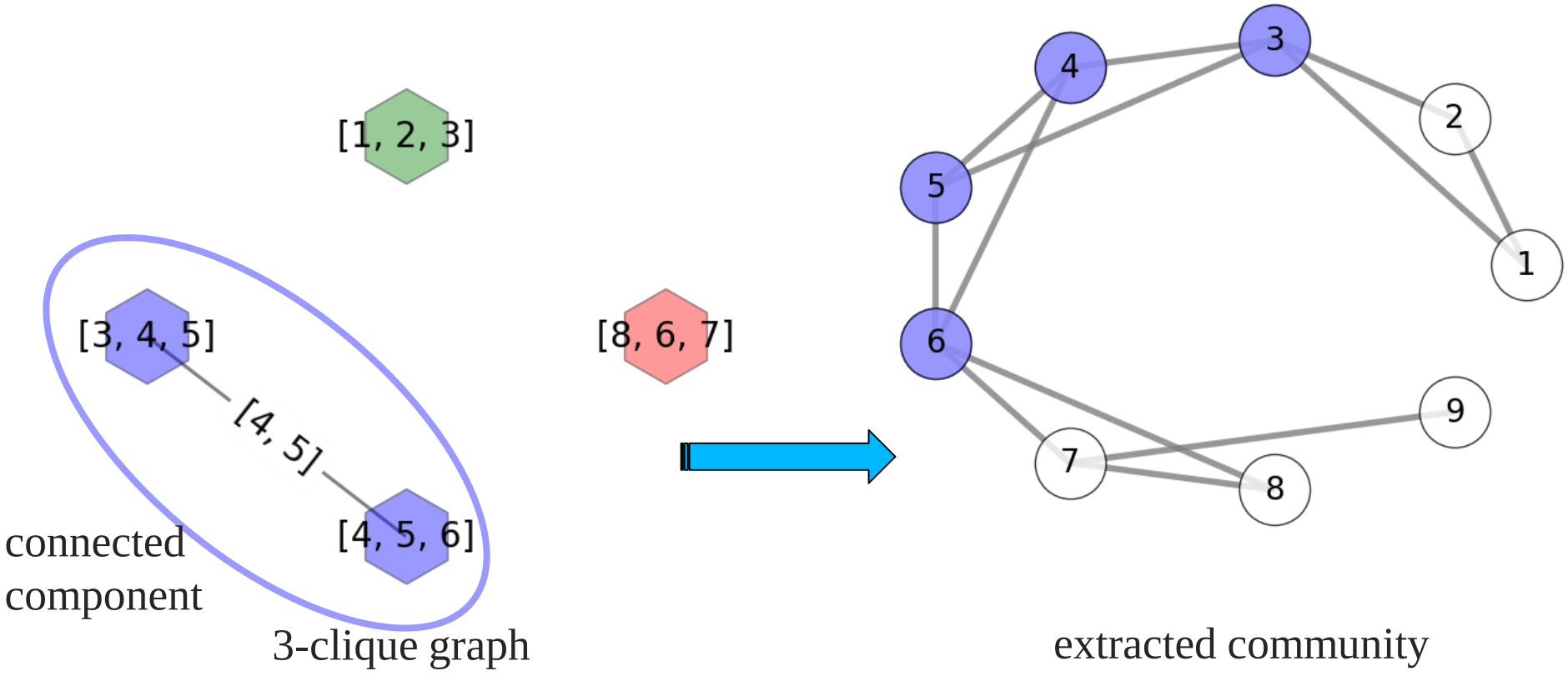
3-clique graph

extracted community

Category1: Clique Percolation

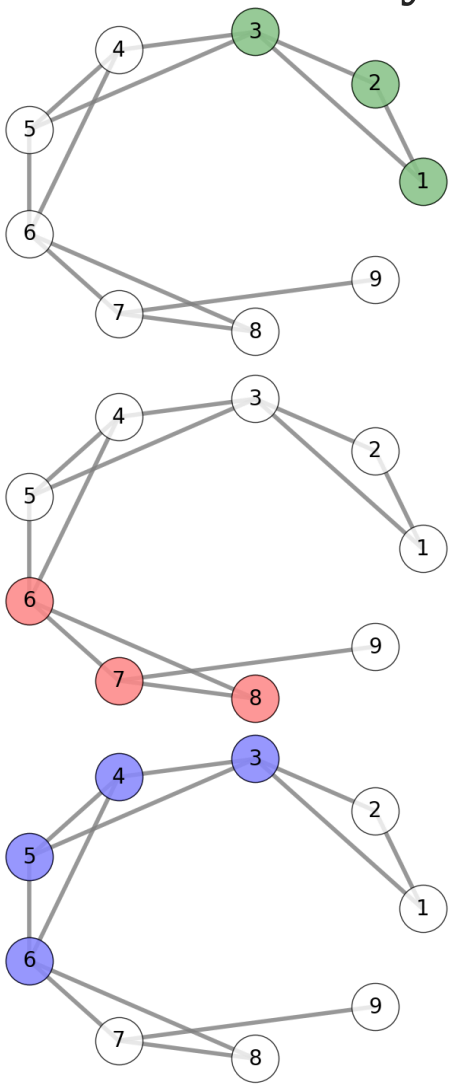
- **Community Extraction**

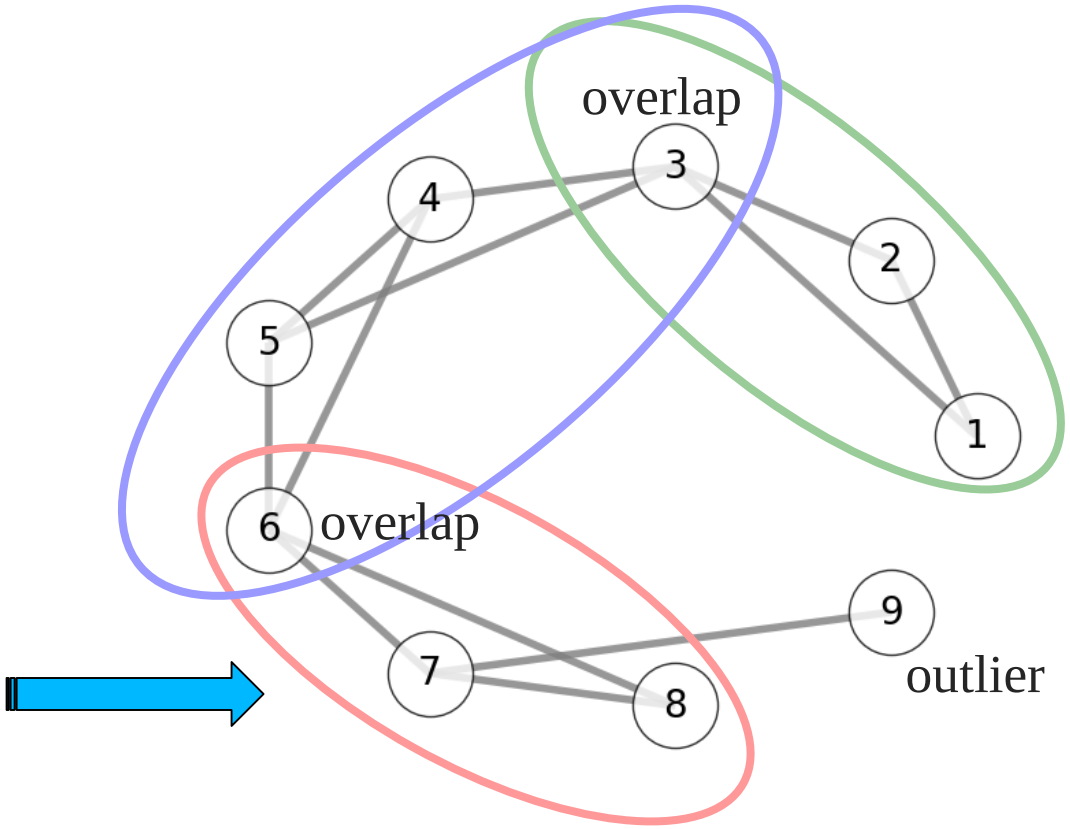
set_union({3,4,5},{4,5,6}) = {3,4,5,6}



Category1: Clique Percolation

- **Community Detection Result**





overlapping communities

Category2: Link Partition

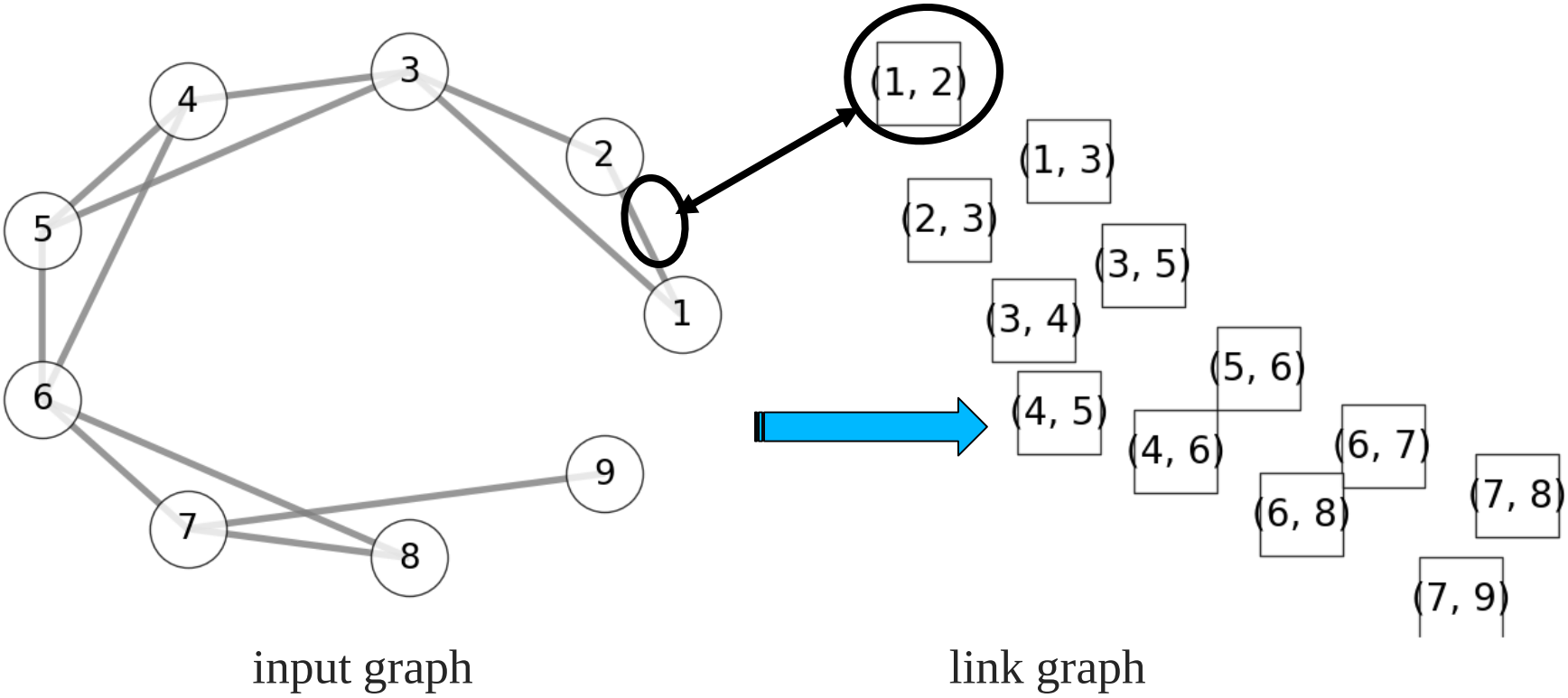
- **Link:** edge in the original graph
- **Link Partition:** partition the links in the original graph into several communities

- **Step1: Two Approaches for Link Partition**
  - extract the link graph and apply graph partition algorithms on that, or
  - directly cluster the links via some link community similarity function

- **Step2: Community Extraction:** assign the link community label to the source and destination vertices of the link

Category2: Link Partition

- **Approach1: Link Graph Extraction**
  - **construct link graph vertices**
  - construct link graph edges

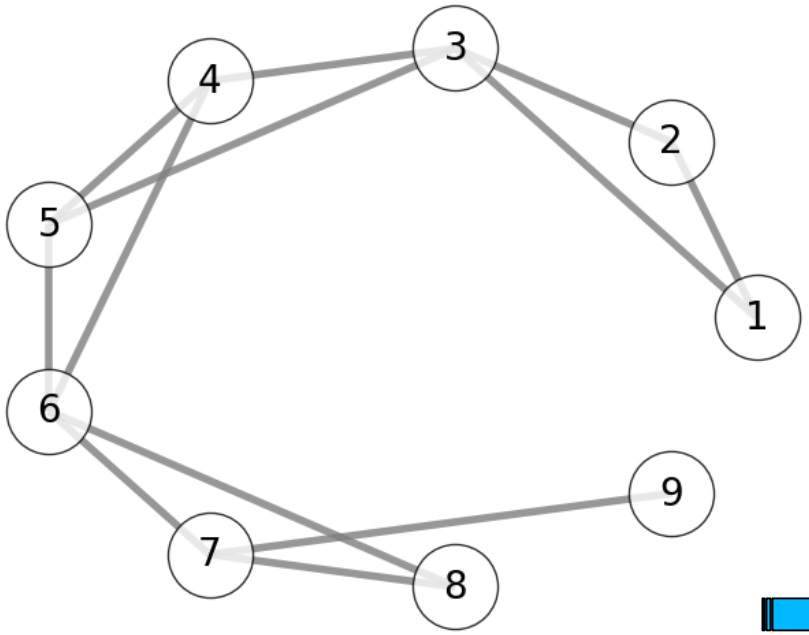


input graph

link graph

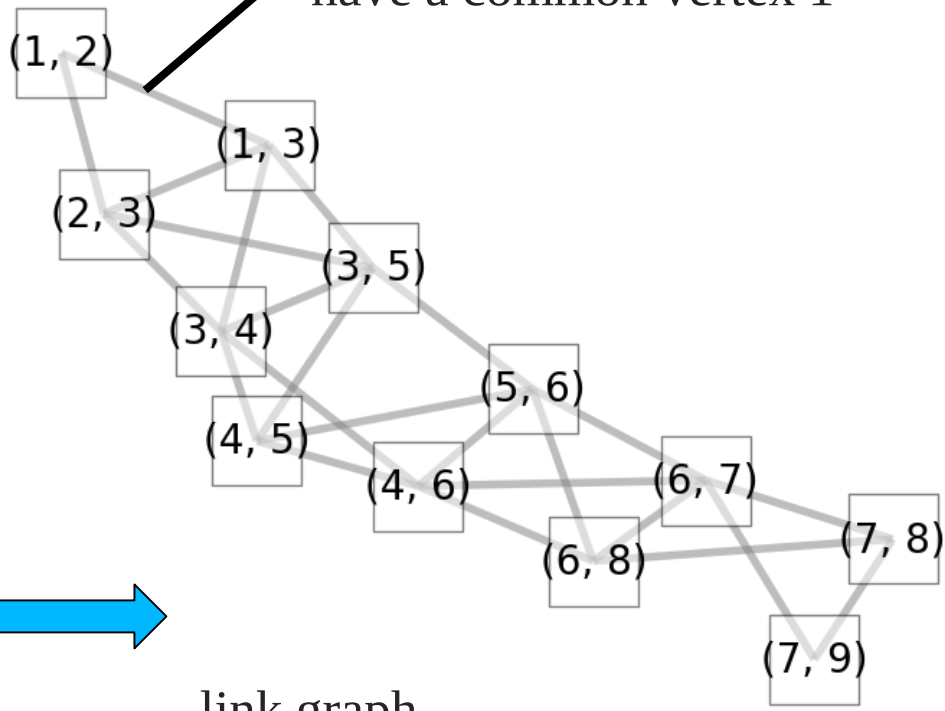
Category2: Link Partition

- **Approach1: Link Graph Extraction**
  - construct link graph vertices
  - **construct link graph edges**



input graph

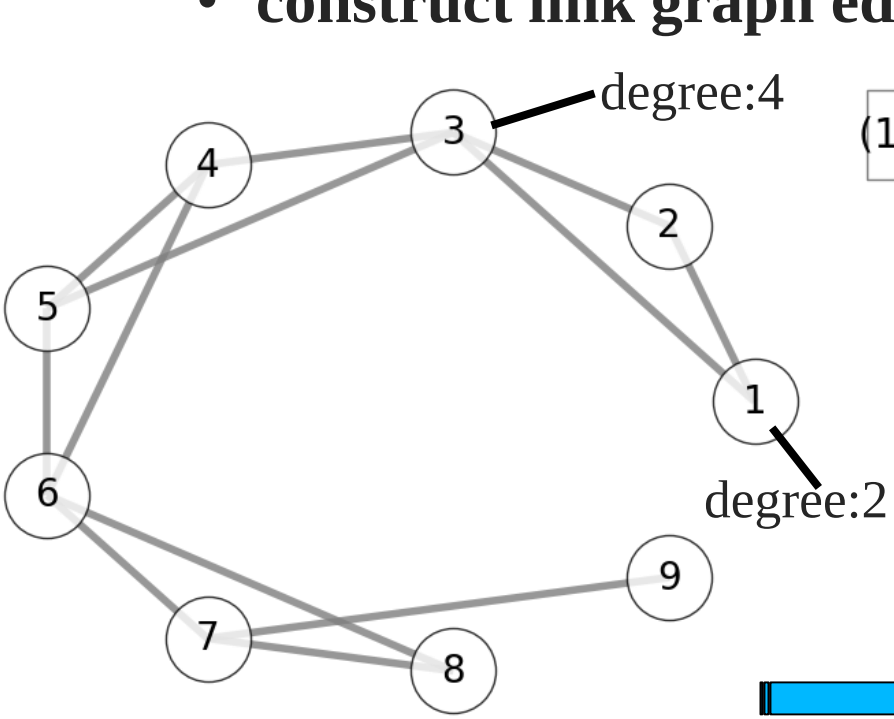
edge(1,2) and edge(1,3) have a common vertex 1



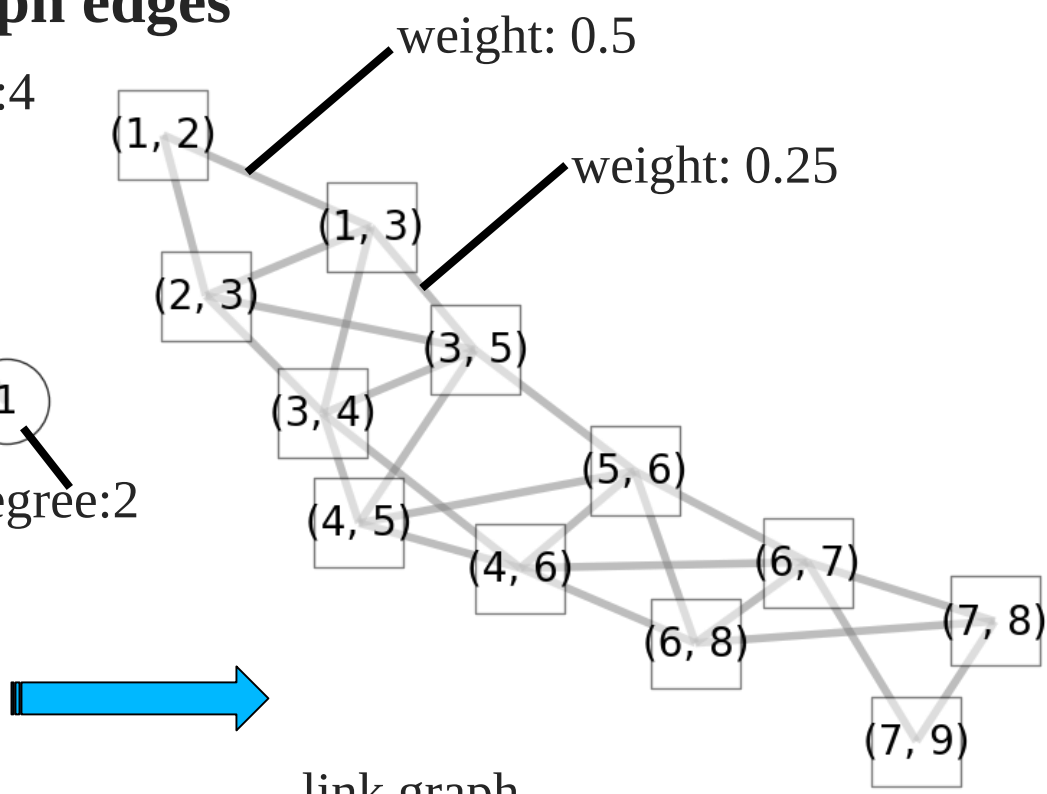
link graph

Category2: Link Partition

- **Approach1: Link Graph Extraction**
  - construct link graph vertices
  - **construct link graph edges**



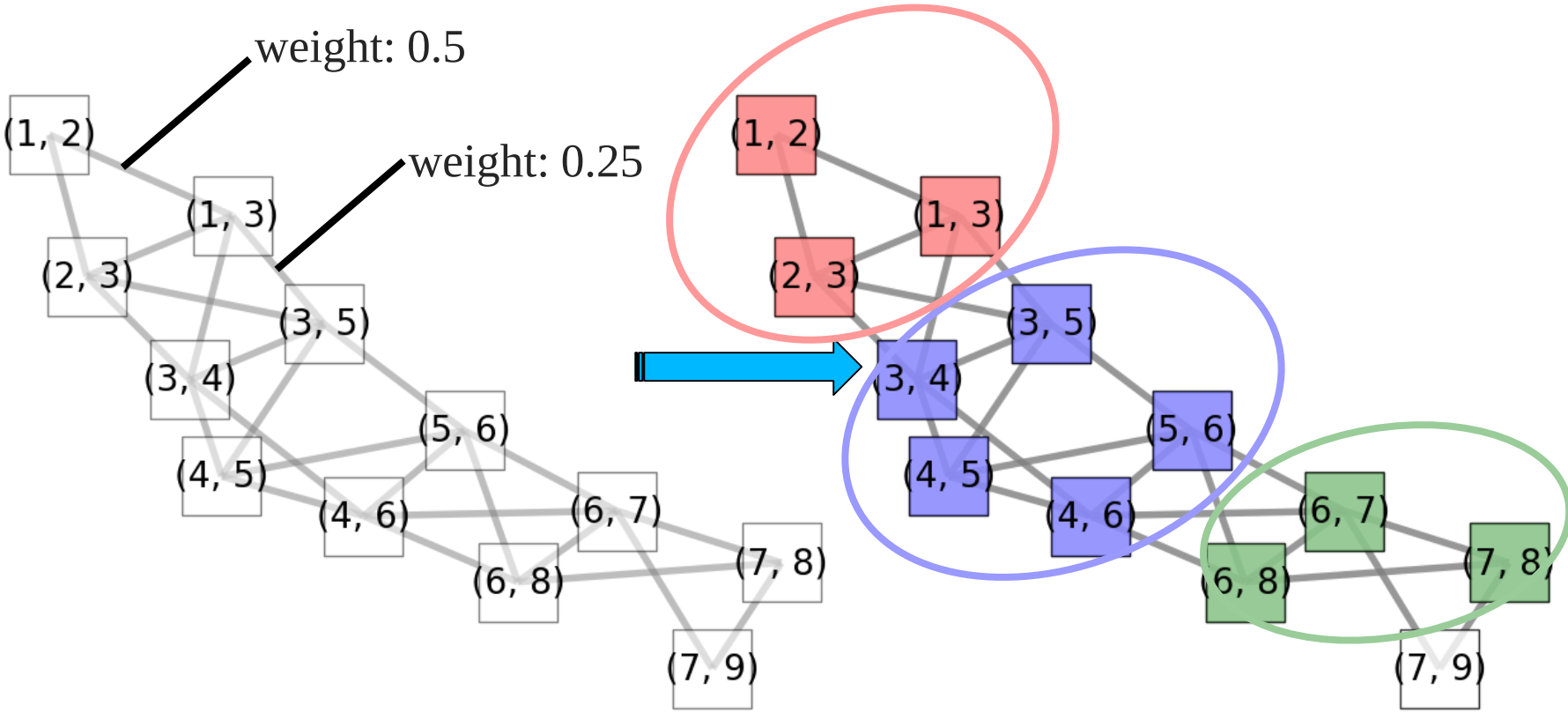
input graph



link graph

Category2: Link Partition

- **Approach1: Link Graph Extraction**
  - after all edges get properties, apply graph partition algorithms or disjoint community detection algorithms

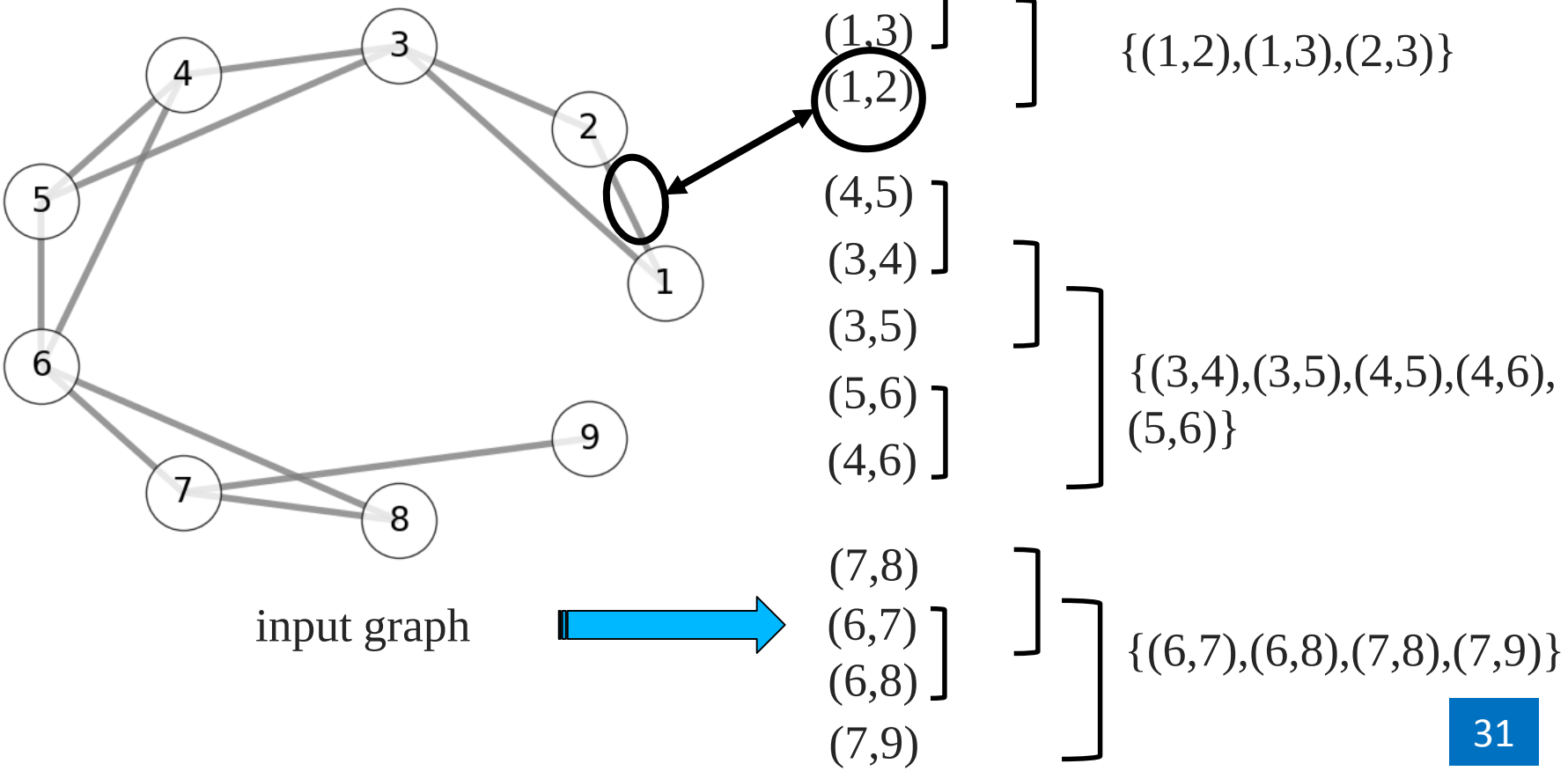


weighted link graph

disjoint link communities

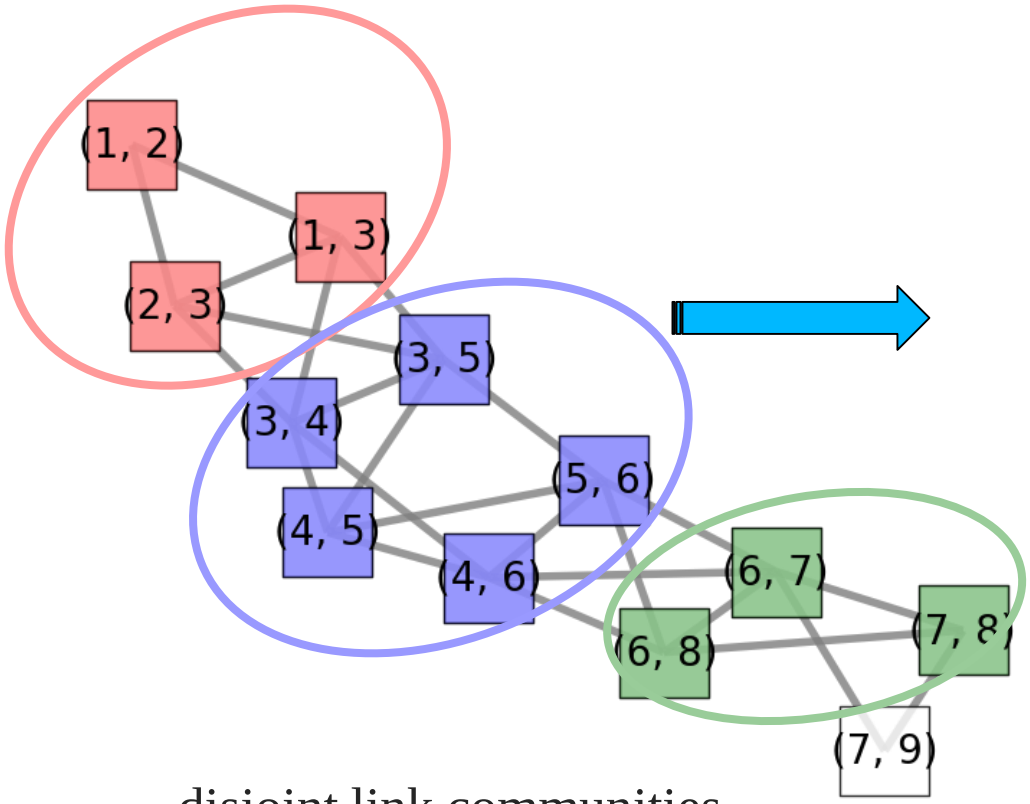
Category2: Link Partition

- **Approach2:** Cluster the links with similarity function
  - clusters are a list of edges
  - similarity function measures the correlations between two lists of edges



Category2: Link Partition

- **Step2: Community Extraction**
  - set union on link partitions



disjoint link communities

set_union(**{1,2},{1,3},{2,3}**)
={1,2,3}

set_union(**{3,4},{3,5},{4,5},{4,6}{5,6}**)
={3,4,5,6}

set_union(**{6,7},{6,8},{7,8}**)
={6,7,8}

communities:
{{1,2,3}, {3,4,5,6},{6,7,8}}

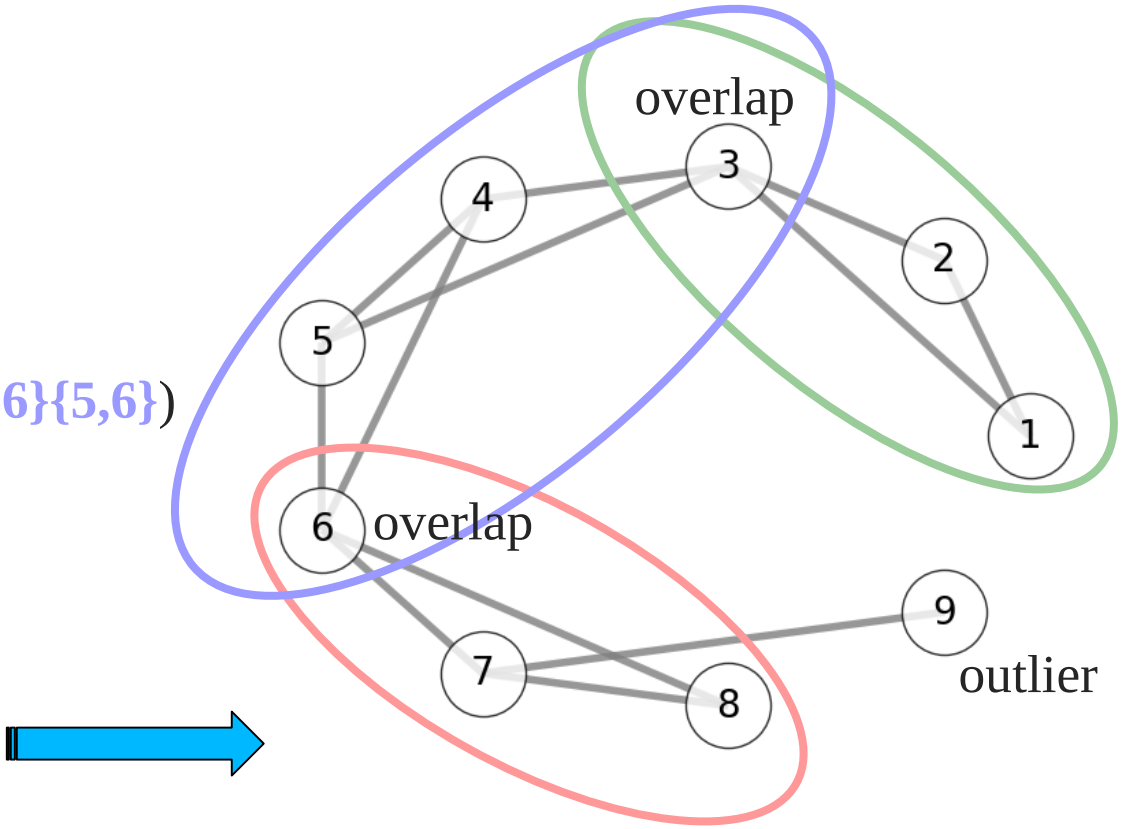
Category2: Link Partition

- **Community Detection Result**

set_union**(**{1,2},{1,3},{2,3}**)**
={1,2,3}

set_union(**{**3,4},{3,5},{4,5},{4,6}{5,6}**)**
={3,4,5,6}

set_union(**{**6,7},{6,8},{7,8}**)**
={6,7,8}



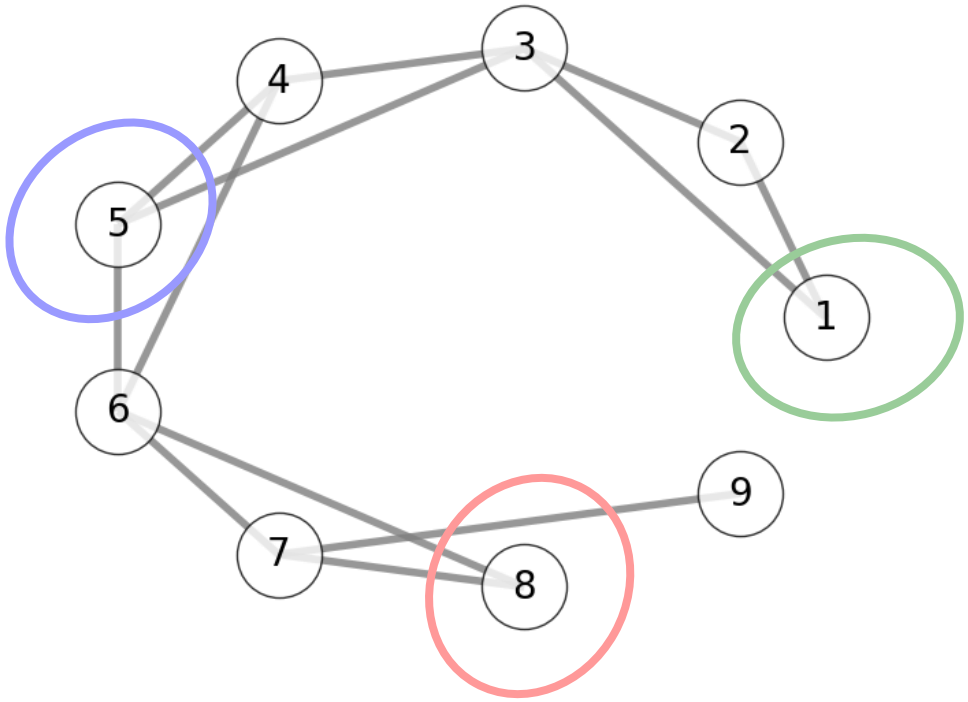
overlapping communities

Category3: Local Expansion

- **Step1: Select Seeds**
  - each seed: a list of influential vertices
- **Step2: Expand Seed**
  - compute local fitness values of expansion alternatives
  - apply some heuristics, to explore the topology of the graph
- **Step3: Merge Intermediate Communities**
  - merge intermediate communities into global communities

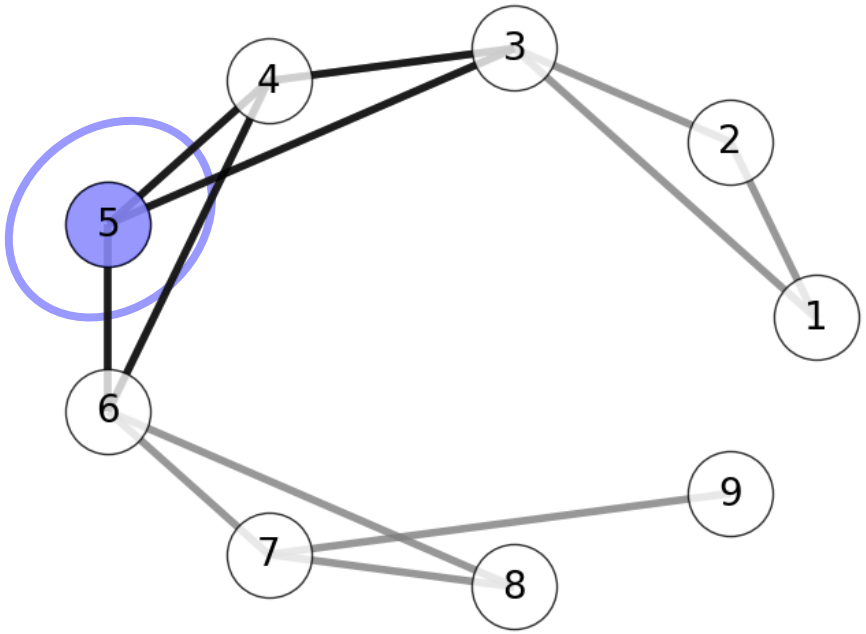
Category3: Local Expansion

- **Step1: Select Seeds**
  - for example, select {1}, {5}, {8} as a list of seeds for local expansions



Category3: Local Expansion

- **Step2: Expand Seed(Seed:{5})**
  - expand it through maximizing a local fitness function $w_{in}/(w_{in} + w_{out})$, via adding or removing members

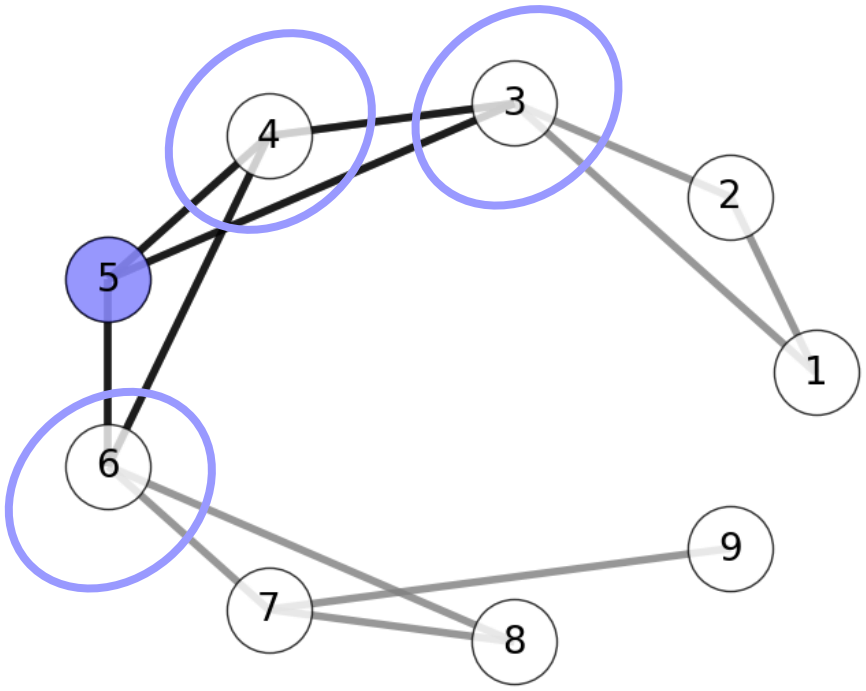


members:

- vertices selected to be members of a community
- at the beginning, members are vertices in the seed

Category3: Local Expansion

- **Expand Seed(Seed:{5})**
  - expand it through maximizing a local fitness function $w_{in}/(w_{in} + w_{out})$, via adding or removing members

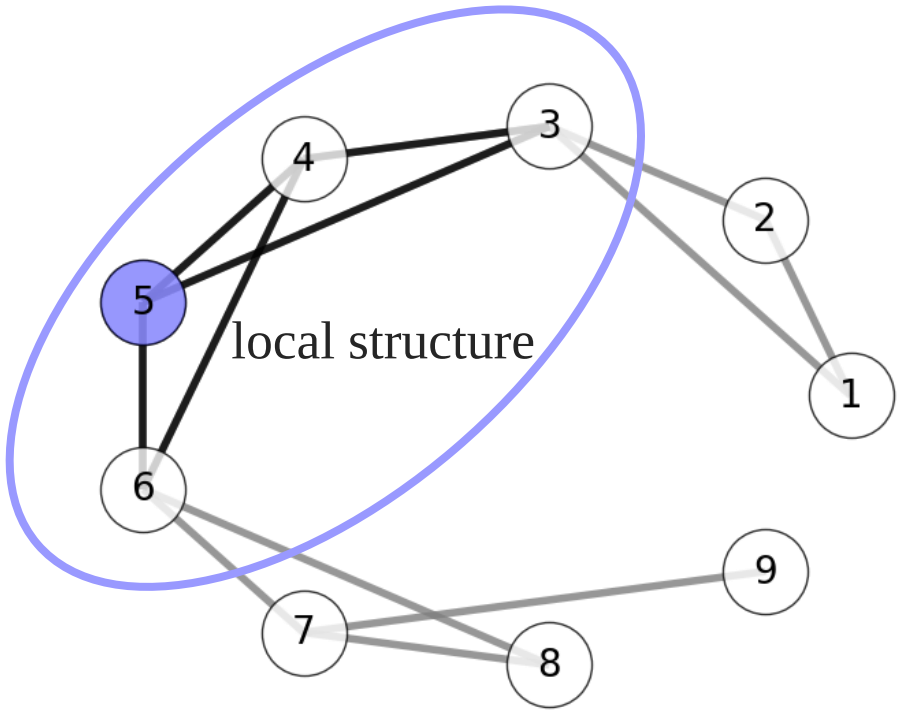


neighbors:

- adjacent vertices of members
- at the beginning, neighbors are the adjacent vertices of the vertices in the seed

Category3: Local Expansion

- **Expand Seed(Seed:{5})**
  - expand it through maximizing a local fitness function $w_{in}/(w_{in} + w_{out})$, via adding or removing members

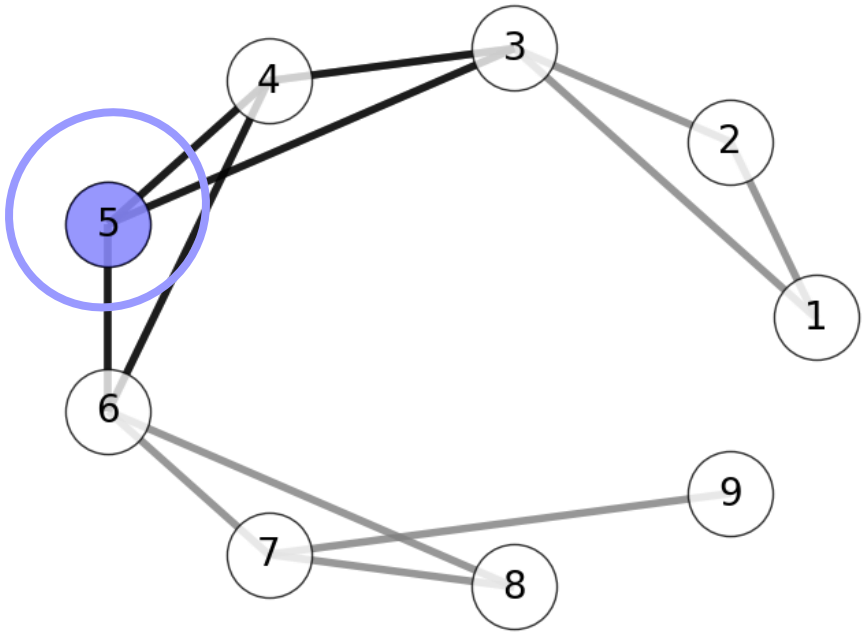


local fitness function :

- evaluate the local structure, whether possible to add or remove members to explore a better community
- w_{in} : edges among members
- $w_{in} + w_{out}$: edges among the union of members and neighbors

Category3: Local Expansion

- **Expand Seed(Seed:{5})**
  - at the beginning, no edges among the members, $w_{in}= 0$
  - black edges contribute to $w_{out,}$ $w_{out}= 5$

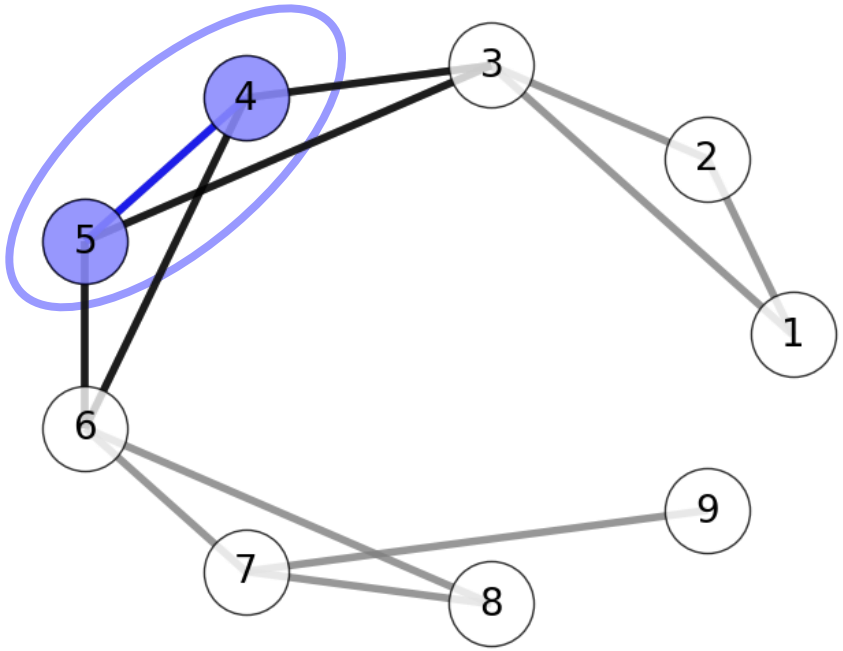


Intermediate Variables

- members: {5}
- neighbors: {3,4,6}
- w_{in}: 0
- w_{out}: 5
- $w_{in} + w_{out}$: 5
- fitness: 0/5

Category3: Local Expansion

- **Expand Seed (Seed:{5})**
  - add the member {4}

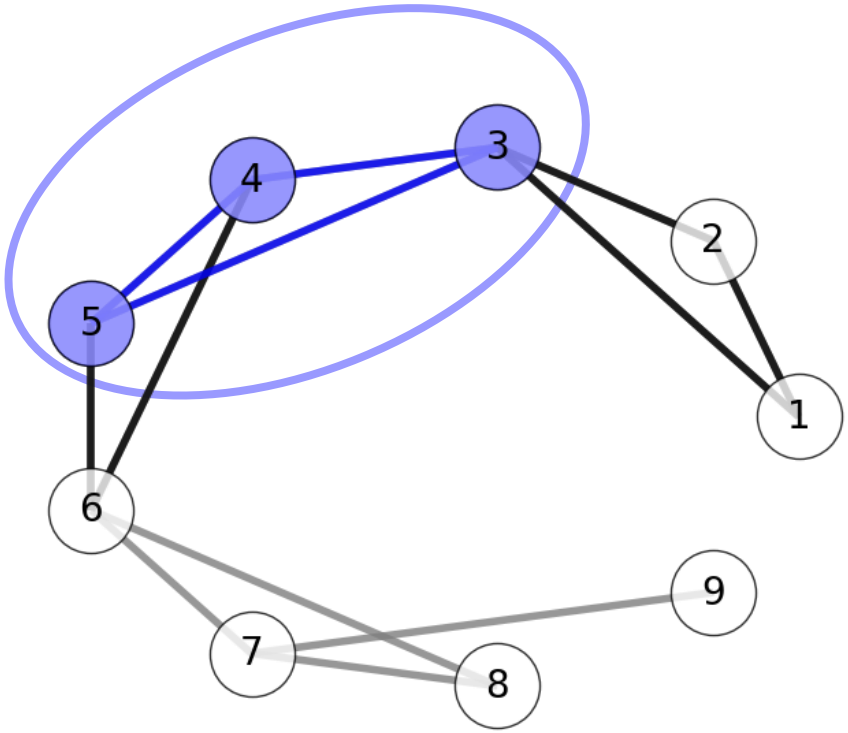


Intermediate Variables

- members: {4,5}
- neighbors: {3, 6}
- w_{in}: 1
- w_{out}: 4
- $w_{in} + w_{out}$: 5
- fitness: 1/5

Category3: Local Expansion

- **Expand Seed (Seed:{5})**
  - add the member {3}

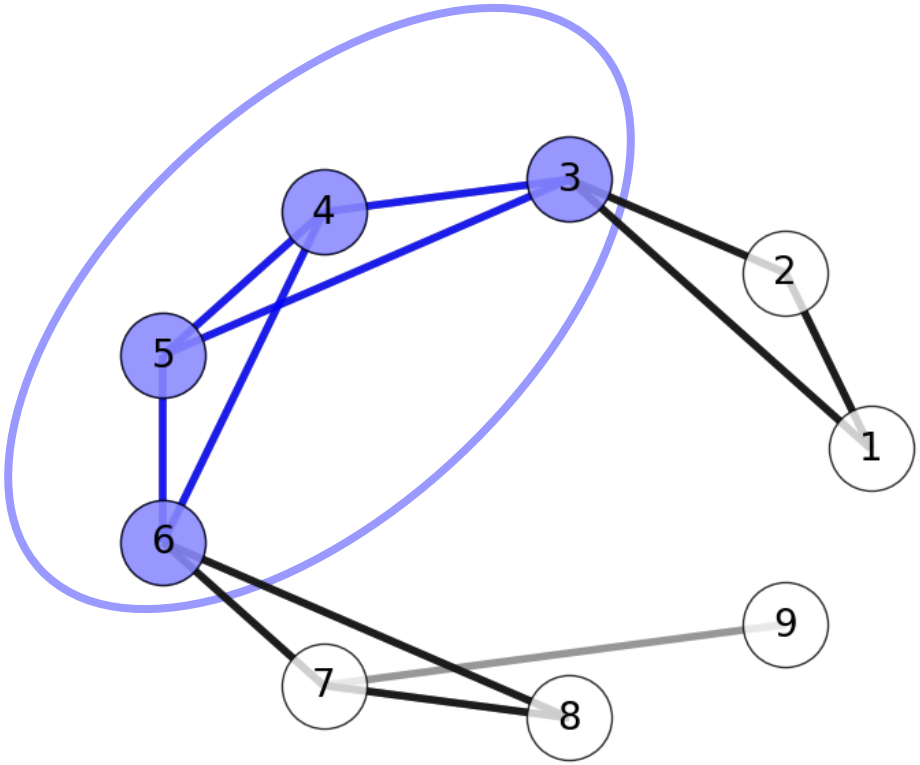


Intermediate Variables

- members: {3,4,5}
- neighbors: {1,2,6}
- w_{in}: 3
- w_{out}: 5
- $w_{in} + w_{out}$: 8
- fitness: 3/8

Category3: Local Expansion

- **Expand Seed (Seed:{5})**
  - add the member {6}

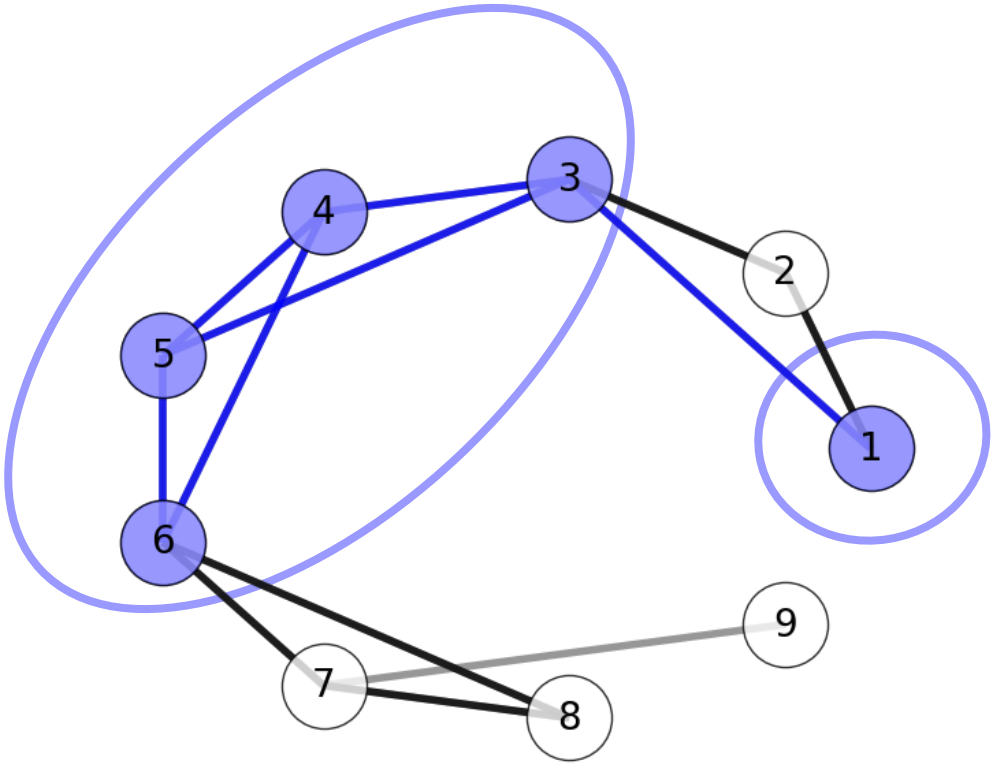


Intermediate Variables

- members: {3,4,5,6}
- neighbors: {1,2,7,8}
- w_{in}: 5
- w_{out}: 6
- $w_{in} + w_{out}$: 11
- fitness: 5/11

Category3: Local Expansion

- **Expand Seed (Seed:{5})**
  - add the member {1}

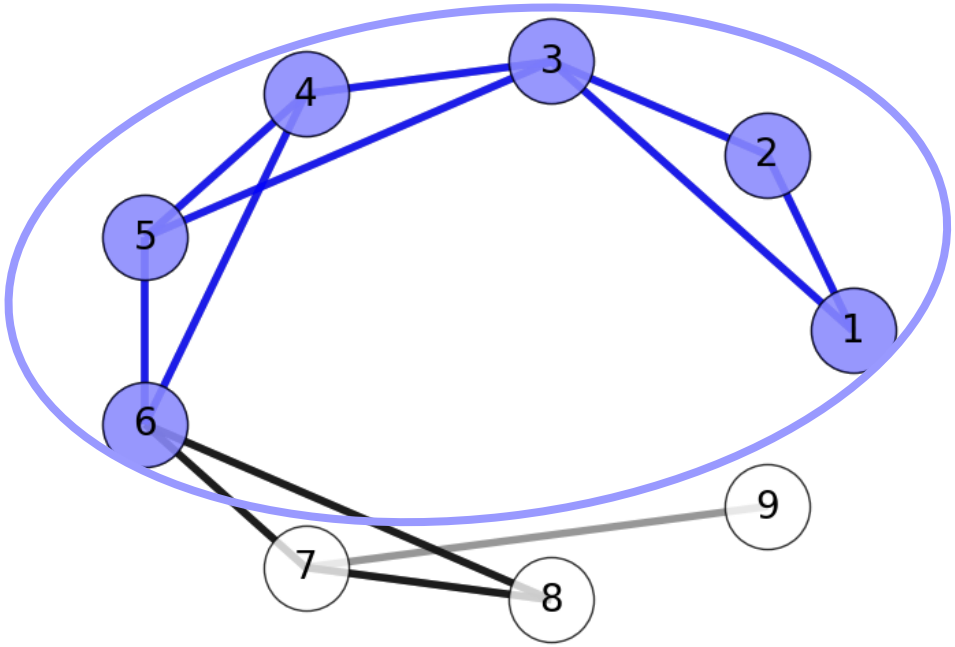


Intermediate Variables

- members: {1,3,4,5,6}
- neighbors: {7,8}
- w_{in}: 6
- w_{out}: 5
- $w_{in} + w_{out}$: 11
- fitness: 6/11

Category3: Local Expansion

- **Expand Seed**
  - add the member {2}

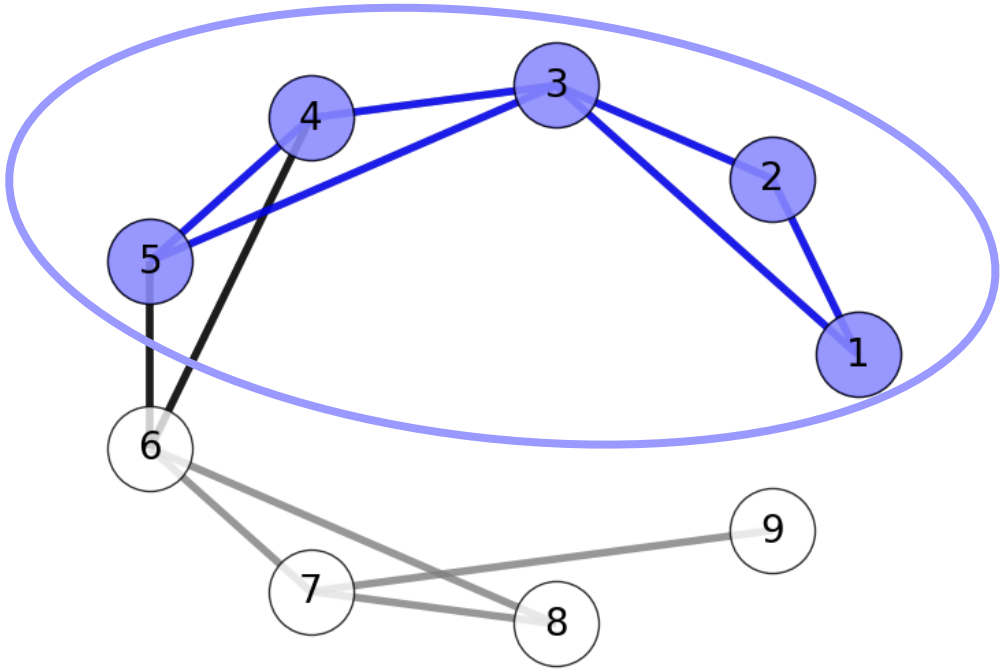


Intermediate Variables

- members: {1,2,3,4,5,6}
- neighbors: {7,8}
- w_{in}: 8
- w_{out}: 3
- $w_{in} + w_{out}$: 11
- fitness: 8/11

Category3: Local Expansion

- **Expand Seed(Seed:{5})**
  - remove the member {6}

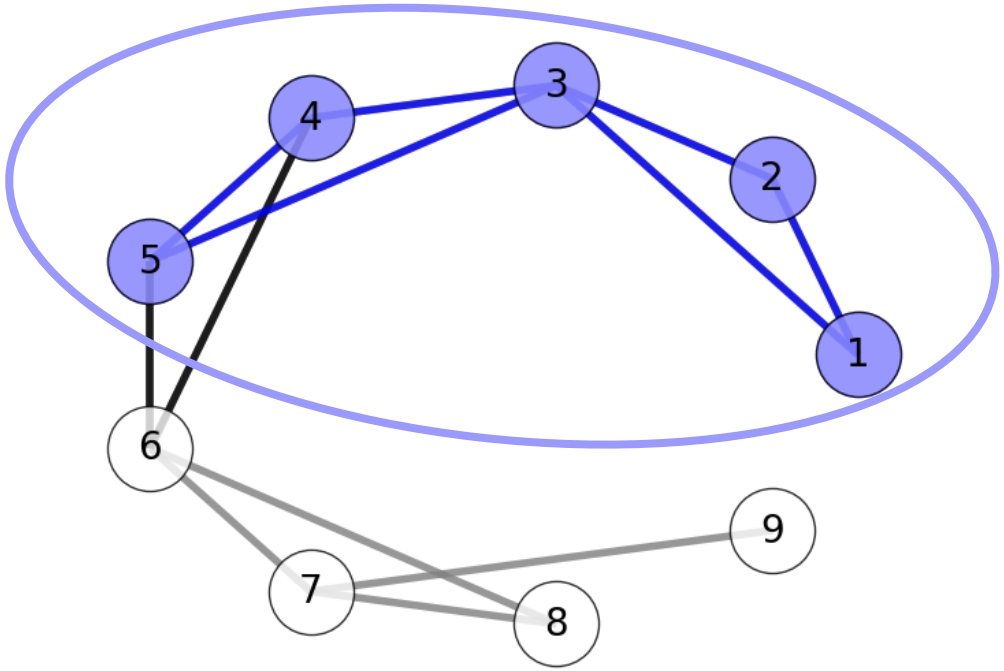


Intermediate Variables

- members: {1,2,3,4,5}
- neighbors: {6}
- w_{in}: 6
- w_{out}: 2
- $w_{in} + w_{out}$: 8
- fitness: 6/8

Category3: Local Expansion

- **Seed Expansion Result(Seed:{5})**
  - when the local fitness is not improved any more
  - reach the end of the seed expansion for the seed {5}

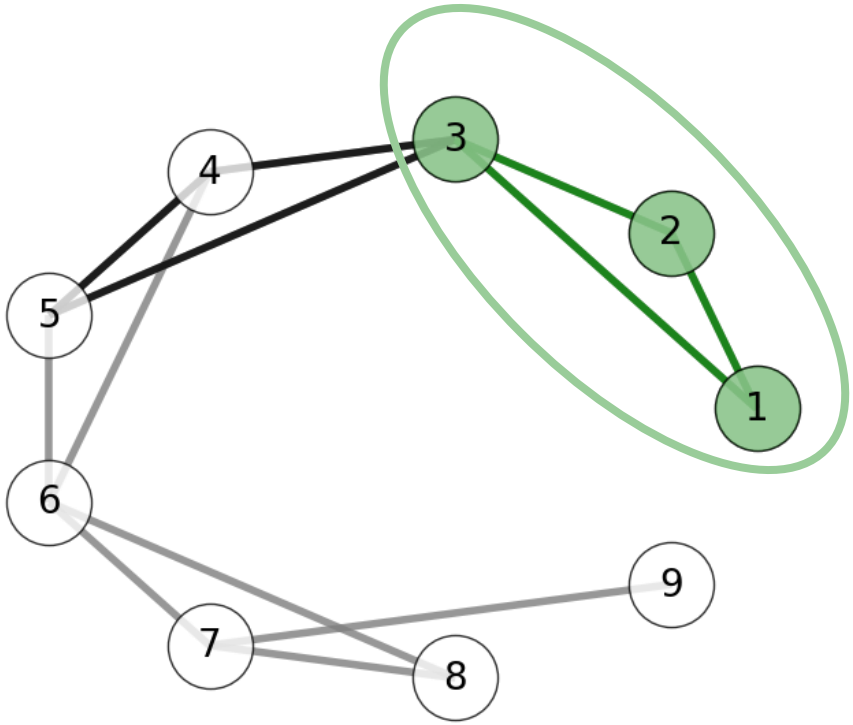


Intermediate Variables

- members: {1,2,3,4,5}
- neighbors: {6}
- w_{in}: 6
- w_{out}: 2
- $w_{in} + w_{out}$: 8
- fitness: 6/8

Category3: Local Expansion

- **Seed Expansion Result(Seed:{1})**

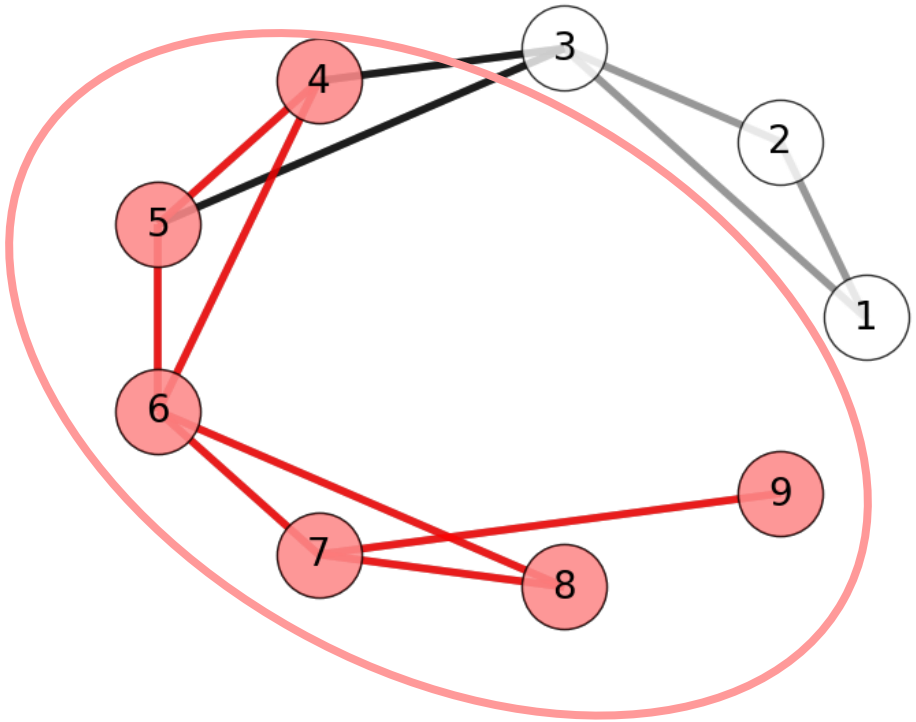


Intermediate Variables

- members: {1,2,3}
- neighbors: {4,5}
- w_{in}: 3
- w_{out}: 3
- $w_{in} + w_{out}$: 6
- fitness: 3/6

Category3: Local Expansion

- **Seed Expansion Result(Seed:{8})**

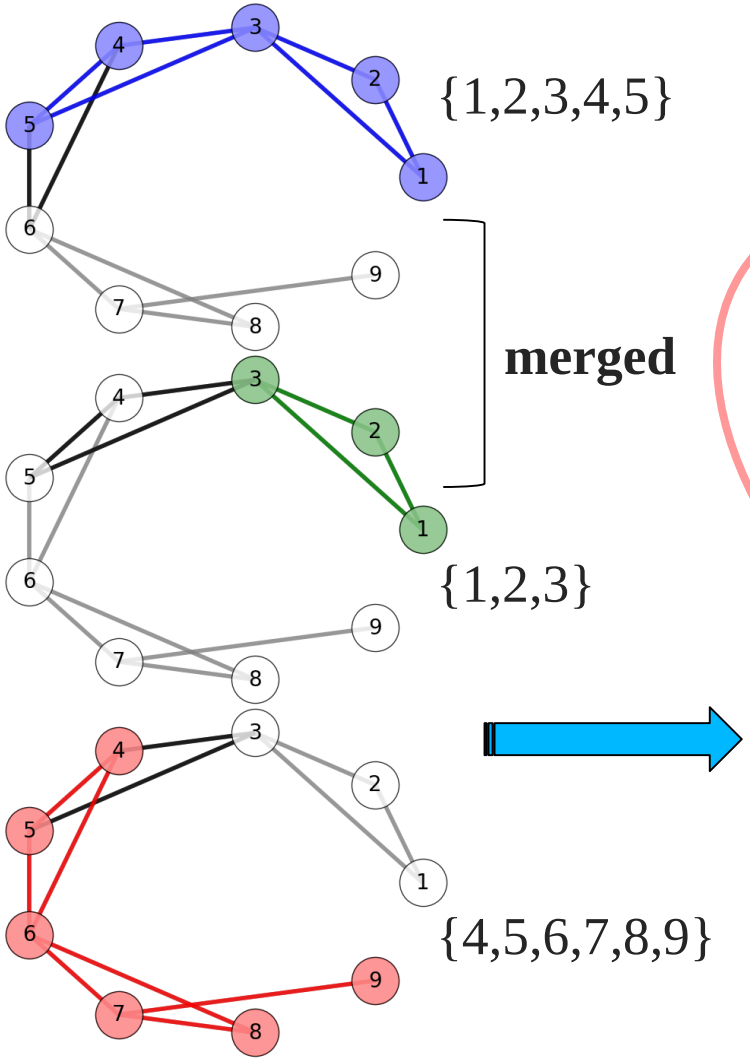


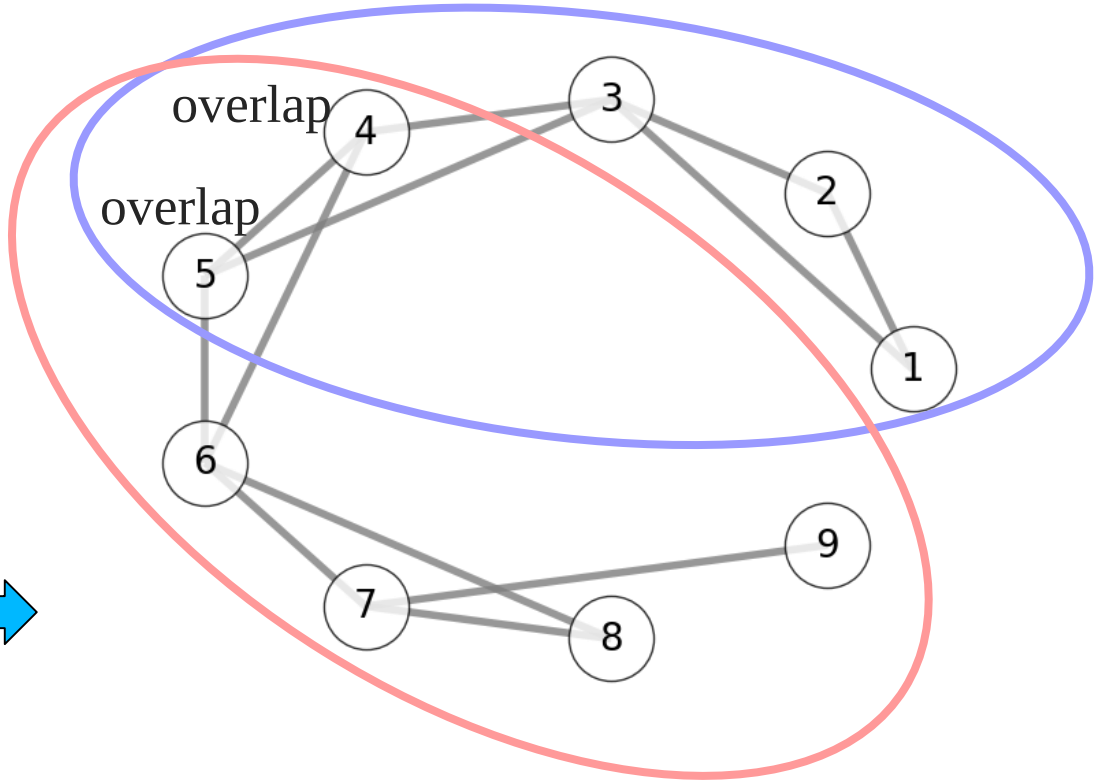
Intermediate Variables

- members: {4,5,6,7,8,9}
- neighbors: {3}
- w_{in}: 7
- w_{out}: 2
- $w_{in} + w_{out}$: 9
- fitness: 7/9

Category3: Local Expansion

- **Step3: Merge Intermediate Communities**





overlapping communities

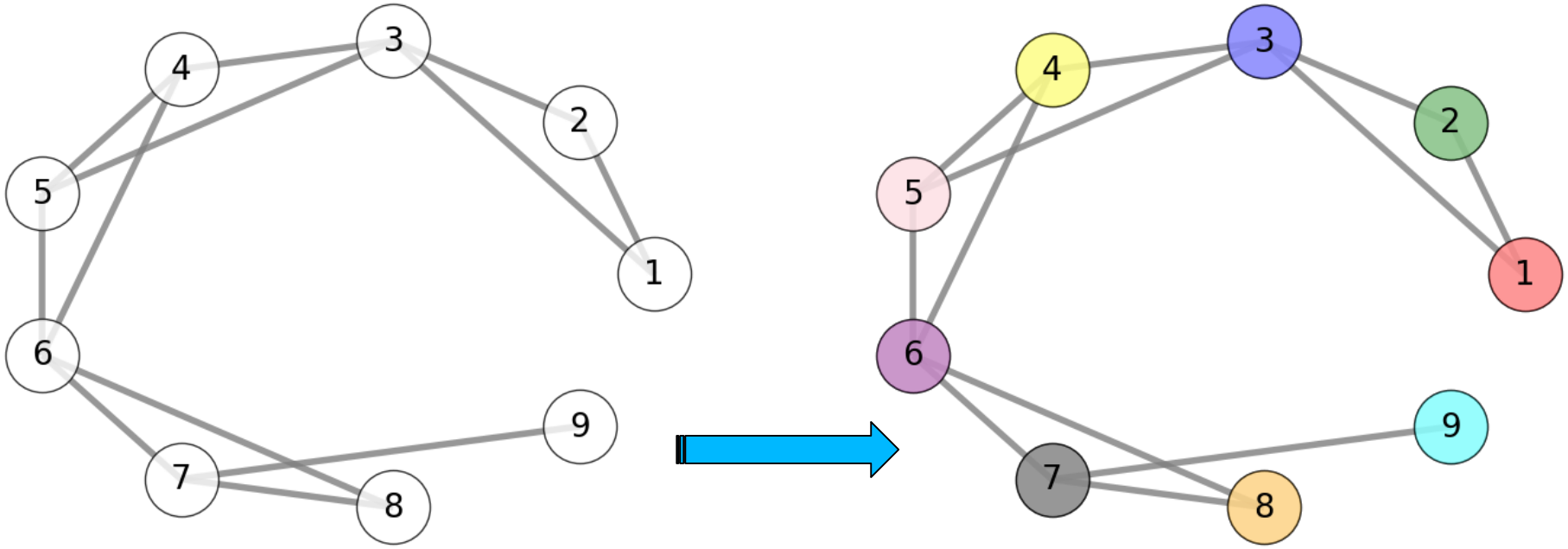
Category4: Dynamics

- **Label Propagation Algorithm**
  - **One Round Computation**
    - outer iterations over vertices
    - inner iterations over neighbors' labels
  - **Label Update Strategy**
    - vote and choose the majority, randomly break tie
  - **Label Update Mode**
    - asynchronous: update after finishing inner iterations
    - synchronous: update after finishing outer iterations

- **Extensions**
  - Apply on a subgraph
  - Allow a vertex to have multiple labels with belonging factors(belonging factor: correlation between a vertex and a list of communities)

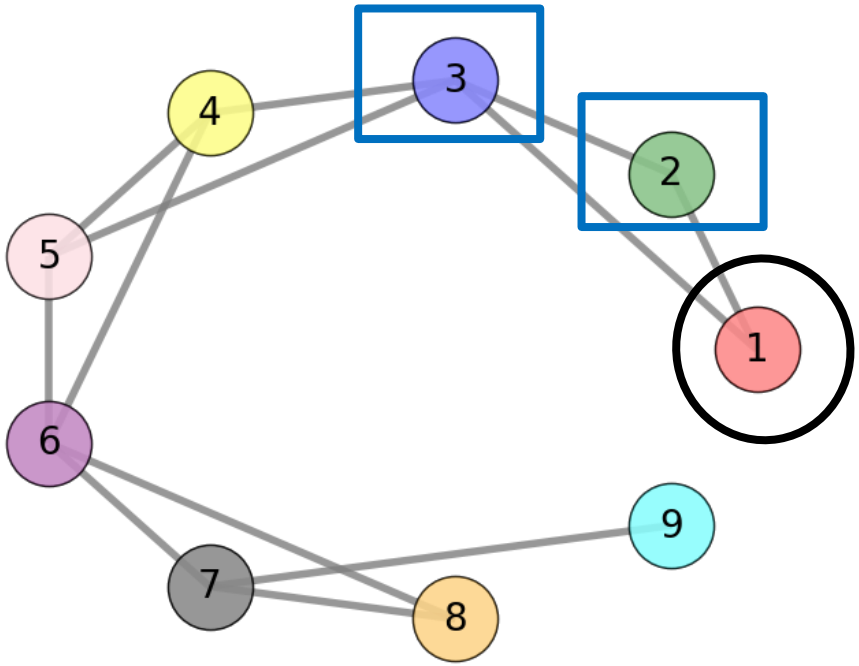
Category4: Dynamics

- **Step1: Assign Labels**
  - assign each vertex a unique label(color)



Category4: Dynamics

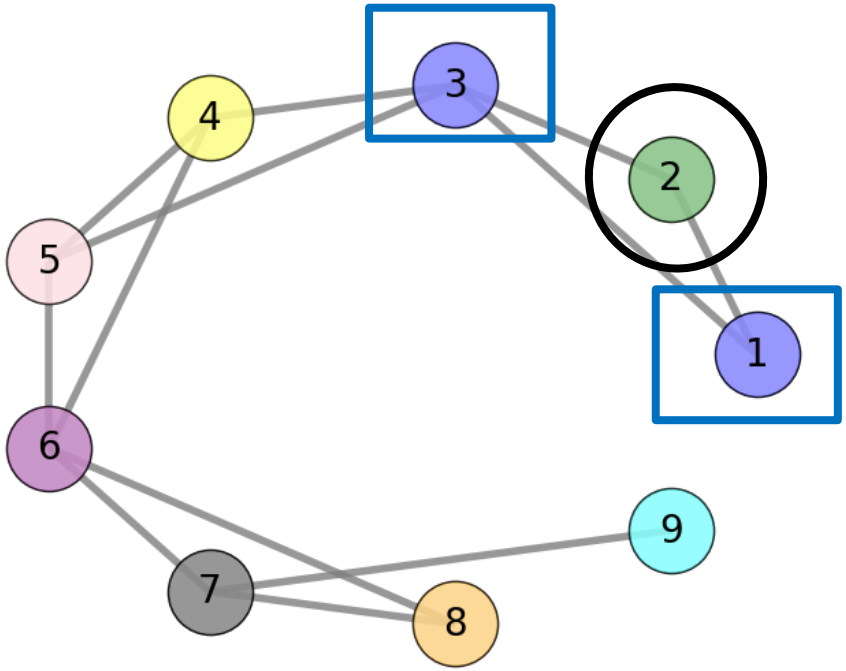
- **Step2: Asynchronous Mode**
  - status after initializing labels



- vertex 1 is receiving labels from neighbors
- labels from vertices 2 and 3 are a tie
- randomly break tie

Category4: Dynamics

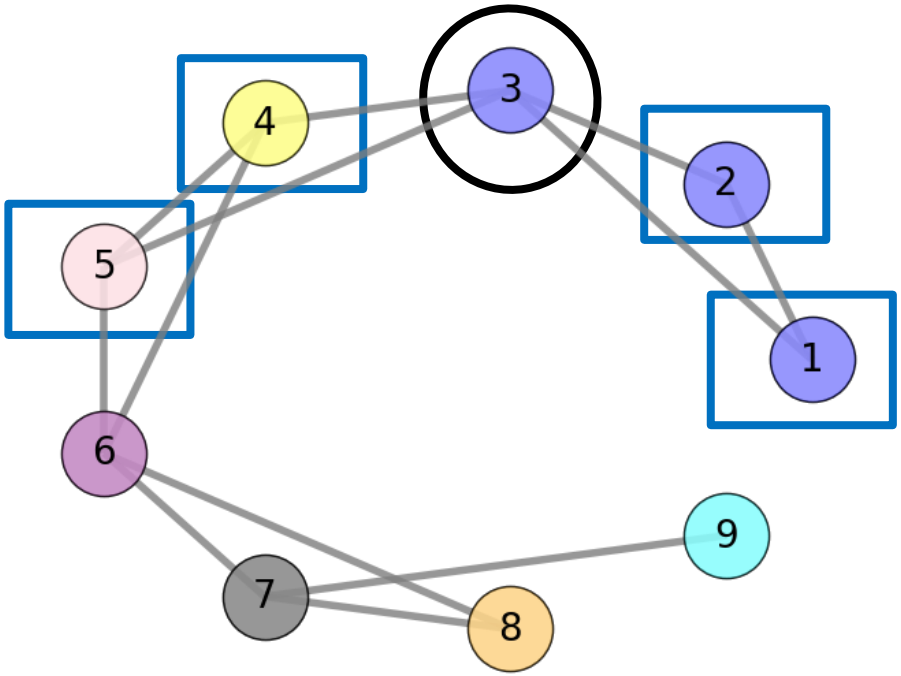
- **Asynchronous Mode**
  - status after first inner iteration



- vertex 2 is receiving labels from neighbors
- vertices 1 and 3 are in the same color
- only choice is blue

Category4: Dynamics

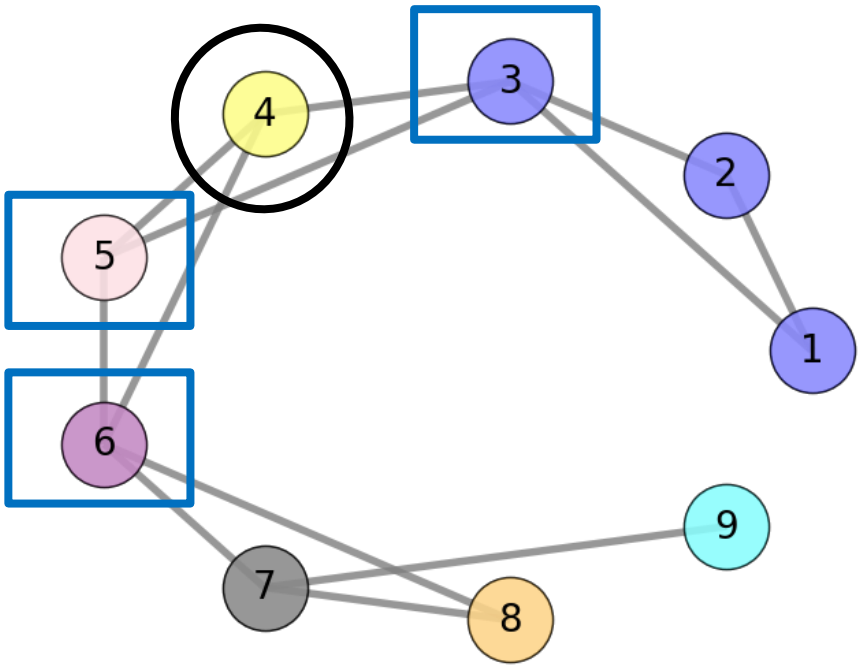
- **Asynchronous Mode**
  - status after second inner iteration



- vertex 3 is receiving labels from neighbors
- neighbors are in three colors
- majority is blue

Category4: Dynamics

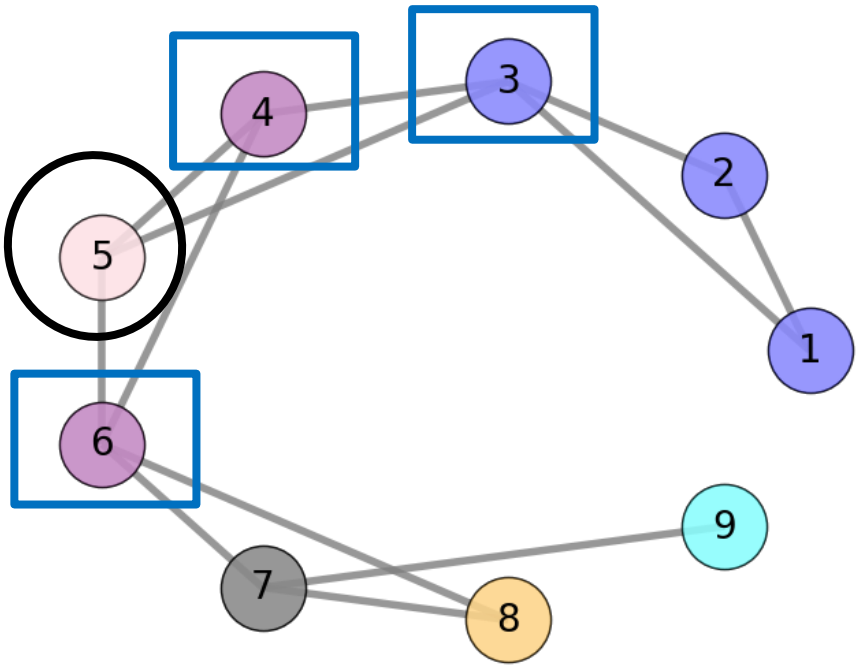
- **Asynchronous Mode**
  - status after third inner iteration



- vertex 4 is receiving labels from neighbors
- neighbors are in three colors and in tie
- randomly break tie

Category4: Dynamics

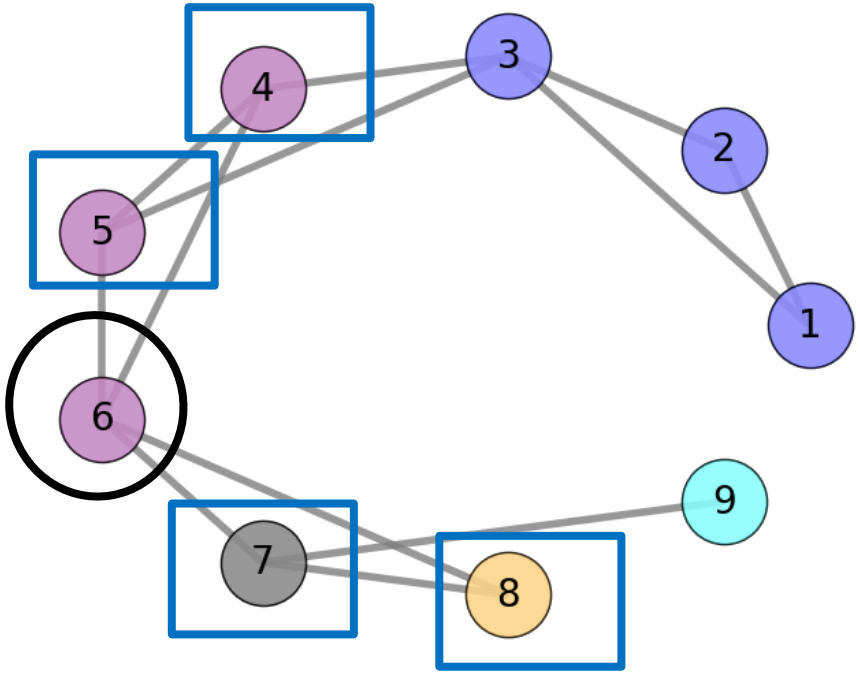
- **Asynchronous Mode**
  - status after fourth inner iteration



- vertex 5 is receiving labels from neighbors
- two colors of neighbors
- majority is purple

Category4: Dynamics

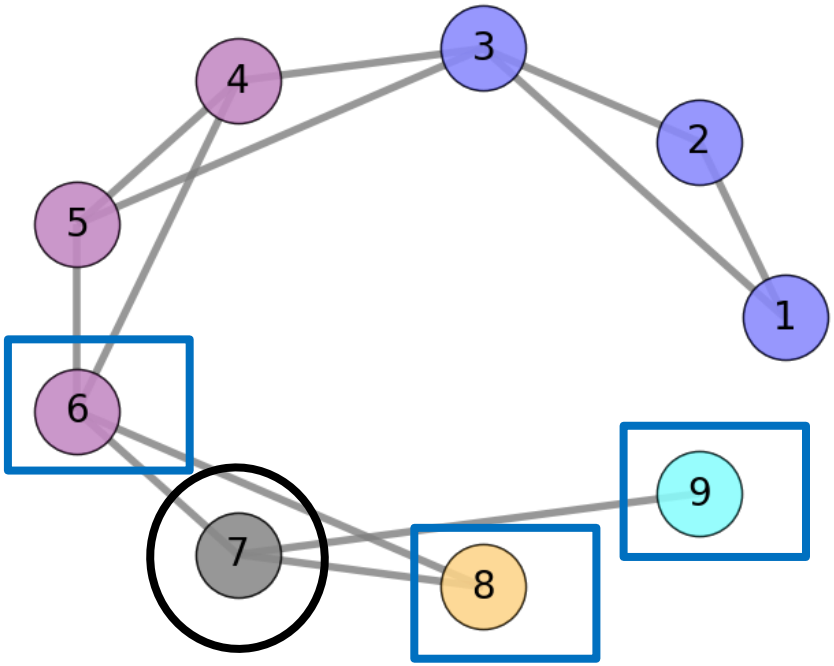
- **Asynchronous Mode**
  - status after fifth inner iteration



- vertex 6 is receiving labels from neighbors
- three colors of neighbors
- majority is purple

Category4: Dynamics

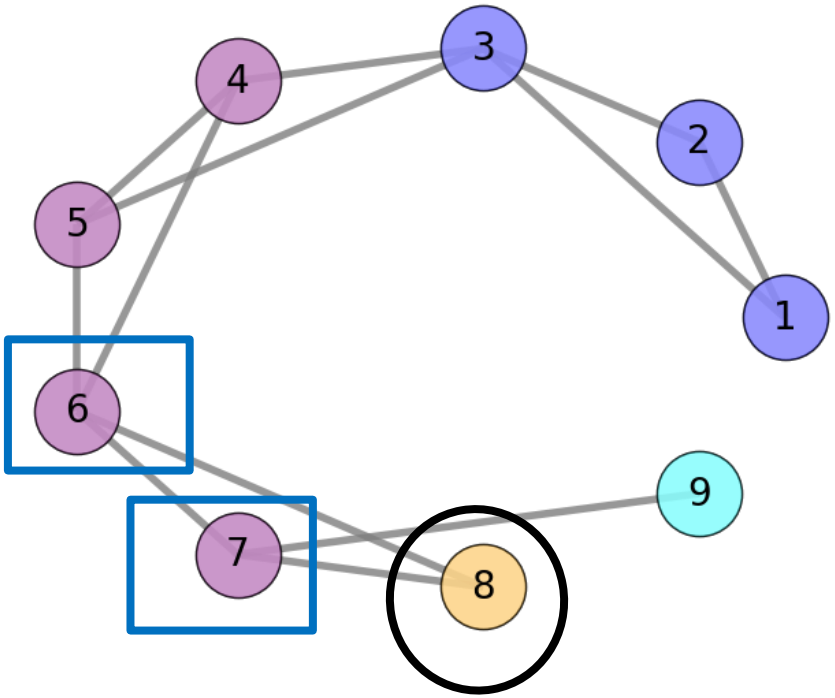
- **Asynchronous Mode**
  - status after sixth inner iteration



- vertex 7 is receiving labels from neighbors
- three colors of neighbors and in tie
- randomly break tie

Category4: Dynamics

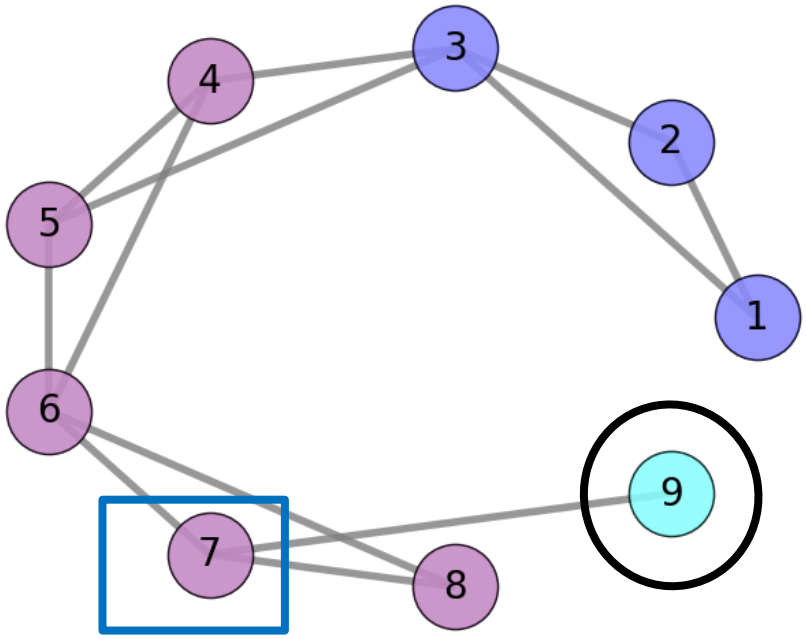
- **Asynchronous Mode**
  - status after seventh inner iteration



- vertex 8 is receiving labels from neighbors
- both neighbors in the same color
- only choice is purple

Category4: Dynamics

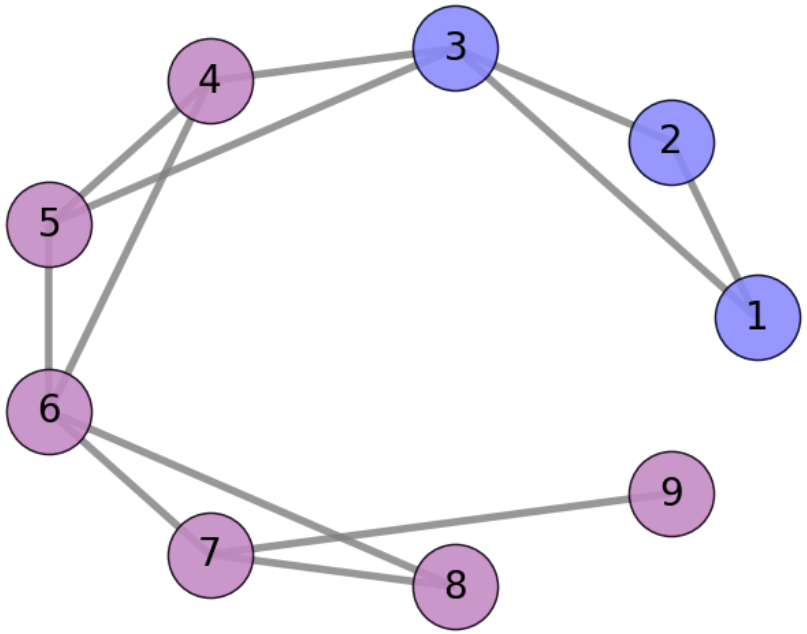
- **Asynchronous Mode**
  - status after eighth inner iteration



- vertex 9 is receiving labels from neighbors
- only one neighbor
- only choice is purple

Category4: Dynamics

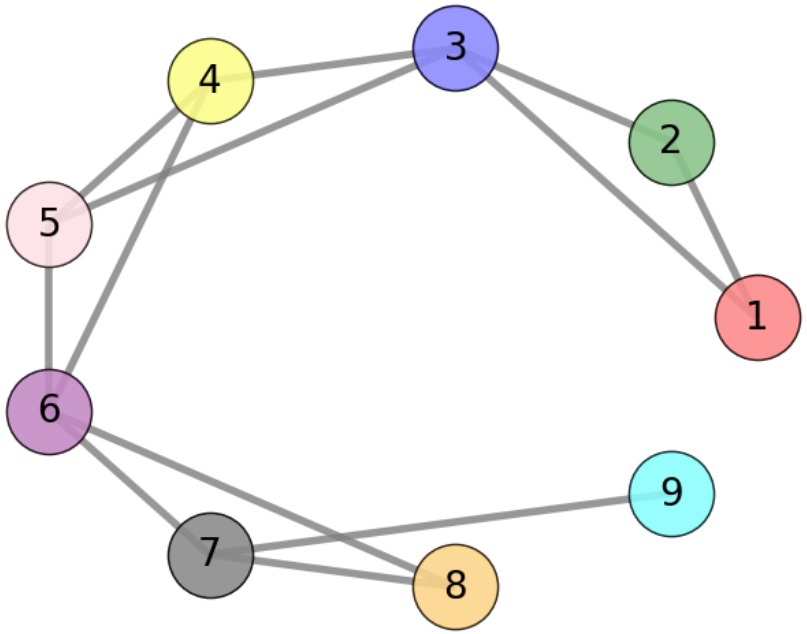
- **Asynchronous Mode**
  - status after ninth inner iteration



- reach the end of first outer iteration
- we can further do more outer iterations in the same manner

Category4: Dynamics

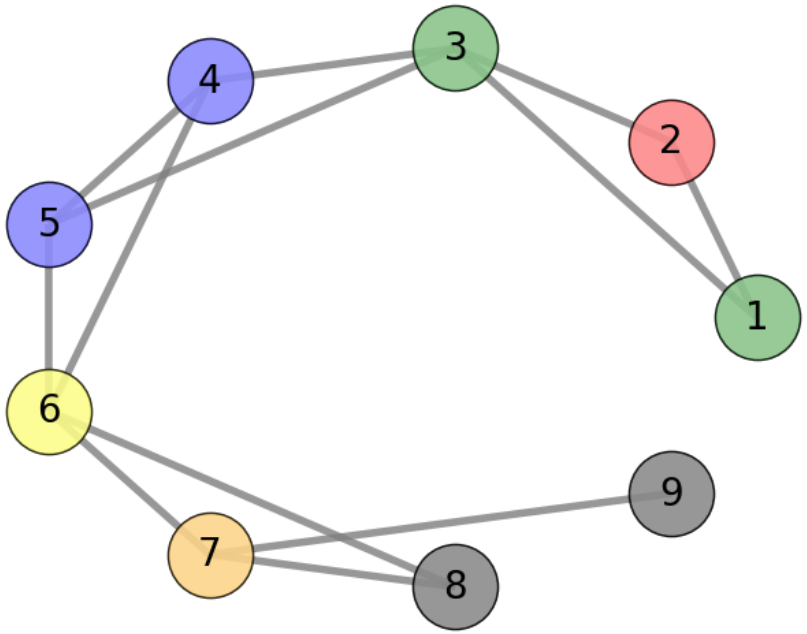
- **Step2: Synchronous Mode**
  - intermediate labels during the inner iterations are stored
  - labels get updated after one outer iteration



outer iteration 1

Category4: Dynamics

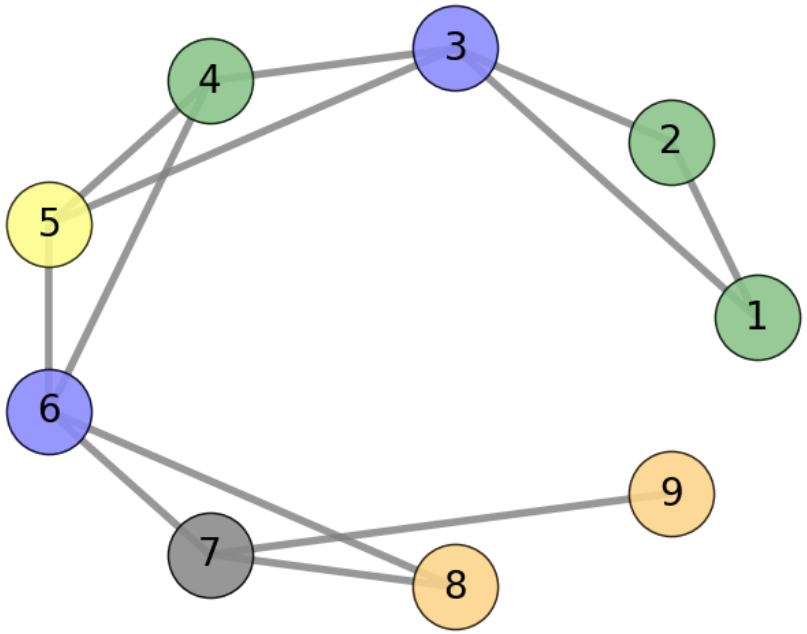
- **Step2: Synchronous Mode**
  - intermediate labels during the inner iterations are stored
  - labels get updated after one outer iteration
  - changes are more stable than those in asynchronous mode



outer iteration 2

Category4: Dynamics

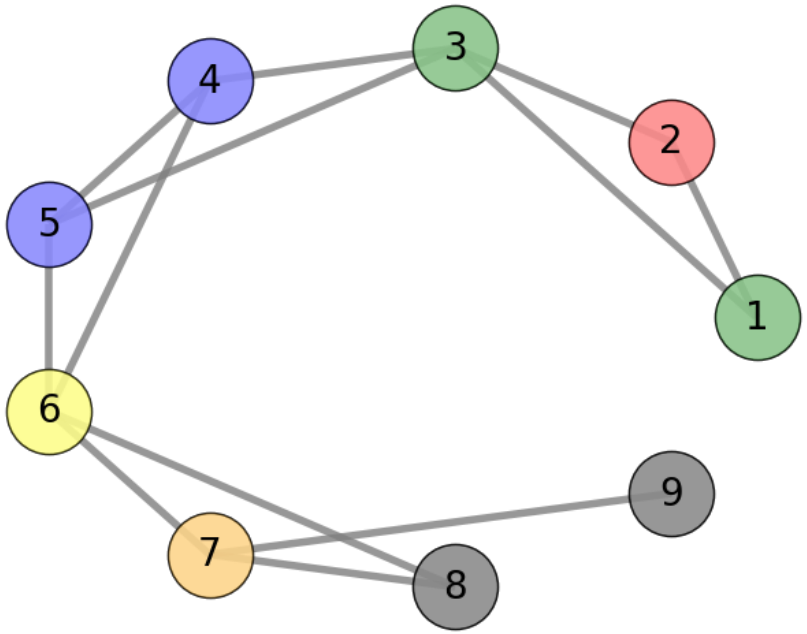
- **Synchronous Mode**
  - intermediate labels during the inner iterations are stored
  - labels get updated after one outer iteration
  - changes are more stable than those in asynchronous mode



outer iteration 3

Category4: Dynamics

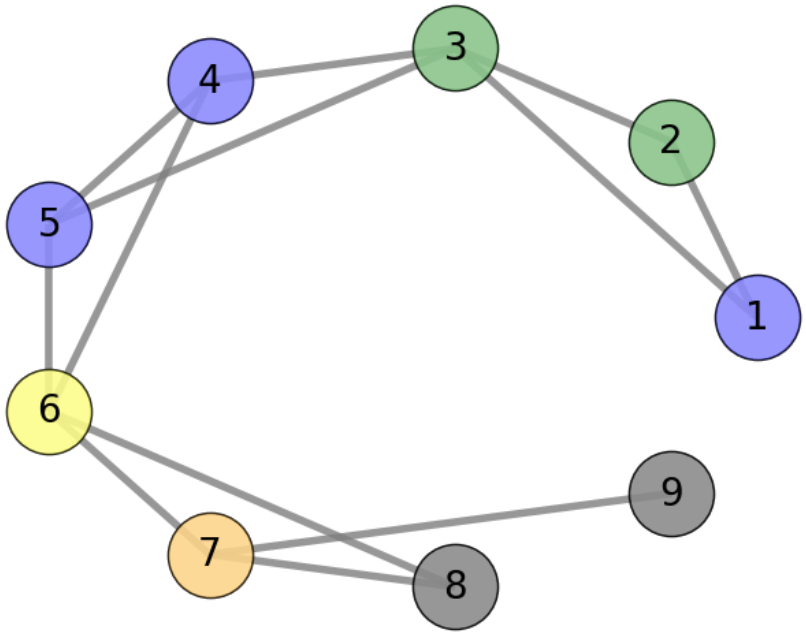
- **Synchronous Mode**
  - intermediate labels during the inner iterations are stored
  - labels get updated after one outer iteration
  - changes are more stable than those in asynchronous mode



outer iteration 4

Category4: Dynamics

- **Synchronous Mode**
  - intermediate labels during the inner iterations are stored
  - labels get updated after one outer iteration
  - changes are more stable than those in asynchronous mode



do more iterations

Algorithm Summary – Pros & Cons

Algorithms	Pros	Cons
Clique Percolation	• have many alternatives to replace the structure of clique, e.g., k-cores, k-truss	• look like pattern matching methods • high complexity
Link Partition	• make it possible to use previous available disjoint community detection algorithms to detect overlapping communities	• weak interpretations of the realistic meanings behind it
Local Expansion	• various expansion strategies, e.g., personalized page-rank, different fitness improvement with local-search strategy	• each implementation differs a lot

Algorithm Summary – Pros & Cons

Algorithms	Pros	Cons
Label Propagation	• quite fast, linear complexity in each outer iteration	• not stable, with randomness in the algorithm • hard to tell the best number of iterations
Statistical Inference	• have a model to illustrate the relationship between the community and graph topology	• require expert knowledge of the graph topology and statistical analysis

Content

1、Background

2、Algorithms

3、Evaluations Metrics

4、Summary & Discussion

Evaluation Metrics Overview

- **With Ground Truth**

	Precision	**Information Gain(X|Y, Y|X)**
Disjoint	• rand index(1973) • adjusted rand index(1985)	• normalized mutual information (2002)
Overlapping	• omega index(1990)	• overlap normalized mutual information (2009)

- **Without Ground Truth**

	Modular Structure
Disjoint	• modularity(2004)
Overlapping	• link-belonging modularity(2009)

Evaluation Metrics Overview

- **With Ground Truth**

	Precision	Information Gain(X|Y, Y|X)
Disjoint	• rand index(1973) • adjusted rand index(1985)	• normalized mutual information (2002)
Overlapping	• omega index(1990)	• overlap normalized mutual information (2009)

- **Without Ground Truth**

	Modular Structure
Disjoint	• modularity(2004)
Overlapping	• link-belonging modularity(2009)

With Ground Truth: Rand Index

- **Two Partitions**
  - two lists of disjoint communities for {1,2,3,4,5,6}
    - ground truth: {1,2,3}, {4,5},{6}
    - result: {1,2,5}, {3,4,6}
- **Pairs**
  - select 2 elements from 6 elements: 15 possible combinations
- **Matched Pairs**
  - type1: {element0, element1} is a subset of a community in parition1 and a subset of a community in parition2
  - type2: {element0, element1} is not a subset of any community in partition1, and not a subset of any community in parition2

Rand Index Example

- **Two Partitions**
  - two lists of disjoint communities for {1,2,3,4,5,6}
    - ground truth, parition1: {1,2,3}, {4,5}, {6}
    - result, partition2: {1,2,5}, {3,4,6}

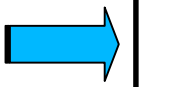
- **Matched Pairs**
  - type1 examples: {1,2}, since a subset of {1,2,3} in parition1, {1,2,5} in parition2
  - type2 examples: {3,5}, since no community in parition1 or parition2 containing it
  - mismatch examples: {3,6} not a subset of any community in parition1, but a subset of {3,4,6} in partition 2

- **Rand Index**
  - matched number / total combination number

Rand Index Example

type 1 match number

p2\p1	{1,2,3}	{4,5}	{6}
{1,2,5}	1	0	0
{3,4,6}	0	0	0



in p1 not in p2 pair number

p2\p1	{1,2,3}	{4,5}	{6}
{1,2,5}, {3,4,6}	3-1-0=2	1-0-0=1	0-0-0=0

in p2 not in p1 pair number

p2\p1	{1,2,3}, {4,5}, {6}
{1,2,5}	3-1-0-0=2
{3,4,6}	3-0-0-0=3

- **Rand Index**
  - matched number / total combination number
  - matched number =15 - (2+1+0) - (2+3) = 7
  - total combination number = 15
  - value of rand index = 7 /15

Adjusted Rand Index

- **Two Partitions**
  - two lists of disjoint communities for {1,2,3,4,5,6}
    - ground truth, partition1: {1,2,3}, {4,5}, {6}
    - result, partition2: {1,2,5}, {3,4,6}

- **Why adjusted this value?**
  - elements in two partitions randomly permuted also yield a positive value, while keeping the size of each community stable

- **Adjusted Rand Index**
  - difference : rand index - expected rand index
  - max difference : max index – expected rand index
  - adjusted rand index : difference / (max difference)

Extension to Omega Index

- **Extended From Adjusted Rand Index**

- **Two Overlapping Community Lists**
  - two lists of overlapping communities for {1,2,3,4,5,6}
    - ground truth: {1,**2**,**3**}, {**2**,**3**,4,5}, {6}
    - result: {1,**2**,**3**,**4**,5}, {**2**,**3**,**4**,6}

- **Matched Pairs Notation Change**
  - previous type1: {element0, element1} is a subset of a community in ground truth and a community in result
  - overlapping version type1:
    - {element0, element1} is a subset of k communities in ground truth
    - {element0, element1} is a subset of k communities in result

Omega Index Example

- **Two Overlapping Community Lists**
  - two lists of overlapping communities for {1,2,3,4,5,6}
    - ground truth: {1,**2**,**3**}, {**2**,**3**,4,5}, {6}
    - result: {1,**2**,**3**,**4**,5}, {**2**,**3**,**4**,6}
- **Matched Pairs**
  - overlapping version type1:
    - {element0, element1} is a subset of k communities in ground truth
    - {element0, element1} is a subset of k communities in result
- **Examples**
  - **counted pair** {2,3}: appears twice both in ground truth and result
  - **not counted pair** {3,4}: appears once in ground truth but twice in result

Evaluation Metrics Overview

- **With Ground Truth**

	Precision	**Information Gain(X|Y, Y|X)**
Disjoint	• rand index(1973) • adjusted rand index(1985)	• normalized mutual information (2002)
Overlapping	• omega index(1990)	• overlap normalized mutual information (2009)

- **Without Ground Truth**

	Modular Structure
Disjoint	• modularity(2004)
Overlapping	• link-belonging modularity(2009)

Without Ground Truth: Modularity

- **Random Graph(Null Model)**
  - fixed degrees, given a list of vertex degrees
  - no prior knowledge about how vertices linked
  - probability to link vertex i and vertex j, where k_i represent the degree of vertex i

$$P_{ij} = \frac{k_i k_j}{4m^2}, \qquad \forall i, j$$

- **Sub Graphs**
  - type1 sub graph: extracted from a community of vertices in the original graph
  - type 2 sub graph: extracted from a community of vertices in the random graph

- **Assumption**
  - type 1 sub graphs are supposed to be denser than type 2

Modularity

- **Sub Graphs**
  - type1 sub graph: extracted from a community of vertices in the original graph
  - type 2 sub graph: extracted from a community of vertices in the random graph
- **Assumption**
  - type 1 sub graphs are supposed to be denser than type 2

- **Local Modular Score**
  - observed number of edges – expected number of edges
  - with normalization

$$Q_S = \sum_{i,j \in V'} \left[\frac{A_{ij}}{2m} - P_{ij} \right] \qquad A_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are connected} \\ 0 & \text{otherwise.} \end{cases}$$

Modularity

- **Sub Graphs**
  - type1 sub graph: extracted from a community of vertices in the original graph
  - type 2 sub graph: extracted from a community of vertices in the random graph
- **Assumption**
  - type 1 sub graphs are supposed to be denser than type 2

- **Global Modular Score**
  - sum of all local modular scores
  - with normalization

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

$$\delta(c_i, c_j) = \begin{cases} 1 & \text{if } i \text{ and } j \text{ belong to the same community} \\ 0 & \text{otherwise.} \end{cases}$$

Extension to Link Belonging Modularity

- Designed for **directed graph** with **overlapping** communities
- **Change: Random Graph Parameter**

$$P_{ij} = \frac{k_i k_j}{4m^2}, \quad \forall i, j \quad \Rightarrow \quad P_{ij} = \frac{k_i^{\text{out}} k_j^{\text{in}}}{m^2}$$

- **Addition: Vertex Belonging Factors**
  - vertex i belongs to multiple communities

$$0 \leq \alpha_{i,c} \leq 1 \qquad \forall i \in V, \quad \forall c \in C$$

$$\sum_{c=1}^{|C|} \alpha_{i,c} = 1$$

- **Addition: Belonging Coefficient**
  - relationship between (vertex I, vertex j) and community c

$$\beta_{l(i,j),c} = \beta_{l,c} = \mathcal{F}(\alpha_{i,c}, \alpha_{j,c}) \quad l = (i, j)$$

Extension to Link Belonging Modularity

- **Addition: probability an edge relates to a community**
  - probability (vertex i, vertex j) , destination vertex j in community c

$$\beta^{\text{out}}_{l(i,j),c} = \frac{\sum_{j \in V} \mathcal{F}(\alpha_{i,c}, \alpha_{j,c})}{|V|}$$

  - probability (vertex i, vertex j) , source vertex i in community c

$$\beta^{\text{in}}_{l(i,j),c} = \frac{\sum_{i \in V} \mathcal{F}(\alpha_{i,c}, \alpha_{j,c})}{|V|}$$

- **Change: Global Modular Score Function**

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

$$Q_{\text{ov}} = \frac{1}{m} \sum_{c \in C} \sum_{i,j \in V} \left[\beta_{l(i,j),c} A_{ij} - \frac{\beta^{\text{out}}_{l(i,j),c} k^{\text{out}}_i \beta^{\text{in}}_{l(i,j),c} k^{\text{in}}_j}{m} \right]$$

Content

1、Background

2、Algorithms

3、Evaluations Metrics

4、Summary & Discussion

Algorithm Application Issues

- **Too Many Algorithms**
  - different implementations, languages, platforms
  - complexity, efficiency differs a lot

- **Non-Determinism**
  - randomness in some algorithms, e.g, randomly break tie, randomly select seeds
  - different ways of iterations change the result a lot

- **Evaluation Standard**
  - in most cases, ground truth is not available for a large scale graph
  - few empirical study on large datasets to interpret the quality of overlapping community detection results

Algorithm Efficiency Issues

- **Memory Cost**
  - In some papers, **100MB for a 1 million edges** sparse graph
  - (number of bytes for each edge) * 1MB is required for basic graph storage
  - build hash tables for expected constant time checking the adjacency
  - not scale to billions of edges, which at least requires 100GB

- **Time Cost**
  - In some sequential local expansion algorithms I have implemented and optimized, it takes **up to an hour** to do the computation on a **one-million-edge** graph

Future Work Directions

- **Toolkit**
  - integrate evaluation metrics, datasets and representative algorithms
  - efficiency statistics, summarize some optimization rules
  - visualize some representative algorithms, illustrating internal states and efficiency bottleneck
- **Algorithm**
  - possible improvement, new design and parallelization

End

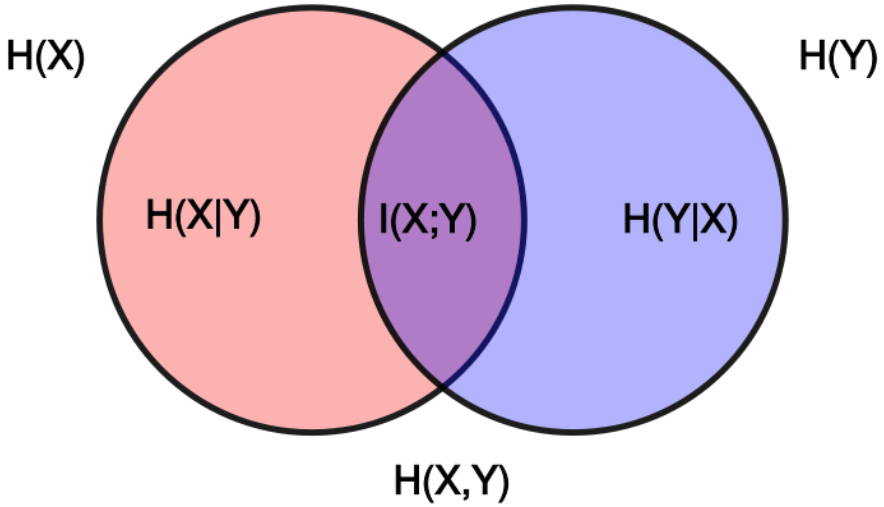
For Detailed Information(Codes/Figures):
https://github.com/GraphProcessor/CommunityDetectionCodes

This PPT:
https://www.dropbox.com/s/on3id2rg6lc691b/yche-20292673-pqe-ppt-complete.ppt?dl=0

With Ground Truth : NMI

- **Mutual Information Basics**
  - H(X|Y): given Y, information required to infer X, a conditional entropy
  - X, Y are discrete random variables here
- **Derivations**

$$
\begin{aligned}
I(X;Y) &= H(X) - H(X|Y) \\
&= H(Y) - H(Y|X) \\
&= H(X) + H(Y) - H(X,Y) \\
&= H(X,Y) - H(X|Y) - H(Y|X)
\end{aligned}
$$



With Ground Truth : NMI

- **I(X;Y) = H(Y) – H(Y|X) Derivation**

Cover T M, Thomas J A. Elements of information theory[M]. John Wiley & Sons, 2012.

$$I(X;Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

$$= \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)} - \sum_{x,y} p(x,y) \log p(y)$$

$$= \sum_{x,y} p(x)p(y|x) \log p(y|x) - \sum_{x,y} p(x,y) \log p(y)$$

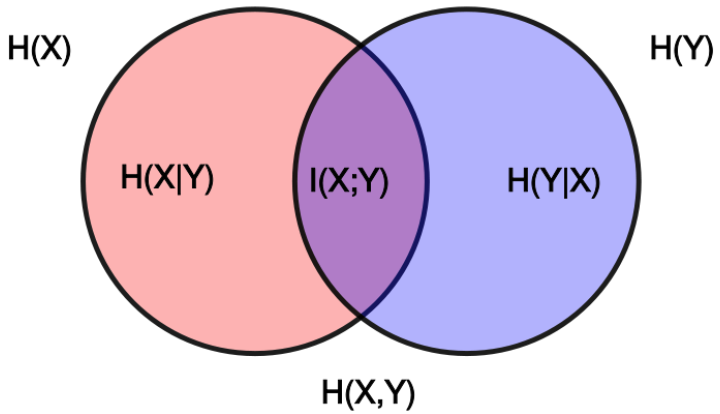
$$= \sum_{x} p(x) \left(\sum_{y} p(y|x) \log p(y|x) \right)$$

$$- \sum_{y} \log p(y) \left(\sum_{x} p(x,y) \right)$$

$$= -\sum_{x} p(x) H(Y|X=x) - \sum_{y} \log p(y)p(y)$$

$$= -H(Y|X) + H(Y)$$

$$= H(Y) - H(Y|X).$$



With Ground Truth : NMI

- **LFR NMI**
  - **V (X, Y) = H(X|Y) + H(Y |X),** variation of information
  - **Normalized Version:** $V'_{norm}(X,Y) = \frac{1}{2}\left(\frac{H(X|Y)}{H(X)} + \frac{H(Y|X)}{H(Y)}\right)$
  - **Random Variables(use community size):**

$$P(X_k = 1) = n_k/N \quad P(X_k = 0) = 1 - n_k/N$$

  - **Joint Distribution:**

$$P(X_k = 1, Y_l = 1) = \frac{|C'_k \cap C'''_l|}{N},$$

$$P(X_k = 1, Y_l = 0) = \frac{|C'_k| - |C'_k \cap C'''_l|}{N},$$

$$P(X_k = 0, Y_l = 1) = \frac{|C'''_l| - |C'_k \cap C'''_l|}{N},$$

$$P(X_k = 0, Y_l = 0) = \frac{N - |C'_k \cup C'''_l|}{N}.$$

Overlap Normalized Mutual Information

- **LFR NMI**
  - **Use Derivation H(X|Y) = H(X,Y) – H(Y)**

$$H(X_k|Y_l) = H(X_k, Y_l) - H(Y_l)$$

for overlapping NMI

$$H(X_k|\mathbf{Y}) = \min_{l \in \{1,2 \ldots |\mathcal{C}''|\}} H(X_k|Y_l)$$

normalization

$$H(X_k|\mathbf{Y})_{norm} = \frac{H(X_k|\mathbf{Y})}{H(X_k)}$$

aggregation

$$H(\mathbf{X}|\mathbf{Y})_{norm} = \frac{1}{|\mathcal{C}'|} \sum_k \frac{H(X_k|\mathbf{Y})}{H(X_k)}$$

final average

$$N(\mathbf{X}|\mathbf{Y}) = 1 - \frac{1}{2}[H(\mathbf{X}|\mathbf{Y})_{norm} + H(\mathbf{Y}|\mathbf{X})_{norm}].$$

Github Repository

My Survey Github Repository:
https://github.com/GraphProcessor/CommunityDetectionCodes

CheYulin add graph layout figures		Latest commit 5582e8c 2 hours ago
Algorithms	refactor some, remains a lot to do	16 days ago
Benchmark/2009-LFR-Benchmark	update lfr benchmark refactored codes description	a month ago
Datasets	update some readmes	19 days ago
Metrics	update some readmes	19 days ago
NonOverlappingCodes	update some readmes	19 days ago
Prensentation	add graph layout figures	2 hours ago
Scripts	aggregation dataset info	3 months ago
SubModules	add submodules	18 days ago
Survey	update readme	17 days ago
.gitignore	add oslom readme pdf file	a month ago
.gitmodules	add submodules	18 days ago
License.md	Create License.md	3 months ago
README.md	update readme	5 days ago

Github Repository

Graph Benchmarks

Synthetic Tool

Content	Detail	Status
2009-LFR-Benchmark	LFR Benchmark to generate five types of graphs	build success, some files unused

Real-World DataSets(Edge List)

Detailed information is in Datasets.

Quality-Evaluation Metrics

Without-Ground-Truth

Evaluation Metric Name	Implementation	Heuristic
Link-Belonging-Based Modularity	link_belong_modularity.py	compare to random graph

With-Ground-Truth

Evaluation Metric Name	Implementation	Heuristic
Overlap-NMI	overlap_nmi.py	info-theory entropy-measure
Omega-Idx	omega_idx.py	unadjusted compared to expected

Github Repository

Algorithms

In each algorithm, there is a `ReadMe.md`, which gives brief introduction of corresponding information of the algorithm and current refactoring status. Category information are extracted, based on Xie's 2013 Survey paper Overlapping Community Detection in Networks: The State-of-the-Art and Comparative Study.

All c++ projects are built with `cmake`, java projects are built with `maven`, python projects are not specified how to build.

Algorithm	Category	Language	Dependency	Status
2008-CPM	clique percolation	c++, python	lcelib	build success
2009-CIS	seed expansion	c++		build success
2009-EAGLE	seed expansion	c++	igraph, boost	build success
2010-LinkComm	link partition	python	optparse	python okay
2010-iLCD	seed expansion	java	args4j, trove4j	build success
2010-CONGA	dynamics	java		build success
2010-TopGC	statistical inference	java		build success
2011-GCE	seed expansion	c++	boost	build success
2011-OSLOM	seed expansion	c++		

Github Repository

2011-MOSES	fuzzy detection	c++	boost	build success
2011-SLPA	dynamics	c++, java, python	networkx, numpy	build success for java
2012-FastCPM	clique percolation	python, c++	networkx	build success
2012-ParCPM	clique percolation	c	igraph	build success
2012-DEMON	seed expansion	python	networkx	python okay
2013-SVINET	statistical inference	c++	gsl, pthread	build success
2013-SeedExpansion	page-rank	c++, matlab	graclus, matlab-bgl	
2014-HRGrow	matrix-exponential	c++, matlab, python	pylibbvg	python okay
2015-LEMON	seed expansion	python	pulp	python okay

Submodules(Dependencies)

Detailed information is in SubModules.

Submodule	Implementation Language	Detail
igraph	c	also provide python wrapper, graph utilities
matlab-bgl	c++, matlab	matlab implementation with bgl dependency
graclus	c++	graph partition algorithm
lcelib	c++	graph utilities by CxAalto