Development of an Agent-based Graph Database Benchmarking Dataset

Michelle Dea

Term Report Winter 2024

Project Committee:

1. Project Overview

The Multi-Agent Spatial Simulation (MASS) library is a parallel programming library using agent-based modeling to simulate a number of collective behaviors, ex: biological agents, and computational geometry algorithms [1]. MASS at its core is comprised of two main components, Places and Agents. Places represents a matrix of elements dynamically allocated over a cluster of nodes. These elements can exchange information with any other element, i.e. Place. Agents can perform computations, and can migrate between different Places, allowing them to interact with other Agents and Places.

When considering big data computing, frameworks such as MapReduce and Spark are some of the more common tools being used to process large volumes of data. However, these tools mainly deal with data in the form of text. For more complex data structures, such as graphs, where the data points are interconnected, a more suitable approach is to use a graph database. More specifically, agents can be leveraged to support analysis of graphs, as they can be deployed into data structures mapped over distributed memory. In this way, we can build a graph database application that can construct graphs over distributed memory.

Harshit Rajvaidya's recent work implemented an agent-based graph database using a MASS application, i.e. MASS Graph Database (MASS Graph DB) [2]. More specifically, his work used a combination of OpenCypher queries, and Agents to perform those queries. As part of his work, he conducted performance comparisons against two popular graph database systems: Neo4j, and RedisGraph. Both of which support OpenCypher queries. Rajvaidya's performance comparison was done using very simple graph datasets he created. The use of these simple datasets was largely to showcase the querying capabilities of MASS Graph DB, rather than execution speed. As part of a continuation of his work, my project focuses on creating several larger graph datasets (at a minimum 500 nodes, and 500 edges/relationships per dataset) for benchmarking purposes with a focus on evaluating execution speed and spatial scalability. Throughout my work, my references to nodes and vertices as they relate to graphs represent the same thing, an entity within a graph.

It was recently announced that the application RedisGraph will soon be sunsetted. In the interest of longevity, a new graph database application was selected for my project. Instead of RedisGraph, I will be using ArangoDB for benchmarking purposes.

We will be comparing our performance against Neo4j and ArangoDB to evaluate the potential of MASS Graph DB. We can identify the strengths and weaknesses of an agent-based graph database in comparison to these popular graph databases by using graph datasets that vary in different aspects, specifically type of data and size of the data. Specifically, I will be doing a comparative analysis of the execution speed (query throughput) and spatial scalability (query response time) of these three graph databases. To ensure the comparison is fair, I will be creating a tool that randomly generates the graph dataset based on certain parameters, such as topology and size, and I will be developing a way for the resulting dataset to be loaded into the three different graph databases. For my project, I identified three popular use cases for graph databases. These include recommendation engines, fraud detection, and social networks. The datasets for each of these use cases are unique in their topology, which is why a parameter for this tool will account for topology to generate a realistic random graph for benchmarking.

The purpose of this work is to be able to measure execution performance of MASS against Neo4j and ArangoDB using these benchmarking applications. As part of these benchmarking applications, standard queries will be written to provide an efficient way for future researchers to benchmark their MASS Graph DB programs.

2. Goals

The main goal of this project is to create a benchmarking dataset and tools to simplify benchmarking MASS Graph DB against popular graph databases. The motivation behind this is to create tools such that future researchers can devote their time to improving the MASS Graph DB, rather than spending resources on investigating methods to conduct performance or execution testing. This work will also standardize how future MASS Graph DB changes will be evaluated, establishing consistency and improving accuracy when it comes to performance testing. To further outline the specific goals for this project, here are the explicit tasks I hope to accomplish:

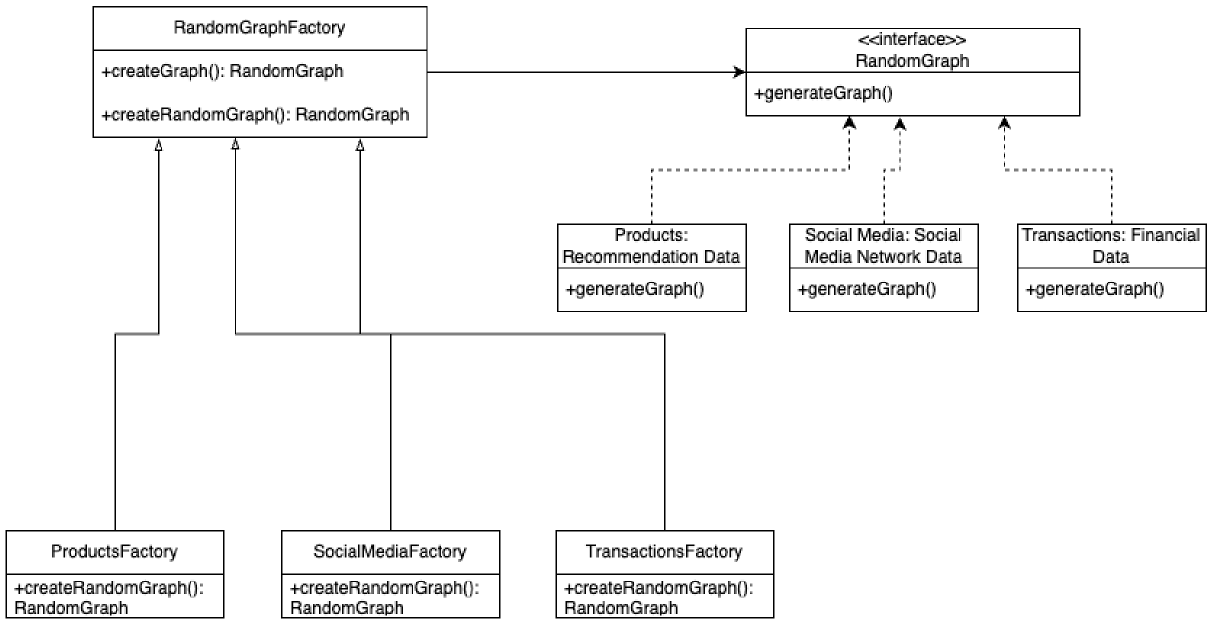
1. Create tools to randomly generate several graph datasets that can be loaded into MASS Graph DB, Neo4j, and ArangoDB. These datasets are modeled after graphs pulled from real world applications based on the use cases mentioned above, and will be automatically generated with some randomness.
2. Create standard queries for each of the datasets that can be run across each graph database for measuring performance.
3. Identify the strengths and weaknesses of the MASS Graph DB compared to Neo4j and ArangoDB.
4. Create an efficient process or tool for running these programs for future researchers.

3. Achievements

This quarter I began by finishing my work of creating csv files for the other two datasets (Twitch and Cryptocurrency)[3][4]. After implementing functions to take care of csv generation for those datasets, I tested loading them into Neo4j to confirm the data loads correctly. I used the built-in visualizer in the Neo4j web interface to spot check the validity of the graphs that were created based on the generated csv input.

The next task I worked on was creating functions to generate csv files specifically for importing into MASS. This mainly involved using a pipe “ | ” as the delimiter between fields, and grouping fields according to how Lilian’s graph builder works. Lilian Cao is a student in this research group with a focus on updating the implementation of MASS’ graph database application. After converting the Amazon data into this format, I moved on to adding this same functionality for the Twitch and Cryptocurrency datasets. I then pulled the latest code from Lilian’s QueryDB branch and tested importing these various datasets into MASS. As a starting point, I am limiting the number of nodes to 8,000 per a former student’s research that MASS can handle around 8,000 nodes [5]. To reiterate the overall structure of the graphs, each of my generated datasets held 8,000 nodes and different numbers of edges (Amazon: ~59,000 edges, Twitch: ~9,000 edges, Cryptocurrency: ~20,000 edges).

After completing these functions, I started working on my random graph generator design. I decided to use a factory design pattern for the generator since it makes the code less coupled, allowing us to create graph objects without tying the code to specific classes. It is also easier to extend should a new graph dataset type be added to the generator. The initial design is detailed in Figure 3.1 below:


RandomGraphFactory
+createGraph(): RandomGraph
+createRandomGraph(): RandomGraph
<<interface>>
RandomGraph
+generateGraph()
Products: Recommendation Data
+generateGraph()
Social Media: Social Media Network Data
+generateGraph()
Transactions: Financial Data
+generateGraph()
ProductsFactory
+createRandomGraph(): RandomGraph
SocialMediaFactory
+createRandomGraph(): RandomGraph
TransactionsFactory
+createRandomGraph(): RandomGraph
Figure 3.1: Random graph generator initial design

The following describes the pairings between the graph objects I am creating compared to the practical data I am using as the basis for my random graph generator: Products uses Amazon data, Social Media uses Twitch data, Transactions uses Cryptocurrency data. My implementation uses the practical datasets as the foundation for the values that will be written into the generated csvs, i.e. the fields and values generated by the random graph generator are pulled from the practical datasets. The random piece comes from rearranging of the relationships between nodes, i.e. the edges that connect the nodes. Originally, the edges that connect a pair of nodes were created at random. After speaking with Professor Fukuda, maintaining the degree of distribution among edges is an important consideration for graph generation. Professor Kim shared her lecture slides regarding Network Motif, subgraph patterns that occur frequently and uniquely in a network [6]. The algorithm [6] for maintain this degree distribution includes the following steps:

1. Get degree sequences
2. Each vertex is repeated by the number of its degree
3. Randomly choose two vertices to make an edge

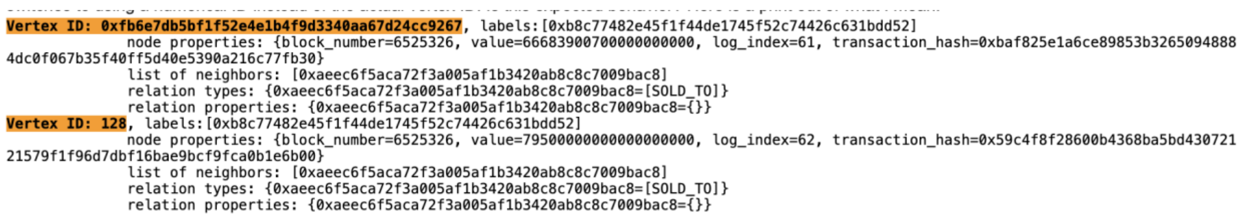
I implemented this degree distribution algorithm as part of my random graph generator for the Products graph, and for the Social Media graph.

For the benchmarking piece, I discussed my future plan with Professor Fukuda, and I will be creating a program for benchmarking against MASS. The standard queries for this program will include Create, Delete, and Match. For Neo4j, I will be creating a manual with a list of commands to run in the Neo4j web interface, since the interface has a helpful visualization capability, and the interface displays the time it takes to run each query.

I also began evaluating TigerGraph as an alternative to ArangoDB. A major appeal of TigerGraph is it's scalability and performance[7]. Its distributed architecture supports fast loading speeds for both graph creation and traversal. However, I think the licensing of TigerGraph is going to pose a problem. There is a free developer license that lasts on a month-to-month basis, that needs to be renewed with an account manager at the end of the trial period. This type of licensing may not be appropriate for research that will span multiple months, as the renewal is left to the account manager's discretion.

4. Results

When loading the data into MASS, I uncovered a bug that I reported to Lilian. After the 128th node, MASS does not print the vertex ID, it instead prints a numerical value that I believe to be assigned by MASS as shown in Figure 4.1:



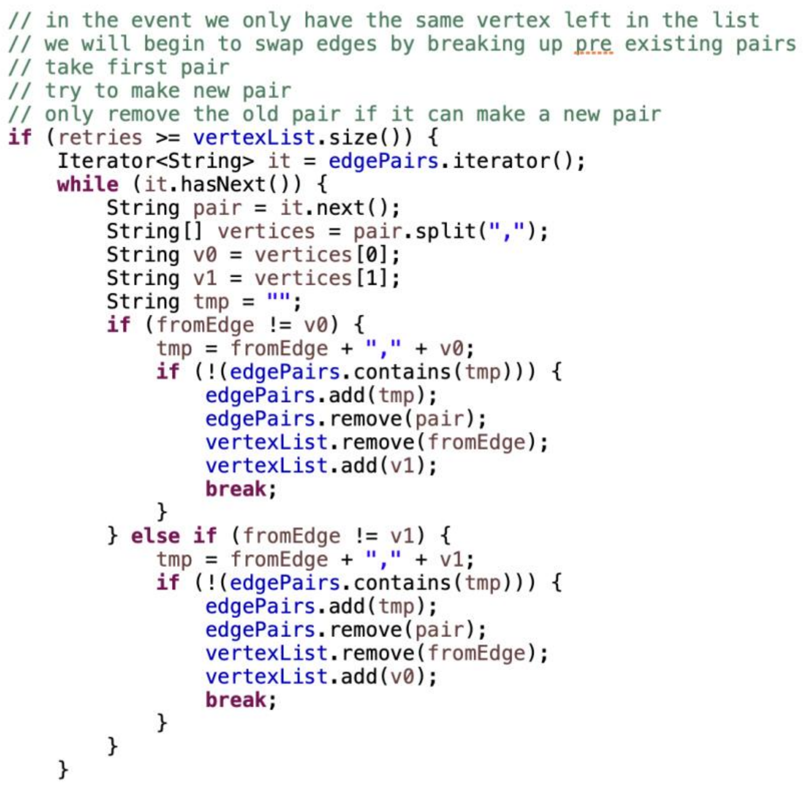
```
Vertex ID: 0xfb6e7db5bf1f52e4e1b4f9d3340aa67d24cc9267, labels:[0xb8c77482e45f1f44de1745f52c74426c631bdd52]
                node properties: {block_number=6525326, value=66683900700000000000, log_index=61, transaction_hash=0xbaf825e1a6ce89853b3265094888
4dc0f067b35f40ff5d40e5390a216c77fb30}
                list of neighbors: [0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8]
                relation types: {0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8=[SOLD_TO]}
                relation properties: {0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8={}}
Vertex ID: 128, labels:[0xb8c77482e45f1f44de1745f52c74426c631bdd52]
                node properties: {block_number=6525326, value=79500000000000000000, log_index=62, transaction_hash=0x59c4f8f28600b4368ba5bd430721
21579f1f96d7dbf16bae9bcf9fca0b1e6b00}
                list of neighbors: [0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8]
                relation types: {0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8=[SOLD_TO]}
                relation properties: {0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8={}}
```

Figure 4.1: Vertex ID displays numerical value instead of ID

A challenge I experienced was implementing the degree distribution algorithm from Professor Kim. With the Twitch dataset, because the degree distribution for more popular users is significantly higher than others, when we randomly pair vertices together for edge creation, we run the risk of being left with a list of only one vertex left, ex: if we have a graph that looks like this:

Vertex ID	1	2	3
Degrees	2	1	1

After step 2 where each vertex is repeated based on the number of degrees, we have a list that looks as such: [1,1,2,3]. If 2 and 3 are randomly paired to create an edge, we are only left with [1,1] to create an edge pair. A node should not be linked to itself. To combat this, I added in logic that runs through the list of edge pairings so far, and tests to see if a new acceptable edge pair can be created with the remaining unpaired vertices. If a new pair can be created, that edge is broken apart, and the new edge is added to my list of edge pairings. A snippet of the code is shown below in Figure 4.2:



```
// in the event we only have the same vertex left in the list
// we will begin to swap edges by breaking up pre existing pairs
// take first pair
// try to make new pair
// only remove the old pair if it can make a new pair
if (retries >= vertexList.size()) {
    Iterator<String> it = edgePairs.iterator();
    while (it.hasNext()) {
        String pair = it.next();
        String[] vertices = pair.split(",");
        String v0 = vertices[0];
        String v1 = vertices[1];
        String tmp = "";
        if (fromEdge != v0) {
            tmp = fromEdge + "," + v0;
            if (!(edgePairs.contains(tmp))) {
                edgePairs.add(tmp);
                edgePairs.remove(pair);
                vertexList.remove(fromEdge);
                vertexList.add(v1);
                break;
            }
        } else if (fromEdge != v1) {
            tmp = fromEdge + "," + v1;
            if (!(edgePairs.contains(tmp))) {
                edgePairs.add(tmp);
                edgePairs.remove(pair);
                vertexList.remove(fromEdge);
                vertexList.add(v0);
                break;
            }
        }
    }
```

Figure 4.2: Code snippet detailing swapping implementation

5. Next Quarter's Plan

Spring 2024

Week	Tasks
March Week 3 & 4	Finish Random Graph Generator Create standard queries for benchmarking in MASS • Create • Match • Delete Research related works in the domain of benchmarking and graph data
April Week 1 & 2	Create scripts / manual for benchmarking against Neo4j and ArangoDB
April Week 3 & 4	Benchmark MASS performance against Neo4j and ArangoDB using variously sized graphs: • 8000 vertices • 10000 vertices • 20000 vertices
May Week 1 & 2	Whitepaper and final defense prep
May Week 3 & 4	Whitepaper and final defense prep
June Week 1	Incorporate feedback into whitepaper

6. Summary

This quarter I was able to complete most of the development for graph generation. This data is going to serve as the benchmarking data for performance testing and database evaluation. By first creating functions that read and generate data from the practical datasets, I was able to leverage this logic for my random graph generator which made the conversion more efficient than having to start from scratch.

One piece I struggled with was being able to maintain degree distribution within the Social Media graph. My original implementation of the degree distribution algorithm mentioned earlier worked with the Amazon data, as the average degree per edge was not high. However, with the Twitch data, because there are very popular users who may have many edges, there is a higher risk of ending up with a list of unmatched vertices of just one particular user. After reevaluating my original implementation, I needed to update my swapping logic to only break up an edge pair if the new resulting edge pair was acceptable.

7. References

1. “MASS: A parallelizing library for multi-agent spatial simulation.” [Online]. Available: http://depts.washington.edu/dslab/MASS/
2. “An Agent-based Graph Database” White Paper, Accessed on: June 30, 2023. [Email]. Available: via Professor Munehiro Fukuda.
3. Leskovec, J., & Krevl, A. (2014, June). SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data
4. “Recommendation on Live-Streaming Platforms: Dynamic Availability and Repeat Consumption”
   Jérémie Rappaz, Julian McAuley and Karl Aberer
   RecSys, 2021
5. “Agent-based Graph Applications in MASS Java and Comparison with Spark” Term Paper. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/CarolineTsui_whitepaper.pdf
6. “Network Motif” Lecture Slides, Accessed on: February 8, 2024. [Email]. Available: via Professor Wooyoung Kim.
7. “TigerGraph Recognized for the First Time in the 2022 Gartner® Magic Quadrant™ for Cloud Database Management Systems” [Online]. Available: https://www.tigergraph.com/press-article/tigergraph-recognized-for-the-first-time-in-the-2022-gartner-magic-quadrant-for-cloud-database-management-systems-2/