Unsupervised and self-supervised Learning for Graphs

CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Project Proposal

- **No late days** (give yourself a fall break)
- Make sure that you do preliminary analysis on dataset!
  - **Not enough**: Size, number of graphs
  - **Reasonable**: Degree distribution, centrality measures, PageRank …
  - **Great**: Spectral embeddings, degree of homophily (Homework 2, Qn 2), motifs, clustering … **Innovative methods** to analyze the dataset from different perspectives
- Graduate and undergrad course versions have different requirements
- **Suggestion**: mention how you plan to split the work among group members
  - This is necessary if the group consists of students taking undergraduate as well as graduate versions of the course

Comments on Colabs

- **Colab filenames** for submission: colab2_firstname_lastname.ipynb
- **Coding convention**
  - __getitem__, __setitem__ (use slicing [])
    - See many different slicing functionalities
    - We usually inherit __item__ method to empower these slicing functionalities
  - __call__ : directly call the class
  - Call forward function of nn.Module (use model())
  - Similar functionalities: __len__(), __str__(), __contains__()
- **Try to respect the prompt** (e.g. write within the designated blanks)
  - If possible, don't alter the return values, arguments etc. You could simply name your variables with the specified names of the return values / arguments

Unsupervised and Self-supervised Learning for Graphs

- **Network Community Detection**
  - **Network Communities**
  - **Louvain Algorithm**

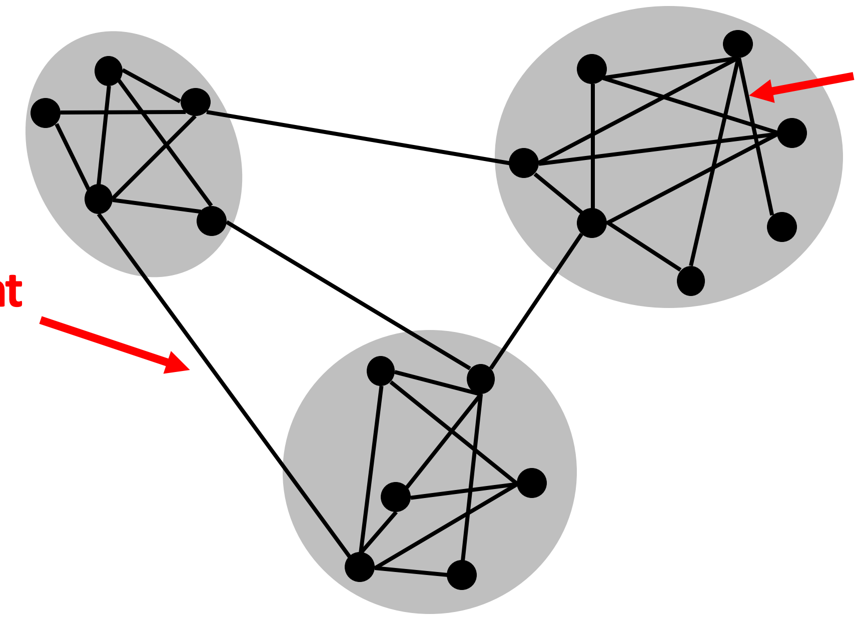
- **Strategies for Pre-training Graph Neural Networks**

Unsupervised and Self-supervised Learning for Graphs

- **Network Community Detection**
  - **Network Communities**
  - **Louvain Algorithm**
- **Strategies for Pre-training Graph Neural Networks**

Networks & Communities

- We often think of networks "looking" like this:



A set of nodes that is densely connected internally

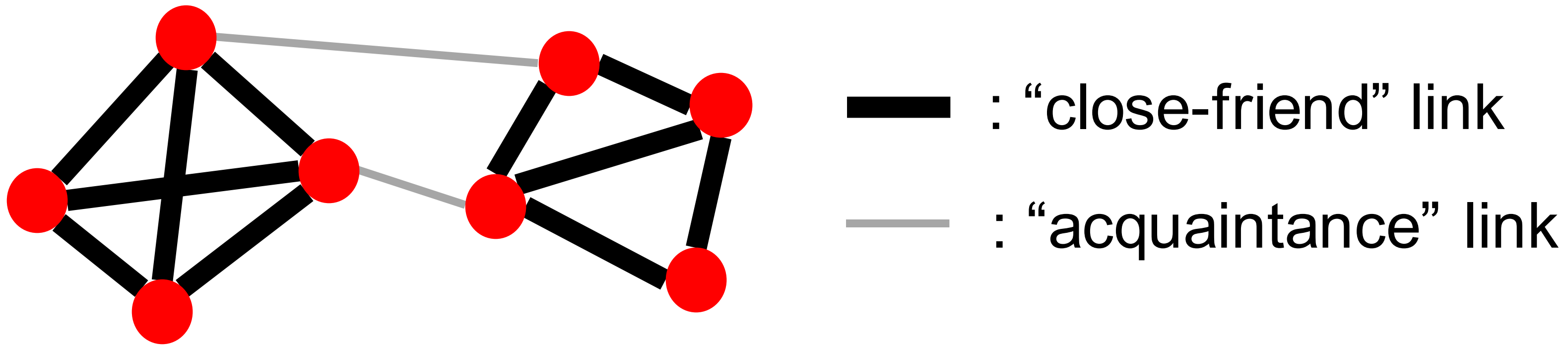
Connections between different groups of nodes are scarce

- What led to such a conceptual picture?

Flow of Job Information

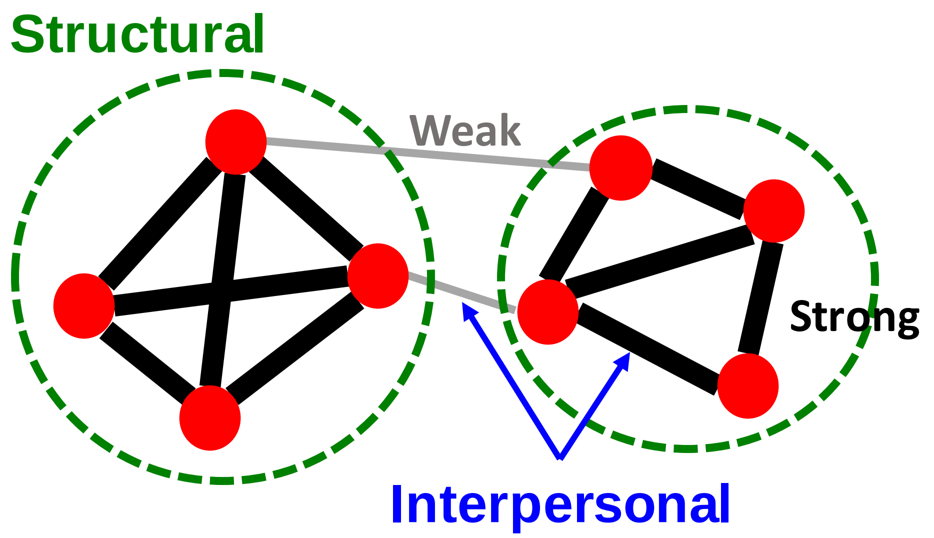
- **How do people find out about new jobs?**
  - This question is studied by Mark Granovetter, part of his PhD in 1960s
  - People find the information **through personal contacts**
- **But:** Contacts are often **acquaintances** rather than close friends
  - **This is surprising:** One would expect your friends to help you out more than casual acquaintances
- **Why is it that acquaintances are most helpful?**

Personal contact network



Granovetter's Answer (1)

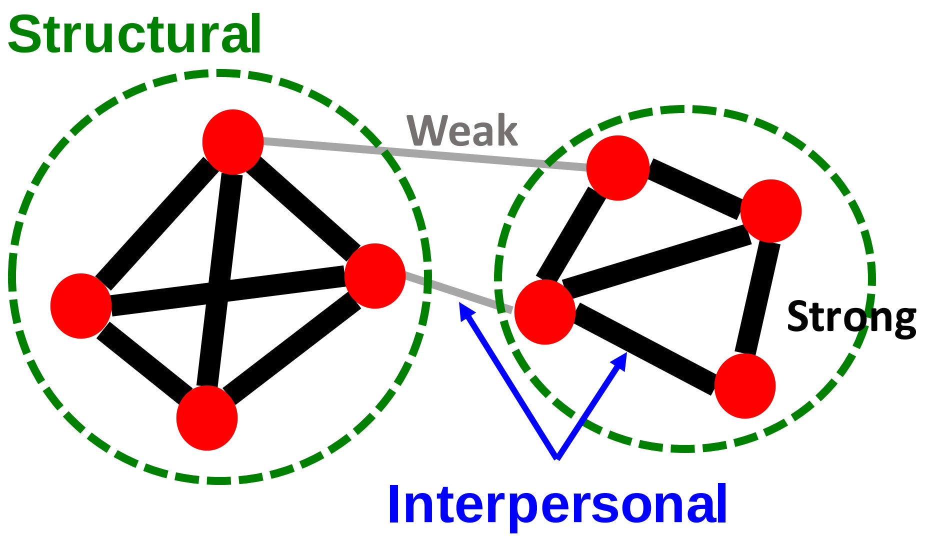
- **Two perspectives on friendships**
  - **Structural:** Friendships span different parts of the network
  - **Interpersonal:** Friendship between two people is either **strong** or **weak**
    - Structurally embedded (tightly-connected) edges are also socially **strong**
    - Long-range edges spanning different parts of the network are socially **weak**



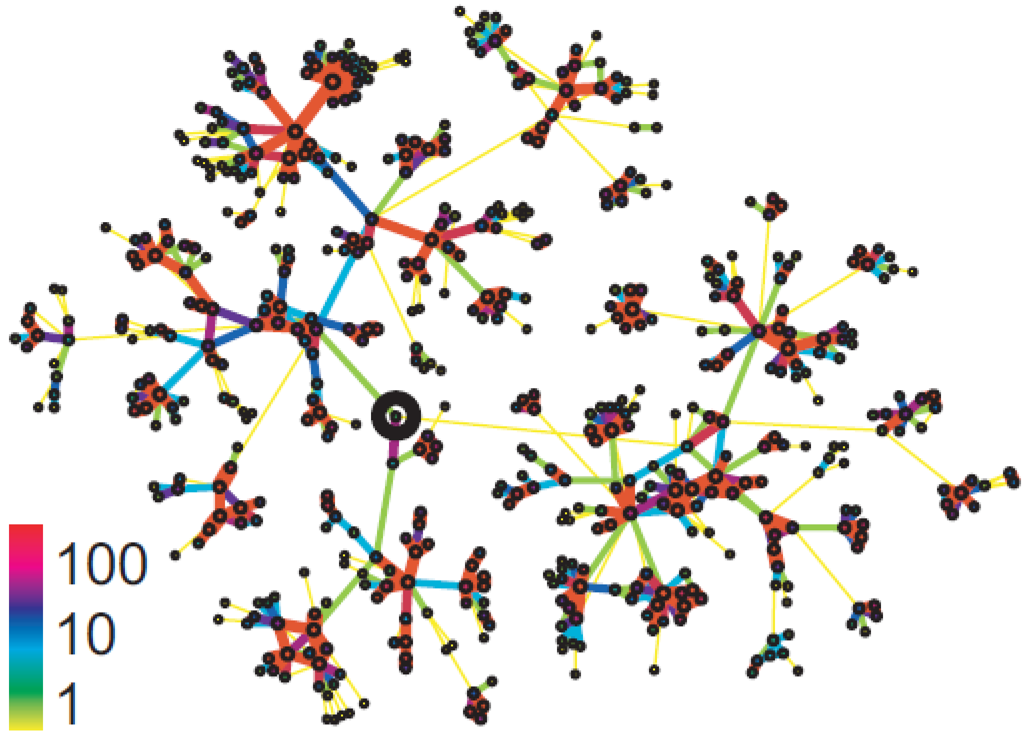
[Granovetter '73]

Granovetter's Answer (2)

- **Why is it that acquaintances are most helpful?**
  - Long-range edges allow you to gather information from **different parts of the network** and get a job
  - Structurally embedded edges are **heavily redundant** in terms of information access



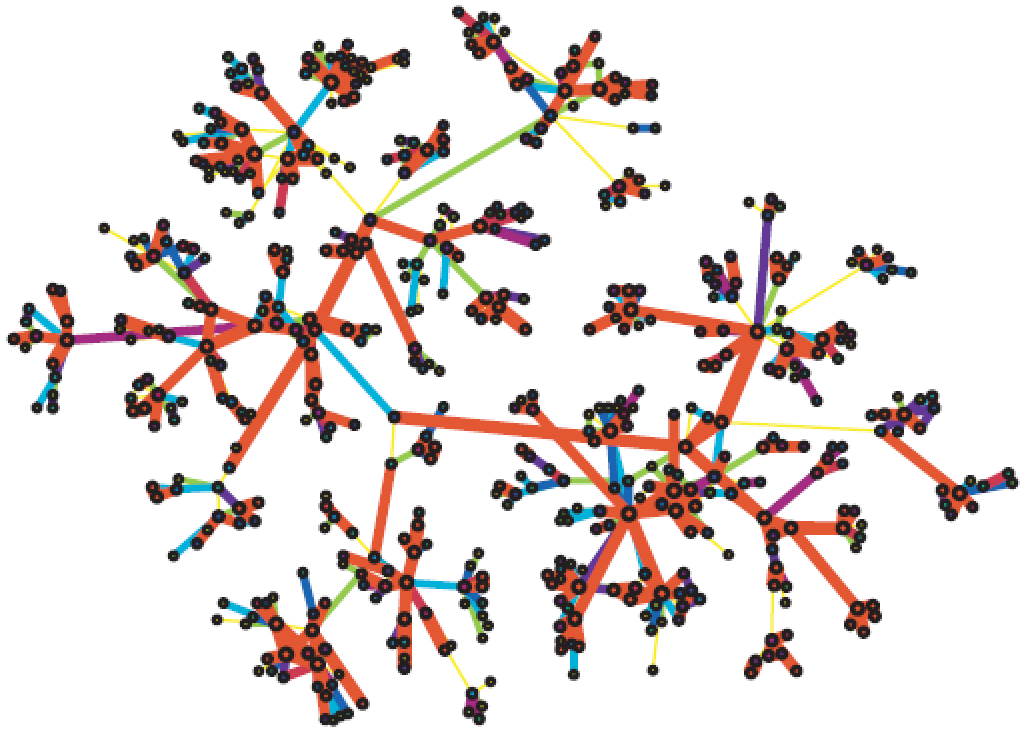
Real Network, Real Edge Strengths



"Structure and tie strengths in mobile communication networks" Onnela et al. (2007)

- **Real edge strengths in mobile call graph**
  - Strong ties are more embedded (have higher overlap)

Real Network, Permuted Tie Strengths

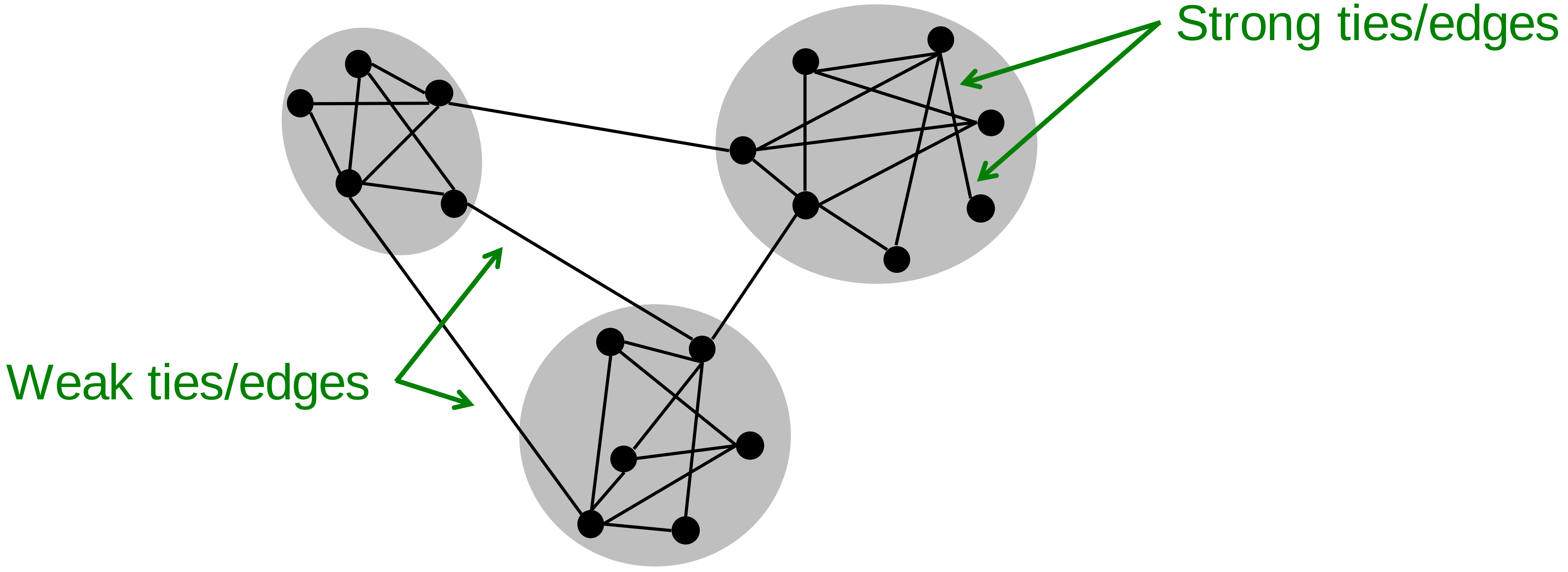


“Structure and tie strengths in mobile communication networks” Onnela et al. (2007)

- **Same network, same set of edge strengths** but now **strengths are randomly shuffled**

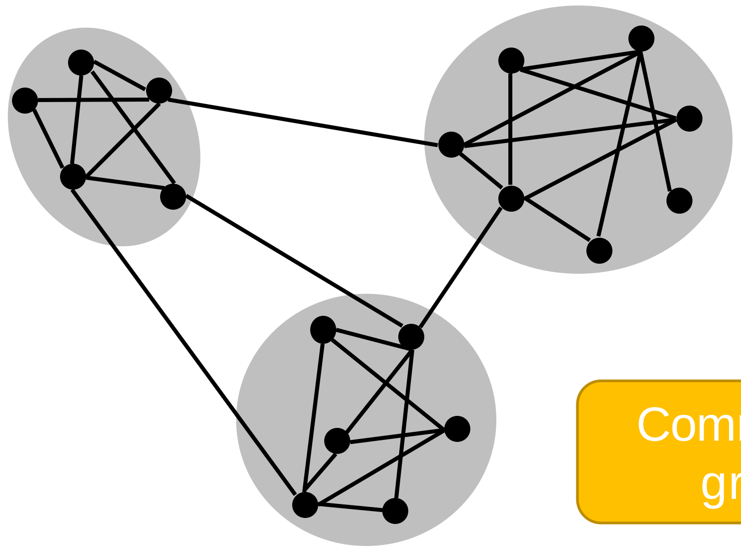
Conceptual Picture of Networks

- **Granovetter's theory leads to the following conceptual picture of networks**



Network Communities

- Granovetter’s theory suggests that networks are composed of **tightly connected sets of nodes**
- **Network communities:**
  - Sets of nodes with **lots** of **internal** connections and **few external** ones (to the rest of the network).

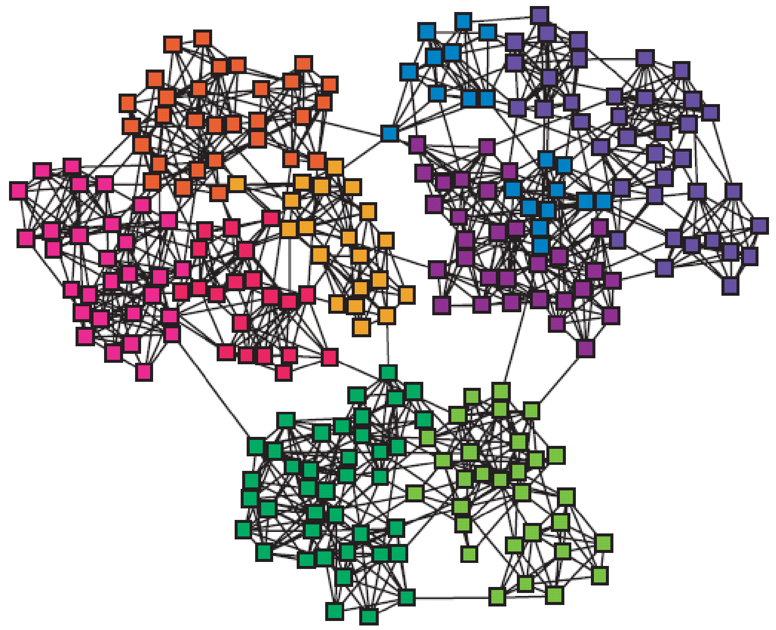


Communities, clusters, groups, modules

How to Find Network Communities

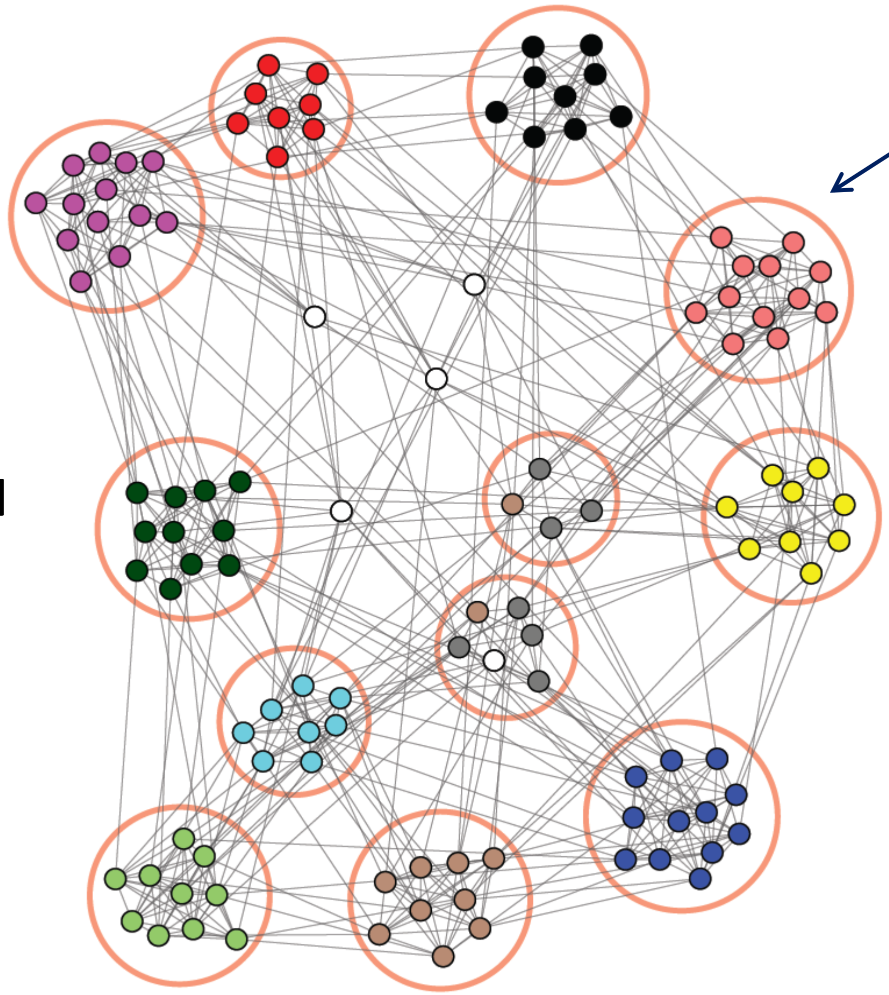
- **How do we automatically find such densely connected groups of nodes?**
- Ideally such automatically detected clusters would then correspond to real groups
- **For example:**

Communities, clusters, groups, modules

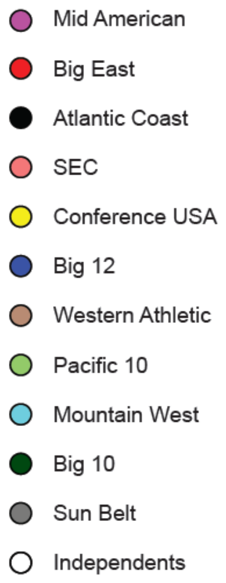


Example: NCAA Football Network

Nodes: Teams
Edges: Games played



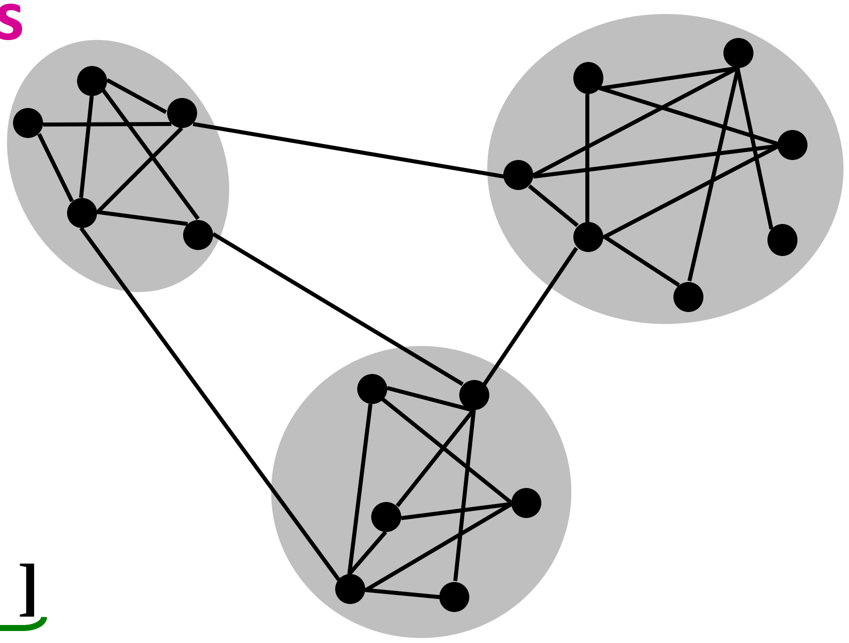
NCAA conferences



Network Communities Measure

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

$$Q \propto \sum_{s \in S} \underbrace{[\ (\#\ \text{edges within group } s) - (\text{expected}\ \#\ \text{edges within group } s)\ ]}_{\text{Need a null model}}$$



Null Model: Configuration Model (1)

- Given real $\boldsymbol{G}$ on n nodes and m edges, how to know the **expected number of edges** between nodes i and j?
  - We can construct a network G' with uniformly random connections that has the **same degree distribution** as $\boldsymbol{G}$
    - Consider $\boldsymbol{G}$' as a multigraph (multiple edges can exist between nodes)
- **The expected number of edges between nodes $\boldsymbol{i}$ and $\boldsymbol{j}$** of degrees $\boldsymbol{k_i}$ and $\boldsymbol{k_j}$ equals: $\boldsymbol{k_i \cdot \frac{k_j}{2m} = \frac{k_i k_j}{2m}}$
  - There are $2m$ **directed** edges (counting i->j and j->i) in total.
  - For each of k_i out-going edges from node i, the chance of it landing to node j is $k_j/2m$, hence $k_i k_j/2m$.

Null Model: Configuration Model (2)

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

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

- **Under null model, both the degree distribution and the total number of edges are preserved.**

Notice: This model applies to both weighted and unweighted networks. For weighted networks we use the weighted degree (sum of the edge weights).

Modularity (1)

- **Modularity of partitioning S of graph G:**

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

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

Normalizing const.: $-1 \leq Q \leq 1$

$$A_{ij} = \begin{cases} 1, \text{if } i \rightarrow j \\ 0, \text{otherwise} \end{cases}$$

(if G is weighted then A_{ij} is the edge weight)

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

Modularity (2)

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

Equivalently modularity can be written as:

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

- A_{ij} represents the edge weight between nodes i and j;
- k_i and k_j are the sum of the weights of the edges attached to nodes i and j, respectively;
- $2m$ is the sum of all of the edge weights in the graph;
- c_i and c_j are the communities of the nodes; and
- δ is a simple delta function. $\delta(c_i, c_j) = 1 \; if \; c_i = c_j \; else \; 0$

Idea: We can identify communities by maximizing modularity

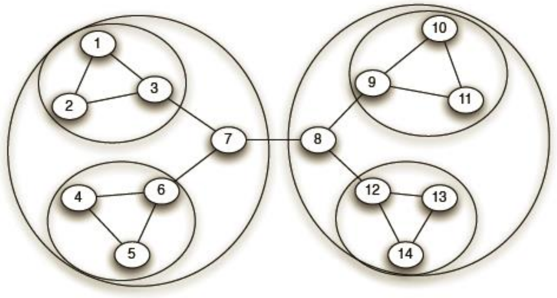
Unsupervised and Self-supervised Learning for Graphs

- **Network Community Detection**
  - Network Communities
  - **Louvain Algorithm**
- Strategies for Pre-training Graph Neural Networks

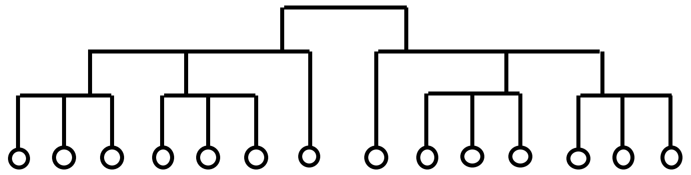
Louvain Algorithm (1)

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

Network and communities:



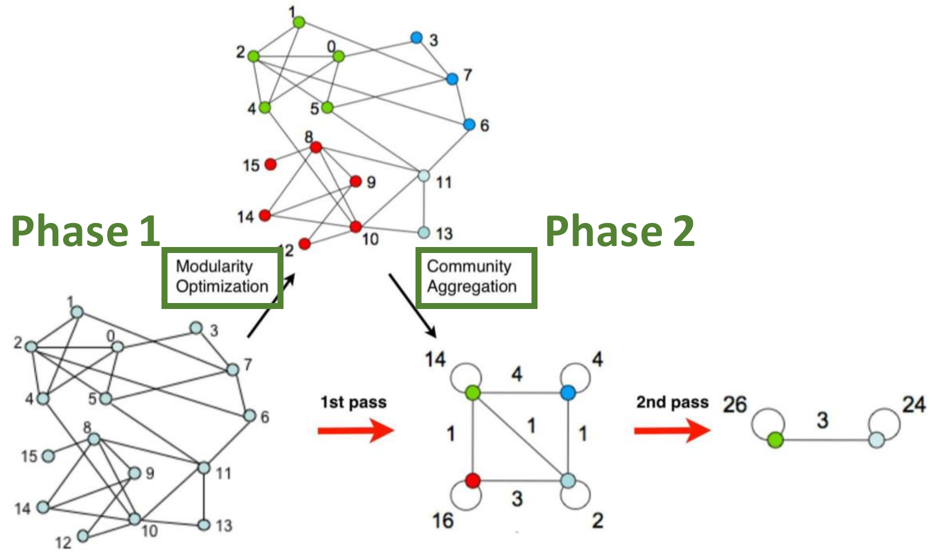
Dendrogram:



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

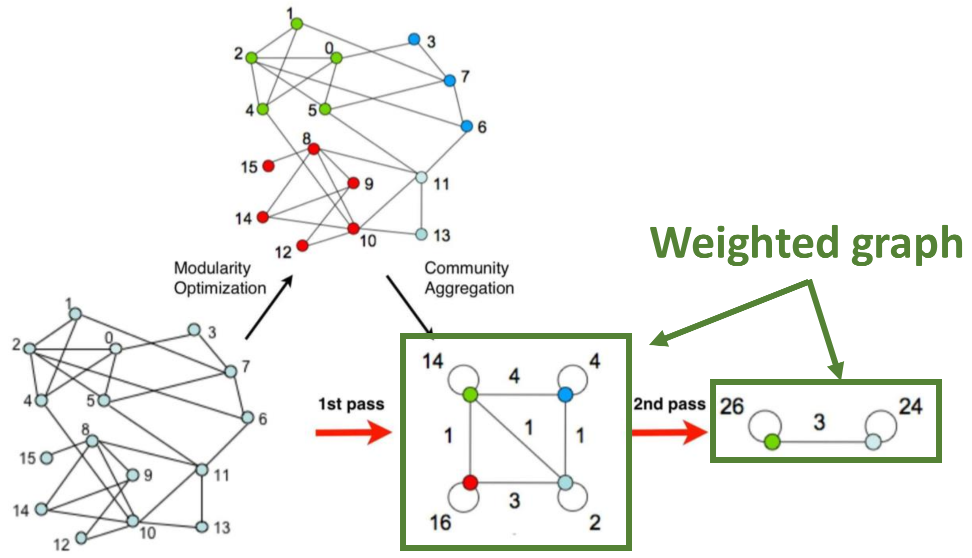
Louvain Algorithm (2)

- Louvain algorithm **greedily maximizes** modularity
- **Each pass is made of 2 phases:**
  - **Phase 1 (Partitioning):** Modularity is optimized by allowing only local changes to node-communities memberships
  - **Phase 2 (Restructuring):** The identified communities are aggregated into super-nodes to build a new network
  - **Run Phase 1 on the new network**
- The passes are repeated iteratively until no increase of modularity is possible.



Louvain Algorithm (3)

- Louvain algorithm considers graphs as **weighted**
  - The original graph can be unweighted (i.e., edge weights are all 1)
  - As the communities get identified and aggregated into super-nodes, weighted graphs are created (weights count the number of edges in the original graph)
  - Weighted version of Modularity is applied



Louvain: 1st phase (Partitioning)

- Put each node in a graph into a distinct community (one node per community)
- For each node i, the algorithm performs two calculations
  - Compute the modularity delta (ΔQ) when putting node i into the community of some neighbor j
  - Move i to a community of node j that yields the largest gain in ΔQ
- **Phase 1 runs until no movement yields a gain**

Note that the output of the algorithm depends on the order in which the nodes are considered. Research indicates that the ordering of the nodes does not have a significant influence on the overall modularity that is obtained.

Louvain 1st phase: Modularity Gain

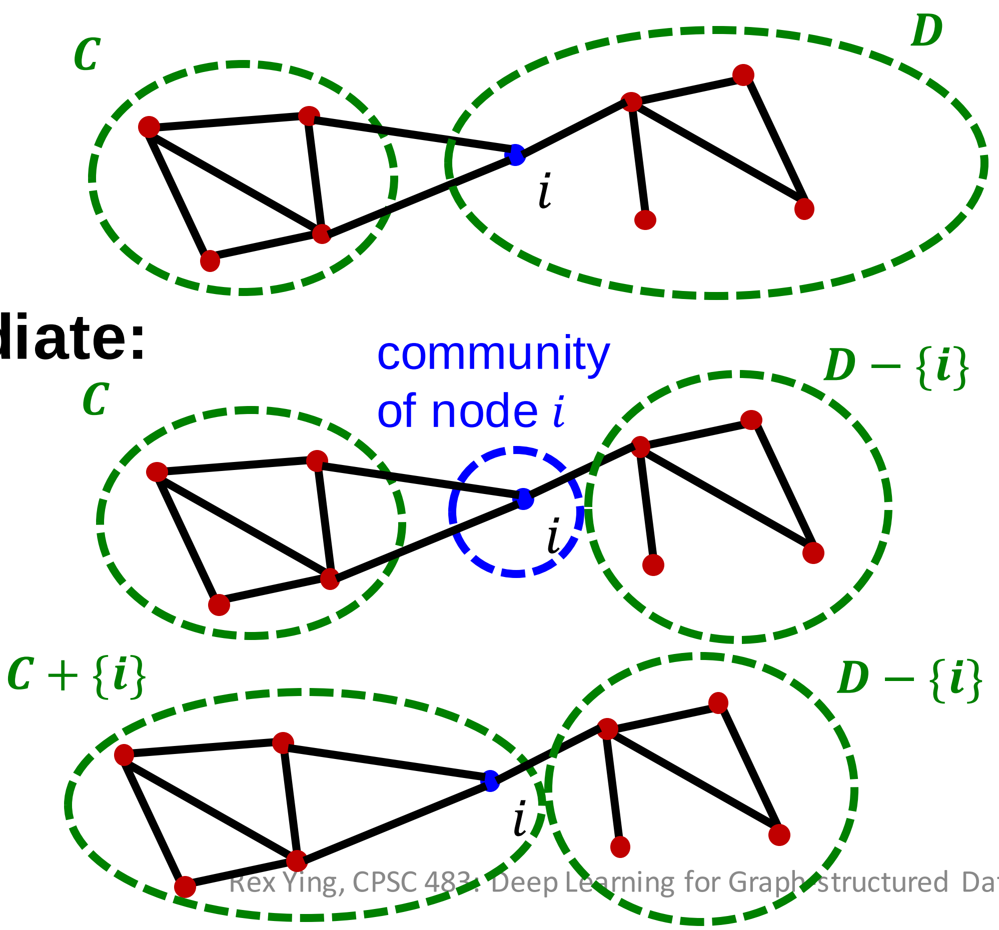
What is ΔQ if we move node i from community D to C?

$$\Delta Q(D \to i \to C) = \Delta Q(D \to i) + \Delta Q(i \to C)$$

Before:

Intermediate:

After:



Removing i from D

$\Delta Q(D \to i)$

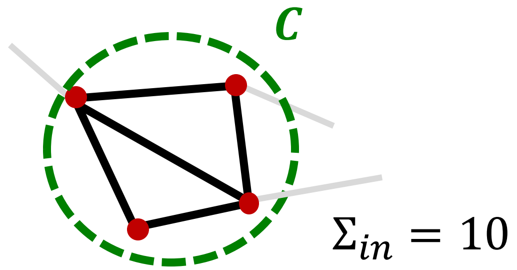
Merging i into C

$\Delta Q(i \to C)$

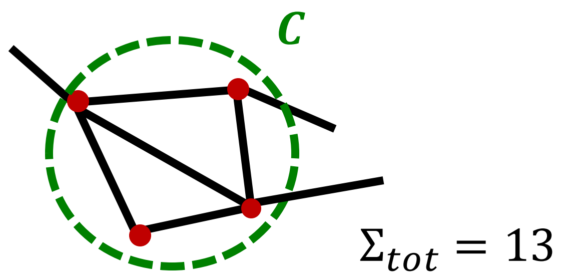
Louvain 1st phase: Deriving $\Delta Q(i \rightarrow C)$ (1)

- **Let's derive $\boldsymbol{\Delta Q(i \rightarrow C)}$**
- First, we derive modularity **within $\boldsymbol{C}$**, i.e., $Q(C)$.
- **Define:**
  - $\boldsymbol{\Sigma_{in}} \equiv \sum_{i,j \in C} A_{ij}$
    - sum of link weights <u>between</u> nodes in C
  - $\boldsymbol{\Sigma_{tot}} \equiv \sum_{i \in C} k_i$
    - sum of <u>all</u> link weights of nodes in C

$\boldsymbol{\Sigma_{in}}$:



$\boldsymbol{\Sigma_{tot}}$:



Louvain 1st phase: Deriving $\Delta Q(i \to C)$ (2)

- **Define:**
  - $\boldsymbol{\Sigma_{in}} \equiv \sum_{i,j \in C} A_{ij}$… sum of link weights <u>between</u> nodes in C
  - $\boldsymbol{\Sigma_{tot}} \equiv \sum_{i \in C} k_i$… sum of <u>all</u> link weights of nodes in C
- Then, we have

$$Q(C) \equiv \frac{1}{2m} \sum_{i,j \in C} \left[A_{ij} - \frac{k_i k_j}{2m} \right] = \frac{\sum_{i,j \in C} A_{ij}}{2m} - \frac{(\sum_{i \in C} k_i)(\sum_{j \in C} k_j)}{(2m)^2}$$

Links within the community

$$= \frac{\boxed{\Sigma_{in}}}{2m} - \left(\frac{\boxed{\Sigma_{tot}}}{2m} \right)^2$$

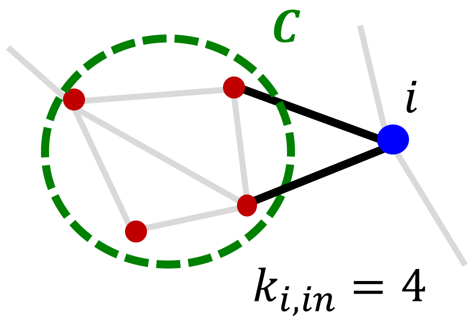
Total links

***$Q(C)$* is large when most of the total links are within-community links**

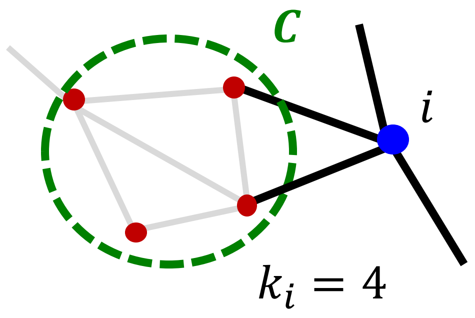
Louvain 1st phase: Deriving $\Delta Q(i \to C)$ (3)

- **Further define:**
  - $\boldsymbol{k_{i,in}} \equiv \sum_{j \in C} A_{ij} + \sum_{j \in C} A_{ji}$ … sum of link weights connecting node i and C
    - **Note that each edge gets counted twice, see formula**
  - $\boldsymbol{k_i}$… sum of all link weights (i.e., degree) of node i

$\boldsymbol{k_{i,in}}$:

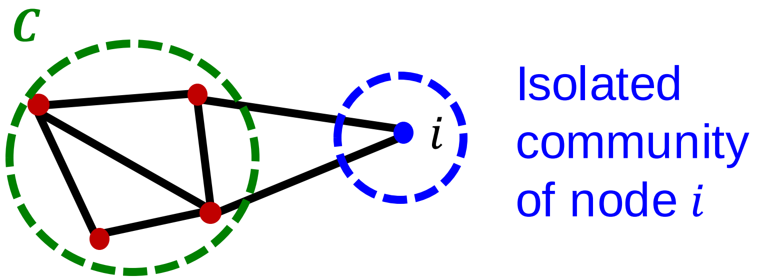


$\boldsymbol{k_i}$:



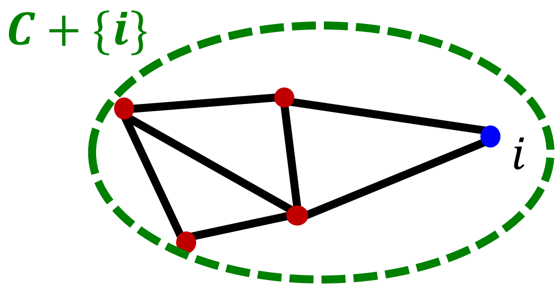
Louvain 1st phase: Deriving $\Delta Q(i \rightarrow C)$ (4)

Before merging



$$Q_{\text{before}} = Q(C) + Q(\{i\})$$
$$= \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2\right] + \left[0 - \left(\frac{k_i}{2m}\right)^2\right]$$

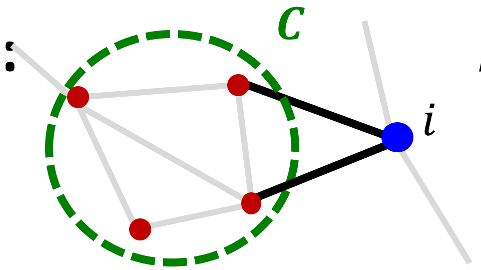
After merging



$$Q_{\text{after}} = Q(C + \{i\})$$
$$= \frac{\boxed{\Sigma_{in} + k_{i,in}}}{2m} - \left(\frac{\boxed{\Sigma_{tot} + k_i}}{2m}\right)^2$$

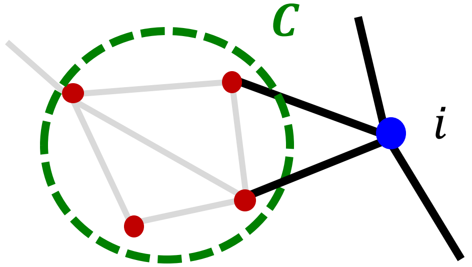
"Σ_{in}" of $\boldsymbol{C} + \{\boldsymbol{i}\}$ "Σ_{tot}" of $\boldsymbol{C} + \{\boldsymbol{i}\}$

Recall: $k_{i,in}$:



$k_{i,in} = 4$

k_i :



$k_i = 4$

Louvain 1st phase: Modularity Gain

- $\Delta Q(i \rightarrow C) = Q_{\text{after}} - Q_{\text{before}}$

$$= \left[\frac{\Sigma_{in}+k_{i,in}}{2m} - \left(\frac{\Sigma_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

- $\Delta Q(D \rightarrow i)$ can be derived similarly.

- **In summary, we can compute:**

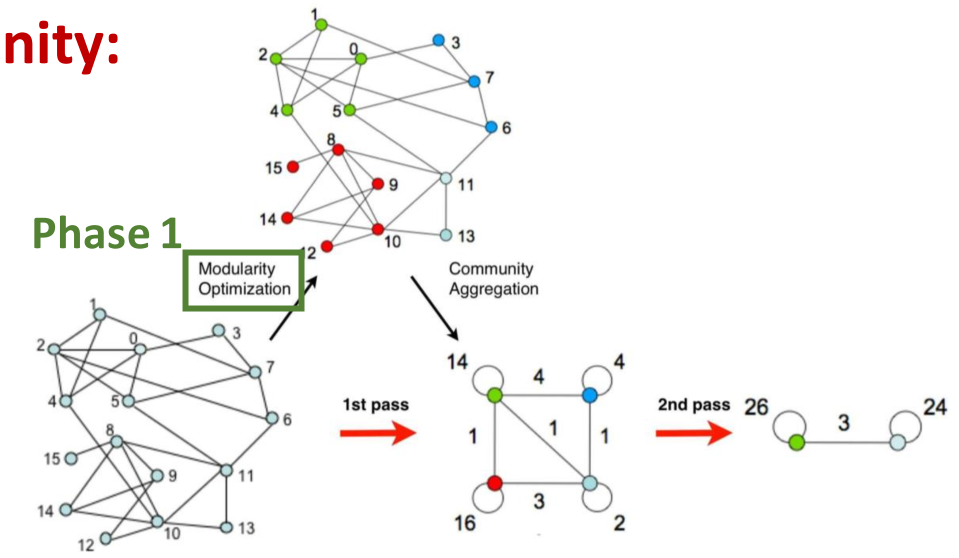
$$\boxed{\Delta Q(D \rightarrow i \rightarrow C) = \Delta Q(D \rightarrow i) + \Delta Q(i \rightarrow C)}$$

Louvain 1st Phase: Summary

- **Iterate until no node moves to a new community**
  - For each node $i \in V$ currently in community C, compute the best community C':

$$C' = \text{argmax}_{C'} \Delta Q(C \rightarrow i \rightarrow C')$$

  - If $\Delta Q(C \rightarrow i \rightarrow C') > 0$, then **update the community:**
    - $C \leftarrow C - \{i\}$
    - $C' \leftarrow C' + \{i\}$

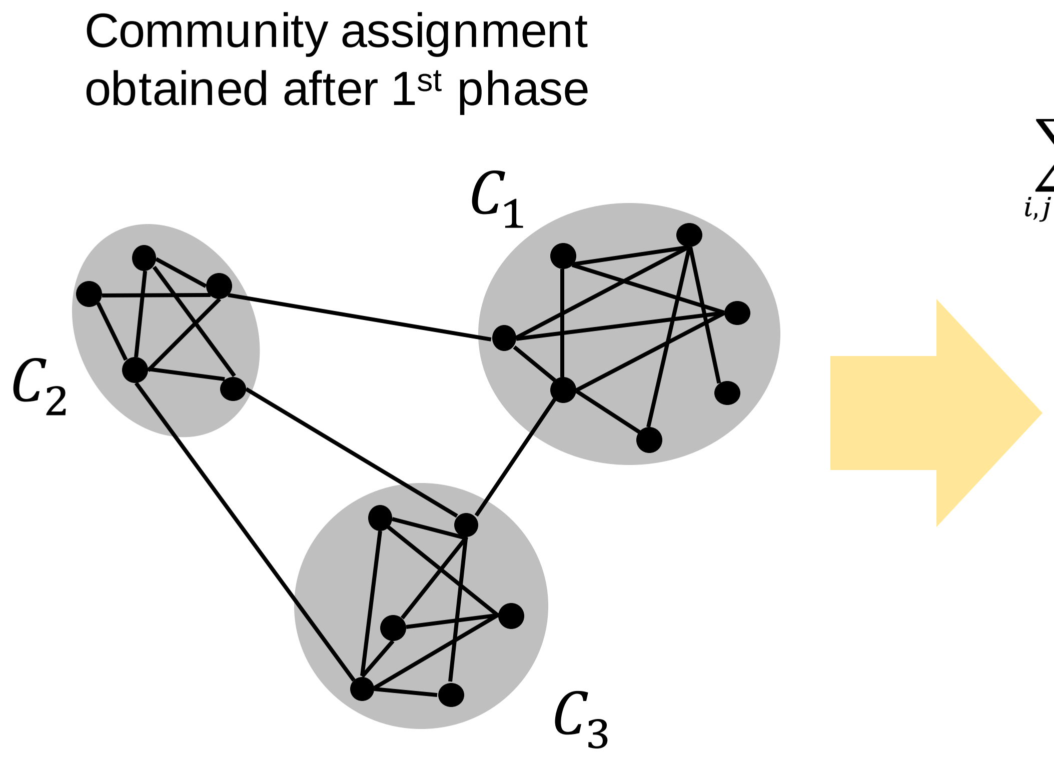


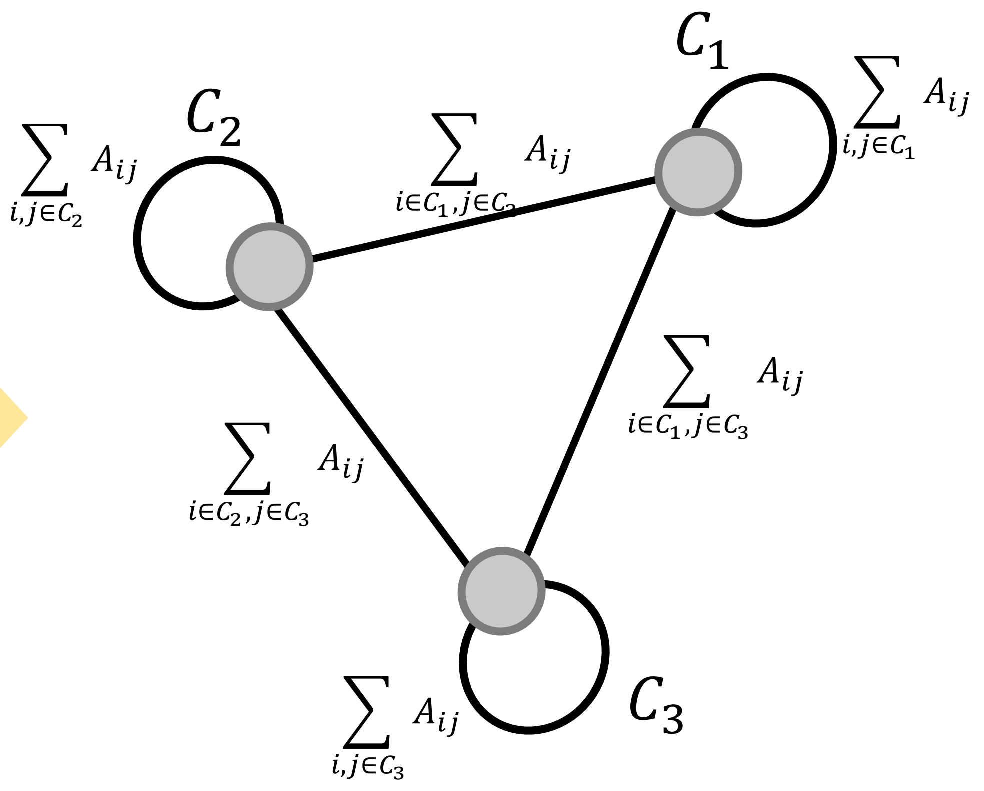
Louvain: 2nd phase (Restructuring)

- The communities obtained in the first phase are contracted into **super-nodes**, and the network is created accordingly:
  - Super-nodes are connected if there is at least one edge between the nodes of the corresponding communities
  - The weight of the edge between the two super-nodes is the sum of the weights from all edges between their corresponding communities
- **Phase 1 is then run on the super-node network**

Louvain 2nd Phase: Summary

- Super nodes are constructed by merging nodes in the same community.





C_1
C_2
C_3
$\sum_{i,j \in C_1} A_{ij}$
$\sum_{i,j \in C_2} A_{ij}$
$\sum_{i,j \in C_3} A_{ij}$
$\sum_{i \in C_1, j \in C_2} A_{ij}$
$\sum_{i \in C_1, j \in C_3} A_{ij}$
$\sum_{i \in C_2, j \in C_3} A_{ij}$

Interesting Questions

- Can we use GNNs to perform clustering / community detection?
    - Direct classification of nodes into clusters
    - Link prediction
- Can we use community information to improve GNN models
    - Additional node features
    - ClusterGCN
    - DiffPool
- Theoretical questions
    - Investigate the relation between clustering and over-smoothing phenomenon

Unsupervised and Self-supervised Learning for Graphs

- Network Community Detection
  - Network Communities
  - Louvain Algorithm

- **Strategies for Pre-training Graph Neural Networks**

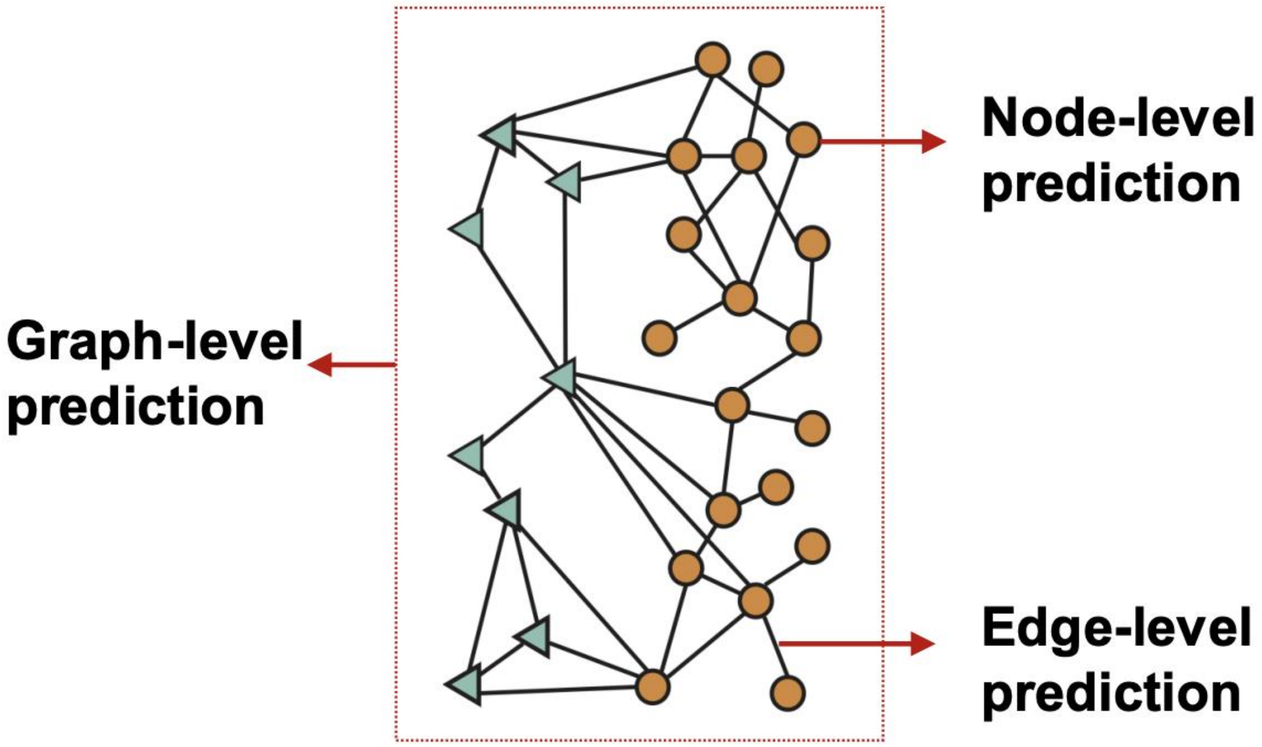
Supervised Learning on Graphs (1)

- Usually we train a GNN with some supervised labels:
  - **Node labels** y_v: in a citation network, which subject area does a node belong to
  - **Edge labels** y_{uv}: in a transaction network, whether an edge is fraudulent
  - **Graph labels** y_G: among molecular graphs, the drug likeness of graphs
- And apply a **loss function** $l(\cdot)$ to optimize the parameters

$$\min l(y, \underbrace{\hat{y}}_{\text{Predicted labels}})$$

Supervised Learning on Graphs (2)

- Different supervised labels on graph



Challenge of Supervised Learning on Graphs

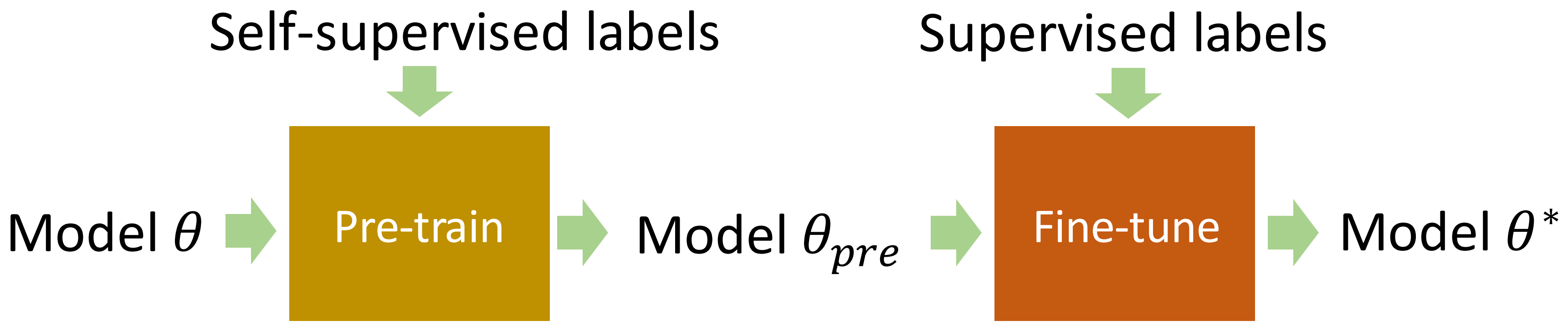
- Task-specific labels can be extremely **scarce**
  - Obtaining labels requires **expensive** lab experiments and human annotation
    - Machine learning models easily overfit to small training data
  - E.g., testing the chemical property of a molecule in a wet laboratory
- Graph data often contains **out-of-distribution samples**
  - Graphs in the training set are structurally very **different** from graphs in the test set
  - E.g., predict chemical properties of a new molecule which is different from all molecules in training set

Self-supervised Learning on Graphs (1)

- How to improve model's out-of-distribution prediction performance even with limited data?
- The solution: **self-supervised learning**!
  - We can find abundant supervision signals within the graph
    - **Node-level** y_v. Node statistics: such as clustering coefficient, PageRank
    - **Edge-level** y_{uv}. Link prediction: hide the edge between two nodes, predict if there should be a link
    - **Graph-level** y_G. Graph statistics: for example, predict if two graphs are isomorphic
  - Self-supervised learning can inject **domain knowledge** into a model
    - Model can generalize well without many task-specific labeled data

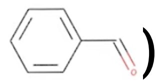
Self-supervised Learning on Graphs (2)

- Key idea: Use self-supervised signals to **pre-train** a model, then **finetune** the model with (scarce) training data
  - **Pre-train**: train the model on relevant tasks with self-supervised signals
  - **Finetune**: adapt the model to downstream task by using task-specific labels to tune the pretrained model



Pre-training GNNs (1)

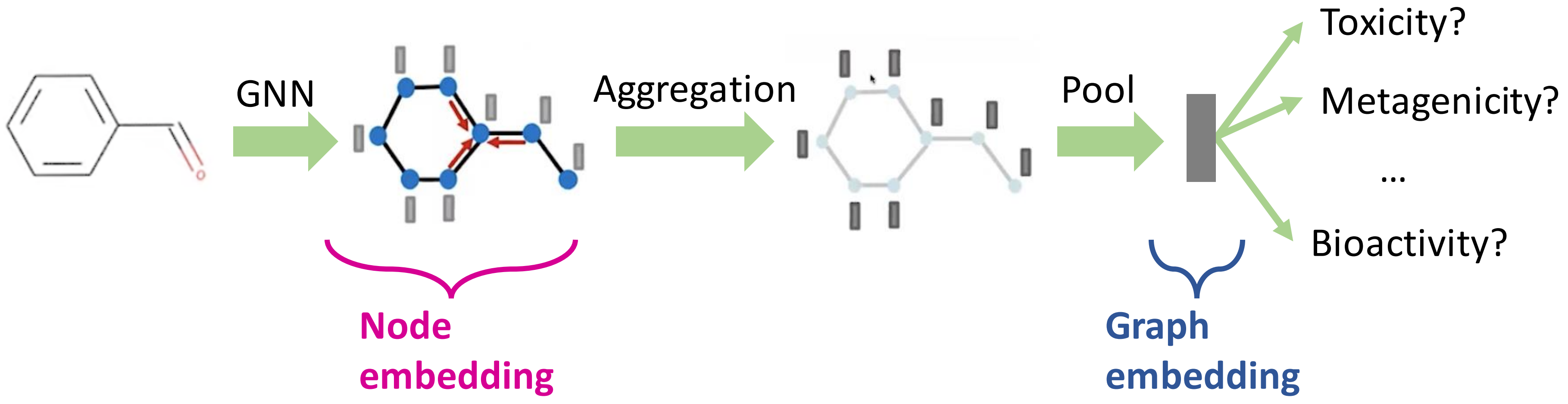
- How to design pre-training strategies for GNNs?
- Let’s think about **molecular property prediction task**
  - Node: atom
  - Edge: interaction between two atoms (bonds)
  - Task: Given a molecular graph, predict its corresponding chemical property

$f($  $) = \begin{cases} \text{Toxicity?} \\ \text{Metagenicity?} \\ \vdots \\ \text{Bioactivity?} \end{cases}$

model

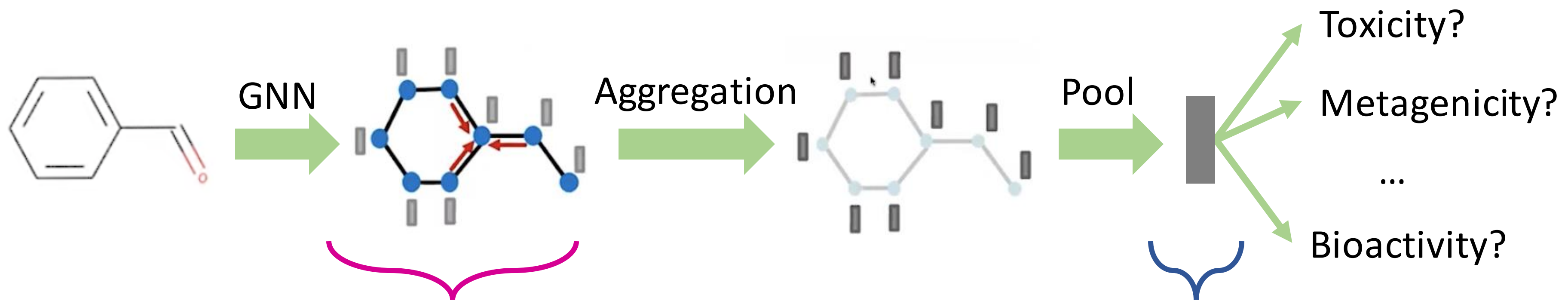
Pre-training GNNs (2)

- We can apply GNN to learn the molecular graph representation
  - **Node level**: Iteratively perform neighbor aggregation to obtain **node** representation
  - **Graph level**: Use pooling operation to obtain **graph** representation



Pre-training GNNs (3)

- How to pre-train GNN for molecular graph?
  - We can pre-train both **node** and **graph** embeddings
  - GNN can capture domain-specific knowledge of both **local** and **global** structure



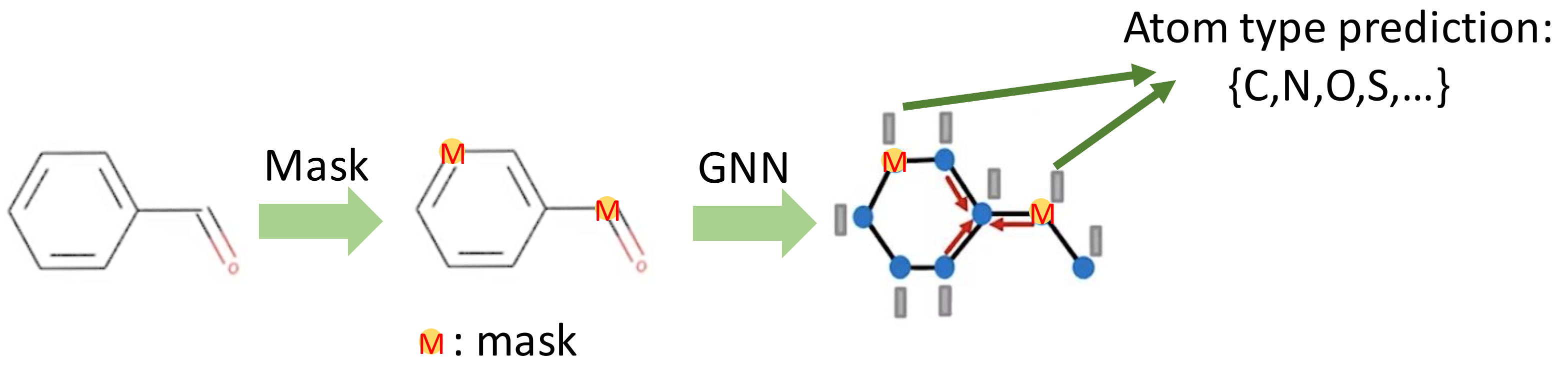
Pre-train node embeddings on node labels

Pre-train graph embeddings on graph labels

- Some strategies of pre-training GNNs
  - Attribute masking
  - Context pre-training
  - Attribute prediction
  - Contrastive learning

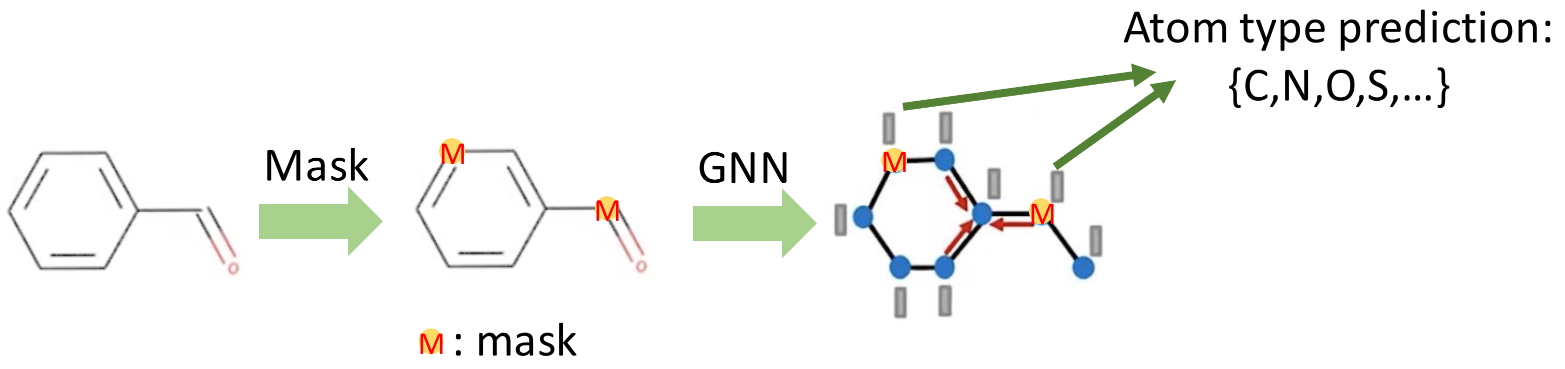
Node-level Pre-training: Attribute Masking (1)

- Node-level self-supervised task: Attribute Masking
    - **Self-supervised** signal: original node feature (atom type)
    - Mask some node features
    - Use GNN to generate node embeddings
    - Apply a linear model on the embeddings to predict the masked node feature



Node-level Pre-training: Attribute Masking (2)

- Intuition: atoms on the molecular graph follow **chemistry rules**
- Attribute masking can enforce GNN to learn such **domain knowledge**
  - Valency
  - Electronic or steric properties of functional groups

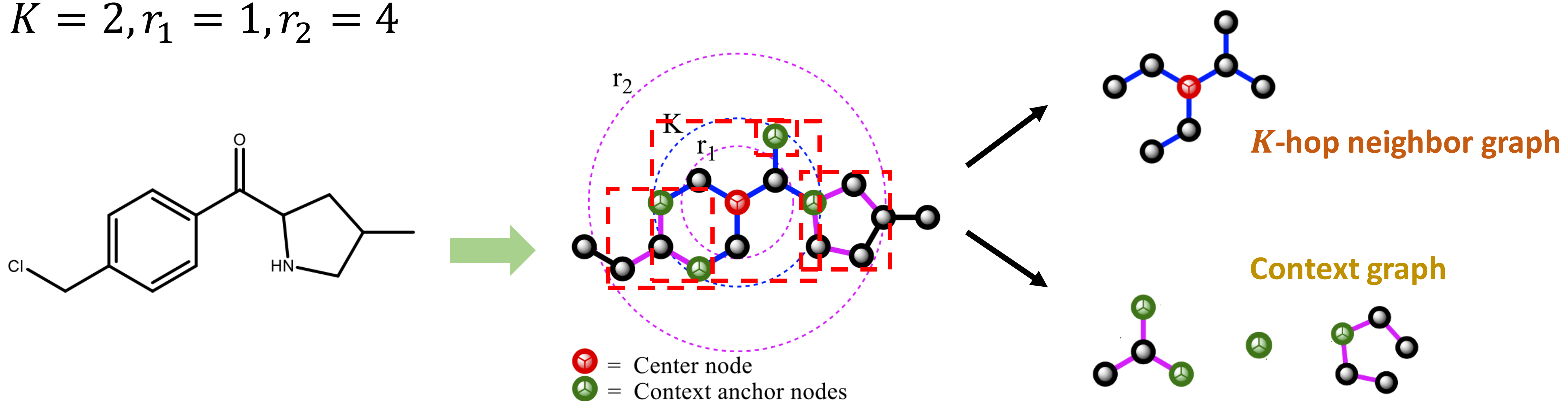


Node-level Pre-training: Context Prediction (1)

- Node-level self-supervised task: Context prediction
  - **Self-supervised** signal: context graph
- **K-hop neighbor graph** G_v^K
  - The subgraph containing all nodes and edges that are at most K-hop away from v
- **Context graph** $G_v^c(r_1, r_2)$
  - The subgraph containing all nodes and edges that are between r_1-hop and r_2-hop away from v
  - It is a ring of width $r_2 - r_1$
- Given a center node v, we require $r_1 < K$ and the nodes which are shared between neighbor and context graph are **context anchor nodes**

Node-level Pre-training: Context Prediction (2)

- An example of ***K*-hop neighbor graph** G_v^K and **Context graph** $G_v^c(r_1, r_2)$
  - $K = 2, r_1 = 1, r_2 = 4$



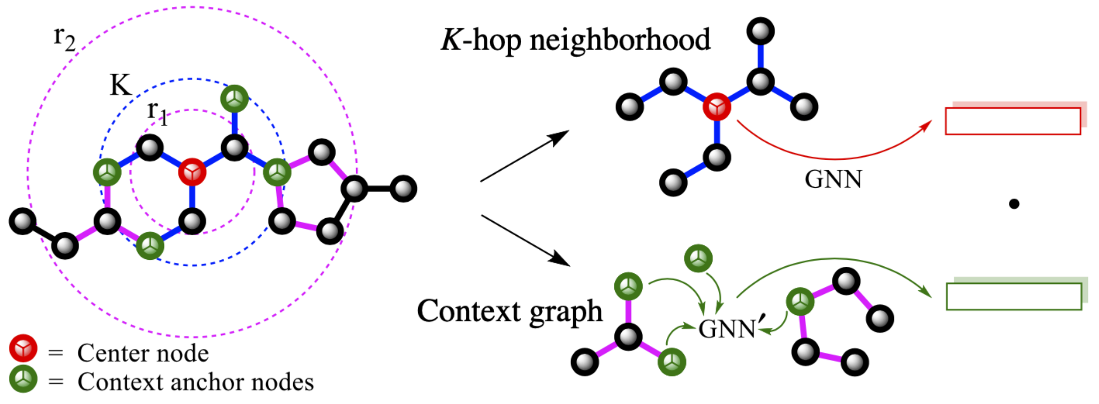
Node-level Pre-training: Context Prediction (3)

- Node-level self-supervised task: Context prediction
  - **Key idea:** use subgraphs to predict their surrounding graph structures
  - Encode the G_v^K using the **main GNN** to obtain center node embedding $\mathbf{h}_v^K$
  - Encode the $G_v^c(r_1, r_2)$ using **context GNN** to obtain the context anchor node embeddings
    - Context GNN is an auxiliary GNN
  - Simply **average** all the context anchor nodes' embedding to obtain context embedding $\mathbf{h}_v^c$
  - **Minimize the distance** between $\mathbf{h}_v^K$ and $\mathbf{h}_v^c$:

$$\min \sum_v 1 - \underbrace{\sigma}_{\text{Inner product}}(\mathbf{h}_v^K \underbrace{\cdot}_{\text{Inner product}} \mathbf{h}_v^c)$$

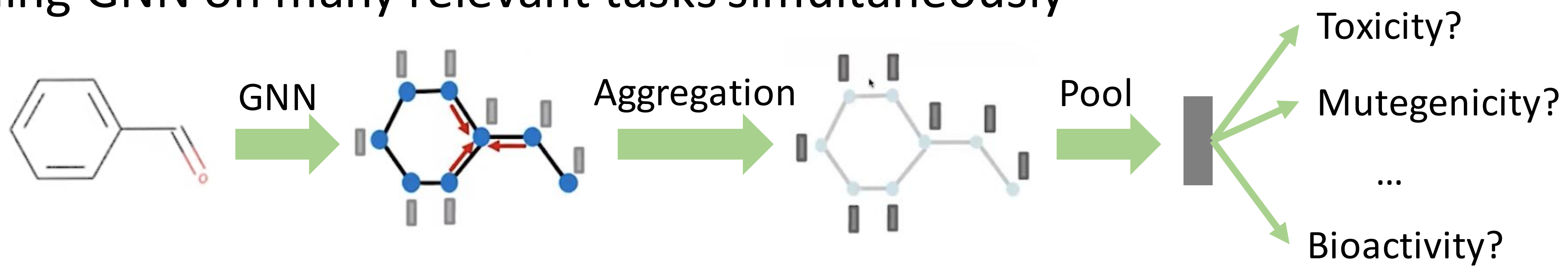
Node-level Pre-training: Context Prediction (4)

- Intuition: subgraphs that are surrounded by similar contexts are semantically similar
- Pre-trained with context prediction task, GNN can map nodes appearing in similar structural contexts to nearby embeddings



Graph-level Pre-training: Attribute Prediction

- Graph-level supervised task: Attribute prediction
  - **Multi-task supervised pretraining**: Predict a diverse set of supervised labels
    - Toxicity? Mutegenicity? Bioactivity? Solubility?
    - Each property corresponds to a **binary classification task**
    - **After training, the model can be fine-tuned to perform additional downstream tasks**
  - Training GNN on many relevant tasks simultaneously



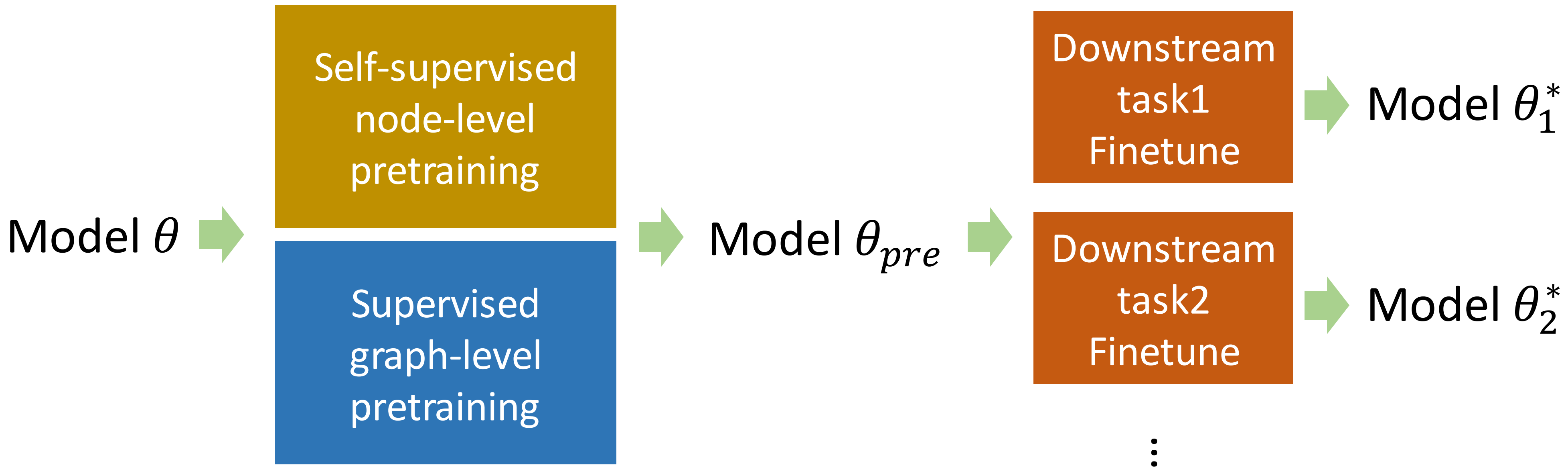
Graph-level Pre-training: Contrastive Learning

- Contrastive learning is a popular self-supervised learning strategy
  - Train the model to learn to contrast positive and negative examples

- **Positive** example: molecules in the dataset
  - Maximize predicted probability
- **Negative** example: **perturbed** molecules
  - Minimize predicted probability

Pre-training on Graph: Overview

- Pretraining a GNN on molecular graph
  - Node-level pretraining: attribute masking and context prediction
  - Graph-level pretraining: attribute prediction



Model θ
Self-supervised node-level pretraining
Supervised graph-level pretraining
Model θ_{pre}
Downstream task1 Finetune
Downstream task2 Finetune
⋮
Model θ_1^*
Model θ_2^*

Summary

- Unsupervised learning for graph: community detection
  - What is Network Communities?
    - Network community measure **Modularity**
  - Detection method: Louvain Algorithm
    - Phase 1 (**Partitioning**) and Phase 2 (**Reconstructing**)
- Strategies for pretraining Graph Neural Networks
  - **Node-level** pretraining
  - **Graph-level** pretraining