Big Data Computing

Master's of Science in Computer Science

Sapienza University of Rome

2026/2027

Gabriele Tolomei

tolomei@di.uniroma1.it

Module 2: High-Dimensional Data Representations

Lecture 2.4: Graph Representations & Embeddings

Philosophy of Graph Representation

> *"Everything is a graph, if you look at it long enough."*
> *— Folklore of network science*

Text gave us sparse-then-dense vectors (Lecture 2.3). Graphs give us something structurally irregular: relational structure itself is the data — characterized at web scale by catastrophic sparsity, power-law degree distributions, and non-trivial partitioning challenges.

Lecture Roadmap

This lecture studies how massive real-world networks are represented computationally, stored across distributed clusters, and mapped back into dense vector spaces.

1. **From Vectors to Graphs** — the relational data model & ubiquity
2. **Classical Representations & The Scaling Wall** — Adjacency matrices ($O(n^2)$) vs. lists ($O(n+m)$)
3. **Power-Law Degree Distributions & Extreme Sparsity** — why real graphs break uniformity
4. **Adjacency Decomposition & Distributed Storage** — Edge-Cut vs. Vertex-Cut partitioning
5. **Node-to-Vector Mappings: Graph Embeddings** — DeepWalk, node2vec, and matrix factorization
6. **Synthesis** — unifying Module 2's representation tour

> *Goal: Understand how irregular relational networks become scalable distributed structures and geometric vector embeddings.*

Why This Matters: From Vectors to Network Structure

Lectures 2.1–2.3 operated in vector spaces: points in $\mathbb{R}^d$ with distance metrics and fixed dimensionality.

- **Graphs break regular geometric assumptions:** nodes have variable degrees (from 1 to 10^7), no natural coordinate ordering, and arbitrary connectivity.
- **Web-scale data is relational:** web hyperlinks, social graphs, knowledge bases, citation networks, transaction graphs.
- **The Core Dilemma:** How do we store hundreds of billions of edges without wasting petabytes of memory, and how do we bridge graphs back into vector-space machine learning?

This lecture provides the structural and geometric foundations to resolve both challenges.

§ 1 — From Vectors to Graphs

What Is a Graph?

A **graph** $G = (V, E)$ consists of:

- A set of **vertices** (nodes) V, with $|V| = n$.
- A set of **edges** (links) $E \subseteq V \times V$, with $|E| = m$, representing pairwise relationships.

Structural Variations

- **Undirected vs. Directed:** symmetric interactions (e.g., co-authorship, friendships) vs. asymmetric ties (e.g., hyperlinks, Twitter follows, financial transfers).
- **Unweighted vs. Weighted:** binary connectivity vs. scalar values $w(u, v)$ indicating bandwidth, interaction count, or similarity.

In big data systems, $|V|$ *routinely exceeds* 10^9 *and* $|E|$ *exceeds* 10^{11}.

Why Graphs Resist Standard Vector Tools

In Lectures 2.1–2.3, every data point was a vector $\mathbf{x} \in \mathbb{R}^d$ of fixed dimension d.

- **No uniform dimensionality:** vertex neighborhoods vary by orders of magnitude; there is no fixed set of coordinates.
- **Permutation invariance:** reordering vertex labels does not alter graph topology, but permuting vector dimensions changes the vector.
- **Non-independent samples:** in standard ML, data instances $\mathbf{x}_i$ are typically assumed independent (i.i.d.). In graphs, vertices are explicitly connected — dependencies are the signal!

Before we can apply vector-space ML or dimensionality reduction, we must first determine how to represent and store G.

§ 2 — Classical Representations & The Scaling Wall

The Adjacency Matrix & Its Linear Algebra Power

The canonical mathematical representation of $G = (V, E)$ is the **adjacency matrix** $A \in \{0,1\}^{n \times n}$:

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Why theoreticians love A:

- **Degree as row/column sums:** $\deg_{\text{out}}(i) = \sum_j A_{ij}$, $\deg_{\text{in}}(j) = \sum_i A_{ij}$.
- **Path counting:** $(A^k)_{ij}$ is the number of distinct walks of length k from node i to node j.
- **Spectral graph theory:** eigenvalues and eigenvectors of A (and graph Laplacians $L = D - A$) uncover connected components, clusters, and random-walk behavior (e.g., PageRank).

The $O(n^2)$ Scaling Wall at Web Scale

Storing A explicitly as a dense 2D array requires:

$$\text{Space}(A) = O(n^2)$$

regardless of how many edges actually exist. Consider storing just 1 bit per cell:

Network Scale	$n = \lvert V \rvert$	n^2 Entries	Storage (1 bit/entry)	Feasibility
Campus Network	10^4	10^8	12.5 MB	Trivial (RAM)
Citation Graph	10^7	10^{14}	12.5 TB	Cluster Memory
Web / Social Graph	10^9	10^{18}	**125 PB**	**Catastrophic!**

Even distributed across 1,000 machines, each node would need 125 TB of storage purely for a matrix that is $> 99.9999\%$ *zeros!*

Adjacency Lists: Exploiting Edge-Proportional Space

An **adjacency list** stores, for each vertex i, only its actual neighbors: $\mathrm{Adj}[i] = \{j : (i,j) \in E\}$.

$$\mathrm{Space}(\mathrm{Adj}) = O(n + m)$$

Comparison on a Billion-Node Social Network ($n = 10^9$, avg degree 200, $m = 10^{11}$):

Representation	Space Complexity	Actual Storage Required
Dense Adjacency Matrix	$O(n^2)$	$\approx \mathbf{125}$ PB
Adjacency List	$O(n + m)$	$\approx \mathbf{800}$ GB (fits in single high-end RAM!)

A reduction of $> 150,000\times$ in memory footprint! Whenever $m \ll n^2$, storing zeros is computational suicide.

Storage vs. Query Trade-offs

Operation	Dense Matrix (A)	Adjacency List (Adj)
Space	$O(n^2)$	$O(n+m)$
Edge check $(u,v) \in E$?	$O(1)$	$O(\deg(u))$
Enumerate neighbors of u	$O(n)$ (must scan row)	$O(\deg(u))$ (optimal)
Matrix-Vector Mult ($A\mathbf{x}$)	$O(n^2)$ flops	$O(m)$ flops (only nonzeros)

Bridging Lists and Linear Algebra: Compressed Sparse Row (CSR)

Real big-data systems (e.g., GraphX, PyTorch Geometric, SciPy) store sparse graphs using **CSR**:

- `values`: array of edge weights (length m).
- `col_idx`: target column indices (length m).
- `row_ptr`: pointers into `col_idx` for the start of each vertex's row (length $n+1$).
- Enables $O(m)$ matrix-vector operations with zero storage overhead for absent edges!

✅ Checkpoint 1 — Graph Representations

Let's test our understanding:

1. Why does an adjacency matrix require $O(n^2)$ memory regardless of edge count?
2. For a graph with 10^9 nodes and 10^{11} edges, what is the storage gap between a dense matrix and an adjacency list?
3. How does the Compressed Sparse Row (CSR) format allow algebraic operations without paying $O(n^2)$ space?

Take a moment to verify before we explore the structure of real-world networks.

§ 3 — Power-Law Degree Distributions & Extreme Sparsity

Degree Distributions: Theory vs. Reality

The **degree distribution** $P(k)$ is the fraction of vertices having degree k:

$$P(k) = \frac{|\{v \in V : \deg(v) = k\}|}{n}$$

The Naive Baseline: Erdős–Rényi $G(n, p)$

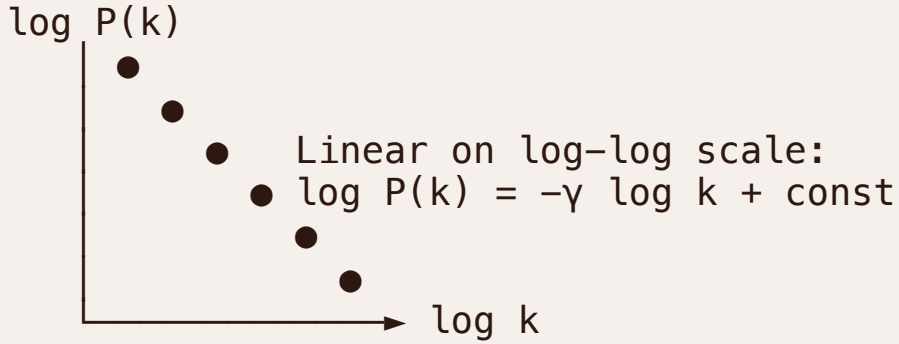
- Every pair of nodes is connected with independent probability p.
- Degree follows a **Poisson distribution**: $P(k) \approx \frac{e^{-\lambda}\lambda^k}{k!}$ (where $\lambda = (n-1)p$).
- **Concentration:** All nodes cluster tightly around the mean degree $\bar{k} = \lambda$. The probability of finding a node with $k = 10 \times \lambda$ the average degree gets exponentially quick to zero.

Do real-world networks look like this? ***Never.***

Power-Law Degree Distributions (Scale-Free Networks)

Empirical web graphs, social networks, and citation graphs follow a **Power Law**:

$$P(k) \sim k^{-\gamma}, \qquad \text{typically } 2 < \gamma < 3$$



- **Heavy Tail:** The polynomial decay $k^{-\gamma}$ decays vastly slower than exponential e^{-k}.
- **Generative Mechanism (Preferential Attachment):** "The rich get richer" — new nodes connect preferentially to already popular hubs.

Erdős–Rényi vs. Power-Law: The Hub Phenonemon

Feature	Erdős–Rényi $G(n,p)$	Power-Law Graph (Scale-Free)
Distribution	Poisson (light tail)	Power law $P(k) \sim k^{-\gamma}$ (heavy tail)
Typical Node	Represents the population well	No typical node: extreme variance!
Max Degree	$\approx O(\log n)$ or close to $\bar{k}$	**Hubs** with $\deg(v) \sim 10^5 - 10^7$
Small-World Effect	Moderate path lengths	Ultra-short paths (hubs act as cosmic bridges)
Clustering	Near zero (p)	High local clustering

Takeaway for System Designers:

In a power-law network, **a tiny fraction ($< 0.01\%$) of nodes holds a massive percentage of all edges**. Uniform assumptions will catastrophically break!

Quantifying Extreme Structural Sparsity

The **density** of an undirected graph is the fraction of possible edges present:

$$\text{Density}(G) = \frac{m}{\binom{n}{2}} \approx \frac{2m}{n^2}$$

Orders of Magnitude in Real Graphs:

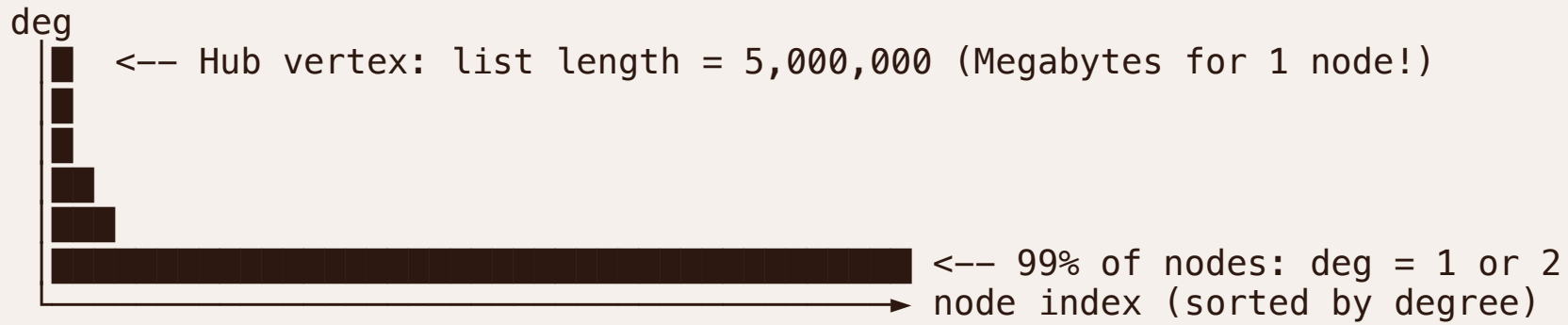
Network	Vertices n	Edges m	Graph Density
Web Graph (pages)	5×10^{10}	10^{12}	$\approx 8 \times 10^{-10}$
Twitter / X (users)	5×10^{8}	5×10^{10}	$\approx 4 \times 10^{-7}$
Citation Network	10^{8}	2×10^{9}	$\approx 4 \times 10^{-7}$

Notice: Density is not just 5% or 1% — it is 10^{-7} **to** 10^{-10}!

Real graphs are essentially vast voids punctuated by sparse links and dense superstar hubs.

Non-Uniform Sparsity: The Engineering Consequence

Because sparsity is non-uniform, adjacency list storage is **skewed**:



- When allocating memory: 90% of nodes take 16 bytes; 0.01% of nodes take 50 megabytes each.
- In multi-threaded or distributed processing, iterating over neighbors produces **severe straggler effects**: the worker handling a hub takes hours, while workers handling average nodes finish in milliseconds!

✅ Checkpoint 2 — Power Laws & Sparsity

Let's test our understanding:

1. What is the mathematical definition of a power-law degree distribution?
2. Why does a log-log plot convert $P(k) \sim k^{-\gamma}$ into a straight line?
3. What is a "hub", and why does the existence of hubs break naive load-balancing assumptions in distributed systems?

Now let's see how distributed systems partition these monster graphs across a cluster.

§ 4 — Adjacency Decomposition & Distributed Storage

Why We Must Partition Graphs

When $n = 10^{10}$ and $m = 10^{12}$, an adjacency list or CSR matrix exceeds hundreds of gigabytes or terabytes, and iterative algorithms (PageRank, GNNs) require distributed memory.

The Graph Partitioning Goal

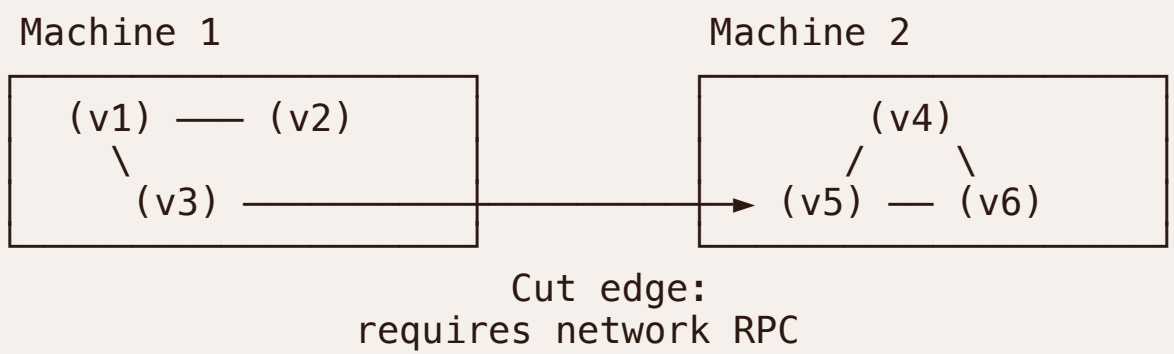
Given graph $G = (V, E)$ and k worker machines:

1. **Load Balancing:** Distribute data and computation evenly across all k workers.
2. **Minimize Communication:** Minimize edges or vertices that cross machine boundaries, preventing network shuffle bottlenecks.

Unfortunately, optimal graph partitioning is NP-hard. Real systems rely on structural heuristics.

Strategy 1: Edge-Cut Partitioning (Vertex Assignment)

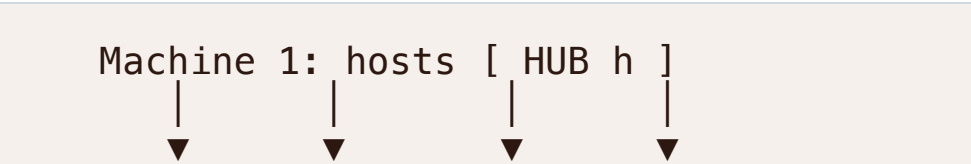
Assign each **vertex** to exactly one machine: $V = V_1 \cup V_2 \cup \cdots \cup V_k$.
Edges between vertices on different machines are **cut edges**.



- **Objective:** $\min |\{(u, v) \in E : \text{mach}(u) \neq \text{mach}(v)\}|$.
- **Best for:** Uniform, planar, or mesh-like graphs with bounded degree.

Why Edge-Cut Catastrophically Fails on Power-Law Graphs

What happens when an edge-cut partitioner encounters a **hub vertex** h with degree 10^6?



1. **Massive Communication Bottleneck:** Nearly all incident edges of h become cut edges. Machine 1 saturates its network interface!
2. **Incompatible Goals:**
   - If we balance $|V_i|$ (number of vertices per machine), Machine 1 handles 10^6 edges while Machine 2 handles 10^2 edges $\implies$ **extreme compute imbalance**.
   - If we balance $|E_i|$ by putting h alone on Machine 1, Machine 1 becomes an isolated communication hub $\implies$ **extreme network imbalance**.

Strategy 2: Vertex-Cut Partitioning (Edge Assignment)

Instead of splitting vertices, **split edges**: each edge $e \in E$ is assigned to exactly one machine.
Vertices with multiple incident edges are **replicated** across machines as mirrors.



- High-degree hub h has replicas (mirrors) on Machines 1 and 2, with one **master** replica.
- During computation, each machine aggregates edges locally for h's mirror, and only a **single aggregated value** is synced back to the master!
- **Result:** $O(k)$ synchronization messages instead of $O(\deg(h))$ individual edge messages!

Comparing Edge-Cut vs. Vertex-Cut

Property	Edge-Cut (Vertex Partitioning)	Vertex-Cut (Edge Partitioning)
What is divided?	Vertices are partitioned ($V_i \cap V_j = \emptyset$)	Edges are partitioned ($E_i \cap E_j = \emptyset$)
What crosses nodes?	Cut edges	Vertex replicas / mirrors
Communication cost	Proportional to number of cut edges	Proportional to replica factor $\times \lvert V \rvert$
Behavior on Hubs	**Terrible:** hub machine overwhelmed	**Excellent:** hub edges divided evenly
Target Workload	Road networks, finite-element meshes	**Social networks, Web graphs (Power-Law)**

System Adoption

Systems like **PowerGraph (GraphLab)** and **Spark GraphX** specifically adopt **vertex-cut** partitioning for big data analytics.

✅ Checkpoint 3 — Graph Partitioning

Let's test our understanding:

1. In edge-cut partitioning, why does a single high-degree hub cause a network bottleneck?
2. How does vertex-cut partitioning split a hub vertex's workload?
3. What is a "vertex replica" (or mirror), and why does syncing replicas require less network traffic than transmitting cut edges?

With distributed graph storage solved, how do we transform graphs into dense continuous vectors for Machine Learning?

§ 5 — Node-to-Vector Mappings: Graph Embeddings

The Representation Gap: Why We Need Embeddings

We've seen that raw graphs are irregular, sparse, and non-Euclidean:

Modality	Raw Representation	Dense Embedding Space	Lecture
Text	Sparse TF-IDF in $\mathbb{R}^{\lvert V_{\text{vocab}}\rvert}$	Dense Word2Vec in $\mathbb{R}^d$ ($d \approx 300$)	Lecture 2.3
Graphs	Adjacency list / CSR ($n \times n$)	**Node Embeddings in $\mathbb{R}^d$ ($d \approx 128$)**	**Lecture 2.4**

The Goal of Node Embedding

Learn a mapping function $\mathbf{z} : V \to \mathbb{R}^d$ (where $d \ll |V|$) such that:

$$\text{similarity}(u, v) \approx \mathbf{z}_u^\top \mathbf{z}_v$$

> *Geometric proximity in $\mathbb{R}^d$ directly mirrors topological similarity in graph G.*

What Should a Graph Embedding Preserve?

What does it mean for two nodes u, v to be "similar"?

1. **Direct Proximity (First-Order Proximity):**
   - If $(u, v) \in E$, their embeddings should be close: $\mathbf{z}_u^\top \mathbf{z}_v$ is large.
2. **Neighborhood Overlap (Second-Order Proximity):**
   - If u and v share many common neighbors (even if $(u, v) \notin E$), $\mathbf{z}_u$ and $\mathbf{z}_v$ should be close (e.g., two classmates who don't know each other).
3. **Community Structure (Homophily):**
   - Nodes in the same dense subgraph / cluster should map to a geometric cluster in $\mathbb{R}^d$.
4. **Structural Equivalence (Roles):**
   - Two hub nodes or two bridge nodes playing identical topological roles in distant parts of the graph should have similar representations.

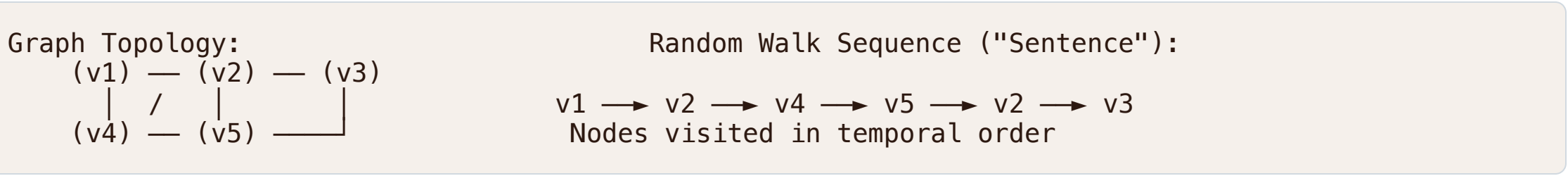
The Core Breakthrough: Random Walks as "Sentences"

In Lecture 2.3, **Word2Vec (Skip-gram)** learned word vectors by sliding a window over text sentences:

$$\text{Sentence:} \quad [\ldots, w_{t-1}, \mathbf{w_t}, w_{t+1}, w_{t+2}, \ldots]$$

The DeepWalk Insight (Perozzi et al., 2014)

Can we generate "sentences" from a graph? **Yes: via Random Walks!**



- Start at node v_0. At each step, jump to a uniformly chosen neighbor: $P(v_{t+1} = x \mid v_t = u) = \frac{1}{\deg(u)}$.
- A random walk of length L generates a sequence of node IDs $(v_1, v_2, \ldots, v_L)$ — **a graph "sentence"!**

DeepWalk: Word2Vec Directly Applied to Graph Walks

Once random walks are generated, DeepWalk executes **Skip-gram with Negative Sampling** (from Lecture 2.3!) verbatim:

$$\max_{\mathbf{z}} \sum_{t=1}^{L} \sum_{-c \le j \le c, j \neq 0} \log P(v_{t+j} \mid v_t)$$

where conditional probability is modeled via the softmax:

$$P(v_{\text{ctx}} \mid v_{\text{tgt}}) = \frac{\exp(\mathbf{z}'^{\top}_{v_{\text{ctx}}} \mathbf{z}_{v_{\text{tgt}}})}{\sum_{u \in V} \exp(\mathbf{z}'^{\top}_{u} \mathbf{z}_{v_{\text{tgt}}})}$$

Why This Works:

- If two nodes u and v are topologically close, a random walk starting at u frequently visits v.
- Thus, v appears repeatedly within the Skip-gram context window of u.
- Word2Vec pulls their embedding vectors $\mathbf{z}_u$ and $\mathbf{z}_v$ together via gradient descent!

Scalable Optimization: Negative Sampling on Graphs

Just as in Lecture 2.3, computing the full softmax denominator over $|V| = 10^9$ nodes is impossible.

DeepWalk and node2vec adopt **Negative Sampling (SGNS)**:

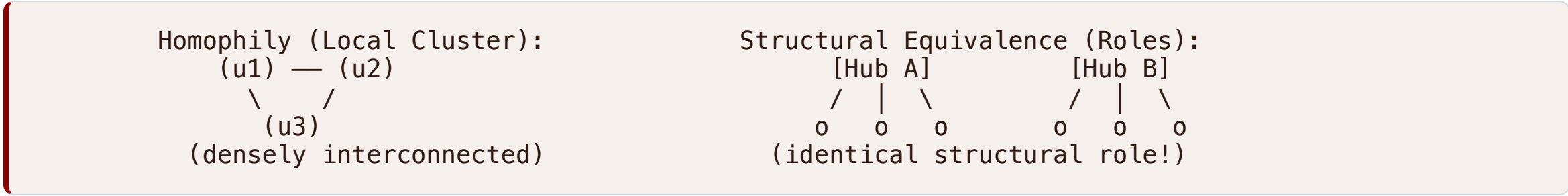
$$\mathcal{L}_{\text{SGNS}} = \log \sigma(\mathbf{z}_v'^{\top} \mathbf{z}_u) + \sum_{i=1}^{k} \mathbb{E}_{v_n \sim P_n(V)} \left[\log \sigma(-\mathbf{z}_{v_n}'^{\top} \mathbf{z}_u) \right]$$

- For a true co-occurring node pair (u, v) in a walk: maximize dot product.
- For k randomly sampled negative nodes v_n: minimize dot product.
- **Complexity per update:** reduced from $O(|V|)$ to $O(k \cdot d)$ — fully scalable to billions of edges!

Node2Vec: Biased Random Walks (Grover & Leskovec, 2016)

DeepWalk uses *uniform* random walks. But graphs exhibit two fundamentally different types of similarity:

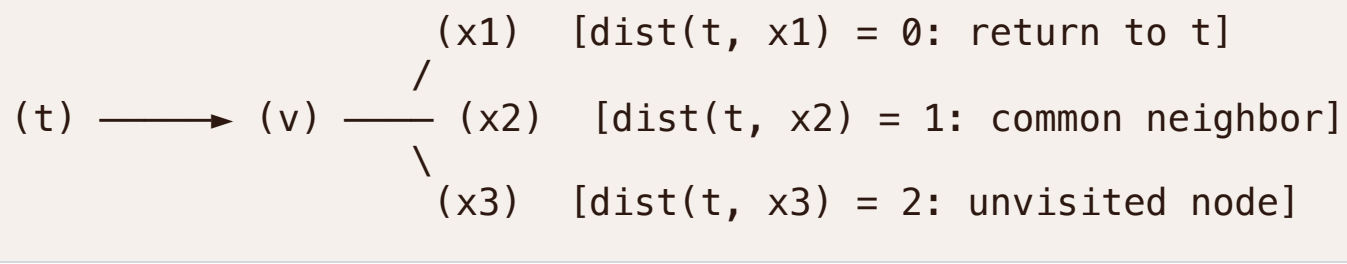
1. **Homophily:** Nodes in the same local community should have similar embeddings (BFS behavior).
2. **Structural Equivalence:** Nodes with similar structural roles (e.g., bridges, hubs) should have similar embeddings, even if far apart (DFS behavior).



node2vec introduces a 2nd-order random walk parameterized by two hyperparameters: p and q.

Node2Vec: The p and q Transition Probabilities

Suppose the random walk just traversed edge $(t \to v)$ and is currently at node v. Where does it step next?



The unnormalized transition probability $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ is guided by bias α_{pq}:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } \text{dist}(t,x) = 0 \quad (\text{return to previous node}) \\ 1 & \text{if } \text{dist}(t,x) = 1 \quad (\text{stay in local neighborhood}) \\ \frac{1}{q} & \text{if } \text{dist}(t,x) = 2 \quad (\text{explore outward / DFS}) \end{cases}$$

Node2Vec: Edge Weights and Normalization

Role of w_{vx}: the weight of the edge $(v, x) \in E$ connecting the current node v to a candidate next node x.

- Only the neighbors $x \in N(v)$ are candidates; for unweighted graphs, $w_{vx} = 1$.
- So $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ combines **edge strength** (w_{vx}) with the **walk bias** (α_{pq}, from the previous node t).

Normalization: the π_{vx} are arbitrary positive scores, so we divide by their sum over all neighbors of v:

$$P(v_{i+1} = x \mid v_i = v,\, v_{i-1} = t) = \frac{\pi_{vx}}{Z_{tv}} \quad \text{for } x \in N(v), \qquad Z_{tv} = \sum_{x' \in N(v)} \alpha_{pq}(t, x')\, w_{vx'}$$

Sanity check: with $p = q = 1$ and $w \equiv 1$, we get $\alpha \equiv 1$ and $P = 1/\deg(v)$ — exactly the uniform DeepWalk step.

x	$\text{dist}(t, x)$	w_{vx}	α_{pq} ($p = 0.5$, $q = 2$)	π_{vx}	$P = \pi_{vx}/Z_{tv}$
x_1	0 (return)	1	$1/p = 2$	2	$4/9$
x_2	1 (common nbr.)	2	1	2	$4/9$
x_3	2 (outward)	1	$1/q = 0.5$	0.5	$1/9$

Here $Z_{tv} = 2 + 2 + 0.5 = 4.5$, and the three probabilities sum to 1.

Intuition of p and q

Return Parameter p:

- **Low p ($p < 1$):** Walk tends to return to the node it just left (t). Keeps the walk local (BFS-like). Encourages **micro-neighborhood homophily**.
- **High p ($p > 1$):** Walk avoids immediately backtracking. Encourages discovering new nodes.

In-Out Parameter q:

- **Low q ($q < 1$):** Walk aggressively ventures away from t towards unvisited nodes (DFS-like). Discovers **global structural roles** (bridges, hubs).
- **High q ($q > 1$):** Walk stays confined to nodes connected to t. Emphasizes **local cluster membership**.

By tuning (p, q), node2vec seamlessly interpolates between community detection and role discovery!

Approach 2: Matrix Factorization-Based Embeddings

Instead of generating random walks, can we factorize graph matrices directly?

Recall from Lecture 2.2: SVD decomposes $X \approx U\Sigma V^\top$.
For graphs, define a proximity matrix $S \in \mathbb{R}^{n \times n}$ where S_{ij} measures connection strength:

$$\min_{U,V \in \mathbb{R}^{n \times d}} \|S - UV^\top\|_F^2$$

- Row $\mathbf{u}_i$ gives the d-dimensional embedding of node i.
- If $S = A$: factorizing the 1st-order adjacency matrix (high sparsity, limited multi-hop signal).
- If $S = A + A^2 + \cdots + A^k$: captures higher-order paths!

The Theoretical Bridge (NetMF, Qiu et al., 2018)

Remarkable mathematical result: **DeepWalk is implicitly factorizing a matrix!**
Specifically, DeepWalk with Skip-gram factorizes the Pointwise Mutual Information (PMI) matrix of graph transition probabilities:

$$S_{ij} = \log\left(\frac{\sum_{r=1}^{c}(P^r)_{ij}}{\sum_k \sum_{r=1}^{c}(P^r)_{kj}}\right) - \log(k_{\text{neg}})$$

Downstream Big Data Tasks with Node Embeddings

Once nodes are mapped to dense vectors $\mathbf{z}_v \in \mathbb{R}^d$, standard ML algorithms apply immediately:

1. Link Prediction ("People You May Know", Recommendations)

Predict probability of edge (u, v):

$$P((u, v) \in E) = \sigma(\mathbf{z}_u^\top \mathbf{z}_v) \quad \text{or} \quad \text{MLP}([\mathbf{z}_u \circ \mathbf{z}_v])$$

2. Node Classification (Fraud Detection, Customer Churn)

Use $\mathbf{z}_v$ as fixed input feature vectors into a classifier (Logistic Regression, Random Forest, SVM) to predict node labels y_v.

3. Graph Clustering & Community Detection

Run k-means on vectors $\{\mathbf{z}_v\}_{v \in V}$ in $\mathbb{R}^d$! Avoids expensive combinatorial graph cuts.

A Critical Systems Reality Check

> *Embeddings transform graph data into vectors, but they DO NOT eliminate the big data systems challenge!*

To compute DeepWalk or node2vec on a graph with 10^{10} edges:

1. **Walk Generation:** Generating millions of random walks requires distributed graph traversal across the vertex-cut cluster (§4).
2. **SGD on Vectors:** Updating $\mathbf{z}_u$ and $\mathbf{z}_v$ over billions of sampled walk steps requires distributed parameter synchronization (Module 4's Parameter Server & Hogwild!).
3. **Storage:** Storing 10^9 node vectors in $\mathbb{R}^{128}$ (float32) takes:

$$10^9 \times 128 \times 4 \text{ bytes} \approx \mathbf{512} \text{ GB}$$

   Manageable, but still requires distributed memory or flash SSDs.

Embeddings are the bridge between distributed graph infrastructure and downstream machine learning.

✅ Checkpoint 4 — Graph Embeddings

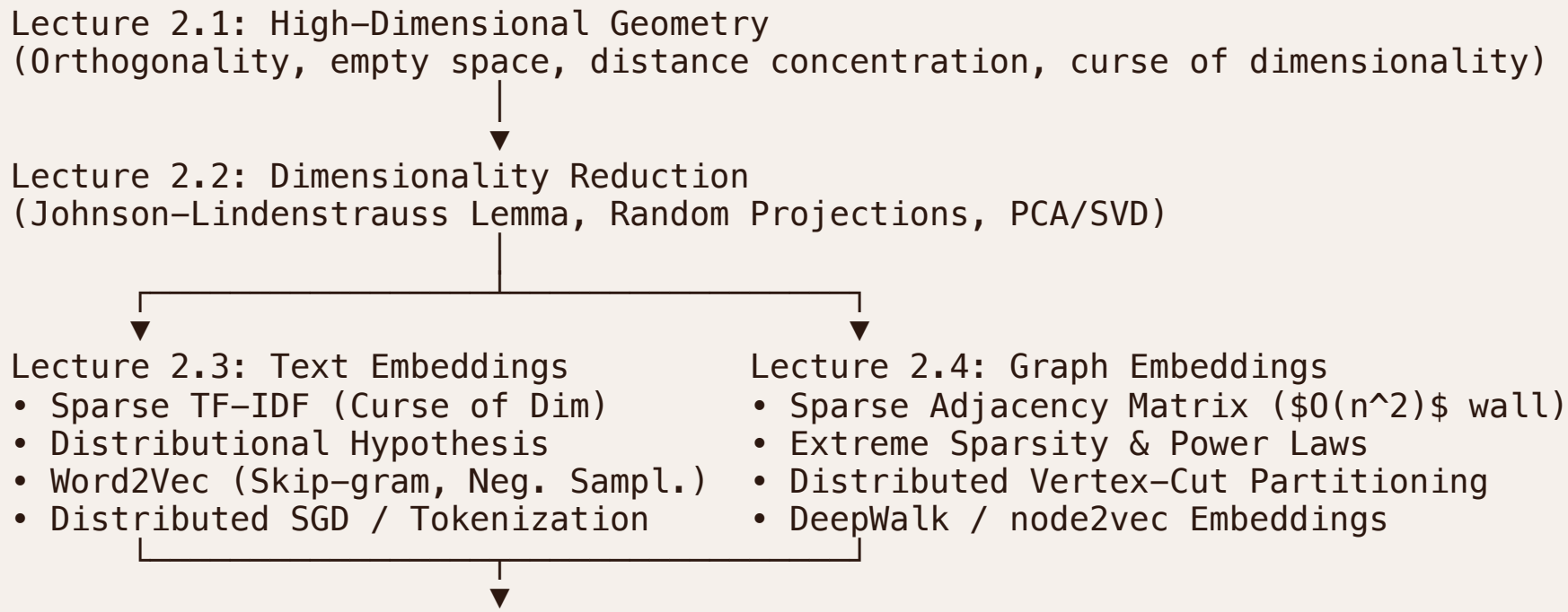
Let's test our understanding:

1. How does DeepWalk turn an irregular graph into "text sentences" suitable for Word2Vec?
2. What role do the p (return) and q (in-out) parameters play in node2vec?
3. What is the fundamental connection between DeepWalk and matrix factorization?
4. Once node embeddings are computed, how is link prediction performed?

Let's synthesize all of Module 2.

§ 6 — Synthesis & Module 2 Wrap-Up

The Master Arc of Module 2



Unified Representation Paradigm

Modality	Unstructured / Raw Form	Sparse Scaling Wall	Dense Latent Space	Algorithmic Bridge
High-Dim Data (2.1–2.2)	Arbitrary points $\mathbf{x} \in \mathbb{R}^D$	$D \to \infty$ (Curse of Dim)	$\mathbb{R}^d$ ($d \ll D$)	JL Lemma / SVD
Text (2.3)	Raw strings / tokens	$\lvert V_{\text{vocab}} \rvert \sim 10^7$ (Sparse TF-IDF)	Word vectors $\mathbb{R}^{300}$	Word2Vec (Skip-gram + Neg. Sampling)
Graphs (2.4)	Irregular graph $G = (V, E)$	$n^2 \sim 10^{18}$ (Dense Adjacency)	Node vectors $\mathbb{R}^{128}$	DeepWalk / node2vec (Walks + SGNS)

Across all modalities: Big data systems replace intractable, high-dimensional sparse representations with dense, low-dimensional continuous vector embeddings.

Key Concepts Glossary

Concept	Definition & Systems Impact
Adjacency Matrix	Dense $n \times n$ matrix; algebraically powerful but hits $O(n^2)$ storage wall.
Adjacency List / CSR	Stores only $O(n+m)$ nonzeros; essential for web graphs where density $\leq 10^{-7}$.
Power-Law ($k^{-\gamma}$)	Heavy-tailed degree distribution; causes severe load skew and straggler issues.
Vertex-Cut	Edge-partitioning with vertex mirrors; solves hub network bottlenecks.
DeepWalk	Random walks as sentences + Skip-gram Word2Vec $\Longrightarrow$ node embeddings.
Node2vec	2^{nd}-order biased random walks (p, q) balancing homophily & structural roles.

Forward Look: Where Module 2 Connects Next

Module 3: Scaling Classical Tasks

- We now have millions of dense vectors in $\mathbb{R}^d$ for words and graph nodes.
- How do we find the nearest neighbor of a query point in sub-linear time without scanning all N vectors?
- **Locality-Sensitive Hashing (LSH)** and Count-Min Sketches!

Module 4: Scaling Learning Tasks

- How do we update billions of embedding parameters in parallel across hundreds of GPUs/workers without locking?
- **Hogwild! asynchronous SGD** and **Parameter Server architectures**!

Contact & Q&A

- **Course:** Big Data Computing
- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!