Link Prediction in Agent-based Graph Database System

Sumit Hotchandani

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2025

Committee:

Munehiro Fukuda

Min Chen

Wooyoung Kim

University of Washington

Abstract

Link Prediction in Agent-based Graph Database System

Sumit Hotchandani

Chair of the Supervisory Committee:
Munehiro Fukuda

This work presents a scalable and interpretable link prediction framework embedded natively within the Multi-Agent Spatial Simulation (MASS) library. By extending MASS's distributed graph infrastructure and property-aware computation model, we implement both classical topological heuristics and embedding-based approaches—most notably Fast Random Projection (FastRP) combined with k-Nearest Neighbors (kNN)—to infer potential connections in graph-structured data.

Topological algorithms such as Adamic-Adar and Resource Allocation, implemented as distributed primitives, demonstrate parity with Neo4j in accuracy and outperform it in execution time on large-scale query workloads. FastRP embeddings are generated via an agent-driven propagation pipeline that mirrors adjacency-based diffusion, enabling full-graph vector generation in distributed environments. Though the current FastRP + kNN pipeline in MASS exhibits higher latency due to agent overhead and synchronization, it achieves competitive recall, especially at higher K values, validating its utility for applications that prioritize coverage over ranking precision.

Experimental results on the Cora citation network show that MASS supports interactive and batch link prediction tasks at scale, offering a memory-local alternative to centralized systems like Neo4j. This project transforms MASS from a simulation-only platform into a programmable, graph-native AI engine—capable of powering graph reasoning tasks for knowledge graphs, recommendations, and retrieval-augmented generation.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Table Number Page

LISTINGS

Chapter 1

INTRODUCTION

1.1 Problem Statement

In graph-structured data, the task of link prediction is to infer likely or missing connections between nodes based on structural properties or learned representations. This task underpins critical applications ranging from recommendation and fraud detection to knowledge graph completion and retrieval-augmented generation (RAG). Formally, given a graph $G = (V, E)$, link prediction seeks to identify pairs $(u, v) \notin E$ that are likely to form edges in future iterations of the graph.

While this problem has been studied extensively in both topological and learning-based paradigms, most scalable implementations rely on centralized systems that either precompute similarity metrics or apply node embedding techniques followed by external similarity search. Systems like Neo4j's Graph Data Science (GDS) library support both heuristics and embeddings such as FastRP but are confined to single-node execution, limiting their applicability to truly large-scale or distributed deployments [7].

At the same time, existing agent-based simulation platforms like MASS (Multi-Agent Spatial Simulation) provide the infrastructure for scalable, parallel computation across distributed spatial datasets but lack support for intelligent inference such as link prediction. Integrating link prediction into a distributed agent-based graph database requires bridging a gap between traditional graph simulation models and modern data science workloads—a gap this project seeks to address.

1.2 Background and Motivation

Recent enhancements to the MASS framework have significantly extended its capabilities for distributed graph analytics. Cao et al. introduced a property graph model atop MASS, enabling expressive graph data with vertex and edge attributes to be queried efficiently

[11]. Ma's work added a Distributed Shared Graph (DSG) abstraction that enables multi-agent, concurrent access to vertex-level graph state in a spatially distributed system [6]. Together, these extensions create the foundation for an agent-based graph database that is both semantically expressive and computationally scalable.

Despite these improvements, MASS has not yet incorporated graph inference primitives. Link prediction is a particularly important example—where unconnected nodes are scored based on topological closeness or learned embedding similarity. This project fills that gap by embedding two complementary approaches directly into the MASS Core runtime: topological scoring and vector-based embedding inference.

The topological path integrates heuristics like Common Neighbors, Adamic-Adar [1], Resource Allocation [17], and Preferential Attachment [3]—all implemented as distributed algorithms over MASS's locality-aware memory model. By embedding them as native primitives, MASS now supports fast, interpretable link prediction using only graph topology.

The second path is embedding-based and centers on Fast Random Projection (FastRP) [7], a lightweight method that projects high-dimensional adjacency relations into low-dimensional space. This project reimplements FastRP using a fully agent-driven pipeline for distributed embedding propagation. Once embeddings are generated, a centralized k-Nearest Neighbor (kNN) inference engine computes pairwise similarities (e.g., cosine or dot product) to identify link candidates.

Together, these two pipelines form a unified, hybrid approach to link prediction that is native to MASS. This is especially valuable for downstream applications like GraphRAG, where long-context reasoning is achieved by fusing symbolic graph traversal with dense embedding retrieval [8]. Scalable inference is a prerequisite for these systems, and the current work provides precisely that by embedding both embedding generation and inference natively within MASS.

By integrating predictive analytics with distributed graph simulation, this project transforms MASS from a simulation engine into a scalable, end-to-end graph intelligence platform.

1.3 Objectives

This whitepaper sets out to extend the MASS framework with native link prediction capabilities through the following objectives:

First, to implement both topological and embedding-based methods—such as Adamic-Adar, Resource Allocation, Preferential Attachement and FastRP + kNN—directly within MASS Core, preserving interpretability and scalability.

Second, to match or exceed Neo4j's performance on standard link prediction metrics including Precision@k, Recall@k MAP, and MRR, while reducing overhead through distributed execution.

Third, to ensure spatial scalability through distributed memory and parallel computation, allowing MASS to support graph workloads beyond the memory limits of single-node systems.

Finally, to support intelligent link recommendations for downstream AI systems, particularly for use in GraphRAG and other neural-symbolic pipelines, positioning MASS as a building block for graph-centric AI.

Chapter 2

RELATED WORK

This chapter reviews the foundational work in topological and embedding-based link prediction, and Neo4j's Graph Data Science pipeline. By examining existing techniques and their limitations, we highlight the motivations behind our distributed, property-aware implementation in MASS.

2.1 Topological Link Prediction

Topological link prediction methods estimate the likelihood of edge formation using only the structure of the graph. These heuristics remain effective in domains where attribute data is unavailable or costly to process.

Common Neighbors (CN) is the foundational method, scoring node pairs by the size of their shared neighborhood. Introduced by Liben-Nowell and Kleinberg[10], it captures the intuitive idea that nodes with more mutual contacts are more likely to connect.

Adamic-Adar (AA) improves on CN by discounting high-degree nodes, applying a logarithmic penalty to each common neighbor's degree [1]. This helps filter out noisy hubs and has shown stronger performance in sparse social graphs.

Resource Allocation (RA) offers a simpler variant, replacing the log-weight with a direct inverse of neighbor degree [17]. Originally proposed for biological and technological networks, RA emphasizes exclusivity over connectivity.

Preferential Attachment (PA), derived from the Barabási-Albert model[3], predicts links based on degree-product. It models the "rich-get-richer" phenomenon, where highly connected nodes attract more edges.

Recent evaluations, such as those by Ahmad et al. [2], demonstrate that these heuristics remain competitive even against more complex models, especially in sparse or dynamically evolving graphs.

2.2 FastRP

The Fast Random Projection (FastRP) algorithm was introduced by Chen et al. as a scalable alternative to traditional sampling-based graph embedding methods such as DeepWalk [15] and Node2Vec [14]. While those models rely on stochastic optimization and corpus-based training over random walks, FastRP forgoes sampling entirely in favor of sparse matrix projection—a change that dramatically reduces computational overhead without compromising embedding quality [7].

At its core, FastRP captures multi-hop topological relationships by repeatedly multiplying the normalized adjacency matrix of the input graph, encoding increasingly distant structural proximity. This process is followed by the application of sparse binary random projections that map the high-dimensional features into a low-dimensional vector space. The underlying theory is grounded in the Johnson–Lindenstrauss lemma [9], which ensures that such projections preserve pairwise distances with high probability, thereby maintaining the integrity of the graph's geometry.

What sets FastRP apart is its deterministic, algebraic nature. Instead of relying on stochastic gradient descent or sampling strategies, the algorithm uses a closed-form propagation pattern with tunable hyperparameters that allow developers to emphasize different levels of neighborhood influence. This simplicity allows for significantly faster training, and empirical evaluations have demonstrated 10x–100x speedups on large graphs compared to Node2Vec and DeepWalk—while delivering comparable link prediction performance on benchmark datasets [7].

Moreover, FastRP is well-suited for integration into production-grade graph systems due to its composability and efficiency. It has been adopted by Neo4j as a native embedding algorithm in the Graph Data Science (GDS) library [12], where its speed and compatibility with downstream tasks like kNN-based link prediction make it a practical choice for real-world deployments.

2.3 Neo4j

Neo4j is a widely adopted graph database system that integrates advanced analytics through its Graph Data Science (GDS) library. GDS provides robust support for link prediction using both heuristic and learning-based approaches. Topological measures such as Common Neighbors, Adamic-Adar, Preferential Attachment, and Resource Allocation are implemented as unsupervised similarity algorithms, while supervised pipelines allow for classifier-based predictions using extracted features from candidate node pairs [13].

For embedding-based workflows, Neo4j implements Fast Random Projection (FastRP), a highly efficient node embedding algorithm that leverages sparse random projections over powers of the normalized adjacency matrix. This technique preserves structural proximities while generating low-dimensional, memory-efficient vectors. Parameters such as embedding dimension, number of hops, and weight decay allow fine-grained tuning of locality versus globality in representation [13].

Once embeddings are computed, Neo4j supports link prediction through vector similarity search using metrics like cosine similarity or dot product. The resulting candidate links can be directly consumed by downstream tasks such as recommendation, graph completion, or node classification. Together, the FastRP and link prediction modules offer a tightly integrated pipeline for scalable graph intelligence within the Neo4j ecosystem.

2.4 Current Challenges

While the FastRP and link prediction pipelines introduced by Neo4j and earlier academic literature represent significant advances in graph analytics, several limitations remain unaddressed in current implementations.

The original FastRP algorithm, as introduced by Chen et al. [7], focuses solely on structural proximity and does not account for node properties during embedding generation. Although Neo4j's Graph Data Science library addresses this, it operates entirely within a single-node architecture. Despite its rich set of algorithms and optimizations, scalability is inherently bounded by the memory and compute resources of a single machine. This becomes a bottleneck for applications requiring distributed processing, particularly on graphs

that span millions of nodes and edges.

Finally, agent-based graph frameworks like MASS, though well-suited for distributed graph simulation and traversal, have historically lacked native support for inference-based tasks such as link prediction. Bridging the gap between symbolic simulation and predictive analytics remains an open challenge in this space.

These gaps motivate the development of a distributed, inference-capable link prediction pipeline that integrates both structural and property-aware embeddings—while maintaining compatibility with parallel agent-based computation models.

Chapter 3

PREVIOUS ACHIEVEMENTS

This chapter outlines the foundational work upon which this project builds. It reviews three key enhancements within the MASS framework: the transformation of the graph database to support property graphs and Cypher queries; the implementation of a distributed shared graph (DSG) system for multi-user concurrent access; and the development of smart agent migration patterns that enable scalable and efficient distributed graph traversal. Together, these components provide the structural, architectural, and computational groundwork necessary for implementing link prediction and embedding-based analytics.

3.1 GraphDB

The groundwork for this project was established through prior work by Shenyan Cao, who significantly enhanced the existing agent-based graph database built on the Multi-Agent Spatial Simulation (MASS) Java library[11]. The primary aim of this enhancement was to evolve the system's capabilities to support complex, real-world datasets by aligning it with the **property graph model** and incorporating the **Cypher query language** for expressive data interaction.

One of the key improvements was the introduction of the `PropertyVertexPlace` class, which enabled nodes and edges in the graph to store rich attribute information in the form of key-value pairs. This structural upgrade allowed the system to represent heterogeneous data more accurately, accommodating entities and relationships with multiple properties. Figure 3.1 illustrates this redesigned agent-based graph database, which forms the backbone of more advanced analytical tasks.

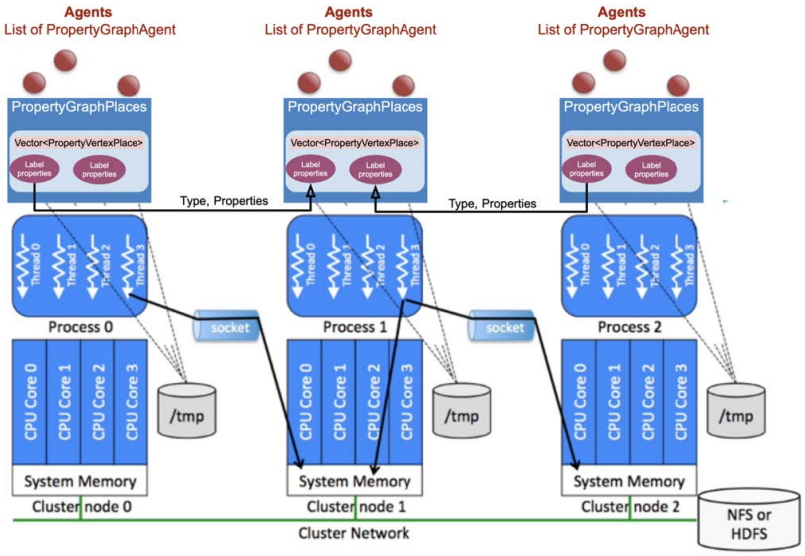

Figure 3.1: Enhanced agent-based graph DB system with PropertyGraphPlaces.[11]

Complementing this change, the system also integrated **Cypher**, a declarative graph query language originally developed for Neo4j. The inclusion of Cypher enables users to perform sophisticated graph operations such as pattern matching (MATCH), insertion (CREATE) on the graph. These features are especially important in research contexts and enterprise applications where complex queries are the norm. The integration of Cypher elevates the querying capabilities of the system from basic key-based lookups to semantically rich graph traversals and manipulations.

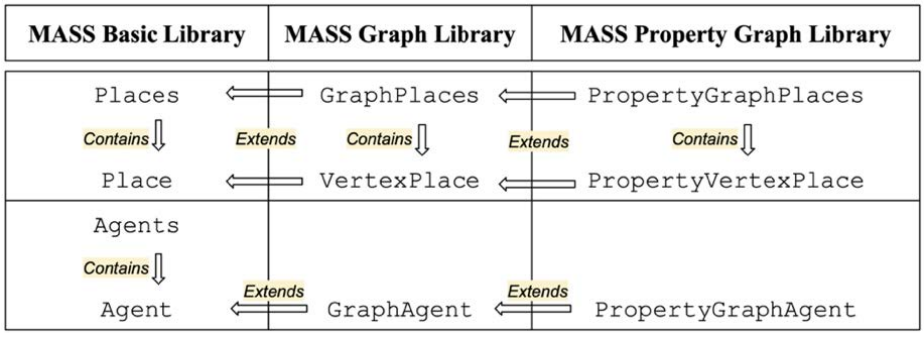

Figure 3.2: Enhanced property graph DB system leveraging MASS Java library.[11]

To enable distributed traversal and parallel query execution, `PropertyGraphAgent`, was introduced. This agent is designed to perform graph traversal and data collection across the distributed environment of MASS while minimizing inter-node communication. As shown in Figure 3.2, this enhancement tightly integrates the graph querying and traversal layers with the MASS framework's agent-based computation model.

Together, these upgrades transformed the graph database from a simple agent-based system into a property-aware, query-capable platform. More importantly, they provided the essential infrastructure for this project's contribution: developing an intelligent link prediction layer on top of a scalable and semantically rich graph database.

3.2 Distributed Shared Graph

Yuan Ma's work addresses the lack of multi-user concurrency in existing ABM libraries by enhancing MASS for distributed graph data access and modification[6].

A key contribution is the `getVertex()` function, which enables access to vertices in a distributed graph stored across nodes in the DSG system. As shown in Figure 3, each node uses a distributed HashMap to store vertices as key-value pairs, enabling direct and efficient access.

The need for concurrent, multi-user access to distributed graph data structures has become increasingly critical in the design of modern graph databases and simulation frameworks. Yuan Ma addressed this limitation in agent-based modeling libraries by proposing and implementing a Distributed Shared Graph (DSG) system within the MASS Java library [6].

At the heart of this system is the `getVertex()` function—a critical operation that enables efficient vertex-level access across a distributed graph stored in memory. In the DSG implementation, each computing node maintains a distributed HashMap, where vertices are stored as key-value pairs. The key represents the vertex identifier, while the value holds the full vertex object, including its adjacency list. This design allows each node to operate independently on its local partition of the graph, with minimal inter-node communication.

As shown in Figure 3.3, the DSG system distributes graph data using a combination of a hashing mechanism and distributed vector structures. This facilitates direct lookups

via getVertex(), which hashes a vertex ID to determine the owning node and then retrieves it from the node's local memory. This approach ensures low-latency, in-memory access—a vital capability for tasks such as neighborhood discovery in link prediction.

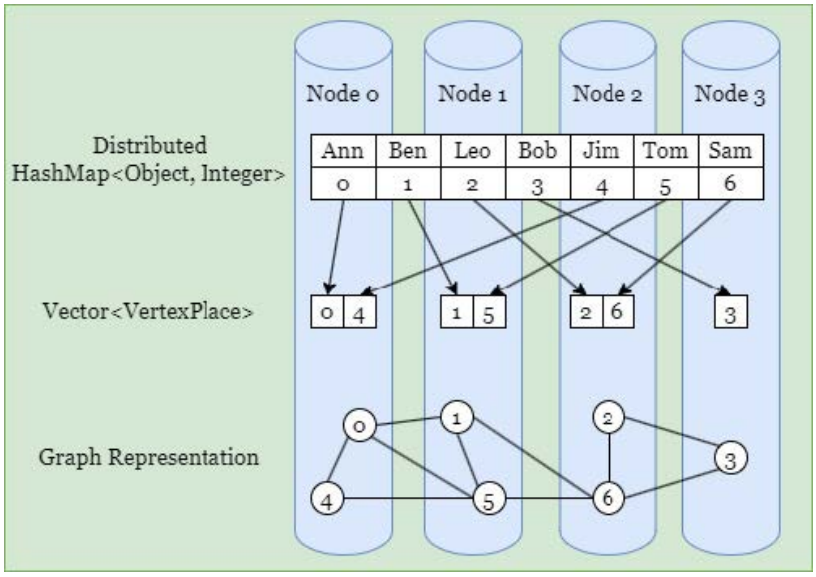

Figure 3.3: Graph representation by distributed map and vector.[6]

Performance benchmarks comparing DSG with Hazelcast, a widely used distributed cache platform, demonstrate DSG's efficiency. Specifically, DSG outperforms Hazelcast in `getVertex()` operations across both single-node and multi-node configurations. This is largely due to its lightweight, in-memory architecture, which avoids the replication and backup overheads present in systems like Hazelcast. Unlike Hazelcast—which ensures high availability through redundant backups—DSG relies on write-back and write-update protocols to maintain consistency while maximizing throughput.

From a functional standpoint, DSG's core emphasis is on enabling scalable and multi-user concurrent access to a mutable distributed graph. The `getVertex()` operation, in conjunction with distributed HashMaps, provides a foundation for topological link prediction algorithms by enabling fast retrieval of a vertex's neighbors, degrees, and adjacency relationships. These capabilities are instrumental for algorithms like Adamic-Adar, Prefer-

ential Attachment, and Resource Allocation, which rely on local graph structure for similarity computation.

3.3 Smart Agent Movement

A pivotal advancement in agent-based computing within the MASS framework was introduced by Mohan et al., who formalized and automated common agent migration patterns for distributed data structure[16]. Their work addressed a key limitation in traditional agent-based modeling (ABM): the manual, repetitive specification of agent migration logic across distributed graphs and spatial grids.

To address this, they introduced two core abstractions, SmartAgent and SmartPlace, which augment MASS's existing Agent and Place classes with spatial awareness and navigational intelligence. These abstractions enable agents to autonomously manage their state and traverse structured data layouts without requiring user-defined migration code for each application scenario.

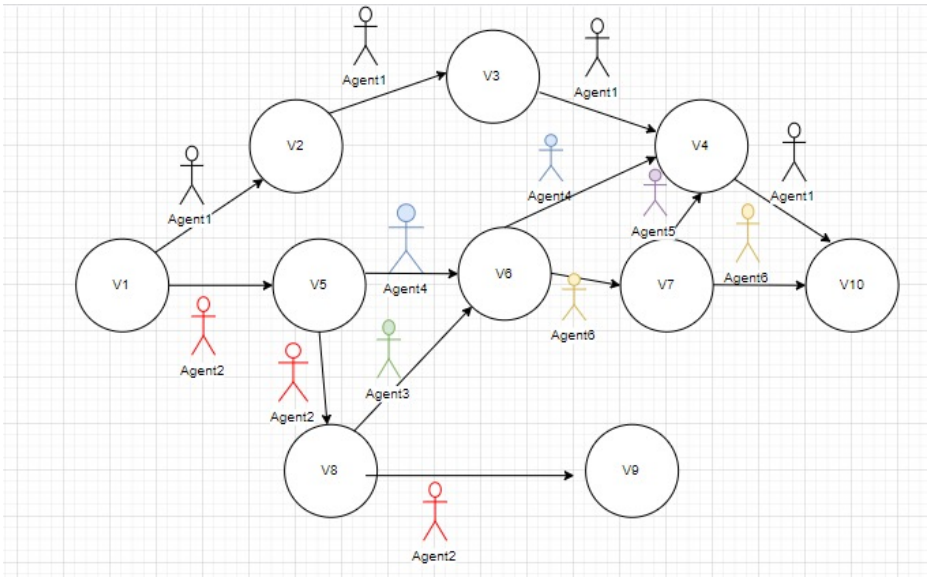

Figure 3.4: Agent propagation over a graph using migratePropagate.[16]

Central to their design is the `migratePropagate()` function, which enables agents to

move through a graph by visiting each neighbor exactly once. This function eliminates redundant migrations by allowing a parent agent to move to one neighbor while spawning child agents to visit the remaining neighbors. As shown in Figure 3.4, this model enables efficient propagation by balancing workload distribution.

In addition to forward propagation, Mohan et al. introduced `migrateOriginalSource()`, a function that allows agents to return to their originating node after completing a traversal. This function supports tasks that require backtracking or result aggregation—such as triangle counting, where an agent must confirm whether it can form a triangle by returning to its initial vertex. This backtracking mechanism is crucial for maintaining computation locality and avoiding unnecessary agent persistence across the graph.

The introduction of automated migration patterns reduced both programming complexity and execution overhead. Empirical evaluations demonstrated up to 83% reduction in lines of code for applications such as BFS and triangle counting, along with significant speedups in agent-based simulations across multiple benchmarks.

In the context of this project, these smart agent migration strategies directly informed the design of FastRP's orchestrator-collector model in MASS. Specifically, the propagation of embedding vectors via orchestrator agents, and the subsequent collection and reduction by collector agents, mirrors the behavior defined in `migratePropagate()` and `migrateOriginalSource()`. By adopting these patterns, the FastRP embedding pipeline in MASS achieves both spatial scalability and synchronization efficiency, essential for distributed embedding computation over large graphs.

Chapter 4
IMPLEMENTATION

This chapter details the architectural and algorithmic design of the link prediction system integrated within MASS Core. It describes how topological heuristics and embedding-based methods were implemented using MASS's distributed, agent-based runtime for scalable and interpretable inference.

4.1 Design Principles

The link prediction framework in MASS is guided by modular, extensible design choices that align with object-oriented best practices. This section outlines how principles such as abstraction, separation of concerns, and scalability shape the system's architecture and integration within the MASS Core runtime.

4.1.1 Extensible design and deep integration in MASS Core

The primary design goal of this system is to embed link prediction directly within the MASS Core, elevating it to a first-class capability rather than a standalone application. This approach ensures that predictive algorithms operate at the same abstraction level as core MASS operations like traversal, simulation, and transformation—enabling seamless integration across pipelines.

As illustrated in Figure 4.1, link prediction components such as `Adamic-Adar`, `Resource Allocation`, and `FastRP` are built atop the MASS property graph infrastructure. These modules interact directly with `PropertyGraphPlaces`, `PropertyVertexPlace`, and also `PropertyGraphAgent`, leveraging in-memory neighbor access, agent coordination, and distributed state updates across nodes.

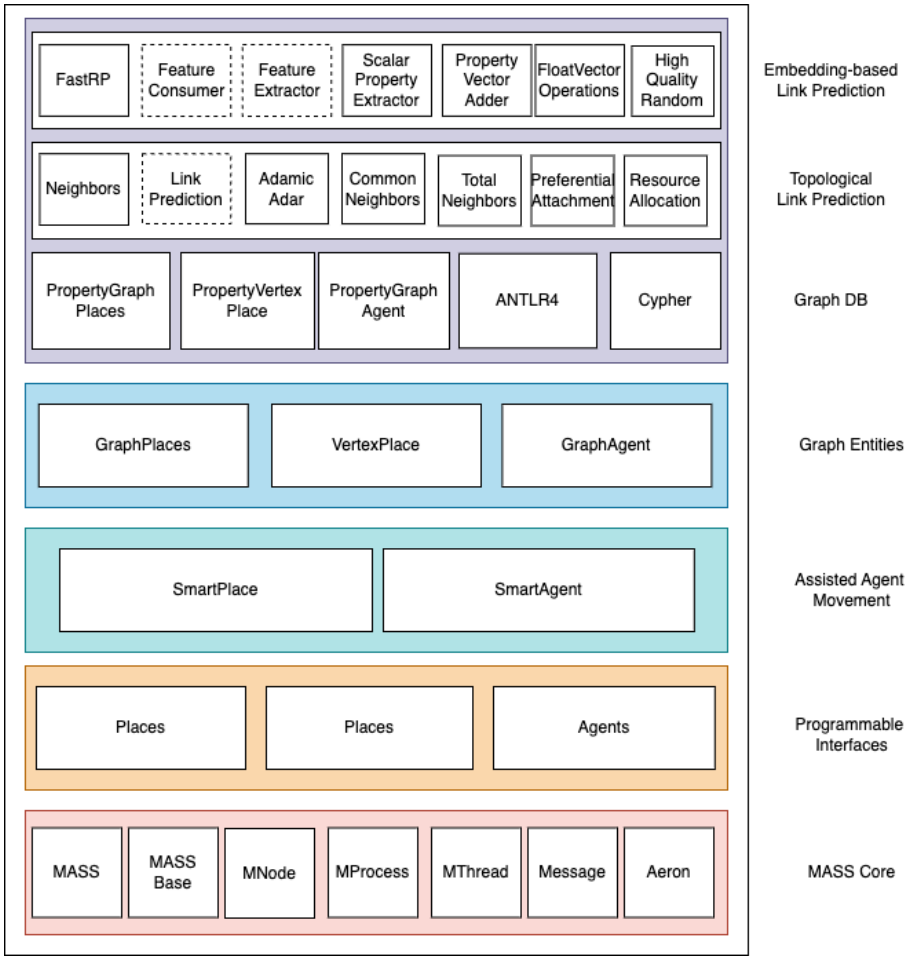

Figure 4.1: The MASS Stack

This tightly coupled architecture enables efficient reuse of embeddings and similarity scores across workflows. The feature extraction layer—comprising `FeatureExtractor`, `ScalarPropertyExtractor`, and `FeatureConsumer`—adds support for semantic enrichment by encoding node attributes alongside structural signals.

Scalability is achieved through MASS's mobile agent model. During FastRP execution, agents propagate vectors, aggregate neighbor embeddings, and compute updates with minimal synchronization. This distributed orchestration allows MASS to scale link prediction across multi-node clusters while preserving performance and memory locality.

4.1.2 Adherence to SOLID principles

The architecture of MASS Core's link prediction and embedding modules adheres to the SOLID principles of object-oriented design, ensuring modularity, extensibility, and robustness across distributed components. Figures 4.2 and 4.3 illustrate the structural layout of the FastRP embedding pipeline and the topological prediction hierarchy, respectively.

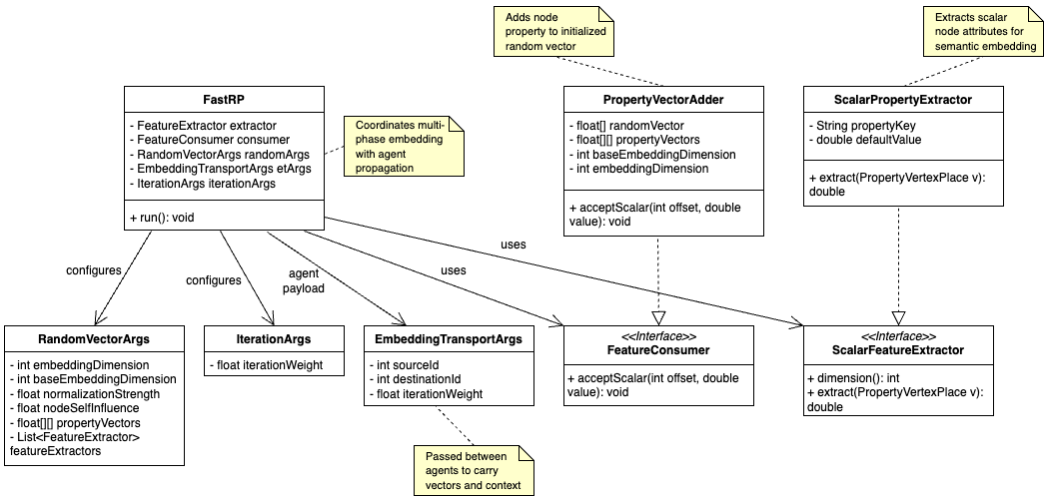

Figure 4.2: FastRP class structure supporting semantic and structural feature integration

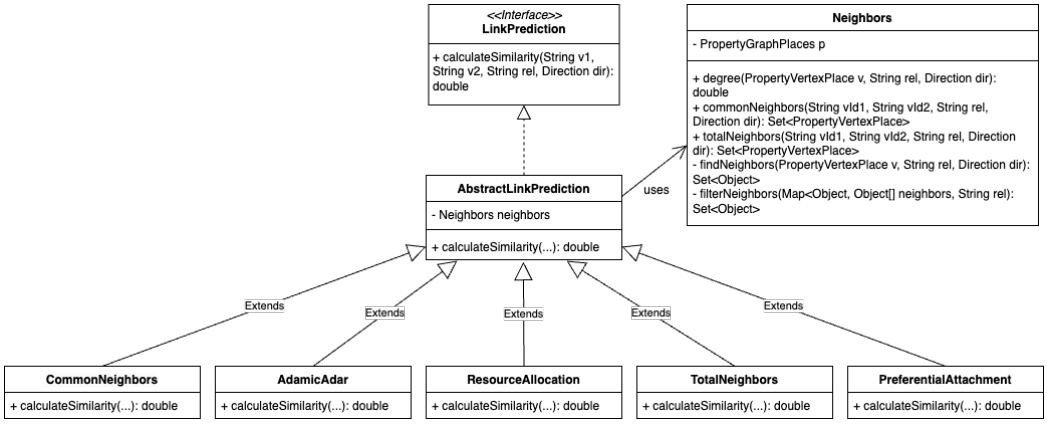

Figure 4.3: Topological link prediction class hierarchy extending shared interfaces

Single Responsibility Principle (SRP): Each class in the system serves a clearly defined role. For instance, Neighbors handles neighbor discovery, FastRP coordinates multi-phase embedding, and the trio of `FeatureExtractor`, `ScalarPropertyExtractor`, and `FeatureConsumer` manages semantic feature transformation. This separation simplifies testing, debugging, and targeted refactoring.

Open/Closed Principle (OCP): The design is open for extension but closed for modification. New prediction algorithms like Jaccard or Katz can be introduced by subclassing `AbstractLinkPrediction`, while new feature types can be added to the embedding pipeline via custom `FeatureExtractor` implementations—without altering existing functionality.

Liskov Substitution Principle (LSP): All link prediction algorithms extend a shared interface (`LinkPrediction`) and respect the same method contract (`calculateSimilarity`). This guarantees that any implementation—Adamic-Adar, Resource Allocation, or Preferential Attachment—can be used interchangeably in evaluation pipelines without breaking behavior.

Interface Segregation Principle (ISP): Interfaces are narrow and role-specific. `LinkPrediction` focuses solely on computing similarity between node pairs, while `FeatureExtractor` isolates logic for property-based feature encoding. This avoids forcing consumers to implement unnecessary functionality and promotes decoupling.

Dependency Inversion Principle (DIP): High-level modules like FastRP depend on abstractions rather than concrete implementations. It accepts user-defined extractors and consumers as configurable components, allowing substitution during testing or specialization—thus promoting flexibility and reusability.

Together, these design patterns ensure that the MASS link prediction framework remains flexible, maintainable, and capable of scaling alongside future graph analytics needs.

4.2 Topological Link Prediction

4.2.1 Distributed Execution Model

Figure 4.3 sketches the flow of a single similarity query inside the MASS Core runtime.

A request begins in user code with a call to `calculateSimilarity(v1 ,v2 , relation`

`, direction)`. Both vertex identifiers are first converted to their internal integer keys through the global `HashMap<Object, Integer>` that is shared by every JVM in the cluster. Each key is hashed to a computing node using the simple rule:

$$NodeID = VertexID \mod |Computing\ nodes|$$

so the caller can immediately decide whether the data are local or must be fetched remotely.

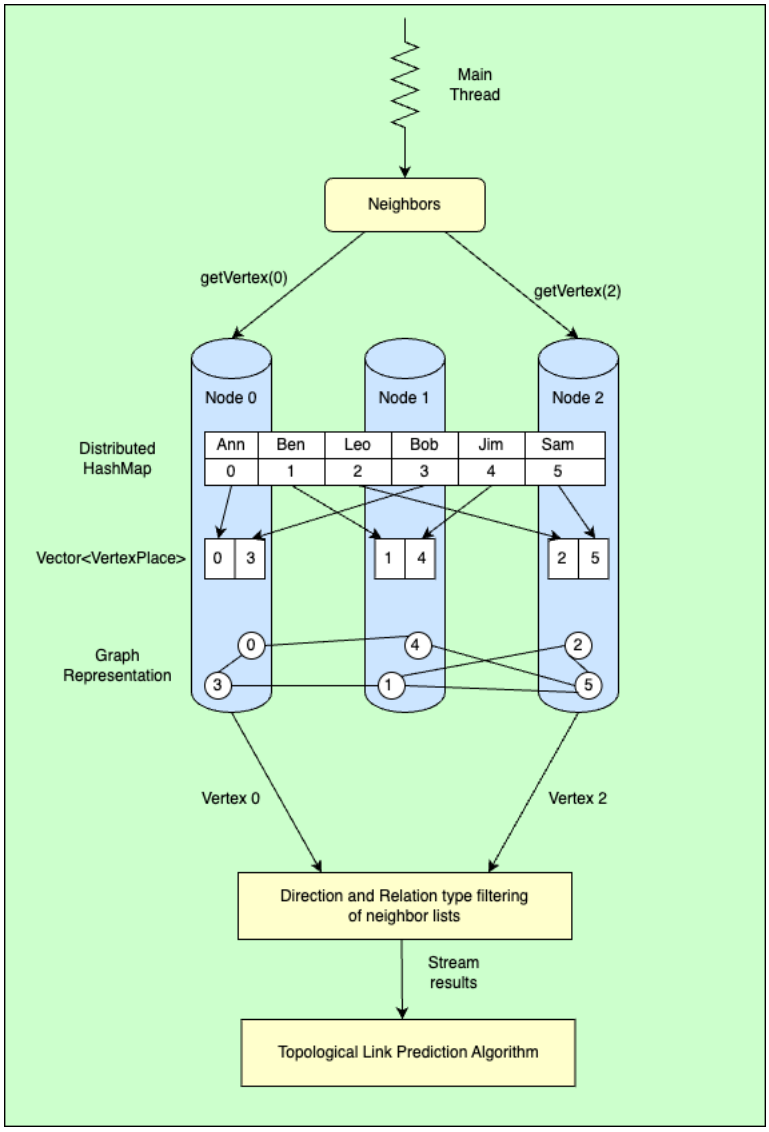

Figure 4.4: Topological link prediction - Distributed execution model

This is where the data-access layer kicks in, if the requested `PropertyVertexPlace` already resides in the local JVM, the reference is returned in constant time. Otherwise `getVertex(vertexID)` issues a one–way RPC to the owning `MProcess`; the remote process serialises the vertex, sends it back across the socket created at start-up, and the caller blocks only for this round-trip latency. Subsequent computations on the vertex, including all neighbor-set manipulation, are therefore executed entirely in local memory, so at most one network hop is incurred per vertex and per query.

Once both vertices are resident, the Neighborhood layer comes into play. `findNeighbors( VertexPlace v, String rel, Direction dir)` streams the raw adjacency lists that are stored inside every `PropertyVertexPlace` as primitive int[]. These lists are pruned in-situ by the helper method `filterNeighbors`, which discards edges that do not match the user-supplied relationship label and honors the directionality flag (`TO`, `FROM`, `BOTH`).

Because all filtering is performed after the data have been copied into the JVM, no further communication is necessary; the cost of this stage is strictly linear in the degrees D_{v1} and D_{v2}.

The Algorithm layer is built on the `LinkPrediction` interface and its abstract base class `AbstractLinkPrediction`. Each concrete scorer — `CommonNeighbors`, `AdamicAdar`, `ResourceAllocation`, `PreferentialAttachment`, `TotalNeighbors` — overrides a single method, `calculateSimilarity`, and relies on the Neighbors instance for all set operations. Because the reduction phase is executed with Java 8 streams, a single `parallelStream()` call activates the fork–join pool so that large neighborhoods are processed in parallel on every core of the worker JVM.

Putting the three layers together, the end-to-end complexity of every topological measure is $T = O(D_{v1} + D_{v2})$, since the only operations beyond neighbor traversal are constant-time arithmetic and set cardinality calculations. All expensive work happens after the vertices have been shipped to the caller, which keeps network traffic to an unavoidable minimum and allows MASS to exploit local, cache-friendly data structures. In practical workloads the same vertex is often reused across thousands of queries; after the first fetch it remains pinned in the JVM's heap, so later similarity computations avoid even the initial RPC.

The design therefore reconciles algorithmic simplicity with cluster scalability. Researchers

who wish to add new link-prediction formulas need only implement a single-method class; MASS transparently handles data placement, remote access, neighbor filtering, and parallel reduction, enabling high-throughput analysis on graphs that would exceed the memory of a single machine.

4.2.2 Algorithms

This section details the topological link prediction algorithms integrated within the MASS Core library. Each method builds on the distributed architecture described previously, where vertex-level neighborhood information is retrieved using `getVertex()`from either local or remote memory, followed by application of `filterNeighbors()` inside the `Neighbors` helper to respect the user–supplied `Direction` (`TO`, `FROM`, `BOTH`) and relation type. All methods operate with a time complexity of $O(D_{v1} + D_{v2})$ where D_{v1} and D_{v2} are the degrees of the input vertex pair.

4.2.2.1 Common Neighbor based methods

These algorithms quantify the similarity between two vertices based on the structure of their shared neighborhood.

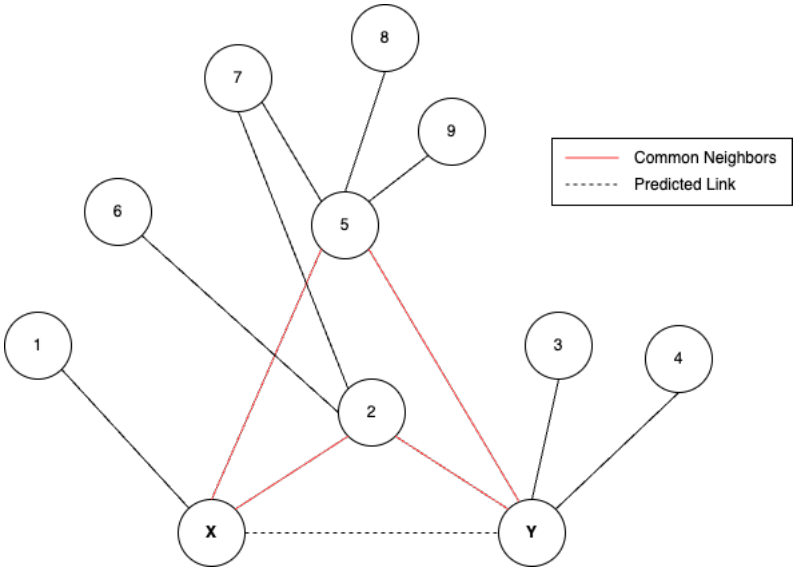

Figure 4.5: Common neighbors example

Figure 4.4 shows the running example that we will use to illustrate the scores. Vertices X and Y share two neighbors $\{2, 5\}$; their degrees are $|N(2)| = 3$ and $|N(5)| = 3$.

1. **Common Neighbors (CN):** Computes the count of shared neighbors between two nodes. A higher number indicates greater likelihood of a link. This method directly uses the `commonNeighbors()` routine from the `Neighbors` class.

   This is the most direct similarity metric. It simply counts the number of shared neighbors:

   $$CN(X, Y) = |N(X) \cap N(Y)|$$

   From Figure 4.4, we find:

   $$CN(X, Y) = |2, 5| = 2$$

   In MASS, this is implemented via the `commonNeighbors()` method in the Neighbors class, which intersects the filtered neighbor lists of both input vertices. The

`CommonNeighbors.java` class then returns the size of this intersection as the final similarity score.

2. **Adamic-Adar Index (AA):** The Adamic-Adar index improves upon CN by giving more weight to rare common neighbors:

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

Using the degrees above:

$$AA(X,Y) = \frac{1}{\log 4} + \frac{1}{\log 5} \approx 3.092$$

In MASS, this is implemented in `AdamicAdar.java`, which iterates over common neighbors and uses `Neighbors.degree(u)` to obtain degree information in a distributed fashion:

This approach uses distributed calls for each common neighbor, but computation is performed locally after retrieval.

The MASS implementation uses the degree of each common neighbor, obtained via distributed look-ups, to compute the log-inverse weight.

Adamic and Adar [1] argue that a common neighbor of high degree conveys little information about a specific pair; weighting each witness by the inverse of its self–information $\frac{1}{\log |N(u)|}$ rewards rare connections and penalises hubs.

3. **Resource Allocation Index (RA):** Resource Allocation is similar to Adamic-Adar but substitutes the log function with a direct inverse degree:

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

From our example:

$$RA(X,Y) = \frac{1}{4} + \frac{1}{5} = 0.45$$

The implementation (`ResourceAllocation.java`) is almost identical to Adamic-Adar, but skips the logarithm for lower computational overhead.

4.2.2.2 Total Neighbor Methods

Total neighbor scores treat each vertex's *entire* connectivity profile as an indicator of future attachment. Unlike the overlap–centric metrics in Common neighbor methods, these measures reward vertices that are already well-connected or that jointly span a large portion of the local graph.

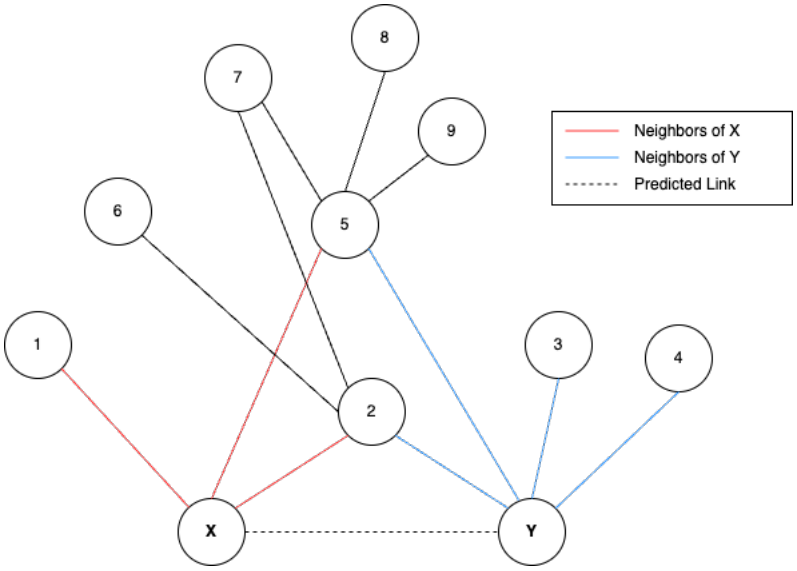

Figure 4.6: Total neighbors example

Figure 4.6 shows the running example that we will use to illustrate the scores. Node X has three neighbors (red) $\{1, 2, 5\}$, while node Y has four neighbors (blue) $\{2, 3, 4, 5\}$ yielding $|N(X)| = 3$, $|N(Y)| = 4$ and $|N(x) \cup N(y)| = 5$.

1. **Preferential Attachment (PA):** The *"rich-get-richer"* principle of Barabási and Albert [3] posits that high-degree vertices attract new links at a super-linear rate.

   MASS realises this idea via the straightforward degree product

   $$PA(x, y) = |N(x)| \times |N(y)|.$$

   In `PreferentialAttachment.java` the degree of each vertex is fetched once, regardless of where the data reside, and multiplied.

For Figure 4.5 we obtain $PA(x, y) = 3 \times 4 = 12$, indicating strong affinity because both endpoints are already well-connected.

2. **Total Neighbors (TN):** TN measures the *breadth* of the two neighborhoods:

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

The implementation in `TotalNeighbors.java` first merges the two neighbor sets streaming from `filterNeighbors()` and then returns the cardinality—no additional distributed synchronization is needed.

In the running example $TN(x, y) = 5$, reflecting that the pair jointly covers five distinct vertices.[1]

4.3 Embedding-based Link Prediction

Topological scores inspect one pair of neighborhoods at a time; embedding methods first encode the entire graph into dense vectors and judge links by geometric closeness. MASS uses Fast Random Projection (FastRP) to embed every vertex, then answers link-queries with a fast k-Nearest-neighbors (kNN) search over those vectors. The pipeline is two-stage: a one-off, parallel FastRP pass generates length-normalised embeddings; at query time, kNN returns the k closest candidates, whose distances serve as link-likelihood scores. The sections that follow explain the distributed FastRP algorithm and the kNN search engine that exploits MASS's locality-aware memory model.

4.3.1 FastRP

Fast Random Projection (FastRP) provides the dense, low-dimensional node embeddings used as the foundation for all downstream link prediction in MASS. Originally proposed as a scalable alternative to DeepWalk and Node2Vec, FastRP achieves structural similarity encoding via iterative matrix-vector operations combined with sparse random projections.

[1] The union can never be smaller than the intersection used by Common Neighbors; hence $TN \geq CN$ for all pairs.

In MASS, FastRP is reengineered as a distributed algorithm that leverages agent-based parallelism, semantic feature integration, and modular parameterization to operate efficiently across multi-node clusters.

4.3.1.1 Enhancements to FastRP

FastRP in MASS inherits two ideas first popularized by Neo4j's Graph Data Science implementation[12] that augment the canonical FastRP kernel and make the embeddings semantically richer and more tunable in distributed large-scale deployments.

A modular feature–extraction pipeline. Before the first propagation step, every vertex first produces a hybrid random vector of dimensionality d, which is passed directly to `FeatureExtraction.extract`, a lightweight, plug-in framework patterned after Java's functional streams. Concrete `FeatureExtractor` instances—such as the custom built `ScalarPropertyExtractor` push chosen property values into a `FeatureConsumer`; by default this consumer is a `PropertyVectorAdder`. The consumer multiplies each incoming scalar by a learned weight and inserts it into the upper $d_p = \lfloor \texttt{propertyRatio} \cdot d \rfloor$ slots of the vector, where d is the user-defined target dimension and `propertyRatio` (0–1) specifies what fraction of the vector is devoted to explicit attributes. The remaining $d_b = d - d_p$ slots preserve the base embedding produced by earlier layers. For example, with $propertyRatio = 0.4$ and $d = 10$, the model assigns $d_p = \lfloor 0.4 \cdot 10 \rfloor = 4$ slots to property values and keeps $d_b = 10 - 4 = 6$ slots for the original representation. When a vertex lacks a requested scalar, `ScalarPropertyExtractor` substitutes a configurable default, ensuring the pipeline remains numerically stable even with incomplete attribute data. Because the entire procedure runs within the JVM that owns the vertex, property look-ups and arithmetic incur no network overhead, and new extractors can be added at runtime without recompiling the embedding core. The resulting initial vectors already blend structural uncertainty from sparse random projection with typed attribute information, so subsequent FastRP iterations propagate a genuinely semantic signal through the graph.

Node-self-influence. After each aggregation step the vertex merges the normalized neighbor vector with its running embedding according to

$$Z_v^{(t)} = \alpha Z_v^{(0)} + (1 - \alpha)(Z_v^{(t-1)} + \beta_t A_v^{(t)})$$

, where $\alpha =$ `nodeSelfInfluence` $\in [0, 1]$, $Z_v^{(0)}$ is the original hybrid vector and β_t is the FastRP iteration weight. Setting $\alpha \approx 1$ preserves vertex identity (useful for link prediction tasks where self-context matters), whereas $\alpha \approx 0$ replicates the behaviour of Neo4j's implementation by allowing the context signal to dominate. The parameter is applied locally after each call to `computeFinalEmbedding`, so it introduces no additional communication overhead yet exposes a direct bias–variance knob to the modeller.

Together these two extensions let MASS produce embeddings that are simultaneously topology-aware, attribute-aware, and controllably specific—functionality not available in the original FastRP design [12].

4.3.1.2 MASS Implementation

In MASS the FastRP pipeline is expressed as an alternating dialogue between *places*—the distributed storage layer that holds graph state—and mobile *agents* that ferry state-updates across the cluster. Each phase of the algorithm is launched by a single `callAll()` on a container object (`PropertyGraphPlaces` or `PropertyGraphAgent`).

The method executes the requested user function on *every* local object in parallel;the subsequent `manageAll()` commits all pending migrations, spawns and kills, and performs an MPI-style barrier so that every JVM reaches the next superstep simultaneously. This bulk-synchronous discipline gives FastRP exactly the same k–hop message pattern that underlies the original formulation, but without the developer having to manage explicit communication.

1. Property–vector initialization The first `callAll()` on `PropertyGraphPlaces` invokes `computePropertyVectors()`. Inside each `PropertyVertexPlace` the feature–extraction pipeline iterates through a compile-time list of `FeatureExtractor` objects—small strategy

classes that know how to read and scale an attribute (e.g. numeric age, categorical label, one-hot tag).

Each extractor forwards the value to a `PropertyVectorAdder`, which writes the scaled number into the tail of a dense `float[]` property vector. Because the call happens locally there is no network traffic; property vectors are kept side-by-side with the structural adjacency list, so subsequent phases can fuse attribute and topology in a single memory pass.

2. Random–vector initialization A second `callAll()` on `PropertyGraphPlaces` triggers `computeRandomVectors()`. Here every vertex draws a reproducible pseudo-random stream (`HighQualityRandom` is reseeded with `randomSeed ^ vertexId`) and fills the *base* part of its embedding with sparse ± 1 entries scaled by $\sqrt{\text{SPARSITY}/d}$. Directly afterwards the property vector created in step 1 is blended in `PropertyVectorAdder.acceptScalar` for dimension $i \geq d_{\text{base}}$

$$z_i += s \cdot p_i$$

where s is the user-defined scale factor. The entire procedure is embarrassingly parallel.

Algorithm 1 Initial Embedding Construction with Property Integration

procedure COMPUTEINITIALEMBEDDING(Vertex v, Parameters θ)
  Extract embedding dimension d, base dimension d_b, and random seed from θ
  Compute $s = \texttt{degree}(v)^{-\gamma}$ where $\gamma = \texttt{normalizationStrength}$
  Set $entryValue \leftarrow s \cdot \sqrt{\texttt{SPARSITY}/d_b}$
  Seed random generator with $hash(v.\texttt{id}) \oplus \texttt{randomSeed}$
  $z^{(0)} \leftarrow$ new float vector of dimension d
  for $i = 0$ to $d_b - 1$ **do**
    $z_i^{(0)} \leftarrow$ random $\pm entryValue$ with probability p
  end for
  for all features f extracted from v **do**
    Inject scaled f into tail of $z^{(0)}$ using associated property vector
  end for
  $norm \leftarrow \|z^{(0)}\|_2$ ▷ normalization
  if $\alpha > 0$ **then**
    $z_v \leftarrow \frac{\alpha}{norm} \cdot z^{(0)}$ ▷ self-influence weighting
  else
    $z_v \leftarrow z^{(0)}$
  end if
  Store z_v as v's embedding
end procedure

3. Multi-hop embedding propagation. Once embeddings are initialized, the FastRP propagation loop begins. This process simulates multi-hop proximity diffusion across the graph, implemented as a sequence of distributed supersteps orchestrated by alternating calls to `callAll()` and `manageAll()` on the `PropertyGraphAgent` container.

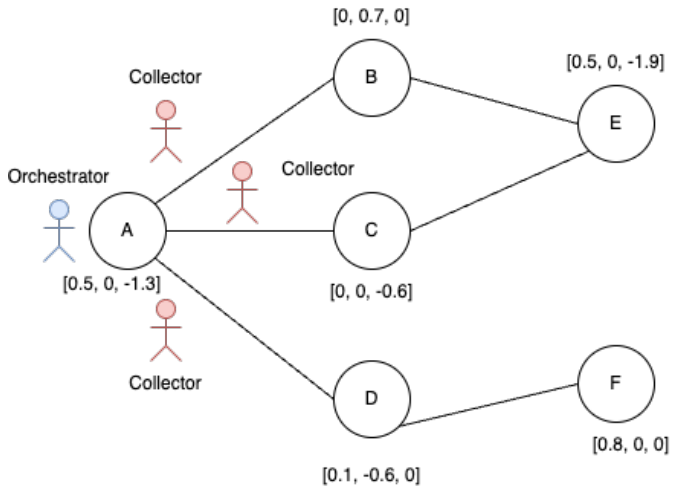

Figure 4.7: Spawn phase

Each superstep corresponds to a single FastRP iteration. It begins with each vertex spawning a single *orchestrator agent*. The orchestrator is initialized with the current embedding vector and the FastRP iteration weight β_t. It identifies the vertex's outgoing and incoming neighbors (refer lines 12-13 in listing B.2)

For each neighbor, the orchestrator constructs a payload (embedding transport arguments) and spawns a dedicated *collector agent* as shown in Figure 4.7. Refer lines 20-38 in listing B.2 for the implementation

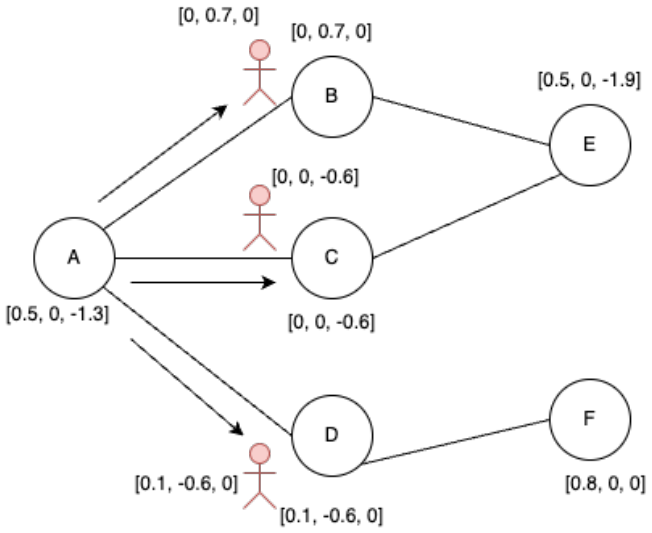

Figure 4.8: Neighbor visit and collect phase

Each *collector agent* migrates to its assigned neighbor and retrieves the most recent embedding stored in the `PropertyVertexPlace`. If the agent arrives at the destination, it executes lines 48-51 in listing B.2

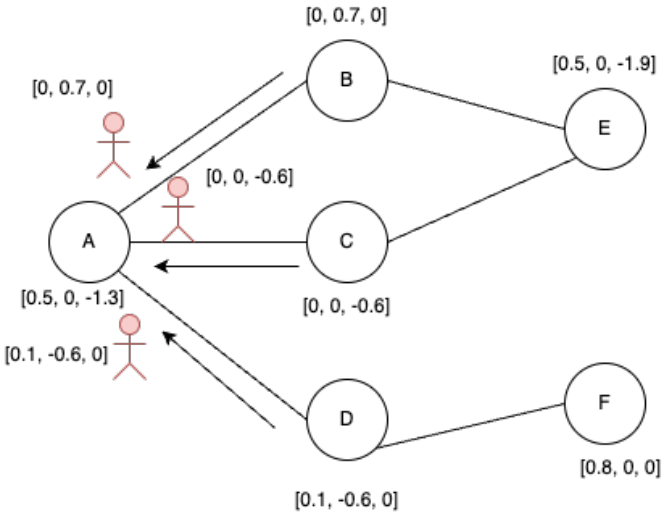

Figure 4.9: Return phase

On return, the collector deposits the embedding at the source vertex, which appends it to an in-memory list. Once all expected embeddings have returned (tracked via a thread-safe atomic counter), the vertex computes its new embedding by averaging, scaling, normalizing, and combining with the existing vector. The computation logic is defined as:

Algorithm 2 Final Embedding Update

procedure UPDATEFINALEMBEDDING
    $d \leftarrow$ degree of the current vertex
    $scale \leftarrow \frac{1}{d}$ if $d > 0$ else 1.0
    $accumulator \leftarrow$ zero vector of dimension d
    for all $z_u \in$ collected neighbor embeddings **do**
        $accumulator \leftarrow accumulator + z_u$
    end for
    $accumulator \leftarrow scale \cdot accumulator$
    $norm \leftarrow \|accumulator\|_2$
    if $norm < \epsilon$ **then**
        $norm \leftarrow 1.0$
    end if
    $accumulator \leftarrow \frac{accumulator}{norm}$
    $z_v \leftarrow z_v + \beta_t \cdot accumulator$ ▷ Update embedding with iteration weight
    Reset temporary neighbor buffers and counters
end procedure

This pattern—spawn, migrate, return—is executed once per iteration. The iteration weight β_t is provided via the `IterationArgs` class, which contains a tunable list of hop-wise decay factors. Typical configurations apply 2–4 such iterations to balance locality and generalization.

The call to `manageAll()` concludes each iteration by committing pending migrations, removing dead agents, and synchronizing all compute nodes, as defined in Listing 4.1

```
while (agents.nAgents() > 0) {
    agents.callAll(1, null);
    agents.manageAll();  // global barrier
}
```

Listing 4.1: Agent life-cycle management

This structure mirrors the formal definition of FastRP as a sequence of matrix-vector multiplications involving adjacency powers, while taking full advantage of MASS's distributed memory model. Each superstep approximates one layer of diffusion across the graph, but instead of building full matrices or relying on batched linear algebra, MASS orchestrates localized computation and movement—preserving memory locality, lowering synchronization costs, and enabling embedding on graphs orders of magnitude larger than what a single node can hold.

4.3.2 KNN

Once every vertex owns a FastRP vector, link prediction becomes a pure similarity–search problem in $\mathbf{R^d}$. The pipeline therefore introduces a k–nearest–neighbor (kNN) stage whose execution is deliberately split in two. First, a single `PropertyGraphPlaces.callAll()` instructs every worker to serialize its slice of the embedding matrix and stream it to the primary JVM; when the broadcast completes the master process holds a fully populated, memory–resident `HashMap<Object,float[]>`. Because this transfer is embarrassingly parallel, the cost of the "collect/map" phase is dominated by network throughput and scales linearly with the number of workers. The vectors remain immutable after this point, so the map is built exactly once and reused by every subsequent query in the second phase.

4.3.2.1 MASS Implementation

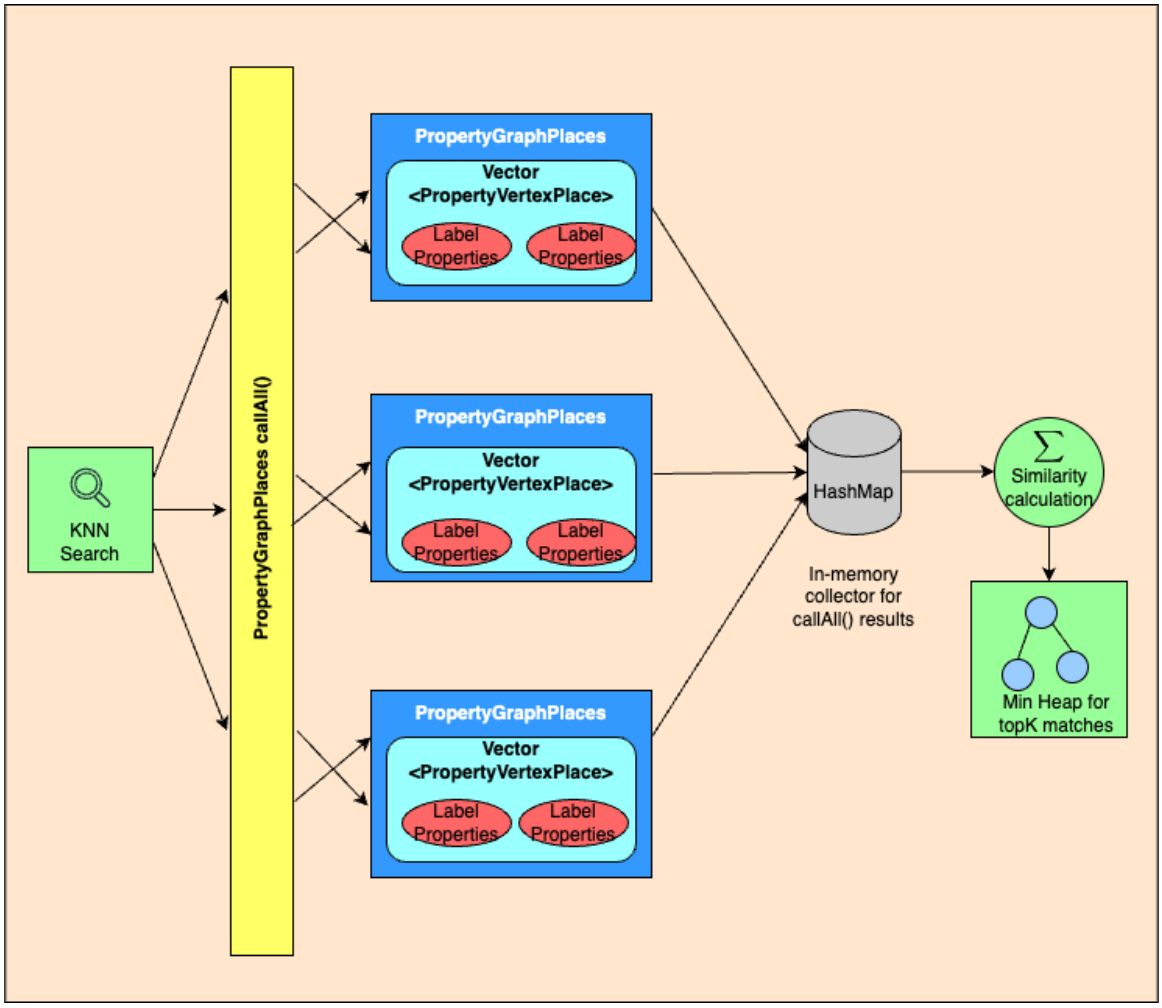

Figure 4.10: MASS KNN workflow

Figure 4.10 sketches the resulting control flow. The query API—`KNN.run(queryId, k)`—is intentionally thin: the client thread supplies a vertex identifier, a neighborhood size k wrapped in a `KNNConfig` object that pins down the similarity metric. The method looks up the query vector in the pre-loaded map and performs a single memory–resident scan over all remaining entries.

The algorithm uses a static helper in `SimilarityMeasures.java` to compute similarity

scores (e.g., cosine or Euclidean) for candidate vectors, then feeds results into a bounded `PriorityQueue<NodeSimilarity>` configured as a min-heap. This heap retains only the top-k candidates by ensuring the smallest score remains at the root. Each insertion or rejection operates in $O(\log k)$ time, resulting in an overall $O(|V| \log k)$ complexity for scanning all nodes, as each vector array is accessed once.

When the traversal finishes, the heap's contents are sorted in descending order to produce the final ranked neighborhood $\langle v_1, \ldots, v_k \rangle$ with scores. This output is directly usable for downstream tasks like threshold-based filtering, semantic validation, or edge classification via supervised models—eliminating post-processing steps. The fixed-capacity heap ensures memory efficiency while maintaining real-time updatability during the scan.

4.3.2.2 Design Considerations

The implemented kNN kernel is intentionally small and cache-friendly. Fewer than twenty lines in `KNN.java` perform an allocation-free dot product (or cosine/Euclidean variant from `SimilarityMeasures`) over the in-memory embedding map that was populated once through `PropertyGraphPlaces.callAll()`. After this one-off transfer, every query is handled completely inside the primary JVM; on a commodity Xeon a single core scans one million 128-dimensional vectors in well under half a second.

A purely centralized heap was favored over a distributed merge after profiling. Shipping tiny partial heaps from each worker erased any gains from parallel similarity arithmetic, and the subsequent merge imposed an extra $O(pklogk)$ barrier that grew with both the number of workers p and the requested neighbor count k. Storing the min-heap locally allows the JVM to optimize the loop for speed and efficiency, eliminating the need for data transfers and synchronization between processes.

During early iterations we experimented with an agent-oriented search that mirrored the FastRP walk pattern. A single query agent was injected at the source vertex; at each hop it spawned children to every neighbor. Each child fetched the neighbor's embedding in local memory, computed its similarity to the query vector (which it carried as passive state), and reported that scalar back to the parent. The parent vertex maintained its own

bounded min-heap of the best scores received so far, deciding—subject to a user-defined depth parameter—whether to continue the expansion. Conceptually this respected MASS's "move computation to the data" philosophy and promised locality-aware pruning. In practice, however, global kNN suffered: every generation required an `Agents.manageAll()` barrier so that parents could merge incoming scores and determine the next wave, leading to three–five full synchronizations per query. Serializing thousands of agents per generation, each carrying the d dimension float query embedding, also proved markedly more expensive than sending the same vectors once in bulk. Consequently the agent design was shelved for whole-graph similarity ranking, though it remains attractive for node-local or very shallow queries that never leave a single NUMA domain—a direction revisited in Section 6 (*Conclusion & Future Work*).

Chapter 5

EVALUATION

This chapter benchmarks the proposed MASS link-prediction pipeline against Neo4j across both sparse (Cora) and dense (OGBL-DDI) graphs. We report system-level execution times and ranking-based accuracy, highlighting where MASS's distributed, in-memory design excels and where Neo4j's centralized optimizations still lead.

5.1 Setup

5.1.1 Datasets

To evaluate the performance and accuracy of our link prediction framework, we conducted a series of experiments on two well-established benchmark datasets: the Cora citation network and the OGBL-DDI biomedical interaction graph[5].

The Cora dataset is a directed citation network introduced by McCallum et al. in their work on automating web portals with machine learning. It has since become a standard benchmark in graph learning research, frequently used for both node classification and link prediction tasks. Each node represents a scientific publication, and edges denote citation relationships. Nodes have a single categorical feature (subject). We used the dataset as a directed graph with the following setup.

The second dataset, OGBL-DDI, is part of the Open Graph Benchmark for large-scale link prediction. It models drug–drug interactions with over a million edges and is significantly denser than Cora. The Neo4j GDS version includes six relationship types representing positive and negative splits for train, validation, and test sets. For this evaluation, only the `TRAIN_POS` relationships were used to build the graph, and predictions were evaluated on the `TEST_POS`[1] edges.

[1] 39,324 out of the 131,078 rows are used for topological prediction due to performance constraints on both MASS and Neo4j, discussed in detail in the Limitations section.

Property	Cora	OGBL-DDI
Nodes	2,708	4,267
Node labels	Paper	Drug
Total Edges	5,429	1,532,370
Edge Types	CITES (directed)	TRAIN_POS, VALID_POS, VALID_NEG, TEST_POS, TEST_NEG
Train Edges	3,800 (70% random samples)	1,059,073 (TRAIN_POS only)
Test Edges	1,629 (30% random samples)	131,078 (TEST_POS only)

Table 5.1: Comparison of dataset statistics and evaluation splits

5.1.2 System Configuration

All experiments were executed under comparable conditions across both platforms:

- **Neo4j:** Experiments were run on a MacBook Pro with an Apple M2 Pro (10-core CPU) and 16 GB RAM, using Neo4j GDS v2.6.9. Both unsupervised (FastRP + kNN) and heuristic-based pipelines were tested through Cypher queries.

- **MASS:** All MASS Core experiments were run on the CSSMPI research cluster, with Java 11 on virtual machines provisioned with Intel Xeon Gold 6130 processors. Memory per VM ranged from 16 GB to 20 GB depending on node allocation. The MASS system was scaled across 1–8 VMs to observe spatial scalability under distributed conditions.

Performance benchmarks were evaluated on the full graph (without pruning or subsetting), measuring total time taken for both embedding and inference phases. For accuracy benchmarks, a standard link prediction setup was followed using the described training-test splits. All results were averaged across three runs to ensure statistical stability.

5.1.3 Evaluation Metrics

Table 5.2 outlines the ranking-based metrics used to assess prediction performance.

Metric	**Definition**
R@k (Recall@k)	Fraction of relevant links correctly predicted in the top-k suggestions.
P@k (Precision@k)	Fraction of the top-k predicted links that are actually correct.
HitRate@k	Fraction of nodes for which at least one correct link is in the top-k predictions.
MAP (Mean Average Precision)	Mean of average precision scores across all nodes; emphasizes ranking quality.
MRR (Mean Reciprocal Rank)	Average of the reciprocal of the rank at which the first relevant link is found.

Table 5.2: Evaluation metrics for ranking-based link prediction

5.2 Execution Performance Analysis

5.2.1 Topological Link Prediction

Figure 5.1 plots execution time for topological link prediction over increasing numbers of node pairs on the Cora dataset. For lower query volumes (1K to 10K node pairs), MASS and Neo4j demonstrate comparable performance, with minimal deviation. However, as query volume increases, MASS exhibits improved scalability. At 1 million node pairs, MASS executes nearly 30% faster than Neo4j, completing in approximately 10 seconds compared to Neo4j's 14 seconds.

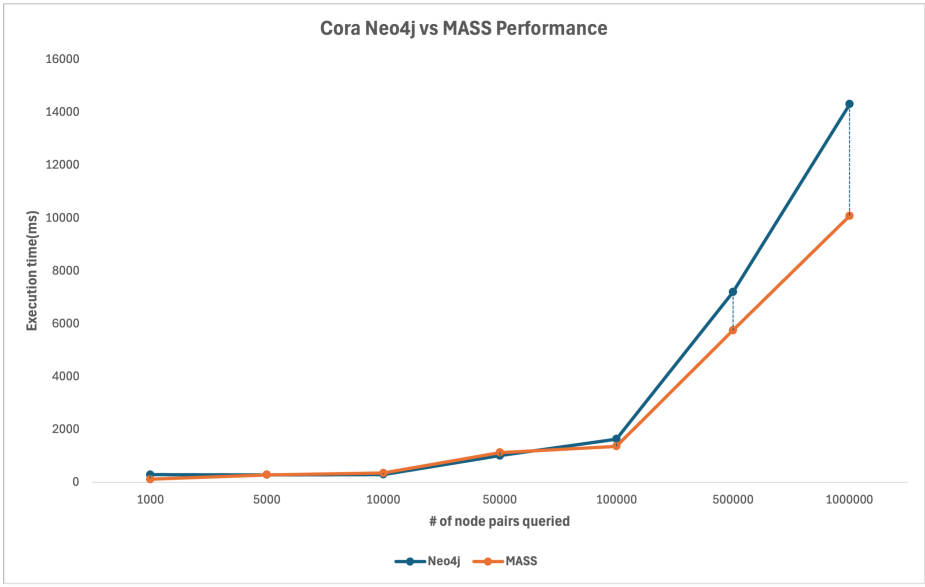

Figure 5.1: Cora topological link prediction performance benchmarks.

This divergence is primarily attributed to architectural differences. MASS stores vertex state entirely in-memory across distributed compute nodes, allowing direct access to neighbors and their attributes. In contrast, Neo4j relies on disk-based access patterns; as the number of node pairs increases, repeated lookups introduce I/O overhead that compounds over time, thereby degrading performance.

These trends are even more pronounced in the OGBL-DDI dataset, shown in Figure 5.2. As a significantly denser graph, OGBL-DDI contains over 1.5 million edges, which amplifies the cost of repeated neighbor access. MASS consistently outperforms Neo4j at higher query volumes, with up to 1.88x speedup observed for 50,000 node pairs.

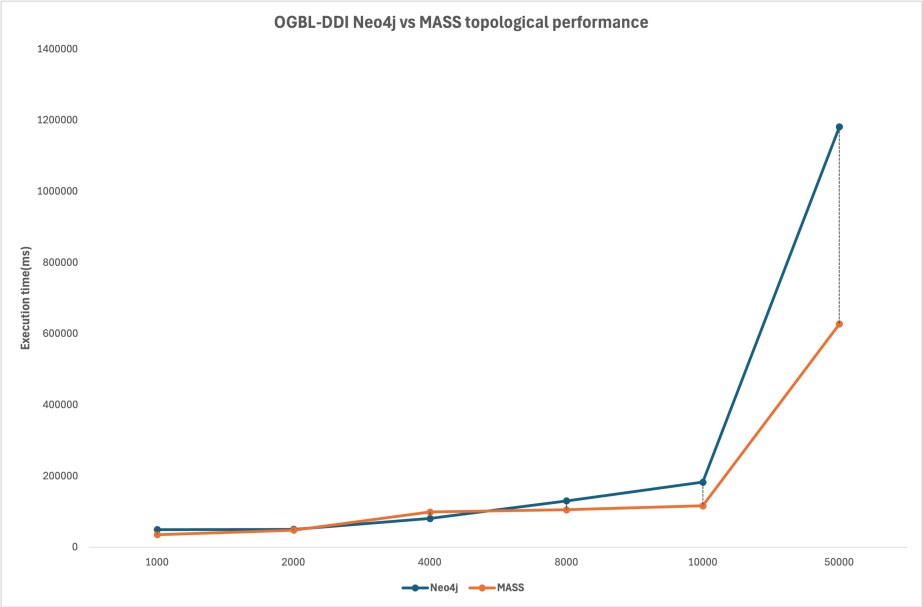

Figure 5.2: OGBL DDI topological link prediction performance benchmarks.

Interestingly, Neo4j initially outpaces MASS for 4,000 node pairs—suggesting that for very small subsets of dense graphs, Neo4j's tight memory layout and on-disk cache optimizations may temporarily compensate for its disk-bound model. However, as the computation scales, MASS's in-memory model yields superior throughput.

To further evaluate MASS's scalability, we measured topological link prediction performance across 1 to 8 compute nodes on the CSSMPI cluster using the Cora dataset (Figure 5.3). Each experiment evaluated approximately 25,000 node pairs generated by comparing 10 random test nodes to all other nodes in the graph. As expected, performance degrades with increasing node count due to inter-node message passing and synchronization costs. While similarity score computation is inexpensive, the cost of distributed vertex access grows linearly, ultimately dominating the runtime.

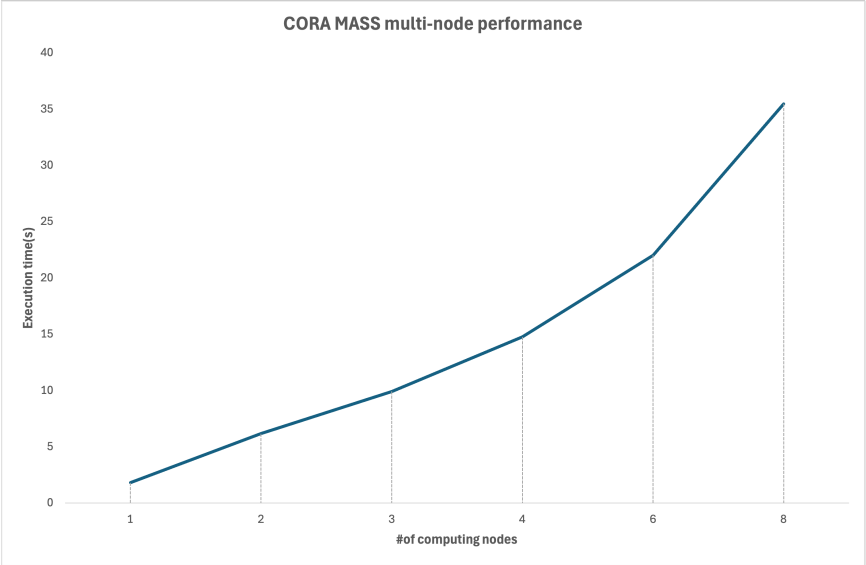

Figure 5.3: MASS multi-node performance on topological queries.

This highlights a tradeoff in the agent-based architecture: the model scales horizontally but is sensitive to the granularity of inter-node communication.

5.2.2 FastRP + KNN Pipeline

In contrast to the topological results, Neo4j outperforms MASS by a wide margin during FastRP embedding generation. As shown in Figure 5.4, Neo4j completed FastRP on the Cora dataset in under 100 milliseconds. In comparison, MASS required between 2 and 7 seconds depending on the number of compute nodes.

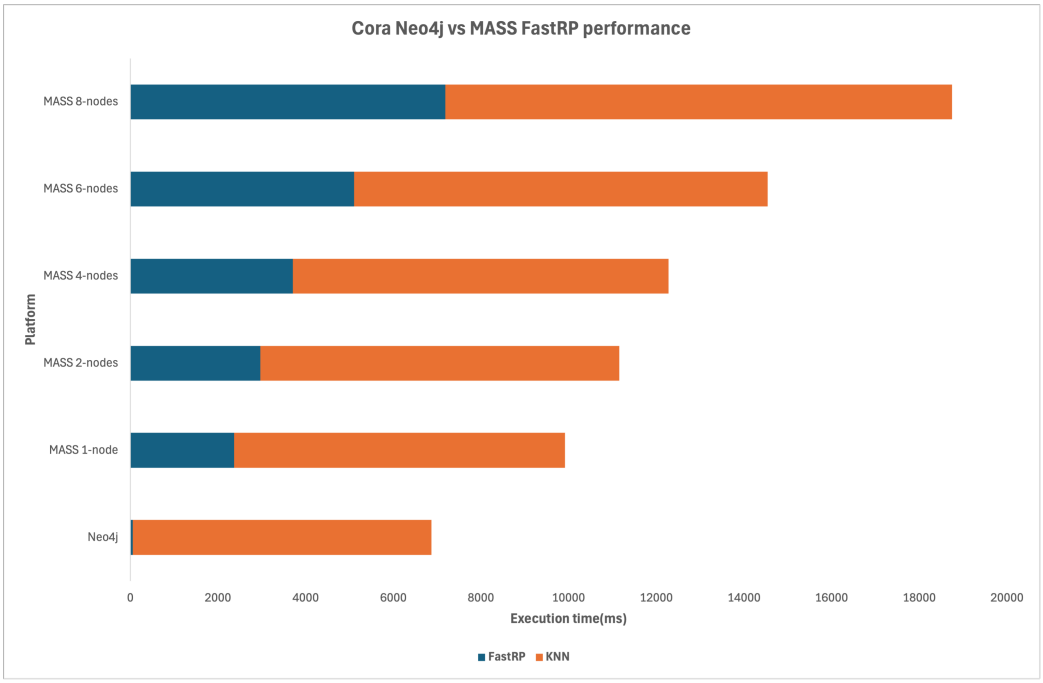

Figure 5.4: Cora - FastRP + KNN time for Neo4j vs. MASS (1–8 nodes).

This performance gap arises from three main factors. First, Neo4j benefits from shared memory locality: node and neighbor embeddings are co-located in the same in-memory space, allowing for fast vector aggregation. Second, MASS relies on migrating agents to fetch embeddings from neighboring nodes, which introduces additional latency from serialization, migration, and synchronization. Third, for lightweight operations such as vector summations and normalizations, the overhead of agent orchestration in MASS becomes disproportionately large.

This trend is also observed in the OGBL-DDI benchmark, albeit with an interesting inversion. While Neo4j again dominates the FastRP embedding step—completing in less than 500 milliseconds, while MASS takes just over 14 mins—its KNN phase takes over 7 minutes. MASS, by contrast, completes the entire KNN computation in under one minute, as shown in Fig 5.5. This substantial improvement highlights the strength of MASS in distributed similarity evaluation.

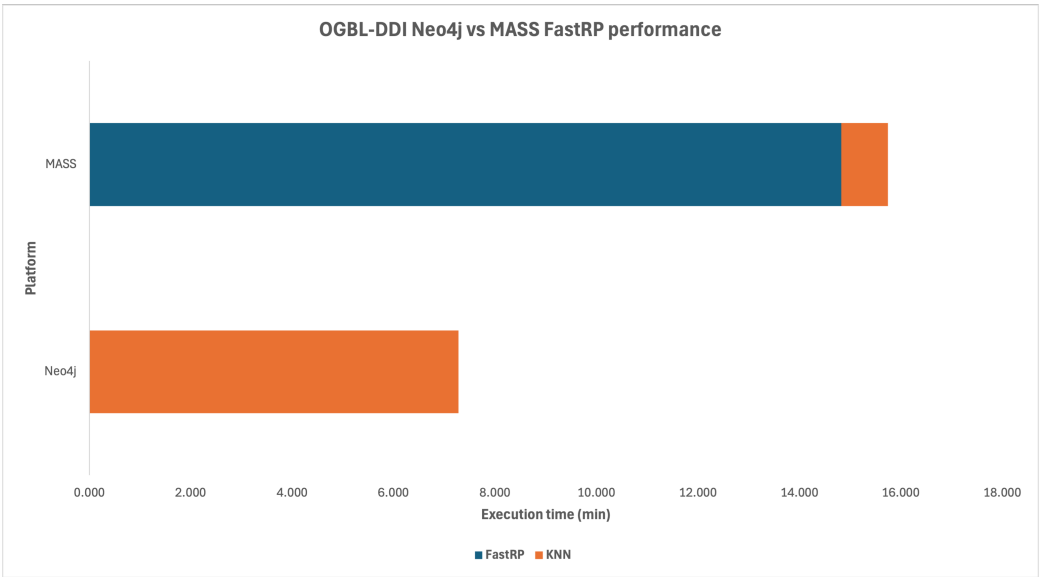

Figure 5.5: OGBL-DDI FastRP + KNN time for Neo4j vs. MASS.

Unlike Neo4j's sequential and approximate CPU-bound KNN evaluation, MASS performs a memory-resident scan over all embeddings using a fixed-size priority queue. By avoiding agent orchestration and keeping inter-node communication minimal, MASS achieves faster similarity ranking—especially on large graphs like OGBL-DDI where Neo4j's performance is limited by data volume and memory constraints.

5.3 Accuracy analysis

5.3.1 Topological Link Prediction

To evaluate link prediction accuracy, we compared five topological algorithms—Adamic-Adar (AA), Common Neighbors (CN), Resource Allocation (RA), Preferential Attachment (PA), and Total Neighbors (TN)—across four ranking-based metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), Precision@K, Recall@K, and HitRate@K. Experiments were conducted identically on both MASS and Neo4j to ensure comparability. Since the underlying implementations and scoring functions are mathematically equivalent, accuracy results are identical across both systems.

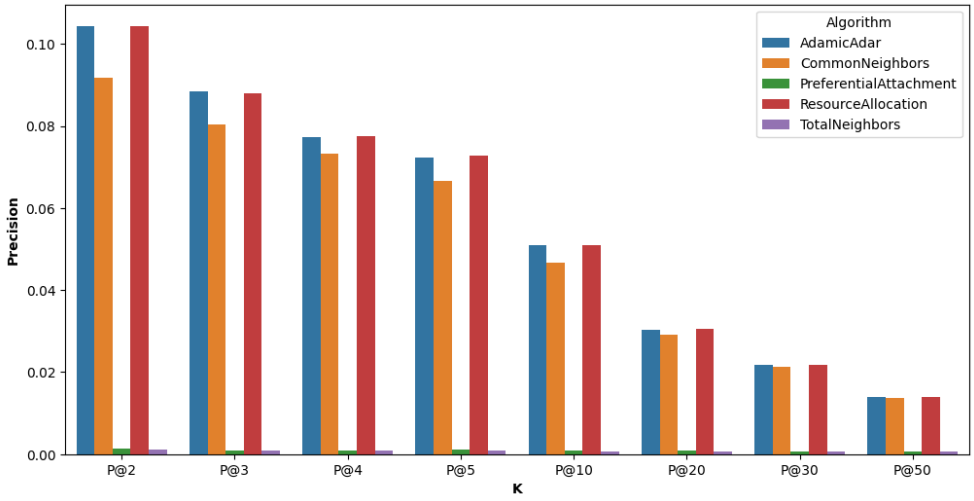

Figure 5.6: Cora - Precision@k by algorithm

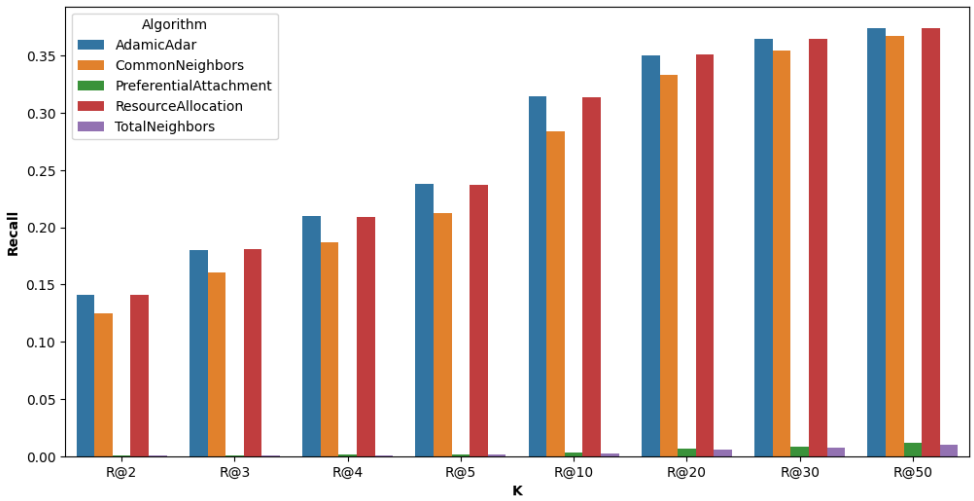

Figure 5.7: Cora - Recall@k by algorithm

Figure 5.6 & 5.7 show that Adamic-Adar and Resource Allocation consistently outperform the remaining heuristics on the Cora dataset.

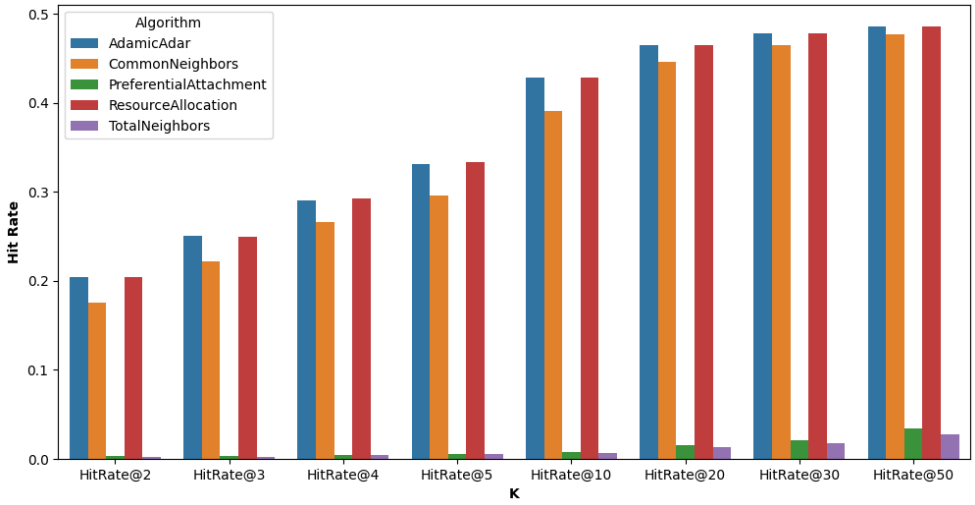

Figure 5.8: Cora - Hitrate@k by algorithm

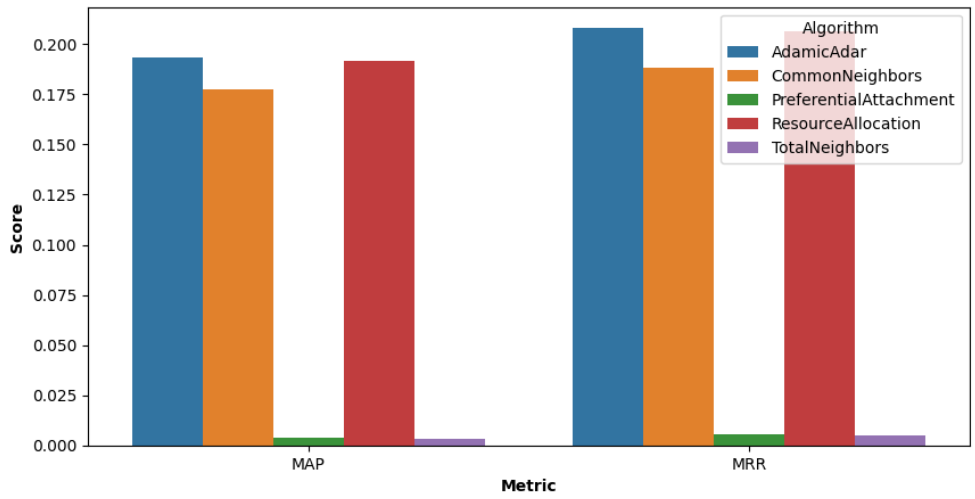

Figure 5.9: Cora - MAP and MRR by algorithm

These algorithms emphasize rare neighbors through inverse-degree weighting, a property that proves particularly effective on sparse graphs. In contrast, Preferential Attachment and Total Neighbors—both biased toward high-degree nodes—perform poorly across all metrics.

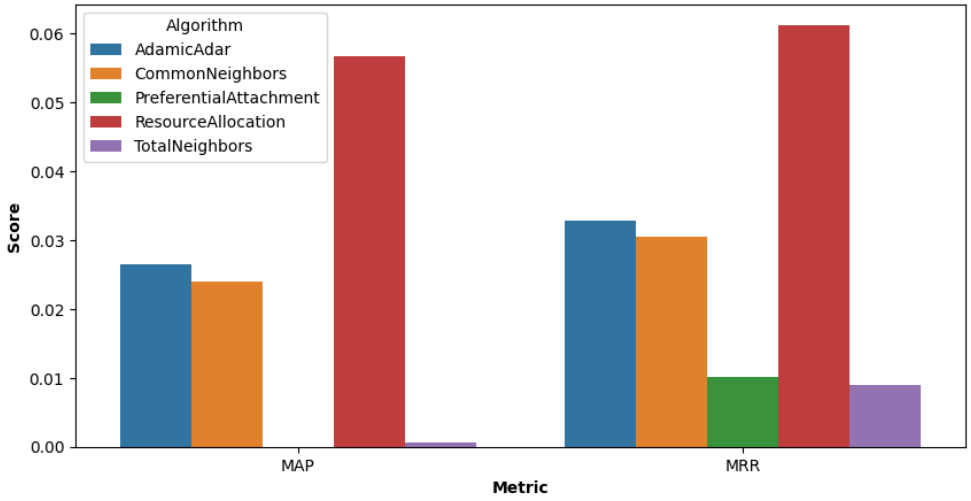

Figure 5.10: OGBL-DDI - MAP and MRR by algorithm

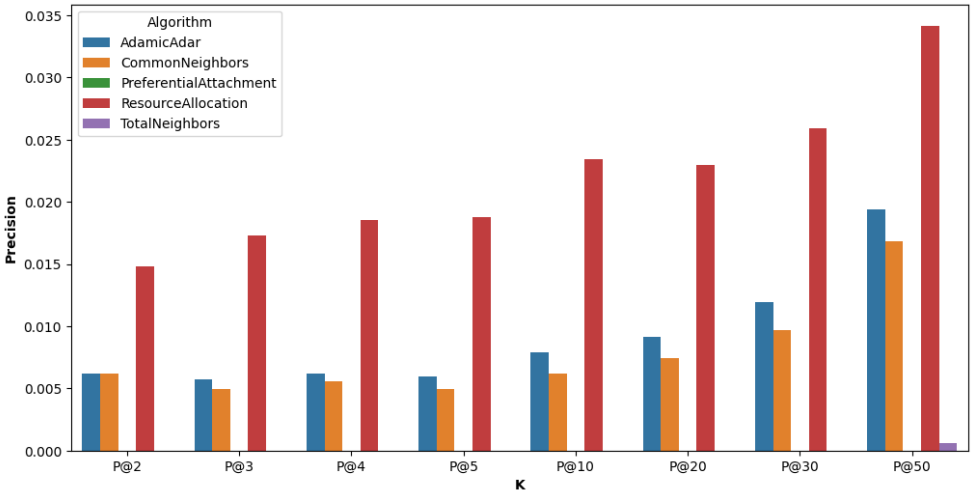

Figure 5.11: OGBL-DDI - Precision@k by algorithm

These trends largely carry over to the denser OGBL-DDI dataset, albeit with lower overall accuracy scores. As seen in Fig. 5.10, MAP and MRR for AA and RA fall from approximately 0.20 on Cora(Fig. 5.9 to 0.03–0.06 on OGBL-DDI.

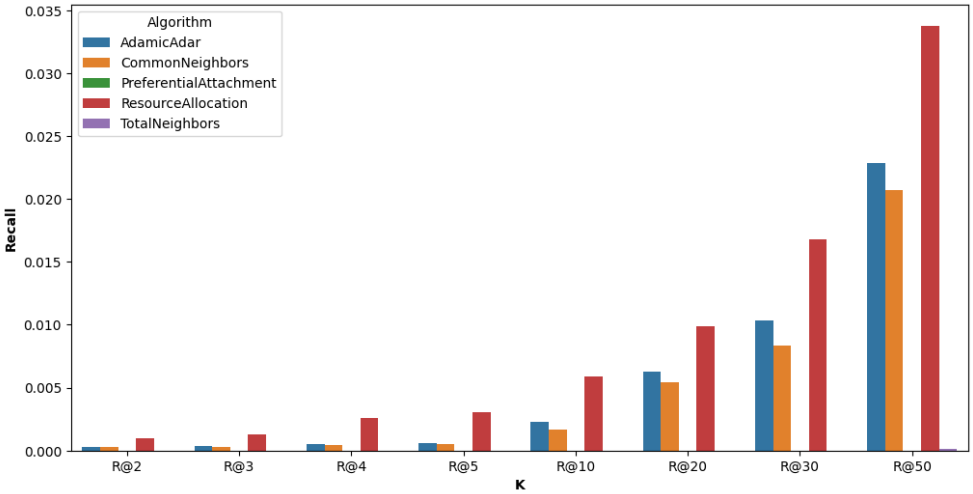

Figure 5.12: OGBL-DDI - Recall@k by algorithm

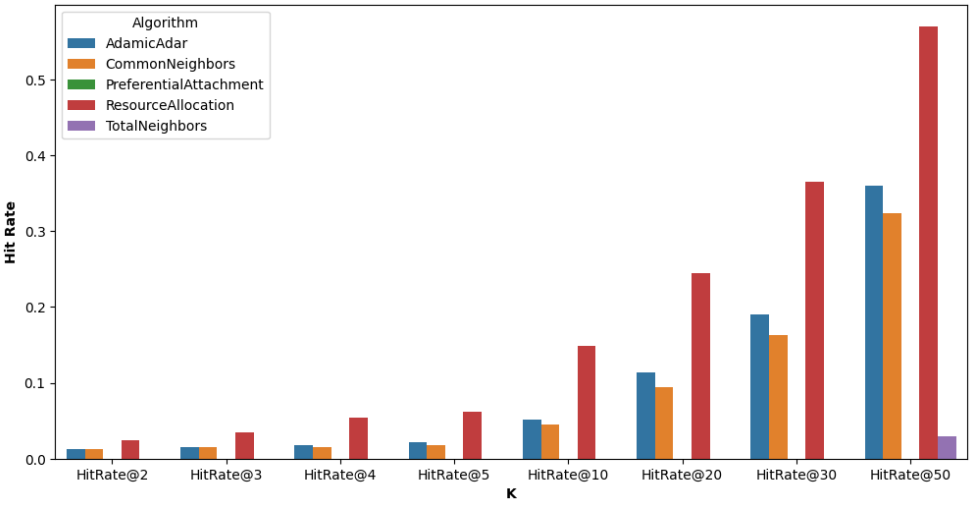

Figure 5.13: OGBL-DDI - Hitrate@k by algorithm

Despite the drop, AA and RA still maintain a clear lead over other methods, affirming their robustness across structural regimes. PA and TN remain ineffective, with near-zero MAP and HitRate scores across both datasets.

Importantly, because MASS and Neo4j use identical implementations for these unsupervised topological algorithms, all accuracy scores—including MAP, MRR, and HitRate@K—are numerically indistinguishable between the two platforms. This validates the correctness of the MASS implementation and confirms that observed differences in runtime are not attributable to discrepancies in algorithmic behavior.

Adamic-Adar and Resource Allocation show strong performance on Precision@K and Recall@K for both datasets, maintaining competitive accuracy even as k increases. On Cora, they achieve HitRate@50 scores of 0.48 and above (Fig. 5.8), while on OGBL-DDI, they reach 0.57 (Fig. 5.13) respectively—despite the greater class imbalance and link sparsity.

The consistency of Adamic-Adar and Resource Allocation across Cora and OGBL-DDI suggests that inverse-degree-based scoring heuristics generalize well across graphs of varying density and connectivity. Their performance remains robust in both sparse and dense regimes, making them strong default choices for scalable, unsupervised link prediction tasks in both MASS and Neo4j.

5.3.2 FastRP + KNN

Although both MASS and Neo4j employ an identical pipeline—FastRP embeddings followed by a KNN-style ranker—their accuracy profiles diverge in ways that trace back to distinct architectural choices.

For the Cora dataset, Neo4j delivers higher early-rank quality: MAP increases from 0.033 (MASS) to 0.053 (Neo4j), while MRR rises from 0.032 to 0.051 (Table A.10). As shown in Fig. 5.14 and Fig. 5.15, at k=5, Precision@K nearly doubles (0.041 vs. 0.022).

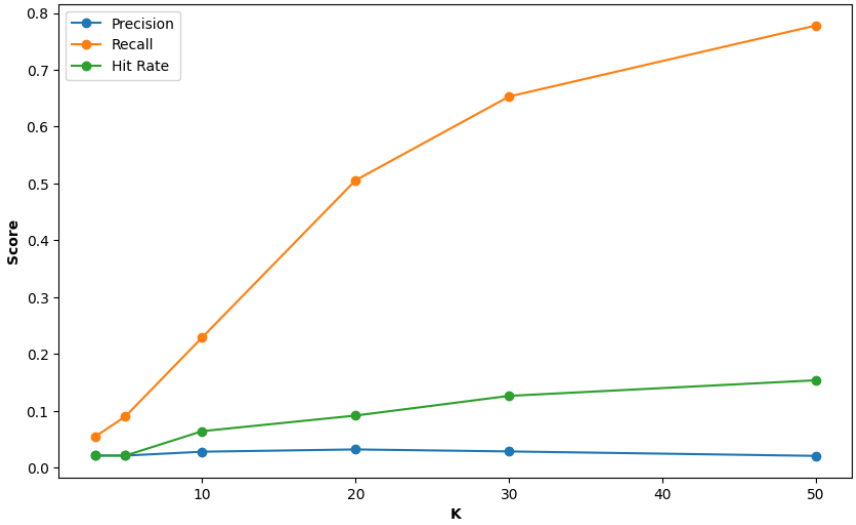

Figure 5.14: Cora - Precision, Recall, and HitRate for MASS

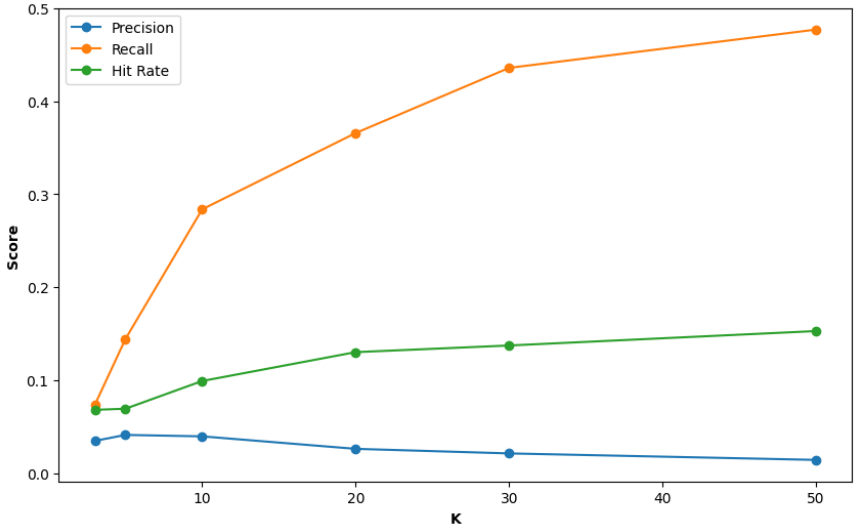

Figure 5.15: Cora - Precision, Recall, and HitRate for Neo4j

MASS, however, surpasses Neo4j in breadth. Its Recall@50 reaches 0.778 (vs. 0.477) and HitRate@50 hits 0.154 (vs. 0.153), indicating that MASS surfaces a broader set of correct links when a larger candidate list is acceptable.

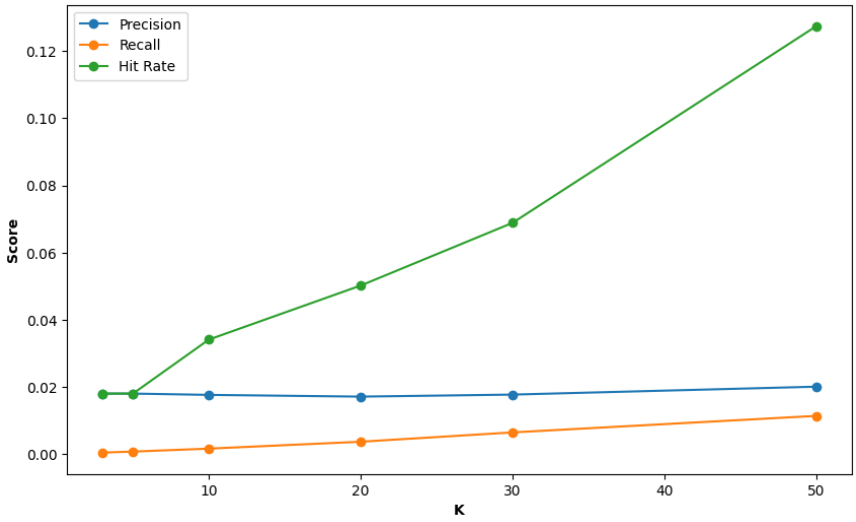

Figure 5.16: OGBL-DDI - Precision, Recall, and HitRate for MASS

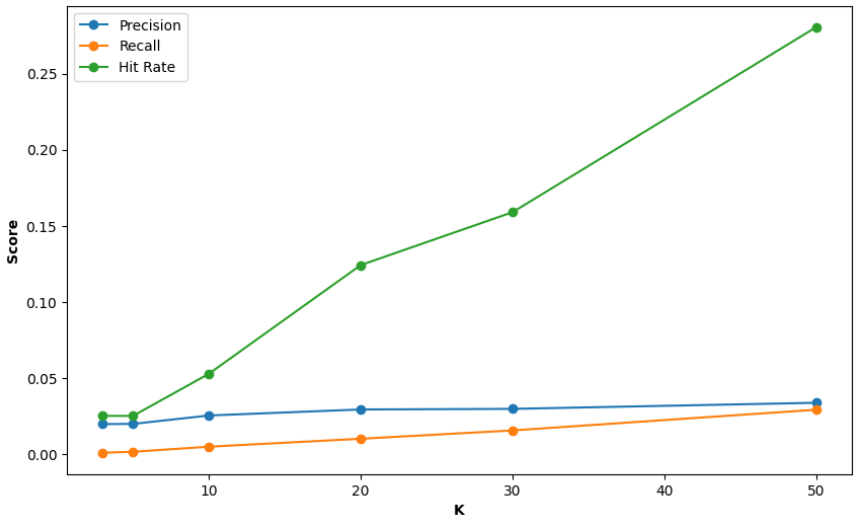

Figure 5.17: OGBL-DDI - Precision, Recall, and HitRate for Neo4j

On the larger, denser OGBL-DDI graph, Neo4j retains a clear advantage in early-rank quality, posting MAP and MRR scores of 0.057 and 0.036 versus 0.034 and 0.028 for MASS. It also edges ahead on Recall and HitRate at every cutoff—for example, Recall@50 is 0.028

for Neo4j compared with 0.011 for MASS. Two architectural factors explain this shift. First, Neo4j's k-nearest-neighbor stage relies on an approximate NN-Descent search that converges rapidly on high-quality neighbors without exhaustively scanning the entire embedding set. This design preserves early-rank precision even as graph size grows. Second, MASS must serialize and deserialize float vectors each time agents migrate, and the non-associative nature of floating-point addition means that the order in which partial similarities arrive can subtly perturb final scores. On dense graphs such as OGBL-DDI, those small inconsistencies accumulate, eroding both precision and recall.

Neo4j's NN-Descent search therefore secures sharper early-rank accuracy on both Cora and OGBL-DDI. MASS's full scan can still match or exceed Neo4j's recall on sparser graphs like Cora, where serialization overhead is modest and embedding updates are more stable. On very dense graphs, however, the combination of serialization cost and non-deterministic accumulation dampens MASS's embedding quality, causing its advantage in coverage to disappear.

In practice, Neo4j is the stronger choice when top-k precision is paramount—for example, in recommendation engines that surface only a handful of results. By contrast, MASS remains attractive for exploratory or retrieval-oriented tasks that value high coverage on moderately sized or sparse graphs and can tolerate slight rank noise as the price of horizontal scalability. Ultimately, each engine aligns with different link-prediction goals: Neo4j favors early-rank fidelity, whereas MASS prioritizes exhaustive discovery across distributed resources.

Chapter 6

LIMITATIONS

Despite demonstrating promising results, the current prototype of MASS exhibits several constraints that temper its applicability and the generalizability of the reported benchmarks. First, MASS ingests graphs in strictly directed form, yet the FastRP implementation treats every edge as undirected by propagating embeddings through both `TO` and `FROM` neighbor lists. This mismatch is harmless on purely undirected datasets such as Cora but complicates analyses on graphs where directionality carries semantic weight. Although the underlying property-graph layer can store edge attributes—including numerical weights—the present FastRP pipeline ignores relationship weights entirely, so weighted interactions have no influence on the learned embeddings.

A second limitation concerns data-loading performance on multi-node clusters. Graph creation in MASS proceeds sequentially: edges are parsed on a single coordinator and forwarded as messages to remote nodes. For large, dense datasets this hand-off becomes the dominant bottleneck. On OGBL-DDI, reliable multi-node ingestion proved impractical, forcing all performance experiments to run on a single machine. Even on a lone node the test split had to be reduced to roughly thirty percent of the available positive links, because a full 100

These creation-time constraints also limited exploration of parallel speed-ups for embedding generation and KNN search. Although MASS's architecture should excel once the graph outgrows the memory of a single Neo4j instance, rigorous evidence of such spatial scalability remains future work. Early tests indicate roughly parity with Neo4j when both use identical hyperparameters; however, MASS's embedding quality could be improved by replacing agent-level aggregation with direct place-to-place vector exchange, thereby eliminating non-deterministic float accumulation and reducing serialization overhead.

Finally, overall accuracy in MASS is highly sensitive to FastRP hyperparameters, which

must be tuned manually by the end user. In biomedical graphs such as OGBL-DDI—where even small errors may have clinical implications—MASS currently offers no built-in mitigation or auditing tools. We therefore emphasize that responsibility for safe deployment rests with practitioners and that the model's predictions should be validated according to domain-specific standards, in line with guidance from the original FastRP authors.

Chapter 7

CONCLUSION & FUTURE WORK

This whitepaper has introduced a hybrid link-prediction framework built directly into MASS Core. By combining classical topological heuristics with FastRP-based embeddings inside MASS's agent-oriented spatial-simulation runtime, the system moves beyond its original modeling focus and into the domain of scalable, graph-native machine learning. The objective was not merely to mirror Neo4j's capabilities, but to rethink link prediction around distributed memory locality and agent coordination.

Experiments show that MASS excels at topological link prediction. As query volume rises, performance scales nearly linearly, and on large graphs the system outpaces Neo4j because vertices remain in memory and can be accessed without disk latency. The picture is more nuanced for embeddings. In the FastRP + KNN pipeline MASS currently trails Neo4j; the cost of serializing vectors, migrating agents, and synchronizing partial results becomes a bottleneck when the underlying computation per step is light. Even so, the embedding pipeline is modular, offering several clear levers—message-based propagation, place-to-place exchange, or localized agent movement—that can be tuned to close the gap.

Accuracy results are encouraging. For topological methods MASS matches Neo4j exactly, an expected outcome given that both systems implement the same scoring formulas. With embeddings the two systems trade strengths. Neo4j enjoys higher MAP and MRR, reflecting sharper precision at very small k, whereas MASS produces higher recall and HitRate at broader cut-offs, particularly on sparser graphs. This suggests that MASS is already well suited for workloads—such as GraphRAG or exploratory analysis—that value exhaustive candidate discovery over early-rank precision. Further work on edge-weighted embeddings and domain-specific hyper-parameter tuning could improve discrimination without sacrificing coverage.

The main technical obstacle lies in KNN-based inference at scale. Neo4j's search remains

centralized, limiting horizontal growth. An initial agent-centric search in MASS proved too expensive, so the current implementation falls back to a master-node scan after distributed vector collection. Revisiting distributed KNN with lightweight message passing or region-bounded agent dispatch could restore scalability while preserving the advantages of locality.

Several avenues now present themselves. Replacing agent migrations with direct place-to-place vector exchange should speed up FastRP propagation and remove floating-point accumulation drift. Expanding the library to include additional heuristics, supervised models, and learning-to-rank pipelines will broaden MASS's analytical repertoire. Support for batched or multi-tenant inference, together with incremental graph updates, would make the framework more attractive for production use. Finally, local-only agent movement can enable sub-graph similarity search—an important capability for recommendation, retrieval-augmented generation, and community detection.

Taken together, these results reposition MASS as more than a simulation engine. It now offers a unified environment in which predictive, symbolic, and agent-based reasoning coexist on the same distributed substrate. Limitations in embedding propagation and global KNN search remain, yet the demonstrated performance, accuracy parity, and architectural flexibility point to a clear path forward. With continued optimization, MASS can mature into an open, programmable platform for the next wave of graph-driven AI systems.

BIBLIOGRAPHY

[1] Lada Adamic and Eytan Adar. "How to search a social network". en. In: *Social Networks* 27.3 (July 2005), pp. 187–203. ISSN: 03788733. DOI: 10.1016/j.socnet.2005.01.007.

[2] Iftikhar Ahmad et al. "Missing Link Prediction using Common Neighbor and Centrality based Parameterized Algorithm". en. In: *Sci Rep* 10.1 (Jan. 2020). Publisher: Nature Publishing Group, p. 364. ISSN: 2045-2322. DOI: 10.1038/s41598-019-57304-y. URL: https://www.nature.com/articles/s41598-019-57304-y.

[3] Albert-Laszlo Barabasi and Reka Albert. "Emergence of scaling in random networks". In: *Science* 286.5439 (Oct. 1999). arXiv:cond-mat/9910332, pp. 509–512. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.286.5439.509. URL: http://arxiv.org/abs/cond-mat/9910332 (visited on 05/29/2025).

[4] Tomaz Bratanic. *A Deep Dive into Neo4j Link Prediction Pipeline and FastRP Embedding Algorithm*. en. Oct. 2021. URL: https://towardsdatascience.com/a-deep-dive-into-neo4j-link-prediction-pipeline-and-fastrp-embedding-algorithm-bf244aeed50d.

[5] *Datasets - Neo4j Graph Data Science Client*. en. https://neo4j.com/docs/graph-data-science-client/1.14/common-datasets/.

[6] Yuan Ma. "An Implementation of Multi-User Distributed Shared Graph". en. In: ().

[7] Haochen Chen et al. *Fast and Accurate Network Embeddings via Very Sparse Random Projection*. en. arXiv:1908.11512 [cs]. Aug. 2019. URL: http://arxiv.org/abs/1908.11512.

[8] *Is Traditional SaaS Behind Us? The Graph + GenAI Revolution*. URL: https://neo4j.com/blog/genai/graph-ai-tier/.

[9] Dimitris Achlioptas. “Database-friendly random projections: Johnson-Lindenstrauss with binary coins”. In: *Journal of Computer and System Sciences*. Special Issue on PODS 2001 66.4 (June 2003), pp. 671–687. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(03)00025-4. URL: https://www.sciencedirect.com/science/article/pii/S0022000003000254.

[10] David Liben-Nowell and Jon Kleinberg. “The Link Prediction Problem for Social Networks”. en. In: ().

[11] Shenyan Cao. “AN INCREMENTAL ENHANCEMENT OF AGENT-BASED GRAPH DATABASE SYSTEM”. In: ().

[12] *Fast Random Projection - Neo4j Graph Data Science*. URL: https://neo4j.com/docs/graph-data-science/current/machine-learning/node-embeddings/fastrp/.

[13] *Topological link prediction - Neo4j Graph Data Science*. en. URL: https://neo4j.com/docs/graph-data-science/2.13/algorithms/linkprediction/.

[14] Aditya Grover and Jure Leskovec. “node2vec: Scalable Feature Learning for Networks”. en. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Francisco California USA: ACM, Aug. 2016, pp. 855–864. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939754. URL: https://dl.acm.org/doi/10.1145/2939672.2939754 (visited on 04/02/2024).

[15] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: Online Learning of Social Representations”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. arXiv:1403.6652 [cs]. Aug. 2014, pp. 701–710. DOI: 10.1145/2623330.2623732. URL: http://arxiv.org/abs/1403.6652 (visited on 06/03/2025).

[16] Vishnu Mohan, Anirudh Potturi, and Munehiro Fukuda. “Automated Agent Migration over Distributed Data Structures:” en. In: *Proceedings of the 15th International Conference on Agents and Artificial Intelligence*. Lisbon, Portugal: SCITEPRESS - Science and Technology Publications, 2023, pp. 363–371. ISBN: 978-989-758-623-1. DOI:

10.5220/0011784500003393. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0011784500003393.

[17] Tao Zhou, Linyuan Lu, and Yi-Cheng Zhang. "Predicting Missing Links via Local Information". In: *Eur. Phys. J. B* 71.4 (Oct. 2009). arXiv:0901.0553 [physics], pp. 623–630. ISSN: 1434-6028, 1434-6036. DOI: 10.1140/EPJB/E2009-00335-8. URL: http://arxiv.org/abs/0901.0553.

Appendix A

BENCHMARKS

A.1 Performance benchmarks

A.1.1 Cora

# of node pairs queried	MASS (ms)	Neo4j (ms)	Gain in performance
1000	118	289	2.45
5000	282	285	1.01
10000	352	296	0.84
50000	1127	1007	0.89
100000	1362	1634	1.20
500000	5748	7196	1.25
1000000	10085	14308	1.42

Table A.1: MASS vs Neo4j topological link prediction execution performance

# of computing nodes	Time (s)
1	1.8
2	6.2
4	9.9
6	22.02
8	35.47

Table A.2: Cora MASS multi-node topological link prediction benchmarks

Platform	FastRP time(ms)	KNN time(ms)
Neo4j	56	6811
Mass 1-node	2373	7550
MASS 2-nodes	2967	8183
MASS 4-nodes	3708	8568
MASS 6-nodes	5100	9437
MASS 8-nodes	7108	11564

Table A.3: MASS vs Neo4j FastRP execution performance

A.1.2 OGBL-DDI

# of node pairs queried	MASS (ms)	Neo4j (ms)	Gain in performance
1000	35574	50011	1.41
2000	48309	50617	1.05
4000	99234	81011	0.82
8000	105402	130670	1.24
10000	116515	183223	1.57
50000	627645	1181716	1.88

Table A.4: MASS vs Neo4j topological link prediction execution performance

Platform	FastRP time(mins)	KNN time(mins)
Neo4j	0.0083	7.2685
MASS	14.8211	0.9237

Table A.5: MASS vs Neo4j FastRP execution performance

A.2 Topological Link Prediction benchmarks

	Cora					**OGBL-DDI**				
Metric	**AA**	**CN**	**PA**	**RA**	**TN**	**AA**	**CN**	**PA**	**RA**	**TN**
MAP	0.1932	0.1776	0.0037	0.1916	0.0032	0.0264	0.0239	0.0000	0.0566	0.0006
MRR	0.2081	0.1882	0.0057	0.2067	0.0048	0.0327	0.0304	0.0101	0.0611	0.0089

Table A.6: MAP and MRR by Algorithm

	CORA					**OGBL-DDI**				
K	**AA**	**CN**	**PA**	**RA**	**TN**	**AA**	**CN**	**PA**	**RA**	**TN**
2	0.1044	0.0917	0.0013	0.1044	0.0012	0.0062	0.0062	0.0000	0.0148	0.0000
3	0.0884	0.0803	0.0009	0.0879	0.0008	0.0058	0.0049	0.0000	0.0173	0.0000
4	0.0773	0.0733	0.0010	0.0776	0.0009	0.0062	0.0056	0.0000	0.0185	0.0000
5	0.0723	0.0667	0.0010	0.0728	0.0010	0.0059	0.0049	0.0000	0.0188	0.0000
10	0.0510	0.0466	0.0008	0.0510	0.0006	0.0079	0.0062	0.0000	0.0235	0.0000
20	0.0303	0.0290	0.0008	0.0305	0.0007	0.0091	0.0074	0.0000	0.0230	0.0000
30	0.0218	0.0212	0.0007	0.0219	0.0006	0.0119	0.0097	0.0000	0.0259	0.0000
50	0.0139	0.0136	0.0008	0.0139	0.0006	0.0194	0.0168	0.0000	0.0342	0.0006

Table A.7: Precision@K by Algorithm

	Cora					OGBL-DDI				
K	**AA**	**CN**	**PA**	**RA**	**TN**	**AA**	**CN**	**PA**	**RA**	**TN**
2	0.1411	0.1250	0.0004	0.1406	0.0004	0.0003	0.0003	0.0000	0.0010	0.0000
3	0.1806	0.1603	0.0004	0.1809	0.0004	0.0004	0.0003	0.0000	0.0012	0.0000
4	0.2098	0.1868	0.0011	0.2088	0.0010	0.0005	0.0005	0.0000	0.0026	0.0000
5	0.2377	0.2126	0.0013	0.2373	0.0012	0.0006	0.0005	0.0000	0.0031	0.0000
10	0.3147	0.2844	0.0028	0.3142	0.0023	0.0023	0.0017	0.0000	0.0059	0.0000
20	0.3507	0.3330	0.0069	0.3509	0.0057	0.0062	0.0054	0.0000	0.0099	0.0000
30	0.3650	0.3545	0.0081	0.3652	0.0074	0.0104	0.0084	0.0000	0.0168	0.0000
50	0.3744	0.3674	0.0119	0.3744	0.0103	0.0229	0.0207	0.0000	0.0338	0.0001

Table A.8: Recall@K by Algorithm

	Cora					OGBL-DDI				
K	**AA**	**CN**	**PA**	**RA**	**TN**	**AA**	**CN**	**PA**	**RA**	**TN**
2	0.2035	0.1754	0.0026	0.2035	0.0024	0.0123	0.0123	0.0000	0.0247	0.0000
3	0.2503	0.2222	0.0026	0.2490	0.0024	0.0148	0.0148	0.0000	0.0346	0.0000
4	0.2905	0.2664	0.0039	0.2918	0.0036	0.0173	0.0148	0.0000	0.0543	0.0000
5	0.3307	0.2959	0.0052	0.3333	0.0048	0.0222	0.0173	0.0000	0.0617	0.0000
10	0.4284	0.3909	0.0078	0.4284	0.0060	0.0519	0.0444	0.0000	0.1481	0.0000
20	0.4645	0.4458	0.0156	0.4645	0.0131	0.1136	0.0938	0.0000	0.2444	0.0000
30	0.4779	0.4645	0.0209	0.4779	0.0179	0.1901	0.1630	0.0000	0.3654	0.0000
50	0.4859	0.4766	0.0339	0.4859	0.0274	0.3605	0.3235	0.0000	0.5704	0.0296

Table A.9: HitRate@K by Algorithm

A.3 FastRP benchamarks

	Cora		OGBL-DDI	
Metric	**MASS**	**Neo4j**	**MASS**	**Neo4j**
MAP	0.0331	0.0533	0.0342	0.0570
MRR	0.0317	0.0510	0.0275	0.0359

Table A.10: MAP and MRR for MASS and Neo4j

	Cora		OGBL-DDI	
k	**MASS**	**Neo4j**	**MASS**	**Neo4j**
3	0.0215	0.0346	0.0180	0.0197
5	0.0215	0.0411	0.0180	0.0198
10	0.0283	0.0395	0.0176	0.0227
20	0.0323	0.0261	0.0171	0.0279
30	0.0288	0.0212	0.0177	0.0301
50	0.0209	0.0142	0.0201	0.0316

Table A.11: Precision@k for MASS and Neo4j

	CORA		OGBL-DDI	
k	**MASS**	**Neo4j**	**MASS**	**Neo4j**
3	0.0543	0.0732	0.0004	0.0008
5	0.0905	0.1443	0.0007	0.0015
10	0.2293	0.2837	0.0016	0.0040
20	0.5058	0.3657	0.0037	0.0101
30	0.6531	0.4358	0.0065	0.0155
50	0.7780	0.4770	0.0114	0.0280

Table A.12: Recall@k for MASS and Neo4j

	Cora		OGBL-DDI	
k	**MASS**	**Neo4j**	**MASS**	**Neo4j**
3	0.0215	0.0680	0.0180	0.0232
5	0.0215	0.0692	0.0180	0.0232
10	0.0644	0.0990	0.0341	0.0528
20	0.0919	0.1301	0.0502	0.0875
30	0.1265	0.1372	0.0689	0.1242
50	0.1539	0.1527	0.1274	0.1905

Table A.13: HitRate@k for MASS and Neo4j

Appendix B

CODE LISTINGS

```
/**
 * Method to filter neighbors based on relation
 * @param neighbors
 * @param relation
 * @return Set<Object> of neighbors after filtering
 */
private Set<Object> filterNeighbors(Map<Object, Object[]> neighbors, String
    relation) {
    // filter map of neighbors based on relation
    if (relation.isEmpty()) {
        return neighbors.keySet();
    }

    return neighbors.entrySet().stream()
        .filter(e -> ((Set<?>) e.getValue()[0]).contains(relation))
        .map(Map.Entry::getKey)
        .collect(Collectors.toSet());
}
```

Listing B.1: Relationship-based neighbor filtering

```
private Object collectEmbeddings() {
    // agent becomes aware of current position
    PropertyVertexPlace currentPlace = (PropertyVertexPlace) this.getPlace()
        ;

    // ORCHESTRATOR AGENT WORKFLOW STARTS HERE
    if(isOrchestrator) {
        // initialize place's atomic counter to keep track of returning
            agents
        currentPlace.initResponseCount();
```

```
9         // set iteration weight
10        currentPlace.setIterationWeight(iterationWeight);
11        // get all neighbor places
12        Map<Object, Object[]> neighbors = currentPlace.getTONeighbors();
13        neighbors.putAll(currentPlace.getFROMNeighbors());
14
15        // Create args array for all the agents we'll spawn
16        EmbeddingTransportArgs[] args = new EmbeddingTransportArgs[neighbors
              .size()];
17        int i = 0;
18
19        // Prepare args for each collector agent
20        for (Map.Entry<Object, Object[]> neighbor : neighbors.entrySet()) {
21                int neighborId = MASS.distributed_map.getOrDefault(neighbor.
                      getKey(), -1);
22                if (neighborId != -1) {
23                        args[i++] = new EmbeddingTransportArgs(
24                                currentPlace.getIndex()[0],
25                                false,
26                                neighborId,
27                                iterationWeight,
28                                true
29                        );
30                }
31        }
32
33        // Spawn all collector agents at once
34        spawn(args.length, args);
35
36        // ORCHESTRATOR AGENT WORKFLOW ENDS HERE
37        kill();
38        return null;
39    }
40
41    // COLLECTOR AGENT WORKFLOW STARTS HERE
42    if(!hasCollectedEmbedding) {
```

```
43          // migrate collector to neighbor to retrieve embedding
44          if(currentPlace.getIndex()[0] == sourceNode) {
45              migrate(destinationNode);
46          } else {
47              // Collect embeddings from neighbors
48              neighborEmbedding = currentPlace.getPreviousEmbedding();
49              hasCollectedEmbedding = true;
50              // migrate back to source
51              migrate(sourceNode);
52              return null;
53          }
54      }
55
56      if(hasCollectedEmbedding && currentPlace.getIndex()[0] == sourceNode) {
57          // deposit embedding at source node
58          currentPlace.receiveEmbedding(neighborEmbedding);
59          // COLLECTOR AGENT WORKFLOW ENDS HERE
60          kill();
61          return null;
62      }
63
64      return null;
65  }
```

Listing B.2: Orchestrator-Collector agent movement

```
1   /**
2    * Find K nearest neighbors for a given query node
3    *
4    * @param queryNodeId ID of the query node
5    * @param k Number of nearest neighbors to find
6    * @return List of k nearest neighbors with their similarity scores
7    */
8   public List<NodeSimilarity> findKNearestNeighbors(String queryNodeId, int k)
        {
9       // Get the embedding for the query node
10      float[] queryEmbedding = embeddings.get(queryNodeId);
```

```
11
12      if (queryEmbedding == null) {
13              throw new IllegalArgumentException("Query node ID not found: " +
                    queryNodeId);
14      }
15
16      // Use a priority queue to keep track of the k nearest neighbors
17      PriorityQueue<NodeSimilarity> nearestNeighbors = new PriorityQueue<>(k);
18
19      // Compute similarities between the query node and all other nodes
20      for (Map.Entry<String, float[]> entry : embeddings.entrySet()) {
21              String nodeId = entry.getKey();
22              float[] nodeEmbedding = entry.getValue();
23
24              // Skip comparing the query node with itself
25              if (!nodeId.equals(queryNodeId)) {
26                      // Calculate cosine similarity between embeddings
27                      double similarity = SimilarityMeasures.cosineSimilarity(
                            queryEmbedding, nodeEmbedding);
28
29                      // Add to nearest neighbors if similarity is higher than
                             the lowest in our priority queue
30                      if (nearestNeighbors.size() < k) {
31                              nearestNeighbors.add(new NodeSimilarity(nodeId,
                                    similarity));
32                      } else if (similarity > nearestNeighbors.peek().
                            getSimilarity()) {
33                              nearestNeighbors.poll(); // Remove the lowest
                                    similarity
34                              nearestNeighbors.add(new NodeSimilarity(nodeId,
                                    similarity));
35                      }
36              }
37      }
38
39      // Convert priority queue to sorted list (highest similarity first)
```

```
40      List<NodeSimilarity> result = new ArrayList<>(nearestNeighbors);
41      Collections.sort(result, Comparator.comparingDouble(NodeSimilarity::
            getSimilarity).reversed());
42
43      return result;
44  }
```

Listing B.3: Top-K neighbor search

Appendix C

PROJECT SETUP AND EXECUTION GUIDE

This appendix details the steps required to build, configure, and run the MASS-based link prediction system. It is intended to help new users, particularly DSLAB researchers, replicate and extend the experiments conducted in this thesis.

C.1 Repository and Branch Information

The project relies on two codebases, hosted on Bitbucket:

C.1.1 Application Layer

Repository: `mass_java_appl`, `sumitjh/link-prediction` branch

URL: https://bitbucket.org/mass$_{a}pplication_{d}evelopers/mass_{j}ava_{a}ppl/src/sumitjh - link - prediction$

C.1.2 Core Library

Repository: `mass_java_core`, `sumitjh/knn-alternate` branch

URL: https://bitbucket.org/mass$_{l}ibrary_{d}evelopers/mass_{j}ava_{c}ore/src/sumitjh - knn.$

For a full explanation of each directory and the implementation details, refer to the `README.md` files included in both repositories.

C.2 Rebuilding MASS Core

To compile the MASS core library after any modifications:

```
cd ~/mass_java_core/
mvn clean package install
```

This builds and installs the updated core into your local Maven repository, ensuring that the application layer can resolve dependencies correctly.

Important: After rebuilding `mass_java_core`, update the `mass.version` tag in the `pom.xml` of `mass_java_appl` to match the new version.

C.3 Building and Running the Application

To rebuild and execute the link prediction application:

```
cd ~/mass_java_appl/QueryGraphDB
mvn package
```

After compilation, run the application using the provided script:

```
sh build_run_lp.sh
```

This script launches the program with the specified dataset and configuration.

C.4 Input Requirements

To execute successfully, the program expects **five arguments**:

1. **Node File**: Full node list of the graph
2. **Train Split**: Subset of edges used to construct the graph
3. **Test Split**: Subset of edges used for prediction
4. **File Suffix**: A custom suffix to differentiate result CSV filenames
5. **Log Level**: Use `OFF` for optimal runtime performance

C.5 Sample Execution

Snapshot of terminal showing successful program execution:

Figure C.1: Execution snapshot

C.6 Output Artifacts

Two result files will be generated:

- `knn_analysis_results_<suffix>.csv`: Embedding-based similarity scores
- `topology_link_prediction_results_<suffix>.csv`: Topological or hybrid similarity scores

Sample snapshots of outputs:

QueryGraphDB > target > classes > knn_analysis_results_test.csv

```
query_node_id,k_value,neighbor_rank,neighbor_node_id,similarity_score,query_time_ms,total_neighbors_found
137868,3,1,1110000,0.995825,8.166,3
137868,3,2,1131277,0.994019,8.166,3
137868,3,3,1131300,0.988289,8.166,3
137868,5,1,1110000,0.995825,4.185,5
137868,5,2,1131277,0.994019,4.185,5
137868,5,3,1131300,0.988289,4.185,5
137868,5,4,38537,0.979555,4.185,5
137868,5,5,561809,0.973693,4.185,5
137868,10,1,1110000,0.995825,3.909,10
137868,10,2,1131277,0.994019,3.909,10
137868,10,3,1131300,0.988289,3.909,10
137868,10,4,38537,0.979555,3.909,10
137868,10,5,561809,0.973693,3.909,10
137868,10,6,261040,0.973089,3.909,10
137868,10,7,1107325,0.973005,3.909,10
137868,10,8,82664,0.972895,3.909,10
137868,10,9,646195,0.972305,3.909,10
137868,10,10,1131270,0.971745,3.909,10
137868,20,1,1110000,0.995825,4.032,20
137868,20,2,1131277,0.994019,4.032,20
137868,20,3,1131300,0.988289,4.032,20
137868,20,4,38537,0.979555,4.032,20
137868,20,5,561809,0.973693,4.032,20
137868,20,6,261040,0.973089,4.032,20
137868,20,7,1107325,0.973005,4.032,20
137868,20,8,82664,0.972895,4.032,20
137868,20,9,646195,0.972305,4.032,20
137868,20,10,1131270,0.971745,4.032,20
137868,20,11,141324,0.971579,4.032,20
137868,20,12,1129368,0.971106,4.032,20
137868,20,13,22875,0.971071,4.032,20
137868,20,14,1117184,0.970971,4.032,20
137868,20,15,1110947,0.970459,4.032,20
```

Figure C.2: Embedding based results snapshot

QueryGraphDB > target > classes > topology_link_prediction_results_test.csv

```
nodeId1,nodeId2,algorithm,score,time_ms
137868,31336,AdamicAdar,0.000000,7.435
137868,31336,ResourceAllocation,0.000000,0.827
137868,31336,PreferentialAttachment,24.000000,0.083
137868,31336,CommonNeighbors,0.000000,0.154
137868,31336,TotalNeighbors,10.000000,0.712
137868,1061127,AdamicAdar,0.000000,0.143
137868,1061127,ResourceAllocation,0.000000,0.123
137868,1061127,PreferentialAttachment,0.000000,0.036
137868,1061127,CommonNeighbors,0.000000,0.128
137868,1061127,TotalNeighbors,6.000000,0.172
137868,1106406,AdamicAdar,0.000000,0.167
137868,1106406,ResourceAllocation,0.000000,0.137
137868,1106406,PreferentialAttachment,18.000000,0.055
137868,1106406,CommonNeighbors,0.000000,0.128
137868,1106406,TotalNeighbors,9.000000,0.157
137868,13195,AdamicAdar,0.000000,0.205
137868,13195,ResourceAllocation,0.000000,0.168
137868,13195,PreferentialAttachment,18.000000,0.036
137868,13195,CommonNeighbors,0.000000,0.109
137868,13195,TotalNeighbors,9.000000,0.813
137868,37879,AdamicAdar,0.000000,0.159
137868,37879,ResourceAllocation,0.000000,0.127
137868,37879,PreferentialAttachment,6.000000,0.051
137868,37879,CommonNeighbors,0.000000,0.124
137868,37879,TotalNeighbors,7.000000,0.177
137868,1126012,AdamicAdar,0.000000,0.145
137868,1126012,ResourceAllocation,0.000000,0.134
137868,1126012,PreferentialAttachment,6.000000,0.047
137868,1126012,CommonNeighbors,0.000000,0.082
137868,1126012,TotalNeighbors,7.000000,0.155
137868,1107140,AdamicAdar,0.000000,0.151
137868,1107140,ResourceAllocation,0.000000,0.140
137868,1107140,PreferentialAttachment,6.000000,0.035
```

Figure C.3: Topological results snapshot

These files contain link prediction results and are saved in the target/classes directory with headers denoting node pairs and their similarity scores.

C.7 Post-Processing and Evaluation

A sample Python notebook is provided in the `mass_java_appl/QueryGraphDB` folder (`sumitjh/link-prediction` branch) to:

- Parse the result CSVs
- Calculate evaluation metrics such as Precision@k, Recall@k, MAP, and MRR
- Visualize performance across algorithms and datasets