Q13 — Embeddings & GNN (15 points)

Part 1 (2 points) — Why Antonyms Have Similar Embeddings

Answer: Antonyms frequently appear in **identical syntactic and semantic contexts**. For example, "increase" and "decrease" both modify words like "revenue," "temperature," "rate" and appear in the same sentence structures. Since word embeddings (e.g., Word2Vec) are learned from co-occurrence within context windows, words sharing similar contexts end up with similar vectors — regardless of whether their actual meanings are opposite.

Part 2 (2 points) — Undercomplete Autoencoder: Bottleneck Dimension

Answer: Bottleneck dimension < input dimension.

This constraint forces the network to learn a **compressed, informative representation**. Without an information bottleneck (i.e., if bottleneck ⩾ input dimension), the autoencoder could trivially learn the identity function — simply memorizing the input without extracting any meaningful latent structure.

Part 3 (1 point) — DeepWalk as Special Case of Node2Vec

$$\boxed{p = 1, \quad q = 1}$$

DeepWalk runs **unbiased** random walks. In Node2Vec, p controls the return probability and q controls inward/outward exploration. Setting both to 1 means no bias in any direction — all transition probabilities are equal, recovering DeepWalk's uniform random walk.

Part 4 (2 points) — GNN Layer Complexity

For a graph G = (V, E), each GNN layer performs:

- **Message passing:** one message per edge → $\boxed{O(|E|)}$
- **Neighbor aggregation:** one aggregation per node → $\boxed{O(|V|)}$

Part 5 (8 points) — GNN for Prim's Algorithm

Setup & Notation

- $h_v^k \in \{0, 1\}$: 1 if node v is in the partial spanning tree after round k, else 0
- $h_v^1 = 1$ iff $v = r$ (root node)
- e_{vj} = weight of edge (v, j)
- Each GNN layer adds exactly **one** node to the tree

Part 5a — Number of GNN Layers

Prim's algorithm starts with 1 node (root) and adds 1 node per iteration until all nodes are included:

$$\boxed{|V| - 1 \text{ layers}}$$

($|V|$ is also accepted)

Part 5b — GNN Functions

Message function:

Each node broadcasts its tree membership status:

$$\boxed{M(h_v^k) = h_v^k}$$

Aggregation function:

Identify the single non-tree node closest to the tree and mark it for addition:

$$\boxed{h_{N(v)}^{k+1} = \begin{cases} 1 & \text{if } v = \arg \min_{j\,:\,h_j^k=0} \min_{i\,:\,h_i^k=1} e_{ji} \\ 0 & \text{otherwise} \end{cases}}$$

Intuition: Among all non-tree nodes, find the one with the globally smallest edge weight to any tree node. That node (and only that node) receives aggregation value 1.

Update rule:

A node is in the tree if it was already there OR it was just selected:

$$\boxed{h_v^{k+1} = \max\left(h_v^k,\ h_{N(v)}^{k+1}\right)}$$

Predecessor tracking:

$$p_v^{k+1} = \begin{cases} v & \text{if } v = r \\ p_v^k & \text{if } v \neq r \text{ and } h_v^k = 1 \quad \text{(already in tree)} \\ \arg\min_{j\,:\,h_j^k = 1} e_{vj} & \text{if } h_v^k = 0 \text{ and } h_v^{k+1} = 1 \quad \text{(just added)} \\ \text{UNDEFINED} & \text{otherwise (not yet in tree)} \end{cases}$$

Intuition: When a node v is newly added to the tree, its predecessor is the tree node j that provided the minimum-weight edge connecting v to the tree. Nodes already in the tree retain their predecessor. Nodes not yet in the tree have no predecessor.

Why This Works

- **Layer-by-layer:** Each layer simulates one iteration of Prim's algorithm, adding the cheapest connection from the tree frontier.
- **Global argmin:** Unlike BFS or Bellman-Ford which can update multiple nodes per layer, Prim's selects exactly one node per round — the globally closest non-tree node.
- **Message = membership:** The message function simply communicates whether a neighbor is in the tree, which is the only information needed to evaluate candidate edges.