Agent-based Link Prediction in Graph Database

Sumit Hotchandani

Term Report submitted in partial fulfillment of the requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

March 22, 2025

Project Committee

Dr. Munehiro Fukuda, Committee Chair

Dr. Min Chen, Committee Member

Dr. Wooyoung Kim, Committee Member

Contents

1 Introduction

Graph databases are pivotal in managing complex datasets characterized by intricate relationships, utilizing graph structures where nodes represent entities and edges depict relationships. This structure is particularly advantageous for applications requiring analysis of interconnected data, such as social networks, bioinformatics, and recommendation systems. The Multi-Agent Spatial Simulation (MASS) library has been instrumental in developing agent-based graph database systems by enabling parallel computation and efficient data management across distributed environments.

Lilian Cao's project enhances an existing agent-based graph database system using the MASS library. The enhancements involve adopting the property graph model and integrating the Cypher query language to support complex datasets and sophisticated querying capabilities. Concurrently, Chris Ma's work on a multi-user distributed shared graph provides a robust framework for managing graph data in distributed environments, ensuring scalability and performance through efficient data distribution and parallel processing[1][2].

The introduction of link prediction within this enhanced graph database system is driven by the need to predict potential connections in various domains, such as social networks, e-commerce, and biological networks. Link prediction involves forecasting the likelihood of future connections between nodes based on existing data. Traditional methods often face challenges with scalability and computational efficiency when applied to large-scale graphs[11].

To address these challenges, the project integrates Fast Random Projection (FastRP) for node embeddings with MASS's parallel computation capabilities. FastRP efficiently generates node embeddings that capture high-order proximities within the graph, essential for accurate link prediction as they reflect both local and global graph structures[11]. By leveraging MASS's ability to handle distributed computations, the project aims to develop a scalable and efficient link prediction system.

Incorporating topological link prediction methods further enhances this system by utilizing the graph's topology to predict new relationships. Topological methods like Adamic-Adar, Preferential Attachment, etc. complement FastRP's embedding-based approach by directly leveraging the network structure for predictions. This dual approach hopes to achieve improvements in fields relying on accurate link prediction, such as recommendation systems and semantic search.

Building upon Lilian Cao's enhancements and Chris Ma's distributed framework, this project seeks to create a comprehensive solution that addresses the limitations of traditional link prediction methods while capitalizing on agent-based modeling and distributed computing strengths.

2 Existing Work

2.1 GraphDB in MASS

Shenyan Cao focused on improving an existing agent-based graph database (DB) system using the Multi-Agent Spatial Simulation (MASS) Java library[1]. The project addresses limitations in the current system by re-engineering it to handle complex datasets in line with the property graph model and integrating the Cypher query language to enhance querying capabilities.

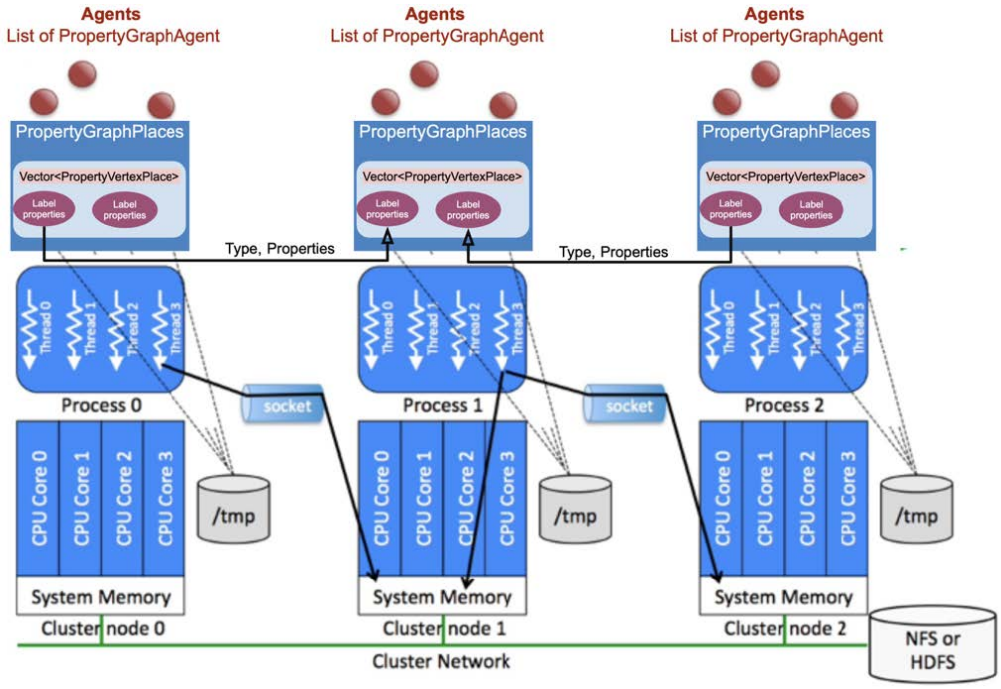

Figure 1: Enhanced agent-based graph DB system with PropertyGraphPlaces.[1]

The system was redesigned to align with the property graph model, enabling it to manage nodes, relationships, and their associated properties effectively. This change supports the storage and manipulation of rich data structures.

The integration of Cypher, a query language optimized for property graph models, is a significant advancement. This enables users to perform CREATE and MATCH operations on the graph DB, facilitating more sophisticated data manipulation and retrieval.

A new PropertyGraphAgent class was introduced to support efficient path traversal and query execution, minimizing inter-node communication and improving system performance.

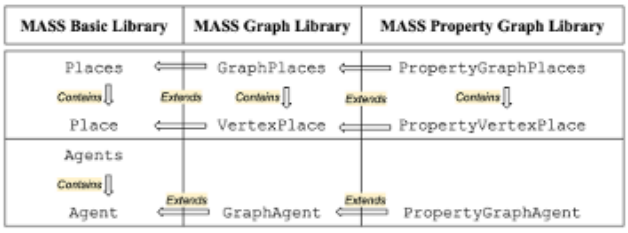

Figure 2: Enhanced property graph DB system leveraging MASS Java library.[1]

The re-engineered system was evaluated using test cases that validated its ability to build property graphs, execute Cypher queries, and traverse the graph efficiently. These tests demonstrated the system's capability to handle complex datasets and query operations in a distributed environment.

While the project significantly enhances the agent-based graph database (DB) system's ability to handle complex datasets and improve querying capabilities through Cypher integration, an additional layer for generating actionable insights is necessary. This can be achieved through the implementation of link prediction methods. Link prediction can provide valuable insights by forecasting potential connections within the graph, which is crucial for applications such as recommendation systems, social network analysis,

and fraud detection. By incorporating topological and embedding-based link prediction techniques, the enhanced system can offer richer analysis and deeper understanding of the underlying data relationships, thereby maximizing its utility in real-world applications.

2.2 Distributed Shared Graph

This work aims to address the limitations of existing agent-based modeling (ABM) libraries, which lack support for multi-user concurrent access and modification of distributed graph data. The project builds on prior research in graph computing within the MASS framework and integrates novel features to enhance performance, scalability, and usability.

The `getVertex()` function is a crucial component of the multi-user distributed shared graph (DSG) system implemented within the MASS Java library, as outlined in Yuan Ma's whitepaper[2]. This function is responsible for accessing vertex data within the distributed graph structure, which is stored in an adjacency list format across multiple computing nodes.

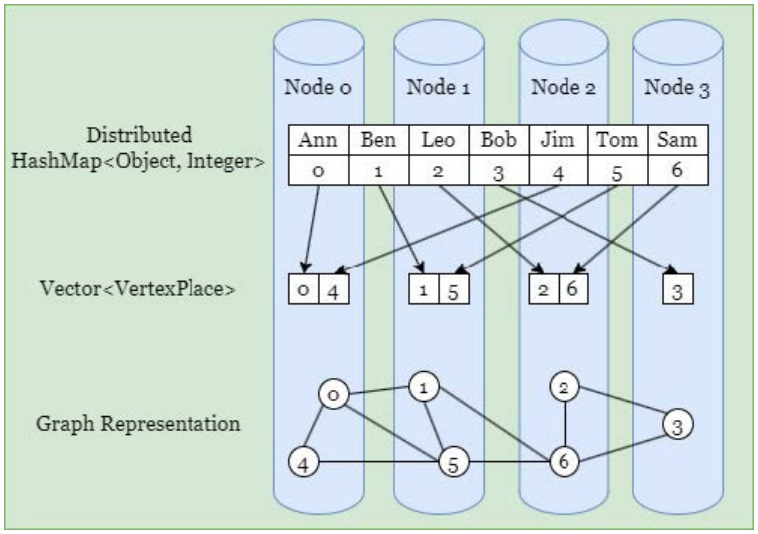

Figure 3: Graph representation by distributed map and vector.[2]

As shown in Figure 3, the DSG system, graph data is stored using distributed HashMaps on each computing node. Each vertex is represented as a key-value pair, where the key is the vertex identifier and the value is the vertex object. This structure facilitates efficient retrieval of vertex information using the `getVertex()` method.

The `getVertex()` method retrieves a vertex's data by accessing its corresponding entry in the distributed HashMap. This operation involves locating the correct computing node where the vertex resides, based on a hash function that evenly distributes vertices across nodes.

The performance of `getVertex()` was evaluated against Hazelcast's implementation of graph storage. The DSG system demonstrated superior performance in both single-node and multi-node scenarios, attributed to its efficient in-memory storage and minimal overhead from fault tolerance mechanisms like those used in Hazelcast.

The `getVertex()` function plays a crucial role in implementing topological link prediction methods. It retrieves the target vertex, enabling the exploration of its neighbors. This step is essential for identifying common neighbors, which are then processed and streamed for subsequent analysis.

3 Related Work

3.1 Neo4j

Neo4j is a leading graph database system that enhances graph analytics with its advanced Graph Data Science (GDS) library, offering robust tools for topological link prediction and node embeddings.

For link prediction, the GDS library includes a pipeline that leverages machine learning models to predict potential relationships between nodes based on graph topology[5]. It extracts features from node pairs, such as common neighbors, and the Adamic-Adar index, and trains classifiers like logistic regression to determine the likelihood of link formation. Neo4j also supports unsupervised methods by applying similarity metrics, such as Preferential Attachment and Common Neighbors, to rank potential links.

Additionally, Neo4j's Fast Random Projection (FastRP) algorithm[6] generates efficient low-dimensional node embeddings by projecting high-dimensional adjacency matrices into lower dimensions using random projections. FastRP preserves the structural properties and relationships within the graph, making it highly scalable for large graphs.

These embeddings are valuable for downstream tasks like link prediction, as they encode local and global structural information and can be combined with other features to improve prediction accuracy by incorporating both topological and property-based insights.

3.2 TigerGraph

TigerGraph is a prominent graph database system known for its scalability and high performance in graph analytics, with significant contributions to topological link prediction and node embeddings.

For link prediction, TigerGraph provides built-in algorithms to compute similarity scores between node pairs based solely on graph topology, utilizing methods such as Common Neighbors, Resource Allocation , and the Adamic-Adar Index[7]. These purely topological approaches are effective even in the absence of detailed node attributes, and the platform's distributed architecture ensures these computations scale efficiently across large datasets.

Additionally, TigerGraph implements Fast Random Projection (FastRP)[8] as part of its node embedding algorithms, generating low-dimensional embeddings by applying random projections to high-dimensional adjacency matrices. Designed for computational efficiency, FastRP enables real-time analytics on massive graphs by capturing meaningful structural patterns, which can then be applied to tasks like clustering, classification, and link prediction.

In TigerGraph's ecosystem, these embeddings can be directly used in queries or exported for external machine learning pipelines, making the platform particularly well-suited for scalable and efficient link prediction in large networks.

4 Design and Implementation

4.1 Design Principles

The primary design goal is to integrate link prediction functionality directly into the MASS Core library rather than implementing it as a standalone MASS application.

This approach ensures that the link prediction algorithms are accessible as foundational components of the MASS framework, enabling broader utility across various applications.

1. **Layered Architecture:** As shown in Figure 4, link prediction algorithms (e.g., Adamic-Adar, Common Neighbors, Resource Allocation) are implemented as a layer on top of the existing graph database system. This design leverages the `PropertyGraphPlaces`, `PropertyVertexPlace`, and `PropertyGraphAgent` classes to provide seamless access to graph traversal and analysis capabilities. Additionally, the FastRP (Fast Random Projection) graph embedding algorithm is integrated as a foundation within this layer,

offering dimensionality reduction capabilities that transform complex graph structures into dense vector representations. The embedding layer interfaces directly with the property graph components to efficiently extract both topological structure and semantic node attributes from the underlying graph.

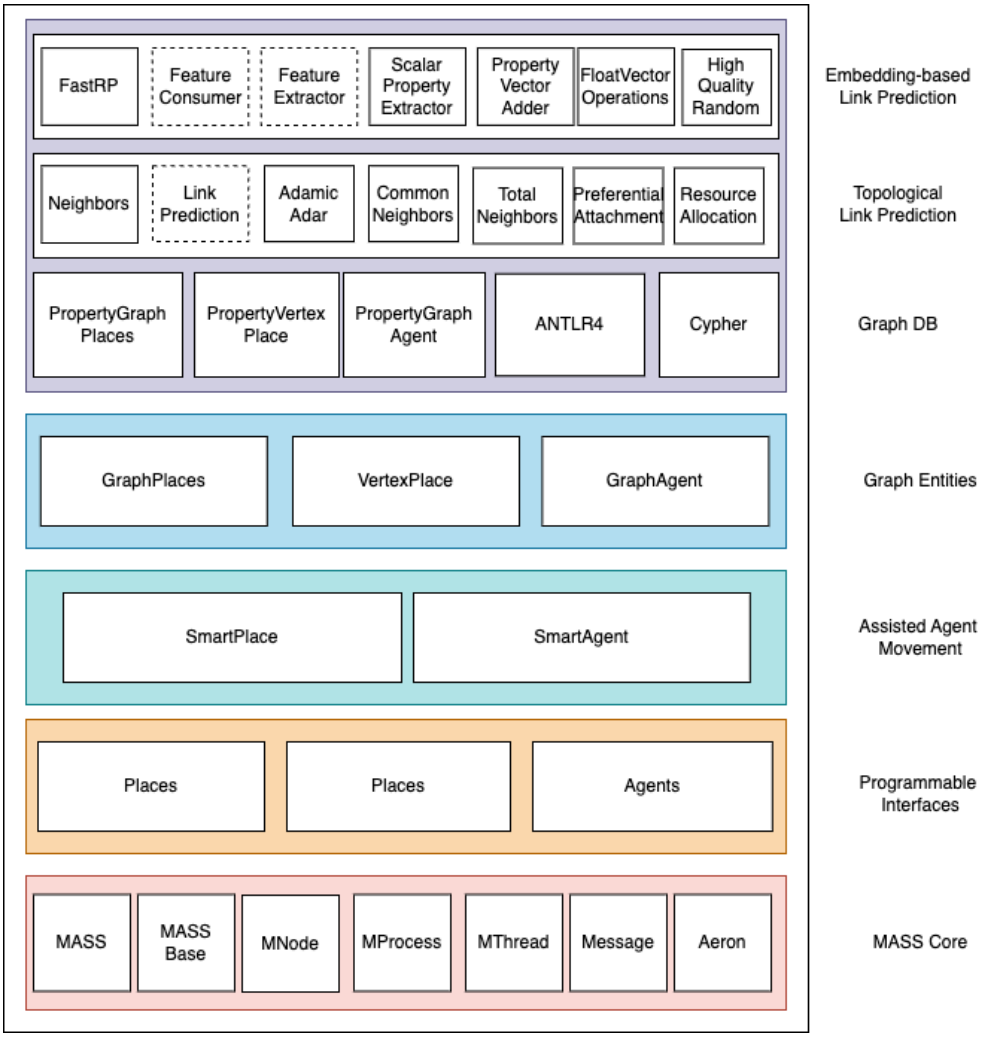

Figure 4: The MASS Stack.

2. **Reusable Functions for Broader Applications:** By embedding link prediction methods within the core library, researchers and developers can use these algorithms as building blocks for custom analyses. For example: Developers can pipeline link prediction results into machine learning workflows for tasks like recommendation systems or fraud detection. Researchers can extend or combine these algorithms with other MASS functionalities to explore novel graph-based insights. The FastRP implementation further enhances this reusability by providing configurable embedding parameters (dimension size, normalization strength, iteration weights) that allow users to fine-tune the balance between computational requirements and embedding quality. These embeddings serve as versatile intermediate representations that can power downstream tasks beyond link prediction, including node classification, clustering, and similarity search.
3. **Scalability and Distributed Processing:** The integration into MASS Core ensures that link prediction algorithms benefit from MASS' distributed memory architecture, enabling efficient execution on large-scale graphs across multiple computing nodes. FastRP particularly leverages this architecture through its agent-based implementation, where `PropertyGraphAgents` migrate between vertices to

collect and propagate embeddings through the network. This distributed approach to embedding computation follows a multi-phase process of random vector initialization followed by iterative neighborhood information propagation, all orchestrated through MASS' agent migration and messaging systems. The embedding vectors are progressively refined through weighted accumulation of neighbor information across iteration boundaries, enabling efficient parallel processing even for very large graphs.

4.2 Topological Link Prediction

Topological link prediction is a critical area of research in network science and machine learning, focusing on predicting the likelihood of future connections between nodes in a network based solely on its current topology, as shown in Figure 5. This task is essential for understanding the dynamics of social networks, biological networks, and knowledge graphs, among others. The problem of link prediction was formalized by Liben-Nowell and Kleinberg, who demonstrated that network topology alone can provide significant insights into potential future interactions[3]. This concept has broad applications, from recommending new friends on social media platforms to identifying potential collaborations in scientific research.

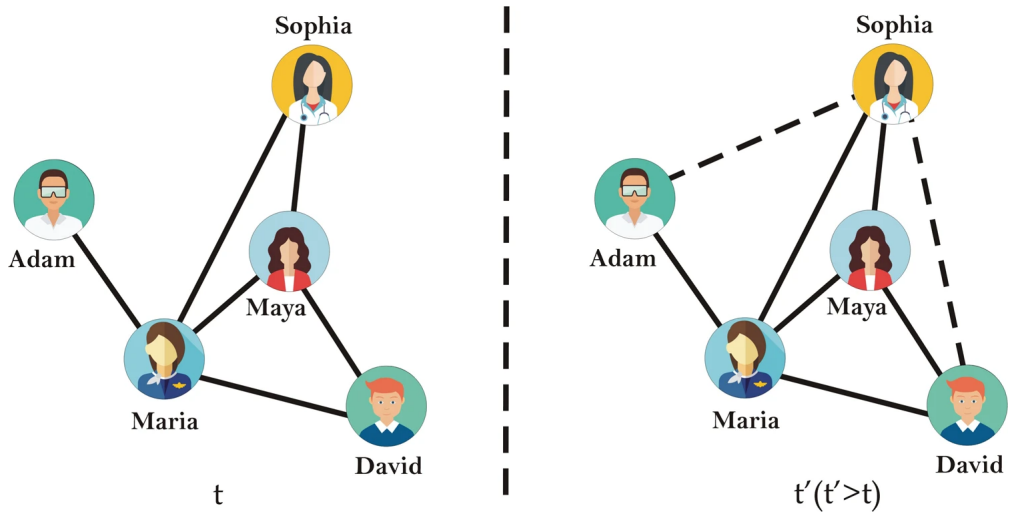

Figure 5: Graphical representation of missing link prediction; dashed lines depict possible edges.[4]

4.2.1 Neighbors Class

The `Neighbors` class is a key component in the link prediction functionality, responsible for managing and retrieving neighbor-related data within a property graph. This class is designed to work seamlessly with the MASS framework, leveraging its distributed architecture to efficiently handle graph operations.

Core Functionalities

1. **Finding Degree of a Vertex:** Determines the number of connections (degree) of a vertex based on edge direction (`TO`, `FROM`, or `BOTH`) and relation type. It uses a helper method `filterNeighbors` based on the Direction input as highlighted in Listing 1 to count relevant neighbors.If the vertex is null, it logs an error and returns -1.

```
switch (direction) {
    case TO:
        degree = this.filterNeighbors(vertex.getTONeighbors(), relation).size();
        break;
    case FROM:
        degree = this.filterNeighbors(vertex.getFROMNeighbors(), relation).size();
        break;
    case BOTH:
        degree = this.filterNeighbors(vertex.getTONeighbors(), relation).size() +
                 this.filterNeighbors(vertex.getFROMNeighbors(), relation).size();
```

```
        break;
}
```

Listing 1: Finding degree of a vertex

2. **Common Neighbors:** The `commonNeighbors` method identifies shared neighbors between two vertices, facilitating link prediction by highlighting potential connections. It begins by retrieving the `PropertyVertexPlace` objects for the two vertices using the `getVertex` method (lines 1–2 in Listing 2) and finds neighbors of each vertex using `findNeighbors` before returning an intersection of both sets of neighbors(line 4-8)

```
PropertyVertexPlace x = this.getVertex(vertexId1);
PropertyVertexPlace y = this.getVertex(vertexId2);

Set xNeighbors = this.findNeighbors(x, relation, direction);
Set yNeighbors = this.findNeighbors(y, relation, direction);

Set intersection = new HashSet<>(xNeighbors);
intersection.retainAll(yNeighbors);

commonNeighbors = intersection.stream()
    .map(node -> (PropertyVertexPlace) this.propertyGraphPlaces.getVertex(node))
    .collect(Collectors.toSet());
```

Listing 2: Finding common neighbors

3. **Total Neighbors:** The `totalNeighbors` method computes the union of neighbors for two vertices, providing a comprehensive view of their combined network. It begins by retrieving the `PropertyVertexPlace` objects for the two vertices using the `getVertex` method (lines 1–2 in Listing 3) and finds neighbors of each vertex using `findNeighbors` before returning a union of both sets of neighbors(line 4-8).

```
PropertyVertexPlace x = this.getVertex(vertexId1);
PropertyVertexPlace y = this.getVertex(vertexId2);

Set xNeighbors = this.findNeighbors(x, relation, direction);
Set yNeighbors = this.findNeighbors(y, relation, direction);

Set union = new HashSet<>(xNeighbors);
union.addAll(yNeighbors);

totalNeighbors = union.stream()
    .map(node -> (PropertyVertexPlace) this.propertyGraphPlaces.getVertex(node))
    .collect(Collectors.toSet());
```

Listing 3: Finding total neighbors

4. **Filter Neighbors:** Filters a map of neighbors based on a specific relation. This private method filters a map of neighbor entries to return only those that contain the specified relation, it uses Java Streams to process entries, checking each entry's value set for the presence of the given relation. The keys of matching entries are collected into a set, which represents the filtered neighbors.

```
private Set filterNeighbors(Map neighbors, String relation) {
    // filter map of neighbors based on relation
    return neighbors.entrySet().stream()
                .filter(e -> ((Set<?>) e.getValue()).contains(relation))
                .map(Map.Entry::getKey)
                .collect(Collectors.toSet());
}
```

Listing 4: Filtering neighbors

The `Neighbors` class provides foundational operations like finding common neighbors and calculating degrees, which are essential inputs for algorithms like Adamic-Adar, Resource Allocation, and Preferential Attachment.

4.2.2 LinkPrediction Interface

The LinkPrediction interface defines the contract for all link prediction algorithms. It ensures that any algorithm implementing this interface adheres to a consistent structure, making it easier to integrate new algorithms into the system.

The interface defines a single method, `calculateSimilarity`, which calculates a similarity score between two vertices in the graph. This score is used to predict potential links between them.

Parameters:

- `vertexId1` and `vertexId2`: Unique identifiers for the two vertices being compared.
- `relation`: Specifies the type of relationship (e.g., "friendship" or "collaboration") to consider during similarity calculation.
- `direction`: Indicates whether to consider outgoing edges (`TO`), incoming edges (`FROM`), or both (`BOTH`).

The interface provides a unified API for all link prediction algorithms, ensuring consistency across implementations. It allows developers to implement new algorithms without modifying existing code, adhering to the Open/Closed Principle (OCP) of SOLID design.

4.2.3 AbstractLinkPrediction Class

The `AbstractLinkPrediction` class serves as a base class for implementing specific link prediction algorithms. It provides shared functionality and utilities that are commonly needed across different algorithms, reducing code duplication and improving maintainability.

The class includes an instance of the `Neighbors` class to handle neighbor-related operations like finding common neighbors or calculating degrees. This integration allows all subclasses to leverage the efficient neighbor management functionality provided by Neighbors.

Provides default values for edge direction (`BOTH`) and relationship (empty string). These defaults simplify algorithm implementation by handling cases where parameters are not explicitly provided.

The abstract class acts as a common foundation for all link prediction algorithms, encapsulating shared logic and utilities. It enforces adherence to the `LinkPrediction` interface while allowing subclasses to focus on their specific algorithmic logic. By centralizing common functionality, it reduces redundancy and improves maintainability.

4.2.4 Link Prediction Algorithms

This section provides a detailed explanation of the link prediction algorithms implemented in MASS Core. All algorithms have a time complexity of $O(D1 + D2)$, where D1 and D2 are the degrees of the two vertices.

1. **Adamic-Adar:** The Adamic-Adar index[14] is a similarity measure used to predict links between nodes in a graph. It assigns higher weights to less common neighbors, if rare connections provide more meaningful information.

$$A(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{log|N(u)|}$$

   Let's consider two nodes, X and Y, that we aim to predict a link between, N(u) is the set of common neighbors of X and Y. Once the common neighbors have been identified, the index is computed as the sum of the reciprocals of the logarithms of the degrees of each neighbor.

   A neighbor with a lower degree (fewer connections) provides more specific or unique information about a potential link between two nodes than a highly connected neighbor. In other words, if a common neighbor is connected to many nodes in the network, their presence as a shared neighbor is less informative.

The `AdamicAdar` class computes this index by iterating over common neighbors and applying the formula defined above and implemented in line 9 of Listing 5.

```
1  public double calculateSimilarity(String vertexId1, String vertexId2, String relation,
2      Direction direction) {
3      double result = 0.0;
4      try {
5          Direction dir = this.getDirection(direction);
6          String rel = this.getRelationship(relation);
7          Set commonNeighbors = this.neighbors.commonNeighbors(vertexId1, vertexId2,
8              rel, dir);
9          result = commonNeighbors.stream().mapToDouble(nb -> 1.0 /
10             Math.log(this.neighbors.degree(nb, rel, dir))).sum();
11     } catch (Exception e) {
12         System.out.println("Error in adamicAdar: " + e);
13     }
14     return result;
15 }
```

Listing 5: Adamic-Adar similarity

2. **Resource Allocation:** The Resource Allocation Index[15] is a network-based measure that predicts the likelihood of a link between two nodes by distributing "resources" from one node to another via their common neighbors.

$$RA(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$

Let's consider two nodes, X and Y, that we aim to predict a link between, N(u) is the set of common neighbors of X and Y. The common neighbors act as "resource distributors." Nodes with fewer neighbors distribute more resources, making them more influential in predicting a link.

The algorithm gives higher weight to common neighbors with fewer connections (similar to Adamic-Adar) and focuses on local network structure for link prediction.

In the `ResourceAllocation` class, this index is calculated similarly to Adamic-Adar but uses a simpler division by degree (lines 8-9 in Listing 6).

```
1  public double calculateSimilarity(String vertexId1, String vertexId2, String relation,
2  Direction direction) {
3
4      Direction dir = this.getDirection(direction);
5      String rel = this.getRelationship(relation);
6      Set commonNeighbors = this.neighbors.commonNeighbors(vertexId1, vertexId2,
7          rel, dir);
8      result = commonNeighbors.stream().mapToDouble(nb -> 1.0 /
9          (this.neighbors.degree(nb, rel, dir))).sum();
10 }
```

Listing 6: Resource Allocation similarity

3. **Preferential Attachment:** Preferential Attachment[16] predicts links based on the product of degrees of two nodes. This principle is based on the idea that nodes with higher degrees are more likely to form new links:

$$PA(x, y) = |N(x)| * |N(y)|$$

This measure is particularly useful in networks where some nodes have significantly more connections than others.

The `PreferentialAttachment` class calculates this metric by multiplying the degrees of the two nodes (lines 7-8 in Listing 7).

```
1  public double calculateSimilarity(String vertexId1, String vertexId2, String relation,
2      Direction direction) {
3      double result = 0.0;
```

```
    try {
        Direction dir = this.getDirection(direction);
        String rel = this.getRelationship(relation);
        result = this.neighbors.degree(vertexId1, rel, dir) *
            this.neighbors.degree(vertexId2, rel, dir);  // Product of degrees
    } catch (Exception e) {
        System.out.println("Error in preferentialAttachment: " + e);
    }
    return result;
}
```

Listing 7: Preferential Attachment similarity

4. **Common Neighbors:** The Common Neighbors algorithm measures the similarity between two nodes based on the number of neighbors they share. Formally, if $N(x)$ and $N(y)$ represent the sets of neighbors for nodes x and y, the similarity score is given by:

$$CN(x, y) = |N(x) \cap N(y)|$$

This simple yet effective measure assumes that nodes sharing more neighbors are more likely to be similar or connected.

In the `CommonNeighbors` class, the algorithm calculates the number of shared neighbors between two vertices. The implementation of the `commonNeighbors` function (line 7 in Listing 8), was discussed in Listing 2.

```
public double calculateSimilarity(String vertexId1, String vertexId2, String relation,
    Direction direction) {
    double result = 0.0;
    try {
        Direction dir = this.getDirection(direction);
        String rel = this.getRelationship(relation);
        Set commonNeighbors = this.neighbors.commonNeighbors(vertexId1, vertexId2,
            rel, dir);
        result = commonNeighbors.size();  // Number of common neighbors
    } catch (Exception e) {
        System.out.println("Error in common neighbors: " + e);
    }
    return result;
}
```

Listing 8: Common Neighbor similarity

5. **Total Neighbors:** The Total Neighbors algorithm calculates the similarity score between two nodes x and y by determining the size of the union of their neighbor sets. Mathematically, it is expressed as::

$$TN(x, y) = |N(x) \cup N(y)|$$

where, $N(x)$ and $N(y)$ are the sets of neighbors for nodes x and y, respectively.

The `TotalNeighbors` class implements this algorithm by using the `totalNeighbors` method (refer Listing 3 for details) from the `Neighbors` class to find and count unique neighbors as shown in line 8 of Listing 9.

```
public double calculateSimilarity(String vertexId1, String vertexId2, String relation,
    Direction direction) {
    double result = 0.0;
    try {
        Direction dir = this.getDirection(direction);
        String rel = this.getRelationship(relation);
        // Calculate total unique neighbors
        result = this.neighbors.totalNeighbors(vertexId1, vertexId2, rel, dir).size();
    } catch (Exception e) {
        System.out.println("Error in total neighbors: " + e);
    }
```

```
    return result;
}
```

Listing 9: Total Neighbor similarity

4.3 Fast Random Projection

Fast Random Projection (FastRP) represents a significant advancement in graph embedding algorithms, offering linear time complexity while maintaining high-quality embeddings suitable for downstream machine learning tasks. This chapter presents a detailed implementation of FastRP within the Multi-Agent Spatial Simulation (MASS) framework, with particular emphasis on the agent-based propagation mechanism and the mathematical foundations that enable efficient embedding computation across distributed environments.

The core motivation behind integrating FastRP with MASS is to leverage the inherent parallelism of agent-based architectures when processing large graph structures. Unlike traditional graph embedding approaches that rely on expensive random walks (e.g., DeepWalk, Node2Vec) or complex optimization procedures, FastRP utilizes sparse random projections combined with an iterative refinement process that is naturally compatible with distributed agent-based systems.

4.3.1 Mathematical Foundations of FastRP

4.3.1.1 Johnson-Lindenstrauss Lemma

The theoretical underpinning of FastRP comes from the Johnson-Lindenstrauss lemma[9], which provides guarantees on the preservation of pairwise distances when projecting high-dimensional data into lower-dimensional spaces. Formally, the lemma states that for any set of n points in a high-dimensional Euclidean space, there exists a linear mapping into a space of dimension $O(\log(n)/\epsilon^2)$ that preserves all pairwise distances within a factor of $(1 \pm \epsilon)$.

This theoretical foundation enables FastRP to project the adjacency matrix of a graph (which can be viewed as an n-dimensional representation where n is the number of nodes) into a much lower-dimensional space (typically 128-512 dimensions) while preserving the essential structural relationships between nodes.

4.3.1.2 Random Projection Matrices

The random projection in FastRP is implemented using sparse random matrices. For a graph with n nodes, we generate a random projection matrix $\mathbf{R} \in \mathbb{R}^{n \times d}$ where d is the embedding dimension[9]. The sparsity of this matrix is controlled by the parameter `SPARSITY` (typically set to 3), resulting in approximately 2/3 of the entries being zero:

```
public static final int SPARSITY = 3;
public static final double ENTRY_PROBABILITY = 1.0 / (2 * SPARSITY);
```

Listing 10: FastRP constants

Each non-zero entry in the matrix follows a three-valued distribution:

$$R_{ij} = \begin{cases} +\frac{s_i \cdot \sqrt{\text{SPARSITY}}}{\sqrt{d}} & \text{with probability } \frac{1}{2 \cdot \text{SPARSITY}} \\ 0 & \text{with probability } 1 - \frac{1}{\text{SPARSITY}} \\ -\frac{s_i \cdot \sqrt{\text{SPARSITY}}}{\sqrt{d}} & \text{with probability } \frac{1}{2 \cdot \text{SPARSITY}} \end{cases}$$

With α representing the normalization strength (typically in the range [0, 0.5])[6]. This degree-based normalization is crucial for preventing high-degree nodes from dominating the embedding space.

In the implementation, this is reflected in the `computeRandomVector` method:

```
private float[] computeRandomVector(Random random, float entryValue, int embeddingDimension,
    int baseEmbeddingDimension, float[][] propertyVectors,
    List<? extends FeatureExtractor> featureExtractors) {

        // initialize an empty float array of embedding dimension to hold vector
        var randomVector = new float[embeddingDimension];
        // populate with random entries
        for (int i = 0; i < baseEmbeddingDimension; i++) {
            randomVector[i] = EmbeddingUtils.computeRandomEntry(random, entryValue,
                FastRP.ENTRY_PROBABILITY);
        }

        // add node properties to random vector
        PropertyVectorAdder propertyVectorAdder = new PropertyVectorAdder(
            propertyVectors, baseEmbeddingDimension, embeddingDimension);
        propertyVectorAdder.setRandomVector(randomVector);
        FeatureExtraction.extract(this, featureExtractors, propertyVectorAdder);

        return randomVector;
}
```

Listing 11: Random vector computation

The `computeRandomEntry` function implements the three-valued distribution:

```
public static float computeRandomEntry(Random random, float entryValue,
    double ENTRY_PROBABILITY) {
        double randomValue = random.nextDouble();

        // return entryValue based on probability
        if (randomValue < ENTRY_PROBABILITY) {
            return entryValue;
        } else if (randomValue < ENTRY_PROBABILITY * 2.0) {
            return -entryValue;
        } else {
            return 0.0f;
        }
}
```

Listing 12: Random entry computation

This sparse representation offers computational efficiency while maintaining sufficient randomness for dimensionality reduction.

4.3.1.3 Node Property Integration

A distinctive feature of the MASS implementation is the ability to incorporate node properties into embeddings[6]. This is achieved by splitting the embedding dimension into two components:

1. **Base Embedding Dimension:** Captures structural information through random projections.
2. **Property Dimension:** Encodes node attributes through property vectors

The total embedding dimension is divided according to a property ratio parameter:

```
this.embeddingDimension = embeddingDimension;
this.propertyDimension = (int) (propertyRatio * embeddingDimension);
this.baseEmbeddingDimension = embeddingDimension - propertyDimension;
```

Listing 13: Embedding dimension parameters

Property vectors are integrated with the random vectors through the `PropertyVectorAdder` class, which applies a scaling factor to each property and adds it to the corresponding dimension in the embedding:

```
public void acceptScalar(long ignored, int offset, double value) {
    float floatValue = (float) value;
    for (int i = baseEmbeddingDimension; i < embeddingDimension; i++) {
```

```
            randomVector[i] += floatValue * propertyVectors[offset][i - baseEmbeddingDimension];
        }
    }
```

Listing 14: Property vector integration with random vector

This approach allows embeddings to capture both structural information (from the graph topology) and semantic information (from node properties), resulting in more expressive representations.

4.3.2 The MASS Agent-Based Implementation

The core innovation in our implementation is the utilization of MASS's agent-based architecture to distribute the embedding propagation process across nodes. This section details how agents are employed to collect and distribute embedding information throughout the graph.

4.3.2.1 System Architecture

Our implementation consists of three primary components working in concert:

1. **Core FastRP Class (FastRP.java):** Orchestrates the overall embedding process, managing configuration parameters and coordinating the different phases of the algorithm.
2. **Feature Extraction Layer:** Processes node properties into numerical representations that can be incorporated into embeddings, ensuring that both structural and semantic information is captured.
3. **PropertyGraphDB Components[1]:** Handles the computation and propagation of embeddings through agents. This includes:
    - `PropertyGraphPlaces`: Manages the graph structure and initiates agent-based computation.
    - `PropertyVertexPlace`: Represents individual vertices and stores their embeddings.
    - `PropertyGraphAgent`: Mobile agents that collect and propagate embedding information.

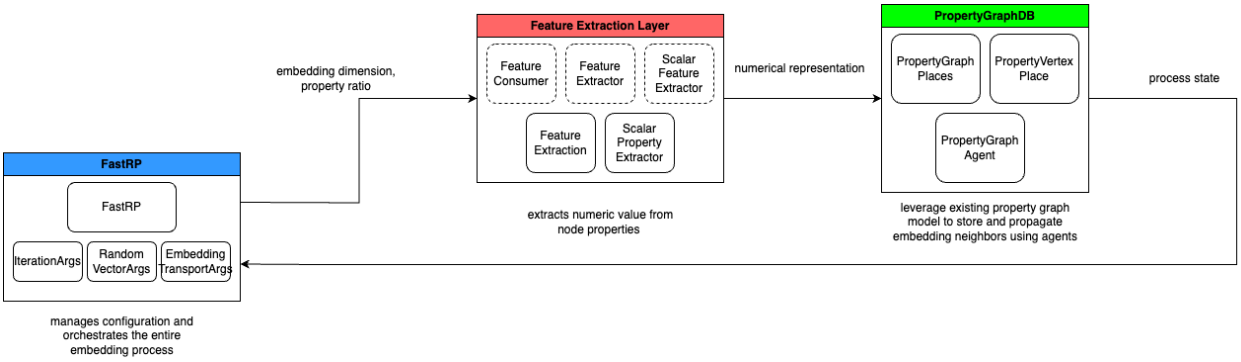

Figure 6: FastRP architecture

The system adheres to SOLID principles, with clear separation of concerns and well-defined interfaces between components. This modular design facilitates maintenance and extensibility, allowing for future enhancements without significant architectural changes.

4.3.2.2 Embedding Initialization Process

The embedding process begins with the initialization of random vectors for each node. This is coordinated by the `FastRP` class through a call to `initRandomVectors()`:

```
private void initRandomVectors() {
    // Create a new instance of RandomVectorParameters to be passed to each node
    RandomVectorArgs params = new RandomVectorArgs(
        this.randomSeed,
        this.normalizationStrength,
        this.embeddingDimension,
        this.baseEmbeddingDimension,
        this.nodeSelfInfluence,
        this.propertyVectors,
        this.featureExtractors);

    // callAll to trigger computation on all the nodes in the graph
    this.propertyGraphPlaces.computeRandomVectors(params);
}
```

Listing 15: Random vector initialization callAll

This initiates a `callAll` operation on the `PropertyGraphPlaces` object, which distributes the computation to all vertices in the graph. Each vertex then executes its `computeEmbedding` method:

```
// generate a random value
var random = new HighQualityRandom(randomSeed);
var sqrtEmbeddingDimension = (float) Math.sqrt(baseEmbeddingDimension);
var sqrtSparsity = (float) Math.sqrt(FastRP.SPARSITY);

// get degree of the node to determine scaling factor
int degree = this.degree();
float scaling = degree == 0 ? 1.0f : (float) Math.pow(degree, normalizationStrength);
// determine entryValue based on scaling factor and SPARSITY constant
float entryValue = scaling * sqrtSparsity / sqrtEmbeddingDimension;
// add node ID to the random seed
random.reseed(randomSeed ^ this.ItemID.hashCode());

// compute random vector and factor in node properties
var randomVector = this.computeRandomVector(random, entryValue, embeddingDimension,
    baseEmbeddingDimension, propertyVectors, featureExtractors);

// add node self influence to the initial embedding
this.embedding = Float.compare(nodeSelfInfluence.floatValue(), 0.0f) != 0 ?
    this.addNodeSelfInfluence(randomVector, nodeSelfInfluence, embeddingDimension) :
    randomVector;
```

Listing 16: Random vector initialization

Several key factors influence the initialization:

1. **Degree-Based Normalization:** The `normalizationStrength` parameter controls how the degree of the node affects the embedding. A higher value reduces the influence of high-degree nodes, preventing them from dominating the embedding space.
2. **Random Seed Management:** The system uses a base random seed that is combined with the node's unique identifier to generate deterministic but node-specific randomization. This ensures consistency across distributed environments.
3. **Node Self-Influence:** This parameter determines how much of a node's original random vector is preserved in subsequent iterations[6]. Higher values maintain node identity, while lower values allow for stronger neighbor influence.

The initialized embeddings serve as the foundation for the subsequent propagation phase, where neighbor information is iteratively incorporated.

4.3.2.3 Agent-Based Embedding Propagation

The most distinctive aspect of our implementation is the agent-based approach to embedding propagation. Unlike traditional matrix-based implementations, we leverage mobile agents to collect and distribute embedding

information throughout the graph. This section provides a detailed examination of the agent movement patterns and the mathematical computations performed during propagation.

1. **Propagation Initialization:** The propagation process is initiated by the FastRP class through the `propagateEmbeddings` method:

   This method initiates multiple iterations of propagation, each with a different weight specified in the `iterationWeights` list. Typically, these weights decrease with each iteration (e.g., [1.0, 0.5, 0.25, 0.125]), giving more importance to closer neighbors and less to distant ones.

   The PropertyGraphPlaces class then orchestrates the agent-based propagation:

```
// initial agent population is equal to the number of nodes in the graph
int initAgentPopulation = MatrixUtilities.getMatrixSize(getSize());

// pack arguments for each agent, initialize them as orchestrators
EmbeddingTransportArgs transportArgs = new EmbeddingTransportArgs(
    -1,
    true,
    -1,
    iterationWeight,
    true
);

// initialize an Agent at each Place object
Agents agents = new Agents(0, PropertyGraphAgent.class.getName(), transportArgs,
    this, initAgentPopulation);

// while all Agents, repeatedly trigger callAll and manageAll to carry out
// the movement phases
while(agents.nAgents() > 0) {
    agents.callAll(1, null);
    agents.manageAll();
}
```

   Listing 17: Agent propgation callAll and manageAll

   This creates an initial population of orchestrator agents, one for each vertex in the graph. These agents are responsible for coordinating the embedding collection process.

2. **Agent Movement Patterns:** The agent movement follows a sophisticated pattern designed to efficiently collect and distribute embedding information. This process involves two types of agents:

   (a) **Orchestrator Agents:** Initialize the process at each vertex and spawn collector agents.

   (b) **Collector Agents:** Travel to neighboring vertices, collect their embeddings, and return to the source.

   The movement pattern is implemented in the `collectEmbeddings` method of the `PropertyGraphAgent` class:

```
PropertyVertexPlace currentPlace = (PropertyVertexPlace) this.getPlace();

// define orchestrators actions, triggered by
// the first iteration of callAll and manageAll - initialization phase
if(isOrchestrator) {
    currentPlace.initResponseCount();
    // set iteration weight
    currentPlace.setIterationWeight(iterationWeight);
    // get all neighbor places
    Map<Object, Object[]> neighbors = currentPlace.getTONeighbors();
    neighbors.putAll(currentPlace.getFROMNeighbors());

    // Create args array for all the agents we'll spawn
    EmbeddingTransportArgs[] args = new EmbeddingTransportArgs[neighbors.size()];
    int i = 0;

```

```
17      // Prepare args for each collector agent, assign each collector
18      // a different neighbor to visit
19      for (Map.Entry<Object, Object[]> neighbor : neighbors.entrySet()) {
20              int neighborId = MASS.distributed_map.getOrDefault(neighbor.getKey(), -1);
21              if (neighborId != -1) {
22                      args[i++] = new EmbeddingTransportArgs(
23                              currentPlace.getIndex()[0],
24                              false,
25                              neighborId,
26                              iterationWeight,
27                              true
28                      );
29              }
30      }
31
32      // Spawn all collector agents at once
33      spawn(args.length, args);
34
35      // initial agent job ends here
36      kill();
37      return null;
38  }
39
40  // define collector agents actions, triggered by the second iteration
41  // of callAll and manageAll - neighbor visit and collect phase
42  if(!hasCollectedEmbedding) {
43      if(currentPlace.getIndex()[0] == sourceNode) {
44          migrate(destinationNode);
45      } else {
46          // Collect embeddings from neighbors
47          neighborEmbedding = currentPlace.getPreviousEmbedding();
48          hasCollectedEmbedding = true;
49          // send back to source
50          migrate(sourceNode);
51          return null;
52      }
53  }
54
55  // triggered by the third iteration of callAll and manageAll - return phase
56  if(hasCollectedEmbedding && currentPlace.getIndex()[0] == sourceNode) {
57      // arrive at source and deposit embedding
58      currentPlace.receiveEmbedding(neighborEmbedding);
59      kill();
60      return null;
61  }
```

Listing 18: Agent movement logic

Several key aspects of this pattern are worth highlighting:

(a) **Initialization Phase:** The orchestrator agent identifies all neighbors (both incoming and outgoing edges) and spawns collector agents with a specialized transport argument for each (refer lines 5-38 in Listing 18).

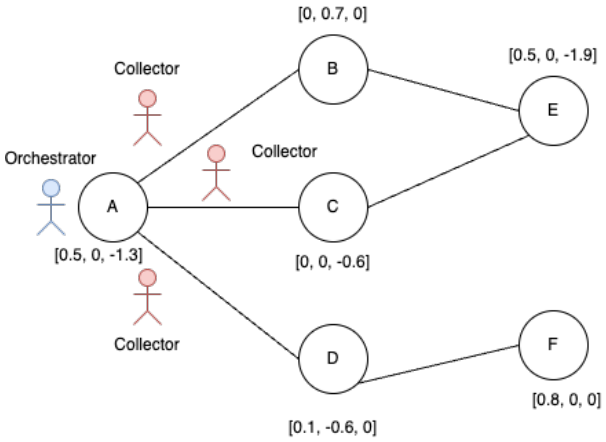

Figure 7: Initialization phase

(b) **Neighbor Visitation:** Collector agents migrate to their assigned neighbors and collect the current embedding (refer lines 42-53 in Listing 18).

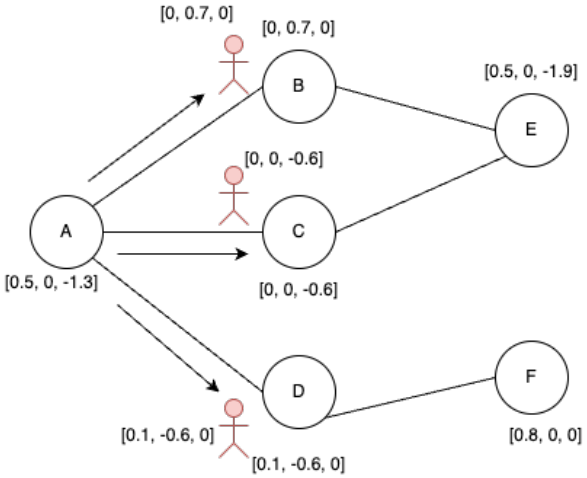

Figure 8: Neighbor visit and collect phase

(c) **Return Phase:** After collecting embeddings, agents return to their source vertex and deliver the embedding information(refer lines 56-61 in Listing 18).

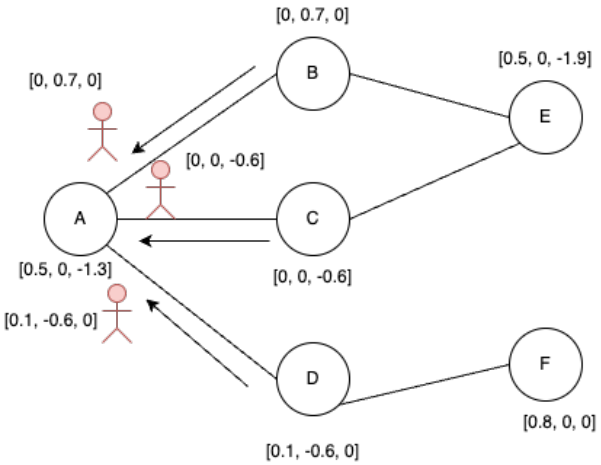

Figure 9: Return phase

(d) **Agent Lifecycle Management:** Agents are terminated once their task is complete, preventing unnecessary resource consumption

This multi-agent approach enables parallel collection of embeddings, efficiently distributing the computational load across the system.

3. **Synchronization Mechanisms:** A critical aspect of the agent-based propagation is the synchronization mechanism that ensures all embeddings are collected before computing the next iteration. This is implemented using an atomic counter in the `PropertyVertexPlace` class:

```
public synchronized void receiveEmbedding(float[] embedding) {
    if(neighborEmbeddings == null) {
        neighborEmbeddings = new ArrayList<>();
    }
    neighborEmbeddings.add(embedding);

    if(responseCount.incrementAndGet() == this.degree()) {
        this.computeFinalEmbedding();
    }
}
```

Listing 19: Agent synchronization mechanism

The `initResponseCount` method initializes an atomic counter:

```
public void initResponseCount() {
    if(responseCount == null) {
        responseCount = new AtomicInteger(0);
    }
}
```

Listing 20: Synchronization counter

This counter tracks the number of collector agents that have returned with embeddings. When the count equals the vertex's degree, the system triggers the final embedding computation for this iteration.

4. **Computation of Final Embeddings:** Once all neighbor embeddings have been collected, each vertex computes its updated embedding using a weighted combination of its current embedding and the aggregated neighbor information. This computation is performed in the `computeFinalEmbedding` method:

```
public void computeFinalEmbedding() {
    // get node degree to cacluclate scaling factor
```

```
    int degree = this.degree();
    float degreeScale = degree == 0 ? 1.0f : 1.0f / degree;

    // Create accumulator for neighbor embeddings
    float[] accumulator = new float[embedding.length];

    // add all neighbor embeddings
    for(float[] neighborEmbedding : neighborEmbeddings) {
        FloatVectorOperations.addInPlace(accumulator, neighborEmbedding);
    }

    // scale the resulting vector according to degree
    FloatVectorOperations.scale(accumulator, degreeScale);

    // normalize the scaled vector
    float l2Norm = FloatVectorOperations.l2Norm(accumulator);
    float adjustedL2Norm = l2Norm < FastRP.EPSILON ? 1f : l2Norm;
    FloatVectorOperations.scale(accumulator, adjustedL2Norm);

    // multiply normalized and scaled vector with iteration weight and
    // add it to node embedding
    FloatVectorOperations.addWeightedInPlace(this.embedding, accumulator,
        this.iterationWeight);

    // Clear collection state for next iteration
    neighborEmbeddings.clear();
    responseCount.set(0);
}
```

Listing 21: Final vector computation

The mathematical operations in this method can be formalized as follows:

- **Aggregation:** Compute the sum of all neighbor embeddings.

$$\mathbf{A}_i = \sum_{j \in \mathcal{N}(i)} \mathbf{Z}_j^{(t-1)}$$

  where $\mathcal{N}(i)$ represents the neighbors of node i, and $\mathbf{Z}_j^{(t-1)}$ is the embedding of node j at the previous iteration.

- **Degree Normalization:** Scale the aggregated embeddings by the inverse of the node's degree.

$$\mathbf{A}_i = \frac{\mathbf{A}_i}{|\mathcal{N}(i)|}$$

  This computes the average embedding of the node's neighbors.

- **L2 Normalization:** Normalize the aggregated embeddings to unit length.

$$\mathbf{A}_i = \frac{\mathbf{A}_i}{||\mathbf{A}_i||_2}$$

  This ensures numerical stability and prevents magnitude explosion over multiple iterations.

- **Weighted Combination:** Combine the node's current embedding with the normalized neighbor embeddings.

$$\mathbf{Z}_i^{(t)} = \mathbf{Z}_i^{(t-1)} + \beta_t \cdot \mathbf{A}_i$$

  where β_t is the iteration weight for iteration t.

This process is repeated for each iteration, with decreasing weights to prioritize closer neighbors. The weights control how much influence each "hop" of neighbors has on the final embedding.

Several important vector operations are utilized in this process, implemented in the `FloatVectorOperations` class.

5 Evaluation

5.1 Experimental Setup

The performance evaluation compared MASS GraphDB with Neo4j on two primary graph analytics tasks: topological link prediction and Fast Random Projection (FastRP) graph embedding. It is important to note that the benchmarks were conducted on different hardware platforms: Neo4j ran on an Apple M2 Pro processor, while MASS benchmarks were performed on Intel Xeon 5150 processors. This hardware disparity creates an inherent disadvantage for MASS in raw computational performance comparisons.

Three standard datasets[17] were selected for the topological link prediction evaluation: OGBL-DDI, Cora, and IMDB. OGBL-DDI is a homogeneous, unweighted, undirected graph containing 4,267 nodes representing FDA-approved or experimental drugs, with 1,334,889 edges indicating synergistic interactions. The Cora dataset is a homogeneous directed citation graph with 2,708 nodes representing academic papers and 5,429 edges representing citation links. Each node in Cora is associated with a subject property categorizing papers into seven categories. The IMDB dataset is a heterogeneous graph from the TU dataset collection, featuring 1,742 nodes representing movie collaborators (actors and directors) connected by 7,870 collaboration relationships. These datasets were chosen primarily because they are built into Neo4j's graph data science library, facilitating direct comparison. For the FastRP evaluation, two datasets were used: the Zachary Karate Club (a small social network) and Cora (a citation network).

5.2 Topological Link Prediction

Topological link prediction performance was assessed by measuring the execution time of five different link prediction algorithms on progressively larger subsets of the datasets. For each dataset, we used random sampling to split the data into 85% training and 15% testing sets. The node pairs for prediction queries were randomly selected regardless of whether a link already existed between them.

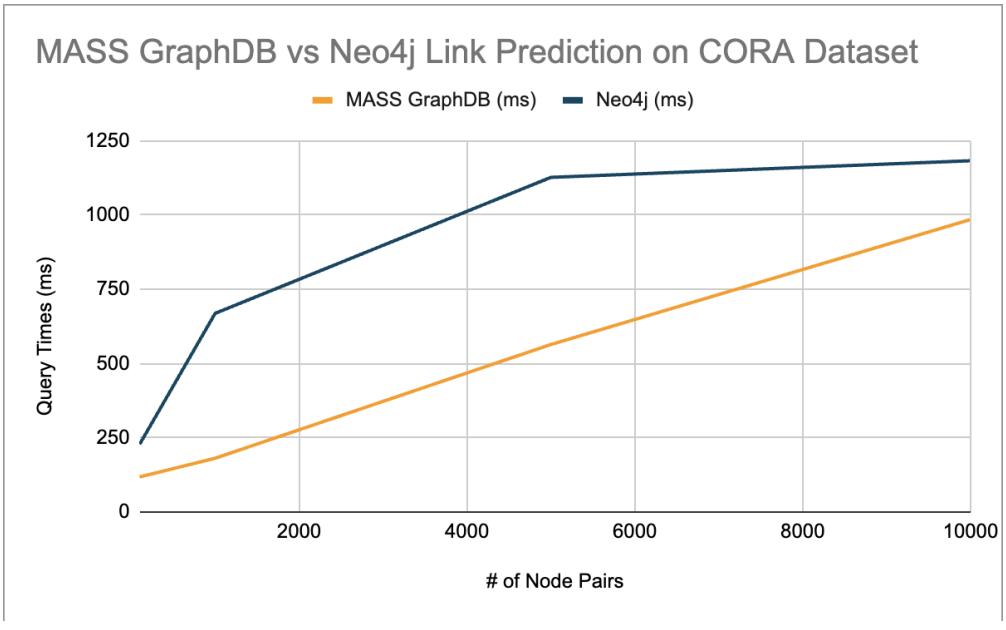

Figure 10: Topological link prediction benchmarks - Cora Dataset

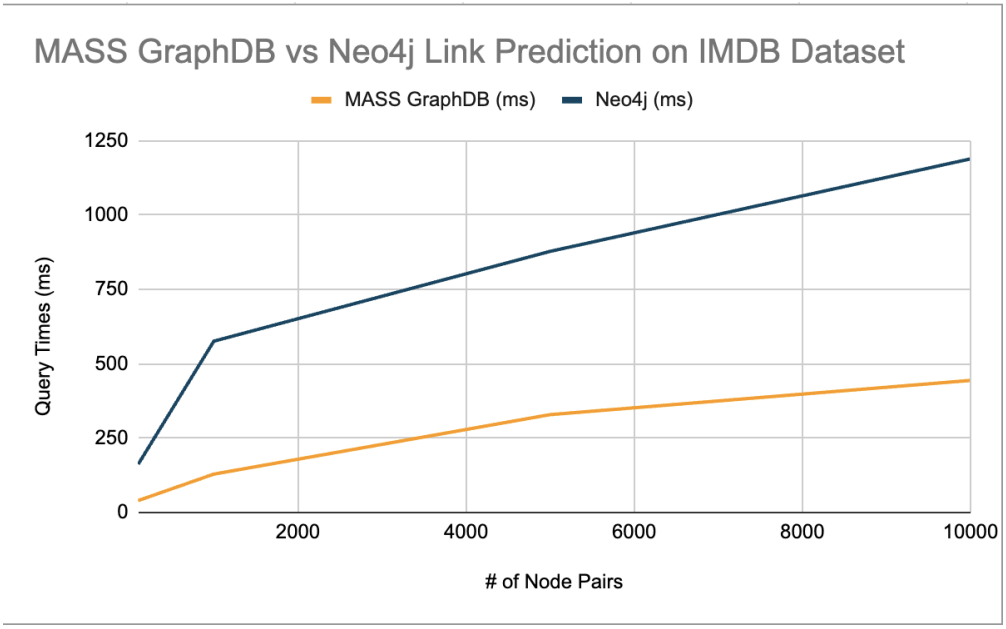

Figure 11: Topological link prediction benchmarks - IMDB Dataset

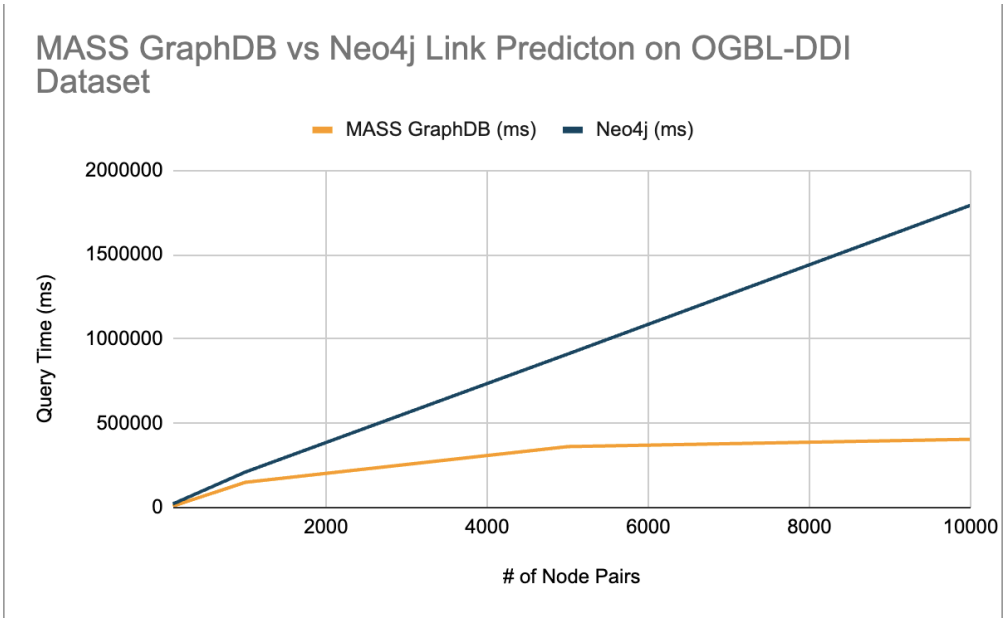

Figure 12: Topological link prediction benchmarks - OGBL-DDI Dataset

The results demonstrate that MASS GraphDB consistently outperforms Neo4j in topological link prediction tasks across all three datasets and at various scales. This performance advantage is particularly significant for larger graph sizes, as seen in the OGBL-DDI dataset with 10,000 nodes, where MASS is over 4 times faster than Neo4j. A key factor in MASS's superior performance is its in-memory graph representation. While Neo4j pulls data from disk for each query, MASS GraphDB keeps the entire graph in memory, enabling faster vertex retrieval through direct access. This memory locality advantage enables MASS to execute link prediction algorithms more efficiently, despite running on older hardware.

5.3 FastRP performance

For the FastRP graph embedding algorithm, a different performance pattern emerged:

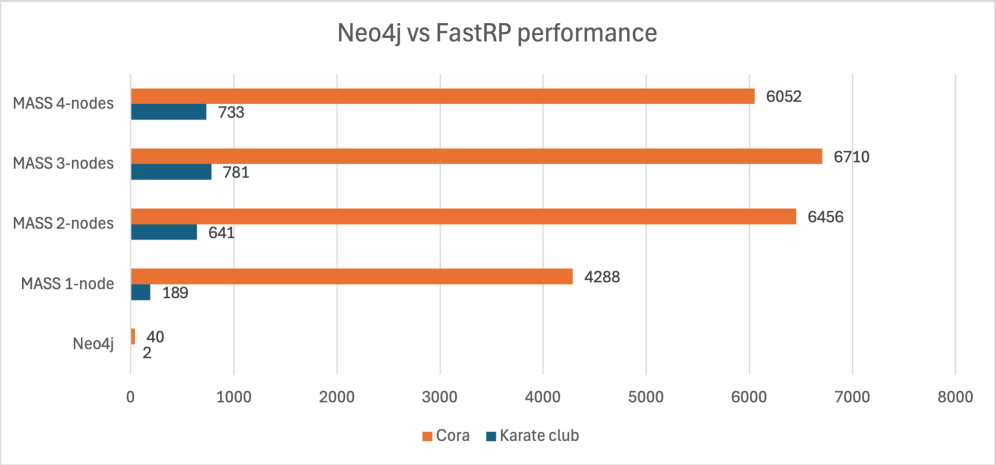

Figure 13: FastRP performance benchmarks

In contrast to the link prediction results, Neo4j significantly outperforms MASS GraphDB in FastRP graph embedding. This performance disparity can be attributed to several factors:

1. **Memory locality advantage:** In Neo4j, all neighbor information is available directly in the node object, making it easily accessible when the graph is in memory. This is particularly beneficial for embedding algorithms that require frequent access to neighborhood information.
2. **Agent-based implementation overhead:** MASS relies on agents to move across the graph to neighbors and gather information, which introduces considerable communication and synchronization overhead. This agent migration is particularly expensive when the computational intensity of vector operations (as in FastRP) is relatively low compared to the communication costs.
3. **Distributed computing overhead:** While distribution across multiple nodes provides scalability advantages, it introduces additional communication costs. The results show that increasing the number of nodes from 1 to 4 does not improve performance for these relatively small datasets, and in some cases degrades it due to increased communication overhead.

6 Conclusion and Next Steps

The benchmarking results reveal both strengths and areas for improvement in the MASS framework's implementation of link prediction algorithms. While MASS achieves on-par accuracy with Neo4j and significantly outperforms it in single-node performance for most topological link prediction algorithms (by factors of 1.2x to 4.5x depending on the dataset), its multi-node performance for embedding-based approaches like FastRP is considerably worse. This discrepancy highlights the algorithm-dependent nature of distributed graph processing performance.

For topological link prediction algorithms, MASS's in-memory graph representation provides substantial speed advantages over Neo4j's disk-based approach, particularly as graph sizes increase. However, for FastRP graph embedding, the current agent-based implementation introduces significant communication and synchronization overhead that outweighs computational benefits. Neo4j's superior performance in embedding tasks (40ms vs. 4,288ms on Cora) stems from its optimized memory locality, where neighborhood information is readily accessible without requiring explicit communication between nodes.

The network latency introduced by the distributed nature of the graph remains a primary challenge, as retrieving nodes from remote machines incurs additional overhead. Neo4j's single-machine setup avoids these issues entirely. Moreover, current benchmarking has been conducted on relatively small datasets, limiting the ability to evaluate scalability and robustness under high-scale conditions. The lack of access to TigerCloud has also prevented benchmarking against TigerGraph, a distributed graph database designed for large-scale analytics. Another limitation is the absence of integrated pipeline support for machine learning operations, which restricts the ability to seamlessly incorporate link prediction results into broader workflows.

To address these challenges, future work will focus on optimizing multi-node performance by reducing inter-node communication, caching frequently accessed data, and improving graph partitioning strategies. For FastRP specifically, implementing a place-based exchange mechanism where neighboring place objects exchange embeddings through message passing without involving agents could significantly reduce the overhead while maintaining MASS's distributed processing advantages. Benchmarking with larger datasets will provide insights into temporal and spatial scalability, while gaining access to TigerCloud will enable direct comparisons with TigerGraph, offering a more comprehensive evaluation of the system's performance.

As a next step, the ongoing implementation of k-Nearest Neighbors (kNN) will complete the FastRP link prediction pipeline, allowing the system to leverage the generated embeddings for similarity-based link prediction. This approach combines the dimensionality reduction benefits of FastRP with the classification power of kNN, potentially offering more accurate predictions than purely topological methods. Additionally, the kNN implementation will provide valuable insights into how MASS handles the combination of embedding generation and similarity computation in a distributed environment, potentially revealing new optimization opportunities.

Benchmarking this enhanced system on large datasets against Neo4j and TigerGraph will test its temporal and spatial scalability, offering valuable insights into its competitiveness in distributed graph analytics. While MASS currently trails in embedding generation performance, its inherent ability to partition large graphs across multiple machines provides a foundation for handling graph data that exceeds single-machine memory capacity — a significant advantage as graph sizes continue to grow in real-world applications.

These efforts aim to evolve the MASS framework into a robust, scalable, and versatile solution capable of addressing real-world challenges in graph-based machine learning and analytics, balancing the trade-offs between computational efficiency and distributed processing capabilities.

7 References

[1] S. Cao, "AN INCREMENTAL ENHANCEMENT OF AGENT-BASED GRAPH DATABASE SYSTEM".

[2] Y. Ma, "An Implementation of Multi-User Distributed Shared Graph".

[3] D. Liben-Nowell and J. Kleinberg, "The Link Prediction Problem for Social Networks".

[4] I. Ahmad, M. U. Akhtar, S. Noor, and A. Shahnaz, "Missing Link Prediction using Common Neighbor and Centrality based Parameterized Algorithm," Sci Rep, vol. 10, no. 1, p. 364, Jan. 2020, doi: 10.1038/s41598-019-57304-y.

[5] "Topological link prediction - Neo4j Graph Data Science," Neo4j Graph Data Platform. [Online]. Available: `https://neo4j.com/docs/graph-data-science/2.13/algorithms/linkprediction/`

[6] "Fast Random Projection - Neo4j Graph Data Science." Accessed: Dec. 10, 2024. [Online]. Available: `https://neo4j.com/docs/graph-data-science/current/machine-learning/node-embeddings/fastrp/`

[7] "Topological Link Prediction Algorithms - Graph Data Science Library," TigerGraph Documentation. [Online]. Available: `https://docs.tigergraph.com/graph-ml/3.10/link-prediction/`

[8] "Fast Random Projection - Graph Data Science Library," TigerGraph Documentation. [Online]. Available: https://docs.tigergraph.com/graph-ml/3.10/node-embeddings/fast-random-projection

[9] Chen, H., Sultan, S. F., Tian, Y., Chen, M., & Skiena, S. (2019). Fast and Accurate Network Embeddings via Very Sparse Random Projection.

[10] C. J. Sullivan, "Behind the scenes on the Fast Random Projection algorithm for generating graph embeddings," Medium. Accessed: May 09, 2024. [Online]. Available: https://towardsdatascience.com/behind-the-scenes-on-the-fast-random-projection-algorithm-for-generating-graph-embeddings-efb1db089

[11] T. Bratanic, "A Deep Dive into Neo4j Link Prediction Pipeline and FastRP Embedding Algorithm," Medium. Accessed: May 09, 2024. [Online]. Available: https://towardsdatascience.com/a-deep-dive-into-neo4j-link-prediction-pipeline-and-fastrp-embedding-algorithm-bf244aeed50d

[12] D. Achlioptas, "Database-friendly random projections: Johnson-Lindenstrauss with binary coins," Journal of Computer and System Sciences, vol. 66, no. 4, pp. 671–687, Jun. 2003, doi: 10.1016/S0022-0000(03)00025-4.

[13] V. Mohan, A. Potturi, and M. Fukuda, "Automated Agent Migration over Distributed Data Structures:," in Proceedings of the 15th International Conference on Agents and Artificial Intelligence, Lisbon, Portugal: SCITEPRESS - Science and Technology Publications, 2023, pp. 363–371. doi: 10.5220/0011784500003393.

[14] L. Adamic and E. Adar, "How to search a social network," Social Networks, vol. 27, no. 3, pp. 187–203, Jul. 2005, doi: 10.1016/j.socnet.2005.01.007.

[15] T. Zhou, L. Lu, and Y.-C. Zhang, "Predicting Missing Links via Local Information," Eur. Phys. J. B, vol. 71, no. 4, pp. 623–630, Oct. 2009, doi: 10.1140/EPJB/E2009-00335-8.

[16] "Barabási–Albert model," Wikipedia. Dec. 02, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Barab%C3%A1si%E2%80%93Albert_model&oldid=1260756305

[17] "Datasets - Neo4j Graph Data Science Client," Neo4j Graph Data Platform. Accessed: Mar. 22, 2025. [Online]. Available: https://neo4j.com/docs/graph-data-science-client/1.14/common-datasets/

8 Appendix

8.1 Topological Link Prediction benchmarks

# of node pairs	MASS GraphDB (ms)	Neo4j (ms)	Gain in performance
100	118	228	1.93
1000	181	669	3.7
5000	564	1127	2
10000	985	1183	1.2

Table 1: Cora topological link prediction benchmarks

# of node pairs	MASS GraphDB (ms)	Neo4j (ms)	Gain in performance
100	40	162	4.05
1000	129	576	4.47
5000	329	878	2.67
10000	444	1189	2.68

Table 2: IMDB topological link prediction benchmarks

# of node pairs	MASS GraphDB (ms)	Neo4j (ms)	Gain in performance
100	8724	21461	2.46
1000	149725	210776	1.41
5000	361627	910476	2.52
10000	404991	1793143	4.43

Table 3: OGBL-DDI topological link prediction benchmarks

8.2 FastRP benchamarks

Platform	Karate club(ms)	Cora (ms)
Neo4j	2	40
MASS 1-node	189	4288
MASS 2-nodes	641	6456
MASS 3-nodes	781	6710
MASS 4-nodes	733	6052

Table 4: OGBL-DDI topological link prediction benchmarks