Node Embeddings Generation in Graph Databases using Node2Vec

Deepak Sujay Gudiseva

Fall 2025 Term Report

submitted in partial fulfillment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

Dec 7, 2025

Project Committee:

Dr. Munehiro Fukuda, Committee Chair

Dr. Min Chen, Committee Member

Dr. Bill Erdly, Committee Member

1. Introduction

1.1. Project Overview

Graph representation learning has emerged as a critical field in data science, allowing complex relational data to be transformed into vector spaces where standard machine learning algorithms can be applied. **Node2Vec**, a widely adopted algorithm, generates these embeddings by simulating biased random walks across a graph to capture both homophily and structural equivalence. However, structurally, Node2Vec is computationally intensive and memory-demanding, making it difficult to scale to massive graphs on single-machine architectures.

The Multi-Agent Spatial Simulation (MASS) library is a distributed framework designed to parallelize graph computations across a cluster of computing nodes. This project, **"Enabling Distributed Node2Vec on MASS,"** focuses on implementing a scalable, distributed version of the Node2Vec algorithm that leverages MASS's agent-based paradigm.

1.2. Fall Quarter Objectives

The primary objective for the Autumn 2025 quarter was to architect and implement the core engine of Distributed Node2Vec. Unlike traditional graph algorithms in MASS that operate on a strict vertex-centric model, deep learning training requires efficient sharing of model weights (the neural network parameters).

Our specific goals were:

1. **Design a "Compute-Node-Centric" Architecture**: Move away from verifying updates per-vertex to processing updates per-computing-node (JVM) to minimize network synchronization overhead.
2. **Implement Biased Random Walks**: Enable MASS Agents to perform second-order random walks (controlled by parameters ‘p’ and ‘q’) to generate training data.
3. **Develop Distributed Training Logic**: Create a mechanism for local model training on each worker node, followed by a global synchronization step ("Pack-Reduce-Broadcast").

4. **Integration**: Seamlessly integrate this new engine into the existing MASS application layer for ease of use.

This report details the successful implementation of these components, the technical challenges overcome, and the resulting architectural operational flow.

2. Background and Challenges

2.1. The Node2Vec Algorithm

Node2Vec extends the Skip-Gram architecture (originally from Word2Vec) to graphs. It works in two main phases:

1. **Random Walk Generation**: Agents traverse the graph to create "sentences" (sequences of nodes). The walk is biased by two parameters:
    - **Return parameter (p)**: Controls the likelihood of revisiting the previous node.
    - **In-out parameter (q)**: Controls the likelihood of exploring further away vs. staying locally.
2. **SGD Optimization (Skip-Gram)**: The algorithm maximizes the probability of predicting context nodes (neighbors in the walk) given a target node. This requires maintaining two large embedding matrices (Input and Output weights) for the entire vocabulary (all nodes in the graph).

2.2. Challenges in a Distributed Setting

Implementing this on a distributed system like MASS presents unique challenges:

- **Network Latency**: Standard Stochastic Gradient Descent (SGD) requires updating weights after every sample. In a distributed graph, vertices (and their embeddings) are scattered. Sending weight updates across the network for every step of a random walk is prohibitively slow.
- **Memory Constraints**: Replicating the entire embedding matrix on every node consumes vast amounts of memory. Storing it only on a master node creates a communication bottleneck.

- **State Synchronization**: Random walkers need to know their *previous* location to calculate bias (‘p’ and ‘q’), which requires carrying state across boundaries.

2.3. Limitations of Previous MASS Implementations

Prior iterations of graph algorithms in MASS were largely vertex-centric. Attributes were stored on VertexPlace objects. While excellent for algorithms like BFS, this is inefficient for training embeddings because:

- Accessing a global weight matrix from a VertexPlace would require a remote procedure call (RPC) for every dot product operation.
- The "Compare-and-Swap" pattern used in typical synchronized algorithms is too rigid for the stochastic nature of neural network training.

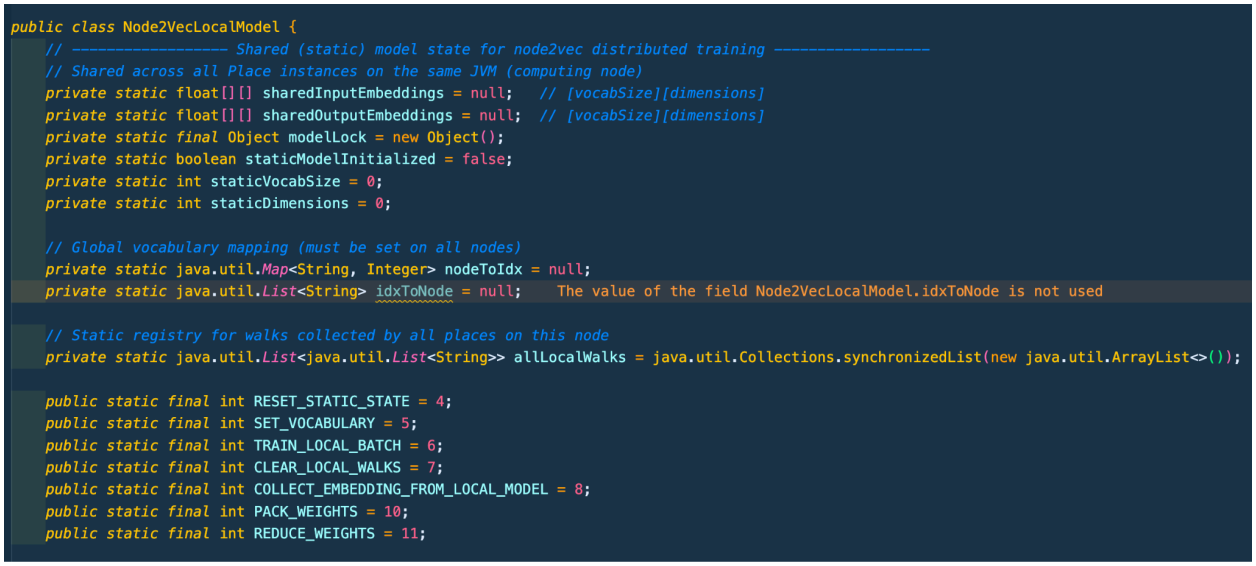
To solve this, we devised a new **Compute-Node-Centric** approach, detailed in the implementation section.

3. Autumn Quarter Implementation (AU25)

This quarter, we successfully implemented the complete end-to-end pipeline for Distributed Node2Vec. The implementation required significant modifications to mass_java_core and the development of new classes to handle distributed state.

3.1. Architectural Design: Compute-Node-Centric Training

The defining innovation of this implementation is using **Node-Granularity** instead of **Vertex-Granularity**.



```java
public class Node2VecLocalModel {
    // ------------------- Shared (static) model state for node2vec distributed training -------------------
    // Shared across all Place instances on the same JVM (computing node)
    private static float[][] sharedInputEmbeddings = null;   // [vocabSize][dimensions]
    private static float[][] sharedOutputEmbeddings = null;  // [vocabSize][dimensions]
    private static final Object modelLock = new Object();
    private static boolean staticModelInitialized = false;
    private static int staticVocabSize = 0;
    private static int staticDimensions = 0;

    // Global vocabulary mapping (must be set on all nodes)
    private static java.util.Map<String, Integer> nodeToIdx = null;
    private static java.util.List<String> idxToNode = null;    The value of the field Node2VecLocalModel.idxToNode is not used

    // Static registry for walks collected by all places on this node
    private static java.util.List<java.util.List<String>> allLocalWalks = java.util.Collections.synchronizedList(new java.util.ArrayList<>());

    public static final int RESET_STATIC_STATE = 4;
    public static final int SET_VOCABULARY = 5;
    public static final int TRAIN_LOCAL_BATCH = 6;
    public static final int CLEAR_LOCAL_WALKS = 7;
    public static final int COLLECT_EMBEDDING_FROM_LOCAL_MODEL = 8;
    public static final int PACK_WEIGHTS = 10;
    public static final int REDUCE_WEIGHTS = 11;
```

Figure-1: Node2VecLocalModel class

- **Traditional Approach**: Each Vertex updates its own embedding. (Too much communication).
- **Modified Approach**: Each **Computing Node (JVM)** maintains a **Static Local Model**.
    - All random walks that end up on a specific machine are collected locally.
    - A static class, Node2VecLocalModel as shown in Figure-1, aggregates these walks and performs training batch updates locally.
    - The Master Node synchronizes these local models only once per epoch, rather than once per vertex update.

3.2. Core Library Modifications (mass_java_core)

We modified the core library to support complex mobile agents and static state management.

3.2.1. PropertyGraphAgent.java: The Biased Walker

We enhanced the PropertyGraphAgent to support "stateful" migration. For Node2Vec, the next step depends on the *previous* step. We implemented walkStep() which calculates transition probabilities based on the edge weights and p/q bias.

Key Logic Implemented:

- **State Carriage**: The agent carries wargs (Walker Arguments), which track prevPlaceIndex and prevNeighborIds. This allows the agent to know where it came from after migrating to a new machine.

```java
/**
 * Arguments for Node2Vec walker agents.
 */
public class Node2VecWalkerArgs implements Serializable {
    public int walkLength = 80;
    public int walksPerNode = 10;
    public double p = 1.0;
    public double q = 1.0;
    // runtime fields
    public int sourcePlaceIndex = -1; // linear index of origin place
    public int prevPlaceIndex = -1;   // previous place visited (for sampling)
    public int remainingSteps = 0;
    public boolean isOrchestrator = true; // initial agents are orchestrators that spawn walkers
    public boolean endWalk = false;       // signal to end the walk and to store
    // optional itemID fields for convenience
    public String sourceItemID = null;
    public String prevItemID = null;
    // carry previous node's neighbor ids to allow triangle-aware p/q sampling
    public String[] prevNeighborIds = null;

    // if non-null, only traverse edges whose type set intersects this set
    // null means include all edge types
    public Set<String> includeEdgeTypes = null;
    // toggle for using full triangle-aware alpha; default true
    public boolean useFullAlpha = true;

    public Node2VecWalkerArgs() {}
}
```

Figure-2: Node2VecWalkerArgs class

- **Second-Order Markov Chain**: The transition probability **alpha** is dynamically calculated:
    - If neighbor == previous node: alpha = 1/p
    - If neighbor is connected to previous node: alpha = 1
    - If neighbor is disconnected from previous node: alpha = 1/q

```
if (wargs == null || !wargs.useFullAlpha || wargs.prevItemID == null) {
    alpha = 1.0;
} else if (wargs.prevItemID.equals(neighborId)) {
    alpha = 1.0 / wargs.p;
} else if (prevNeighborSet != null && prevNeighborSet.contains(neighborId)) {
    alpha = 1.0;
} else {
    alpha = 1.0 / wargs.q;
}
```

Figure-3: alpha calculation logic in PropertyGraphAgent

3.2.2. PropertyVertexPlace.java: The Distributed Hook

The PropertyVertexPlace acts as the interface between the MASS parallel system and our static local models. We implemented a robust command handling system in callMethod() to intercept distributed messages.

New Function IDs implemented:

- TRAIN_LOCAL_BATCH: Triggers the static model on this node to run SGD on its local buffer of walks.
- PACK_WEIGHTS / REDUCE_WEIGHTS: Handles the serialization of the large float arrays representing the model, allowing them to be sent to the Master for averaging.
- COLLECT_EMBEDDING: Allows a vertex to "reach into" the static local model and pull its final trained vector.

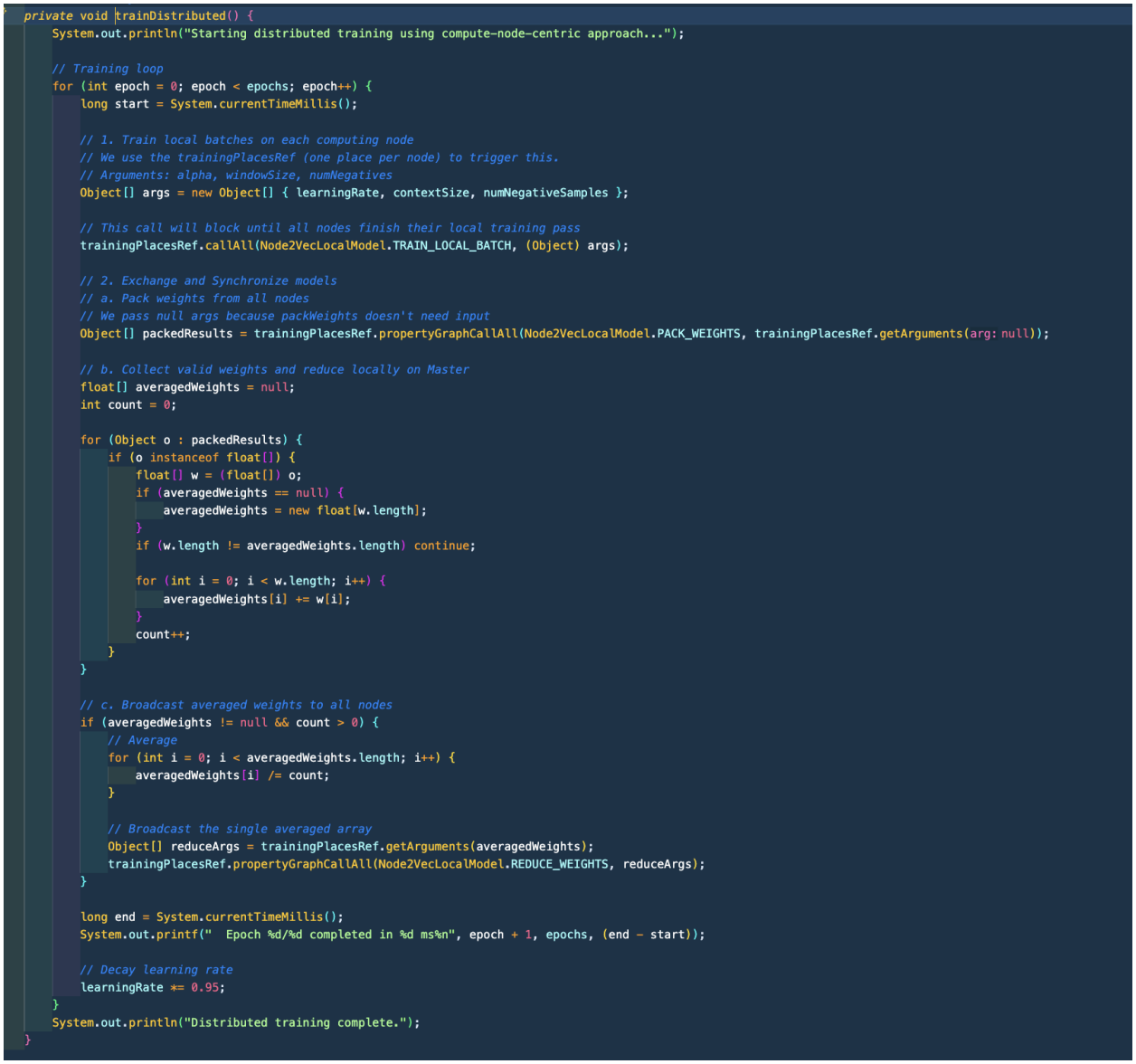
```
case Node2VecLocalModel.RESET_STATIC_STATE:
        if (argument instanceof Object) {
            Long[] args = (Long[]) argument;
            int vSize = args[0].intValue();
            int dim = args[1].intValue();
            long seed = args[2];
            Node2VecLocalModel.resetStaticState(vSize, dim, seed);
        }
        break;
case Node2VecLocalModel.SET_VOCABULARY:
        if (argument instanceof List) {
            Node2VecLocalModel.setVocabulary((List<String>) argument);    Type safety: Unchecked cast from Object to List<String>
        }
        break;
case Node2VecLocalModel.TRAIN_LOCAL_BATCH:
    MASS.getLogger().debug("[Static Model] Training local batch call all " + (argument instanceof Object));
    if (argument instanceof Object) {
        Object[] args = (Object[]) argument;
        double alpha = (Double) args[0];
        int win = (Integer) args[1];
        int neg = (Integer) args[2];
        MASS.getLogger().debug("[Static Model] Training local batch with alpha=" + alpha + ", window=" + win + ", negative=" + neg);
        Node2VecLocalModel.trainLocalBatch(alpha, win, neg);
    }
    break;
case Node2VecLocalModel.CLEAR_LOCAL_WALKS:
        Node2VecLocalModel.clearLocalWalks();
        if (this.localWalks != null) {
            this.localWalks.clear();
        }
        break;
case Node2VecLocalModel.COLLECT_EMBEDDING_FROM_LOCAL_MODEL:
        float[] emb = Node2VecLocalModel.getEmbeddingForNode(this.ItemID);
        if (emb != null) {
            // Copy to avoid reference issues if static model changes
            this.setEmbedding(java.util.Arrays.copyOf(emb, emb.length));
        }
        break;
case Node2VecLocalModel.PACK_WEIGHTS:
        return Node2VecLocalModel.packWeights();
case Node2VecLocalModel.REDUCE_WEIGHTS:
        if (argument instanceof float[]) {      You, last week • Uncommitted changes
            Node2VecLocalModel.updateModelWeights((float[]) argument);
        }
        break;
```

Figure-4: Case handling logic in PropertyVertexPlace class

3.2.3. Node2Vec.java: The Orchestrator

This is the driver class that manages the distributed lifecycle. We implemented a strict synchronization barrier logic:

1. **Broadcast**: Send latest global weights to all nodes.
2. **Train**: Order all nodes to train for one epoch on their local data.
3. **Pack & Reduce**: Gather all local updates, average them, and update the global model.



```java
private void trainDistributed() {
    System.out.println("Starting distributed training using compute-node-centric approach...");

    // Training loop
    for (int epoch = 0; epoch < epochs; epoch++) {
        long start = System.currentTimeMillis();

        // 1. Train local batches on each computing node
        // We use the trainingPlacesRef (one place per node) to trigger this.
        // Arguments: alpha, windowSize, numNegatives
        Object[] args = new Object[] { learningRate, contextSize, numNegativeSamples };

        // This call will block until all nodes finish their local training pass
        trainingPlacesRef.callAll(Node2VecLocalModel.TRAIN_LOCAL_BATCH, (Object) args);

        // 2. Exchange and Synchronize models
        // a. Pack weights from all nodes
        // We pass null args because packWeights doesn't need input
        Object[] packedResults = trainingPlacesRef.propertyGraphCallAll(Node2VecLocalModel.PACK_WEIGHTS, trainingPlacesRef.getArguments(arg: null));

        // b. Collect valid weights and reduce locally on Master
        float[] averagedWeights = null;
        int count = 0;

        for (Object o : packedResults) {
            if (o instanceof float[]) {
                float[] w = (float[]) o;
                if (averagedWeights == null) {
                    averagedWeights = new float[w.length];
                }
                if (w.length != averagedWeights.length) continue;

                for (int i = 0; i < w.length; i++) {
                    averagedWeights[i] += w[i];
                }
                count++;
            }
        }

        // c. Broadcast averaged weights to all nodes
        if (averagedWeights != null && count > 0) {
            // Average
            for (int i = 0; i < averagedWeights.length; i++) {
                averagedWeights[i] /= count;
            }

            // Broadcast the single averaged array
            Object[] reduceArgs = trainingPlacesRef.getArguments(averagedWeights);
            trainingPlacesRef.propertyGraphCallAll(Node2VecLocalModel.REDUCE_WEIGHTS, reduceArgs);
        }

        long end = System.currentTimeMillis();
        System.out.printf("  Epoch %d/%d completed in %d ms%n", epoch + 1, epochs, (end - start));

        // Decay learning rate
        learningRate *= 0.95;
    }
    System.out.println("Distributed training complete.");
}
```

Figure-5: Decentralized training logic in Node2Vec class

This "Pack-Reduce-Broadcast" loop ensures that while nodes train independently (async), they remain mathematically converged over time.

3.3. Application Layer Integration (mass_java_appl)

To demonstrate the usability of this system, we integrated it into the GraphManager workflow.

3.3.1. GraphManager.java Modifications

We introduced a secondary set of MASS Places called trainingPlaces.

- **Why?** The main graph might have millions of vertices. We don't want to spawn millions of training threads.
- **Solution**: We create exactly **one trainingPlace per computing node**. The Node2Vec driver interacts with these trainingPlaces to trigger the node-level static operations. This is a crucial optimization that decouples graph size from training overhead.

3.3.2. KNNPipeline.java

We updated the K-Nearest Neighbors pipeline to seamlessly perform Node2Vec embedding before analysis. The system now:

1. Loads the Graph.
2. **Computes Embeddings via Distributed Node2Vec.**
3. Uses the resulting vectors to find similar nodes (KNN).

4. Execution Workflow Summary

The implementation follows this robust execution path, which has been verified in our local test environment:

1. **Initialization**:
    - GraphManager boots up MASS.
    - Graph data is loaded into PropertyGraphPlaces.
    - trainingPlaces are initialized (1 per Node).
2. **Vocabulary Build**:
    - All vertices report their existence.
    - Master builds a mapping (NodeID -> Index).
3. **Random Walk Phase**:

- Agents are spawned at every vertex.
- Agents hop ‘walk length’ times using the logic in PropertyGraphAgent.
- When an agent finishes, it dumps its walk path into the allLocalWalks list of the Node2VecLocalModel where it originated.

4. **Training Phase (Skip-Gram)**:

    - For E epochs:
        - Master triggers TRAIN_LOCAL_BATCH.
        - Each JVM trains on the walks stored locally.
        - Master triggers PACK -> Average -> BROADCAST.

5. **Collection**:

    - Vertices query their final embeddings and store them for the KNN application.

```
public Map<String, float[]> learnFeatures() {
    if (placesRef == null) {
        throw new IllegalStateException("Node2Vec must be run in distributed mode with a valid PropertyGraphPlaces reference.");
    }
    // build vocab
    buildVocabulary();
    // init embeddings
    initializeEmbeddings();
    // generate walks (use agent-based generation when we have a places reference)
    System.out.println("Using agent-based walk generation.");
    generateWalksWithAgents();
    // train
    trainSkipGram();

    // If distributed, trigger places to collect their own embeddings from the local static model
    System.out.println("Collecting embeddings from distributed static models...");
    placesRef.callAll(Node2VecLocalModel.COLLECT_EMBEDDING_FROM_LOCAL_MODEL);
    return new HashMap<>(); // Return empty map as data is distributed
}
```

Figure-6: Logic Flow in Node2Vec class

5. Conclusion & Future Work

5.1. Conclusion

This term, we successfully enabled **Distributed Node2Vec on MASS**. By architecting a solution that respects the constraints of distributed systems, specifically by minimizing

network synchronization through a compute-node-centric design we have laid the groundwork for training graph embeddings on datasets that exceed the memory of a single machine. The core logic for biased walking, distributed state management, and weight synchronization is now complete and integrated into MASS.

5.2. Future Work (Winter)

With the engine built, the next steps focus on validation and optimization:

- **Verification**: Rigorous unit testing of the p and q bias logic, back propagation, training etc.
- **Benchmarking**: Running the system on standard datasets and comparing accuracy/runtime against standard single-machine implementations & other embedding generation algorithms on MASS.
- **Optimization**: Implementing "Triangle Checks" in the agent logic to strictly enforce 2nd-order properties without carrying excessive state.
- **Hyperparameter Tuning**: Experimenting with synchronization frequency (syncing every N batches instead of every epoch) to further improve speed.