Machine Learning with Graphs

Prepared by: Justin Ho Jia Yinn

Based on Lectures of CSCE224W by Jure Leskovec in Fall 2019
(Stanford University)

Contents

1 Properties of Network **6**
- 1.1 Degree Distribution $P(k)$ . . . 6
- 1.2 Paths in a Graph (h) . . . 6
  - 1.2.1 Distance (Shortest/Geodesic Path) . . . 6
  - 1.2.2 Diameter . . . 6
  - 1.2.3 Average Path Length . . . 6
- 1.3 Clustering Coefficient C . . . 6
- 1.4 Connectivity s . . . 7

2 Erdös-Renyi Graph (Random/Poisson Graph) **8**
- 2.1 Degree Distribution . . . 8
- 2.2 Clustering Coefficient . . . 8
- 2.3 Distance (Shortest Path) . . . 9
- 2.4 Connectivity . . . 10
- 2.5 Real Networks vs Random Graphs . . . 10
  - 2.5.1 Problems with Random Graph Models . . . 10
  - 2.5.2 Importance of Analysis of Random Graphs . . . 10

3 Small World Model (Watts-Strogatz Model) **11**
- 3.1 Locality . . . 11
- 3.2 Watts-Strogatz Model (Algorithm) . . . 12

4 Kronecker Graph Model **13**
- 4.1 Intuition . . . 13
- 4.2 Kronecker Product . . . 13
- 4.3 Kronecker Graph . . . 13
- 4.4 Stochastic Kronecker Graph . . . 14

5 Motifs in Networks **15**
- 5.1 Decomposing Structure . . . 15
- 5.2 Motifs . . . 16
- 5.3 Significance of a Motif . . . 16
- 5.4 Configuration Model . . . 17
  - 5.4.1 Decision Decision of the Null Model . . . 17
  - 5.4.2 Method 1: Spokes . . . 17
  - 5.4.3 Method 2: Switching . . . 18
- 5.5 Variations of Motifs . . . 18

6 Graphlets as Node-Level Characteristics **19**
- 6.1 Key Definitions and Concepts . . . 19
- 6.2 Graphlets . . . 19
- 6.3 Automorphism Orbits . . . 19
  - 6.3.1 Definition . . . 19
  - 6.3.2 Example . . . 20
  - 6.3.3 Graphlet Degree Vector . . . 21

7 Finding Motifs and Graphlets **22**
- 7.1 Exact Subgraph Enumeration (ESU) Algorithm . . . 22
- 7.2 Counting . . . 23
  - 7.2.1 Graph Isomorphism . . . 24

8 Structural Roles in Networks **25**
- 8.1 Defining Roles and Groups . . . 25
  - 8.1.1 Example of Roles and Groups . . . 25
  - 8.1.2 Structural Equivalence . . . 26
- 8.2 Why Roles? . . . 27
- 8.3 Discovering Roles using `RolX` . . . 27
  - 8.3.1 Overview of `RolX` . . . 27

8.3.2 Recursive Feature Extraction . . . 28
8.3.3 Making Sense of Roles . . . 29

9 Network Communities **30**
9.1 Concept of a Community . . . 30
9.1.1 Structural vs. Interpersonal Role of an Edge . . . 30
9.1.2 Information Flow . . . 30
9.2 Strength of Ties . . . 30
9.3 Finding Network Communities . . . 31
9.3.1 Defining the Null Model . . . 31
9.3.2 Modularity Measure . . . 31
9.4 Louvain's Algorithm . . . 32
9.4.1 At a High Level . . . 32
9.4.2 Partitioning . . . 32
9.4.3 Restructuring/Aggregation . . . 33

10 Overlapping Communities **34**
10.1 Community Affiliation Graph Model (AGM) . . . 34
10.1.1 Assigning Edges by Probability . . . 35
10.1.2 Flexibility of AGMs . . . 35
10.2 Detecting Communities Using AGMs . . . 36
10.2.1 Maximum Likelihood Estimation . . . 36
10.2.2 Graph Likelihood $P(G|F)$ with Bernoulli Distributions . . . 36
10.2.3 Modelling Membership Strength . . . 37
10.2.4 Graph Likelihood $P(G|F)$ of AGMs . . . 37
10.2.5 Gradient Ascent for Optimization . . . 38

11 Spectral Clustering **39**
11.1 Defining the Problem . . . 39
11.1.1 Naive Cut . . . 39
11.1.2 Conductance as a Criterion `Shi-Malik, (1997)` . . . 40
11.2 Eigenvalues and Eigenvectors in Spectral Graph Partitioning . . . 40
11.2.1 Calculating Neighbours . . . 40
11.2.2 *d-Regular* Graph . . . 40
11.2.3 Disconnected **d-regular** Graph . . . 41
11.2.4 Intuition on Finding Eigenvalue/Eigenvector Pairs . . . 42
11.3 Matrix Representations . . . 42
11.3.1 Properties of the Laplacian Matrix L . . . 43
11.4 Solving λ_2 as a Minimization Problem . . . 43
11.4.1 Meaning of $x^\top Lx$ on Graph G . . . 44
11.4.2 Fiedler's Optimal Cut (1973) . . . 44
11.4.3 Rayleigh Theorem . . . 45
11.4.4 Approximation Guarantee of Spectral Clustering . . . 45
11.5 Spectral Clustering Algorithm . . . 45
11.5.1 Finding the Split . . . 45
11.5.2 Examples of Spectral Clustering . . . 46
11.5.3 k-way Spectral Clustering . . . 47
11.5.4 Clustering Multiple Eigenvectors . . . 47
11.5.5 Reasons for Clustering Multiple Eigenvectors . . . 47
11.5.6 Determining k . . . 47

12 Motif-Based Spectral Clustering **48**
12.1 Finding Motif-Based Clusters . . . 49
12.2 Motif-Based Spectral Clustering Algorithm . . . 49
12.2.1 Transformation of G . . . 49
12.2.2 Decomposition of $W^{(M)}$ . . . 49
12.2.3 Sweeping for Community Identification . . . 50
12.2.4 Approximation Guarantee . . . 50
12.3 Application with Unknown Motif: Food Webs . . . 50

12.3.1 Sweeping Profile . . . 51
12.3.2 Ground Truth Food Web . . . 51
12.4 Application with Known Motif: Gene Regulatory Networks . . . 52
12.4.1 Yeast Regulatory Network . . . 53
12.5 Other Partitioning Methods . . . 53

13 Message Passing and Node Classification **54**
13.1 Correlations in Networks . . . 54
13.2 Guilt-by-Association . . . 54
13.2.1 Formalizing the Problem . . . 54
13.3 Collective Classification . . . 55
13.4 Probabilistic Relational Classifier . . . 55
13.4.1 Algorithm . . . 55
13.4.2 Limitations of Algorithm . . . 55
13.5 Iterative Classification . . . 55
13.5.1 Basic Architecture . . . 56
13.6 Loopy Belief Propogation . . . 56
13.6.1 Message Passing . . . 56
13.6.2 Cyclic Graphs . . . 57
13.6.3 Advantages and Limitations . . . 57
13.6.4 Application: Online Fraud Detection (Pandit et al., 2007) . . . 58

14 Graph Representational Learning **59**
14.1 Feature Learning in Graphs . . . 59
14.1.1 Challenges in Graph-Based Representational Learning . . . 60
14.2 Embedding Nodes . . . 61
14.2.1 Learning Node Embeddings . . . 61
14.2.2 Two Key Components . . . 62
14.2.3 Shallow Encoding . . . 62
14.2.4 Defining Similarity . . . 62
14.3 DeepWalk Perozzi et al. 2014 . . . 63
14.3.1 Random Walk Node Embeddings . . . 63
14.3.2 Problem Setup . . . 64
14.3.3 Feature Learning as an Optimization Problem . . . 64
14.3.4 The Optimization Problem . . . 64
14.3.5 Estimation based on Negative Sampling . . . 65
14.4 Node2Vec Grover & Leskovec, 2016 . . . 65
14.4.1 Local and Global Neighbourhoods . . . 65
14.4.2 Interpolating between DFS and BFS . . . 65
14.4.3 Biased Random Walks . . . 66
14.4.4 Node2Vec Algorithm . . . 66
14.5 Usage of Node Embeddings . . . 66

15 Introduction to Graph Neural Networks **67**
15.1 Basis for GNNs . . . 67
15.1.1 Neighbourhood Aggregation . . . 68
15.1.2 Depth of Model . . . 68
15.1.3 Math Behind the Deep Encoder . . . 69
15.2 Usage of Deep Encoders . . . 69
15.2.1 Unsupervised Task . . . 69
15.2.2 Supervised Task (Node Classification) . . . 70
15.3 Inductive Capabilities in GNNs . . . 70

16 General Framework of Graph Neural Networks **71**
16.1 Defining a Single GNN Layer . . . 71
16.1.1 Message Computation . . . 72
16.1.2 Message Aggregation . . . 72
16.1.3 Information Loss in Aggregation . . . 72
16.1.4 Graph Convolutional Network . . . 72

16.1.5 GraphSAGE . . . 73
16.1.6 Graph Attention Network . . . 73
16.1.7 Incorporating Other Deep Learning Modules . . . 75
16.2 Stacking GNN Layers . . . 77
16.2.1 The Problem of Oversmoothing and the Receptive Field . . . 77
16.2.2 Expressivity of Shallow GNNs . . . 78
16.2.3 Skip Connections in Deep GNNs . . . 79
16.3 Graph Augmentations . . . 80
16.3.1 Node Level Feature Augmentation . . . 80
16.3.2 Structure Level Feature Augmentation . . . 81
16.3.3 Adding Virtual Nodes and Edges . . . 82
16.3.4 Neighbourhood Sampling . . . 83
16.4 Machine Learning Task . . . 83
16.4.1 Node-Level Prediction . . . 84
16.4.2 Edge-Level Predictions . . . 84
16.4.3 Graph Level Prediction . . . 85

17 Theoretical Basis of Graph Neural Networks 87
17.1 Differentiating Nodes . . . 87
17.1.1 Local Neighbourhood Structure . . . 87
17.1.2 Symmetric Nodes . . . 88
17.1.3 Computational Graphs . . . 88
17.2 Injective Functions . . . 89
17.3 Neighbourhood Aggregations of Popular Architectures . . . 90
17.3.1 GCN and Mean Pooling . . . 90
17.3.2 GraphSAGE and Max Pooling . . . 90
17.4 Injective Multi-Set Function . . . 91
17.4.1 Universal Approximation Theorem . . . 91
17.5 Ranking Pooling Methods . . . 91

18 Heterogeneous Graphs and Relational Architectures 92
18.1 Relational Graph Convolution Networks . . . 92
18.2 Issues of Scalability . . . 93
18.3 Regularization and Scaling Methods . . . 93
18.3.1 Block Diagonal Matrices . . . 93
18.3.2 Basis Learning . . . 94

1 Properties of Network

Four common properties to observe

1. Degree Distribution $P(k)$
2. Paths in Graph h
3. Clustering Coefficient C
4. Connectivity s

1.1 Degree Distribution $P(k)$

Probability that the randomly chosen node has a degree (number of connecting edges) of k.

$$P(k) = \frac{N_k}{N}$$

whereby N_k is the number of nodes with degree k and N the total number of nodes in the Graph.
For directed graphs, `in degree` and `out degree` can be distinguished.

1.2 Paths in a Graph (h)

`Path` is a sequence of nodes linked together from one to the next.

$$P_n = \{v_1, v_2, v_3, \ldots, v_N\}$$

Paths allow `intersection` of nodes. Eg. **AB**C**D**V**DCA**BS

1.2.1 Distance (Shortest/Geodesic Path)

Minimum number of edges (or sum of weights) to connect a pair of nodes (u, v)
Undirected Graph $\rightarrow h_{u,v} = h_{v_u}$ (symmetric)
Directed Graph $\rightarrow h_{u,v} \neq h_{v_u}$ (asymmetric)

1.2.2 Diameter

`Definition` the maximum *shortest path* for every node in a graph
Tends to be extremely "fragile". Potential huge change with minor changes in graph

1.2.3 Average Path Length

Calculating connected nodes and averaging the shortest path lengths.

$$\bar{h} = \frac{1}{2E_{max}} \sum_{i,j \neq i} h_{i,j}$$

E_{max} is the maximum number of possible edges $\binom{n}{2} = \frac{n(n-1)}{2}$.

1.3 Clustering Coefficient C

`Informal Definition` - Measure of how connected the node i's neighbours are to one another
For each node

$$C_i = \frac{e_i}{\binom{k_i}{2}} = \frac{e_i \times 2! \times (k_i - 2)!}{k!} = \frac{2e_i}{k_i(k_i - 1)}, C_i \in [0, 1]$$

1. e_i is the number of edges between neighbours of node i
2. k_i is the degree of node i
3. $\binom{k_i}{2}$ is the maximum number of the edges that is possible given k degree

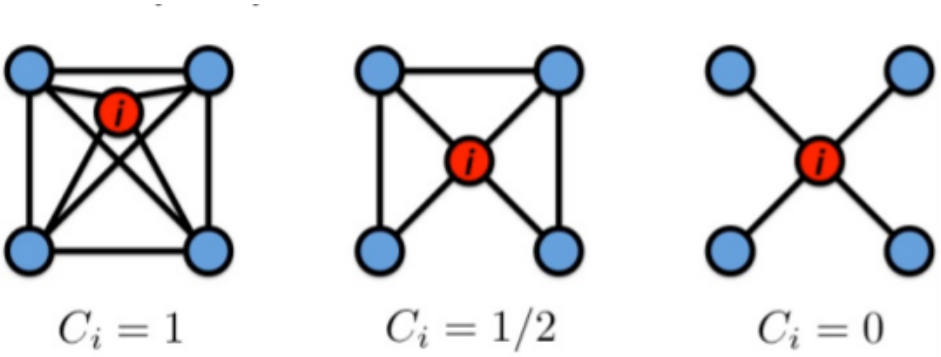


Figure 1: Clustering Coefficients

Hence, the value C_i can be interpreted as the fraction of "realized" edges over the theoretical maximum.

`Clustering Coefficient` - average of all C_i

$$C = \frac{1}{N} \sum C_i$$

1.4 Connectivity s

`Connected Component` - there exists a path for every pair of nodes (u, v) in the component.
Distribution of nodes for each component - What is the size of the largest component?

2 Erdös-Renyi Graph (Random/Poisson Graph)

Stochastic (Random Graph) - for given parameters, different graphs can be realized.

G_{np}, whereby G has n number of nodes with iid probability of p for an edge (u, v) *to appear*

2.1 Degree Distribution

Degree distribution of G_{np} is `binomial`.

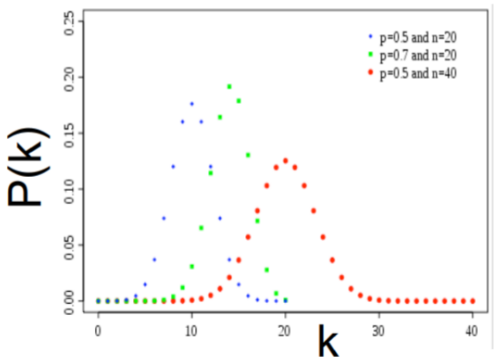


Figure 2: Binomial Distribution of Degree k with Different Parameters

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

1. $\binom{n-1}{k}$ - select k nodes out of $n-1$ choices
2. p^k - probability of selecting k number of nodes
3. $(1-p)^{n-1-k}$ - the probability of not selecting the rest of the nodes $(n-1-k)$

Mean of Degree Distribution

$$\bar{k} = p(n-1)$$

Variance of Degree Distribution

$$\sigma^2 = p(1-p)(n-1)$$

2.2 Clustering Coefficient

Given edges are iid with the probability p, then we have the average number of edges that connects to other neighbours of a given node i is

$$E[e_i] = p\frac{k_i(k_i-1)}{2}$$

$$E[C_i] = \frac{2E[e_i]}{k_i(k_i-1)} = \frac{pk_i(k_i-1)}{k_i(k_i-1)} = p = \frac{\bar{k}}{n-1} \approx \frac{\bar{k}}{n}$$

Note that this means that as n grows bigger, the cluster coefficient converges to 0

$$\lim_{n\to\infty} \frac{\bar{k}}{n} = \lim_{n\to\infty} \frac{p(n-1)}{n} = 0$$

2.3 Distance (Shortest Path)

Intuition- For a given subgraph, how much does it expand by connecting to other vertices not in the subgraph?
`Expansion` - A graph $G = (V, E)$ has expansion α if $\forall S \subset V$, such that the number of edges leaving $S \geq \alpha min(|S|, |V - S|)$

$$\alpha = \min_{S \subset V} \frac{number\ of\ edges\ leaving\ S}{min(|S|, |V - S|)}$$

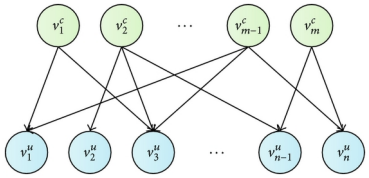


Figure 3: Expansion for nodes from C to nodes in U

Theorem

For a graph with n nodes and expansion of α, then the shortest path length must be $O(\log n/\alpha)$
Since we know $\log n > \log np$ for a G_{np}, the diameter (longest shortest path length of a graph) would be in the order of $O(\log n/\log np) = O(\log n)$.

Intuition

Much like a `BFS`, you have a log number of steps until all nodes are visited ones.
This is similar to growing out a binary tree.

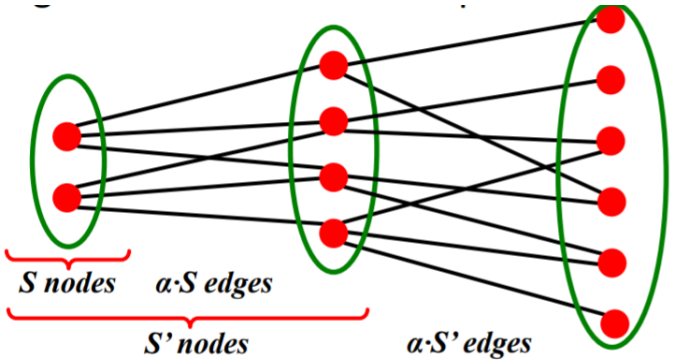


Figure 4: The Notion of an Expanding Graph. It would take $\log n$ steps to include all nodes

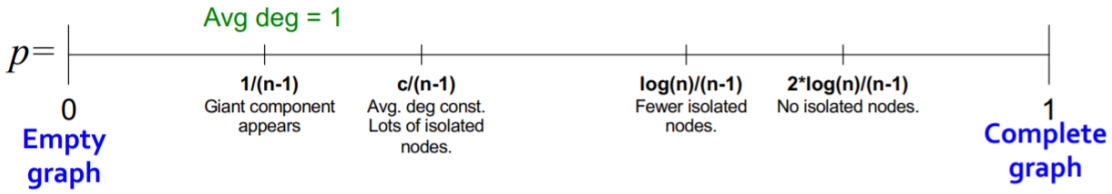


Figure 5: Evolution of Components of a Random Graph wrt p

2.4 Connectivity

Formation of `giant component` occurs when $k = 1$, or equivalently $p = \frac{1}{n-1}$.
Such "evolution" is characterized by Figure 5.

Intuition

With the expectation of each node having **at least** one edge, then a component would be able to form as a result.

2.5 Real Networks vs Random Graphs

How does G_{np} compare to real networks? Based on the network formed from `MSN` communication network, the following comparison is made in Table 1.

	`MSN`	G_{np} (n=180M)	Validity
Degree Distribution	Power law	Binomial	No
Avg. Path Length	6.6	$O(\log n) \approx 8.2$	Yes
Clustering Coefficient	0.11	$\frac{k}{n} \approx 8 \times 10^{-8}$	No
Connectivity ($\% \in$ Largest Component)	99% of nodes	Achieved when $k \approx 14$	Yes

Table 1: Summary of Network Properties of `MSN` Network and a Random Graph

2.5.1 Problems with Random Graph Models

1. `Degree Distribution` is different
2. `Giant Component` forms in real networks without a "transition phase" (when $k = 1$)
3. `Locality` is **absent** in random graphs

Real networks are **NOT** purely random.

2.5.2 Importance of Analysis of Random Graphs

1. Reference Model
2. Which properties of a graph emerge from `randomness` and which does not?

From the analysis above, we can see that small Avg Path Length can easily emerge from a random process, and so does highly connected networks.

3 Small World Model (Watts-Strogatz Model)

Conflicting Constraints

1. High Cluster Coefficient
2. Small Network Diameter

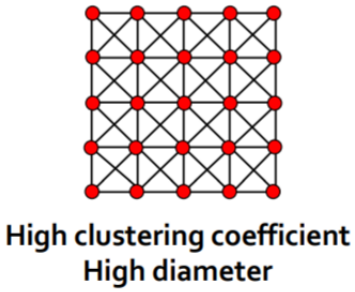


Figure 6: Graph with High Clustering and High Diameter

With a lot of edges, this allows high `clustering` because neighbouring nodes are connected to one another. **BUT**, "excess" edges introduce high diameter.

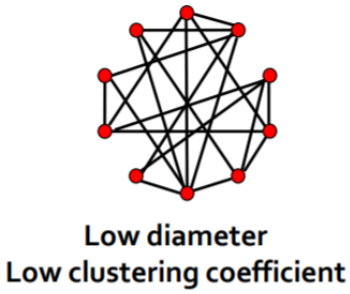


Figure 7: Graph with Low Clustering and Low Diameter

Sparse graphs have low `clustering` but the diameter (longest shortest path) is low due to fewer "redundant" edges.

3.1 Locality

Most real networks exhibit `triadic closure` (a friend of my friend is a friend).
Question - can we have a network such that we have $O(\log n)$ diameter `small world` and high cluster `triadic closure`?

1. `Clustering` implies Locality
2. `Randomness` implies Shortcuts (reducing shortest path)

3.2 Watts-Strogatz Model (Algorithm)

To generate a random graph with both of these properties, the following steps can be taken:

1. Begin with `low-dimensional lattice graph` (refer to Figure 8)
2. `Rewire` (introducing randomness) by selecting a node to connect to a random node with the probability p

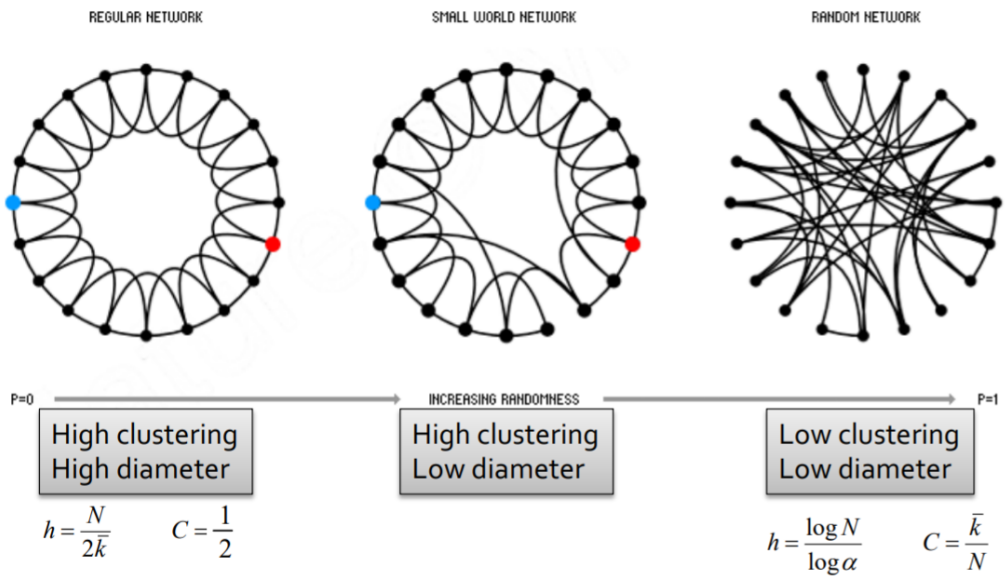


Figure 8: Transition from a Ring Lattice to a Random Graph

Intuition

It takes a lot of rewiring to remove clusters but a few to create shortcuts to reduce the diameter.

From a real world perspective, most friends are friends with people around them (`high locality`) but by having random offshoots like knowing one person out of your social group leads to a `small world`.

Problem

`Degree Distribution` remains incorrect!

4 Kronecker Graph Model

Method of generating realistic huge graphs. This means that we have to "correct" the problem with degree distribution in the `small world model`, which is achieved by the Kronecker Graph Model.

4.1 Intuition

1. `Recursively` generate graphs to capture **self-similarity**.
2. The whole graph exhibits similar properties of its subgraphs.
3. Individual friend networks would be similar to neighbourhood-level friend networks, and county-level friend networks ...

4.2 Kronecker Product

$$K_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}; K_2 = K_1 \bigotimes K_1 = \begin{bmatrix} K_1 & K_1 & 0 \\ 0 & K_1 & 0 \\ K_1 & K_1 & K_1 \end{bmatrix}$$

With K_1 as the initial matrix with dimension 3×3, K_n would have the dimension of $3^n \times 3^n$.

$$\underset{N \times M}{\mathbf{C}} = \underset{N \times M}{\mathbf{A}} \otimes \underset{K \times L}{\mathbf{B}} \doteq \underset{N*K \times M*L}{\begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \dots & a_{1,m}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \dots & a_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\mathbf{B} & a_{n,2}\mathbf{B} & \dots & a_{n,m}\mathbf{B} \end{pmatrix}}$$

Figure 9: Formula for Kronecker Product

4.3 Kronecker Graph

Using the `Kronecker Product`, we can generate a **Kronecker Graph** with the initiator matrix K_1, which is an adjacency matrix of a graph.

$$K_m = K_1 \bigotimes K_1 \bigotimes K_1 \cdots = K_{m-1} \bigotimes K_1$$

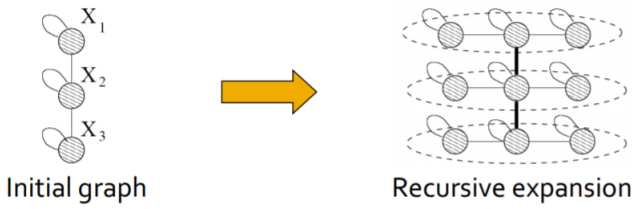


Figure 10: Recursively Generating a Kronecker Graph Visually

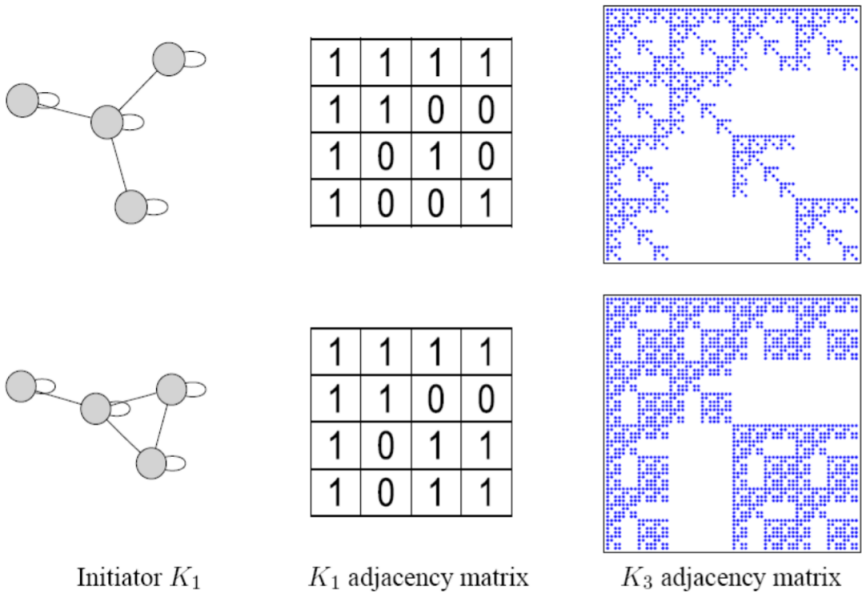


Figure 11: Generating a Kronecker Graph in Terms of the Adjacency Matrices

4.4 Stochastic Kronecker Graph

Instead of a deterministic graph, whereby the edges are defined as 1 (if exists) and 0 (if not), we can have an adjacency matrix with the probability of edge (u, v) existing in the graph.

$$\Theta_1 = \begin{bmatrix} 0.2 & 0.1 \\ 0.6 & 0.4 \end{bmatrix}; \Theta_2 = \Theta_1 \bigotimes \Theta_1 = \begin{bmatrix} 0.04 & 0.02 & 0.02 & 0.01 \\ 0.12 & 0.08 & 0.06 & 0.04 \\ 0.12 & 0.06 & 0.08 & 0.04 \\ 0.36 & 0.24 & 0.24 & 0.16 \end{bmatrix}$$

Efficiency of Algorithm

To instantiate a stochastic Kronecker Graph, the number of operations given n number of nodes is going to be $O(n^2)$.
This would not work for huge graphs.

Efficient Generation of a Stochastic Kronecker Graph

Real graphs are often `sparse`. Hence, we do not have to evaluate all possible edges, which is $\binom{n}{2}$. Instead, we select the number of edges we want and "drop" them. We also divide the matrix into quadrants of the initiator matrix K_1.

Algorithm 1: Effective Generation of a Stochastic Kronecker Graph

Input: Initiator Matrix K_1, degree of growth m, number of edges e
Output: Stochastic Kronecker Graph K_m

1 Initiate $L_{u,v} = \Theta_{uv} / \sum_{op} \Theta_{op}$ // `Normalize Probabilities`
2 **for** $i = 1 \dots e$ **do**
3 **for** $j = 1 \dots m$ **do**
4 Start with $u = 0, v = 0$
5 Pick u, v based on $L_{u,v}$
6 Descend into quadrant (u, v)
7 Add Edge (x, y) into K_m
8 **return** K_m

Note that the inner loop `(lines 3-6)` is a constant (to hone into a specific entry, we have to repeat m coin flips) and so the asymptotic time complexity is $O(e)$.

5 Motifs in Networks

Key Idea

What are the building blocks of the network?
Answer. `subgraphs`. This helps in the **characterization** and **discrimination** of networks.

5.1 Decomposing Structure

Example of Subgraphs

Figure 12 shows all possible configurations of `non-isomorphic` graphs with 3 nodes.
`Graph isomorphism` is loosely defined as two graphs which contain the same number of graph vertices connected in the same way

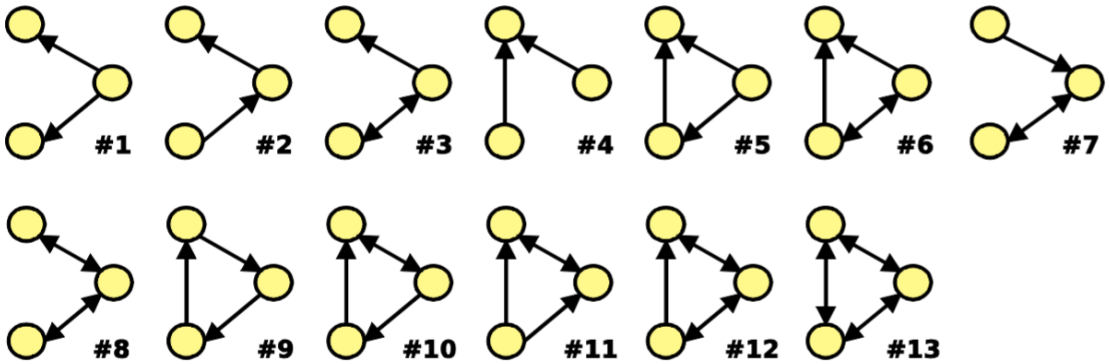


Figure 12: All Possible Jon-isomorphic Directed Graphs with 3 Nodes

Motivation of Analyzing Structures

Being able to characterize such subgraphs can allow us to look for **significance** in the structures of these networks.
This is usually done with a `Network Significance Profile`.

Significance

The significance metric is $\in [-1, 1]$ and it is obtained from measuring subgraph occurence relative to a **baseline null graph**.

1. `Positive Value` $s > 0$ - means that a certain subgraph is **over-represented** relative to the baseline graph
2. `Negative Value` $s < 0$ - means that a certain subgraph is **under-represented** relative to the baseline graph

Figure 13 shows different significant profiles of different networks that characterizes each network.

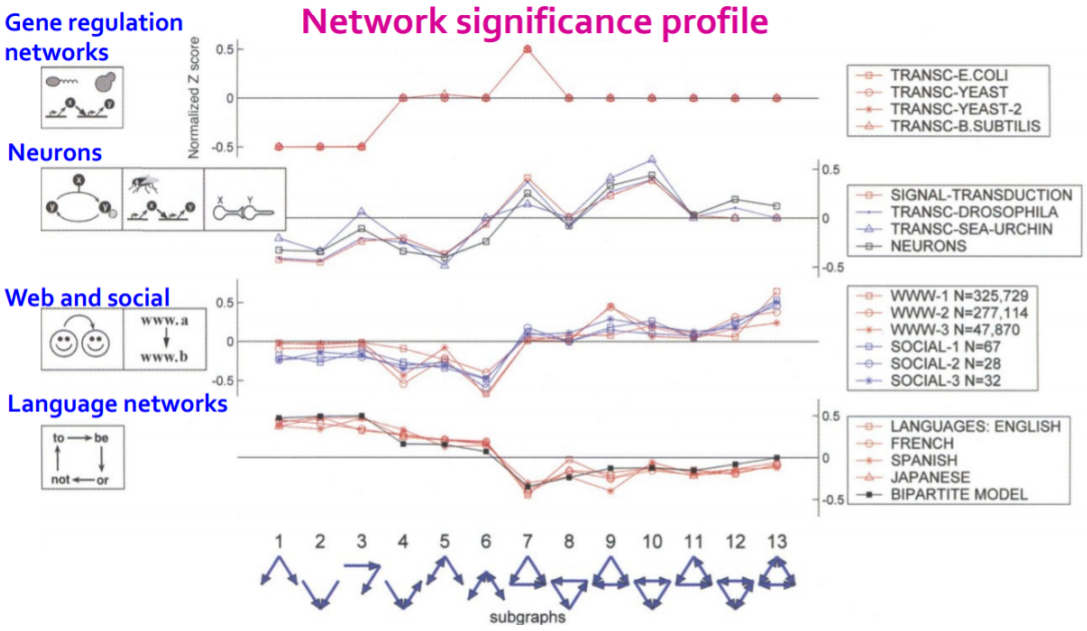


Figure 13: Network Significance Profiles of Different Networks. Notice that Similar Networks tend to have Similar Profiles that Characterizes the Network

5.2 Motifs

Definition of Motifs

`Recurring, significant patterns` of interconnectivities.

1. **Patterns** - `small induced subgraphs` (for given choices of nodes, all edges have to be included)
2. **Recurring** - found many times and **overlapping** is allowed
3. **Significant** - more/less frequent than `expected`

This helps **understand** how the network works and **predict** its operations.

5.3 Significance of a Motif

Key Idea

Subgraphs that occur in a real network much **more often** than in a random network have **functional significance**.

Statistical Significance of a Motif

Effectively a z-score of the **over/under-representation** of a motif relative to a *randomized network*.

$$Z_i = \frac{N_i^{real} - \bar{N_i}^{rand}}{\sqrt{var(N_i^{rand})}}$$

1. Z_i - statistical significance of motif i
2. N_i^{real} - frequency of occurrence of motif i in the real network
3. $\bar{N_i}^{rand}$ - mean of the frequency of motif i in a random network
4. $\sqrt{var(N_i^{rand})}$ - standard deviation of the frequency of motif i in a random network

Network Significance Profile

$$SP_i = \frac{Z_i}{\sqrt{\sum_j (Z_j)^2}}$$

Effectively just a **normalized** z-score bounded by $[-1, 1]$. This provides a measure of `relative significance`.

1. Allows comparison across graphs of different sizes
2. Larger graphs tend to have larger z-scores

5.4 Configuration Model

Significance is a measure relative to a **baseline model**. The properties of such model plays an extremely important role in determining the significance of the subgraphs.

5.4.1 Decision Decision of the Null Model

What are the properties we want the **null model** to match with the real network?
If we wish to only match $|V|$ and $|E|$ - we can simply use the Erdös-Renyi Model to generate it.
Problem: too simplistic

Goal for using the Configuration Model

Since we are interested in the **interconnectivities** of the network, the baseline network should be similar *wrt* to other properties with **randomized** connections.

5.4.2 Method 1: Spokes

1. Generate $\{k_1, k_2, \ldots, k_N\}$ according to the degree distribution of the real network
2. The degree k_i for node i means that the node would have k number of **spokes** (unconnected edges).
3. Partition each node and assign k "mini-nodes" to each node
4. Randomly pair up mini nodes
5. Collapse graph to form randomized configuration model

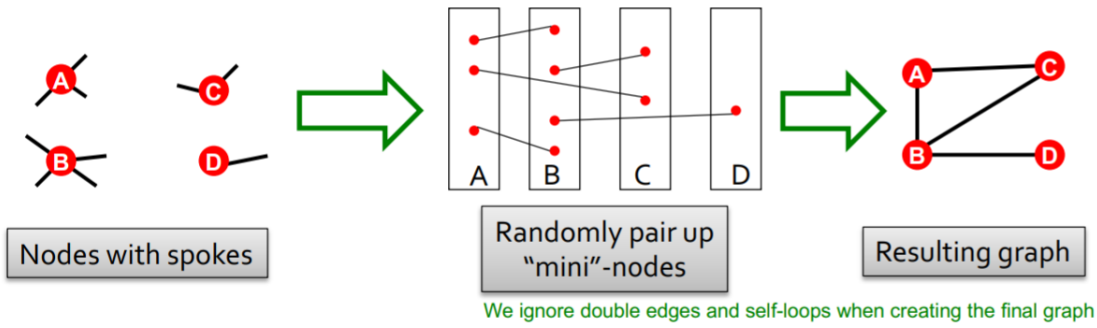


Figure 14: Visual Representation of the Algorithm Above

This algorithm does not guarantee identical degree distribution, though this issue becomes less of a problem as the size of the network grows.

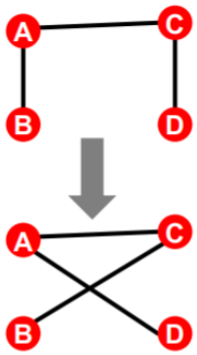


Figure 15: Visual Representation of Switching

5.4.3 Method 2: Switching

1. Start with the real network G
2. Repeat $Q \times |E|$ times, select random pairs of edges (a, b) and (c, d), switch the endpoints into (a, d) and (c, b)

This algorithm preserves `degree distribution` of the real network. However, for the process to converge to a random graph, Q has to be ≈ 100, which means that it is quite computationally expensive.

5.5 Variations of Motifs

Canonical Graph Variations:

1. Directed vs Undirected
2. Coloured vs Uncoloured
3. Temporal vs Static

Conceptual Difference:

1. Different frequency measures
2. Significance measure
3. Anti-motifs (occurrences of complementaries)
4. Different constraints for the null model

6 Graphlets as Node-Level Characteristics

6.1 Key Definitions and Concepts

Difference from Motifs

They are node-level descriptions of graphs unlike motifs which are network-level descriptions.

Definition

Connected non-isomorphic subgraph **rooted at one node**.

Graphlet Degree Vector (GDV)

A vector of counts of the number of graphlets the node touches at a particular **orbit**.

6.2 Graphlets

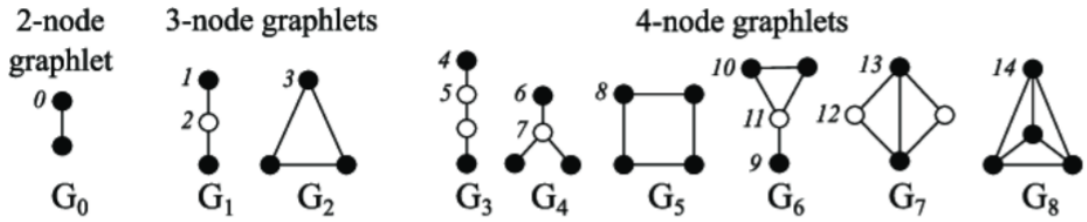


Figure 16: Examples of 2,3 and 4-Node Graphlets

Figure 16 are all the possible graphlets for 2, 3 and 4-node graphlets.

Understanding the Diagrams

Observe that there are two nodes in G_1 such that when rooted at given node would produce different results.

Likewise, for G_2 rooting at any node would still be the same graphlet.

For $n = 3, 4, 5 \ldots 10$, there are correspondingly $2, 6, 21, 11716571$ graphlets!

6.3 Automorphism Orbits

6.3.1 Definition

For a node u of a graph G, the automorphism orbit of u is $Orb(u) = \{v \in V(G); v = f(u)\ for\ some\ f = Aut(G)\}$, whereby $Aut(G)$ denotes an automorphism of group G. `not helpful`

It essentially takes into account the symmetries of the graph.

6.3.2 Example

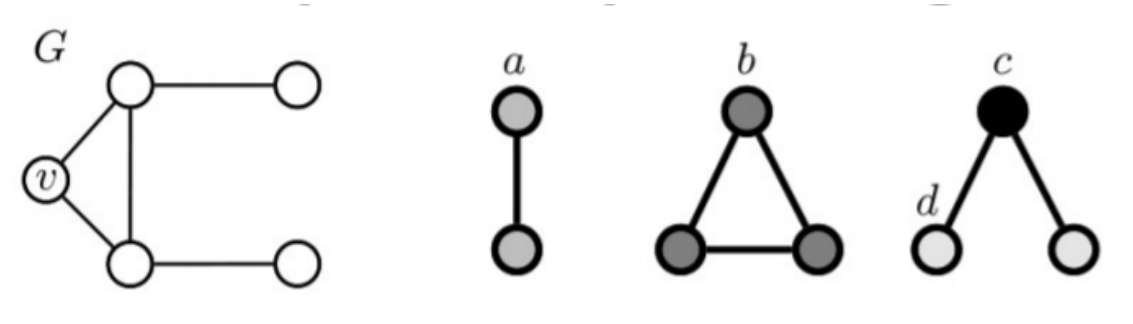


Figure 17: Graph with node v and corresponding graphlets of interests and their orbits

Given the following graph G, and the node of interest is v and graphlets of interest are G_0, G_1 and G_2 in Figure 17.

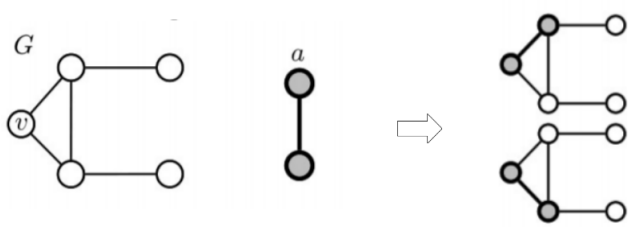


Figure 18: Number of Counts of Graphlet Rooted at a for node v

From Figure 18, we can see that there are **two** counts of nodes for the graphlet.

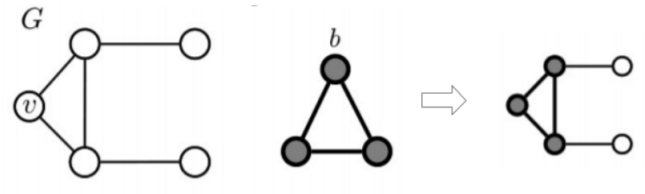


Figure 19: Number of Counts of Graphlet Rooted at b for node v

From Figure 19, we can see that there is **one** counts of nodes for the graphlet.

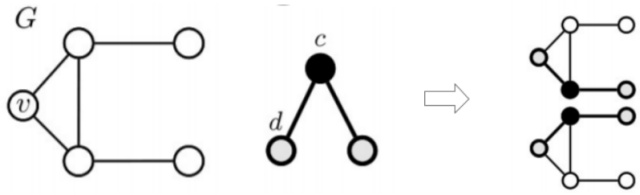


Figure 20: Number of Counts of Graphlet Rooted at d for node v

From Figure 20, we can see that there are **two** counts of nodes for the graphlet rooted at d. Note that there are **no** corresponding graphlet rooted at c for the graph.

From this, we are can obtain the Graphlet Degree Vector for node v (GDV_v) as the following vector

$$GDV_v = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 2 \end{bmatrix}$$

6.3.3 Graphlet Degree Vector

This vector characterizes the **node's local network topology**, which provides a highly constraining measure of topological similarity.
However, computationally speaking, mapping calculating a graphlet degree vector is **expensive**. Hence, typical uses considers

1. Graphlets of 2 to 5 nodes
2. This gives $|GDV_i| = 73$
3. Captures interconnectivity of nodes up to **4 hops**

7 Finding Motifs and Graphlets

Two main steps are required:

1. **Enumeration** - based on subgraphs up to size k
2. **Count** - counting occurrence of each subgraph

This is a known NPC problem.

7.1 Exact Subgraph Enumeration (ESU) Algorithm

Recursive algorithm that recursively introduces candidate nodes to form an isomorphic subgraph that has to meet two conditions:

1. Candidate node u must have a larger id than the included node $v \in V_{subgraph}$
2. Candidate node u can be neighbours of other node w, but not in any node $v \in V_{subgraph}$

Algorithm: EnumerateSubgraphs(G, k) (ESU)
Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: All size-k subgraphs in G.

01 **for** each vertex $v \in V$ **do**
02 $\quad V_{Extension} \leftarrow \{u \in N(\{v\}) : u > v\}$
03 $\quad$ **call** ExtendSubgraph$(\{v\}, V_{Extension}, v)$
04 **return**

ExtendSubgraph$(V_{Subgraph}, V_{Extension}, v)$
E1 **if** $|V_{Subgraph}| = k$ **then output** $G[V_{Subgraph}]$ and **return**
E2 **while** $V_{Extension} \neq \emptyset$ **do**
E3 $\quad$ Remove an arbitrarily chosen vertex w from $V_{Extension}$
E4 $\quad V'_{Extension} \leftarrow V_{Extension} \cup \{u \in N_{excl}(w, V_{Subgraph}) : u > v\}$
E5 $\quad$ **call** ExtendSubgraph$(V_{Subgraph} \cup \{w\}, V'_{Extension}, v)$
E6 **return**

Figure 21: Algorithm for ESU

This forms a recursively-defined tree called an `ESU-Tree` as shown in Figure **??**.

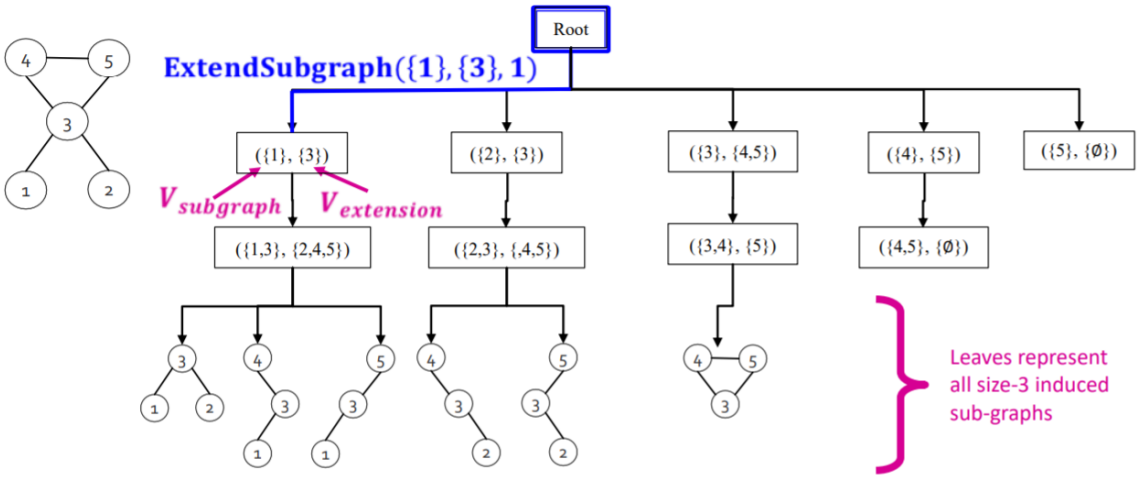


Figure 22: ESU Tree for Given Graph

1. In the first level $k = 1$, $V_{subgraph}$ contains each node and their adjacent nodes such that $u_{id} < v_{id} \in V_{extension}$
2. The steps are repeated until one of two conditions occur
3. Subgraph of size k is obtained
4. or $V_{extension} = \emptyset$
5. The algorithm backtracks and tries other candidates until it exhaustively searches and enumerates all possible subgraphs

7.2 Counting

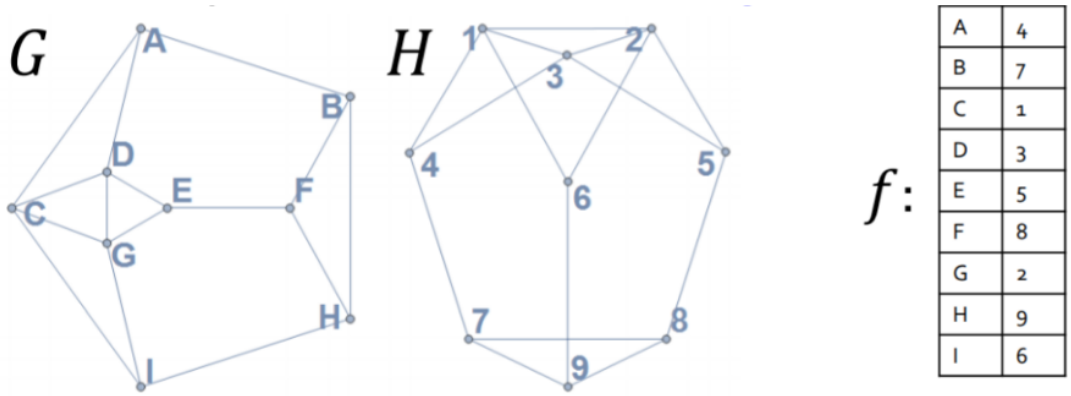
From Figure **??**, we can see quite clearly in terms of non-isomorphic subgraphs, we have two types.

Figure 23: Counts from ESU performed in Figure 22. The one on the left has 5 counts whereas there is 1 on the right

However, identifying such patterns is not an easy problem to solve.

7.2.1 Graph Isomorphism

Formal Definition: Graphs G and H are isomorphic is there exist a bijection $f : G(V) \to H(V)$ such that $u, v \in G(V)$ are adjacent in G if and only if $f(u), f(v)$ are adjacent in H.



A	4
B	7
C	1
D	3
E	5
F	8
G	2
H	9
I	6

Figure 24: Example of isomorphic mapping of Graphs G and H

This is a known NPC problem since we have to evaluate up to $\binom{n}{2}$ possible combinations of bijections, whereby n is the number of nodes.

Considered **canonical**, we use McKay's `NAUTY` Algorithm to evaluate graph isomorphism. It is an **exhaustive** search.

8 Structural Roles in Networks

Measuring structural roles can be done by measuring **structural behaviours**:

1. Center of **Star**
2. Members of **Clique**
3. **Peripheral** (Outlier) Nodes

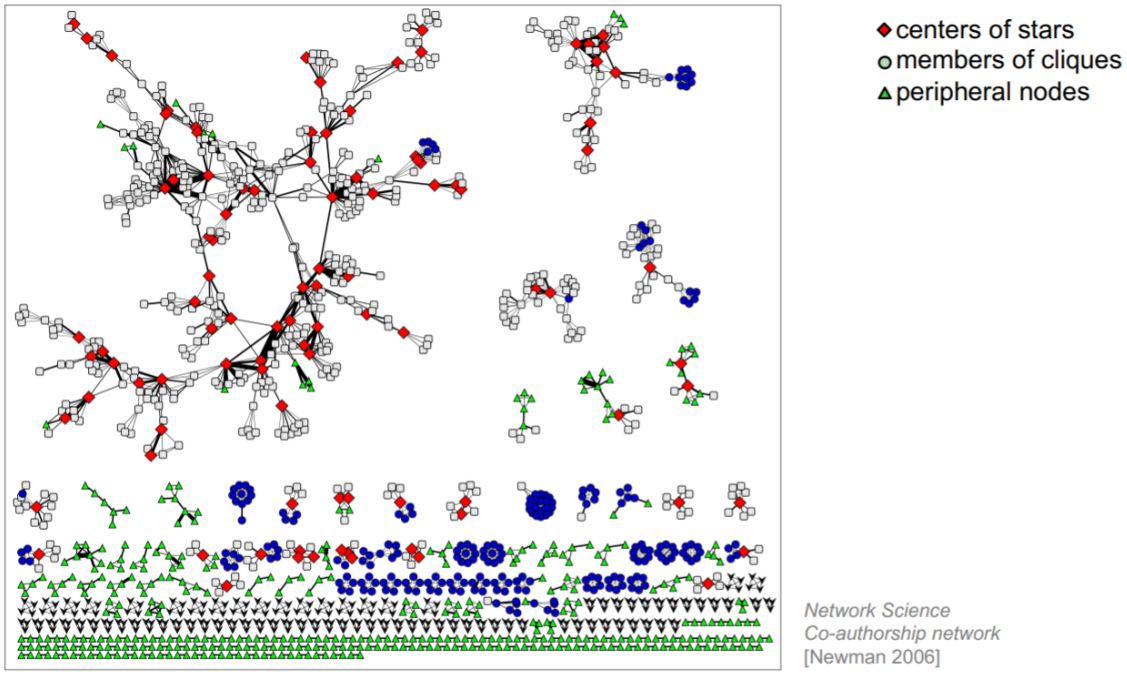


Figure 25: Coauthorship Network and Structural Roles of Different Nodes

8.1 Defining Roles and Groups

Roles and Groups are two similar but distinct concepts.

1. Roles are based on the **similarity of ties** between subset of nodes
2. They are **collection of nodes** with similar positions in the network
3. Adjacency, proximity and reachability is **NOT** considered in roles (they refer to groups/community)
4. Nodes with same roles **DO NOT** need to interact (directly or indirectly) with one another

8.1.1 Example of Roles and Groups

Roles - Students, Faculty and Administration
Groups - College of Business, College of Arts and Sciences etc.

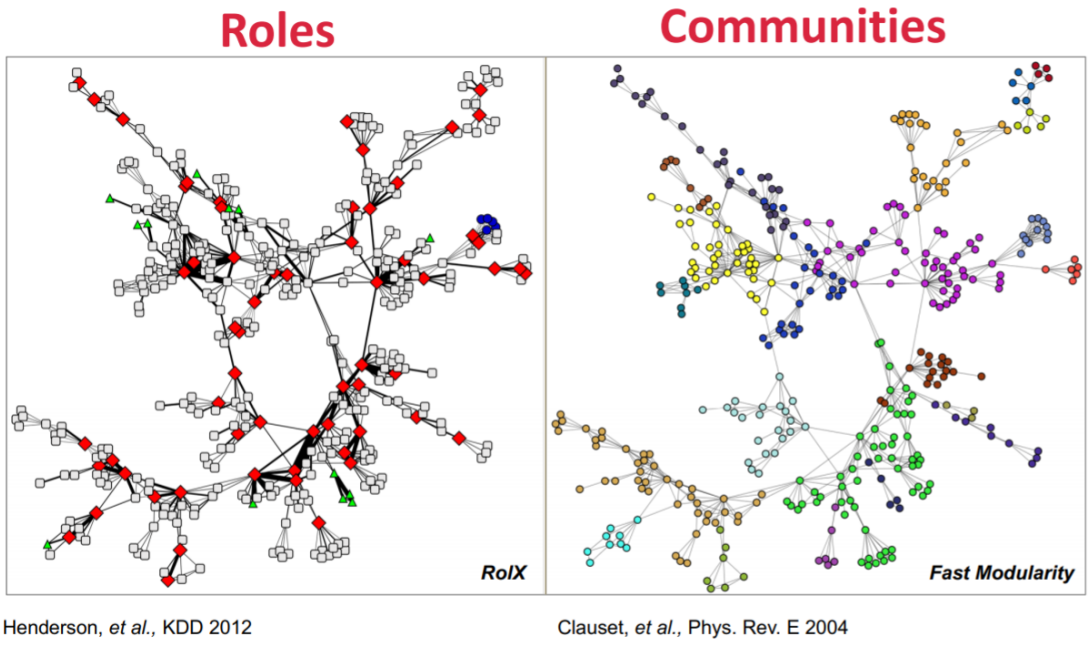


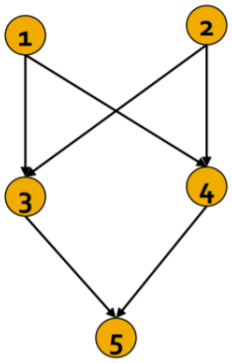
Figure 26: Graph Visualization of Roles vs Communities

8.1.2 Structural Equivalence

Roles are collection of nodes with **structural equivalence**.

Structural Equivalence: Nodes u and v have are structurally equivalent if they have the same relationships with other nodes. u and v is structurally equivalent if they satisfy the following relationship.

$$\forall w \in V, \exists (u, w) \in E \iff \exists (v, w) \in E$$



Adjacency matrix

	1	2	3	4	5
1	-	0	1	1	0
2	0	-	1	1	0
3	0	0	-	0	1
4	0	0	0	-	1
5	0	0	0	0	-

Figure 27: Structural Equivalence in the Graph on the Right and its Adjacency Matrix

Figure 27 shows a graph and the adjacency matrix of the graph. Nodes 3 and 4 are structurally equivalent. 3 has in-edge from 1 and 2, and has an out-edge to 5. This is the same for 4.

8.2 Why Roles?

Task	Explanation
Role Query	Who have similar behaviours based on a known target?
Role Outliers	Indentify unusual behaviours
Role Dynamics	How do these roles function in the network?
Identity Resolution	Uncover anonymous or unknown nodes
Role Transfer	Making predictions of how nodes with the same roles behave
Network Comparison	Calculating network similarity

Table 2: Summary of Usage of Roles

8.3 Discovering Roles using `RolX`

Automatic discovery of nodes' structural roles in networks:

1. Unsupervised
2. No prior knowledge required
3. Assignment by mixed-membership
4. $O(|E|)$

8.3.1 Overview of `RolX`

Recursively extract features and then extract the roles from a given Adjacency Matrix:

1. Given **Adjancency Matrix** $(n \times n)$
2. **Recursive Feature Extraction**
3. Outputs **Feature Matrix** $(n \times f)$
4. **Role Extraction**
5. Outputs **Role Matrix** $(n \times r)$ and **Role-Feature Matrix** $(r \times f)$

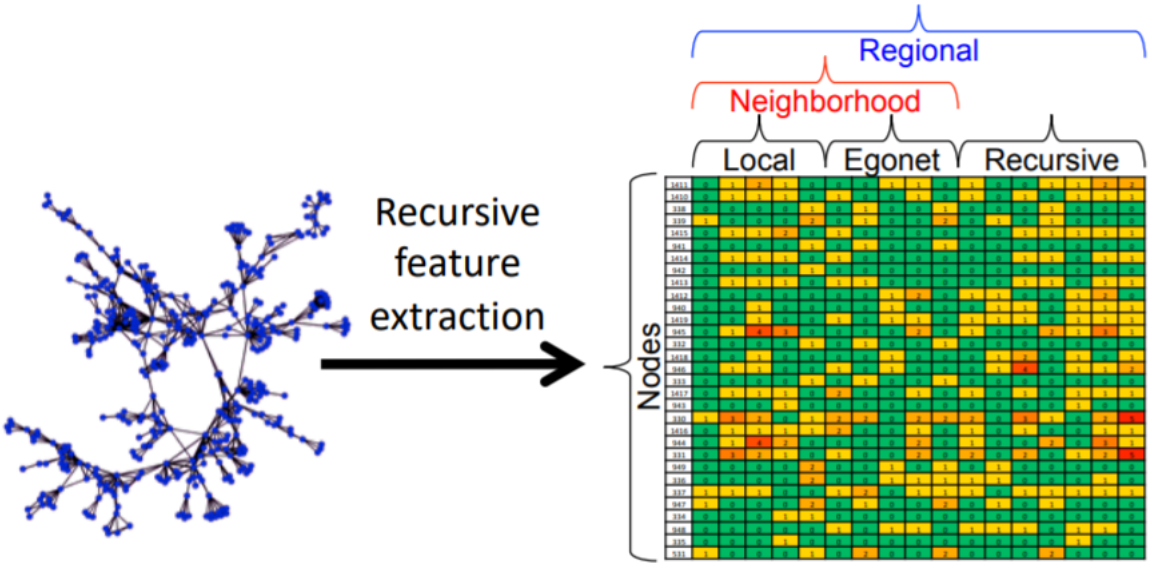


Figure 28: Overview of How `RolX` Recursively Extracts Features

8.3.2 Recursive Feature Extraction

Converts connectivities between nodes into structural features. This is done recursively by first extracting:

1. Local Features - node degree
2. Aggregate Function (mean/sum)
3. Forms Egonet (Adjacent Node) Features - egonet degree
4. Aggregate Function
5. Forms Regional Features - regional degree
6. Repeat

Figure 29: Egonet of Red Node

One of the problems is that features can grow **exponentially**. However, using `pruning technique` based on feature similarity, we can combine features.

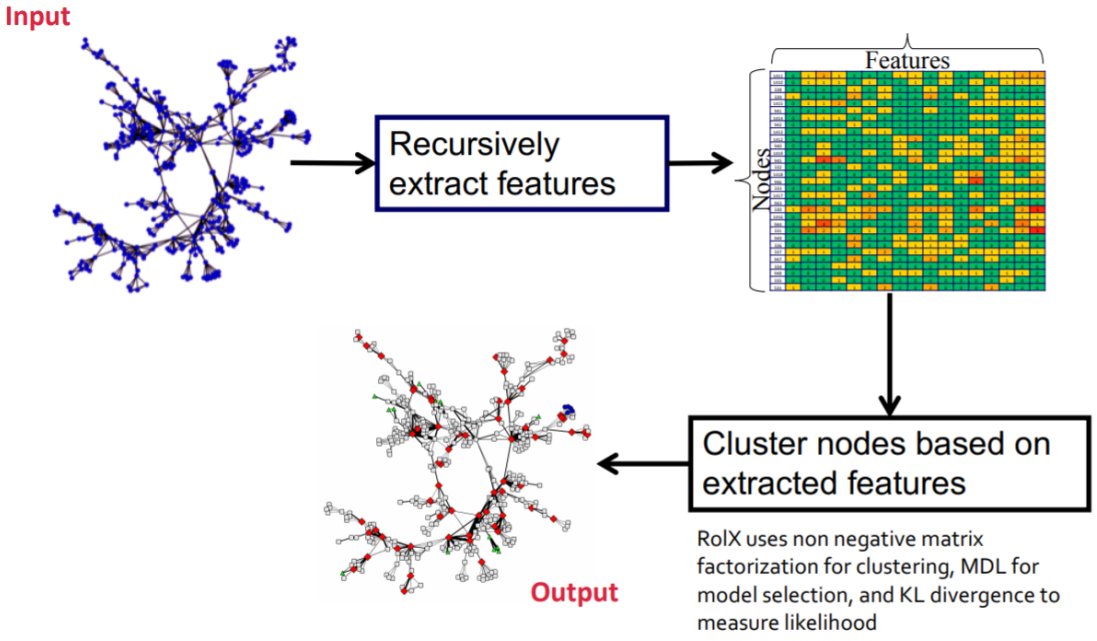


Figure 30: Visual Representation of How `RolX` Works

Based on Figure 30

1. `Non-negative Matrix Factorization` - s a group of algorithms in multivariate analysis and linear algebra where a matrix **V** is factorized into (usually) two matrices **W** and **H**
2. `MDL` - **M**inimum **L**ength **D**escription of the available data as the best model observes the principle identified as *Occam's razor*.
3. `KL-Divergence` - **Kullback–Leibler** divergence (also called **relative entropy**) is a measure of how one probability distribution is different from a second, reference probability distribution.

8.3.3 Making Sense of Roles

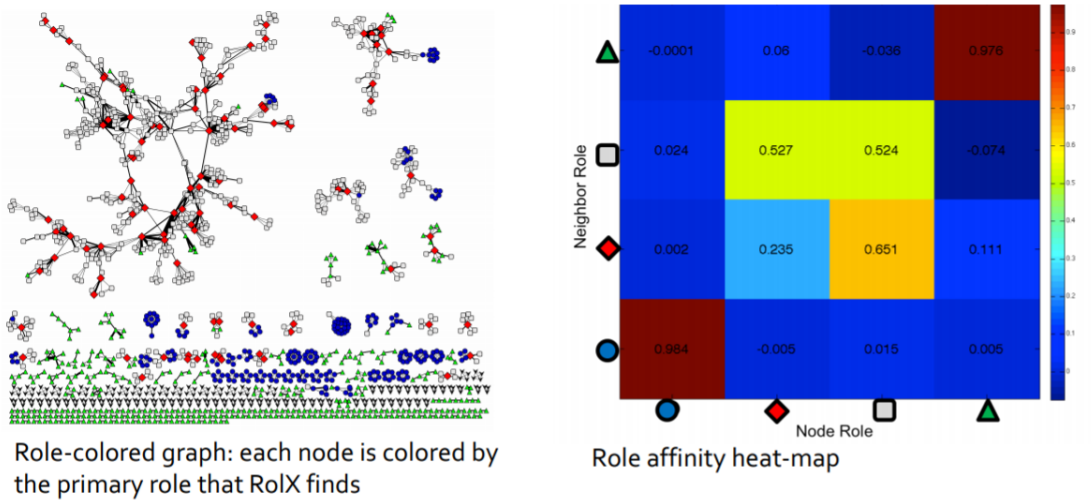


Figure 31: The Coauthorship Network and its Affinity Map produced by `RolX`

After the roles are assigned, the user has to make sense of the roles found:

1. **Blue** - Tightly knit groups of coauthors
2. **Red** - Bridges that connects other nodes
3. **Gray** - "Mainstream", neither tightly knit or clustered
4. **Green** - Elongated groups (not clique but reachable)

9 Network Communities

Having a definition of roles and introducing methods in uncovering them, we know turn to network communities:

1. What is a community conceptually? `Granovetter's Strength of Weak Ties`
2. What makes a community and what are the measurements used to determine a community? `Modularity and Lovain's Algorithm`

9.1 Concept of a Community

A community is a sets of nodes with a lot of internal connections and few external ones (to the rest of the network).

Conceptual Idea of a network community comes from Granovetter's work on **flow of information** about jobs.

Main Question - How do people find out about new jobs? **Acquaintances**, not friends. But why?

9.1.1 Structural vs. Interpersonal Role of an Edge

1. Structural - refers to how closely embedded nodes between edges
2. Interpersonal - refers to how strong/weak these ties are

Observation - Due to the property of **triadic closure**, we observe that structurally close friends tend to have strong interpersonal ties. Likewise, long-edges that connect to distant friends (acquaintances) tend to have weak ties.

9.1.2 Information Flow

Given that, Granovetter argued that weak ties tend to connect a person to new information, unlike close friends. This led to the observation that most people found jobs from acquaintances not friends.

This is validated empirically and now we can define a notion of strength of ties.

9.2 Strength of Ties

Measuring `Edge Overlap`, we can measure its strength of ties, and therefore identifying communities.

$$O_{ij} = \frac{|N_i \cap N_j - \{i, j\}|}{|N_i \cup N_j - \{i, j\}|}$$

1. O_{ij} is the overlap of nodes i and j
2. N_i is the neighbours of i
3. **Numerator** is the shared neighbours of nodes i and j
4. **Denominator** is the total neighbours of nodes i and j

When $O_{ij} = 0$, it is known as a **bridge**.

9.3 Finding Network Communities

On top of overlap, we need to be able to measure how well communities are partitioned into them. This leads to the idea of **modularity** Q.

$$Q \propto \sum_{s \in S} \frac{|E_s|}{\mathbf{E}[|E_s|]}$$

1. s is a partition
2. S is the set of partitions
3. E_s is set of edges in s
4. $\mathbf{E}[|E_s|]$ is the *expected* number of edges in s

To evaluate $\mathbf{E}[|E_s|]$, we have to have `null model` as a reference against the actual network.

9.3.1 Defining the Null Model

Similar to the null model in defining motifs, we use the idea of spokes/endpoints in our model. However, we extend this to **multigraphs** (allows multiple edges between a pair of nodes).

$$\mathbf{E}[|E_{i,j}|] = k_i \times \frac{k_j}{2m}, \sum_{u \in V} k_u = 2m$$

1. $E_{i,j}$ the edges between i and j
2. k_i, the spokes at node i
3. m is the number of edges
4. $|k| = 2m$, since each edge has 2 end points
5. $\frac{k_j}{2m}$ can be interpreted loosely as the probability that one of k_i chooses k_j

9.3.2 Modularity Measure

The Modularity of partitioning the Graph G into the set of partition S is defined as:

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

1. $\frac{1}{2m}$ is the normalizing factor
2. A_{ij} is an indicator function such that it is 1 if $(i,j) \in E$ and 0 otherwise
3. $Q \in [-1, 1]$ based on the above definition

As Q approach to 1, that means that there is **significant community structure**, but typical range spans around 0.3 to 0.7. **Negative values** suggests anti-community (enemies/exclusion).

Equivalently,

$$Q(G,S) = \frac{1}{2m} \sum ij[A(i,j) - \frac{k_i k_j}{2m}]\mathbf{I}\{\mathbf{c_i}, \mathbf{c_j}\}$$

1. c_i community of node i
2. $\mathbf{I}\{\mathbf{c_i}, \mathbf{c_j}\}$ is an indicator function such that it is 1 if $c_i = c_j$, 0 otherwise

9.4 Louvain's Algorithm

Greedy Algorithm for community detection:

1. $O(n \log n)$
2. Supports Weighted Graphs
3. Provides Hierarchical Clustering
4. Non-overlapping Clusters

This is widely used due to its **rapid convergence** and **high modularity** output.

9.4.1 At a High Level

As a greedy algortihm that takes two phases in each iteration:

1. `Phase 1: Partitioning`
   Local changes to membership assignment
2. `Phase 2: Aggregation/Restructuring`
   Aggregating nodes assigned to the same cluster into "supernodes"

9.4.2 Partitioning

The general idea for the first phase is as follows

1. Initiate each node to a distinct community
2. Compute Modularity Gain (ΔQ) by assigning node i to some neighbourhood $c_j, \forall c \in C$
3. Assign i to c_k such that it maximizes ΔQ
4. Repeat for all nodes until movements yield no gains

Note that as a result, order of processing the nodes can affect the results obtained. However, empirically, this has been shown to be not a significant influence overall.

Modularity Gain of assining i into community c is calculated as the following:

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

1. $\sum_{in}$ is the sum of all links of edges in C within community C
2. $\sum_{tot}$ is the sum of all links of edges in C with all other nodes
3. m is the number of edges in the graph
4. k_i is the number of links of node i
5. $k_{i,in}$ is the number of links of node i with nodes in C
6. The left part of RHS is the modularity after placing i into C
7. The right part of RHS is the modularity before placing i into C

Assinging i into C also means taking i out from community D, hence the formula for Modularity Gain is generalized as follows

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

9.4.3 Restructuring/Aggregation

Communities obtained in the first phase is then aggregated into a supernode:

1. Two supernodes are connected if at least one node in the supernode is connected to the another node in the other supernode
2. The weight of the connection is the sum of all connected weights between the supernodes

This is then fed into phase 1 and repeated until convergence (which stops at 2 communities).

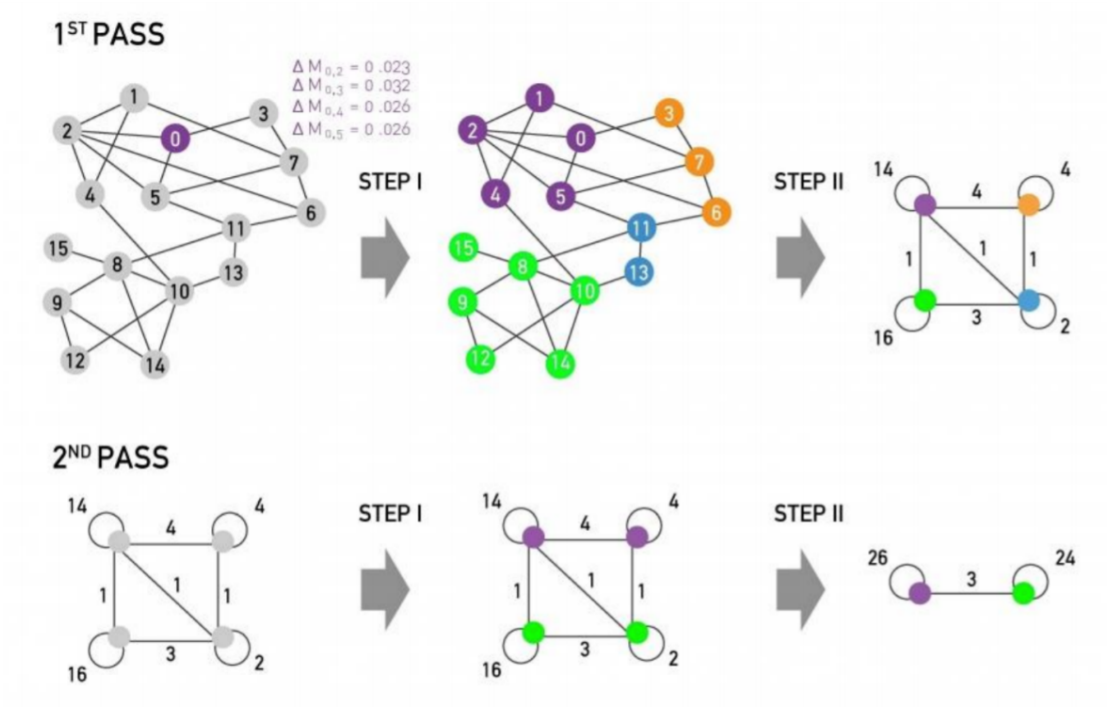


Figure 32: Visualization of How Louvain's Algorithm Work

10 Overlapping Communities

For most networks, assuming a strict assigning of communities is unrealistic. Hence, we need a model that can account for overlapping communities.

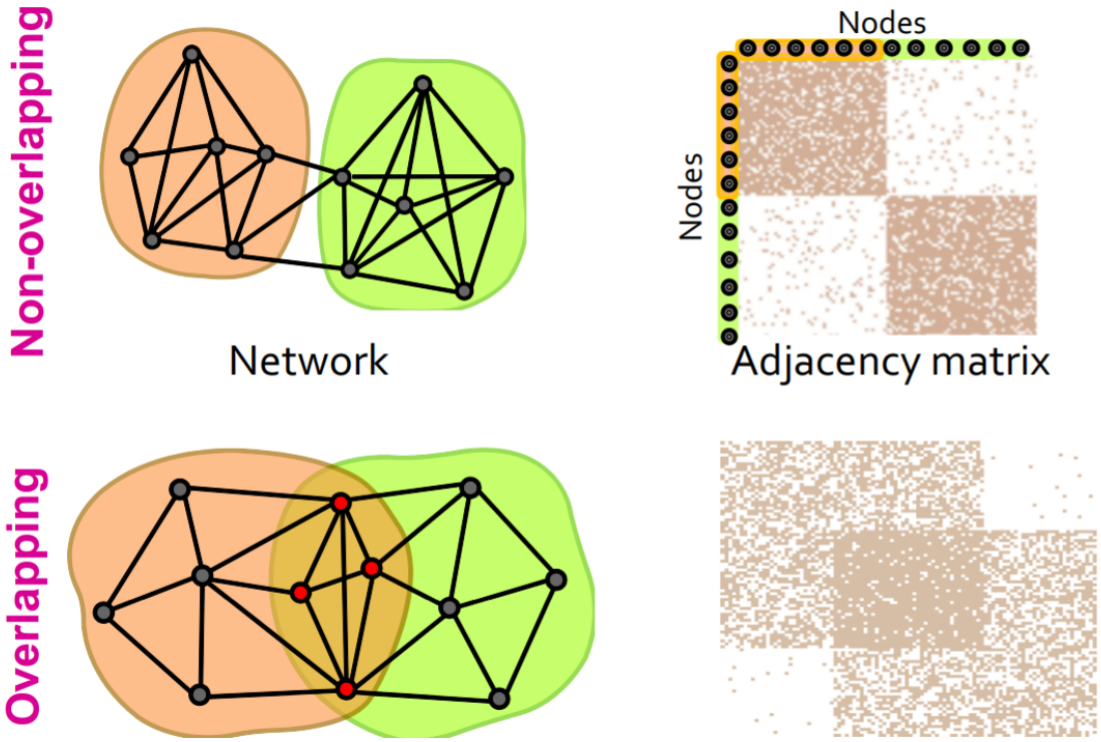


Figure 33: Visual Representation of Overlapping vs Non-Overlapping Communities and their Corresponding Adjacency Matrices

10.1 Community Affiliation Graph Model (AGM)

Generative Model - How is a network generated based on community affiliations?
The parameters of this algorithm are:

1. Nodes V
2. Communities C
3. Memberships M
4. $\forall c \in C$, p_c is the *single* probability that nodes are assigned to the community

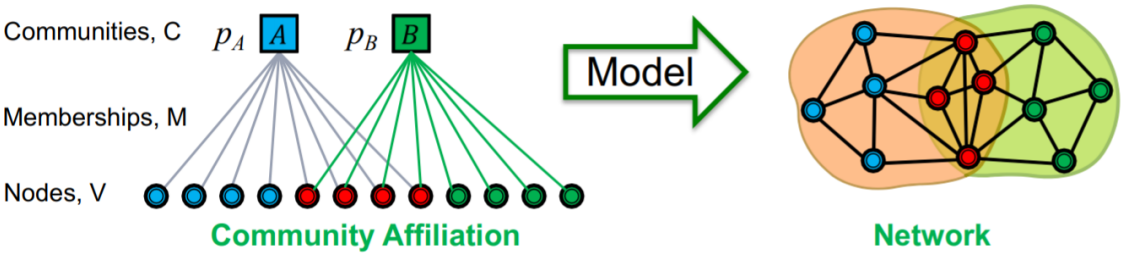


Figure 34: The Generative Process of AGM

10.1.1 Assigning Edges by Probability

Given the parameters of the model $F(V, M, C, \{p_c\})$, we generate the graph G by assigning edges to nodes u and v based on the probability which is defined as the following:

$$p(u, v) = 1 - \prod_{c \in M_u \cap M_v} (1 - p_c)$$

1. Note that the model assumes **independent** assignment of nodes
2. Each community essentially is an *Erdös-Renyi Graph*
3. If u and v shared one membership $|c| = 1$, then $p(u, v) = 1 - 1 + p_c = p_c$
4. Since $p_c \in [0, 1]$, that means that as $|c|$ *increases*, the term $\prod_{c \in M_u \cap M_v} (1 - p_c)$ *monotonically decreases*.
5. Corollary to (4), then $p(u, v)$ *monotonically increases* as $|c|$ *increases*
6. If $c = \emptyset$, then $p(u, v) = 0$, but in practice a small baseline probability ϵ is assigned to each node pairs

10.1.2 Flexibility of AGMs

AGMs are popular models due to its flexibility in representing different affiliation groups. The parameters can be tuned such that it can represent

1. Overlapping Communities
2. Non-overlapping Communities
3. Nested Communities

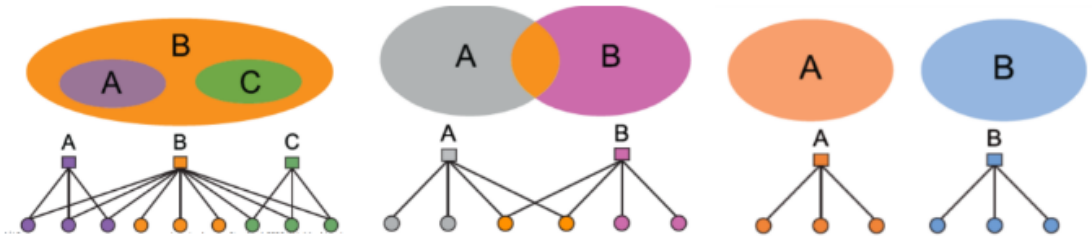


Figure 35: From left to right: Nested, Overlapping, Non-overlapping Community Representations in AGMs

10.2 Detecting Communities Using AGMs

Given a model F, we can generate a desired network G.

However, what we wish to do is to fit G (the actual observed network) to a model F such that it **maximizes the likelihood** that G is generated from the model F. This way we can use the parameters to detect communities.

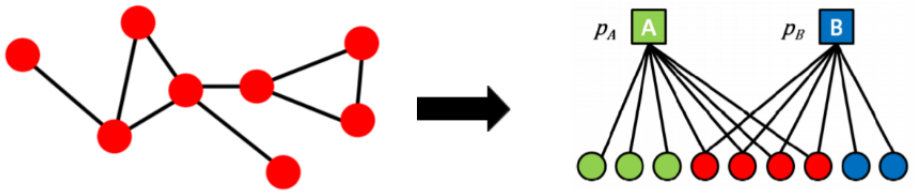


Figure 36: The Task at Hand: Fitting the Graph into an AGM Model

10.2.1 Maximum Likelihood Estimation

In order to fit the graph G, we need an efficient way to estimate the parameters of F. We can more formally define the problem as a maximization problem with the following objective function

$$\underset{\{V,M,C,\{p_c\}\}}{arg\,max} \; P(G|F)$$

10.2.2 Graph Likelihood $P(G|F)$ with Bernoulli Distributions

In the most simple form of Graph Likelihood $P(G|F)$, we can define an independently distributed set of edges, each with a specific Bernoulli distribution with the parameter $p(u, v)$

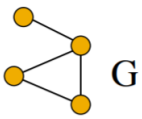


Figure 37: G is an example graph that we attempt to fit in a simple model F

Based on Figure 37, we have the adjacency matrix $G = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

Assuming we have a "guess" model $F = \begin{bmatrix} 0.2 & 0.15 & 0.7 & 0.2 \\ 0.3 & 0.21 & 0.75 & 0.12 \\ 0.4 & 0.2 & 0.1 & 0.14 \\ 0.3 & 0.21 & 0.75 & 0.12 \end{bmatrix}$, where $F_{uv} = p(u, v)$

Having both F and G, we calculate the probability that G is generated by F, $P(G|F)$, using

$$P(G|F) = \prod_{(u,v)\in G} p(u, v) \prod_{(u,v)\notin G} (1 - p(u, v))$$

1. If $(u, v) \in G$, then if $p(u, v)$ is higher, $P(G|F)$ would be higher
2. If $(u, v) \notin G$, then if $p(u, v)$ is lower, $1 - p(u, v)$ would be higher and hence $P(G|F)$ would be higher
3. We want $p(u, v)$ to be high if $(u, v) \in G$ and otherwise if $(u, v) \notin G$

10.2.3 Modelling Membership Strength

Now that we have a simple model F based on independent Bernoulli Distributions, we can complicate things by adding `Membership Strength` as a factor influencing the probability of edge $p(u, v)$.

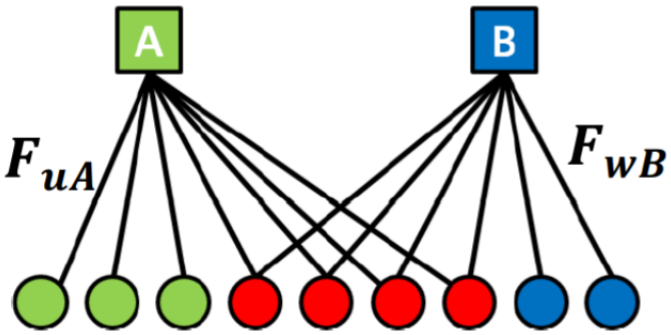


Figure 38: Membership Assignments

Based on Figure 38, we define two concepts

1. F_{uA} - the membership strength of node u to community A
2. For **green** nodes, $F_{green,A} > 0$ and $F_{green,B} = 0$, where 0 means no membership.
3. For **red** nodes, since they have multiple membership $F_{green,A} > 0$ and $F_{green,B} > 0$
4. F_u - a vector of community memberships of u, or $F_u = \begin{bmatrix} F_{uA} \\ F_{uB} \end{bmatrix}$

10.2.4 Graph Likelihood $P(G|F)$ of AGMs

Redefining the probability to take into account membership strengths, we have

$$p(u, v) = 1 - e^{-(F_u \cdot F_v)}$$

1. $F_u \cdot F_v$ is the dot product between F_u and F_v, this provides a measure of similarity in membership
2. The *higher* the similarity in membership, the evaluation of $e^{-(F_u \cdot F_v)}$ is *lower* (due to the negative exponent)
3. Hence, this means that $p(u, v)$ is higher
4. This models the relationship that if there are more shared memberships for a pair of nodes u and v, then the probability that there is an edge $p(u, v)$ increases

We now can define $L(G|F)$, the log-likelihood function of graph G given F based on our previous derivation of $P(G|F)$ by substituting the newly derived probability

$$L(G|F) = \prod_{(u,v)\in G} \log(1 - e^{-(F_u \cdot F_v)}) \prod_{(u,v)\notin G} -(F_u \cdot F_v)$$

10.2.5 Gradient Ascent for Optimization

Now that we have defined the log-likelihood of G given F, we can rephrase the optimization problem as follows:

$$\underset{F}{arg\,max} L(G|F)$$

Optimization of this objective function can be achieved with `Gradient Ascent`:

1. Start with random F
2. Update F_uC for node u while fixing the membership of all other nodes
3. Repeat until convergence

11 Spectral Clustering

Spectral Clustering is based on `spectral graph theory`, which analyzes the "spectrum" (the `eigenvalues and eigenvectors`) of the graph G.

It usually takes the following steps:

1. **Preprocessing** - creating a `Laplacian` representation of the Graph G
2. Decomposition - calculating the eigenvalues and eigenvectors
3. Grouping - Assign to clusters

11.1 Defining the Problem

Clustering a graph G involves partitioning the graph into distinct subgraphs. A good partition will

1. **Maximize** *intracluster* connections
2. **Minimize** *intracluster* connections

11.1.1 Naive Cut

Expressing the task as an *objective function* allows a more definite way to solve the problem. A naive approach would be to define a cut (`or an edge-cut to be specific`) such that

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

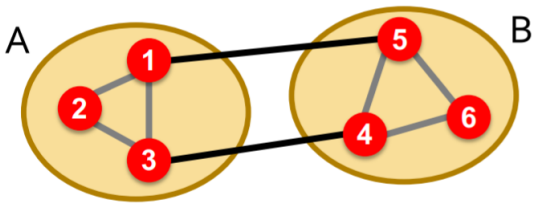


Figure 39: $cut(A, B) = 2$ since there are two edges between the partitions

Intuition would suggest that minimizing the following objective function would allow us to obtain the best partition. However, this could result to a **degenerate** case as in Figure 40

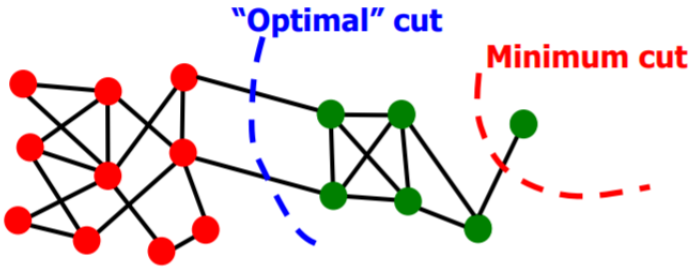


Figure 40: The degenerate case when minimizing the objective funtion wrt to the $cut(A, B)$

This is because it fails to consider **intracluster connectivity**.

11.1.2 Conductance as a Criterion `Shi-Malik, (1997)`

Conductance measures the connectivity between groups relative to the density of the groups.

$$\phi(A, B) = \frac{cut(A, B)}{min(vol_A, vol_B)}$$

1. $vol_A = \sum_{i \in A} k_i$, which is the total number of endpoints in A
2. This includes the consideration of interconnectivity within clusters to produce more balanced graphs
3. However, this is NP-Hard

Is there an efficient way to estimate the solution?

11.2 Eigenvalues and Eigenvectors in Spectral Graph Partitioning

Spectral graph partitioning helps an efficient bounded estimation for minimizing `conductance`, using the ideas of eigenvalues and eigenvectors of the Graph G.

11.2.1 Calculating Neighbours

Let A be the adjacency matrix of G, such that $A_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise.

$\vec{x}$ is a vector $\mathbb{R}^n$ such that $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, where elements $x_i \in \vec{x}$ is a value or label of each node in G.

$$A \cdot \vec{x} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$y_i = \sum_{j=1}^{n} A_{ij} x_j = \sum_{(i,j) \in E} x_j$

This can be interpreted as the number of neighbours x_i has.

11.2.2 *d-Regular* Graph

A `d-regular graph` is that each node has d degree and is **connected**. An obvious *eigenvalue/eigenvector* pair for a `d-regular graph` is

$$A\vec{x} = \lambda\vec{x}$$

Solution $\vec{x} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \lambda = d$

In fact, this d is the **largest** eigenvalue of A.

Proof. Assume that d is not the largest eigenvalue and $\exists d', \vec{x}'$ pair such that $d < d'$.

1. $\lambda = d$ when $\vec{x} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, A\vec{x} = \vec{y} = \begin{bmatrix} \lambda \\ \lambda \\ \vdots \\ \lambda \end{bmatrix}$
2. Let S be the set that contains nodes i that contains the **maximum** value of x_i
3. Since, $x_1 = x_2 = x_3 \cdots = x_n = 1$ when $\lambda = d$, then $S = \{\forall i \in V\}$ with $\lambda = d$

4. With $\vec{x} \neq \vec{x}'$, then $\exists i' \in \vec{x}' \neq 1$, and therefore $|S'| < |S|$ since some nodes i' are not in S
5. Hence, for node $j' \in S'$ and a neighbour $i' \notin S'$, then $j' < d$
6. $\nexists d'$ such that $d' > d$
7. Therefore, d is the largest eigenvalue for G

□

11.2.3 Disconnected d-regular Graph

Suppose that we have a disconnected graph with two components, each are $d - regular$

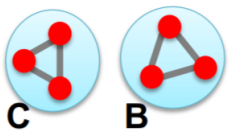


Figure 41: Disconnected Components B and C

Two potential *eigenvectors/eigenvalue* pairs would be to assign $b_i \in B = 1$ and $b_i \in C = 0$ for $\vec{b}$. Likewise, another $c_i \in B = 0$ and $c_i \in C = 1$ for $\vec{c}$.

$$\vec{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, A\vec{b} = \begin{bmatrix} d \\ d \\ \vdots \\ 0 \\ 0 \end{bmatrix} = d \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \lambda = d$$

$$\vec{c} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, A\vec{c} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ d \\ d \end{bmatrix} = d \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \lambda = d$$

For a strictly disconnected component, the largest eigenvalue $\lambda_n = d$ has a multiplicity of 2 (single eigenvalue with two eigenvectors). This gives the following intuition in Figure 42.

Figure 42: For the disconnected graph on the left, the largest eigenvalue would be equal to the second largest eigenvalue. For weakly inter-connected components on the right, the second largest eigenvalue λ_{n-1} would be $\approx \lambda_n$

11.2.4 Intuition on Finding Eigenvalue/Eigenvector Pairs

Eigenvectors are `orthogonal`. This property allows us to "label" nodes into separate components.

1. For $\lambda_n, \vec{x}_n$, all entries of the vector is 1 ($\forall x_i \in \vec{x}_n = 1$)
2. Given `orthogonality`, then the eigenvector $\vec{x}_{n-1}$ would have $\sum_{i=1}^{n} x_i = 0$
3. This is because $\vec{x} \cdot \vec{x}_{n-1} = 0$ (definition of orthogonality)
4. This leads to $\vec{x}_{n-1}[i] > 0$ and $\vec{x}_{n-1}[j] > 0$
5. We can then assign B and C based on their signs

However, what *eigenvalue/eigenvector* pairs of what matrix are we supposed to find?

11.3 Matrix Representations

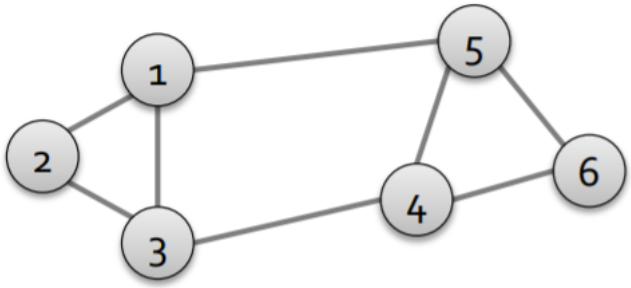


Figure 43: Graph for Various Matrix Representation

Referring to Figure 43, we can have an adjacency matrix $A = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$

Important properties of A

1. **Symmetric**
2. n real **Eigenvalues**
3. **Eigenvectors** are real and orthogonal

The degree matrix is defined D for the graph as $D = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$

Important properties of D

1. Diagonal matrix of dimension $n \times n$
2. Each d_{ii} is the degree for nodes i

A Laplacian Matrix L can be defined as $L = D - A = \begin{bmatrix} 3 & -1 & -1 & 0 & -1 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{bmatrix}$

11.3.1 Properties of the Laplacian Matrix L

L has the following properties

1. The trivial eigenpair for L is $\vec{x} = (1, 1, 1, \ldots, 1)$ and $\lambda_1 = 0$ since $L \cdot \vec{x} = \vec{0}$
2. $\lambda \geq 0$ for all eigenvalues
3. L is symmetric
4. (2) and (3) makes L a positive semidefinite matrix (`Gramian Matrix`)
5. (4) means that L can be written as $L = N^\top \cdot N$
6. (5) can be used to prove that $x^\top Lx = \sum_{ij} L_{ij} x_i x_j \geq 0, \forall x$
7. (2), (5) and (6) are equivalent statements

Proof. of property (6)

$$x^\top Lx = x^\top N^\top Nx = (xN)^\top (xN) \geq 0$$

It is the square of the length of Nx □

Proof. of property (2)
Let λ be an eigenvalue of L, then by (6)

$$x^L x = x^\top \lambda x = \lambda x^\top x \geq 0 \rightarrow \lambda \geq 0$$

□

Proof. of property (5)
TODO □

11.4 Solving λ_2 as a Minimization Problem

Reframing the problem of solving for λ_2 as a minimization problem, we have the following objective:

$$\lambda_2 = \min_{x : x^\top w_1 = 0} \frac{x^\top Mx}{x^\top x}$$

Proof. TODO (and if I can figure it out) □

11.4.1 Meaning of $x^\top Lx$ on Graph G

Rewriting $x^\top Lx$ helps gain the intuition behind minimizing the objective function

$$\begin{aligned}
x^\top Lx &= \sum_{i,j}^{n} L_{i,j}x_i x_j \\
&= \sum_{i,j}^{n} (D_{i,j} - A_{i,j})x_i x_j \\
&= \sum_{i,j}^{n} D_{i,j}x_i x_j - \sum_{i,j}^{n} A_{i,j}x_i x_j \\
&= \sum_{i}^{n} D_{i,i}x_i^2 - \sum_{(i,j)\in E} 2x_i x_j \\
&= \sum_{(i,j)\in E} x_i^2 + x_j^2 - 2x_i x_j \\
&= \sum_{(i,j)\in E} (x_i - x_j)^2
\end{aligned}$$

Since $x^\top x$ is the dot product of x by itself, and since x is unit vector, then $\sum_i x_i^2 = 1$we can effectively write the minimization problem as follows

$$\lambda_2 = \min_{x:x^\top w_1 = 0} \sum_{(i,j)\in E} (x_i - x_j)^2$$

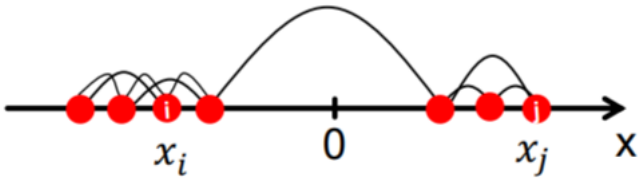


Figure 44: Minimizing $(x_i - x_j)^2$

Since $\vec{w}_2$ must be orthogonal to $\vec{w}_1 = (1, 1, 1, \ldots, 1)$ (eigenvector property), then for all entries of $\vec{w}_2$ would result to $\sum_i x_i = 0$. Figure 44 is an example of the distribution of $x \in \vec{w}$.

Minimizing the objective function would require that the choice of $\vec{w}_2$ would have to have values of x to nodes i such that few nodes as possible cross 0.

11.4.2 Fiedler's Optimal Cut (1973)

In a more simple example, the partition is expressed into (A, B) such that

$$y_i = \begin{cases} +1 & \text{if } i \in A, \\ -1 & \text{if } i \in B. \end{cases}$$

This effectively would give us the following Figure 45

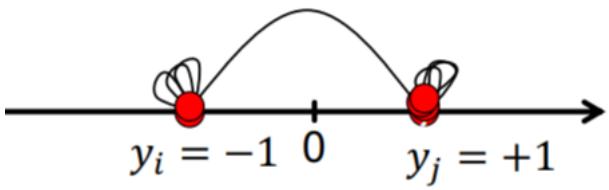


Figure 45: Feidler's Optimal Cut

With the constraint such that $|A| = |B| \rightarrow \sum_i y_i = 0$, then we can show that the optimal cut that minimizes the number of edges cut is

$$\min_{\vec{y} \in \{+1,-1\}^n} \sum_{(i,j) \in E} (y_i - y_j)^2$$

11.4.3 Rayleigh Theorem

Allowing $y_i \in \mathbb{R}$, we can extend `Feidler's Optimal Cut` with constraint such that $\sum_i y_i = 0$ and $\sum_i y_i^2 = 1$

$$\min_{\vec{y} \in \mathbb{R}^n} \sum_{(i,j) \in E} f(y) = (y_i - y_j)^2 = y^\top L y$$

1. λ_2 is obtain from minimizing the objective function above wrt the Laplacian Matrix L
2. $\vec{y}$ is known as the `Feidler vector`, which is the eigenvector that corresponds to the second eigenvalue λ_2
3. We can use the sign of y_i to cluster the graph

11.4.4 Approximation Guarantee of Spectral Clustering

Spectral clustering can provide an approximation guarantee that the it is at worst twice the optimal (minimum) conductance β if Graph G is partitioned into A and B.

$$\lambda_2 \leq 2\beta$$

Proof. TODO (if I can figure it out) □

11.5 Spectral Clustering Algorithm

As mentioned early in this section, spectral clustering algorithm takes three main steps. They are

1. **Preprocessing** - Obtain the Laplacian Matrix L from the adjacency matrix A
2. **Decomposition** - Find the eigenvalue/eigenvector pairs $(\lambda, \vec{x})$ of L and map nodes to values of x_2
3. **Grouping** - Sort components based on reduced 1-dimensional vector by splitting the vector

11.5.1 Finding the Split

The choice of split is largely based on the discretion of the user

1. A *naive* approach could be based on signs (+/-) or some other criteria
2. **Sweeping** - a more *expensive* approach that "sweeps" over nodes (dividing into two groups) and evaluate the conductance to identify the best cluster

11.5.2 Examples of Spectral Clustering

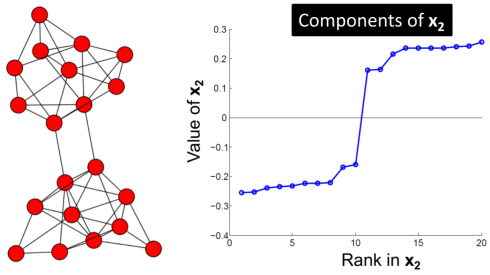


Figure 46: Spectral Clustering on the Graph on the left

Figure **??** is an example of applying spectral clustering on the graph. The values of $\vec{x}_2$ is ranked and they are divided according to the components that we can visually identify.

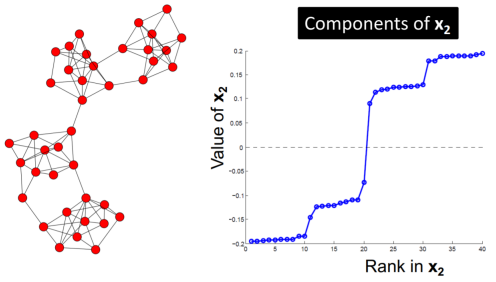


Figure 47: Spectral Clustering on the Graph on the Left with 4 Clusters

For a 4-way Spectral Clustering, we can see in Figure **??** that by ranking values in $\vec{x}_2$, we get a separation of the nodes based on the values in $\vec{x}_2$.

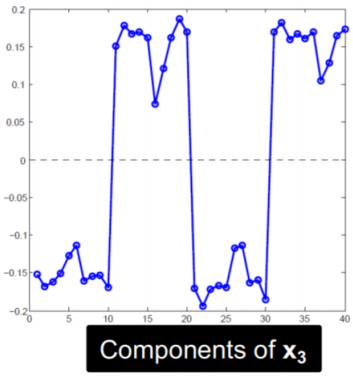


Figure 48: Using $\vec{x}_3$ by Means of Clustering

An alternative approach is to use $\vec{x}_3$ instead. This produces a similar result to the previous graph as well.

11.5.3 k-way Spectral Clustering

Two basic approaches, as explored in the previous subsection in performing k-way spectral clustering are as follows

1. **Recursive Bipartitioning** - Recursively perform spectral clustering on partitioned graph. A `less preferred` approach due to `numerical instability` and it is `inefficient`.
2. **Cluster Multiple Eigenvectors**

11.5.4 Clustering Multiple Eigenvectors

This approach takes the following steps

1. Build a reduced space from multiple eigenvectors by assigning each node with k numbers
2. Apply *k-means* clustering based on their k dimensions as features

$$\begin{matrix} v_1 & v_2 & \dots & v_n \\ \begin{pmatrix} \vec{x}_1^1 & \vec{x}_1^2 & \dots & \vec{x}_1^n \\ \vec{x}_2^1 & \vec{x}_2^2 & \dots & \vec{x}_2^n \\ \vdots & \vdots & \vdots & \vdots \\ \vec{x}_k^1 & \vec{x}_k^2 & \dots & \vec{x}_k^n \end{pmatrix} & \begin{matrix} \lambda_1 \\ \lambda_2 \\ \\ \lambda_k \end{matrix} \end{matrix}$$

The first step is to construct the matrix above based on the number of clusters. Each node would have k features, which is then applied *k-means* clustering to obtain the clusters.

11.5.5 Reasons for Clustering Multiple Eigenvectors

This method is applied more commonly as it resolves issues of `Recursive Bipartitioning`. It is also shown empirically that it

1. **Approximates the optimal cut**
2. **Emphasize cohesive clusters** - this can be seen in Figure 48 where it emphasizes uneven distribution of the data while emphasizes similarity and attenuate dissimilarity
3. **Well-Separated Space** - tranforms the data into k orthogonal vectors
4. Multiple eigenvectors **prevents numerical instability**

11.5.6 Determining k

Unlike other clustering exercises, determining the number of clusters is usually determined by obtaining stable clusters such that it **maximizes** the *eigengap*

$$\Delta_k = |\lambda_k - \lambda_{k-1}|$$

12 Motif-Based Spectral Clustering

Spectral Clustering can be applied to patterns other than just nodes itself.

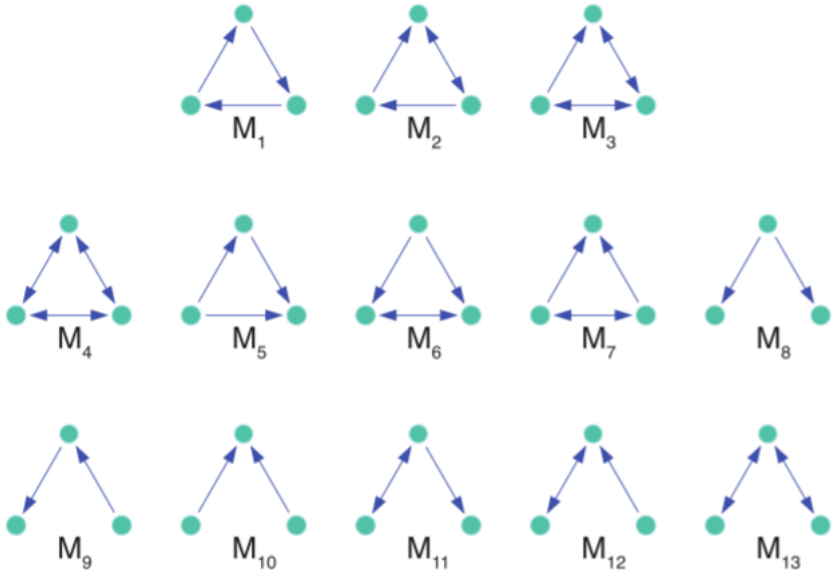


Figure 49: Motifs with $n = 3$

Motif-based spectral clustering is analagous to node-level spectral clustering, and we can define `motif-conductance` as

$$\phi_M = \frac{|cut_M|}{vol_M(S)}$$

Referring to Figure 50,

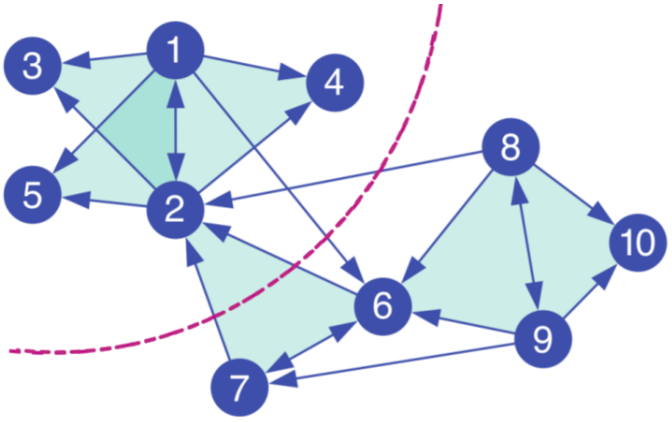


Figure 50: Motif Cut of Graph

1. $|cut_M|$ is the **number of motif of interest** (M_7 from Figure 49) being cut, which is 1 in this case
2. $vol_M(S)$ is the **number of endpoints** in subgraph. Based on the left subgraph, there are three instances of the motifs (each with three endpoints) and 1 node from the "cut" motif (node 2). This gives $3 \times 3 + 1 = 10$ endpoints
3. $\phi_M = \frac{1}{10}$ for this cut

12.1 Finding Motif-Based Clusters

Similar to the problem of spectral clustering, we are interested in a "good" partition such that ϕ_M is **minimized**.

However, this is NP-hard problem and we need a more efficient method approximate it.

12.2 Motif-Based Spectral Clustering Algorithm

The outline of the algorithm is as follows

1. Given Graph G and Motif M
2. Transform G into an undirected weighted graph $W^{(M)}$
3. Apply spectral clustering on $W^{(M)}$

Theorem: the output clusters would obtain near to optimal motif conductance with an approximation guarantee.

12.2.1 Transformation of G

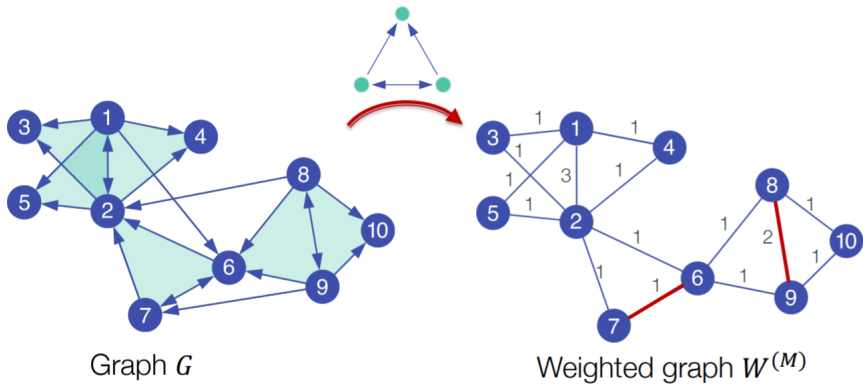


Figure 51: Transformation of G to $W^{(M)}$ based on M

Based on Figure 51, we can perform the transformation as follows

1. Identify the occurences of motif M
2. Each edge $(i, j) \in W$ represents the number of times the two nodes participate in the motif M
3. Nodes (1,2) participated the motif M three times and therefore has a label of 3 in $W^{(M)}$

12.2.2 Decomposition of $W^{(M)}$

Decomposition is the next step of the algorithm, which follows the same steps as standard spectral clustering with $W^{(M)}$ at its place instead

1. Calculate Laplacian $L = D - W^{(M)}$
2. Identify Fiedler Vector $\vec{x}_2$
3. Use $\vec{x}_2$ to identify communities

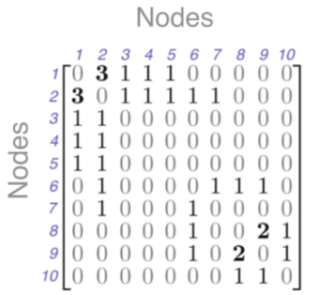


Figure 52: Adjacency Matrix of $W^{(M)}$

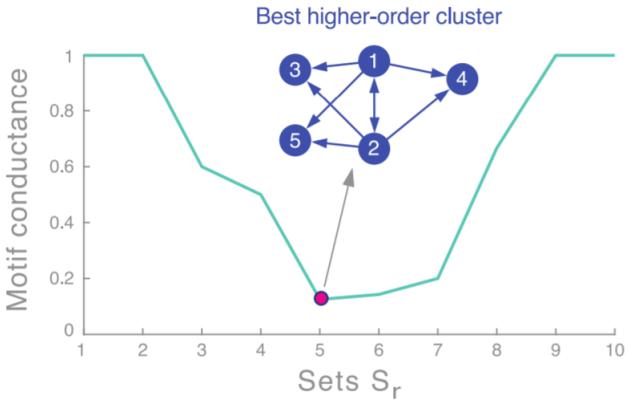


Figure 53: Sweeping through $\vec{x}_2$ and Calculating Motif Conductance

12.2.3 Sweeping for Community Identification

A more common approach in identifying communities in motif-based spectral clustering involves sweeping which is done based on the following steps:

1. Rank values of $\vec{x}_2$
2. For each step, include the next element into S_r and compute conductance
3. Obtain the optimal set S_r such that conductance is **minimized**

12.2.4 Approximation Guarantee

The algorithm provides a near optimal approximation guarantee based on the `Cheeger Inequality`

$$\phi_M(S) \leq 4 \times \sqrt{\phi_M^*}$$

1. $\phi_M(S)$ is the conductance obtained by the algorithm
2. ϕ_M^* is the optimal conductance

12.3 Application with Unknown Motif: Food Webs

Based on the study of Florida Bay's Food Web, we can construct a graph based on the following relationship

1. **Nodes**: species in the ecosystem
2. **Edges**: carbon exchange (nicer way to say who eats whom)

Question: what kind of motif organizes the food web?

12.3.1 Sweeping Profile

Constructing the sweeping profile, we can see which motif fits the best

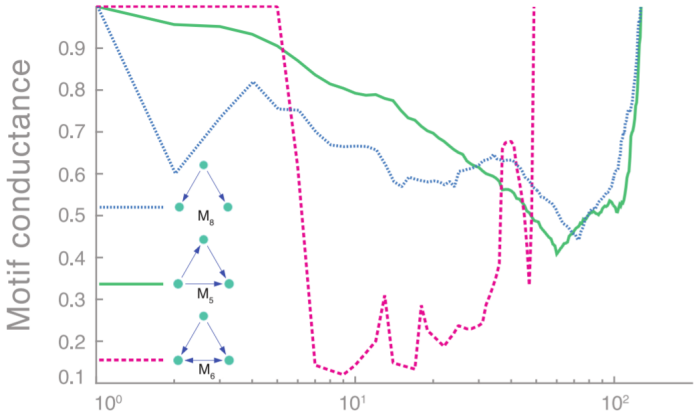


Figure 54: Sweeping Profile of Florida Bay's Food Web

Based on Figure 54, we can see that motif M_6 seems to achieve high conductance very quickly and significantly. This suggest that the food web is organized around M_6

1. M_6 has one node with two out-going edges and two nodes that points at each other
2. This means that the food web is likely to be organized around predators who prey on a common food source and each other

12.3.2 Ground Truth Food Web

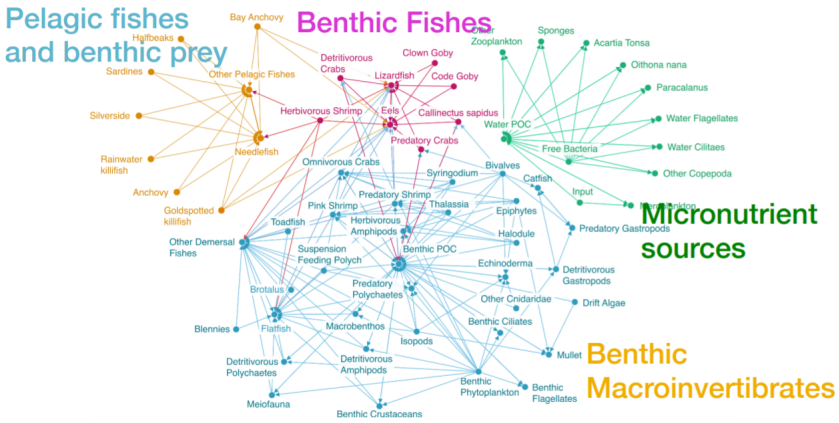


Figure 55: Actual Food Web and the Clusters

The groundtruth food web in Figure 55 revealed known aquatic layers and helped to identify unknown ones.

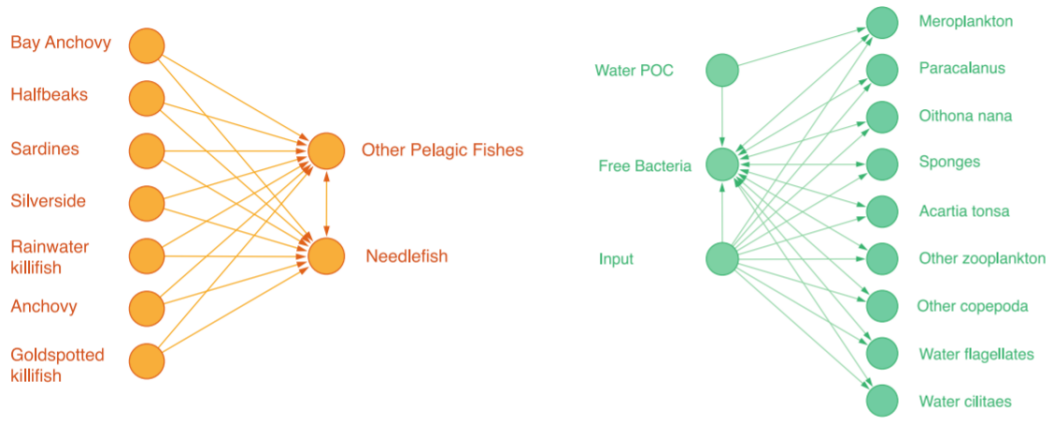


Figure 56: Communities obtained from Motif-Based Spectral Clustering

Breaking down the clusters into individual components, we can see that foodweb is indeed organized according the motif M_6.

12.4 Application with Known Motif: Gene Regulatory Networks

A known motif in Gene Regulatory Networks are **feed-forward loops** (`FFL`), which is a three-gene pattern, composing of two input transcription factors, one of which regulates the other and both targeting a gene. They can either be *activating* or *repressing* as depicted with a positive and negative sign in Figure 57.

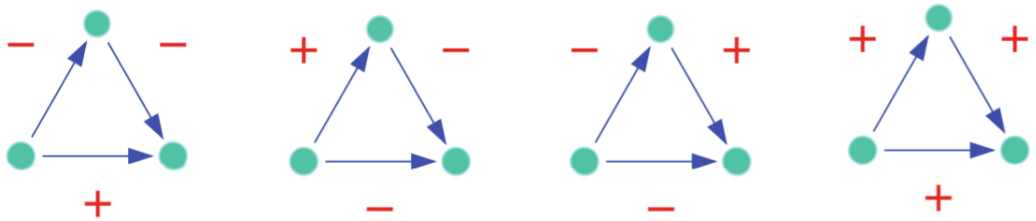


Figure 57: Feed Forward Loops in Gene Regulatory Networks

A Gene Regulatory Network can be defined accordingly

1. **Nodes**: Genes in mRNA
2. **Edges**: Directed transcriptional relationships

Due to the very important biological function of such motifs, we can cluster the network based on such a motif so that we can identify how genes in mRNA work together for a specific function.

12.4.1 Yeast Regulatory Network

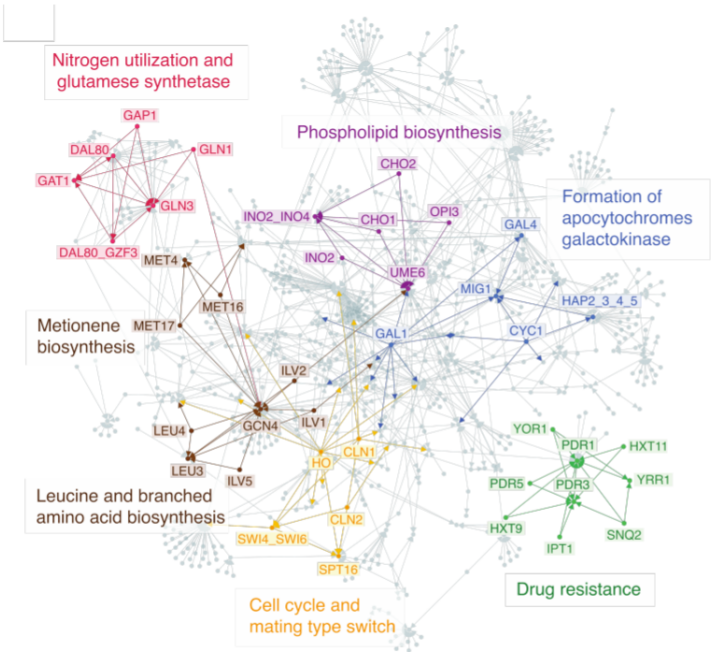


Figure 58: Actual Yeast Regulatory Networks

From Figure 58, we can identify clusters with FFLs and from there, identify its functions.

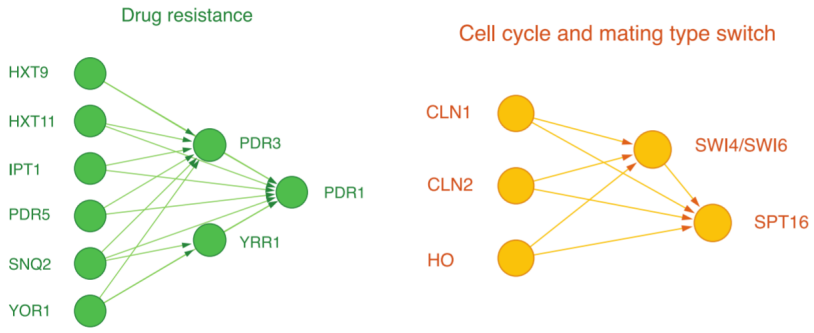


Figure 59: Feed-forward Loops a Closer Look

Genes that are involved in certain functions can be then identified as shown in Figure 59.

12.5 Other Partitioning Methods

Some other partitioning methods that are commonly used include

1. **METIS** - heuristic algorithm that works well in practice
2. **Graclus** - based on kernel k-means
3. **Louvain** - based on Modularity Optimization
4. **Clique Percorlation Method** - allows for overlapping communities

13 Message Passing and Node Classification

Key Problem: Given a network with some labeled nodes, how do we assign labels to other nodes?

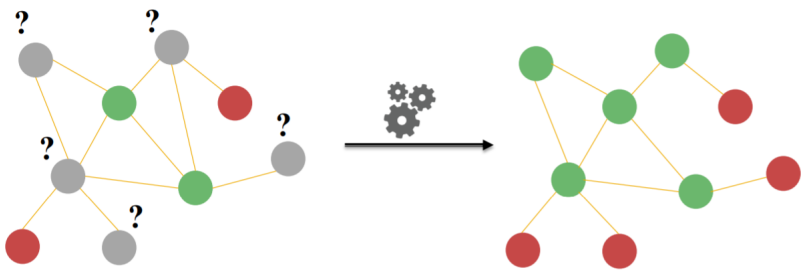


Figure 60: Node Classification Task

This is a task for node classification:

1. Input labeled nodes
2. Predict labels of unlabeled of nodes
3. Semi-supervised learning

13.1 Correlations in Networks

Individual behaviours are correlated with network behaviours. Leveraging that would allow us to classify nodes based on how connected the nodes are. The three types of dependencies are

1. **Homophily** - Individual characteristics influence social connections, which is the result of similar individuals connecting with each other.
2. **Influence** - Social connections influences individual characteristics, which is the result of influence from peers
3. **Confounding** - a third cause "environment" that influences both connections and individuals resulting to correlation

13.2 Guilt-by-Association

If I am connected to X, I am likely to be X. A common example are *malicious websites* that tend to link to each other. This allows us to classify nodes based on

1. **Features** of node i
2. **Labels** of i's *neighbourhood*
3. **Feature** of i's *neighbourhood*

This assumes the existence of **homophily**.

13.2.1 Formalizing the Problem

Task: Given a graph G with a few labeled nodes, find the class of other nodes.

1. Let W be a $n \times n$ adjacency matrix
2. $Y = \{-1, 0, 1\}^n$ is a vector of labels with 0 being unlabeled
3. Predict whether node is 1 or -1

13.3 Collective Classification

Collective classification, which leverages the labels of linked nodes in order to classify the node, uses correlations between the nodes from their links. This depends on the **Markov Assumption** - the label Y_i of node i depends on the neighbours N_i.

$$P(Y_i|i) = P(Y_i|N_i)$$

It involves three main steps

1. **Local Classifier** - initialize label assignments based on node attribute using standard classification task. Does not leverage network features
2. **Relational Classifier** - Capture the correlations on the networks based on its neighbours
3. **Collective Inference** - Propagating the correlation across the network until inconsistencies are minimized

Exact inference is NP-Hard. However, approximations often work very well.

13.4 Probabilistic Relational Classifier

Intuition: Class probability of i, Y_i, is a weighted average of class probabilities.

13.4.1 Algorithm

The basic steps are as follows:

1. Labeled nodes are initialized with "ground-truth"
2. Unlabeled nodes are initialized uniformly (can use prior, but this might result to longer convergence due if it is not accurate)
3. Update all nodes in random until convergence

For the third step, for each node i and each label c, calculate

$$P(Y_i = c) = \frac{1}{|N_i|} \sum_{(i,j) \in E} W(i,j) P(Y_j = c)$$

1. The formula is the weighted average of the neighbours of node i
2. $W(i,j)$ is the edge weight
3. $|N_i|$ is the number of neighbours of i

13.4.2 Limitations of Algorithm

For this algorithm,

1. It cannot guarantee convergence (but works well in practice)
2. Model cannot use node feature information

13.5 Iterative Classification

Intuition: Improve on probabilistic classifier by including node features
This is done by

1. Creating a flat vector a_i for i
2. a_i contains features of i but neighbourhood level data
3. Neighbourhood level data is represented by *aggregation* (mean, mode, exists, count etc.)
4. Train classifier (kNNs, SVM etc.) on a_i
5. Repeat until convergence (neighbourhood level features change)

13.5.1 Basic Architecture

Bootstrap Phase:

1. Convert each node i to a_i (feature vector)
2. Use local classifier $f(a_i)$ to compute Y_i

Iteration Phase:

1. Repeat for each node i and update node vector a_i'
2. Update Y_i using $f(a_i')$
3. Stop when **convergence**

However, this does not guarantee convergence!

13.6 Loopy Belief Propogation

Loopy Belief Propogation is a dynamic programming algorithm that answers **conditional probability queries on a graph model**.

1. Intuition based on **message passing**
2. Node i **believes** node j has a certain likelihood of being in certain states
3. **Update** j's likelihood
4. Repeat for all nodes until **consensus** is reached

13.6.1 Message Passing

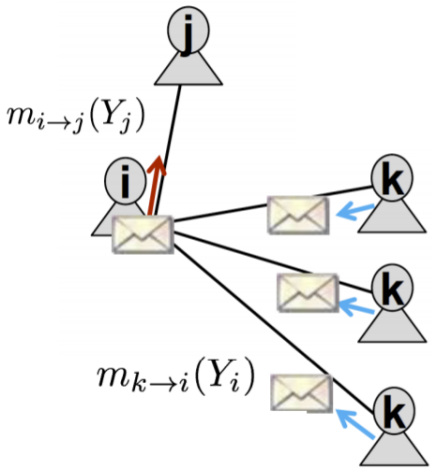


Figure 61: Idea of Message Passing Amongst Nodes

The message passed to j from i is dependent on what it hears from its (i's) neighbours k. Each neighbour k would pass a message about its belief of the state to i

$$m_{i\to j}(Y_j) = \alpha \sum_{Y_i \in \mathcal{L}} \psi(Y_i, Y_j)\phi_i(Y_i) \prod_{k \in \mathcal{N}_\rangle - j} m_{k\to i}(Y_i)$$

1. $m_{i\to j}(Y_j)$ is the message i to j about the likelihood of states of Y_j
2. α can be thought of as the learning rate
3. ψ is the **label-label dependency matrix**, whereby $\psi(Y_i, Y_j)$ is the probability that j is the state Y_j given that node i is in the state of i

4. $\phi_i(Y_i)$ is the **prior belief** of that node i has the state i
5. $\prod_{k \in \mathcal{N}_\backslash - j} m_{k \to i}(Y_i)$ is the *combined* messages i receives from neighbours k other than j
6. Note that the **assumption of independence** of message (that is the message from k to i are not directly dependent) is key for this algorithm to work
7. $\mathcal{L}$ is the set of all possible states

After convergence (or consensus), that is when there is little (or no) change(s) to the to beliefs, the belief that i is in the state Y_i is

$$b(Y_i) = \alpha \phi_i(Y_i) \prod_{k \in \mathcal{N}_\backslash} m_{k \to i}(Y_i)$$

13.6.2 Cyclic Graphs

In a situation where there exist cycles in the graph, the independent assumption of this algorithm would be violated. This results to two problems:

1. No halting condition - loops will update again and again which means no consensus can be reached
2. Incorrect treatment of separate evidence - since the message is no longer independent to the original message, the node should not reconsider this evidence. However, this violation would lead to reinforcing behaviour

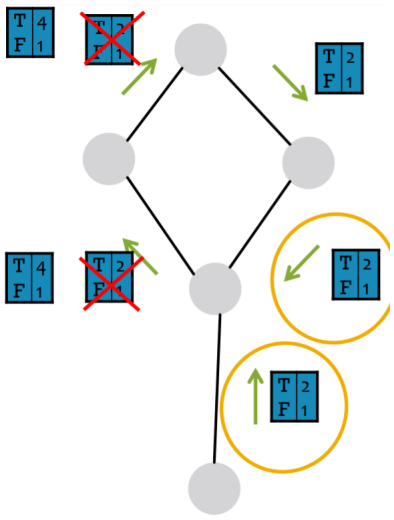


Figure 62: In a situation where a cycle exists, then the aggregation of beliefs increase leading to self-reinforcing beliefs

Solutions to this problem are:

1. Ignore it - in most real networks where **clustering coefficient is low**, this is not a problem since loops tend to be long if they exist and rare
2. Remove cycles before implementing this algorithm

13.6.3 Advantages and Limitations

Advantages:

1. Easy to program and parallelize

2. Can be generalized to various graphical model

Limitations:

1. No convergence guarantee (problematic especially when loops are prevalent)
2. Parameters ψ, ϕ, α requires domain specific knowledge or training (using gradient descent) to estimate

13.6.4 Application: Online Fraud Detection (Pandit et al., 2007)

Auction fraud, the most common of which is not delivering the product, is a common problem in a lot of sites. Most sites provide a rating system, but this is not sufficient because

1. Fraudsters can set individual characteristics to appear legitimate
2. Fraudsters can collude to improve reputation score

How do Fraudsters Interact?

This is where domain specific knowledge comes into play. The researchers realized that Fraudsters do not improve each other's ratings directly, which potentially link themselves to each other. They accomplish this by having a network of accomplices.

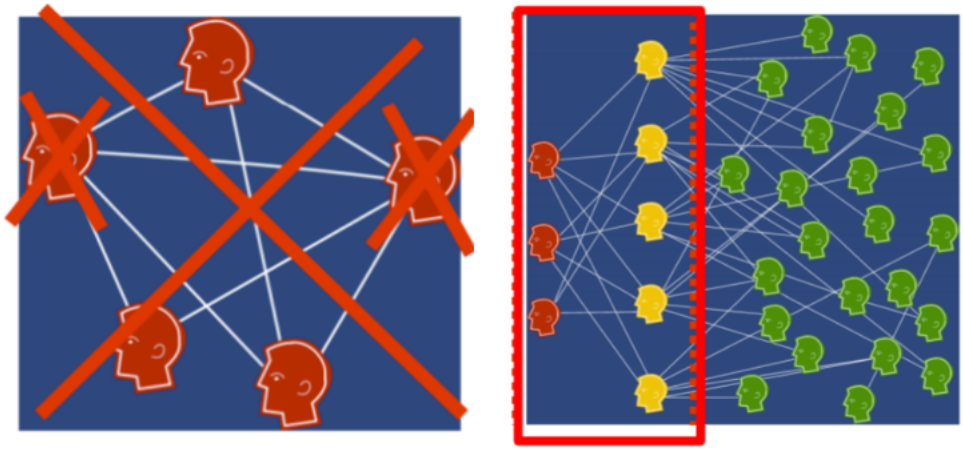


Figure 63: Fraudsters do not directly interact with each other but via accomplices to appear legitimate

Parameters

Prior Beliefs ϕ_i - if there is no reliable prior knowledge, then an unbiased distribution can be used

	Node state		
Neighbor state	**Fraud**	**Accomplice**	**Honest**
Fraud	ϵ_p	$1 - 2\epsilon_p$	ϵ_p
Accomplice	0.5	$2\epsilon_p$	$0.5 - 2\epsilon_p$
Honest	ϵ_p	$(1 - \epsilon_p)/2$	$(1 - \epsilon_p)/2$

Figure 64: Parameters of Label-Label Dependency Matrix ψ

Based on intuition or previous research, the label-label dependency matrix ψ is constructed as shown in Figure 64. This can be interpreted as the probability of interactions between frauds, accomplices and honest users. Based in the beliefs b_i, fraudsters can be identified.

14 Graph Representational Learning

For `Machine Learning` in graphs, two common purposes are

1. Node Classification
2. Link Prediction

However, one of the time constraint of any Machine Learning project is the collection and manipulation of data known as **feature engineering**.

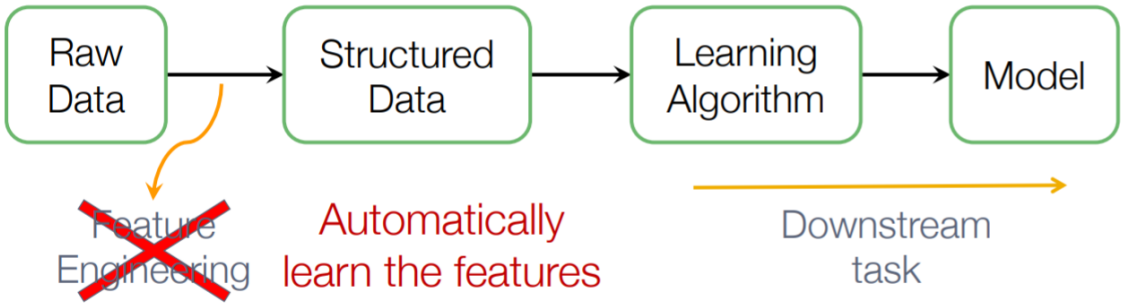


Figure 65: An ML Pipeline Where Most of the Time is Spent on Feature Engineering. Having an Automated System Would Increase Performance

14.1 Feature Learning in Graphs

The overarching **goal** of feature learning is to create a **task-independent** feature learning for ML tasks in graphs.

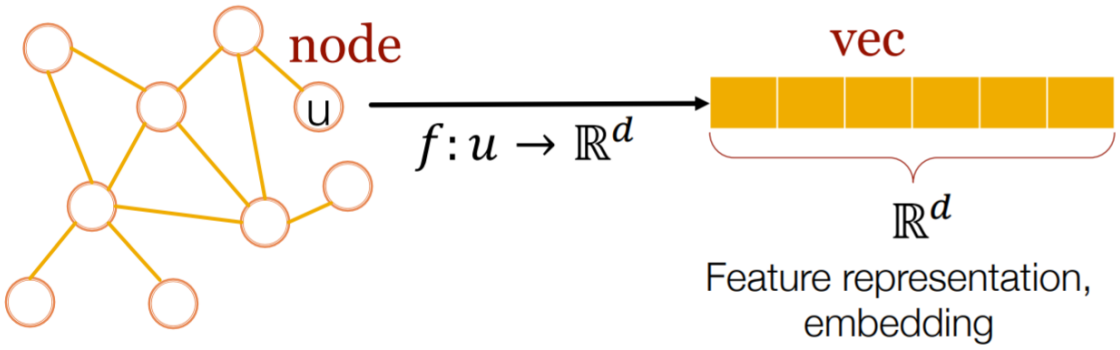


Figure 66: Feature Learning in Graphs

In essence, we want to map node u to a *feature representation* (or *embedding*) that is in the form of a vector with the dimension d ($\vec{u} \in \mathbb{R}^d$). This provides us with

1. Distributed Representations of Nodes
2. Similarity in embedding of nodes indicate similar network structure
3. Encode network information and generate node representation
4. Perform common ML tasks with such representations

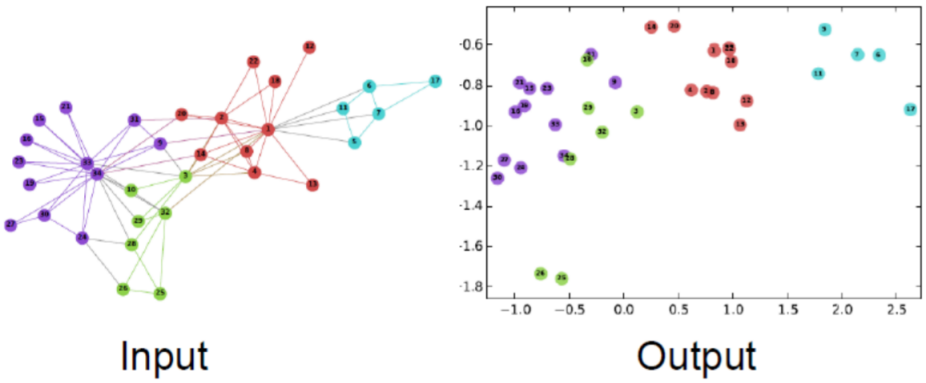


Figure 67: Embedding of `Zachary's Karate Club Network` into a 2D Space

14.1.1 Challenges in Graph-Based Representational Learning

Representational Learning is not a new concept. However, for task such as **computer vision** and **NLP**, the data structures are simpler.

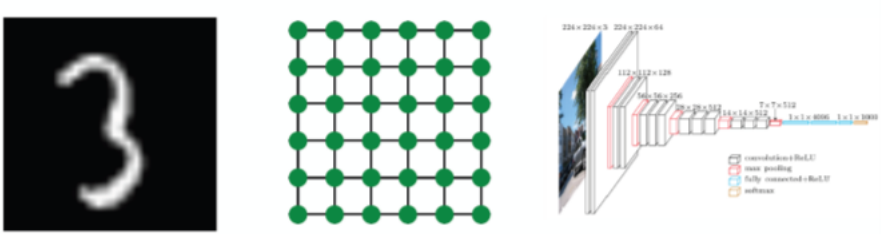


Figure 68: `CNN` Convolutions for Computer Vision Tasks

For Convolutional Neural Networks in Figure 68, convolutions are laid out in grid-like structures and moved across the image.

Figure 69: `RNN` Convolutions for NLP Tasks

Likewise, in Recurrent Neural Networks, words can be vectorized based on a chain graph structure, which is sequential.

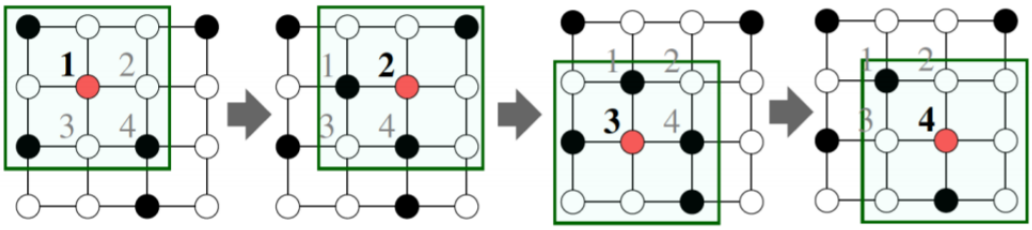


Figure 70: No Easy Fit for Graphs

However, there is no easy solution for representational learning for graphs due to the following reasons

1. **Complex topological structure** - there is no spatial locality
2. **No fixed node ordering** - problem of isomorphism
3. **Dynamic** - node and edges can change over time
4. **Multimodal** - different types of edges can exist simultaneously

14.2 Embedding Nodes

Problem: How do we encode nodes such that the encoding similarity between $\vec{u}$ and $\vec{v}$ is approximately the same as the similarity of the nodes u and v?

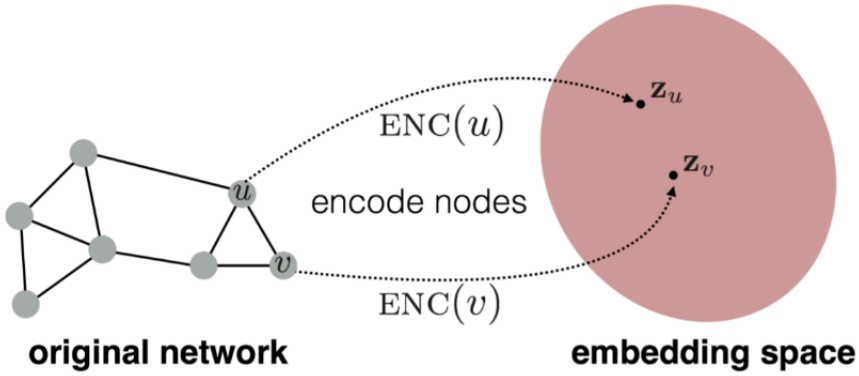


Figure 71: Task of Encoding - Mapping the Original Network to an Embedding Space

More formally, we say

$$similarity(u, v) \approx \vec{u}^\top \vec{v}$$

14.2.1 Learning Node Embeddings

To be able to learn node embeddings, we have to solve the key problems:

1. Define the encoder - how do we map nodes to vectors?
2. Define node similarity - how do we define node similarity in the network context?
3. Optimization of parameters - how do we achieve $similarity(u, v) \approx \vec{u}^\top \vec{v}$?

Note that we use $\vec{u}^\top \vec{v}$ which is the dot product of two vectors. However, we have to **normalize** the vector which effectively makes it a *cosine* similarity. This is because similarity measured by the dot product can increase proportional to the length of the vector, which is problematic.

14.2.2 Two Key Components

Encoder: this function maps each node to a low-level dimensional vector

$$ENC(u) = \vec{u},\ \vec{u} \in \mathbb{R}^d$$

1. u is the node in the graph
2. $\vec{u}$ is the d dimensional embedding

Similarity function: specifies how the relationship in vector space map to the relationships in network

$$similarity(u, v) \approx \vec{u}^\top \vec{v}$$

1. $similarity(u, v)$ is the similarity in the network
2. $\vec{u}^\top \vec{v}$ is the cosine similarity of the node embedding (aka the decoder)

14.2.3 Shallow Encoding

Shallow encoding is the simplest form of encoding which effectively is a form of an **embedding lookup table**.

$$ENC(u) = \mathbf{Z}\vec{v}$$

1. $\mathbf{Z} \in \mathbb{R}^{d \times |V|}$ is the embedding matrix such that each column is a node embedding (THIS IS WHAT WE WANT TO LEARN)
2. $\vec{v} \in \mathbf{I}^{|V|}$ is the indicator vector such that all entries are zeros except for the one which indicates node v

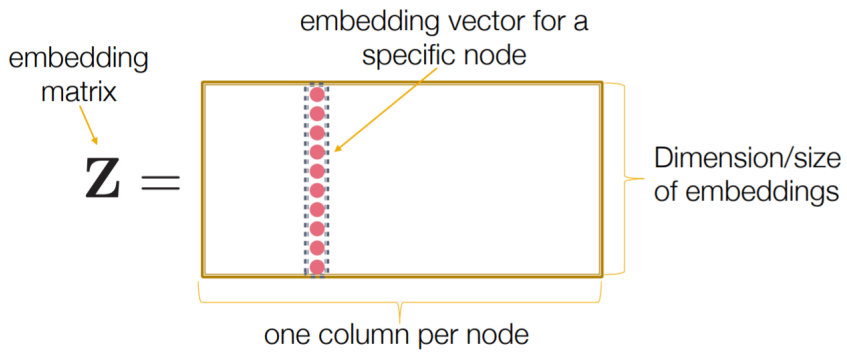


Figure 72: Embedding Matrix and its Dimensions

This effectively assigns each vector to its own column (or a unique embedding vector). This means that $\mathbf{z}_u$ corresponds to column in $\mathbf{Z}$ such that $\vec{v}_u = 1$

14.2.4 Defining Similarity

Similarity is a more difficult idea to capture since we have learned that there are various definitions of what it means to be similar. This include

1. Connectivity of nodes
2. Shared neighbours (egonet intersections)
3. Structural roles
4. Communities

14.3 DeepWalk `Perozzi et al. 2014`

Given a graph, we select a node at random and traverse the graph based on a randomly selected neighbour for a specific number of times (or until all are visited, which is known as unbounded walks). This random sequence is known as a **random walk**.

This allows us to define a similarity function as

$$\mathbf{z}_u^\top \mathbf{z}_u \approx \textit{probability that } u \textit{ and } v \textit{ co-occur in the same random walk}$$

DeepWalk utilizes *unbiased random walk* as its strategy to calculate the similarity function between nodes u and v

14.3.1 Random Walk Node Embeddings

Random Walk is used for node embeddings to estimate the probability of node v starting from node u with the strategy R.

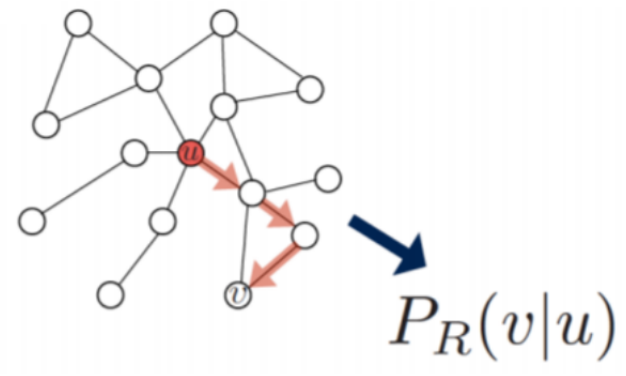


Figure 73: Probability of a Co-occurence in a Random Walk

Similarity in the embeddings $\mathbf{z}_u^\top \mathbf{z}_u$ is optimized such that it *matches* with the random walk statistic.

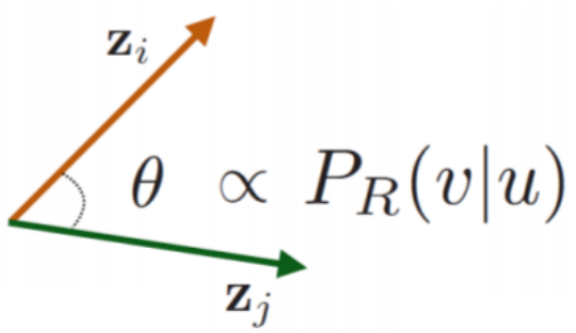


Figure 74: `cosine` similarity

Random Walks are used for node embeddings because of its

1. **Expressivity** - incorporates a stochastic definition for node similarity that incorporates local and higher-order neighbourhood information. Strategies R can be tweaked to express certain information that is more important
2. **Efficiency** - only takes into account co-occuring pair of nodes in random walks, which makes it scalable

14.3.2 Problem Setup

Goal: Find embeddings of nodes in graph such that it preserves similarity in graphs

Idea: Similarity can be defined as nearby nodes with connectivity. u and v would have a lot of co-occurences in random walks if u and v are close to each other, or in the neighbourhood of u ($N_R(u)$)

$N_R(u)$: the neighbourhood of u based on strategy R, which is unbiased for DeepWalk.

14.3.3 Feature Learning as an Optimization Problem

In order to find the following mapping

$$f : u \to \mathbb{R}^d$$

We can formulate this problem as a Maximum Likelihood Estimation problem such that the objective function is as follow

$$\max_f \sum_{u \in V} \log P(N_R(u)|\mathbf{z}_u)$$

Given $\mathbf{z}_u$, we want to learn the feature representations that are predictive of the nodes in the neighbourhood.

This leads to the 3 main steps in the DeepWalk algorithm:

1. Perform fixed length **random walks** with strategy R
2. $\forall u$ **build the multiset** (same node can occur multiple times) $N_R(u)$ which is the neighbourhood of u
3. Perform the optimization problem using **Stochastic Gradient Descent**

14.3.4 The Optimization Problem

The **loss function** (which is the same as the objective function defined above) would be defined as

$$\min_f \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

1. $\sum_{u \in V}$ - sum of all nodes
2. $\sum_{v \in N_R(u)}$ - sum of all nodes that appear in the multiset in u
3. $P(v|\mathbf{z}_u)$ - predicted probability of u and v co-occuring in $N_R(u)$

$P(v|\mathbf{z}_u)$ is going be defined with the `softmax` function, which is used because we want that node v to be the most similar to node u wrt to other nodes in the graph.

$$P(v|\mathbf{z}_u) = \frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{n \in V} e^{\mathbf{z}_u^\top \mathbf{z}_n}}$$

This gives the following loss function

$$\min_f \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\Big(\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{n \in V} e^{\mathbf{z}_u^\top \mathbf{z}_n}}\Big)$$

Notice that this effectively makes (1) $O(|V|)$ (which is unavoidable), (2) to be $\Theta(\mathbf{E}[|N_R(u)|])$ and (3) to be $O(|V|)$. This can lead to **quadratic explosion** $O(|V^2|)$.

14.3.5 Estimation based on Negative Sampling

Instead of normalizing against all nodes $u \in V$, we can take k random samples instead

$$\log\left(\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{n\in V} e^{\mathbf{z}_u^\top \mathbf{z}_n}}\right) \approx \log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - \sum_{i=1}^{k} \log(\sigma(\mathbf{z}_u^\top \mathbf{z}_{n_i})), n_i \ P_V$$

1. k is chosen based on degree (typically $\in [5, 20]$)
2. Higher k leads to more robust estimates
3. Higher k leads to higher bias on negative events

14.4 Node2Vec `Grover & Leskovec, 2016`

DeepWalk uses unbiased random walks for the definition of $N_R(u)$, which constrains the definition based on network connectivity alone. Node2Vec addresses the problem by relaxing this constraint with biased walks.

14.4.1 Local and Global Neighbourhoods

Random Walks can be biased to reflect local and global neighbourhoods. This can be achieved by using `BFS` biased (local) and `DFS` biased (global walks)

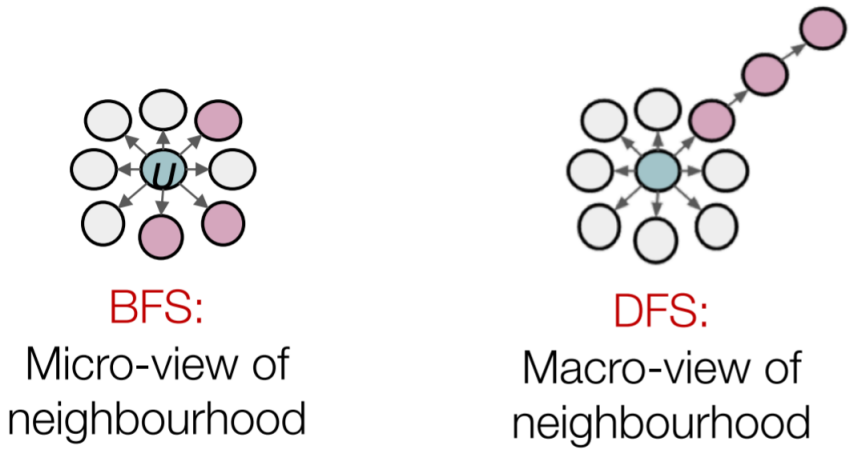


Figure 75: Macro and Micro-view of u's Neighbourhood

14.4.2 Interpolating between `DFS` and `BFS`

How do we achieve biasing the walks between `DFS` and `BFS`? Node2Vec uses two parameters

1. p (Return Parameter) - returning to previous node
2. q (In-Out Parameter) - Ratio of moving inwards `BFS` and outwards `DFS`

14.4.3 Biased Random Walks

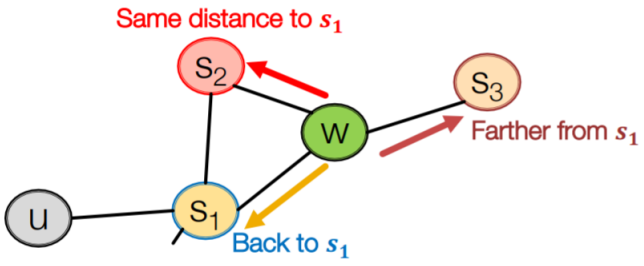


Figure 76: Choice of Biased Walks

Based on Figure 76, with the previous step being from $s_1 \to w$, our next choice would be as follows

1. **Back** to s_1 - this means that the next step after this would be a `BFS` since it would $s_1 \to x$, where x can be any connected node to u
2. **Same distance** to s_1 - this effectively make this step equivalent to `BFS` (with one less step)
3. **Farther distance** from s_1 - this is `DFS`

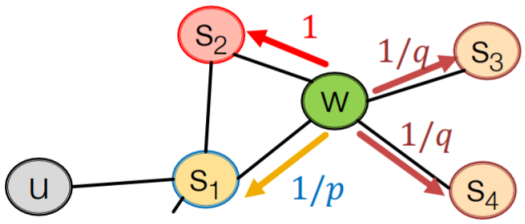


Figure 77: Unnormalized Probabilities of Biased Walks

Based on Figure 77, we can initialize the unnormalized probability (divide each edge assignment with $1 + \frac{1}{p} + \frac{1}{q} + \frac{1}{q}$ to get the actual probabilities) with parameters p and q

1. **Higher** p means that returning is less probable, this **reduces** `BFS`
2. **Higher** q means that exploring out is less probable, this **reduces** `DFS`

14.4.4 Node2Vec Algorithm

The steps are similar to DeepWalk:

1. Perform biased n length **random walks** with strategy $R(p, q)$
2. $\forall u$ **build the multiset** (same node can occur multiple times) $N_R(u)$ which is the neighbourhood of u
3. Perform the optimization problem using **Stochastic Gradient Descent**

14.5 Usage of Node Embeddings

After obtaining node embeddings, typical ML tasks can be performed

1. **Node classification** - $P(Y_i = c|\mathbf{z}_i)$ using the embedding
2. **Cluster** $\mathbf{z}_u$ for community detection
3. **Link Prediction** by aggregating functions $f(\mathbf{z}_i, \mathbf{z}_j)$ to predict if $\exists(i, j)$

15 Introduction to Graph Neural Networks

Up to this point, we have learned how to encode nodes in terms of `shallow encoding`, but what is shallow encoding? In general, shallow encoders are

1. **One layer** of data transformation
2. Node u maps to embedding $\mathbf{z}_u$ by the function $f(\cdot)$ such that $\mathbf{z}_u = f(\mathbf{z}_v, \forall v \in N_R(u))$
3. The embedding **depends on neighbours** of u (how ever it is obtained)

This has led to several problems and limitations to shallow encoders in general:

1. $O(|V)$ parameters are needed - there is no sharing and each node is constructed uniquely
2. Inherently **Transductive** - Node embeddings cannot be generalized to new nodes that is unseen
3. Do not incorporate node level features

15.1 Basis for GNNs

Let out graph by G, such that:

1. V is the vertex set
2. $\mathbf{A}$ is the adjacency matrix
3. $\mathbf{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of node features

The key difference and idea is that instead of using Random Walks to determine node embeddings, we leverage the node's neighbours and define them as **computation graphs**.

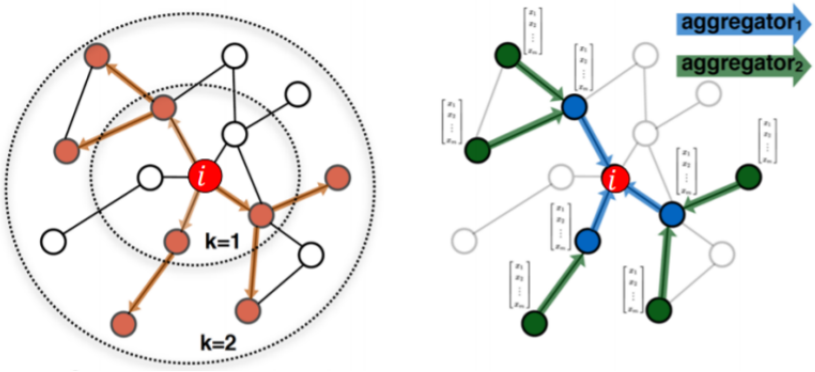


Figure 78: We first determine the computation graph that spans from the node i and then aggregate the neighbourhood level features that feed into i

15.1.1 Neighbourhood Aggregation

To generate the embeddings based on the local neighbourhood, we have to first **aggregate information** from the neighbours.

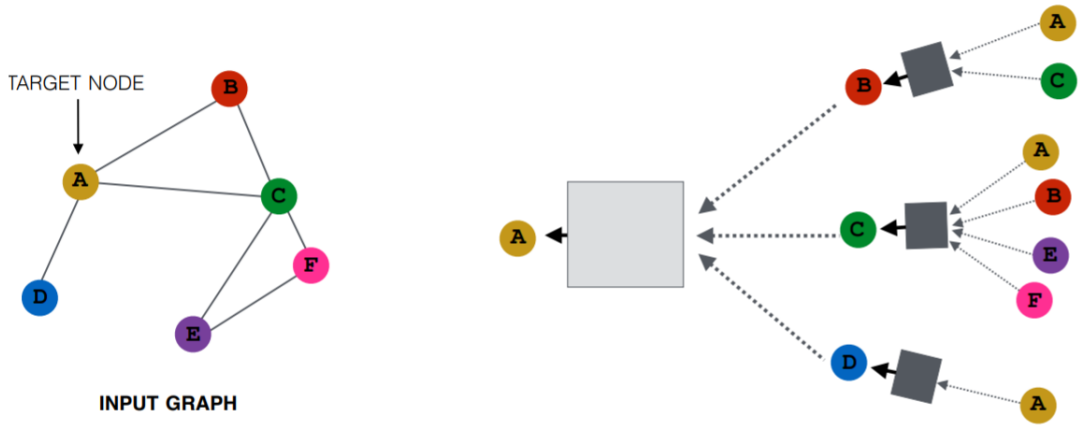


Figure 79: Constructing the Computation Graph for node A

This takes two primary steps:

1. For $\mathbf{X}$ node feature of neighbours, aggregation is performed (summation, average etc.)
2. Aggregated node features are then fed to a Neural Network

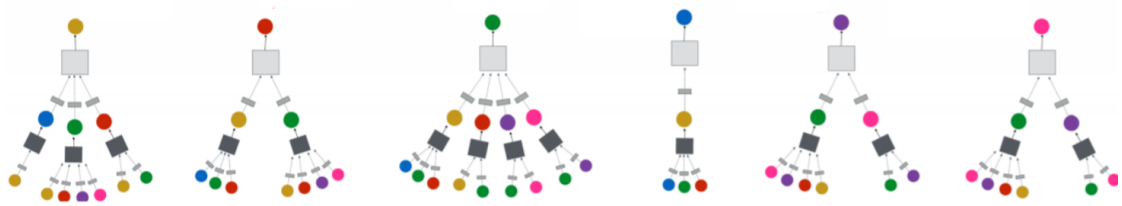


Figure 80: Individual Computation Graphs

Extending from there, each node would have its own computation graph as shown in Figure 80.

15.1.2 Depth of Model

There are two notions of depth when it comes to Graph Neural Networks

1. Depth of Neural Network - this is the number of latent layers within the neural network (in each box)
2. Depth of Layer - this is the number of hops away from the node

Typically, depth of the neural network is held constant and the depth of the model often refers to how many hops away from the node n do we incorporate information from to construct the model. Depth would refer to the second (number of hops) from here.

GNNs can take an arbitrary depth

1. Each node have embeddings at each layer
2. Layer-0 is the the feature $\mathbf{x}_u$ of node u
3. Layer-k embedding gets information from the aggregate of all the nodes k hops away from node u

15.1.3 Math Behind the Deep Encoder

Let the aggregation method be an average (this assumption will be relaxed), we will have

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

1. This is the "intializing" step
2. The 0th layer is the node feature $\mathbf{x}_v$ of node v
3. No neighbourhood level features are taken into account

$$\mathbf{h}_v^k = \sigma\Big(\mathbf{w}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1}\Big),\ \forall k \in \{1, 2, \ldots, K\}$$

1. $\mathbf{h}_v^k$ is the kth layer of the node representation of v calculated from the aggregated data through the neural network from the previous layer
2. $\mathbf{h}_v^k$ is the $k-1$th (previous) layer of the node representation of node v
3. $sigma(\cdot)$ is the activation function (typically `ReLU`)
4. $\mathbf{w}_k$ is the weight matrix at layer k
5. $\sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}$ is the aggregated (in this case, the average), of the neighbours of node v
6. $\mathbf{B}_k$ is the bias in layer k
7. K is the number of layers, which is small (around 3 or 4) since a typical network has a diameter of around 6

$$\mathbf{z}_v = \mathbf{h}_v^K$$

1. This is the *final* embedding that we want
2. It is the embedding (node representation) after K layers of neighbourhood

15.2 Usage of Deep Encoders

Now that we have developed an understanding of the Deep Encoder, we can define any loss functions to conduct training.

15.2.1 Unsupervised Task

If we want to conduct the training in an **unsupervised manner**, that is to use the graph structure to embed nodes that are close together to have **similar embeddings**, we can define the **loss function** based on

$$\min_f \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\Big(\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{n \in V} e^{\mathbf{z}_u^\top \mathbf{z}_n}}\Big)$$

This is the same loss function as `DeepWalk`, which can be made more powerful with `GNNs`. This is easily extended to other approaches such as `Node2Vec`.

15.2.2 Supervised Task (Node Classification)

For node classification task, where supervised learning is used, the `cross entropy loss function` could be used instead

$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(\mathbf{z}_v^\top \theta) + (1 - y_v) \log(1 - \sigma(\mathbf{z}_v^\top \theta))$$

1. $\mathbf{z}_v$ is the **node embedding** of node v
2. θ is the **parameter**
3. $\sigma(\cdot)$ is a **sigmoid function** to convert the values in terms of probability
4. y_v is the **class** of node v

15.3 Inductive Capabilities in GNNs

A key distinction of GNNs as compared to other shallow encoders is that it has shared parameters. Hence, the number of parameters is described as *sublinear* to the number of nodes $|V|$.

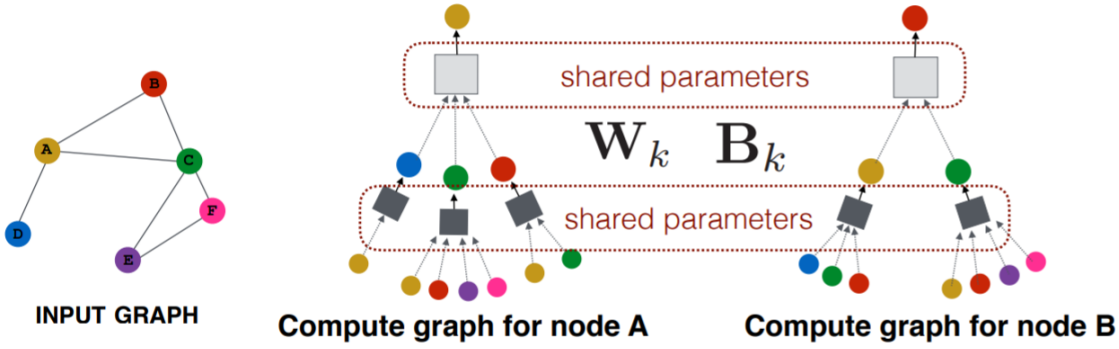


Figure 81: Parameters Sharing in Different Computation Graphs

This is because there exist **shared parameters** between the graphs. Each node contains computation graphs of other nodes, which contains the computation graphs of other nodes. This nested nature means that the parameters are **shareable**.

A more important feature is that it can then be **generalized** to **unseen nodes**.

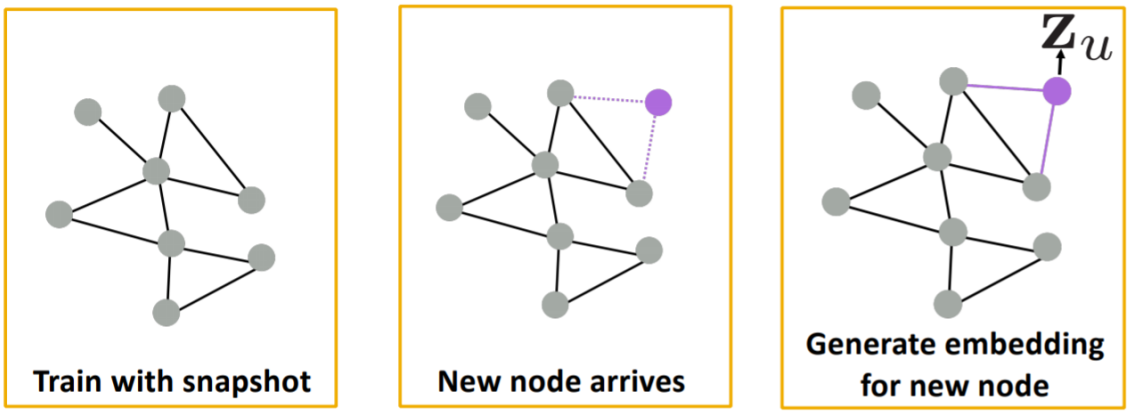


Figure 82: Inducing the New Node's Embedding from Existing Graph

16 General Framework of Graph Neural Networks

`Graph Neural Networks` are constructed based on the **creation of computation graphs** for each node and **aggregate neighbour features and propagate through the network**.

A general GNN Framework is composed of the following components:

1. Defining a single GNN Layer
2. Stacking GNN Layers
3. Defining the Computational Graph
4. Machine Learning Task

16.1 Defining a Single GNN Layer

Different architectures of Graph Neural Networks hinge on the **difference in the construction** of the single layer of the GNN.

Goal: How do you compress a set of vectors (central node and neighbours) into a vector representation.

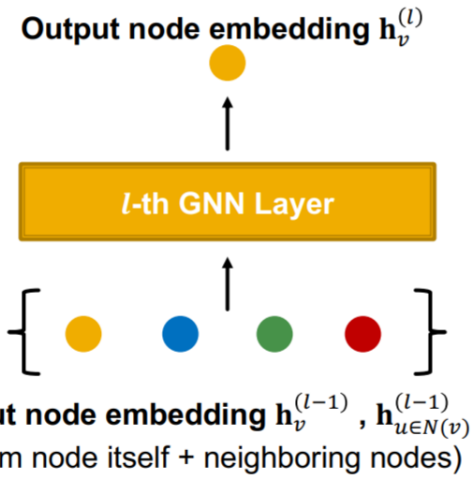


Figure 83: Goal Behind a Single Layer of a GNN

This usually takes two main steps, **Message** and **Aggregation**.

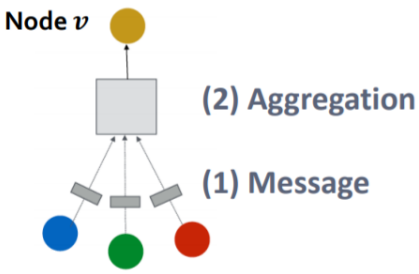


Figure 84: Two Steps in a Single Computation Layer

16.1.1 Message Computation

Each node will create a message and pass down to other nodes, and the message function can be generalized as such

$$m_u^{(l)} = \texttt{MSG}^{(l)}(\mathbf{h}_u^{(l-1)})$$

1. $m_u^{(l)}$ - message of node u at level l
2. $\texttt{MSG}(\cdot)$ - message function that converts inputs into an output. It can be as simple as $MSG(x) = \mathbf{W}x + b$
3. $\mathbf{h}_u^{(l-1)}$ - is the node embedding of node u in the previous layer $l-1$

16.1.2 Message Aggregation

To aggregate all messages from all the neighbours of the node, we will have to use a **permutation invariant function** that aggregates all the messages.

$$\mathbf{h}_v^{(l)} = \texttt{AGG}(\{\mathbf{m}_u^{(l)} : u \in N(v)\})$$

1. $\mathbf{h}_v^{(l)}$ - the node embedding of v at layer l
2. $\texttt{AGG}(\cdot)$ - aggregation function that has to be permutation invariant such as *sum*, *max* or *mean*.
3. $\mathbf{m}_u^{(l)}$ - messages from node u in layer l
4. $u \in N(v)$ - nodes u belong to the neighbourhood of v

A non-linear function is usually added to increase robustness of the model $\sigma(\cdot)$. This can be added either in the message or aggregation computation stage.

16.1.3 Information Loss in Aggregation

A major problem of direct aggregation is **information loss** since $\mathbf{h}_v^{(l)}$ is **not directly dependent** on $\mathbf{h}_v^{(l-1)}$. A solution to this problem is to aggregate the message of node v itself.

$$\mathbf{h}_v^{(l)} = \texttt{CONCAT}(\texttt{AGG}(\{\mathbf{m}_u^{(l)} : u \in N(v)\}), \mathbf{m}_v^{(l)})$$

A typical method to calculate the message of $\mathbf{m}_v^{(l)}$ is by using a different message computation.

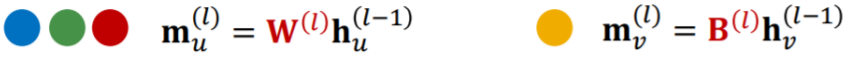


Figure 85: Computing Messages of both $N(v)$ and v itself

16.1.4 Graph Convolutional Network

The first "classical" GNN, Graph Convolutional Network (GCN), has a single computation layer as follows

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

1. **Message Computation** - $\texttt{MSG}(\mathbf{h}_u^{(l-1)})\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$, a single linear layer
2. **Aggregation** - $\texttt{AGG}(x) = \sum_{u \in N(v)} \frac{x}{|N(v)|}$, which is an average
3. **Non-Linear Function** - $\sigma(\cdot)$

16.1.5 GraphSAGE

GraphSAGE relaxes the notion of the **aggregation function**. On top of that, it included the **concatenation** of the "self-message" to prevent information loss. Another important aspect is that also includes L_2 **Normalization**.

$$\mathbf{h}_v^{(l)} = \sigma(\mathbf{W}^{(l)} \cdot \texttt{CONCAT}(\mathbf{h}_v^{(l)}, \texttt{AGG}(\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\})))$$

1. **First-Stage Aggregation** - $\texttt{AGG}(\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\})$
2. **Second-Stage Aggregation** - $\texttt{CONCAT}(\mathbf{h}_v^{(l)}, \mathbf{h}_{N(v)}^{(l)})$

Neighbourhood Aggregation, or the $\texttt{AGG}(\cdot)$ function can be done with different forms as long as they are **permutation invariant**.

1. **Direct Aggregation** - techniques such as sum, mean and max are examples of direct aggregation
2. **Pooling Aggregation** - include a multilayer perceptron to transform neighbours and then perform aggregation
3. **Long-Short Term Memory** - using `LSTM` methods as aggregation. Requires reshuffling for permutation invariance

L_2 Normalization is applied to every layer of $\mathbf{h}_v^{(l)}$

$$\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{||\mathbf{h}_v^{(l)}||_2}, \ where \ ||u||_2 = \sqrt{\sum_i u_i^2}$$

1. L_2 Normalization is used to scale vectors
2. In some cases, this improves performance

16.1.6 Graph Attention Network

In GCN and GraphSAGE, the importance of a node's message is **not considered**. Graph Attention Networks introduce the **concept of attention from cognitive attention** with the parameter α_{vu}.

$$\mathbf{h}_v^{(l)} = \sigma\Big(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\Big)$$

1. α_{vu} - the weighting factor (importance) of message $v \to u$
2. For GCN/GraphSAGE - $\alpha_{vu} = \frac{1}{N(v)}$, where all weights are equal

Attention Mechanism

In most context, the assignment of importance is not explicitly defined but determined implicitly as part of the byproduct of the **attention mechanism** $a(\cdot)$.

$$e_{AB} = a(\mathbf{W}^{(l)} \mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)})$$

1. e_{AB} - attention coefficient of $A \to B$
2. It indicates the importance of message $A \to B$

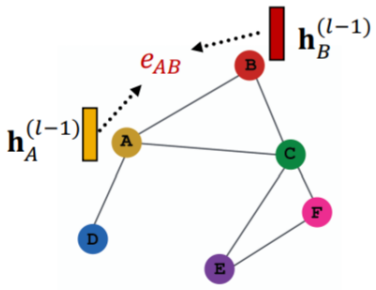


Figure 86: Computing the Attention Mechanism

Attention weights are obtained from the normalization of the attention coefficients using the softmax function

$$\alpha_{AB} = \frac{exp(e_{AB})}{\sum_{k \in N(A)} exp(e_{Ak})}$$

The weighted sum of the is based on the attention weights α_{AB}

Functional Form of the Attention Mechanism

The functional form of attention α is typically **agnostic** to the nature of the problem at hand. A simple functional form would be in the form of a **single layer neural network**.

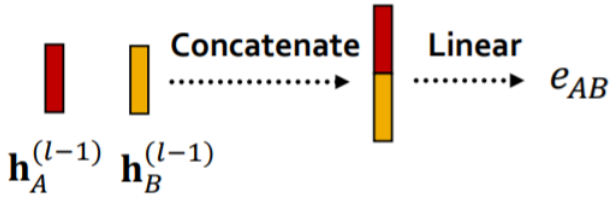


Figure 87: Functional Form of the Attention Mechanism

Based on Figure 87, $a(\cdot)$ is defined as

$$a(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}) = \texttt{LINEAR}(\texttt{CONCAT}(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}))$$

Parameters are trained jointly in an end-to-end fashion.

Stability of $a(\cdot)$

The attention mechanism tends to be **numerically unstable**. This could lead to training to fail to converge. The solution for this problem is by using **multiple attention scores and aggregating them**

$$\mathbf{h}_v^{(l)}[1] = \sigma\Big(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}\Big)$$
$$\mathbf{h}_v^{(l)}[2] = \sigma\Big(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}\Big)$$
$$\mathbf{h}_v^{(l)}[3] = \sigma\Big(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}\Big)$$
$$\mathbf{h}_v^{(l)} = \texttt{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$$

1. $a(\cdot)$ has to be of different functional form
2. Parameters of $a(\cdot)$ has to be randomly initialized

16.1.7 Incorporating Other Deep Learning Modules

A lot of deep learning modules can be easily integrated into **Graph Neural Networks**.

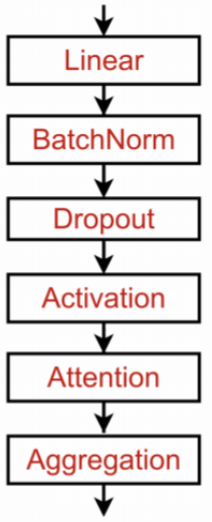


Figure 88: Configurations for a Single GNN Layer

Batch Normalization

Batch Normalization recenters and rescales the input for Neural Networks to provide stability in training. This is done with two key steps:

1. Compute the mean and variance over N embeddings
2. Normalize the feature using computed mean and variance

Using N node embeddings (batch) each with dimension d, $\mathbf{X}_j \in \mathbb{R}^{d \times N}$, we calculate the **mean and variance**

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N}\mathbf{X}_{ij}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{X}_{ij} - \mu_j)^2$$

In the normalization stage, we recenter the values of $\mathbf{X}_{ij}$ around the mean and rescale it with the standard deviation with an added error term for numerical stability.

$$\hat{\mathbf{X}}_{ij} = \frac{\mathbf{X}_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Lastly the **transformation** step using a Batch Normalization Mapping (in this case assuming to be a linear mapping) $BN_{\gamma,\beta} : \hat{\mathbf{X}}_{ij} \rightarrow \mathbf{Y}_{ij}$ is trained in each batch for an optimal normalized node embedding.

$$\mathbf{Y}_{ij} = \gamma\hat{\mathbf{X}}_{ij} + \beta$$

1. γ and β are trainable parameters and $\gamma, \beta \in \mathbb{R}^d$
2. $\mathbf{Y}_{ij} \in \mathbb{R}^{d \times N}$ is the normalized node embedding

Dropout

To prevent the problem of overfitting in Neural Networks, dropout with a probability p randomly sets neurons to 0.

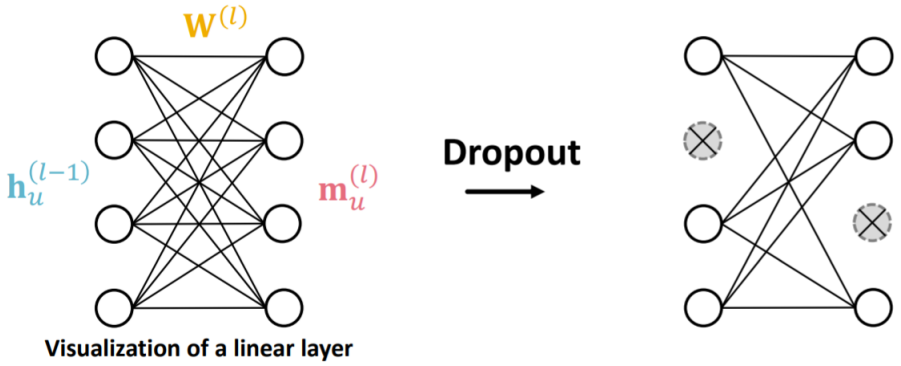


Figure 89: Dropout in GNNs

This is usually applied in the message computation layer, where some of the messages are dropped randomly.

Activation Function

A nonlinear activation function is usually added to the node embedding to **increase the expressive power** of neural networks.

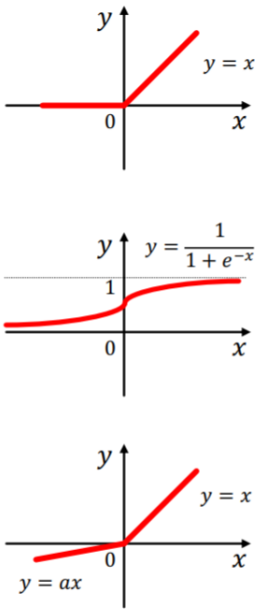


Figure 90: Activation Functions

Based on Figure 90,

1. **Rectified Linear Unit** - $\texttt{ReLU}(\mathbf{X}_i) = max(\mathbf{X}_i, 0)$
2. **Sigmoid Function** - $\sigma(\mathbf{X}_i) = \frac{1}{1+exp(-\mathbf{X}_i)}$
3. **Parametric Rectified Linear Unit** - $\texttt{PReLU}(\mathbf{X}_i) = max(\mathbf{X}_i, 0) + a \cdot min(\mathbf{X}_i, 0)$

16.2 Stacking GNN Layers

Graph Neural Networks, like other forms of Neural Networks, rely on stacking many layers for the expressivity of these networks. However, the notion of stacking is fundamentally different

1. Each layer represents a hop from the "center" node
2. Stacking more layers lead to oversmoothing

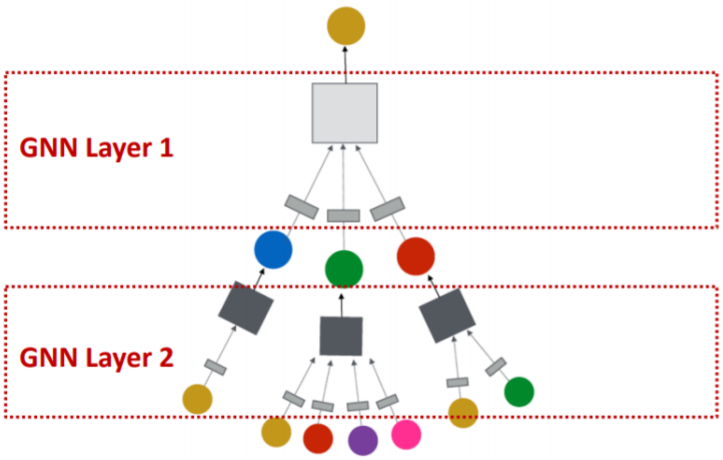


Figure 91: GNN Layers as Sequential Hops

16.2.1 The Problem of Oversmoothing and the Receptive Field

Unlike other Neural Network architectures, stacking more layers **DOES NOT** improve performance! This is because of the **problem of oversmoothing** - values of all nodes converge to the same value!

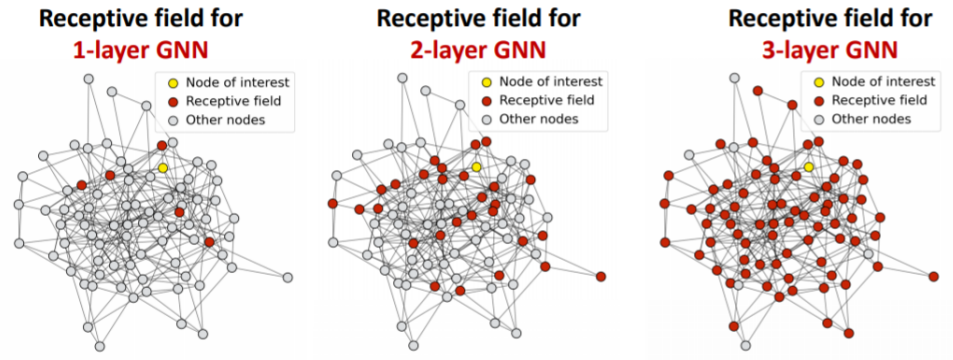


Figure 92: Receptive Field of a Node

This happens because of the nature of graphs, which expands exponentially.

For each node, it has its own Receptive Field that determines the node's embedding. A $K-layer$ GNN would have a receptive field of $K-hop$ neighbourhood for each node.

In Figure 92, we can see that by the third hop, the receptive field almost covers the whole graph. This results to huge overlaps for each node if K is "too big", which leads to the convergence of node embeddings.

16.2.2 Expressivity of Shallow GNNs

Shallow GNNs can be expressive as well. This is done by increasing the **complexity of the aggregation/transformation step**.

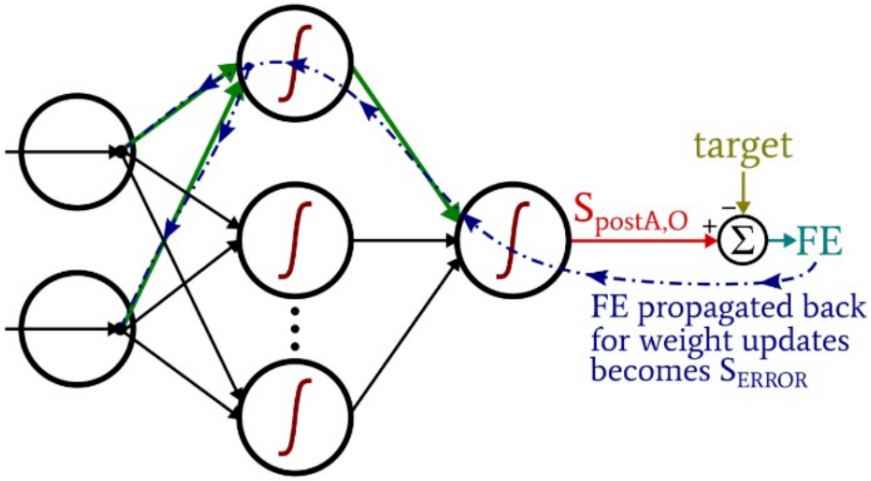


Figure 93: The Message and Aggregation Step can be a Mulit-Layer Perceptron that Increases GNN's Expressivity

An alternative method is to add layers that does not pass messages. This is done by combining different ML methods as **pre-processing** and/or **post-processing** layers.

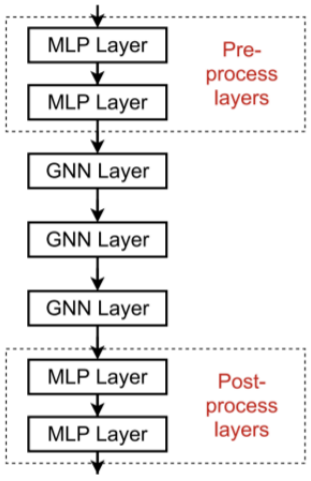


Figure 94: Layering with pre and post processing

1. **Pre-processing** - if there are feature level information that can learned from the nodes (i.e. images or texts), a pre-processing layer can be added to learn features of the nodes itself.
2. **Post-processing** - for graph classification and knowledge graphs, a post-processing layer can be used to classify/transform or reason over the graphs

16.2.3 Skip Connections in Deep GNNs

For certain tasks where **far reaching dependencies** is a key feature of the problem, deep GNNs are required. **Skip connections** is a very useful tool to reduce the problem of oversmoothing.

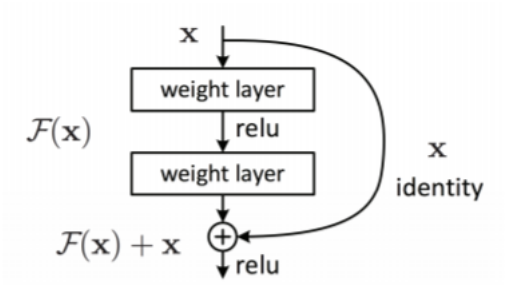


Figure 95: Skip Connections

This effectively add "shortcuts" to the graph, where the computed message and aggregated embedding $\mathcal{F}(\mathbf{x})$ is combined with its identity (unprocessed embedding) $\mathbf{x}$. This allows the previous layer to impact the future node embedding more significantly which is done by a simple linear combination $\mathcal{F}(\mathbf{x}) + \mathbf{x}$.

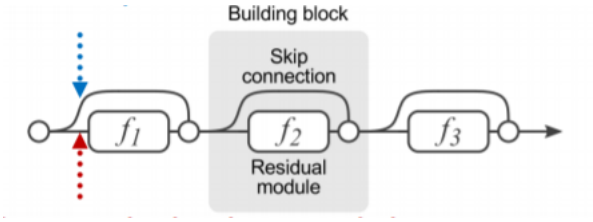


Figure 96: Implementation of Skip Connections

Adding skip connections between each module creates in effect a **mixture model**. Given N skip connections, there are 2^N possible paths that can be created, which gives a mixture of shallow and deep GNNs.

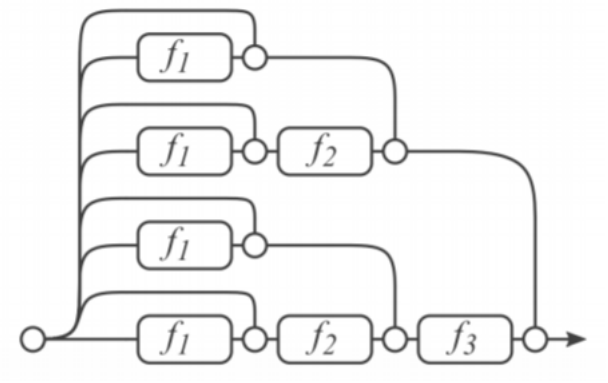


Figure 97: Unraveled View of Skip Connections

A Graph Convolutional Network Layer with skip connections would be

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)}\right)$$

16.3 Graph Augmentations

In many problems, the **raw input graph is not the optimal computation graph** for GNNs. Up to this point, the computation graph that is defined by the nodes are based on raw computation graph.

1. **Lack of Features** - the input graph can lack features
2. Graph is too **sparse** - message passing is inefficient
3. Graph is too **dense** - message passing can be too costly
4. Graph is too **large** - cannot fit the computational graph

16.3.1 Node Level Feature Augmentation

In some problems, we are simply given an adjacency matrix. This creates a problem of the lack of node representations to propagate throughout the network.

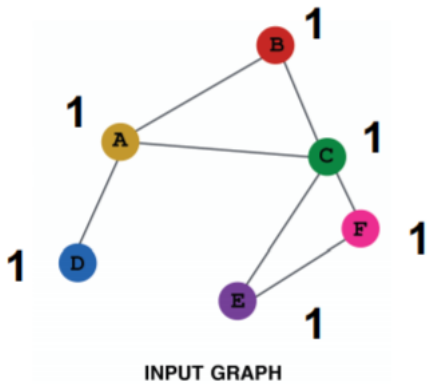


Figure 98: Assigning One to All Nodes

A standard option is to assign the value of one to all nodes, initializing a standard value to all. This allows

1. The **expressivity of such representation is limited**. It captures the graph structure (when the aggregations of values) but fails to capture node-specific values
2. **Generalization is possible** for new nodes, or inductive
3. Representation is of $\in \mathbb{R}$

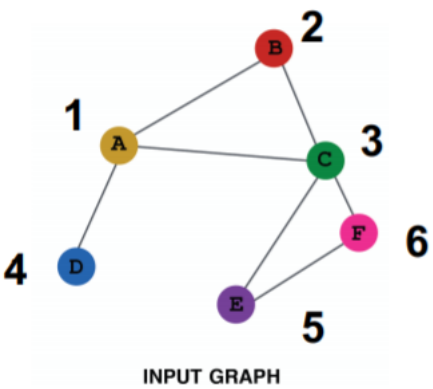


Figure 99: Utilizing One-Hot Encoding for Each Node

An alternative is to use one hot encoding. This is set arbitrarily and is NOT permutation invariant. For a graph of 6 nodes and for node 5, we have the representation

$$[0, 0, 0, 0, 1, 0]$$

This introduces some benefits but also a few problems

1. Given unique values, this allows to **capture node-specific representation**
2. However, due to arbitrary assignment, trained parameters are **inherently non-inductive**. There is no reason why a new node should have the encoding $[0, 0, 0, 0, 0, 0, 1]$ outside of the previous context, and training is hence not transferrable.
3. Dimension of representation grows, which is of $\mathbb{R}^{O(|V|)}$

16.3.2 Structure Level Feature Augmentation

A key problem with GNNs is that it **fails to capture the structure of the graph**

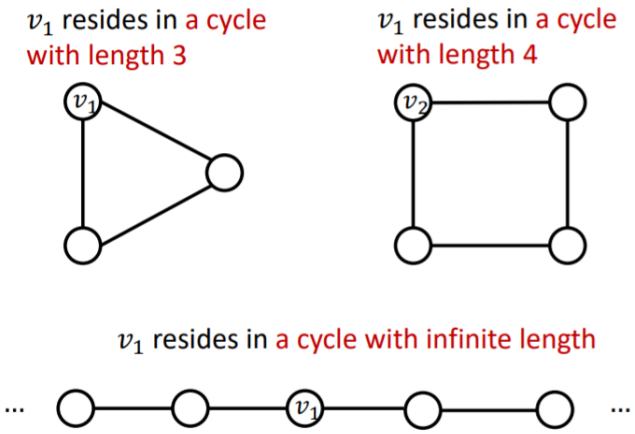


Figure 100: Node v_1 from Different Structures

Consider in Figure 100, we have three distinct graphs. Nevertheless, to a GNN architecture, they are similar to one another.

1. v_1 shares two neighbours
2. All other nodes share two neighbours
3. This happens despite different structures as shown in Figure 101

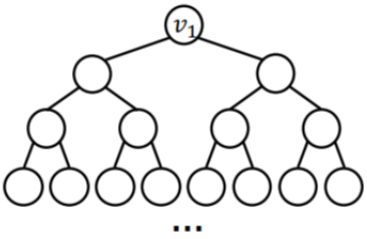


Figure 101: Similar Computation Graph in GNN Construction

A potential solution is to create **structural node level feature**, which includes but not limited to

1. Cycle Count
2. Node Degree
3. Clustering Coefficient
4. PageRank
5. Centrality

16.3.3 Adding Virtual Nodes and Edges

In situations where **sparse graphs is the problem**, adding virtual nodes and edges could enhance sparse representations.

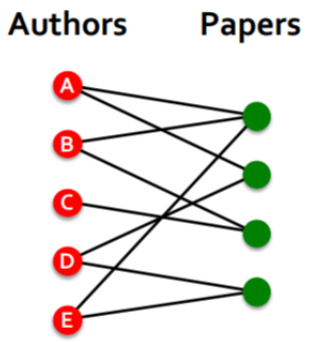


Figure 102: Bipartite Graph of Author-Paper Relationship

Consider the situation in Figure 102, where it is an author-paper network. A common augmentation is be considering $N - hops$ by introducing $\mathbf{A}' = \sum_{n=1}^{N} \mathbf{A}^n$.

Interpretation of $\mathbf{A}^n$ - n represents the number of hops from node i to node j. This represents an interesting relationship for the context in Figure 102 (or any bipartite graph for that matter)

1. 2-hops for "Author" nodes link authors together, forming a co-authorship relationship
2. 2-hops for "Paper" nodes link papers together, forming a sharing-authors relationship

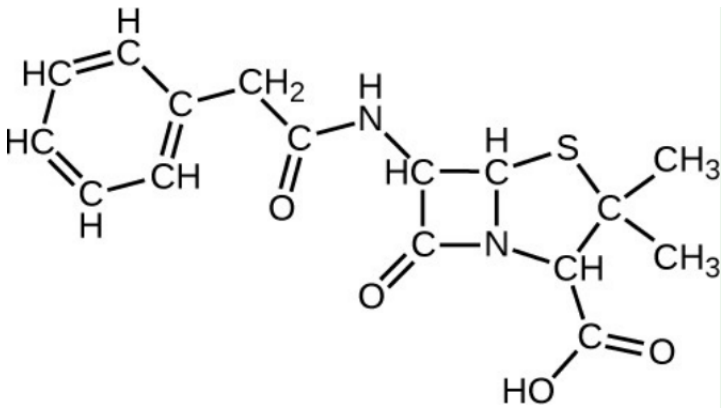


Figure 103: Structure of Penicillin

Consider instead the structure of penicillin in Figure 103 and the task is to classify molecules with similar structure. The dependencies on the whole graph is required for its functionality. This can be problematic if the raw input graph is used.

Figure 104: Adding a Virtual Node as Sink or Source

For nodes with strong dependency requirements with far nodes, a virtual node can be created that connects with the whole structure in Figure 104, which allows effective message passing.

16.3.4 Neighbourhood Sampling

Given a different problem altogether, some graphs are too dense! Sometimes, there are a few nodes that are too dense (which is common in natural graphs with power-law node degrees), computation using the whole input graph can be extremely expensive.

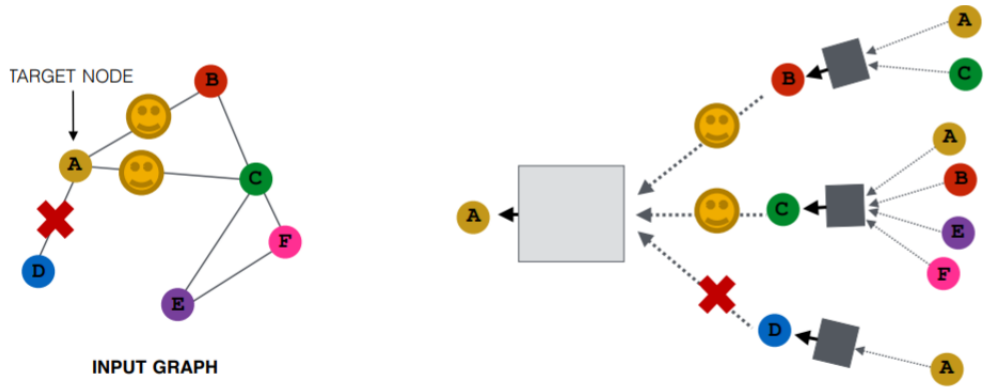


Figure 105: Neighbourhood Sampling

Random sampling can work well in situations like this. However, for some situations, biased sampling for important nodes based on certain graph-level or node-level feature can be helpful.

16.4 Machine Learning Task

In general, GNNs can be trained to predict three prediction "heads". They are

1. Node-level task
2. Edge-level task
3. Graph-level task

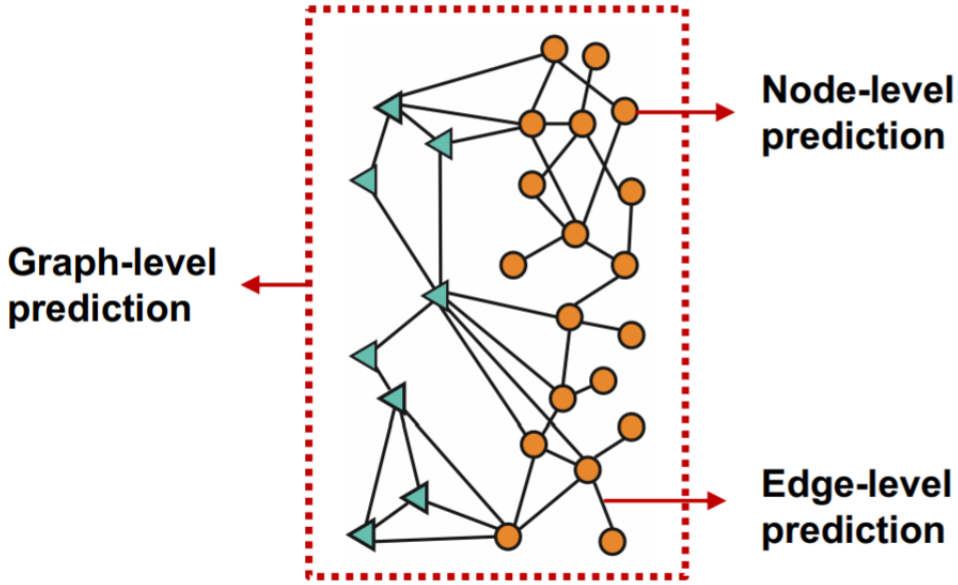


Figure 106: Different Heads of Prediction Tasks

16.4.1 Node-Level Prediction

After the GNN computation, we would obtain the set of node embeddings for all nodes

$$\{\mathbf{h}_v \in \mathbb{R}_d, \forall v \in G\}$$

With this, we could use the node embeddings directly for any classification or regression tasks as needed.

16.4.2 Edge-Level Predictions

Suppose we want to make a k-way prediction for the existence of an edge (k types of edges), or

$$\hat{\mathbf{y}}_{uv} = \texttt{EDGE}(\mathbf{h}_u, \mathbf{h}_v)$$

The first option is to concatenate and use a linear transformation

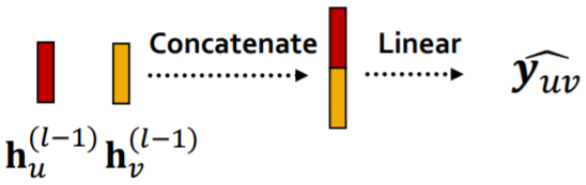


Figure 107: Concatenation and Linear Transformation for Edge Level Prediction

This gives the "edge embedding" denoted as $\hat{\mathbf{y}}_{u \to v} \in \mathbb{R}^{2d}$ and the linear function will map the embedding of the edges. The $\texttt{EDGE}(\cdot)$ function is to predict the existence of the edge, which can be of any functional form or complexity.

$$\hat{\mathbf{y}}_{uv} = \texttt{EDGE}(\texttt{LINEAR}(\texttt{CONCAT}(\mathbf{h}_u, \mathbf{h}_v))) = \texttt{EDGE}\Big(\mathbf{W}^{(l)} \begin{bmatrix} \mathbf{h}_u \\ \mathbf{h}_v \end{bmatrix}\Big)$$

An alternative is to use the `dot product`. This is only applicable for a 1-way prediction task used to predict the existence of a single edge.

$$\hat{\mathbf{y}}_{uv} = \texttt{EDGE}(\mathbf{h}_u^\top \mathbf{h}_v),\ \mathbf{h}_u^\top \mathbf{h}_v \in \mathbb{R}$$

The function $\texttt{EDGE}(\cdot)$ can be as simple as a sigmoid function.

For a k-way prediction, we can utilize the same method of the multi-head attention networks by concatenating the values of the node products

$$\begin{aligned}
\mathbf{h}_{u\to v}^{(1)} &= \mathbf{h}_u^\top \mathbf{W}^{(1)} \mathbf{h}_v \\
&\dots \\
\mathbf{h}_{u\to v}^{(k)} &= \mathbf{h}_u^\top \mathbf{W}^{(k)} \mathbf{h}_v \\
\hat{\mathbf{y}}_{uv} &= \texttt{EDGE}(\texttt{CONCAT}(\mathbf{h}_{u\to v}^{(1)}, \dots, \mathbf{h}_{u\to v}^{(k)}))
\end{aligned}$$

16.4.3 Graph Level Prediction

For graph level predictions, the main idea is to be able to obtain a form of graph embedding that represents the graphs concerned.

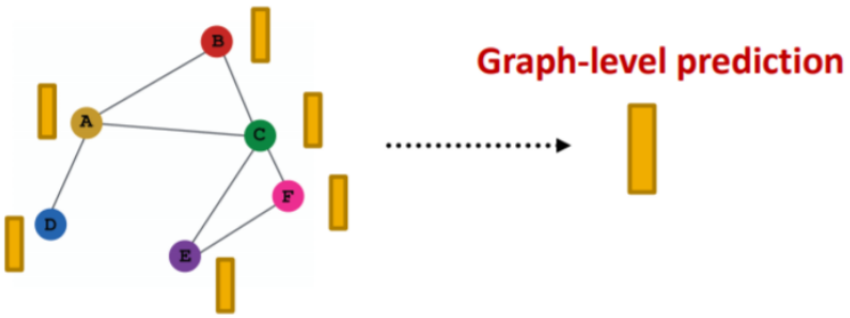


Figure 108: Graph Level Prediction

The task is similar to the one of **aggregation** of nodes. Some of the obvious options are

1. Min pooling - $\texttt{MIN}(\{\mathbf{h}_v \in \mathbb{R}_d, \forall v \in G\})$
2. Max pooling - $\texttt{MAX}(\{\mathbf{h}_v \in \mathbb{R}_d, \forall v \in G\})$
3. Mean pooling - $\texttt{MEAN}(\{\mathbf{h}_v \in \mathbb{R}_d, \forall v \in G\})$

However, this leads to the issue of information loss, which leads to deterioration of performance especially for large graphs. Consider the following example using mean pooling ,

$$G_1 = \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \\ \mathbf{h}_4 \\ \mathbf{h}_5 \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \\ 0 \\ 1 \\ 2 \end{bmatrix} \underset{mean}{\to} 0$$

$$G_2 = \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \\ \mathbf{h}_4 \\ \mathbf{h}_5 \end{bmatrix} = \begin{bmatrix} -10 \\ -20 \\ 0 \\ 10 \\ 20 \end{bmatrix} \underset{mean}{\to} 0$$

This leads to indistinguishable graphs based on mean pooling even though the two graphs are extremely different.

The alternative is using hierarchical strategy for pooling. The aggregation is applied hierarchically and applying a non linear function $\sigma(\cdot)$.

For G_1 from above

1. Aggregation by splitting into groups a and b
   $\mathbf{h}_a = \texttt{ReLU}(\texttt{SUM}(-1, -2)) = 0$ and $\mathbf{h}_b = \texttt{ReLU}(\texttt{SUM}(0, 1, 2)) = 3$
2. $\mathbf{h}_{G_1} = \texttt{ReLU}(\texttt{SUM}(\mathbf{h}_a, \mathbf{h}_b)) = 3$

Repeating for G_2, we get $\mathbf{h}_{G_2} = 30$, which allows the differentiation between the two graphs.

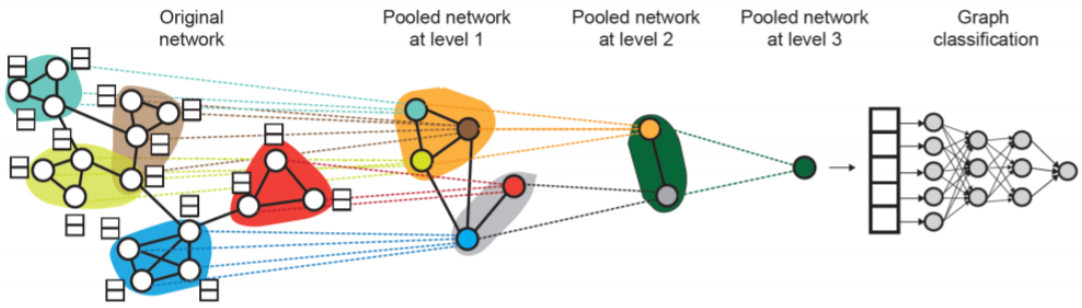


Figure 109: DiffPool Architecture as an Implementation of Hierarchical Pooling

An example of hierarchical pooling is implemented using DiffPool (`Ying et. al, 2018`).

1. Leverages two GNNs, one computing embeddings and another clusters
2. Pooling hierarchy are based on clusters
3. The two GNNs can be implemented in parallel

17 Theoretical Basis of Graph Neural Networks

With the proliferation of different kinds of architectures for Graph Neural Networks, it can be daunting to decide on which architecture to use. In this section, we want to

1. Mathematically show the **expressivity** of different architectures, that is, how well can the GNN differentiate different graph structures
2. **Derive a maximally expressive** GNN with current framework

17.1 Differentiating Nodes

The key task of a GNN is to be able to differentiate two nodes with different features given the graph structure.

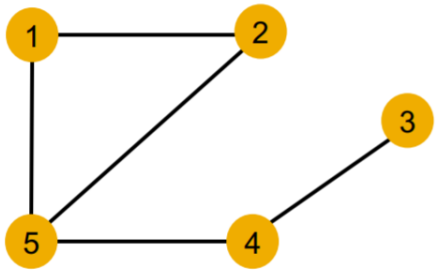


Figure 110: Graph with Nodes of the Same Feature

Consider the graph in Figure 110, we colour all the nodes of the same color since it has the same node features. In this case, a classifier would NOT be able to distinguish the nodes 1 to 5.

17.1.1 Local Neighbourhood Structure

A key part of Machine Learning with graphs is to leverage the existence of **local neighbourhood structures** and relationships with one another.

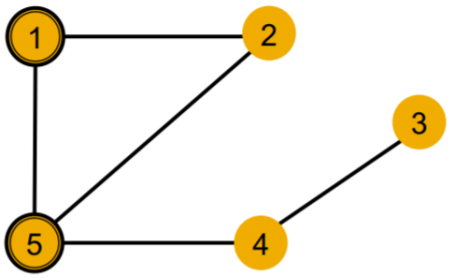


Figure 111: Closer Look on Neighbourhood Structure of 1 and 5

Based on Figure 111, we can distinguish nodes 1 and 5 by their local neighbourhood structure by counting the node degrees

1. Node 1 has 2 neighbours
2. Node 5 has 3 neighbours

There is no reason to stop there. After all, the power of GNNs come from its power to leverage neighbourhood structures and we can extend to the neighbours of neighbours.

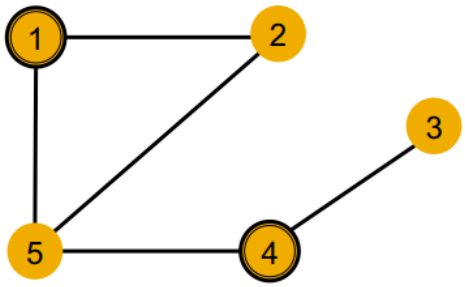


Figure 112: Closer Look on Neighbourhood Structure of 1 and 4

1. We can differentiate nodes 1 and 4
2. Node 1 has neighbours of degrees 2 and 3
3. Node 2 has neighbours of degrees 1 and 3

17.1.2 Symmetric Nodes

A place where GNN fails when it comes down to symmetric nodes. This is the case for nodes 1 and 2 in the graph we have in Figure 112.

1. Nodes 1 and 2 has neighbours of degrees 2 and 3
2. Two hops would lead to the same problem

In this situation, we cannot use GNNs to differentiate the graphs! In order to detect such issues we have to examine the **computational graph** of each node.

17.1.3 Computational Graphs

A key consideration when designing a GNN is to consider the computational graph from the perspective of the GNN. Though we have different node IDs for the nodes, given the same features, GNNs are not able to differentiate them.

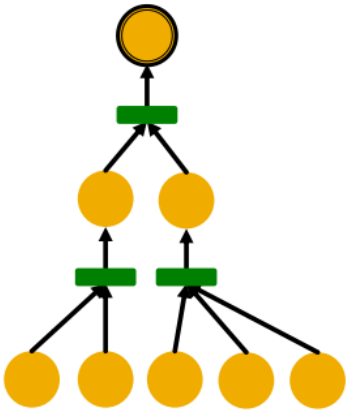


Figure 113: Computational Graph of 1 and 2

For a depth of 2, this would be the graph for both nodes 1 and 2. This effectively means that there is no way to actually differentiate either nodes from one another.

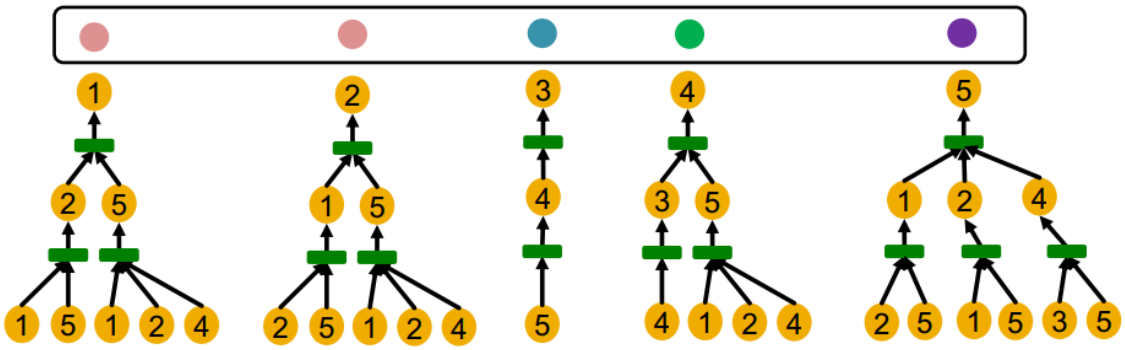


Figure 114: All Computational Graphs of the Nodes

17.2 Injective Functions

The computation graph of a node is effectively a function that maps the nodes' structure to an embedding space. In order to mathematically analyze the expressivity of a GNN, we must consider the **injectivity of the function**.

$f : X \to Y$ is *injective* if it maps **different values to different outputs**. This means that **there is no information loss** in the mapping process.

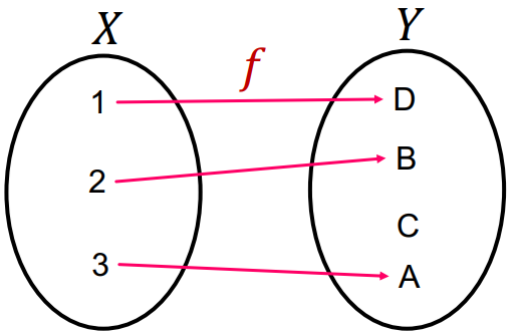


Figure 115: Injective Function

Theorem: A GNN can distinguish different subtree structures **if every step of the neighbourhood aggregation function is injective**

This logically should make sense. Given that the example in Figure 112 shows an example of **a non-injective aggregation function**, we fail to differentiate nodes 1 and 2. Designing a **provable injective function** would allow us to design the **most expressive GNN** with the current framework.

17.3 Neighbourhood Aggregations of Popular Architectures

Neighbourhood Aggregation is equivalent to a function on a **multiset**, which is a set that allows repeating elements. Analyzing two popular architecture would allow us to understand GNNs in more depth

1. **GCN** - mean pooling
2. **GraphSAGE** - max pooling

17.3.1 GCN and Mean Pooling

GCN uses an element wise mean pooling function and a ReLU activation function.

$$\texttt{AGG}(x) = \texttt{ReLU}(\texttt{MAX}(x))$$

An obvious degenerate case would be when the multisets **consist of the same proportion of values**.

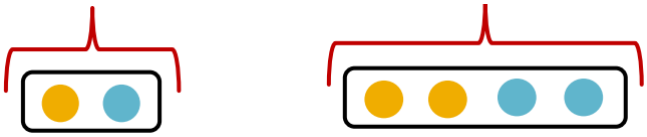


Figure 116: Degenerate Case for GCN

17.3.2 GraphSAGE and Max Pooling

The GraphSAGE architecture uses a MLP and perform element-wise max pooling after.

$$\texttt{AGG}(x) = \texttt{MAX}(\texttt{MLP}(x))$$

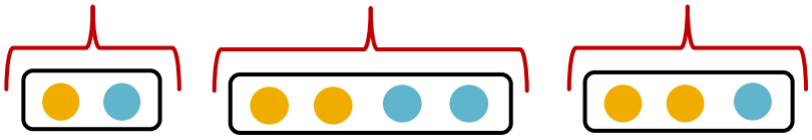


Figure 117: Degenerate Case for GraphSAGE

While not obvious, the function **cannot distinguish multisets with the same set of distinct colors**.

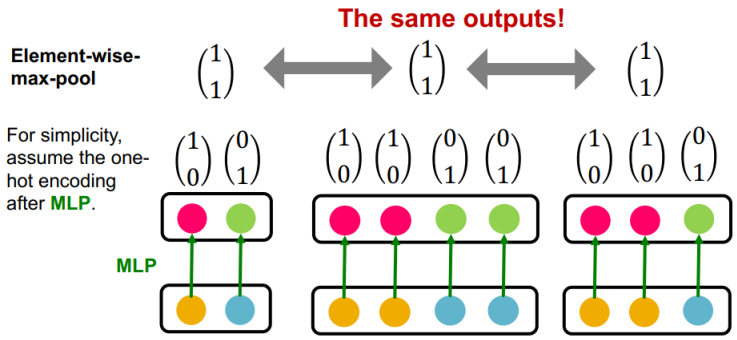


Figure 118: Failure Case for GraphSAGE

17.4 Injective Multi-Set Function

Any injective function can be expressed as the following

$$\Phi\Big(\sum_{s\in S} f(s)\Big)$$

1. $\Phi(\cdot)$ is a non-linear function
2. $\sum_{s\in S}$ sums over the multiset
3. $f(\cdot)$ is another non-linear function

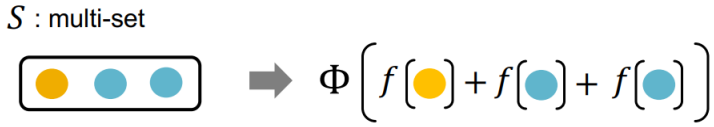


Figure 119: Multiset Injective Function

Theorem: summation retains information unlike other aggregation functions.

17.4.1 Universal Approximation Theorem

Theorem: 1-hidden-layer `MLP` with sufficiently-large hidden dimensionality and appropriate non-linearity $\sigma(\cdot)$ (including `ReLU` and sigmoid) can *approximate any continuous function to an arbitrary accuracy.* (Hornik et al., 1989)

This means that if we define $\Phi(\cdot)$ and $f()\dot{}$ as a single hidden MLP, we can obtain the most expressive GNN (with dimensions ranging from 50-100 in practice). This is the aggregation function used by the **Graph Isomorphic Network (GIN)**

$$\texttt{MLP}_\Phi\Big(\sum_{s\in S} \texttt{MLP}_f(s)\Big)$$

17.5 Ranking Pooling Methods

The following shows a summary of the expressivity of pooling methods and cases when it fails.

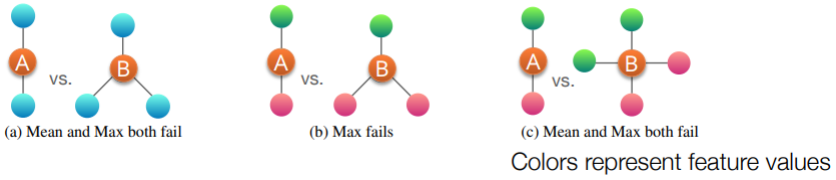


Figure 120: Failure Cases

As for ranking,

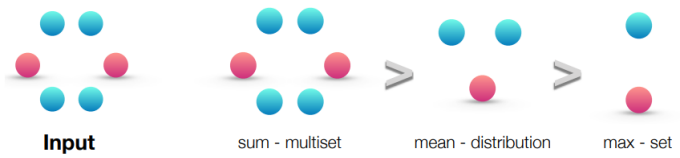


Figure 121: Ranking of Pooling Methods

18 Heterogeneous Graphs and Relational Architectures

Up to this point, most graphs that are discussed are assumed to be **homogeneous**. That means that the nodes and edges are of the same class (or label). Relaxing this assumption, we are able to express data in more complex and expressive forms by having a **heterogeneous** graph.

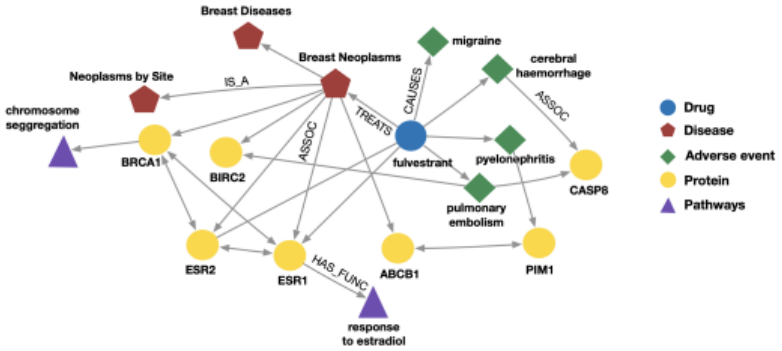


Figure 122: Example of a Heterogeneous Graph in the Biomedical Field

18.1 Relational Graph Convolution Networks

How do we then learn such graphs? The most simple form of a graph neural network for heterogeneous graphs is the `Relational Graph Convolution Network`.

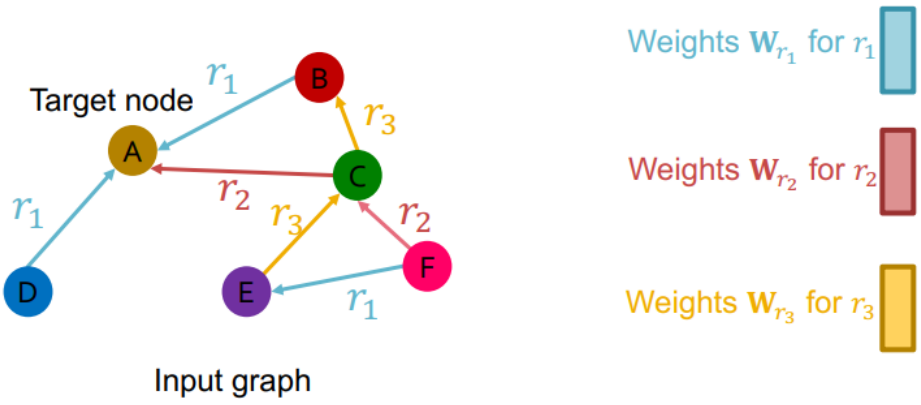


Figure 123: Having Multiple Relations to Distinguish Relations

Based on Figure 123

1. Assuming homogeneous node, we have three distinct types of edges are r_1, r_2, r_3
2. Each relation has weights $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$

The node embeddings are hence defined as

$$\mathbf{h}_v^{(l+1)} = \sigma\Big(\sum_{r\in R}\sum_{u\in N(v)_r}\frac{1}{c_{v,r}}\mathbf{W}_r^{(l)}\mathbf{h}_u^{(l)} + \mathbf{W}_0^{(l)}\mathbf{h}_v^{(l)}\Big)$$

1. $\frac{1}{c_{v,r}}$ is the normalized node degree by relation
2. $\frac{1}{c_{v,r}}\mathbf{W}_r^{(l)}\mathbf{h}_u^{(l)}$ is the message passing from each neighbour given relation
3. $\mathbf{W}_0^{(l)}\mathbf{h}_v^{(l)}$ is the self-loop (to prevent information loss)

18.2 Issues of Scalability

Scalability is a problematic issue for **RGCNs** and similar architectures. This is obvious when we look at `Knowledge Graphs`.

For each relation $r \in R$, we have

$$\mathbf{W}_r^{(1)}, \mathbf{W}_r^{(2)} \dots \mathbf{W}_r^{(L)}. \forall r \in R$$

The parameters grow by $O(r)$. This presents a problematic issue for the **scalability** and **overfitting** for **RGCNs**.

18.3 Regularization and Scaling Methods

Two common methods to scale and regularize weights are

- Block Diagonal Matrices
- Basis Learning

18.3.1 Block Diagonal Matrices

Using block diagonal matrices, whereby the square diagonal of the matrix can be any values and the off-diagonal values are zero. By making the weights sparse, we are able to reduce the size of the parameters by a factor of

$$\frac{\texttt{Number of Elements in Block Diagonal}}{\texttt{Number of Elements in Matrix}}$$

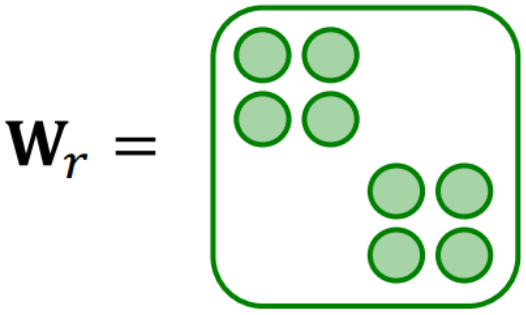


Figure 124: Block Diagonal Matrix

This has the issue of limiting nearby dimensions through $\mathbf{W}_r$.

18.3.2 Basis Learning

Also known as **Dictionary Learning**, this method allows weight sharing by defining the parameters in terms of its basis

$$\mathbf{W}_r = \sum_{b=1}^{B} a_{r,b} \cdot \mathbf{V}_b$$

1. $\mathbf{V}_b$ is the basis matrices which is shared across all relations
2. $a_{r,b}$ is the importance