Graphs

Data Structures & Algorithms

Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: http://cs224w.Stanford.edu

Stanford CS224W: Machine Learning with Graphs Fall 2025/26

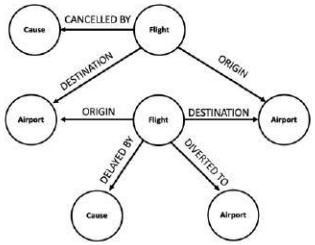
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
Charilaos Kanatsoulis, Stanford University

http://cs224w.stanford.edu

Why Graphs?

Graphs are a general language for describing and analyzing entities with relations/interactions

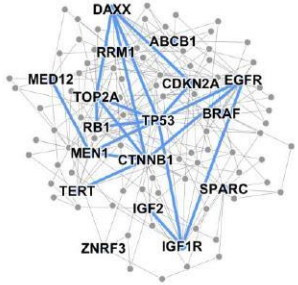
Many Types of Data are Graphs (1)



Event Graphs

Image credit: SalientNetworks

Computer Networks



Disease Pathways

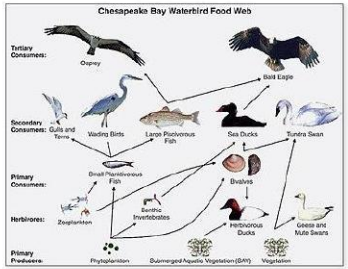


Image credit: Wikipedia

Food Webs

Image credit: Pinterest

Particle Networks

Image credit: visitlondon.com

Underground Networks

Many Types of Data are Graphs (2)

Image credit: Medium

Social Networks

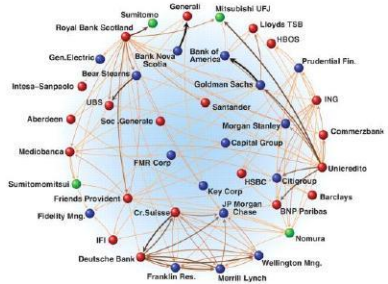
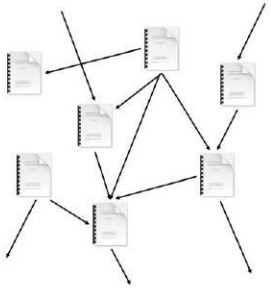


Image credit: Science

Economic Networks

Image credit: Lumen Learning

Communication Networks



Citation Networks

Image credit: Missoula Current News

Internet

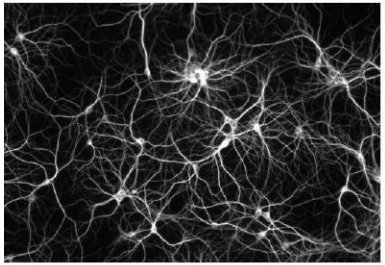
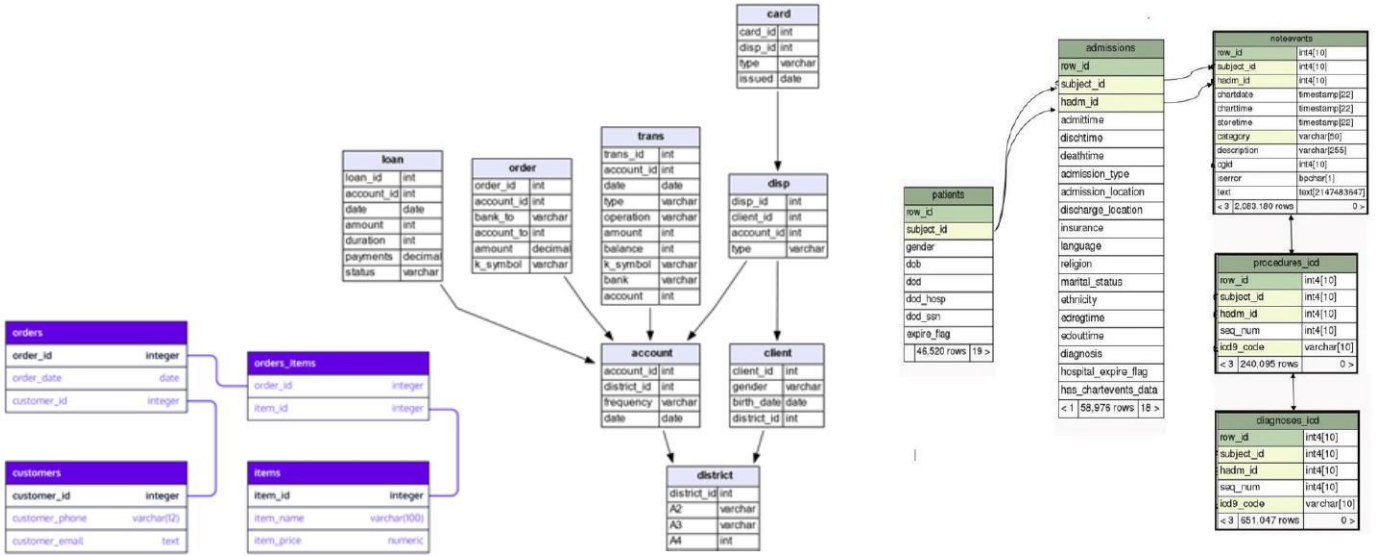


Image credit: The Conversation

Networks of Neurons

Databases are Graphs!

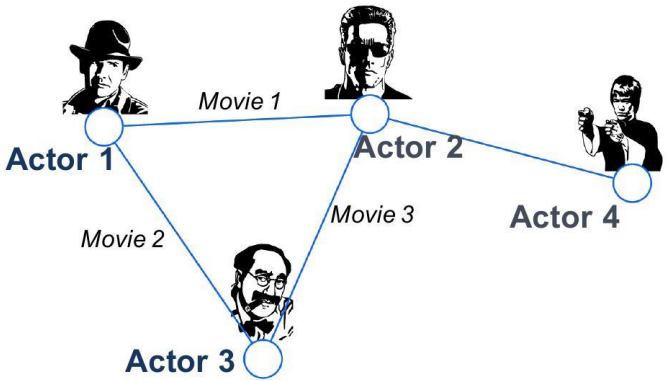


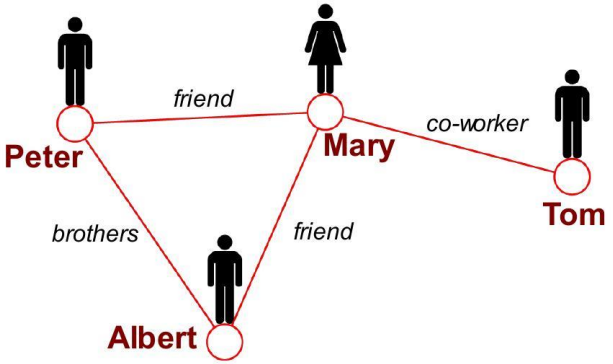
28

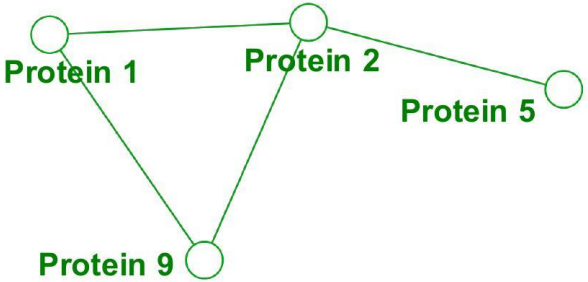
Choice of Graph Representations

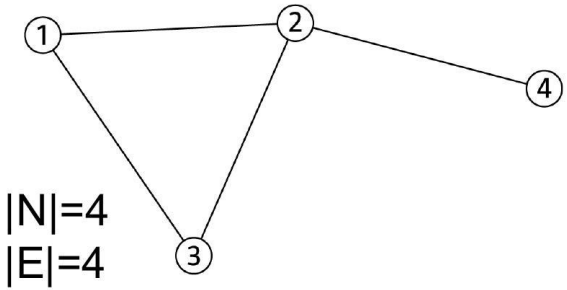
How structure shapes learning and performance

Graphs: A Common Language

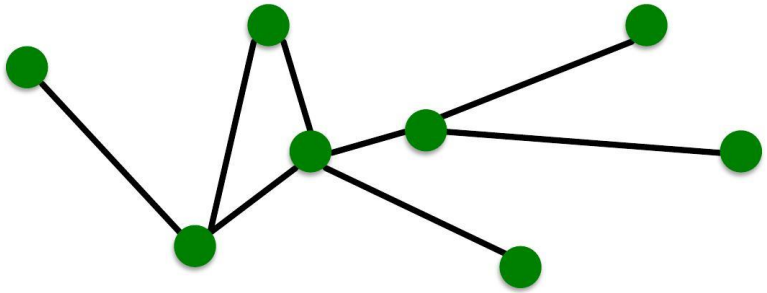








Components of a Network



- **Objects:** nodes, vertices — N
- **Interactions:** links, edges — E
- **System:** network, graph — $G(N,E)$

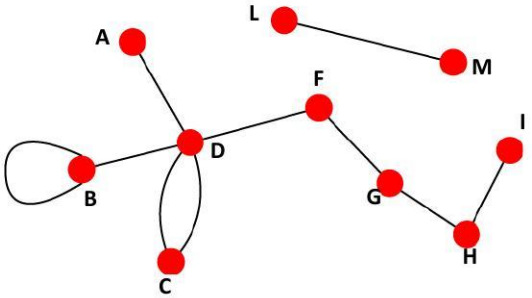
Choosing a Proper Representation

- **How to build a graph:**
  - **What are nodes?**
  - **What are edges?**
- **Choice of the proper network representation of a given domain/problem determines our ability to use networks successfully:**
  - In some cases, there is a unique, unambiguous representation
  - In other cases, the representation is by no means unique
  - The way you assign links will determine the nature of the question you can study

Directed vs. Undirected Graphs

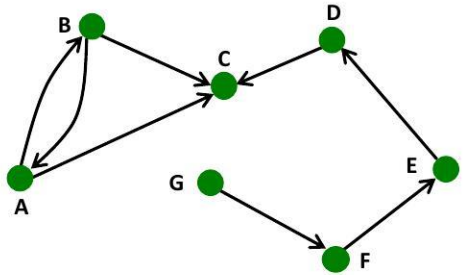
Undirected

- **Links:** undirected (symmetrical, reciprocal)



Directed

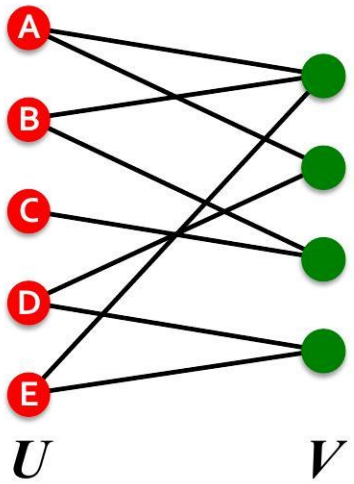
- **Links:** directed



- **Other considerations:**
  - Weights
  - Properties
  - Types
  - Attributes

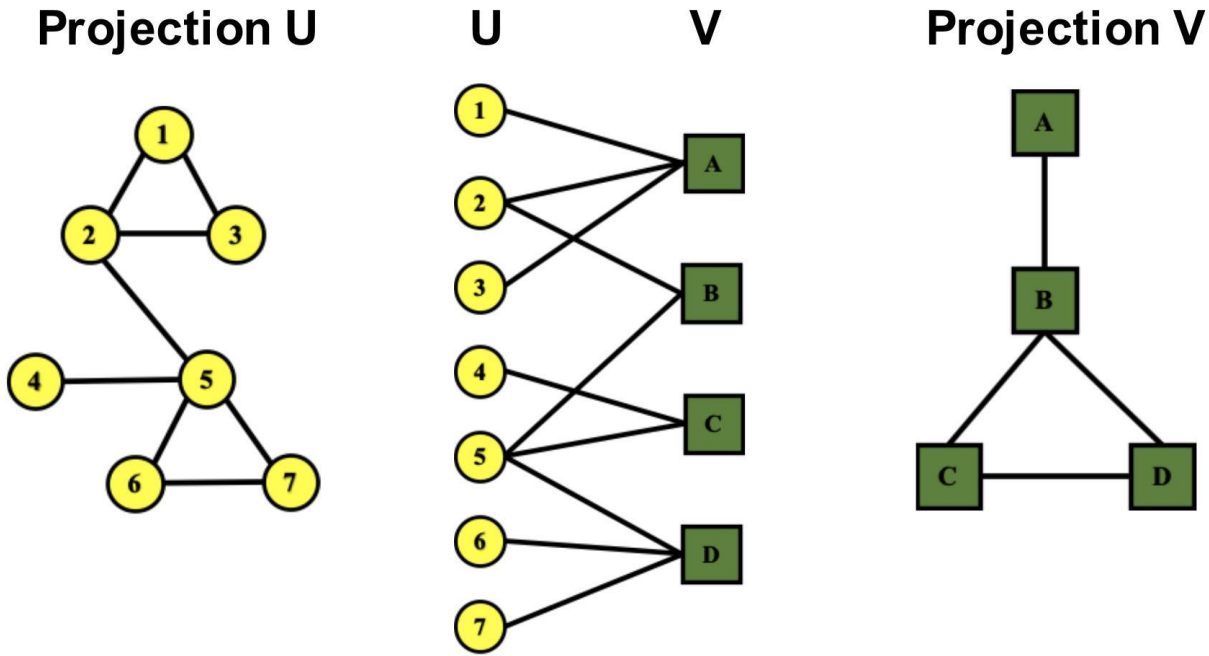
Bipartite Graph

- **Bipartite graph** is a graph whose nodes can be divided into two disjoint sets ***U*** and ***V*** such that every link connects a node in ***U*** to one in ***V***; that is, ***U*** and ***V*** are **independent sets**



- **Examples:**
  - Authors-to-Papers (they authored)
  - Actors-to-Movies (they appeared in)
  - Users-to-Movies (they rated)
  - Recipes-to-Ingredients (they contain)
- **"Folded" networks:**
  - Author collaboration networks
  - Movie co-rating networks

Folded/Projected Bipartite Graphs



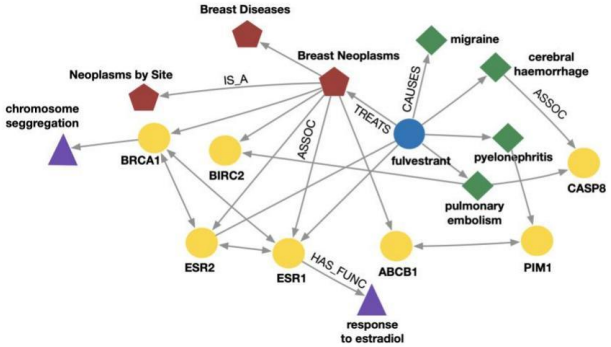
Heterogeneous Graphs

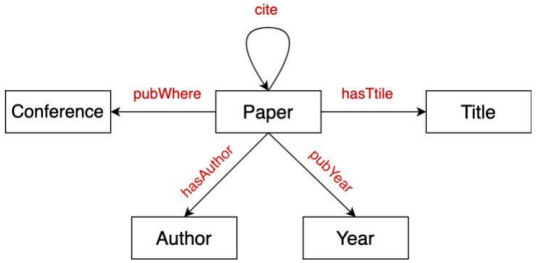
- **A heterogeneous graph is defined as**

$$\boldsymbol{G = (V, E, R, T)}$$

  - Nodes with node types $v_i \in V$
  - Edges with relation types $(v_i, r, v_j) \in E$
  - Node type $T(v_i)$
  - Relation type $r \in R$

Many Graphs are Heterogeneous Graphs





Biomedical Knowledge Graphs

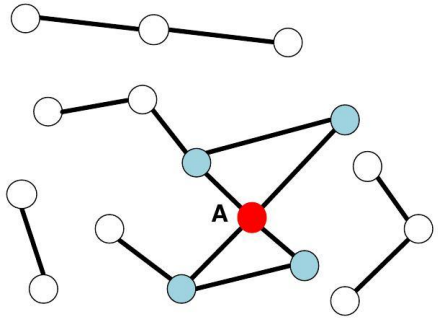
Example node: Migraine
Example edge: (fulvestrant, Treats, Breast Neoplasms)
Example node type: Protein
Example edge type (relation): Causes

Academic Graphs

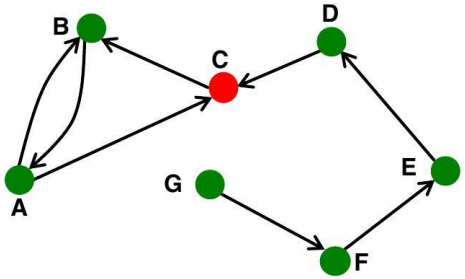
Example node: ICML
Example edge: (GraphSAGE, NeurIPS)
Example node type: Author
Example edge type (relation): pubYear

Node Degrees

Undirected



Directed



Source: Node with $k^{in} = 0$
Sink: Node with $k^{out} = 0$

Node degree, k_i: the number of edges adjacent to node i

$k_A = 4$

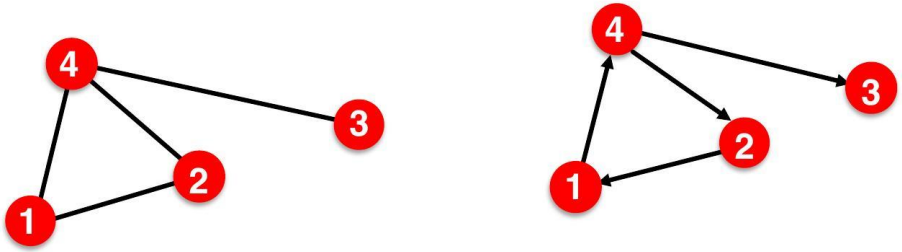
Avg. degree: $\bar{k} = \langle k \rangle = (1/N) \Sigma k_i = 2E/N$

In directed networks we define an **in-degree** and **out-degree.**
The (total) degree of a node is the sum of in- and out-degrees.

$k_C^{in} = 2 \quad k_C^{out} = 1 \quad k_C = 3$

$\bar{k} = E/N \qquad \bar{k}^{in} = \bar{k}^{out}$

Representing Graphs: Adjacency Matrix



$A_{ij} = 1$ if there is a link from node i to node j

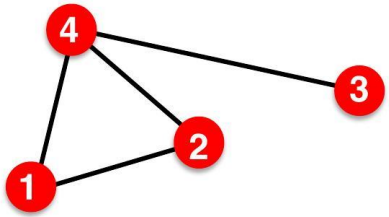
$A_{ij} = 0$ otherwise

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Note that for a directed graph (right) the matrix is not symmetric.

Adjacency Matrix

Undirected



$$A_{ij} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

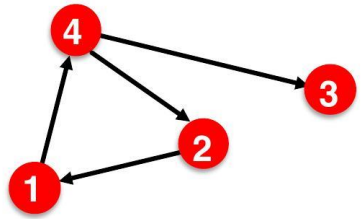
$$A_{ij} = A_{ji}$$
$$A_{ii} = 0$$

$$k_i = \sum_{j=1}^{N} A_{ij}$$

$$k_j = \sum_{i=1}^{N} A_{ij}$$

$$L = \frac{1}{2}\sum_{i=1}^{N} k_i = \frac{1}{2}\sum_{ij} A_{ij}$$

Directed



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

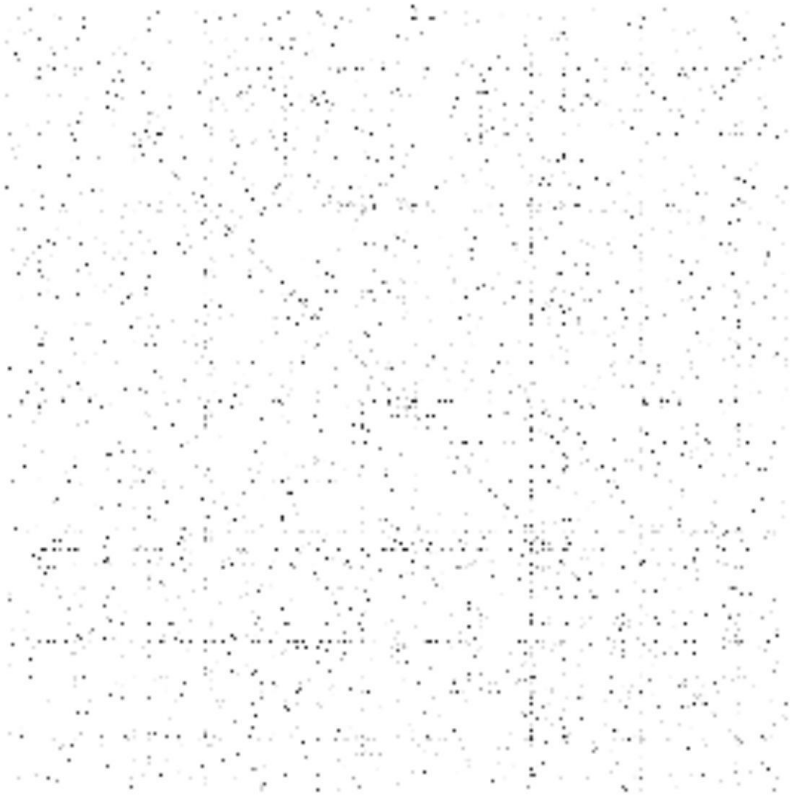
$$A_{ij} \neq A_{ji}$$
$$A_{ii} = 0$$

$$k_i^{out} = \sum_{j=1}^{N} A_{ij}$$

$$k_j^{in} = \sum_{i=1}^{N} A_{ij}$$

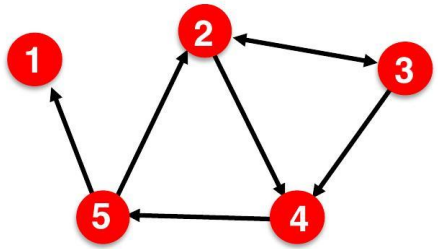
$$L = \sum_{i=1}^{N} k_i^{in} = \sum_{j=1}^{N} k_j^{out} = \sum_{i,j} A_{ij}$$

Adjacency Matrices are Sparse



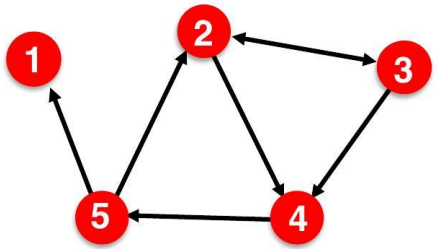
Representing Graphs: Edge list

- **Represent graph as a list of edges:**
  - (2, 3)
  - (2, 4)
  - (3, 2)
  - (3, 4)
  - (4, 5)
  - (5, 2)
  - (5, 1)



Representing Graphs: Adjacency list

- **Adjacency list:**
  - Easier to work with if network is
    - Large
    - Sparse
  - Allows us to quickly retrieve all neighbors of a given node
    - 1:
    - 2: 3, 4
    - 3: 2, 4
    - 4: 5
    - 5: 1, 2



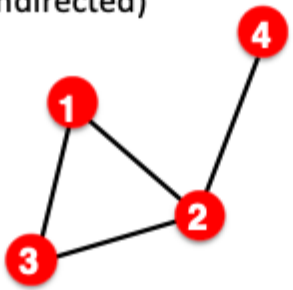
Node and Edge Attributes

Possible options:

- Weight (*e.g.*, frequency of communication)
- Ranking (best friend, second best friend…)
- Type (friend, relative, co-worker)
- Sign: Friend vs. Foe, Trust vs. Distrust
- Properties depending on the structure of the rest of the graph: Number of common friends

More Types of Graphs

- **Unweighted**
  (undirected)



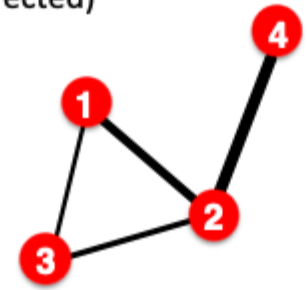
$$A_{ij} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0 \qquad A_{ij} = A_{ji}$$

$$E = \frac{1}{2}\sum_{i,j=1}^{N} A_{ij} \qquad \bar{k} = \frac{2E}{N}$$

Examples: Friendship, Hyperlink

- **Weighted**
  (undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 0.5 & 0 \\ 2 & 0 & 1 & 4 \\ 0.5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0 \qquad A_{ij} = A_{ji}$$

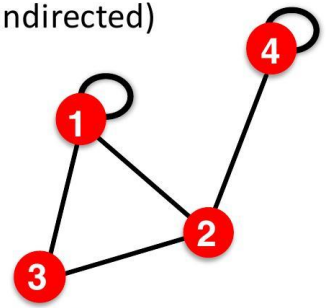
$$E = \frac{1}{2}\sum_{i,j=1}^{N} nonzero(A_{ij}) \qquad \bar{k} = \frac{2E}{N}$$

Examples: Collaboration, Internet, Roads

More Types of Graphs

- **Self-edges (self-loops)**

(undirected)



$$A_{ij} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

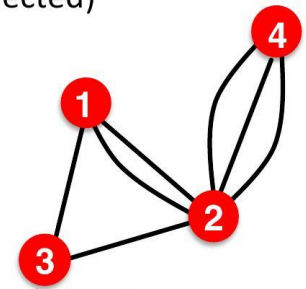
$A_{ii} \neq 0$ $\quad A_{ij} = A_{ji}$

$$E = \frac{1}{2}\sum_{i,j=1,i\neq j}^{N} A_{ij} + \sum_{i=1}^{N} A_{ii}$$

Examples: Proteins, Hyperlinks

- **Multigraph**

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 3 \\ 1 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

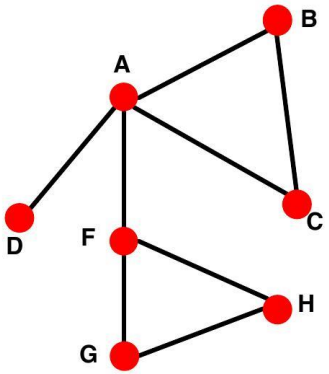
$A_{ii} = 0$ $\quad A_{ij} = A_{ji}$

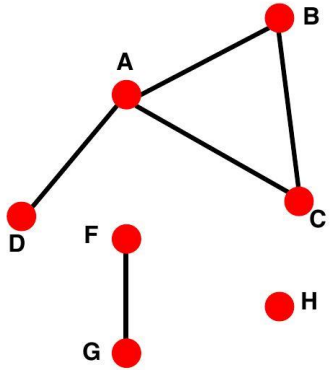
$$E = \frac{1}{2}\sum_{i,j=1}^{N} nonzero(A_{ij}) \qquad \bar{k} = \frac{2E}{N}$$

Examples: Communication, Collaboration

Connectivity of Undirected Graphs

- **Connected (undirected) graph:**
  - Any two vertices can be joined by a path
- A disconnected graph is made up by two or more connected components





Largest Component:
Giant Component

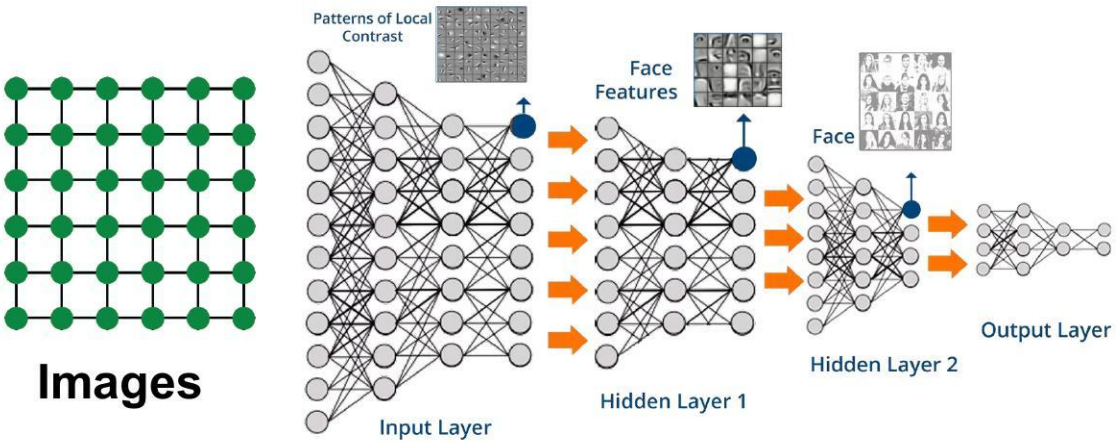
Isolated node (node H)

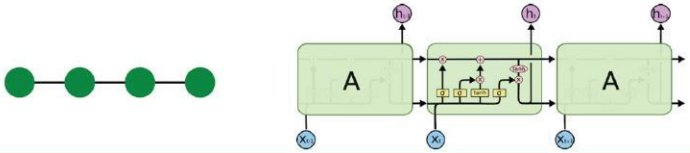
18

ML Tasks in Graphs

Node, edge, and graph-level prediction

Today: Modern ML Toolbox



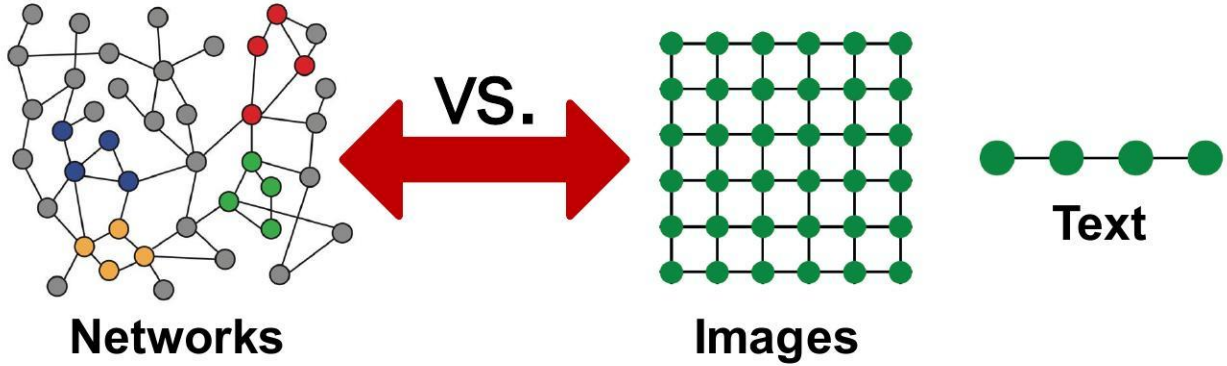


Modern deep learning toolbox is designed for simple sequences & grids

Why is Graph Deep Learning Hard?

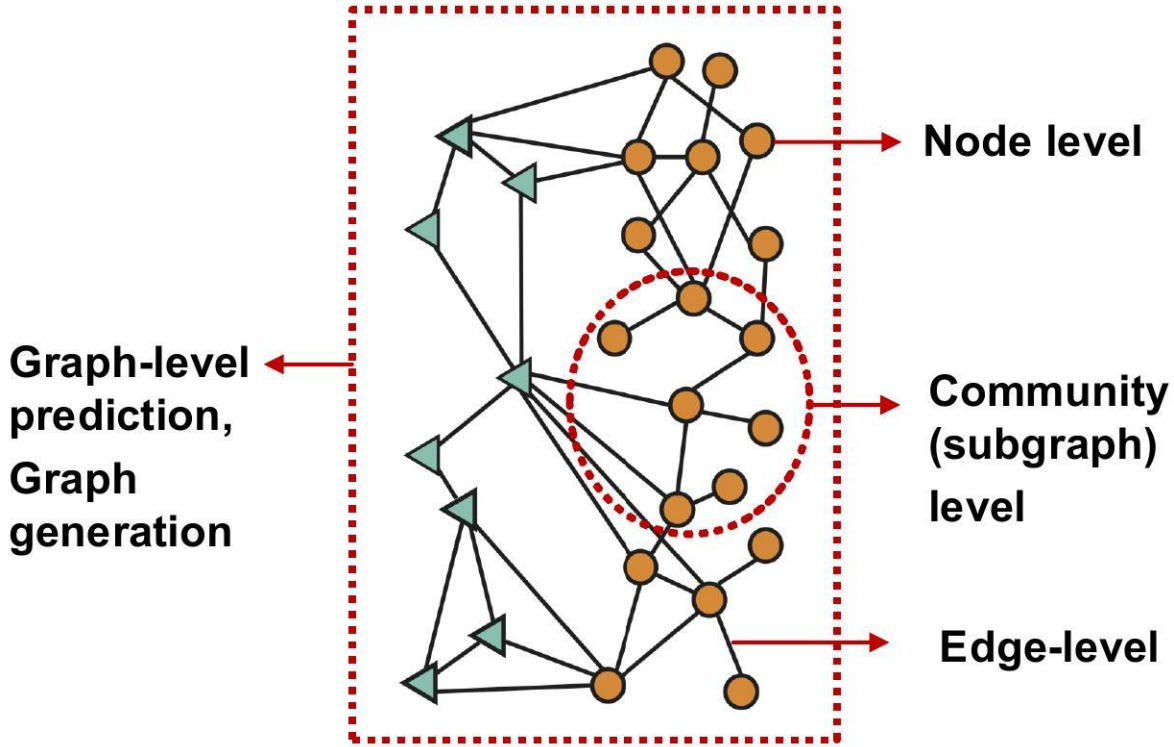
Networks are complex.

- Arbitrary size and complex topological structure (*i.e.*, no spatial locality like grids)



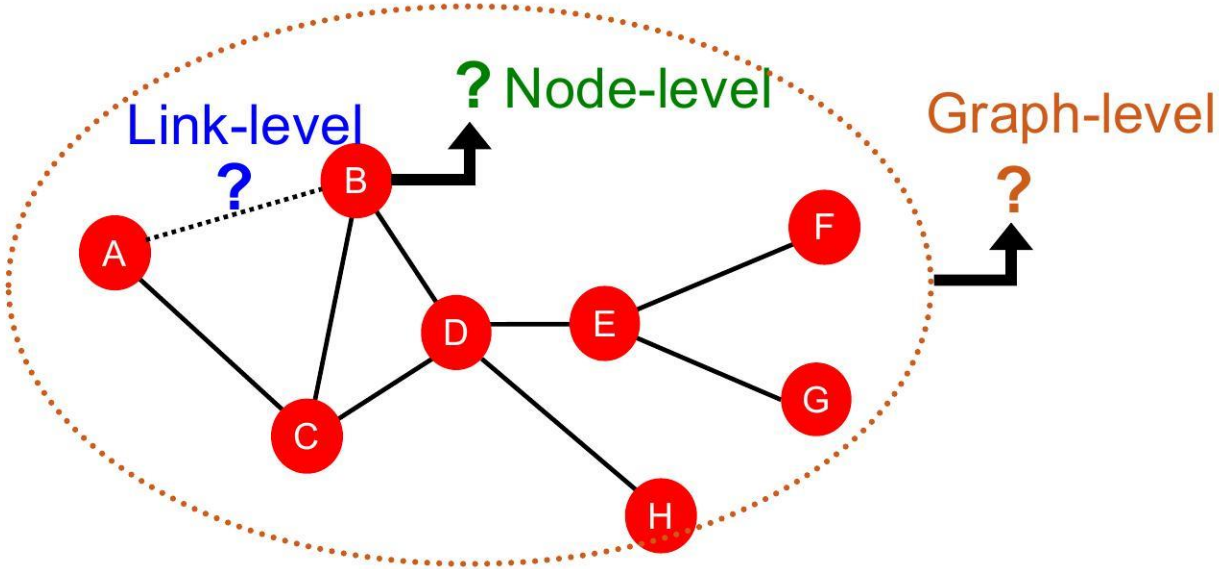
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Different Types of Tasks

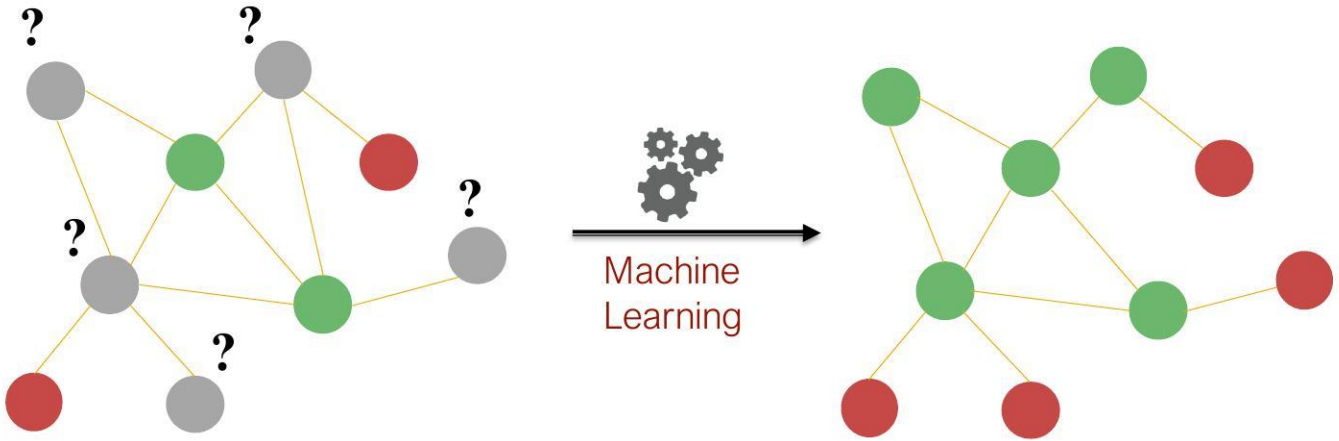


Machine Learning Tasks: Review

- Node-level prediction
- Link-level prediction
- Graph-level prediction



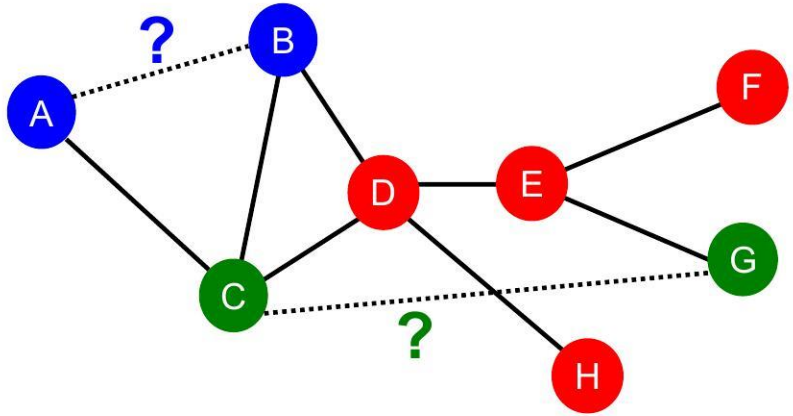
Node-Level Tasks



Node classification

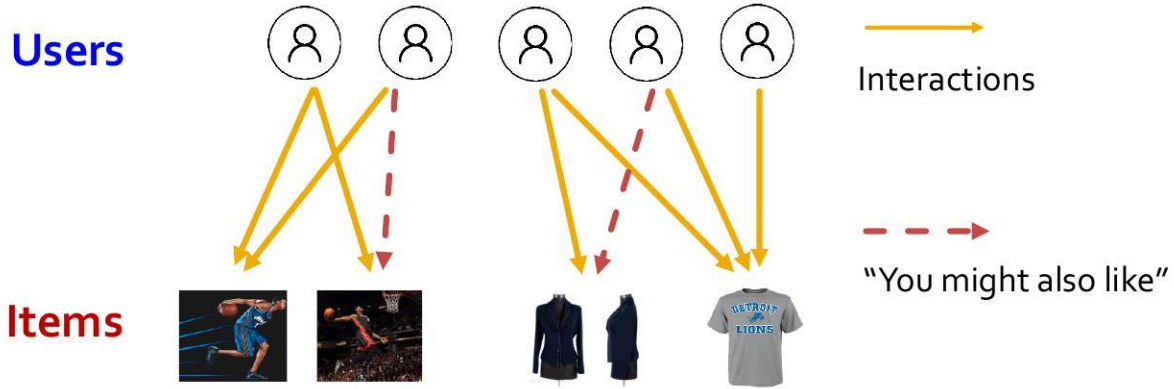
Link-Level Prediction Task

- The task is to predict **new/missing/unknown links** based on the existing links.
- At test time, node pairs (with no existing links) are ranked, and top K node pairs are predicted.
- Task: Make a prediction for **a pair of nodes**.



Example (1): Recommender Systems

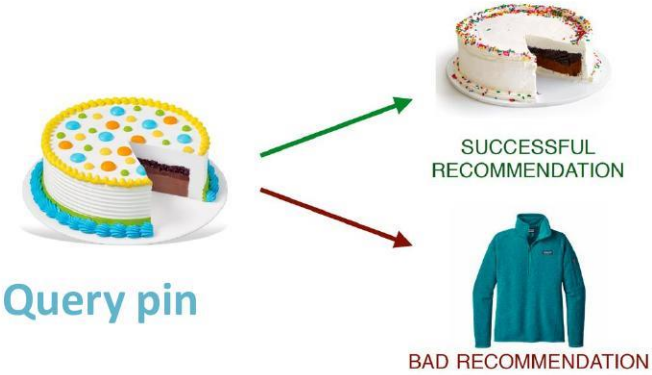
- **Users interacts with items**
  - Watch movies, buy merchandise, listen to music
  - **Nodes:** Users and items
  - **Edges:** User-item interactions
- **Goal: Recommend items users might like**



Ying et al., Graph Convolutional Neural Networks for Web-Scale Recommender Systems, KDD 2018

PinSage: Graph-based Recommender

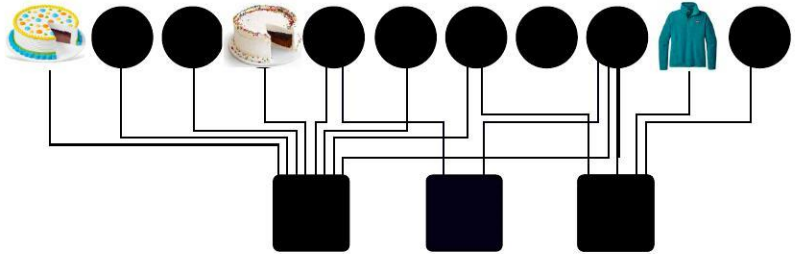
Task: Recommend related pins to users

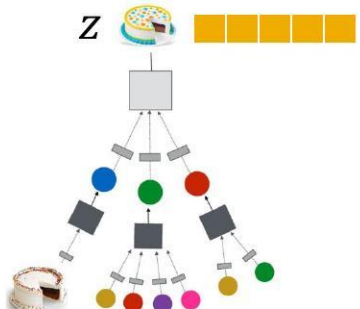


Task: Learn node embeddings z_i such that

$$d(z_{cake1}, z_{cake2}) < d(z_{cake1}, z_{sweater})$$

Predict whether two nodes in a graph are related

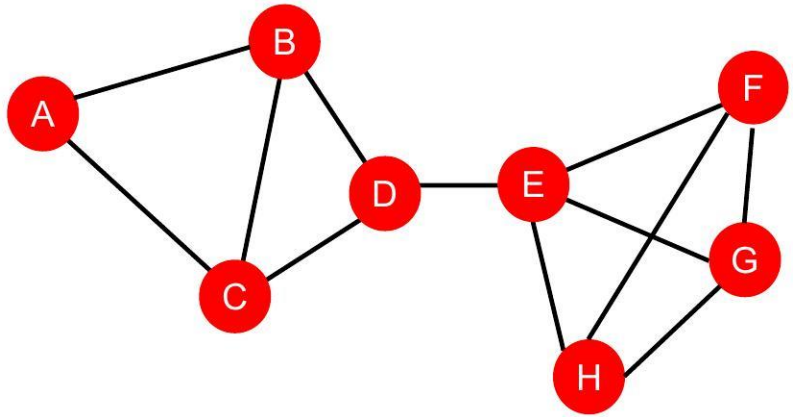




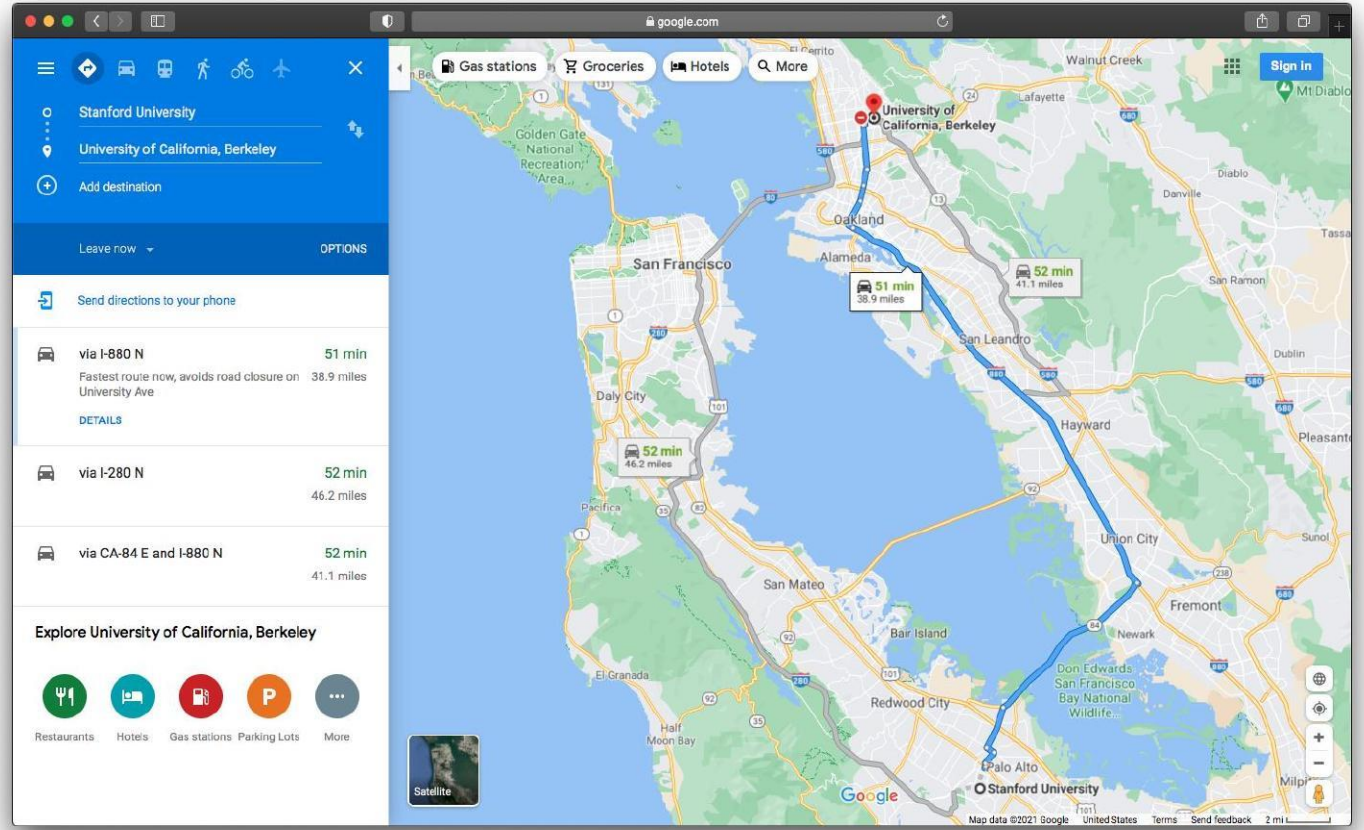
Graph-Level Prediction

- **Goal:** We want make a prediction for an entire graph or a subgraph of the graph.

- **For example:**

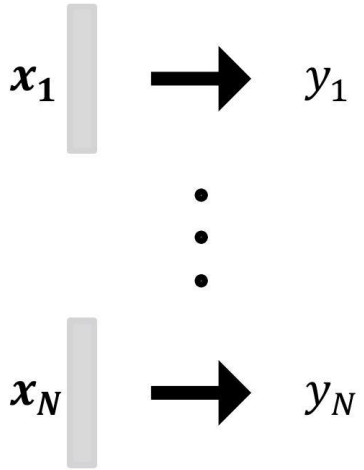


Example (1): Traffic Prediction

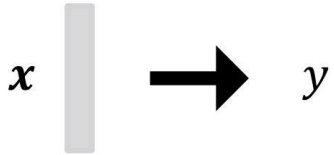


Traditional ML Pipeline

- **Train an ML model:**
  - Random forest
  - SVM
  - Neural network, etc.



- **Apply the model:**
  - Given a new node/link/graph, obtain its features and make a prediction



This Lecture: Feature Design

- **Using effective features over graphs is the key to achieving good model performance.**
- Traditional ML pipeline uses hand-designed features.
- **In this lecture, we overview the traditional features for:**
  - Node-level prediction
  - Link-level prediction
  - Graph-level prediction
- For simplicity, we focus on undirected graphs.

Machine Learning in Graphs

Goal: Make predictions for a set of objects

Design choices:

- **Features:** *d*-dimensional vectors
- **Objects:** Nodes, edges, sets of nodes, entire graphs
- **Objective function:**
  - What task are we aiming to solve?

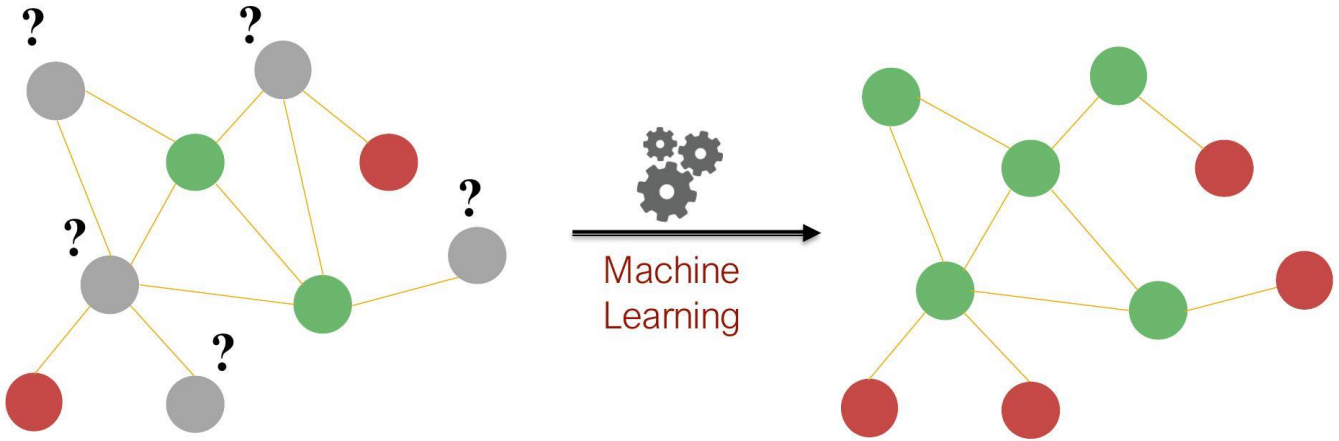
Machine Learning in Graphs

Example: Node-level prediction

- Given: $G = (V, E)$
- Learn a function: $f : V \rightarrow \mathbb{R}$

How do we learn the function?

Node-Level Tasks



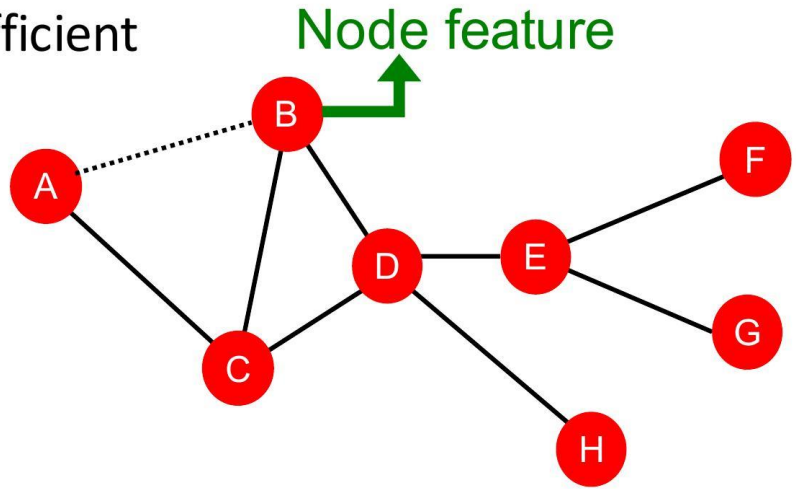
Node classification

ML needs features.

Node-Level Features: Overview

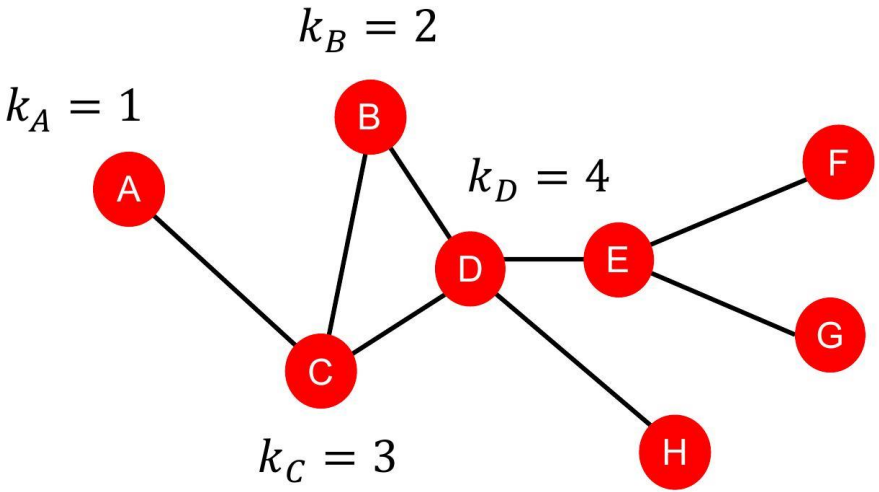
Goal: Characterize the structure and position of a node in the network:

- Node degree
- Node centrality
- Clustering coefficient
- Graphlets



Node Features: Node Degree

- The degree k_v of node v is the number of edges (neighboring nodes) the node has.
- Treats all neighboring nodes equally.



Node Features: Node Centrality

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality c_v takes the node importance in a graph into account
- **Different ways to model importance:**
  - Eigenvector centrality
  - Betweenness centrality
  - Closeness centrality
  - and many others…

Node Centrality (1)

- **Eigenvector centrality:**
  - A node v is important if surrounded by important neighboring nodes $u \in N(v)$.
  - We model the centrality of node v as the sum of the centrality of neighboring nodes:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

λ is normalization constant (it will turn out to be the largest eigenvalue of A)

  - Notice that the above equation models centrality in a recursive manner. **How do we solve it?**

Node Centrality (1)

- **Eigenvector centrality:**
  - Rewrite the recursive equation in the matrix form.

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \quad \Longleftrightarrow \quad \lambda \boldsymbol{c} = \boldsymbol{A}\boldsymbol{c}$$

λ is normalization const (largest eigenvalue of A)

- $\boldsymbol{A}$: Adjacency matrix
  $\boldsymbol{A}_{uv} = 1$ if $u \in N(v)$
- $\boldsymbol{c}$: Centrality vector
- λ: Eigenvalue

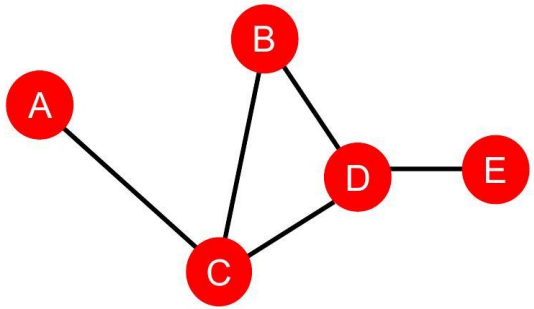
  - We see that centrality c is the **eigenvector of $\boldsymbol{A}$**!
  - The largest eigenvalue λ_{max} is always positive and unique (by Perron-Frobenius Theorem).
  - The eigenvector $\boldsymbol{c}_{max}$ corresponding to λ_{max} is used for centrality.

Node Centrality (2)

- **Betweenness centrality:**
  - A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths betwen } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

  - **Example:**



$c_A = c_B = c_E = 0$

$c_C = 3$

(A-C-B, A-C-D, A-C-D-E)

$c_D = 3$

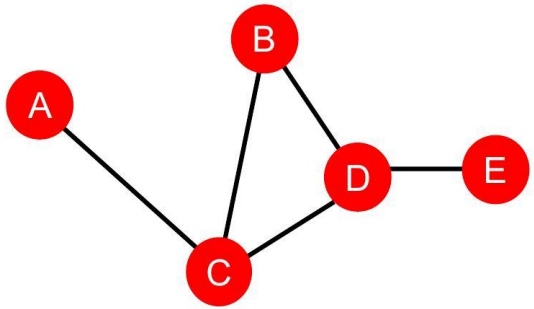
(A-C-D-E, B-D-E, C-D-E)

Node Centrality (3)

- **Closeness centrality:**
  - A node is important if it has small shortest path lengths to all other nodes.

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

  - **Example:**



$c_A = 1/(2+1+2+3) = 1/8$
(A-C-B, A-C, A-C-D, A-C-D-E)

$c_D = 1/(2+1+1+1) = 1/5$
(D-C-A, D-B, D-C, D-E)

Node Features: Clustering Coefficient

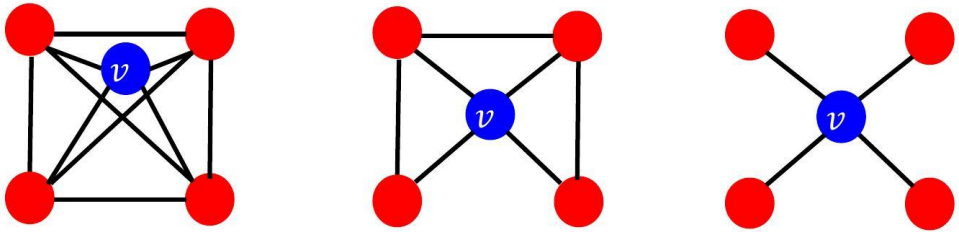
- Measures how connected $v's$ neighboring nodes are:

$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\binom{k_v}{2}} \in [0,1]$$

$\#(\text{node pairs among } k_v \text{ neighboring nodes})$

In our examples below the denominator is 6 (4 choose 2).

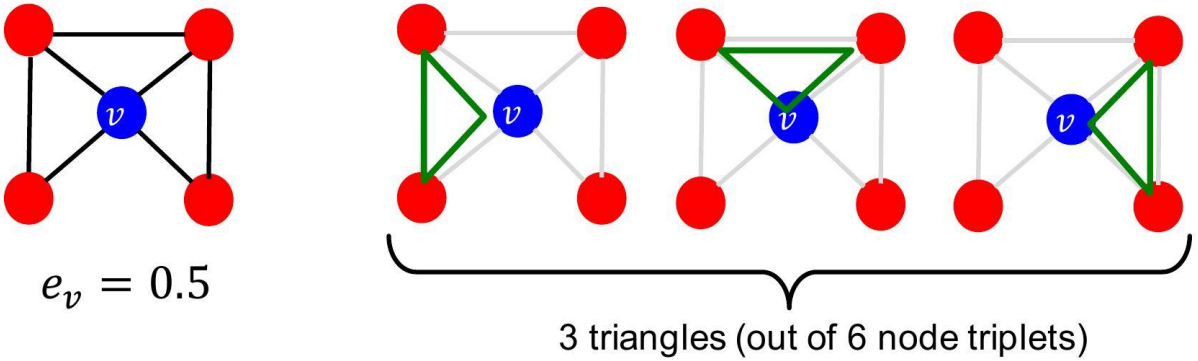
- **Examples:**



$e_v = 1$ $e_v = 0.5$ $e_v = 0$

Node Features: Graphlets

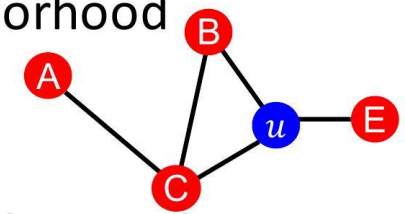
- **Observation:** Clustering coefficient counts the #(triangles) in the ego-network



- We can generalize the above by counting #(pre-specified subgraphs, i.e., **graphlets**).

Node Features: Graphlets

- **Goal:** Describe network structure around node u
  - **Graphlets** are small subgraphs that describe the structure of node u's network neighborhood



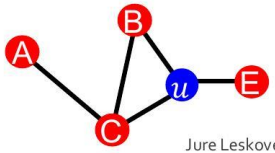
Analogy:

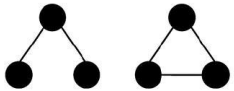
- **Degree** counts **#(edges)** that a node touches
- **Clustering coefficient** counts **#(triangles)** that a node touches.
- **Graphlet Degree Vector (GDV):** Graphlet-base features for nodes
  - **GDV** counts **#(graphlets)** that a node touches

Node Features: Graphlets

- Considering graphlets of size 2-5 nodes we get:
  - **Vector of 73 coordinates** is a signature of a node that describes the topology of node's neighborhood

- Graphlet degree vector provides a measure of a **node's local network topology**:
  - Comparing vectors of two nodes provides a more detailed measure of local topological similarity than node degrees or clustering coefficient.





Stanford CS224W:
Link Prediction Task and Features

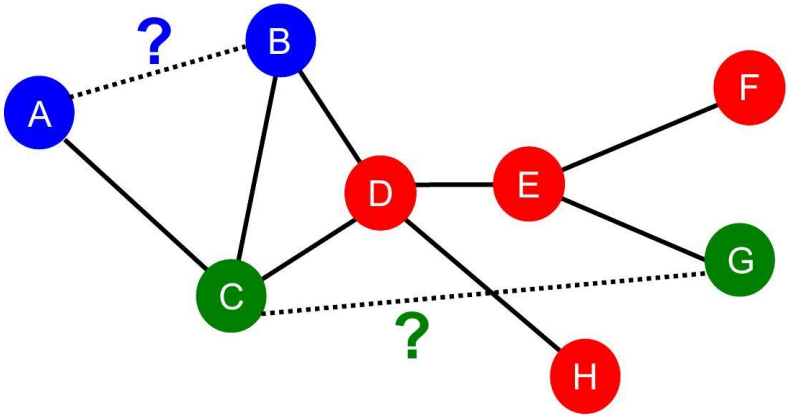
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Link-Level Prediction Task: Recap

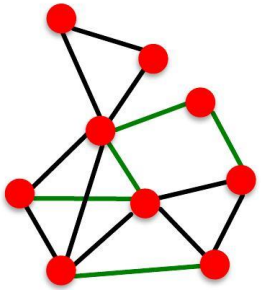
- The task is to predict **new links** based on the existing links.
- At test time, node pairs (with no existing links) are ranked, and top K node pairs are predicted.
- The key is to design features for **a pair of nodes**.



Link Prediction as a Task

Two formulations of the link prediction task:

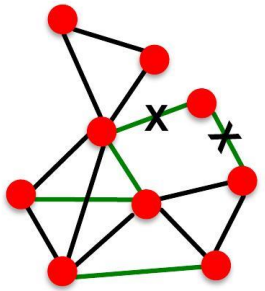
- **1) Links missing at random:**
  - Remove a random set of links and then aim to predict them
- **2) Links over time:**
  - Given $G[t_0, t_0']$ a graph defined by edges up to time t_0', **output a ranked list L** of edges (not in $G[t_0, t_0']$) that are predicted to appear in time $G[t_1, t_1']$
  - **Evaluation:**
    - $n = |E_{new}|$: # new edges that appear during the test period $[t_1, t_1']$
    - Take top n elements of L and count correct edges



$G[t_0, t_0']$
$G[t_1, t_1']$

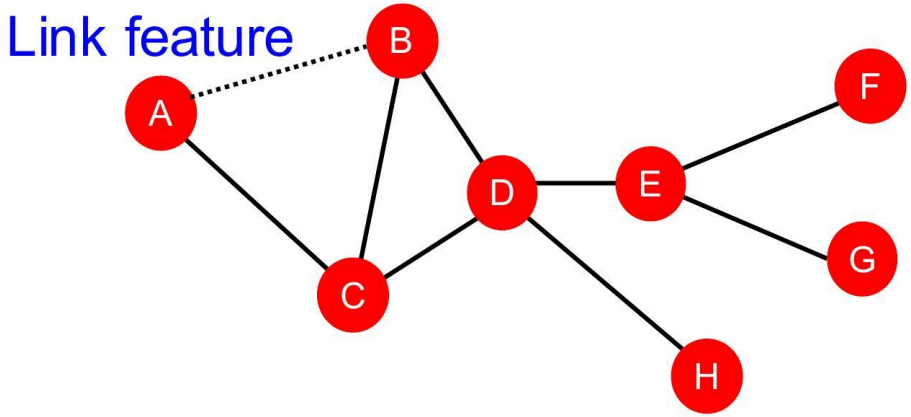
Link Prediction via Proximity

- **Methodology:**
  - For each pair of nodes *(x,y)* compute score *c(x,y)*
    - For example, *c(x,y)* could be the # of common neighbors of *x* and *y*
  - Sort pairs *(x,y)* by the decreasing score *c(x,y)*
  - **Predict top n pairs as new links**
  - **See which of these links actually appear in $G[t_1, t_1']$**



Link-Level Features: Overview

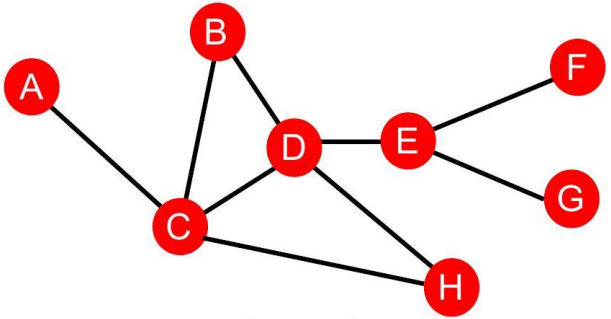
- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap



Distance-Based Features

Shortest-path distance between two nodes

- Example:



$$S_{BH} = S_{BE} = S_{AB} = 2$$
$$S_{BG} = S_{BF} = 3$$

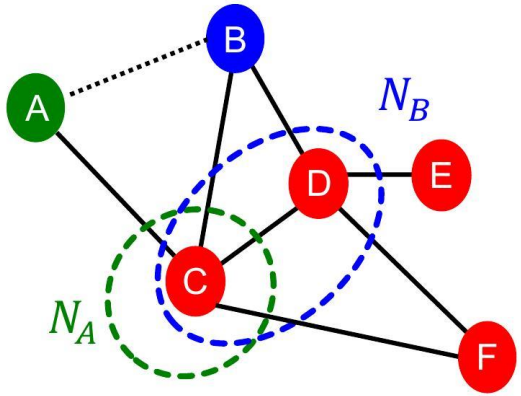
- However, this does not capture the degree of neighborhood overlap:
  - Node pair *(B, H)* has 2 shared neighboring nodes, while pairs *(B, E)* and *(A, B)* only have 1 such node.

Local Neighborhood Overlap

Captures # neighboring nodes shared between two nodes v_1 and v_2:

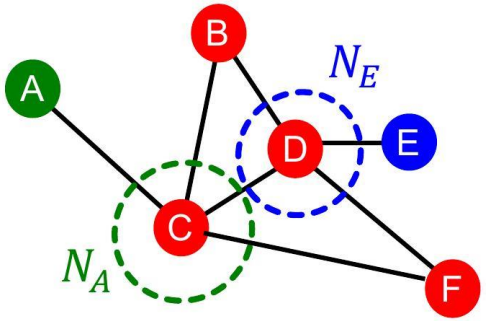
- **Common neighbors:** $|N(v_1) \cap N(v_2)|$
  - Example: $|N(A) \cap N(B)| = |\{C\}| = 1$
- **Jaccard's coefficient:** $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
  - Example: $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C,D\}|} = \frac{1}{2}$
- **Adamic-Adar index:**

  $\sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)}$
  - Example: $\frac{1}{\log(k_C)} = \frac{1}{\log 4}$



Global Neighborhood Overlap

- **Limitation of local neighborhood features:**
  - Metric is always zero if the two nodes do not have any neighbors in common.



$$N_A \cap N_E = \phi$$
$$|N_A \cap N_E| = 0$$

  - However, the two nodes may still potentially be connected in the future.
- **Global neighborhood overlap** metrics resolve the limitation by considering the entire graph.

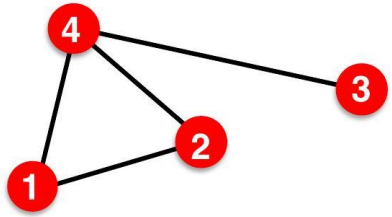
Global Neighborhood Overlap

- **Katz index**: count the number of walks of all lengths between a given pair of nodes.

- **Q: How to compute #walks between two nodes?**
- Use **powers of the graph adjacency matrix**!

Intuition: Powers of Adj Matrices

- **Computing #walks between two nodes**
  - **Recall**: $A_{uv} = 1$ if $u \in N(v)$
  - Let $P_{uv}^{(K)}$ = #walks of length K between u and v
  - We will show $P^{(K)} = A^k$
  - $P_{uv}^{(1)}$ = #walks of length 1 (direct neighborhood) between u and $v = A_{uv}$



$$P_{12}^{(1)} = A_{12}$$

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Intuition: Powers of Adj Matrices

- **How to compute $P_{uv}^{(2)}$?**
  - **Step 1:** Compute **#walks** of length 1 **between each of u's neighbor and v**
  - **Step 2**: **Sum up** these #walks across u's neighbors
  - $P_{uv}^{(2)} = \sum_i A_{ui} * P_{iv}^{(1)} = \sum_i A_{ui} * A_{iv} = A_{uv}^2$

Node 1's neighbors

#walks of length 1 between Node 1's neighbors and Node 2

$P_{12}^{(2)} = A_{12}^2$

$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 3 \end{pmatrix}$$

Power of adjacency

Global Neighborhood Overlap

- **Katz index:** count the number of walks of all lengths between a pair of nodes.

- How to compute #walks between two nodes?
- Use **adjacency matrix powers**!
  - $\boldsymbol{A}_{uv}$ specifies #walks of length 1 (direct neighborhood) between u and v.
  - $\boldsymbol{A}_{uv}^{\boldsymbol{2}}$ specifies #walks of **length 2** (neighbor of neighbor) between u and v.
  - And, $\boldsymbol{A}_{uv}^{\boldsymbol{l}}$ specifies #walks of **length $\boldsymbol{l}$**.

Global Neighborhood Overlap

- **Katz index** between v_1 and v_2 is calculated as

Sum over *all walk lengths*

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1 v_2}$$

$A^l_{v_1 v_2}$: #walks of length l between v_1 and v_2

$0 < \beta < 1$: discount factor

- Katz index matrix is computed in closed-form:

$$\boldsymbol{S} = \sum_{i=1}^{\infty} \beta^i \boldsymbol{A}^i = \underbrace{(\boldsymbol{I} - \beta \boldsymbol{A})^{-1}}_{= \sum_{i=0}^{\infty} \beta^i \boldsymbol{A}^i \text{ by geometric series of matrices}} - \boldsymbol{I},$$

Link-Level Features: Summary

- **Distance-based features:**
  - Uses the shortest path length between two nodes but does not capture how neighborhood overlaps.
- **Local neighborhood overlap:**
  - Captures how many neighboring nodes are shared by two nodes.
  - Becomes zero when no neighbor nodes are shared.
- **Global neighborhood overlap:**
  - Uses global graph structure to score two nodes.
  - Katz index counts #walks of all lengths between two nodes.

Stanford CS224W: Graph-Level Features and Graph Kernels

CS224W: Machine Learning with Graphs

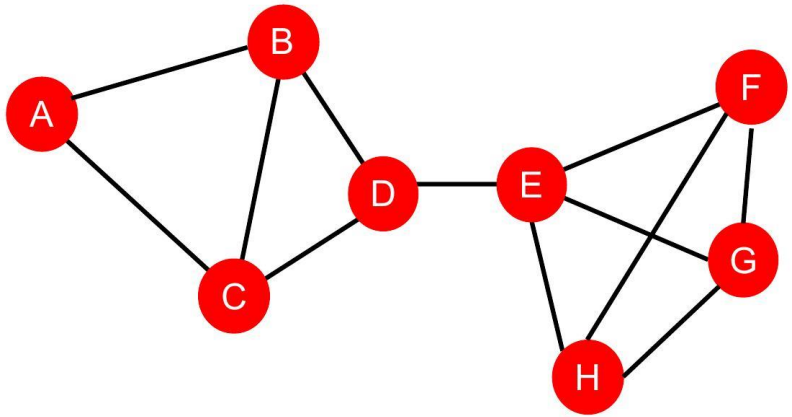
Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Graph-Level Features

- **Goal:** We want features that characterize the structure of an entire graph.

- **For example:**



Background: Kernel Methods

- **Kernel methods** are widely-used for traditional ML for graph-level prediction.
- **Idea: Design kernels instead of feature vectors.**
- **A quick introduction to Kernels:**
  - Kernel $K(G, G') \in \mathbb{R}$ measures similarity b/w data
  - Kernel matrix $\boldsymbol{K} = \left(K(G, G')\right)_{G,G'}$ must always be positive semidefinite (i.e., has positive eigenvalues)
  - There exists a feature representation $\phi(\cdot)$ such that $K(G, G') = \phi(\mathrm{G})^{\mathrm{T}}\phi(G')$
  - Once the kernel is defined, off-the-shelf ML model, such as kernel SVM, can be used to make predictions.

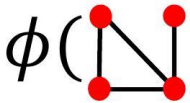
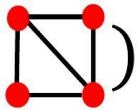
Graph-Level Features: Overview

- **Graph Kernels:** Measure similarity between two graphs:
  - Graphlet Kernel [1]
  - Weisfeiler-Lehman Kernel [2]
  - Other kernels are also proposed in the literature (beyond the scope of this lecture)
    - Random-walk kernel
    - Shortest-path graph kernel
    - And many more…

[1] Shervashidze, Nino, et al. "Efficient graphlet kernels for large graph comparison." Artificial Intelligence and Statistics. 2009.
[2] Shervashidze, Nino, et al. "Weisfeiler-lehman graph kernels." Journal of Machine Learning Research 12.9 (2011).

Graph Kernel: Key Idea

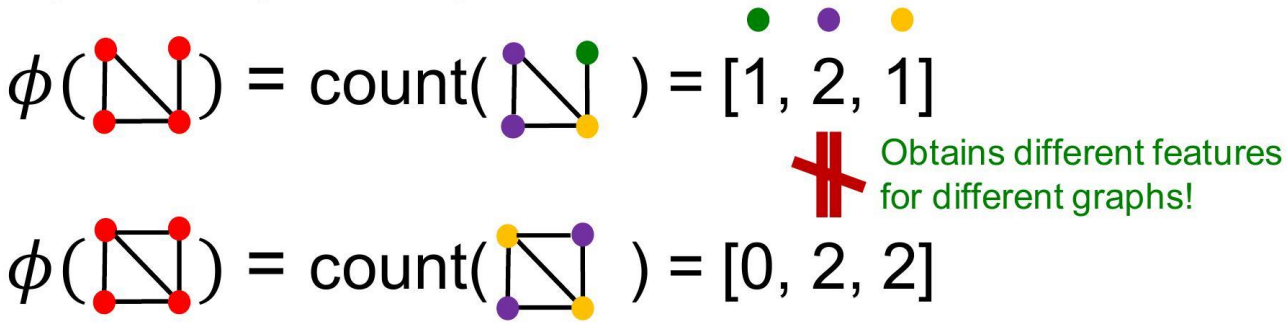
- **Goal**: Design graph feature vector $\phi(G)$
- **Key idea**: Bag-of-Words (BoW) for a graph
  - **Recall:** BoW simply uses the word counts as features for documents (no ordering considered).
  - Naïve extension to a graph: Regard nodes as words.
  - Since both graphs have 4 red nodes, we get the same feature vector for two different graphs…

$\phi($  $) = \phi($  $)$

Graph Kernel: Key Idea

What if we use Bag of **node degrees**?

Deg1: ● Deg2: ● Deg3: ●



- Both Graphlet Kernel and Weisfeiler-Lehman (WL) Kernel use **Bag-of-*** representation of graph, where * is more sophisticated than node degrees!

Weisfeiler-Lehman Kernel

- **Goal**: Design an efficient graph feature descriptor $\phi(G)$
- **Idea**: Use neighborhood structure to iteratively enrich node vocabulary.
  - Generalized version of **Bag of node degrees** since node degrees are one-hop neighborhood information.
- **Algorithm to achieve this**:

Color refinement

Color Refinement

- **Given:** A graph G with a set of nodes V.
  - Assign an initial color $c^{(0)}(v)$ to each node v.
  - Iteratively refine node colors by

$$c^{(k+1)}(v) = \text{HASH}\left(\left\{c^{(k)}(v), \left\{c^{(k)}(u)\right\}_{u \in N(v)}\right\}\right),$$

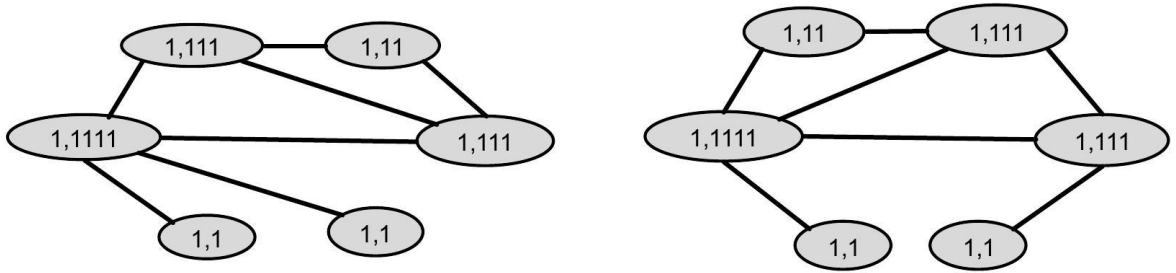
    where HASH maps different inputs to different colors.
  - After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood

Color Refinement (1)

Example of color refinement given two graphs

- Assign initial colors

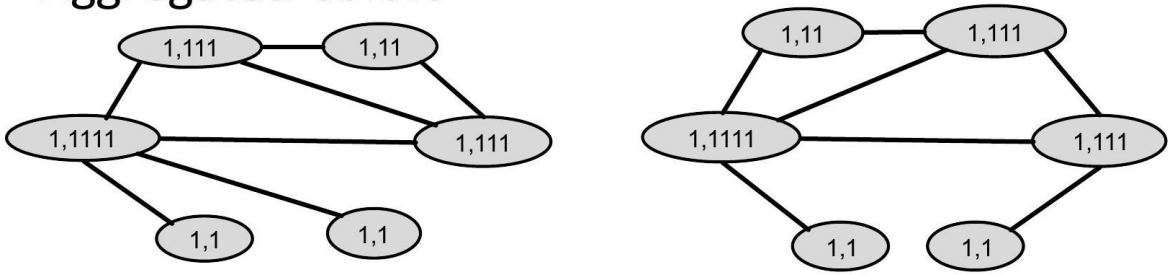
- Aggregate neighboring colors



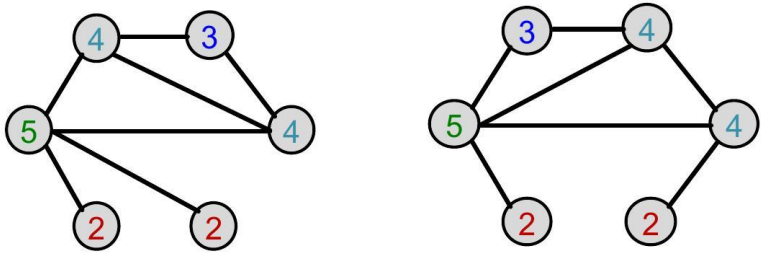
Color Refinement (2)

Example of color refinement given two graphs

- Aggregated colors



- Hash aggregated colors



Hash table

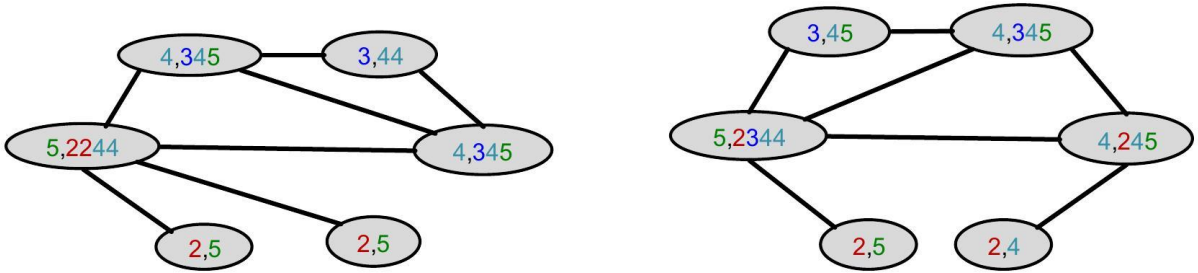
1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

Color Refinement (3)

Example of color refinement given two graphs

- Aggregated colors

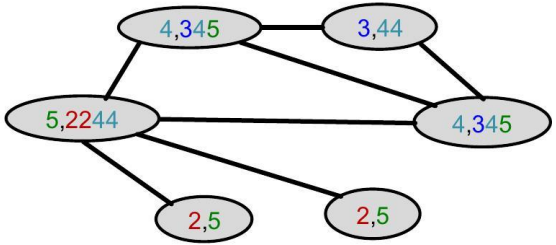
- Hash aggregated colors

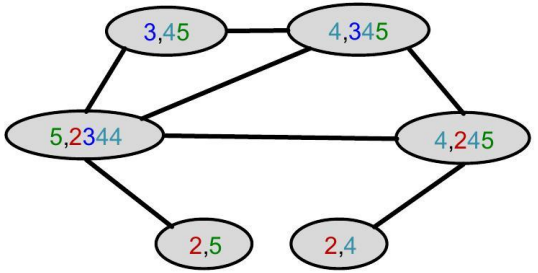


Color Refinement (4)

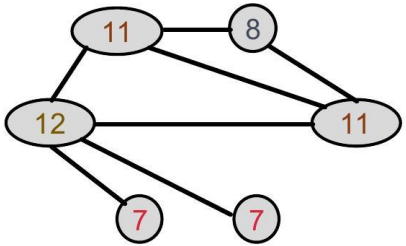
Example of color refinement given two graphs

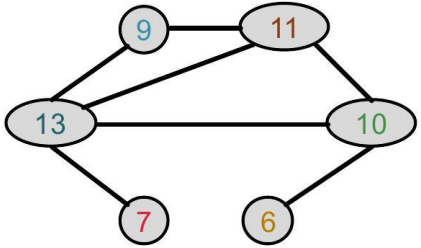
- Aggregated colors





- Hash aggregated colors



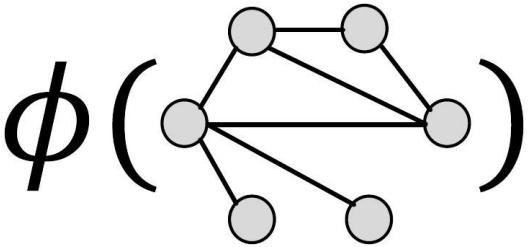


Hash table

2,4	-->	6
2,5	-->	7
3,44	-->	8
3,45	-->	9
4,245	-->	10
4,345	-->	11
5,2244	-->	12
5,2344	-->	13

Weisfeiler-Lehman Graph Features

After color refinement, WL kernel counts number of nodes with a given color.

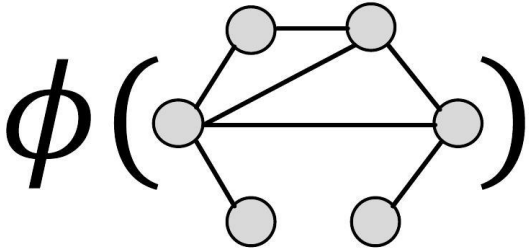


Colors

1,2,3,4,5,6,7,8,9,10,11,12,13

= [6,2,1,2,1,0,2,1,0, 0, 0, 2, 1]

Counts



1,2,3,4,5,6,7,8,9,10,11,12,13

= [6,2,1,2,1,1,1,0,1, 1, 1, 0, 1]

Weisfeiler-Lehman Kernel

The WL kernel value is computed by the inner product of the color count vectors:

$$K(\,,\,) = \phi(\,)^{T}\phi(\,) = 49$$

Weisfeiler-Lehman Kernel

- WL kernel is **computationally efficient**
  - The time complexity for color refinement at each step is linear in #(edges), since it involves aggregating neighboring colors.
- When computing a kernel value, only colors appeared in the two graphs need to be tracked.
  - Thus, #(colors) is at most the total number of nodes.
- Counting colors takes linear-time w.r.t. #(nodes).
- In total, time complexity is **linear in #(edges)**.