Link Prediction in MASS Java: An Overview of Progress over Autumn 2024 and Winter 2025

Table of Contents

[**1 Purpose**](#_heading=h.kzct9eniz5do) **2**

[**2 Summary of Project Progress**](#_heading=h.4o9cjoss7udu) **2**

[**3 Introduction to MASS Java Link Prediction**](#_heading=h.ytrm30702fjl) **2**

[3.1 Background](#_heading=h.p4dl3b8rah8m) 2

[**4 Autumn Quarter Progress**](#_heading=h.5uc3yoyma9ty) **3**

[4.1 Link Prediction Implementation](#_heading=h.7jwvltmpqmmj) 3

[4.2 Benchmarking in MASS](#_heading=h.w14q7tgpxy3c) 3

[4.3 Benchmarking in Neo4J](#_heading=h.7h73m768j54y) 4

[**5 Winter Quarter Progress**](#_heading=h.n4slwaai3z63) **5**

[5.1 Link Prediction Benchmarking](#_heading=h.c86rfv37jqau) 5

[5.2 Threshold Analysis for OGBL Datasets](#_heading=h.1pfp52gvhn5t) 7

[**6 Appendix**](#_heading=h.hfadp6d1tvnl) **8**

[6.1 Link Prediction Files](#_heading=h.okpc1yq5dkkk) 8

[6.2 Benchmark Files](#_heading=h.nca8nc9bpakx) 9

[6.3 Instructions To Run Program](#_heading=h.nhp6frxfmtub) 9

[6.3.1 MASS Benchmarking](#_heading=h.b3h8zbgg6i0k) 9

[6.3.2 Neo4j Benchmarking](#_heading=h.kpybf1894hd2) 9

[6.3.3 Threshold Analysis](#_heading=h.o89r2xjbqaoo) 9

[6.4 Link Prediction Data](#_heading=h.yq0mz0j02u1m) 9

# 

# 1 Purpose

The purpose of this work is to implement agent-based link prediction algorithms in MASS Java and benchmark their performance, as well as to benchmark the performance of competitor’s link prediction algorithms.

# 2 Summary of Project Progress

All five link prediction algorithms supported by Neo4j have been implemented in MASS Java: common neighbors, total neighbors, adamic adar, preferential attachment, and resource allocation. Though the implementation was initially supposed to be agent-based, we shifted away from the use of agents after deciding that agent use would not be necessary for these algorithms.

Additionally, I have developed a link prediction benchmarking application in MASS Java. The application allows the user to load a graph and time the execution of all link prediction algorithms on all node pairs in the graph.

Finally, I have developed a threshold analysis python program to generate ROC (Receiver Operating Characteristic) graphs to provide users with insight into optimal threshold values to use for any given graph.

# 3 Introduction to MASS Java Link Prediction

## 3.1 Background

My project’s work is done in the MASS GraphDB library. The MASS GraphDB is a graph database built upon the existing MASS Java library for distributed multi-agent spatial simulation applications. For details regarding the implementation of MASS GraphDB, one can reference previous DSLab student Lillian Cao’s Spring 2024 white paper: [An Incremental Enhancement of Agent-Based Graph Database System](https://depts.washington.edu/dslab/MASS/reports/LilianCao_whitepaper.pdf).

MASS GraphDB provides users with a distributed graph database that runs over multiple compute nodes. In the graph database model, information is stored in the form of nodes and relationships. Additionally, properties can be attached to a node or relationship in key-value form. For example, a node can have a property consisting of { “name” : “Alice” }. This data management paradigm allows for more natural data storage as compared to traditional database models, as human thinking tends to align closely with nodes and relationships. MASS GraphDB also has limited support for the Cypher query language; users can use CREATE and MATCH clauses to manipulate and query the database.

# 4 Autumn Quarter Progress

## 4.1 Link Prediction Implementation

The link prediction in MASS was worked on primarily by Sumit Hotchandani. The implementation can be broken down into two parts: the Neighbors class and the LinkPrediction class.

The **Neighbors** class provides an interface to retrieve the neighbors of a given vertex with optional directional and property-based filtering. This class provides utility methods for the LinkPrediction class:  **degree()**, **commonNeighbors()**, and **totalNeighbors()**.

**degree()** takes a single vertex as an argument and returns the number of neighbors it has.  **commonNeighbors()** takes in two vertices and returns the number of neighbors that they have in common.

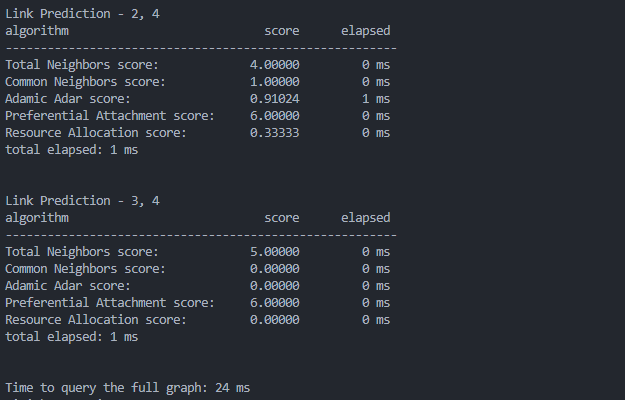
**totalNeighbors()** takes in two vertices and returns the total number of unique vertices that neighbor those two vertices.

The original plan was to use agents to traverse the graphs to determine common neighbors between any two vertices, however it was decided that this was unnecessary as vertices already maintain a list of their neighbors that can be retrieved using getter methods.

The **LinkPrediction** class implements five link prediction algorithms: **common neighbors**, **total neighbors**, **adamic adar**, **resource allocation similarity**, and **preferential attachment**. Each of these algorithms are essentially a mathematical formula that takes neighbor information as input. The LinkPrediction class uses the utilities provided by the Neighbors class to gather this information, execute the specified algorithm, and return a link prediction score.

## 4.2 Benchmarking in MASS

I have written a basic benchmarking program in MASS GraphDB. The program takes in a graph in the format of a pipe-separated list and loads an instance of the graph across a compute cluster. The benchmark program iterates sequentially through every node pair in the graph and executes all five link prediction algorithms on each pair, up to a user-specified number of node pairs. These operations are timed at different levels of granularity. In order from smallest to largest, these are: time to execute a single algorithm, time to execute all algorithms on a single pair, and time to execute all algorithms on the entire graph. Figure 1 shows the output format of this program.

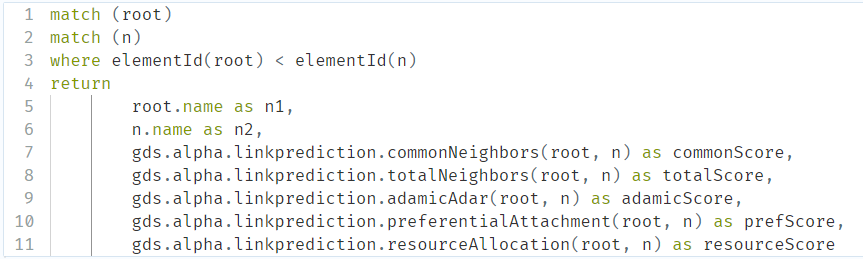


**Figure 1: MASS Benchmark Program Output**

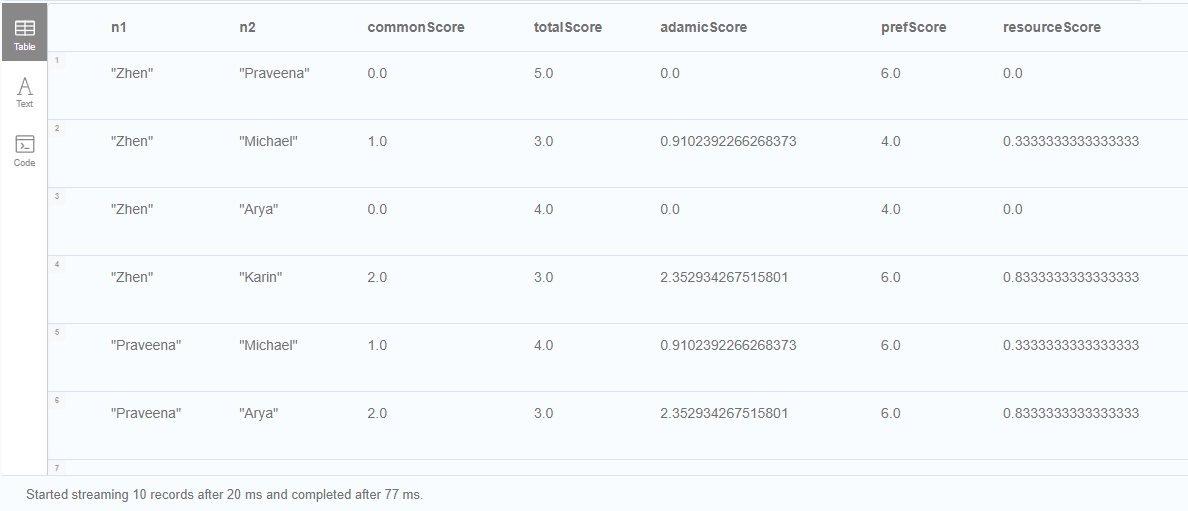
## 

## 4.3 Benchmarking in Neo4J

I have written a similar program in Neo4j that conducts the same set of queries for each node pair. This program is in the form of a cypher query as shown in Figure 2. Figure 3 shows the output from this program in the Neo4j desktop client.



**Figure 2: Neo4j Cypher Benchmark**

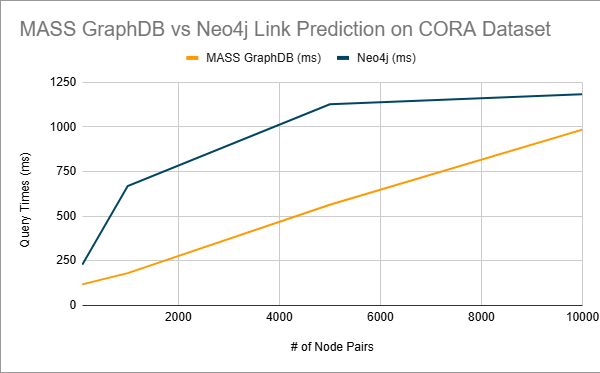
****

**Figure 3: Neo4j Cypher Benchmark Output**

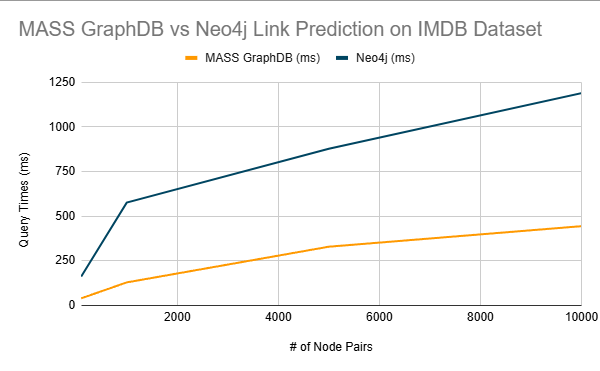
# 5 Winter Quarter Progress

## 5.1 Link Prediction Benchmarking

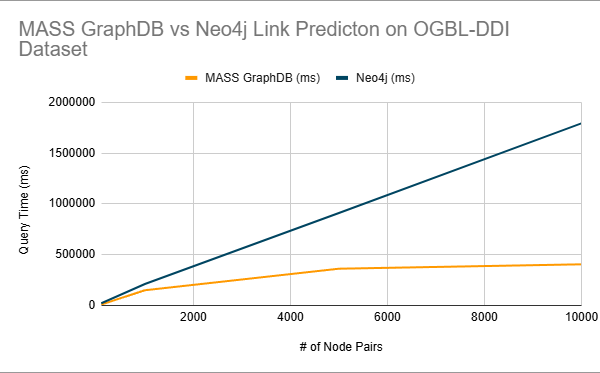
I continued benchmarking datasets in both MASS and Neo4j. Specifically, I have run link prediction queries on the CORA, IMDB, and OGBL-DDI datasets. The benchmarks were run with the following specifications: one query is all five link prediction algorithms run on a node pair, node pairs are sequentially sampled from the graph such that the pairs queried are the same in both MASS and Neo4j, and MASS log level is set to ‘OFF’.



**Figure 4: MASS GraphDB vs Neo4j Link Prediction CORA**



**Figure 5: MASS GraphDB vs Neo4j Link Prediction IMDB**



**Figure 6: MASS GraphDB vs Neo4j Link Prediction OGBL-DDI**

Results demonstrate a 2-3x speedup over Neo4j on average across all datasets. This is mainly due to Neo4j storing the graph on the disk, whereas MASS GraphDB stores the graph in memory. Both programs scale linearly, with a caveat being that Neo4j occasionally catches up to MASS, as can be seen in Figure 4. This is likely due to Neo4j chunking its reads from the disk, so when the amount of data read aligns with the amount needed, the execution is more efficient.

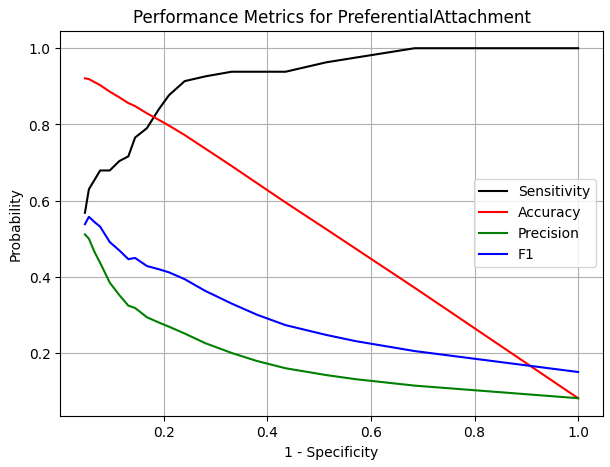
One important detail to note is that in the OGBL-DDI dataset 10,000 node pair query, execution speed in MASS doesn't seem to increase much from 5,000 node pairs. I am not sure why this is, it might be worth further investigation here.

Another important detail is that these results are on a cold start of a Neo4j database. Neo4j becomes significantly faster after an initial query. This is likely either due to traditional caching, or Neo4j’s query plan cache, which stores information about query plans so that similar successive queries execute faster.

These results can be reproduced by following the instructions in the code repo, a link to which can be found in the appendix of this document.

## 5.2 Threshold Analysis for OGBL Datasets

An additional component of this project is threshold analysis. Threshold analysis is the process through which we can determine appropriate threshold values for each link prediction algorithm. I built a python program that takes in output from MASS’s link prediction and processes the data into a series of ROC graphs for each algorithm. This assists users in determining effective algorithms and threshold values for any specific graph.



**Figure 7: ROC curve for Preferential Attachment in OGBL-DDI**

The ROC curve in Figure 7 shows various metrics for a range of different threshold values in the OGBL-DDI program. These results were generated by splitting all edges into a 85/15 test/validation split. The x-axis is 1-specificity, which represents the threshold value decreasing from left to right. It can be seen how at high threshold values, sensitivity or false positive rate is low. As the threshold lowers, sensitivity increases and F1, accuracy, and precision drop.

This program exists as a Jupyter Notebook in the LinkPrediction application linked in the appendix of this document. Instructions for its use are contained in the notebook.

# 

# 6 Appendix

## 6.1 Link Prediction Files

At this time these files can be found on the **sumitjh/link-prediction** branch of the mass\_java\_core repository on bitbucket under the path: /mass\_java\_core/src/main/java/edu/uw/bothell/css/dsl/MASS/graph/ml/linkprediction

**Neighbors.java-**Neighbors interface

**LinkPrediciton.java-**Contains the five link prediction algorithms supported by MASS

## 6.2 Benchmark Files

At this time these files can be found on the **tyler/link-prediction-benchmarking** branch of the mass\_java\_appl repository on bitbucket under the path:

/mass\_java\_appl/Benchmarks/LinkPrediction/

**GraphDBHandler.java -** contains the benchmark application code

**GraphManager.java -** manages loading the graph from the input file

**neo4j\_link\_prediction.ipynb -** notebook containing instructions on benchmarking Neo4j

**threshold\_analysis.ipynb -** notebook containing threshold analysis and documentation

## 

## 

## 

## 6.3 Instructions To Run Program

### 6.3.1 MASS Benchmarking

To run the application on cssmpi or hermes, complete the following steps from your root directory:

1. Clone [mass\_java\_core](https://bitbucket.org/mass_library_developers/mass_java_core/src/master/) and [mass\_java\_appl](https://bitbucket.org/mass_application_developers/mass_java_appl/src/master/) repos.
2. In mass\_java\_core, checkout the sumitjh/link-prediction branch. In mass\_java\_appl, check out the tyler/link-prediction-benchmarking branch. In the future, these changes will be merged into develop or master and may be found there.
3. Cd into mass\_java\_core and run ‘mvn install -Dmaven.test.skip’ to build MASS.
4. Cd into mass\_java\_appl/Benchmarks/LinkPrediction/ and run ‘mvn package -Dmaven.test.skip’ to build the application.
5. From the ‘LinkPrediction/’ directory, run ‘sh run\_appl.sh’

### 6.3.2 Neo4j Benchmarking

Instructions can be found in the **neo4j\_link\_prediction.ipynb** file in the project.

### 6.3.3 Threshold Analysis

Instructions can be found in the **threshold\_analysis.ipynb** file in the project.

## 6.4 Link Prediction Data

**Table 1: MASS GraphDB vs Neo4j Link Prediction OGBL-DDI**

| # Of Node Pairs | MASS GraphDB (ms) | Neo4j (ms) | Gain in Performance (Neo4j / MASS) |
| --- | --- | --- | --- |
| 100 | 8724 | 21461 | 2.46 |
| 1000 | 149725 | 210776 | 1.41 |
| 5000 | 361627 | 910476 | 2.52 |
| 10000 | 404991 | 1793143 | 4.43 |

**Table 2: MASS GraphDB vs Neo4j Link Prediction CORA**

| # Of Node Pairs | MASS GraphDB (ms) | Neo4j (ms) | Gain in Performance (Neo4j / MASS) |
| --- | --- | --- | --- |
| 100 | 118 | 228 | 1.93 |
| 1000 | 181 | 669 | 3.7 |
| 5000 | 564 | 1127 | 2 |
| 10000 | 985 | 1183 | 1.2 |

**Table 3: MASS GraphDB vs Neo4j Link Prediction IMDB**

| # Of Node Pairs | MASS GraphDB (ms) | Neo4j (ms) | Gain in Performance (Neo4j / MASS) |
| --- | --- | --- | --- |
| 100 | 40 | 162 | 4.05 |
| 1000 | 129 | 576 | 4.47 |
| 5000 | 329 | 878 | 2.67 |
| 10000 | 444 | 1189 | 2.68 |