DYNAMIC GRAPH CONSTRUCTION AND MAINTENANCE

Justin Gilroy

A report submitted in partial fulfillment of the
requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington
2020

Project Committee:
Dr. Munehiro Fukuda, Chair
Dr. Min Chen, Committee Member
Dr. Erika Parsons, Committee Member

Abstract

DYNAMIC GRAPH CONSTRUCTION AND MAINTENANCE

Justin Gilroy

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda
Computing and Software Systems

Agent-Based Modeling (ABM) is a method of solving biological and similarly structured problems by simulating the interaction of entities with the notion of Agents. MASS - Multi-Agent Spatial Simulation is a system developed by the Distributed Systems Laboratory at UWB for applying ABM to problems with the addition of distributed computing to expand beyond what is capable of being simulated on a single node. MASS today is targeted at problems that exist in multi-dimensional space thus requiring graph space problems to be mapped by the user. We have responded to this problem by enhancing MASS with: 1) graph input file formats, 2) Dynamic graph construction and modification features, and 3) graph visualization through integration with Cytoscape. This paper describes the goals and implementation of adding graph targeting features to the MASS Java to allow non-professional developers to solve and visualize these large and compute intensive problems efficiently.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Background and Motivation

Agent-based modeling (ABM) is a form of simulation that attempts to mimic real-life scenarios by representing individual actors within a system [1]. In ABM the individual actors can traverse the system or data and act in a predetermined manner. These simulations can be used to solve optimization problems using biological simulations such as Ant-Colony Optimization [2][3].

Agent-based modeling can be extended to perform such modeling on data that is in the form of graphs. By utilizing ABM and a biological algorithm like Ant-Colony Optimization, even graph problems such as finding the shortest path between two vertices can be solved with ABM [2].

A key component of existing work on ABM at UWB is the MASS (multi-agent spatial simulation) library. MASS is a system for distributing a very large dataset, one that cannot fit within the resources of a single computing node, across a computing cluster [4]. MASS uses mobile code called “Agents” to process the data-set and thus traverse the data autonomously.

This distribution and autonomy gives MASS the ability to have agents analyze huge amounts of data that would not fit inside of a single node’s physical resource restrictions.

1.2. Goals

This project has three goals to aid in the development of graph problems with MASS: (1) Various graph inputs support, (2) Programmability improvement in agent-based graph solving, and (3) Distributed graph visualization

1. **Various graph Inputs support**: The project expands MASS to support a variety of input formats to increase the utility of graph support.

2. **Programmability improvement in agent-based-graph solving**: We aim to

increase the programmability of MASS for application developers, particularly as it relates to graph problems. By implementing graph maintenance support directly into the MASS library, we intend to shift the burden of graph programming into the MASS library so that users can focus on their problem domain.

3. **Distributed graph visualization**: We intend to allow users of MASS to visualize their graph network. We will use Cytoscape as an external visualizing tool that is popular among bioinformaticians.

1.3. Project Overview

In order to achieve these project goals, we will modify the MASS library and create two plugins for Cytoscape. Modifications to the MASS library will facilitate (1) Graph Input Formats by implementing two input formats: (a) HIPPIE Tab and (b) MATSim XML, and (2) MASS Graph Maintenance by implementing (a) the ability to add and remove vertices from a graph and (b) the ability to add and remove edges from a graph. The final goal (3) Visualization will be implemented in two Cytoscape plugins: (a) export-network plugin, and (b) import-network plugin; both plugins will require supporting code in MASS to send and receive a graph to and from MASS.

Chapter 2. Related Work

This section is an overview of three existing systems: Repast Simphony, FLAME, and Cytoscape and the inputs, programmability, and visualization capabilities of each package. For each package we evaluate their features as they relate to graph analysis and for Cytoscape in particular we evaluate its graph visualization capability.

2.1. Repast Simphony

Repast Simphony is the latest iteration of an Agent-Based Modeling approach to Complex Adaptive Systems that originated at the University of Chicago in 2000 [5]. The latest incarnation of Repast Simphony was released on Github September 30, 2019 [6]. Repast's data model for graphs is based on a Projection of the data space into a logical view of the data for the user to work with [7]. Modeling a graph with Repast Simphony

requires the user of the library to adapt their graph problem to a strictly 2-Dimensional space within Repast Simphony. Repast supports creating and modifying graphs with the methods addVertex and addEdge before simulation

While Repast does support loading a graph from a file on disk, it is limited to two formats: UCINet's DL format, and Excel format [7][8]. Furthermore, the Excel format requires adherence to Excel files based on UCINet's format further limiting the utility of the input capabilities of Repast Simphony [7].

While Repast does have agents to explore shared data, the agent implementation in Repast does not support temporary suspension of activities [7]. Agent actions in Repast can be controlled by a schedule or triggered by access to a specified variable [9]. Finally, Repast requires that agents are created at the same time as the graph itself is created preventing Repast from supporting the goal of our project of Graph Maintenance [7].

Additionally Repast does not support parallel I/O that is required for performing complex biological network analysis quickly nor can agents migrate asynchronously [7].

Of particular interest in Repast is not only its long history but integration with the Eclipse Project's Integrated Development Environment [7]. Integration with Eclipse sets Repast Simphony apart from the other frameworks described because it allows users to perform actions related to creating, running, and analyzing a simulation view a graphical user interface.

Repast offers visualization of a simulation space without the ability to modify the simulation space [7]. While Repast supports a 'Runtime GUI' [7], setting up a 2D Display of a Projection in Repast involves four pages of configuration dialogs to visualize a network [11].

2.2. FLAME

One agent-based modeling framework that we considered in the for analyzing graph data was FLAME which stands for Flexible Large-scale Agent-based Modeling Environment [12]. One feature of FLAME that would be of interest to us is the framework's ability to simulate a large number of agents which is indicated by using FLAME to run a simulation with 10^6 agents for a European transportation simulation

[12].

While a large number of agents can be seen as beneficial for analyzing a large dataset, FLAME does not have a data model that is separate from the agent construct. In this paradigm, each agent requires the entire dataset to be held within. This is contrary to our goals of analyzing data that will not fit within a single node.

As FLAME does not have a separated data-model, neither does it have any representation of a predefined space outside of the agents. For this reason, FLAME does not support any input format outside of its specialized configuration files called the model [13].

Another consideration that steered us clear of FLAME is the programmability of FLAME. One of the goals is to provide non-computer scientists with the tools to analyze large graphs with an approachable toolset.

Development in FLAME requires not only specifying to the framework a specification for the simulation in an xml file but also providing the definitions for functions utilized within the specification in a second file in the C language [12]. The functional definitions, while in a somewhat simplified and narrowed form of the C language require the user to learn a domain specific ecosystem of functions for memory allocation, arrays, variable definitions, et cetera [12].

2.3. Cytoscape

In addition to adding the graphing capability to the MASS library, this project intends to provide a visualization of the graphs being analyzed and simulated. To this end we explored the visualization package Cytoscape [14].

In contrast to Repast and FLAME, Cytoscape supports a number of file input formats: SIF, NNF, GML, XGMML, CYS (Cytoscape format), Excel, and Delimited Text [14]. Not only does Cytoscape support these file based formats, it also supports importing directly from a number of online databases such as STRING/STITCH, and WikiPathways [14].

Cytoscape also supports as a core feature manipulation of graphs by allowing adding and removing edges and vertices [14]. Despite its extensive graph support, Cytoscape is a desktop application and, as such, is limited to manipulating graphs of sizes that are capable of fitting into the resources of the computer it is run on. This also limits the

package's ability to do deep analysis and simulation on a graph which is not the intention of Cytoscape [14].

2.4. Summary

In order to achieve the goals of this project we require more than what is offered by any one of the above described projects: Repast Simphony, FLAME, or Cytoscape. More specifically, we should have the following features in ABM:

The ability to import a graph from a file is only present in Cytoscape but is limited by the resources of the computer it is run on both for visualization and analysis.

We require the ability to dynamically alter the shape of a graph for analysis and, by extension, we require the ability to control, create, and destroy agents on a dynamically maintained graph.

Our solution and the premise of this project is to achieve our stated goals: (1) Graph Input Formats, (2) Graph Maintenance, and (3) Graph Visualization by incorporating these features into MASS.

Chapter 3. Graph Support in MASS

3.1. MASS Library

MASS approaches Agent-Based modeling with two conceptual entities: an Agent, representing an actor in a simulation, and a Place, representing a logical portion of the simulation space [4]. In MASS, the data is mapped to a multi-dimensional array of Place entities that represent the simulation space for analysis. To simulate actors in this space users of MASS can create Agent entities which can then autonomously traverse the logical space represented by Place objects.

Parallelization in MASS is facilitated by distributing portions of the multidimensional Place array across nodes and further splitting a node's portion of the space between multiple threads on each node. This distribution of the data-set across multiple computing nodes gives MASS the ability to simulate large logical spaces beyond what can be represented on a single node.

Agents can then seamlessly migrate between the logical Place entities regardless of the

node or thread they are associated with. Figure 3.1 is a graphical representation of the distribution of Place entities across a cluster of 3 computing nodes identified as mnode0.uwb.edu, mnode1.uwb.edu, and mnode2.uwb.edu [4]. Each node in Figure 3.1 can be seen to have multiple, 4 in this figure, threads of execution for manipulating the places and agents. Communication between each node is carried out through a basic networking socket.

Above the Place array visualization in Figure 3.1 we can see that each node contains a collection of the Agent entities that are associated with the Place entities the node is responsible for.

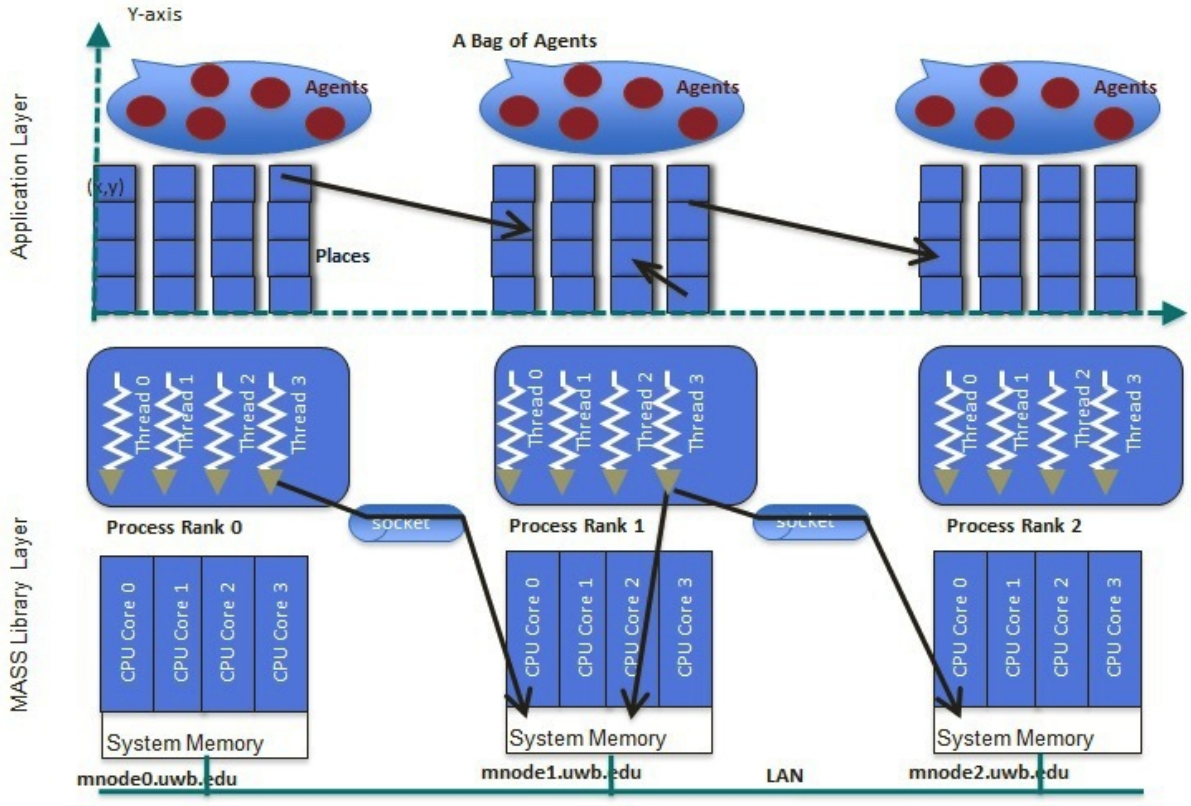


Figure 3.1: MASS library data model [4]

Also illustrated well in Figure 3.1 is the multi-dimensional nature of the Place entities within the MASS data space. This works well for addressing space of a multi-dimensional nature but leaves the graph to array mapping to the users of the library.

3.2. Deliverables

This section describes how the graph mapping is brought into the MASS through our previously stated goals of adding support for graph input files, graph maintenance, and the visualization of graphs beginning with an overview of the components to be developed for the realization of these goals.

3.2.1. Graph Input Formats

The first goal of this project is to add to MASS the ability to import graphs via two input file formats: MATSim XML, an xml-based network format used in the MATSim traffic simulation platform [15], and HIPPIE Tab, a tab-separated value format used as a common data format for document protein-protein interaction networks [7].

a. MATSim XML Input Format

The first of the two formats added to MASS, MatSim is an XML format that was created for traffic simulation framework: Multi-Agent Traffic Simulation [15]. Though MATSim utilizes a variety of XML files for describing a simulation, we focused on a single file that defines the structure of the simulation network. Other files represent the vehicles and other properties that are not immediately necessary for the goals of this project.

The MATSim network format is scarcely documented so for the purpose of this project we relied on sample files as a reference for retrieving the format of the network file [16]. The MATSim network file is an edge list format with two second-level lists: nodes and links [16].

The nodes list contains a list of all vertices of the network with a number of attributes. The attributes include a 1-based integral index and x,y coordinates which we ignored for this project.

The links list is the proper edge list which contains links, or edges, between the previously defined nodes based on the node id. The link elements contain two attributes to encode the source vertex and destination vertex: from, and to respectively.

Further description and an example of MATSim network format can be found in Appendix B.

b. HIPPIE Tab Input Format

Like MATSim, finding a reference for this resulted in using the data file itself as a

reference rather than a detailed specification [16]. In the case of HIPPIE, the website contains a large dataset called HIPPIE_CURRENT, for the current compilation of protein interactions [17]. This reference illustrated the HIPPIE Tab format enough to achieve the input goals of this project.

HIPPIE Tab, as the name implies, is a tab-separated value format similar to comma-separated values, CSV. The HIPPIE file format describes a protein network in the form of an edge list. Each line in a HIPPIE Tab file represents a protein-protein interaction which together represents an edge in the protein graph.

An example line from the HIPPIE_CURRENT dataset is seen in Figure 3.2. Each line has a source protein sequence followed by a numerical id. The source is followed similarly by the neighboring protein sequence and numerical id. The next field represents the interaction attribute, or weight. The final field is a comma-separated metadata field that varies for each edge.

```
AL1A1_HUMAN 216 AL1A1_HUMAN 216 0.76 experiments:in vivo,Two-hybrid;pmids:12081471,16189514,25416956;sources:HPRD,BioGRID,IntAct,MINT,I2D,Rual05
NEB1_HUMAN 55607 ACTG_HUMAN 71 0.65 experiments:in vitro,in vivo;pmids:9362513,12052877;sources:HPRD
SRGN_HUMAN 5552 CD44_HUMAN 960 0.63 experiments:in vivo;pmids:9334256,16189514,16713569;sources:HPRD,I2D,Rual05,Lim06
PAK1_HUMAN 5058 ERBB2_HUMAN 2064 0.73 experiments:in vivo,Affinity Capture-Western,affinity chromatography
```

Figure 3.2: A single protein-protein edge from HIPPIE_CURRENT dataset

One important note to the HIPPIE Tab format is that the primary vertex key is a string. Furthermore, the vertex ids are non-sequential varying from very small to quite large. The small sample HIPPIE snippet contains 22 protein interactions with source protein ids ranging from 216 to 259,197 [18].

Further description and an example of HIPPIE Tab format can be found in Appendix A.

3.2.2. MASS Graph Maintenance Features

The Graph Maintenance features proposed for my project represent a significant portion of the work proposed to extend the MASS library in this project. In this context we are defining graph maintenance as the ability to create a graph without vertices. From this perspective we ‘maintain’ the graph by adding and removing vertices and edges.

a. addVertex/removeVertex

The core feature of graph maintenance is the ability to add vertices to an empty graph.

To achieve this we will create an ‘blank’ constructor for the new GraphPlaces type in MASS that will allow initialization of MASS without importing a network from a file.

The user of the upgraded version of MASS with Graph Maintenance features will be able to call the addVertex method on an instance of GraphPlaces with the only parameter being the id to associate with the new vertex.

The removeVertex method, mirroring addVertex, will also only require a single parameter to specify the id of which vertex to remove. One key difference between adding and removing is that during removal, there are likely references to the vertex being removed in the form of neighbors. MASS will remove these neighbor references from other vertices in the graph during the processing of removeVertex. Complete specification of the MASS Graph interface can be found in Appendix H.

b. addEdge/removeEdge

In the case of maintenance of edges, the user intent is to modify the relationship between vertices already existing in the graph. The user can create edges between any two vertices that exist in the graph with an associated attribute representing the weight of the relationship of the edge.

Edges created in a graph are directed meaning that the source and destination vertices are indicated when the edge is created or removed.

3.2.3. Cytoscape Integration

Integrating MASS with Cytoscape requires two key interactions between the two systems: pulling the graph from MASS into Cytoscape, and sending the graph from Cytoscape to MASS.

Cytoscape allows developers to add functionality to the main application via a plugin architecture which allows such features to be developed independently of the main application and managed at startup. Based on the two interactions required for this project we will develop two plugins for integration with Cytoscape.

The first plugin will be called import-network and serves the first interaction of pulling the in-memory graph from MASS. The plugin will pull the graph from MASS as a simplified data model for network transport and then be rebuilt inside of Cytoscape into a native CyNetwork with a combination of CyNode and CyEdge entities.

The second plugin to be developed will be called export-network. This plugin will address the second interaction with MASS by serializing the currently selected Cytoscape CyNetwork into the network GraphModel for sending to MASS.
Both the import and export network plugins require additions to MASS to support such communication. To facilitate communication between Cytoscape and MASS we use a plain socket connection between Cytoscape and the control node.
To communicate the graph data itself, we intend to create a lightweight representation of the graph with only essential information such as the vertex ids and their neighbors. Reducing the footprint of the data model is essential for communicating across a network socket.
Actions performed on MASS will be determined by one of currently 2 strings to be sent by Cytoscape:

- getGraph - retrieve the in-memory graph from MASS
- setGraph - sends the current CyNetwork to MASS

MASS is then able to determine the processing requirements of the request based on the initial request string. For getGraph, MASS sends the complete GraphModel back to Cytoscape through the socket. When MASS receives the setGraph request it then knows to read the socket for a GraphModel object to be constructed on the cluster.

3.3. Technical Challenges

3.3.1. Graph Input Formats

Implementing the new input formats at the beginning of this project posed a more substantial challenge than anticipated. In particular, the two formats were not as readily documented as I expected with my experience implementing parsers for other formats. Eventually it became clear that deriving the format from a sample file would be most efficient [10]. Sample files for both HIPPIE and MATSim were readily available [10][16].
A second challenge came from further examination of the HIPPIE input format [10]. This format surfaced a significant deficiency in the existing pattern of mapping vertices to ordinals compared to MATSim which uses sequential vertex ids beginning from 1 [16].

In MATSim, the vertex IDs are provided as sequential integers. The existing file importing proof-of-concept in the MASS library for importing a graph via a CSV file gave implicit ids to the vertices based on their ordination within the input file: the first was labeled 0, the second 1, and so on as we typically do with base 0. For implementing the MATSim input format, we were able to extend this model because the MATSim format allows easy manipulation of the vertex ids with a simple offset of 1.

The HIPPIE format, in contrast, contains 2 possible attributes for each vertex: a String, and a non-sequential integer. The integer attribute represents an incomplete range of integers (sampled from ~200 - > 100,000 seen in the hippie_current.txt dataset [18]).

The problem to be seen here is that the MASS library does not have a hashing method that would allow mapping a given vertex id, string or integer, consistently across a cluster of MASS cluster-computing nodes. The problem is compounded by our goal of supporting dynamic modification of the graph so a pre-calculated complete hash was not an option.

The solution options clear to us was to implement a distributed map in MASS to allow consistent hashing across the cluster system or to utilize an existing library with such distributed map capability included. With another group member working on a project that incorporated a Java library called Hazelcast that includes a distributed map and the benefit of re-using well tested code, we elected to use the existing library for this project [20].

The mapping solution we came up with maps an object key to an integer that represents the global index for the place associated with a given key. This allows users to reference the vertices in their data as expected: the logical name of the vertex. MASS can then use the logical name of the vertex as a key into the distributed map to retrieve the global index of the place that represents the vertex.

3.3.2. MASS Graph Maintenance

Another significant design challenge we had to overcome was to create a dynamic structure for holding vertices on the cluster. The existing MASS structure is initialized at the given size and distributed across the cluster evenly based on the input size. This distribution is not possible when dealing with dynamic changes of the data.

Vertex Mapping

Considering vertex mapping, the correlation between a user application's logical vertex ID and the internal memory location of a vertex within the cluster system, posed a significant challenge as we began implementing graph maintenance features in the MASS library. To support a dynamic feature we could not know ahead of time which vertices would go on which node and how to find the vertices after they had been sent off to a node to be initialized. Our solution was to create a second data-structure within the new graph model to allow a modified paradigm within affecting existing applications.

To address the problem of dynamic directly-addressable vertices within a cluster system, we devised a layered duplicate of the existing model used in MASS. By duplicate we specifically mean the size of the layer space. While supporting the dynamic expanding nature of this graph we intend to support, we must also still be able to consistently locate a place across the cluster.

Where MASS has a strictly enforced size based on the input, we expanded the initial size to be the size of layers within the cluster. In our graph infrastructure we maintained the size but duplicated this into layers to allow virtually unlimited depth for dynamic modification of the graph. This strict adherence to the sizing of the data-structure is what would allow us to support dynamic sizing as well as direct addressing of individual places.

With the data structure defined, we still needed a way to address the vertices consistently from any node. For this we devised a sort of hashing algorithm a globally unique integer to the specific vertex location within the cluster. This was achievable thanks to creating layers of a consistent size. From this size we can allocate a stripe to each node and further extend the stripes to any number of layers.

The proposed graph data model built on top of the existing MASS multidimensional model is illustrated in Figure 3.3. The figure breaks down a MASS cluster of 4 nodes. In this proposed layered model, layer 0 is the existing MASS dimensional model. Layers starting from 1 are within the collection maintained within the graph implementation. These layers are represented by Graph Layer 1 and Graph Layer 2 in Figure 3.3.

Each graph layer imitates the MASS layer's size and allocates a fraction of the size to each node in the cluster. In other words, if we map 100 vertices to a cluster system containing 4 computing nodes, each node would be allocated one-quarter of the size,

100 Place entities, as their portion of the data set. In this scenario each node would be allocated 25 Place entities. Therefore, node 0 would be allocated indices 0-24, node 1 indices 25-49, node 2 indices 50-74, and node 3 indices 75-99.

Maintaining this 100 Place scenario, the Place indices 0 - 99 represent the MASS layer, layer 0, in Figure 3.3 which MASS manages with the existing multi-dimensional implementation. In the new model we intend to repeat this association in a round-robin fashion where the next allocation of 25 Place entities, in this particular scenario, would be associated to node 0 at layer 1. This layer, layer 1, also represents the first layer of the graph data model - Graph Layer 1 in Figure 3.3.

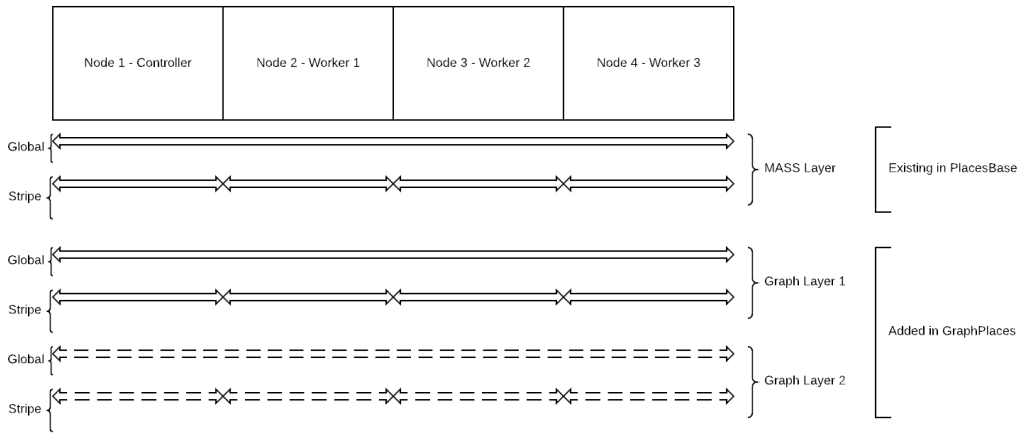


Figure 3.3: New data model enabling support for dynamic places

This specification allows maintaining a linear index for each Place that can be located in MASS with an addressing algorithm defined as follows:

A. nodeCount: the number of nodes in the cluster
B. size: the initial size specified in the GraphPlaces constructor
C. stripeSize: fraction of the initial size delegated per node
D. globalIndex: 0-based global index retrieved from distributed map
E. layerIndex: 0-based index of the layer within graph data
F. localIndex: 0-based index of the place within the layer collection

stripeSize = size / nodeCount

layerIndex = globalIndex % size

localIndex = globalIndex % stripeSize

Using this addressing algorithm we can store a single integer in the distributed map to allow mapping an arbitrary vertex id to a unique place in MASS that is directly addressable. The layering allows the use to expand the graph well beyond the initial size.

3.3.3. Cytoscape Integration

Integrating Cytoscape and MASS, posed a significant challenge in this project due to the fact that we are adding a completely new feature to MASS with no existing code. This meant we must design all interactions from scratch. Furthermore, we are only limited by the programming model that Cytoscape allows.

The larger challenge of integrating MASS and Cytoscape can be further broken down into two task: a) Model to Visualization: Mapping our data-model to a visual representation within Cytoscape, and b) Interaction Model: The method in which MASS and Cytoscape communicate.

a. Model to Visualization

Speaking of the limitations of Cytoscape, perhaps the most apparent is that we must map our vertices and edges to the Cytoscape data model which is represented by putting vertices and edges into separate tables. Attributes of each vertex or edge are then encoded into columns of the table.

The developer is given the ability to add custom columns to the tables but are limited to primitive types and lists of primitive types. For the goals of this project, the default columns on both the vertex (node in Cytoscape), and edge table were adequate.

Another challenge we faced was the styling of vertices as they appear in Cytoscape. The default visual representation of vertices was not entirely representative of a unique vertex in a network so a simple styling transform was applied to incoming graph data to assign names and interactions to the vertices.

b. Interaction Model

In addition to mapping the network graph model to the Cytoscape data model, we required a way for Cytoscape to invoke commands in the MASS cluster and elicit responses. For this problem we elected to implement a simple remote-procedure call interface where Cytoscape could send a request to MASS as a simple string and MASS can parse the request and respond accordingly.

An additional benefit of this RPC interface is that it is easily expandable by adding additional commands with associated parsers.

3.4. Implementation

Implementing the specified features described in the above described deliverables comes in the form of code changes to the MASS library, changes to the MASS applications [21] for testing and verification, and creation of two plugins for integrating with Cytoscape. These features were developed iteratively meaning each feature was not necessarily completed in order, however, at a high level the features were completed as follows:

1. Graph Input formats MATSim and HIPPIE were added to the library first as a way to effectively validate the data structures to be implemented to support graphs in MASS.
2. MASS Graph Maintenance features which allow dynamic modification of the graph structure.
3. Cytoscape Integration
    a. Import graph from MASS into Cytoscape
    b. Export a network from Cytoscape to MASS

3.4.1. Graph Input Formats

Each input format has a unique initialization function in MASS that will parse the input and populate the cluster system with the container vertex data. MASS uses the number of vertices contained in the input file to set the internal size so that the user is not required to do so.

To allow for vertices with arbitrary values as IDs, MASS maintains a mapping between

user-data vertex IDs to a linear index. The map allows an arbitrary but homogeneous structure of keys and values.

3.4.2. MASS Graph Maintenance

The MASS graph maintenance features represent a large portion of this project. As such, while they are primarily implemented as features of the MASS library, they serve as an interface for users of the library as well as for the visualization enhancements added via a Cytoscape plugin discussed in the next section.

At a high level, graph maintenance in MASS can be simplified to the adding and removing of vertices and edges.

a. Data Model

The existing implementation, each computing node in the cluster system is responsible for maintaining an array of Place entities representing a fraction of the total data set.

To support a graph maintenance in MASS we expanded on the existing model. Based on the initial global size, specified in the constructor arguments, or from the input file, we use the same distribution per node as a reference for stripeSize. See section 3.3.2 for a detailed explanation of the data model to be implemented.

b. Vertices

The primary problem to solve with adding vertices is solved with the implementation of mapping arbitrary keys to linear indices. By using a distributed map for mapping vertex attributes to a global index and an addressing scheme for mapping these global indices we are able to distribute the data across a cluster system of computing nodes.

Modification of vertices and edges is made possible by this mapping. MASS is able to determine the vertex representation based on the addressing model and insert or modify the appropriate data structure.

3.4.3. Cytoscape Integration

When planning the work for this project we realized there were two primary interactions that we hoped to address: sending a graph created in Cytoscape to MASS, and retrieving a graph that is currently in-memory in a MASS cluster to visualize in Cytoscape.

To achieve these goals we implemented the infrastructure for such communication in MASS via a new thread to listen for requests from Cytoscape.

In order to facilitate this bi-directional communication channel between MASS and Cytoscape we also required a data model that would be lightweight for transferring over the network. This data model also serves as a bridge between systems that have different internal representations for their graphs. This communication is depicted in Figure 3.4

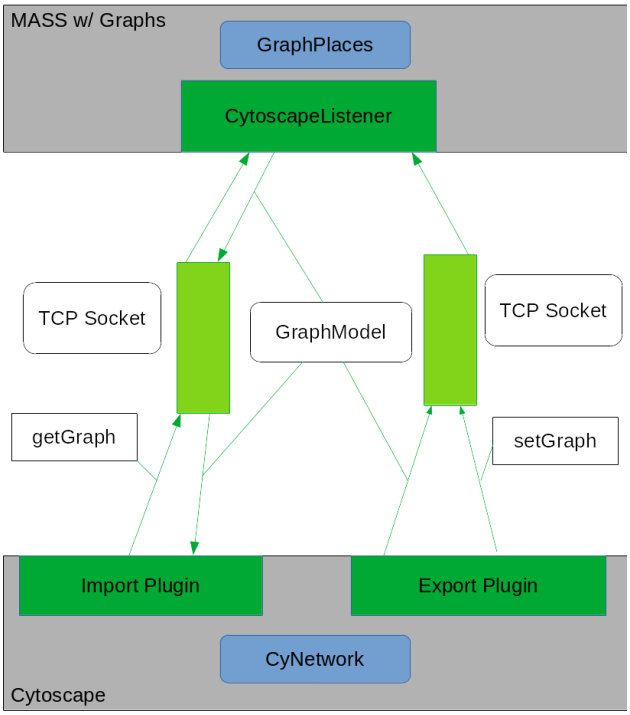


Figure 3.4: Communication between MASS and Cytoscape

a. CytoscapeListener

We have implemented a class called CytoscapeListener in MASS to listen for and process requests from Cytoscape. This class appears in the MASS portion of Figure 3.4.

c. Cytoscape Plugins

Modifications to Cytoscape are made possible via OSGI plugin architecture built into Cytoscape [22][23]. OSGI, or Open Service Gateway Initiative, is a group that maintains

the OSGI standard of developing modularized Java components that can be loaded at runtime [22]. The result of this project includes two such plugins representing each direction of the communication channel: Import Network, and Export network.

A Cytoscape plugin is principally built out of three required classes: an activator, a task factory, and a task. The purpose of each class for the Import Network and Export Network plugins are described in Table 3.1.

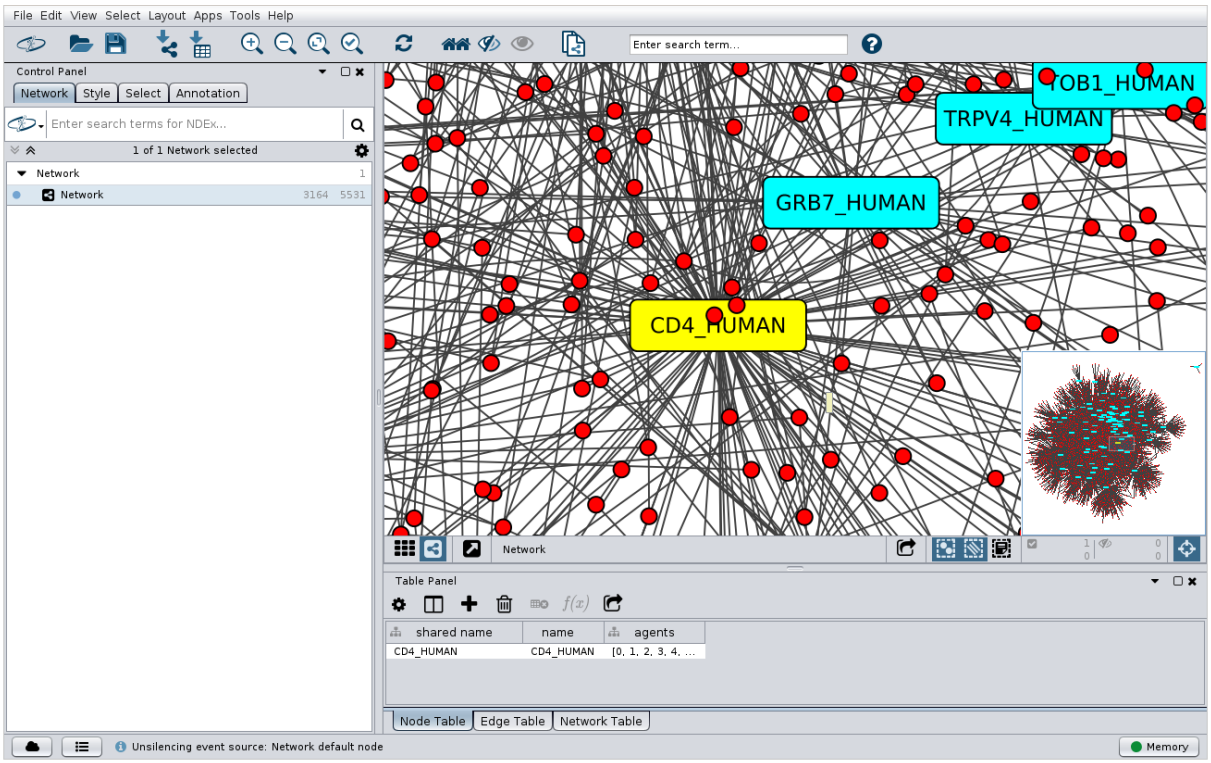
Table 3.1: Breakdown of Cytoscape plugin classes and responsibilities

Class	**Purpose**	**Plugin**	
		Import Network	*Export Network*
Activator	- Entry point of a plugin - Assigns services to factory and associated the factory a menu option	Creates MASS > Import Network menu option	Creates MASS > Export Network menu option
Factory	- Called when the user selects the created menu option - Creates an instance of Task class to process a request	- Pass required Cytoscape APIs to task	- Pass required Cytoscape APIs to task
Task	Actual plugin logic	- Request GraphModel from MASS - Deserializes GraphModel response into CyNetwork	- Serializes CyNetwork in GraphModel - Sends GraphModel to MASS

i. Cytoscape to MASS

The Import Network plugin we implemented in Cytoscape retrieves a graph from MASS for visualizing in Cytoscape. By selecting the Import Network menu option, Cytoscape sends a request to MASS to retrieve its in-memory graph. Our plugin then deserializes the MASS response into a Cytoscape representation of the graph: a CyNetwork.

To achieve a simple and consistent visualization of a graph, a few minor styles were applied to the nodes created which is illustrated in Figure 3.5.



Legend
Blue Rectangle: Vertex with neighbors
Red Circle: Vertex with no neighbors
Yellow Rectangle*: Selected vertex*

Figure 3.5: HIPPIE current dataset with non-source neighbors represented as red circles

iii. Import Network Data Flow

The current user action implemented in the Cytoscape plugin is 'Import Network' in the Apps -> MASS sub-menu. This menu option kicks off the *CreateNetworkTask* implemented in the new MASS plugin for Cytoscape. Figure 3.6 illustrates the process and data flow from the user initiating the Import Network process.

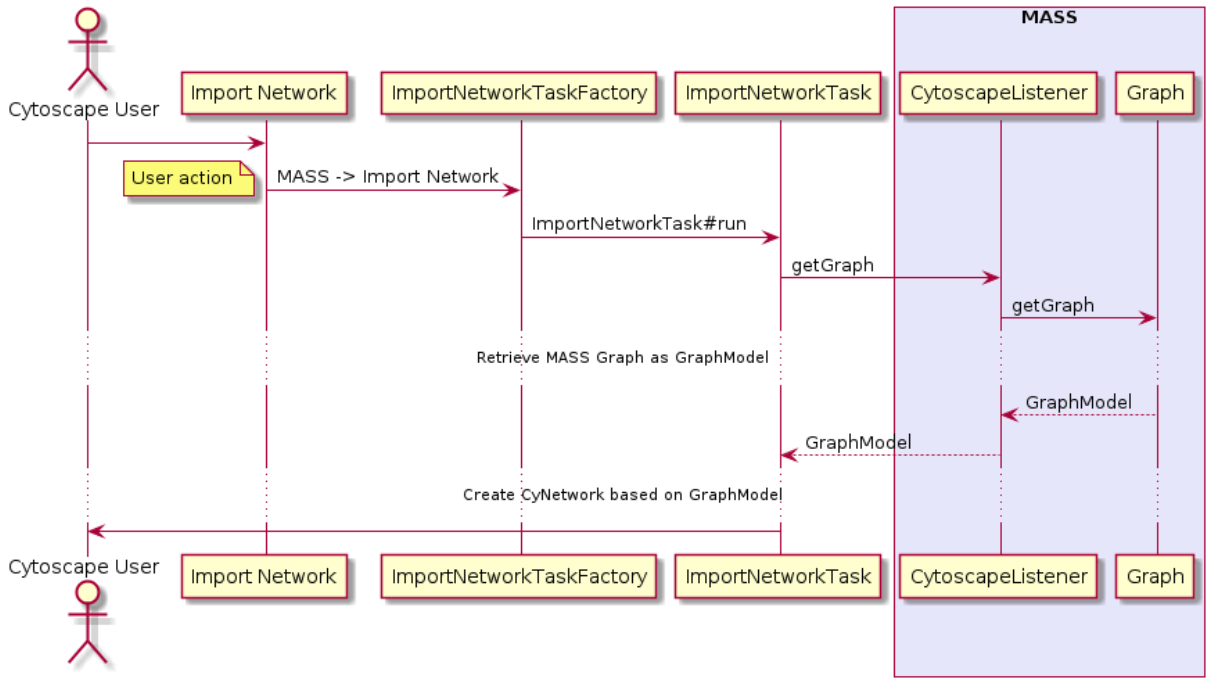


Figure 3.6: Sequence diagram for importing a network into Cytoscape

Chapter 4. Evaluation

This section is an evaluation of the work done for this project. In order to evaluate the work that has been done we will survey the programmability of new features implemented in the MASS library: (a) graph input formats and (b) graph maintenance features. In addition we will evaluate the programmability and usability of our integration of MASS and Cytoscape.

4.1. Results

We will now enumerate the takeaways from this project in the areas of programmability and usability of the new features implemented in this project compared to existing options.

4.1.1. Programmability and Usability

For our purposes, programmability relates to our goal of creating a graph interface to the MASS library that makes application programming accessible to users that are not computer scientists or do not have a strong background in programming.

In order to evaluate the programmability of our new features we compare the implementation of an application using MASS with the existing multidimensional data model. We will measure the programmability based on the number of Java classes required and the number of lines of code to implement a triangle counting application.

Sample Application - Triangle Counting

Throughout the development of the features for this project, we focused our attention on a simple application for counting the number of triangles in a given graph. This application was chosen for 3 reasons: it is a simple application with deterministic results with complete graphs, the code for the MASS library implementation already exists in the sample application repository [21], and the algorithm requires mapping a graph into MASS's multi-dimensional data space.

When re-implementing the application with the use of the graph features of the MASS library, we no longer require the application to map a graph to 2-dimensional space. This alone results in dropping a complete class called Map from the application representing 58 lines of code. Table 4.1 further illustrates the lines of code and class differences between the triangle counting implementations in MASS and MASS w/ graphs.

Table 4.1: Programmability breakdown of triangle counting application

Framework	Classes	Lines of Code	
		Boilerplate/setup	Total
MASS	4	100	407
MASS w/ graphs	2	23	353

In addition to deprecating the Map class, the new graph features ultimately deprecated the custom Node implementation required by the multidimensional MASS implementation of the triangle counting application. This is reflected in Figure 12 as the second reduction of classes required in the second column as well as the reduction of boilerplate code.

The results also show that the boilerplate code is significantly reduced to only 23% lines of code (LoC) in new graph-based implementation of triangle counting as seen in Table

4.2. This reduction indicates that the overall reduction in code is 13% which is concentrated in the setup code.

Breadth-first Search (BFS) is another example of a graph analysis application considered in [24] for programmability. We have re-implemented this application to contrast with the existing analysis based on LoC and number of object instances required.

BFS contrasts with the Triangle Counting application because, while it does benefit from MASS's handling of neighbor management, BFS requires an additional data member to maintained within the vertex data space. This requirement results in BFS requiring a simple VertexPlace implementation provided by the user to complete the algorithm. With the neighbors handled by MASS this class file is reduced to only nine LoC. The complete comparison between MASS and MASS Graph implementation of the BFS algorithm is in Table 4.2.

Table 4.2: Programmability breakdown of breadth-first search application

Framework	Classes	Lines of Code	
		Boilerplate/setup	Total
MASS	5	7	326
MASS w/ graphs	4	8	243

Taking the results of Table 4.2 into account we again see a reduction in classes required to implement the application with the enhanced MASS library. The boilerplate code is nearly the same while the overall line count is reduced as expected due to the removal of a graph management class.

It should be noted that this reduction of classes required by the user application could also be considered deprecation as it is not simply re-organizing code but the MASS library itself is replacing the purpose of the previously required class.

In addition to the Triangle Counting and Breadth-first Search applications, we tested with a variety of programs existing in the MASS sample application repository [21] as well as simple driver programs that were more focused on a specific component under development, and finally a number of new and extensions to existing tests. The

additional modifications and tests were not cleaned up or rigorously tested so not included in the final product.

Functionality

In addition to the raw code reductions allowed by our newly implemented features, we have provided completely new capabilities to users of the library in the form of visualization and dynamic data modification. We will now compare the enhanced features of the MASS library to the existing implementation as well as Repast Simphony summarized in Table 4.3.

Table 4.3: Feature matrix of MASS, Repast Simphony, and MASS w/ Graphs

	Data size	Data modification	Visualization
MASS	static	not available	none
MASS w/ Graph Support	dynamic	dynamic	Import and export
Repast Simphony	static	static	Read only (import)

Our comparison considers three characteristics shared between the three ABM frameworks: MASS, MASS with graphs (as developed by this project), and Repast Simphony. The data size, data modification, agent deployment, and visualization features of each framework are first considered at a high level binary classification in Table 4.3 and then enumerated for the specific features in Table 4.4.

Data Size

MASS: a static specification of the model space size and is created before simulation begins.

MASS w/ Graph Support: an initial size can be specified but can expand or retract without restarting the simulation. The size can also be automatically determined based on the input data.

Repast Simphony: the data model is set once simulation begins

Data Modification

MASS: modification of the data space restarting the simulation

MASS w/ Graph Support: an initial size can be specified but can expand or retract without restarting the simulation. The size can also be automatically determined based on the input data.

Repast Simphony: the data model cannot change once simulation begins

Visualization

MASS: no integrated support for visualization

MASS w/ Graph Support: integrated with Cytoscape via plugins. Graphs can be sent and received from Cytoscape.

Repast Simphony: the “runtime GUI” allows a read-only visualization of the graph.

Table 4.4: Process comparison of MASS vs MASS w/ graphs for graph application

System	Initialization	Interaction
MASS	- Create a tightly packed array representation of the graph	- Neighbors must be mapped to integers
MASS w/ Graph Maintenance	- Create vertices with addVertex - Create edges with addEdge - Import from MATSim or HIPPIE Tab file - Export from Cytoscape	- Neighbors can be any hashable object - Retrieve neighbor index from distributed map

In addition to removing the burden of the graph implementation from the user, the maintenance features added to MASS also increased the usability of MASS by improving the graph interaction paradigm by allowing migration to a vertex without an integer id.

In the existing implementation all neighbors must be integers and the neighbor ids can be used for migration. In contrast, the newly implemented Triangle Counting application keeps the neighbors list as logical names. For migration in the new implementation, the application retrieves the neighbors global index from the map for migration.

To further compare the resulting programming interface of our improved MASS, Table 4.5 breaks down the programming constructs provided by MASS and Repast Simphony in regards to the same three key features: data size, data modification, and visualization.

Table 4.5: Implementation matrix of MASS, Repast Simphony, and MASS w/ Graphs

System	Data size	Data modification	Visualization
MASS	Places constructor parameter	not available	not available
MASS w/ Graph Support	Constructor parameter Input file Graph Maintenance features - addVertex - removeVertex	Graph maintenance functions - addVertex - removeVertex	Import and export in Cytoscape
Repast Simphony	NetworkBuilder (Projection)	Not available	Runtime GUI

In particular, features added to MASS allow dynamic modification of the data space through maintenance features added in this project. This feature is indicated in Table 4.5 by the lack of ability in red in the data modification column.

The contrast of visualization capabilities are a bit more complex. Table 4.5 clearly indicates our improvement over the existing MASS total lacking of the feature. Repast Simphony on the other hand, indeed does have visualization capabilities but does not allow modification of the data space based on the visualization.

4.1.2. Visualization - Cytoscape

The practical benefits of allowing visualizations of graphs in MASS became clear in particular when testing the import plugin with the HIPPIE dataset [10]. In one of our test programs for CytoscapeListener we imported the entire network into Cytoscape.

This quickly surfaced an error because some neighbors of vertices were not in the vertex table. By styling these vertices differently than the rest of the graph, the visualization had evolved to render a clear distinction from vertices with neighbors of other vertices in the graph and those without.

With the added ability of creating a brand new graph in Cytoscape and sending it MASS for processing, we had not only reached a point for visualizing a graph in MASS but we also had an additional method for supplying graph data to MASS for processing.

This project successfully achieved our goals in comparison to Repast Simphony particularly as it relates to Repast's "Runtime GUI" and visualization capabilities described in section 2.1 and in Tables 4.1-4.5. Repast has the ability to visualize a network in progress but has no ability to affect the network with the GUI.

In contrast to Repast Simphony, our CytoscapeListener and Import/Export plugins allow two-way communication between MASS and Cytoscape. Our implementation also improves on the many steps required to visualize a graph: both import and export of the network in Cytoscape are achieved with a single button click.

4.1.3. Execution Performance

For evaluating the performance of our implementation we compare dynamic graph modification implemented for this project with a graph fronted for Apache Spark called GraphX [25]. As spark represents a different paradigm of data analysis, our comparison focuses on modifying the dataspace itself.

The following benchmarks and evaluation evaluate each system's ability to modify an existing graph and continue analysis. To this end we begin each system with a graph before beginning stepped benchmarks of adding to the graph. We have defined this scenario as follows:

1. Each system will begin a with a complete graph of 1000 vertices before timing
2. 100 vertices will be added to the system's data model
3. The system will calculate the total number of vertices
4. Steps 2 and 3 are timed and repeated for 100 iterations

This benchmark results with MASS having a significant lead over Spark-GraphX as seen in Figure 4.1. Though the addition of vertices is constant, 100 vertices are added in each iteration, the runtime increases in both systems due to the increased computation required to generate the complete graph at each iteration. Each iteration compounds the number of neighbors to add to each new vertex.

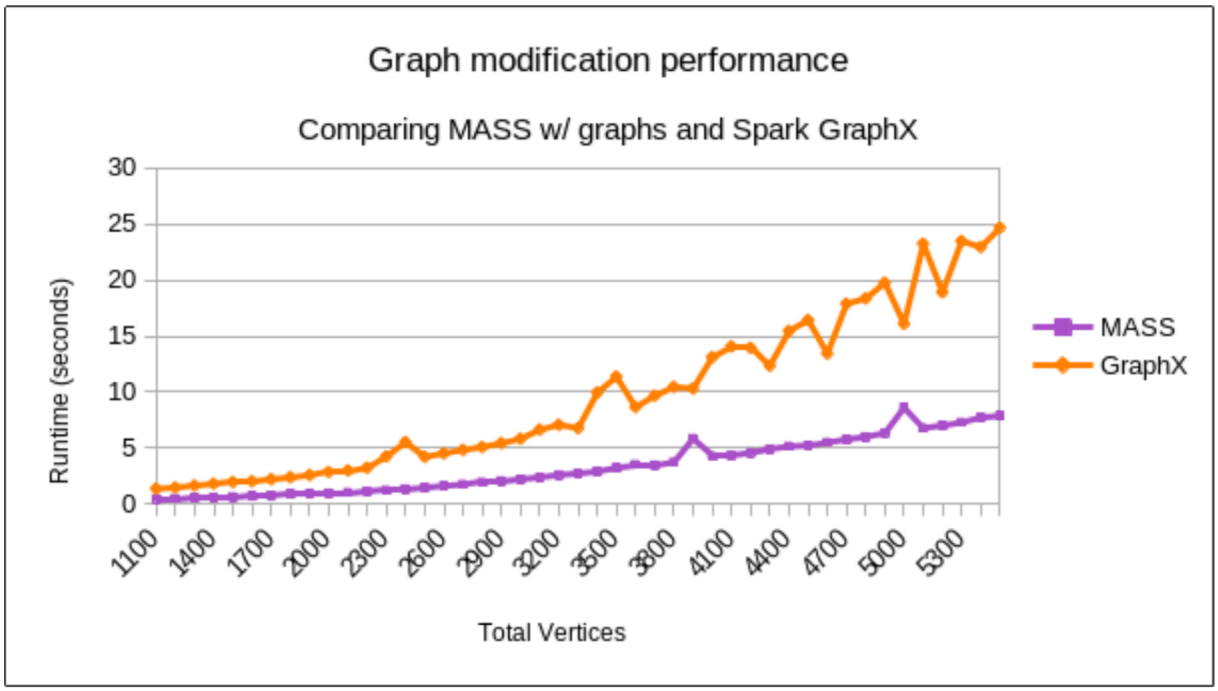


Figure 4.1: Performance comparison of runtime graph modification

To validate these benchmarks we will now review the individual implementations of the benchmark.

Spark GraphX

Apache Spark is a data analysis tool from the Apache Foundation [25]. One particular characteristic that is important to note about Spark is that the data used for analysis is required to be in what Spark calls an RDD, a resilient distributed dataset. The details of RDDs are not important to our discussion, however, the fact that they are immutable is. It is this immutability that significantly impacts Sparks performance when attempting to modify graphs.

In order to represent a graph in GraphX we must create two separate Java collections for the vertices and the edges. These two collections are then converted to RDDs before being combined into a GraphX graph. In order to modify the graph we must extend the complete Java collections and recreate the GraphX graph.

This requirement results in the trade-off of either maintain two complete representations of the graph or recalculating the entire graph on each iterations. For the purposes of these benchmarks we have elected to maintain both graphs in memory which has

limited the size of the graph to be tested. With the default Spark configuration expanding the graph size beyond 45 iterations, 4500 additional vertices, resulted in the Spark task running out of memory.

MASS

For the MASS implementation we utilized the graph maintenance interface developed in this project and specified the base size as 100 vertices. This size specification meant that each iteration would result in an additional layer in the GraphPlaces vertex array.
Modification to the graph was made possible by directly interfacing with the GraphPlaces object just as the initial graph was created. No additional data model was required.

4.2. Discussion

While this project was successful according to its above goals, there are a few misses that could be improved in the future. At this time we iterate 2 shortcomings that are important considerations for future related work as well as expansion to features in this portion of the MASS library.

4.2.1. Mixed Data Model

Firstly, we implemented the graph maintenance features into the library in such a way that the existing data model would be usable unchanged in existing applications. This is an important feature for software development so that new changes to a library do not break existing code.
The first portion of the implementation of this project was the implementation of HIPPIE tab and MATSim file formats. As this code was deep in existing MASS code, we implemented this portion of the project using the existing MASS data model. It was not until we began work on the graph maintenance features that we developed the data model described in section 3.3.2. Vertex Mapping.
This decision led to graph support existing within both the existing multi-dimensional data-mode and the new layered data-model within GraphPlaces. The dimensional model is used for graphs from input files while the layered model is used for vertices and edges added to the graph.

The situation is further exacerbated when vertices are added to a graph imported from a file. This led to complicated conditional logic in agent and place logic for determining actions to take as well as the non-obvious deficiency that, in its current state, graph maintenance features do not support the graph imported from a file.

This is not to say that graph maintenance features are unavailable when importing a graph from a file, however, the vertices created from are created within the existing MASS data space and cannot be connected dynamically. One current workaround for this shortcoming is to populate the graph from a file into MASS then import the MASS graph into Cytoscape followed by exporting back to MASS. This process will result in the entire graph being within the graph maintenance data model and facilitating future modifications to the entire graph.

In hindsight it would have been prudent to update the import formats to use the graph model to unify all graph implementation and avoid this shortcoming.

4.2.2. Cytoscape Integration Limitation

For our initial implementation of integrating MASS and Cytoscape, it became apparent that implementing the integration in a protracted period of time we were required to simplify the initial support. In order to simplify the communication between MASS and Cytoscape, the current implementation requires converging the entire graph data model, the GraphModel, onto the control node and then into Cytoscape.

While this data model is a simplified form of the entire graph, it does limit the size of the graph to one that would fit on a single node. It would be possible to implement the communication in such a way that the data was converged within Cytoscape but this would retain the memory limitation. As such, the current implementation of Cytoscape to MASS integration is not able to stream extremely large graphs from MASS to Cytscape.

For this reason we elected to implement the convergence solution for the initial version of Cytoscape implementation and defer support for larger graphs to a future project.

4.2.3. Performance

The performance improvements seen comparing MASS with graphs to Spark with GraphX are promising, however, the results are based on an elementary understanding

of Spark and GraphX. It is possible there other improvements that would narrow the gap between MASS and Spark. Nonetheless, due to Spark supporting modification of a dataset we expect MASS to ultimately surpass Spark's performance to some extent.
After compiling the benchmark results for this report it was determined that the benchmarks were not continuing to maintain a complete graph in either system. While we began with a complete graph of 1000 vertices, additions to the graph were made as follows:

1. 1000 vertices added
2. all *new* vertices are connected to the entire graph

The outcome of this algorithm is that new vertices are connected to all remaining vertices but existing vertices are not modified resulting in a formally incomplete graph. Despite this oversight, the algorithm is maintained in both MASS and Spark so the results remain valid. A complete graph is note necessary to compare performance of the systems.
An additional point to note from Figure 4.1 is the wide variation of runtime seen in Spark compared to MASS. This could be attributed to the complexity of running a simulation in Spark as well as the overhead of communication between Spark tasks. In order to run Spark we require a control process and a worker process. For this performance analysis both processes were run on the same computing node.

4.2.4. Programmability

Programmability is a nebulous word that based on English grammar and [27] is defined as the an ability to be programmed. By this definition either system is programmable but in the case of this paper we are interested in the ease of programming the systems. One metric we chose to quantify "ease" in this paper was the number of classes required to develop a client application but this comes with a caveat: we could put all the code into a single class and call it a day.
For this reason we also include breakdowns of boilerplate and total lines of code required to help rectify this point. Putting all of the code into a single class would have a lesser affect on the overall lines of code but again we can write very compact code with

lambda functions, anonymous classes, and a variety of programming constructs to reduce the number of lines of code.

In these ways we see that number of classes and lines of code may not be the best metrics for qualifying the programmability of a system. Creating maintainable software requires careful consider of the trade-offs between being concise and being expressive. Longer code that clearly illustrates a process may be much easier to understand than a fragment of code that is crafted to be as short as possible. However, by focusing on the reduction of code in the form of deprecation we can have a better sense of the reduction of code being unbiased.

The code reductions described with respect of triangle counting and breadth-first search are specifically components that are no longer required by the client application. In this way our programmability enhancements to MASS are in the form of reduced responsibility of the client application.

Chapter 5. Conclusion

In conclusion, we achieved our functional goals of this project through the implementation of the graph input formats in MASS, graph maintenance features in MASS, and two-way communication between MASS and Cytoscape. Furthermore, we believe that the work completed in this project will not only enable more broad use cases of the MASS library but the project also opens the doors for the Distributed-Systems Laboratory to many new innovation possibilities for both MASS and its integration with Cytoscape. In addition, by integrating with Cytoscape we have enabled MASS to reach new audiences including other research groups at UW interested in graph visualization.

While as a group working towards a common goal of improving MASS and exploring agent-based modeling solutions, qualifying programmability improvements is difficult. Even considering the reduction of lines of code to complete a task does not account for the cognitive overhead of one implementation compared to another.

This project set out to address 3 goals that lack in similar systems by enhancing the MASS Java library:

1. Graph Input formats
2. Programmability
3. Visualization

The following section addresses our successes, failures, and possible paths forward for MASS.

5.1. Summary

The primary goals of this project are centered around bringing support for graphs inside of MASS to increase programmability and usability of MASS for users that are not computer scientists or experienced developers.

A. Graph Input Formats

The implementation of adding input formats was clear cut from an interface perspective and we successfully enhanced MASS to allow importing graphs from the HIPPIE and MATSim file formats. Our early decisions led to technical debt in the form of having a mixed data model between the old and new implementations resulting in complex code within the library.

B. Graph Maintenance

We have implemented a new user object, GraphPlaces, that allows users to create and manipulate a graph at runtime. These features will allow users to easily approach the dynamic creation and analysis of graphs at runtime with only a few methods.

C. Cytoscape Integration

The project has resulted in plugins and enhancements to MASS that facilitate transferring graph information back and forth between the systems. By creating a simple request interface we have not only enabled the user to use the getGraph and setGraph methods but also opened many other possibilities.

5.2. Future Work

Besides the improvements detailed in section 4.2, this project inspired a number of

ideas for possible additions and enhancements to the MASS library that could be considered for future work or projects.

5.2.1. Graph Input Formats

1. These implementations should be more modularized and not tightly coupled with PlacesBase. Perhaps an interface that allows MASS to retrieve required information without knowing the implementation.
2. The GraphInputFormat enum should be utilized instead of depending on the file extension. Users may not want to change the file extension. For example, the HIPPIE dataset is downloaded as .txt file requiring me to change the extension to .tsv
3. Optimize file input reading and graph generation. While this project found true parallel file I/O with direct indexing into files with inconsistent line characteristics, there are other possibilities to improve the import performance. It could be determined if an input is small enough it can be read entirely into memory for caching on each node. It may even be possible to utilize the existing Parallel I/O code. Then populating the neighbors could look at memory rather than reading the file again. For example, caching the split input lines of HIPPIE brought the import time of the HIPPIE_CURRENT dataset from 3162.353s to 408.852s, a 7.735x speedup. However, this would require each node to maintain the entire input file in-memory which may not always be feasible.

5.2.2. MASS Graph Maintenance

a. Vertices

1. addVertex requires a second parameter to pass additional information to the constructor. This has been added provisionally in a repository branch.

b. Edges

1. Allow replacing existing edges in the event that the user would like to update the weight without adding a new function call such as *setWeight*.
2. Allow adding edges where the neighbor is not a vertex in the graph.
    a. In light of the HIPPIE network resulting in vertices with neighbors that

were not created by the initialization algorithm, it may be prudent to allow edges to be between a source and non-source vertex but that is a problem for the future.

c. Performance

1. Benchmark the graph implementation for comparison with the existing implementation.
2. Investigate and implement possible performance improvements relating to parallelism of the new features. The current implementation is split at the process level but not yet at the thread level.

5.2.3. Cytoscape

1. Modify the Import Network Cytoscape plugin to a layout the the new graph such as ‘Force Directed Layout.’ The current implementation stacks all vertices on top of each other.
2. Give some flexibility to the user for styling in Cytoscape such as additional attributes in the model.
3. Give a name to the graph imported from MASS
4. Extend the plugins to support partial updates to existing graph
5. Extend plugins to allow configuration of MASS port and host. Currently it is hardcoded to localhost on port 8165. This could be a good starting point for adding other configuration options to the plugin.
6. Improve the way that the graph is communicated from being distributed in a MASS cluster to being displayed in Cytoscape to stream parts of MASS data into and out of Cytoscape.
7. Create a plugin that allows running operations on the graph and sending the result back to Cytoscape for display. This would allow better demonstration of how the interaction can be utilized for graph analysis.
8. Create a simple plugin that would allow specifying an arbitrary process to run in MASS. This is implemented on my by allowing the user to register the process but not yet implemented for Cytoscape.
9. Create a plugin that would allow a user to send a program from Cytoscape to

MASS for simulation on the graph. This could be simplified with a convention such as perform this operation at each place or use these rules to determine agent migration where the user sends code snippets to define these actions.

5.2.4. MASS

1. Experiment with other languages. I used Kotlin for some work and it was quite nice to be able to write the restrictions of Java.
2. Standardize an easily repeatable deployment that allows multiple nodes without requiring UW resources. This could be a public cloud (GCP, AWS, Azure), or ideally docker, kubernetes, vagrant. I was using a manually provisioned VM on my local machine that was much less painful than attempting to use the hermes cluster.
3. Instrument the library to better support analyzing performance of a simulation. It could be helpful if MASS tracked memory usage to show statistics such as min, max, mean memory usage.

References

[1] J. Doyne Farmer, & Duncan Foley. (2009). The economy needs agent-based modelling. Nature, 460(7256), 685-686.

[2] Yun-Ming Shih, Collin Gordon, Munehiro Fukuda, Jasper van de Ven, Christian Freksa, Translation of String-and-Pin-Based Shortest Path Construction into Data-Scalable Agent-Based Computational Models, Proc. of the 2018 Winter Simulation Conference, pages 881-892, Gothenburg, Sweden, December 2018

[3] C. Blum, “Ant colony optimization: Introduction and recent trends,” Phys. Life Rev., vol. 2, no. 4, pp. 353–373, 2005. doi: 10.1016/j.plrev.2005.10.001.

[4] Emau, J, Chuang, T., Fukuda, M. “A Multi-Process Library for Multi-Agent and Spatial Simulation.” Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. 2011.

[5] North, MJ, NT Collier, J Ozik, E Tatara, M Altaweel, CM Macal, M Bragen, and P Sydelko, "Complex Adaptive Systems Modeling with Repast Simphony", Complex Adaptive Systems Modeling, Springer, Heidelberg, FRG (2013). https://doi.org/10.1186/2194-3206-1-3.

[6] "Repast/repast.simphony", GitHub, 2020. [Online]. Available: https://github.com/Repast/repast.simphony/releases. [Accessed: 27- May- 2020].

[7] RepastHPC. https://repast.github.io/repast hpc.html. [Accessed 2020].

[8] Borgatti, S.P., Everett, M.G. and Freeman, L.C. 2002. Ucinet for Windows: Software for Social Network Analysis. Harvard, MA: Analytic Technologies.

[9] "Repast Java Getting Started", Repast.github.io, 2020. [Online]. Available: https://repast.github.io/docs/RepastJavaGettingStarted.pdf. [Accessed: 09- May- 2020].

[10] Gregorio Alanis-Lobato, Miguel A. Andrade-Navarro, Martin H. Schaefer, HIPPIE v2.0: enhancing meaningfulness and reliability of protein–protein interaction networks, Nucleic Acids Research, Volume 45, Issue D1, January 2017, Pages D408–D414, https://doi.org/10.1093/nar/gkw985

[11] "Repast Simphony Reference Manual", Repast.github.io, 2020. [Online]. Available: https://repast.github.io/docs/RepastReference/RepastReference.html. [Accessed: 27- May- 2020].

[12] Mariam Kiran, Paul Richmond, Mike Holcombe, Lee Shawn Chin, David Worth and Chris Greenough, FLAME: simulating large populations of agents on parallel hardware architectures, Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems

[13] S. Chin, "FLAME: Flexible Large-scale Agent Modeling Environment",

Flame.ac.uk, 2020. [Online]. Available: http://flame.ac.uk/. [Accessed: 28- Apr- 2020].

[14] K. Ono, "Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization", Cytoscape.org, 2020. [Online]. Available: https://cytoscape.org/. [Accessed: 28- Apr- 2020].

[15] Horni, A., Nagel, K. and Axhausen, K.W. (eds.) 2016 The Multi-Agent Transport Simulation MATSim. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw. License: CC-BY 4.0

[16] "matsim-org/matsim-libs", GitHub, 2020. [Online]. Available: https://github.com/matsim-org/matsim-libs/tree/master/examples/scenarios. [Accessed: 27- May- 2020].

[17] Henning Hermjakob, Luisa Montecchi-Palazzi, Gary Bader, Jérôme Wojcik, Lukasz Salwinski, Arnaud Ceol, . . . Rolf Apweiler. (2004). The HUPO PSI's Molecular Interaction format—a community standard for the representation of protein interaction data. Nature Biotechnology, 22(2), 177-183.

[18] “HIPPIE: Human Integrated Protein-Protein Interaction rEference” Cbdm-01.zdv.uni-mainz.de, 2020. [Online]. Available: http://cbdm-01.zdv.uni-mainz.de/~mschaefer/hippie/hippie_current.txt. [Accessed: 27- May- 2020].

[19] A. von Mayrhauser, J. Wang and Q. Li, "Experience with a reverse architecture approach to increase understanding," Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360), Oxford, England, UK, 1999, pp. 131-138.

[20] “The Leading In-Memory Computing Platform,” Hazelcast. [Online]. Available: https://hazelcast.com/. [Accessed: 14-Mar-2020].

[21] “MASS Java Applications Repository” https://bitbucket.org/mass_application_developers/mass_java_appl. [Accessed: 28-May-2020]

[22] "OSGi Alliance | Specifications / HomePage". osgi.org. 2015. Retrieved June 30, 2016.

[23] Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, Amin N, Schwikowski B, Ideker T. “Cytoscape: a software environment for integrated models of biomolecular interaction networks.” Genome Research 2003 Nov; 13(11):2498-504

[24] C. Gordon, U. Mert, M. Sell, M. Fukuda “Implementation Techniques to Parallelize Agent-Based Graph Analysis.” 2019

[25] "GraphX | Apache Spark", Spark.apache.org, 2020. [Online]. Available: https://spark.apache.org/graphx/. [Accessed: 07- Jun- 2020].

[26] Morris, J., “Cytoscape 3.3 Developers Tutorial” in Intelligent Systems for Molecular Biology, Jan-2020.

[27] B. L. Chamberlain, D. Callahan, and H. P. Zima. “Parallel programmability and the chapel language”, International Journal of High Performance Computing Applications, 21(3):291–312, 2007

[28] Spiger, John. (2011). Analysis of MATSim. Distributed Systems Laboratory UWB. Jan-2020

[29] Cytoscape, “cytoscape/cytoscape-app-samples,” GitHub, 20-Dec-2017. [Online]. Available: https://github.com/cytoscape/cytoscape-app-samples. [Accessed: 14-Mar-2020].

[30] "Repast Simphony Frequently Asked Questions", Repast.github.io, 2020. [Online]. Available: https://repast.github.io/docs/RepastFAQ/RepastFAQ.html. [Accessed: 27- May- 2020].

[31] M. Fukuda, "Distributed Systems Laboratory at UW Bothell", Depts.washington.edu, 2020. [Online]. Available: https://depts.washington.edu/dslab/. [Accessed: 27- May- 2020].

Appendix A: Implementation Details

A.1. Input Formats

Each input format has a unique initialization function *init_all_graph_[format]* in PlacesBase that will populate the MASS cluster with vertices based on the given input file. MASS will use the last 4 characters of the filename to determine which parsing algorithm to execute for populating the graph. For example, if the filename is *network-complete.xml*, MASS determines the filetype to be MATSim and uses init_all_graph_matsim to initialize the cluster.

From this point the *init_all_graph_[format]* will parse the input data for vertices and initialize the associated vertices for the node. To determine the size of the MASS layer 0 *size*, the parsing algorithm must determine the number of vertices represented in the input file.

After MASS determines the dataspace size, the second initialization step requires initializing a Place instance for each vertex. These objects are subclasses of VertexPlace in the type specified by the MASS client. The format type parsing method must then iterate all of the vertices in the input file and use the requested class name to create an instance for the new vertex.

To map the vertex ids to a meaning linear index, the map must allow an arbitrary but homogeneous structure of keys and values. Furthermore, the keys must implement the hashCode method so that keys can be distinguished by the map implementation. Thus we allow the distributed map to have an object as the key and value.

This distributed map must be accessed by all nodes and all threads but is also never written to after the keys are initially inserted. For this reason we decided to keep a public static instance of the map in the MASSBase class for ease of use. The map itself is initialized by MASS's usual initialization functions.

For initialization of each vertex, MASS creates an instance of the users VertexPlace subclass passing 3 arguments to each place as an object array: the input filename, the input weight filename (*ignored*), and the global index of this vertex. The first index to be managed by a node is determined based on the node id multiplied by *stripeSize,* described in 3.3.2 - VertexMapping, then each place is assigned a sequential id starting

from this offset.

It is inside this newly created VertexPlace subclass constructor that a second format-specific initialization method init_neighbors_[format] is called to pull in the vertex's neighbors from the input file. The init_neighbors_[format] method uses a reverse lookup in the distributed map to retrieve the logical vertex id.

The final step of initialization is using the vertex's logical id to parse the input file and extracts the current vertex neighbors. The existing CSV implementation of this algorithm had separated neighbor ids and neighbor weights into 2 files but within MATSim and HIPPIE formats, the file is combined.

Each input format requires a slightly different implementation of init_all_graph_[format] and init_neighbors_[format].

a. HIPPIE

For the HIPPIE vertex id, as described in section 3.2.1.b, we had a choice between a large non-sequential number or a string to identify a vertex. In the future this might be an option to be left up the user we chose to use the protein sequence to represent the vertex id.

Choosing the protein sequence is ideal for this project because of our goal of integrating with Cytoscape will be further enhanced by representing vertices in Cytoscape with their protein sequence rather than a simple integral identifier.

init_all_graph_hippie

As with the existing init_all_graph_csv parser, the first step was to count the lines in the file to determine the number of vertices. The HIPPIE format requires a slight modification to the CSV algorithm in that there are multiple lines representing the same vertex. This is because the CSV format is an adjacency list whereas the HIPPIE format is in the form of an edge list.

The duplicate unique keys were extracted with the help of the distributed map. As each line was parse, the source protein was queried in the map. If the protein was not present, it was inserted with a sequential integer as the value and the total count was incremented. Proteins that were already present are simply skipped.

When importing a graph into the MASS library, the input file currently must be read sequentially first by the control node. For this reason, the counting and population of the

map is sequential on the control node, there are no multithreading concerns.

Following this algorithm, the process of counting the vertices also populates the distributed map with the proteins as the key and the global linear index as the value.

init_neighbors_hippie

The VertexPlace constructor then uses the index parameter to do a reverse lookup in the distributed map to retrieve the logical id of the vertex is it to manage. The logical id can then be used to scan the input file for the vertex's neighbors to populate the neighbors and weights arrays.

For the HIPPIE format, neighbors are determined by lines where the current vertex is the first vertex in the file again meaning a similar parsing method as the CSV parser. Each line in the input file is iterated and converted to a useful format for determining its attributes.

Due to the relative complexity of the HIPPIE line format, we created a data class HIPPIETabEdge with 2 static helper functions. The first function, getPart, takes an array of the line parts and an enum of the desired part returning the value of that part. The second function, fromParts, is a factory method that takes an array of the line values as a parameter and returns a HIPPIETabEdge object populated with attributes based on the input.

With these helper functions, init_neighbors_hippie can then read the input file line-by-line using getPart to determine if the value of PROTEIN_KEY is equivalent to the current vertex's protein key. If so, the line is converted to an edge with fromParts and the neighbor is added with its INTERRACTION_ATTRIBUTE as the weight.

b. MATSim

The MATSim network format, unlike HIPPIE, keeps a complete list of the network vertices and as well as the edge list. This allows simplification of counting the total number of vertices but otherwise the edge list results in a similar implementation. Furthermore, with MATSim being an XML format, we were able to use Java's built-in XML parsing capabilities in the javax.xml.xpath package to streamline parsing the network files.

Knowing there is a list of node elements called nodes in the input file, we deduced the number of vertices in the network based on an XPath query to retrieve a NodeList of the <node /> elements. We could then iterate this list to insert the node ids into the distributed map and at the same time accumulate the total number of nodes which are equivalent to vertices in this instance.

The XPath query we used to retrieve the node list can be seen in Figure 5. While detailed explanation of the XPath query language is beyond the scope of this document, this query can be broken down to a few key components.

//

The first double slash tells the XPath engine to match at any level within the document, in contrast to a single slash which is similar to an absolute path in a filesystem.

//nodes

The nodes component matches <nodes /> elements. Combining double-slash with nodes tells XPath to match a nodes element at any level of the document. The MATSim format has only a single <nodes /> element so a more explicit path is not necessary.

//nodes/node

The node component, being subordinate to the nodes component, matches any <node /> element within a <nodes /> element.

//nodes/node/@id

The final component, @id, tells the Java XPath engine to specifically query the id attribute of the <node /> elements.

//nodes/node/@id
Figure A.1: XPath node query used by PlacesBase to get count of nodes

The result of this query is a NodeList of the node element ids specifically. In other words, the attributes are also considered a node so we have a list of key value pairs for each node where the key is always *id* and the value is that nodes associate *id* value.
With the node ids inserted into the distributed map and the number of vertices determined, MATSim can use the shared algorithm to initialize places by iterating over the stripeSize places and passing each Place entity its associate linear index for reverse searching in the distributed map.

init_neighbors_matsim

Within VertexPlace, we were again able to utilize a slightly more sophisticated XPath query in Figure 6 to retrieve all of the links that contain the attribute 'from' with a value of the vertex's id. In this case the list is links.

/network/links/link[@from=*vertexId*]
Figure A.2: XPath link query. vertexId is replaced by the querer's id

This query is broken down for reference:

/network

Unlike the previous query, we used an explicit path for starting from *network*, the root level element in this file.

/network/links

Similarly to nodes, we are interested in the list of links in this file.

/network/links/link

Each link represents an edge in the network

/network/links/link[@from=*vertexId*]

The final component of this query encoded a few pieces of information. The first to note is the lack of a final slash between link and the attribute reference. Unlike the <node /> elements above, we require all of the attributes of the <link /> elements. This is achieved by not including a slash after *link*.

The second portion of this component, [@from=*vertexId*], tells the query to math <link /> elements that have a from attribute that equals a specific value.

By using the XPath query in Figure 6, replacing *vertexId* with the current vertex's logical vertex id, we can pull all of the link elements from the input file as a NodeList. With the query set up as described above, each Node in the list is a <link /> element with all associated attributed.

This NodeList can then be used to populate the neighbors via the to attribute and the length attribute. The neighbor's logical id being encoding in the to attribute, and the weight within the length attribute.

A.2. MASS Graph Maintenance

At a high level, graph maintenance in MASS can be simplified to the adding and removing of vertices and edges. As with the existing implementation we require a place to store a dynamic collection of vertices represented as VertexPlace or a subclass of VertexPlace.

Concerning context for the following implementation descriptions is that, currently, all graph maintenance operations are performed in the application against the GraphPlaces instance which runs in the main thread on the control node. This means that all graph maintenance operations begin in the context of node 0, the control node.

a. Data Model

The existing implementation, described in section 3.1, treats the data space as a multidimensional array which is linearized similarly to how a programming language like C stores a multidimensional array in row-major order. All of the elements in row 1 are followed by all of the elements of row 2.

In this fashion, each node is responsible for maintaining a 1 dimensional array of Place entities. MASS then uses the node boundaries to calculate which fraction of the total to manage per node.

To support a graph maintenance in MASS we based our new model on the existing model with a few enhancements. Based on the initial global size, specified in the constructor arguments, or from the input file, we use the same distribution per node as a reference for stripeSize. See section 3.3.2 for a detailed explanation of the data model to be implemented.

Each node is allocated an instance of GraphPlaces responsible for managing the node's VertexPlace assignment. The instances of VertexPlace are held within a Vector of Vectors of VertexPlace, `Vector<Vector<VertexPlace>>`, the first vector holds within a vector for each layer of the graph to represent. Each layer will increase to a maximum of stripeSize instances before an additional layer is created.

b. Vertices

The primary problem to solve with adding vertices is solved with the implementation of mapping arbitrary keys to linear indices. By using a distributed map for mapping vertex

attributes to a global index and an addressing scheme for mapping these global indices to a specific vertex, adding and removing vertices is a matter of determining if the vertex exists in the map and sending the newly defined messages ‘MAINTENANCE_ADD_PLACE’ and ‘MAINTENANCE_REMOVE_PLACE’ to the appropriate node. The owning node is a function of the global linear address that is described in section 3.3.2. The addVertex function is called on the node and the first determines whether the vertex already exists.

If the vertex does not exist, the algorithm next determines whether the current node is responsible based on the addressing algorithm from section 3.3.2. More specifically, the owning node id is determined by taking the modulus of the global index retrieved from the distributed map.

An example follows for a new vertex with the logical id *Hello* in Figure 7.

```
Vertex attribute: Hello
Global index: 362
MASS Size: 100
Stripe size: 25

Layer Index: 3 (362 / 100)
Node index: 2 (362 % 100 / 25)
Place index: 12 (362 % 25)
```

Figure A.3: Example vertex addressing

Assuming a new vertex “Hello” is assigned a global index of 362, we can determine the correct address of this new vertex by determining the layer, node, and local place index. In this way a global linear address is translatable to an address in the form of (*layer*, *node*, *localIndex)*.

The value of the key *Hello* in the distributed map will be 362, its global index. In a MASS cluster of 4 nodes with a size of 100 the stripeSize, number of elements assigned per node per layer, becomes 25 elements.

Using these values, we can determine the layer index dividing the global index by the MASS size. The node index is determined by dividing the global index by the MASS size using the modulus operator to get the remainder. The remainder is then divided by

the stripe size to reach the 0-based node index.

The local place index, the index within the vector of VertexPlace, is achieved by taking the remainder of dividing the global index by stripe size using the modulus operator.

Using these results the system can determine where to put and where to retrieve a given vertex. Going back to addVertex, having determined the associate logical name is not within the map, we use a sequential accumulator in the GraphPlaces class on the control node to assign a new global index.

i. addVertex

The new global index is then used to determine which node the vertex will be assigned based on the algorithm in Figure 7. If the current node is the owner, a method called addPlaceLocally is called to create a new instance and add to the correct layer.

If the owning node is a different node, we use MASS's existing message passing infrastructure to send a MAINTENANCE_ADD_PLACE message with the new logical id directly to the correct node. Upon receiving this message, the remote node is able to call addPlaceLocally on the local GraphPlaces object to add the new vertex to its collection.

ii. removeVertex

Removing vertices reverse this process with the additional step of removing references to the vertex throughout the cluster. Since a vertex can be a neighbor to any other vertex in the graph, we can process removing the vertex in a form of broadcast message. We first send the MAINTENANCE_REMOVE_PLACE message with the associated vertex id to all places then proceed to call removePlaceLocally.

The method GraphPlaces#removePlaceLocally must not only remove the place associated with the given vertex but also iterate all vertices managed by this node in all layers and remove any edge referencing this vertex as a destination.

c. Edges

In contrast to vertices, edges are associated with two different vertices. We must find which vertex owns the edge based on the source vertex, but also determine if the destination exists as well.

i. addEdge

For adding an edge we first confirm that both the source and destination vertices exist and that the source vertex does not already contain the given edge. Vertices can be verified with a simple lookup in the distributed map. In this instance, a failed lookup represents a failure condition for adding an edge and the method return.

Once both source and destination vertices have been verified, the global index of the first vertex is used to determine the owner of the vertex with getNodeIdFromGlobalLinearIndex, a helper function in GraphPlaces.

From this step, the flow for edges is decidedly similar to vertex operations, if the current node is the owner we call a method addEdgeLocally, if the owner is another node we send a message,

MAINTENANCE_ADD_EDGE, with the source vertex, destination vertex, and weight, to the owning node.

When a remote node receives this message, the addEdgeLocally method is called locally to complete the process.

The addEdgeLocally method determines the owning vertex place with the addressing algorithm and calls the addNeighbor method to add it to the VertexPlace entity's neighbor array if not already present.

ii. removeEdge

Unlike removeVertex, removeEdge is not a cascading process. An edge is a single neighbor of a single vertex. As with other methods described so far, the removeEdge method checks for the presence of the source and destination vertices in the distributed map.

If both vertices are present in the map, the source vertex linear index is used to find the owning node. If the owner node is 0, removeEdgeLocally is called directly control node, otherwise the parameters, source vertex id, destination vertex id, send in a MAINTENANCE_REMOVE_EDGE message to the owning node.

The owning node then calls its removeEdgeLocally method which determines the correct place associated with the edge and removes the neighbor with that VertexPlace entity's removeNeighborSafely method. This method, as the name implies, removes the neighbor if it exists or is a no-op otherwise.

A.3. Cytoscape Integration

To integration MASS with Cytoscape, we implemented the infrastructure for such communication in MASS via a new thread to listen for requests from Cytoscape. This listening thread serves as a dispatch to respond to requests from Cytoscape making Cytoscape a driver for interactions in both directions. We considered a few options for what methods to communicate over the socket connection but settled on using Java's built-in object serialization because it has the least overhead compared to RMI or JSON serialization and it is already the primary communication method within MASS.

In order to facilitate this bi-directional communication channel between MASS and Cytoscape we also required a data model that would be lightweight for transferring over the network. This data model also serves as a bridge between systems that have different internal representations for their graphs. This communication is depicted in Figure 8 for the request and response of the Import Network plugin.

To this end we created the GraphModel and VertexModel classes seen in Figure A.3. A graph is represented at a high level by the GraphModel class which in turn has a collection of VertexModel objects representing the vertices. Inside of each VertexModel object is a collection of IDs that represent the identifiers associated with their neighbors' vertex id.

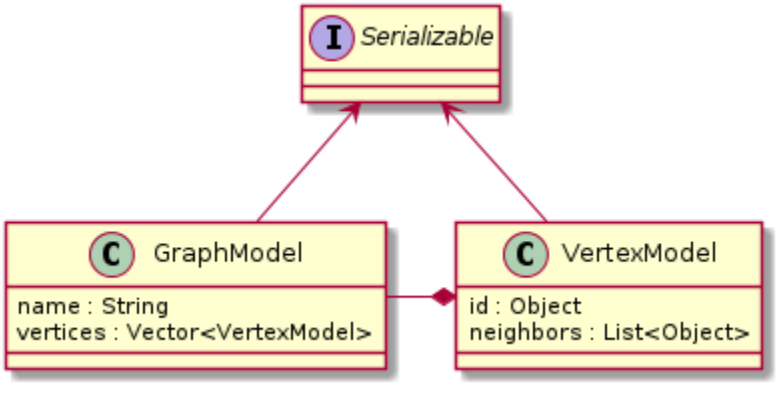


Figure A.4: GraphModel and VertexModel class diagram

By implementing the Serializeable interface in GraphModel and VertexModel classes we can send objects over a socket connection via ObjectOutputStream#writeObject, the ObjectInputStream class's writeObject method, and read objects from the socket stream via ObjectInputStream#readObject casting them to the expected type. This inheritance can be seen in Figure A.3. Furthermore, the one-to-many relationship between GraphModel and VertexModel classes in Figure A.3.

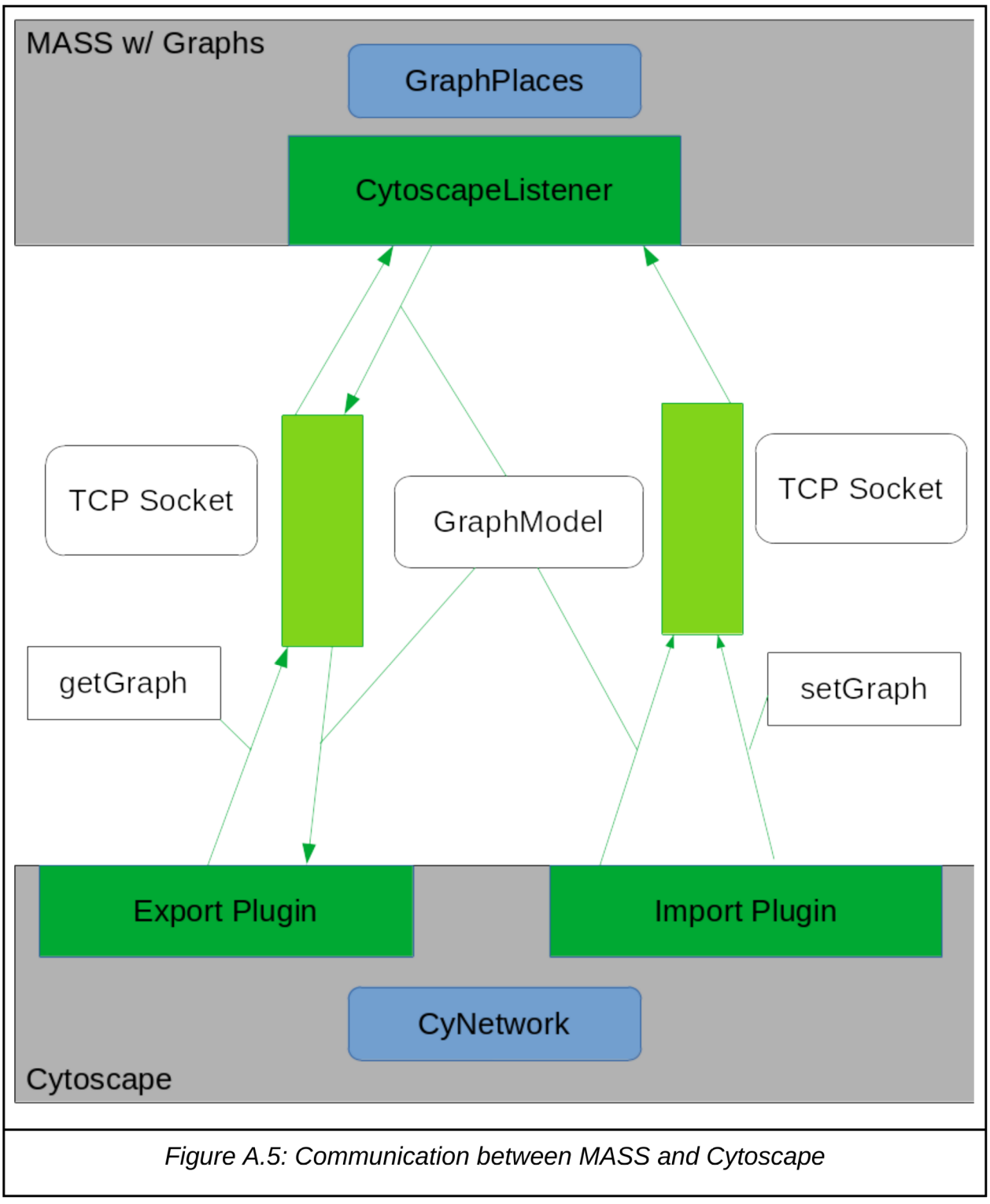


Figure A.5: Communication between MASS and Cytoscape

The VertexModel is intended to be a baseline representation of a vertex in a graph. This data class has only 2 data members for the vertex id and a list of the vertex neighbors.

Both members are represented by Objects for maximum flexibility. For immutability, the vertex id, and neighbor list are set via constructor parameters.

The GraphModel class contains a field for a graph name and a list of VertexModel objects to store the graph's vertices and, by extension, the edges. Vertices within the GraphModel are added with a method addVertex.

As the data model classes are intended for serialization, neither class supports removing data and minimal additional functionality was implemented.

a. CytoscapeListener

In MASS, we give the user the option to enable CytoscapeListener knowing that this feature will not be used in every run of the library and not by every user. The CytoscapeListener is a class that implements a generic interface that could be implemented for visualization in other packages in the future but CytoscapeListener is the focus of this project. This class appears in the MASS portion of Figure A.3.

Creating a CytoscapeListener requires a single parameter: a Graph interface to affect during processing of requests from Cytoscape. The GraphPlaces class implements this interface so the base so it can be passed as a parameter to the constructor. The CytoscapeListener constructor saves this parameter in a field for later reference.

This class creates a thread which opens a ServerSocket on port 8165 and processes each incoming request by first reading a String instance from the socket's input stream. This String is used to determine which method should be performed to handle the request.

Currently the listener implements two methods: getGraph, and setGraph which are associated with import network and export network respectively. The data flow of getGraph is represented in Figure 8.

We have encapsulated determining the request type of a factory method called parseRequest which returns an object of type GraphRequest. GraphRequest is an interface with a single process method defined.

After parsing the request into the appropriate GraphRequest, the request is handled by calling process on the GraphRequest and directly writing the result to the output stream of the socket.

The CytoscapeListener class also has a convenience method registerProcessor that

takes a key and a GraphRequest implementation as parameters. This method allows user applications to extend the CytoscapeListener class to handle arbitrary requests from the Cytoscape plugin. Additional functionality would need to be added to the plugins but MASS would not require changes.

b. GraphModel Conversion in Cytoscape and MASS

On each side of the communication bridge between MASS and Cytoscape, depicted in Figure A.3, we have implemented conversion methods that convert to and from their respective in-memory data structure into the `GraphModel` representation. These functions are called by MASS message processors in the `CytoscapeListener` class and the various Cytoscape plugins to convert to a CyNetwork.

i. Cytoscape

For Cytoscape we have created the methods `graphToCyNetwork` and `cyNetworkToGraph` methods to convert `GraphModel` to an equivalent `CyNetwork` and `CyNetwork` to an equivalent `GraphModel` respectively. The `CyNetwork` data representation that Cytoscape uses is achieved by creating a `CyNode` object to represent each vertex.

While iterating the GraphModel and creating instances of CyNode we must also maintain a reference to each new `CyNode` in a map with a key of the Vertex ID. This map is required to later create `CyEdge` instances between the nodes with `CyNetwork#addEdge` because the `addEdge` method requires direct references to CyNode instances.

Visual Representation

While the CyNetwork, CyNode, and CyEdge references create the graph structure represented in a GraphModel object, the visual representation requires modifying the associated edge tables in Cytoscape to achieve labels associated with the nodes that will be rendered. We have achieved this portion of the task by retrieving the edge row from the associate table and modifying the `name` and `interaction` properties.

ii. MASS

MASS similarly requires methods that facilitate converting from MASS to a GraphModel and vice versa. This conversion is different from Cytoscape because MASS does not

have a simple type that represents all of the data held in-memory - MASS distributes its data over a cluster of nodes and further splits the data to be maintained by different threads within each node. To this end, the conversion currently requires converging all of this data into a single data structure on the control node to be sent to Cytoscape as a single package.

To converge the data in a MASS cluster, the graph implementation sends a message to all nodes, MAINTENANCE_GET_PLACES, to gather their individual VertexPlace data into a partial representational GraphModel for each node. This partial graph is then sent back to the control node with a MAINTENANCE_GET_PLACES_RESPONSE message. In the meantime the control node gathers the local data representation into its owner partial GraphModel and then waits for the remote nodes to complete via waiting for the response message from each node.

Once the control node has received the partial GraphModel from all of the remote nodes and the local model, they are merged together before finally being sent to Cytoscape as a single complete model of the graph.

The corollary response to getGraph is setGraph. When this request is received by the CytoscapeListener, the process method in its GraphRequest must read a GraphModel object from the socket's input stream.

Once the requested graph is read from the CytoscapeSocket, the listener implementation calls the setGraph method of the Graph interface. This method is implemented in GraphPlaces.

In order to convert a GraphModel into an equivalent representation in MASS, the setGraph performs two steps: first the MASS graph currently residing on the cluster must be re-initialized to an empty state, then the method repopulates the graph with those of the received GraphModel.

c. Cytoscape Plugins

Modifications to Cytoscape are made possible via OSGI plugin architecture built into Cytoscape [22][23]. OSGI, or Open Service Gateway Initiative, is a group that maintains the OSGI standard of developing modularized Java components that can be loaded at runtime [22]. The result of this project includes 2 such plugins representing each of the communication channels.

We implemented reading a graph from MASS into a Cytoscape plugin called import network and the reverse function of sending a Cytoscape network to MASS via a plugin called export network.

A Cytoscape plugin is principally built out of 3 required classes: an activator, a task factory, and a task.

The activator, a subclass of AbstractCyActivator is like the entrypoint of the plugin. The implementation overrides the start function which sets up which menu the feature should be in and passes any required services to the factory class. The final step of the CyActivator is to register the plugins task factory to the main Cytoscape application so that it can respond to events.

The factory class, a subclass of AbstractTaskFactory, is responsible for responding to actions registered in the CyActivator class. Our implementation simply passes all registered services into a new ExportNetworkTask or ImportNetworkTask respectively for each plugin.

The task, a subclass of AbstractTask, represents the core functionality of our plugins. This class implements the logic of each plugin and handles the importing and exporting of the networks. Both plugins share the basic communication logic by sending a String request to MASS and awaiting a response.

Each plugin as an initialization step in CyActivator creates a MASS submenu in the Apps menu and further adds its associate option, Import Network, or Export network.

i. Mass to Cytoscape - Import Network

The first plugin we implemented in Cytoscape was to retrieve a graph from MASS for visualizing in Cytoscape. To achieve this result, the plugin creates a menu item to import the graph under Apps -> MASS -> Import Network. By selecting this option, Cytoscape calls on the ImportNetworkTaskFactory to create an ImportNetworkTask.

The ImportNetworkTask opens a socket to localhost on port 8165, and sends a string `getGraph` using Java's ObjectOutputStream to MASS's listener to dispatch the request. From this point the listener reads the request via `ObjectInputStream#readObject`, a blocking call.

MASS receives the getGraph request from Cytoscape and calls the getGraph method on the graph field. The GraphPlaces implementation of getGraph sends out the

message to all nodes and converges the graph on the control node before sending it back to Cytoscape as described in section 3.4.2.b.ii above.

In the case of `getGraph` there is no further request payload to process so no further reads are necessary; the process is implemented as a lambda that calls the Graph interface `getGraph` function which in turn converts the MASS graph into our newly defined `GraphModel` structure. This structure is now ready to be send back to Cytoscape via the socket's output stream and `ObjectOutputStream#writeObject`.

After MASS has serialized its graph into a GraphModel and sent it to Cytoscape, the readObject call returns and the plugin continues its work. The next step

The final step of the plugin's job is to convert the GraphModel response in the CyNetwork representation of the graph with `graphToCyNetwork`. Beyond the conversion and labeling required to convert the GraphModel to a Cytoscape CyNetwork, the visualization of a graph represents much more data than what is found within the GraphModel.

To achieve a simple and consistent visualization of a graph, a few minor styles were applied to the nodes created including setting the size, colors, and shape of nodes which is illustrated in Figure 8 below. In this example, we are visualizing the hippie_current dataset [10] which contains neighbors that were not accounted for as vertices within the network.

In Figure A.4, we see vertices as large rounded blue rectangles with logical vertex id in the center. The current implementation uses a fixed size for best fit on the HIPPIE dataset. Edges appear simply as black lines between neighboring vertices. Interestingly a third type of vertex, the un-accounted neighbor vertices, appeared in this data set which we indicate with small red circles. These vertices are not labelled.

The yellow vertex in the middle of Figure 10 is the currently selected vertex. Details of this vertex are then displayed in the bottom of the window in a section called "Table Panel"

These unaccounted neighbor references that were not in the initialization of the network in MASS, hereafter referred to as non-source vertices, were only apparent when we began deserializing the GraphModel in Cytoscape. Attempting to create edges between the source vertex and destination vertex in Cytoscape failed because Cytoscape

required both vertices to exist.

We encounter this issue with the hippie dataset because the file import features exist below the maintenance features which audit vertices and edges before adding them to the graph.

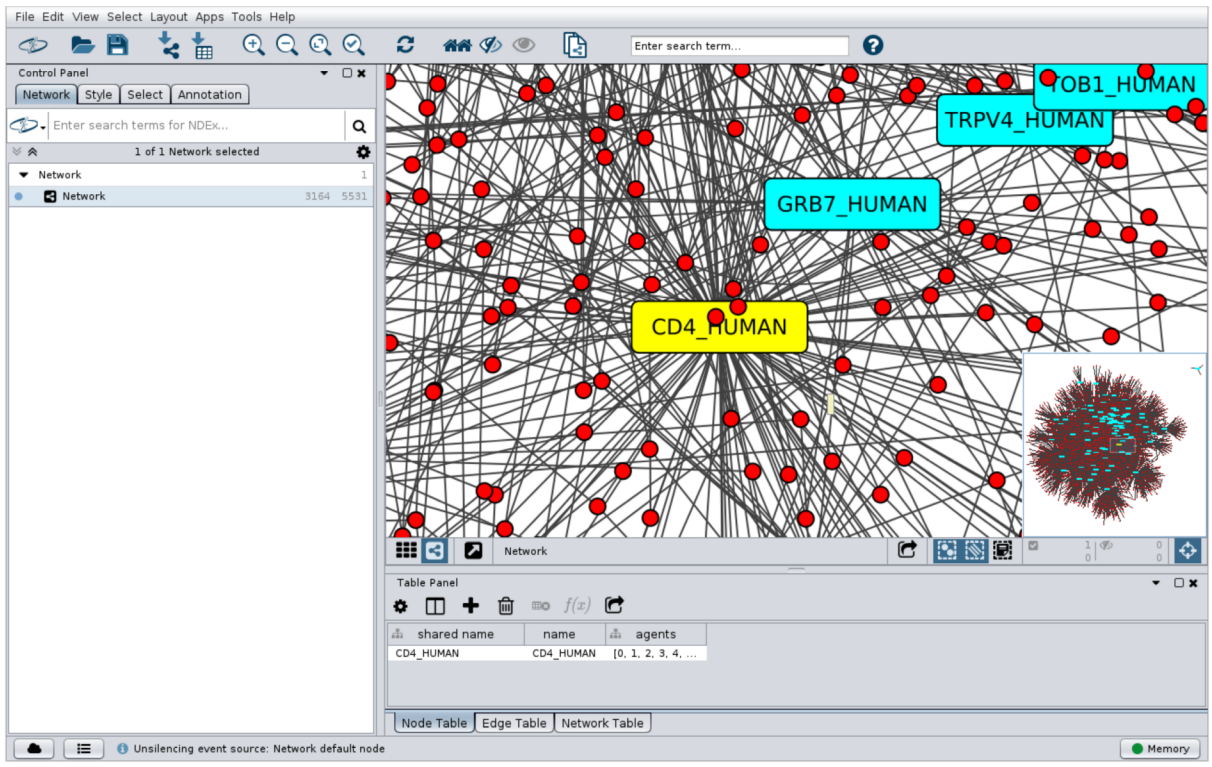


Figure A.6: HIPPIE current dataset with non-source neighbors represented as red circles

ii. Cytoscape to MASS

The export network plugin adds a second option to the MASS menu in Apps to allow sending the currently selected network in Cytoscape to MASS for processing. This plugin expectedly performs the opposite operation as the import network module. By selecting the Export Network options from the menu, the currently selected CyNetwork is converted to a GraphModel representation to be sent to MASS with a `setGraph` request.

In Cytoscape the currently selected network is indicated by a light blue highlight as seen in the left panel of Figure 8 above. In the event that no network is selected, the export network option will be disabled in the Apps menu.

The export network plugin mirrors much of the flow of the import network plugin. The CyActivator adds the Export Network option to the Apps > MASS menu and registers a new task factory: ExportNetworkTaskFactory, and a new task: ExportNetworkTask.

The difference primarily lies within ExportNetworkTask. In the case of the export network plugin, we serialize the CyNetwork in the Cytoscape plugin before sending the request off to MASS.

Unlike import network, this plugin must retrieve an existing CyNetwork which is achieved by calling a Cytoscape API function: getCurrentNetwork. This method is available through the CyApplicationManager which is retrieved in the CyActivator and passed through to ExportNetworkTaskFactory which then passes it to this ExportNetworkTask like other services.

The plugin then serializes the current CyNetwork with the cyNetworkToGraph method, described in 3.4.3.c.ii, which converts the CyNetwork to a representative GraphModel. The plugin then sends the *setGraph* request to MASS and follows up with sending the serialized GraphModel. The plugin has nothing more to do so no wait is performed in the case of *setGraph*.

Upon receiving this request from Cytoscape, the setGraph processor lambda is called which read the incoming `GraphModel` with `ObjectInputStream#readObject`. The model is then sent to MASS through the Graph interface via `Graph#setGraph`.

The GraphPlaces implementation of the Graph interface setGraph method performs 2 tasks, as mentioned earlier, to recreate the new graph. First is must delete the currently represented graph edges and vertices from memory. We call this step re-initialization. The second step is to populate the new empty graph with the one represented in the GraphModel via setGraph.

Reinitialization

From this point MASS is tasked with replacing the current in-memory across-cluster graph representation with the received graph. The first step is to reinitialize the data state of MASS locally and all remote nodes by sending a

MAINTENANCE_REINITIALIZE message to all nodes. This reinitialization invalidates on all nodes all VertexPlace object references, and resets the vertex counter for setting indices.

After the control node has completed its reinitialization it waits for an acknowledgement from all nodes confirming the entire cluster has been reinitialized, the final global step of reinitializing the distributed vertex id map is performed which completes the entire reinitialization.

Model Deployment

Once MASS has been reinitialized the target GraphModel is deployed to MASS via the Graph#addVertex and Graph#addEdge maintenance methods of the Graph interface. Similarly to converting the GraphModel to CyNetwork, all of the vertices must be created before edges are created which results in two separate loops.

Upon completion a result of “Success” is sent back to Cytoscape to confirm completion. The message “Failure” will be sent to Cytoscape upon any failure. The current plugin implementation ignores the completion message.

iii. Import Network Data Flow

The current user action implemented in the Cytoscape plugin is ‘Import Network’ in the Apps -> MASS sub-menu. This menu option kicks off the *CreateNetworkTask* implemented in the new MASS plugin for Cytoscape. Figure #9 attempts to sequence the flow of data from the user clicking “Import Network” to the resulting visualization of the MASS network. The data flow is similar for “Export Network” with slightly different data.

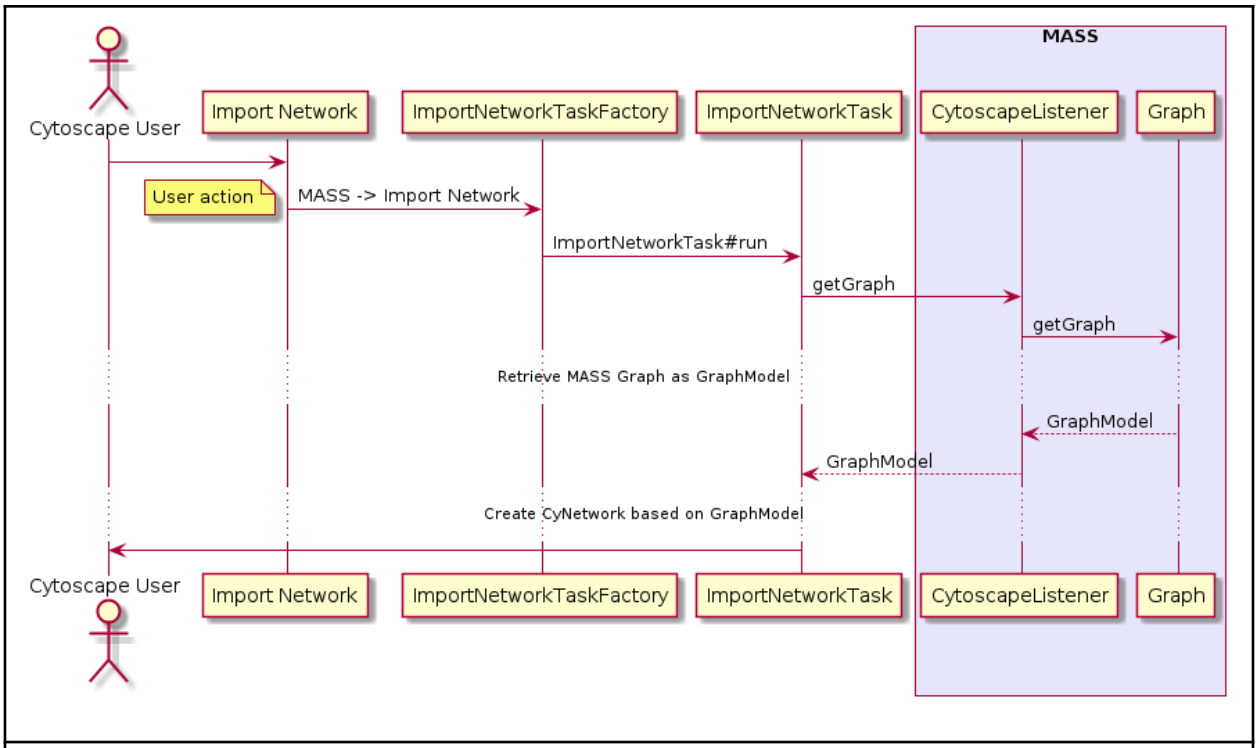


Figure A.7: Sequence diagram for importing a network into Cytoscape

The upper left actor, Cytoscape User, in Figure 11 represents user action inside of the Cytoscape application: the user enters the Apps -> MASS menu and clicks “Import Network.” This action triggers within Cytoscape and initiates ImportNetworkTaskFactory to initialize an ImportNetworkTask and calls the run method.

The ImportNetworkTask, as described above, sends a getGraph request to MASS which receives the message in the Cytoscape listener. The listener then retrieves the GraphModel from its graph field via the getGraph method. The resulting GraphModel is then sent back to Cytoscape.

Inside of the ImportNetworkTask, the process reads the resulting GraphModel from the socket and deserializes the graph into a representative CyNetwork which is then displayed to the Cytoscape User. The user can then refresh the layout from the layout menu to see the newly imported graph.

Appendix B. HIPPIE File format

Excerpt from HIPPIE_CURRENT dataset: http://cbdm-01.zdv.uni-mainz.de/~mschaefer/hippie/hippie_current.txt

AL1A1_HUMAN 216 AL1A1_HUMAN 216 0.76 experiments:in vivo,Two-hybrid;pmids:12081471,16189514,25416956;sources:HPRD,BioGRID,IntAct,MINT,I2D,Rual05
ITA7_HUMAN 3679 ACHA_HUMAN 1134 0.73 experiments:in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:10910772;sources:HPRD,BioGRID,I2D
NEB1_HUMAN 55607 ACTG_HUMAN 71 0.65 experiments:in vitro,in vivo;pmids:9362513,12052877;sources:HPRD
SRGN_HUMAN 5552 CD44_HUMAN 960 0.63 experiments:in vivo;pmids:9334256,16189514,16713569;sources:HPRD,I2D,Rual05,Lim06
PAK1_HUMAN 5058 ERBB2_HUMAN 2064 0.73 experiments:in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:9774445;sources:HPRD,BioGRID,I2D,STRING
DLG4_HUMAN 1742 ERBB2_HUMAN 2064 0.87 experiments:in vivo,Two-hybrid,Affinity Capture-Western,Co-fractionation,affinity chromatography technology;pmids:10839362,16713569;sources:HPRD,BioGRID,I2D,Lim06
P85B_HUMAN 5296 ERBB2_HUMAN 2064 0.89 experiments:in vivo,Reconstituted Complex,Biochemical Activity,protein array,pull down,enzymatic study;pmids:1334406,16273093,16729043;sources:HPRD,BioGRID,MINT,I2D,IntAct,KEGG,STRING
PTN18_HUMAN 26469 ERBB2_HUMAN 2064 0.88 experiments:in vitro,pull down,anti tag coimmunoprecipitation,x-ray crystallography,phosphatase assay;pmids:14660651,25081058;sources:HPRD,I2D,IntAct
SMUF2_HUMAN 64750 RHG05_HUMAN 394 0.88 experiments:Two-hybrid,affinity chromatography technology;pmids:15231748,28514442;species:Mus musculus (Mouse);sources:HPRD,MINT,I2D,Colland04,IntAct,BioGRID
UBX11_HUMAN 91544 ZFYV9_HUMAN 9372 0.73 experiments:Two-hybrid;pmids:16189514,15231748;sources:HPRD,MINT,I2D,Rual05,Colland04,IntAct,BioGRID
NCTR1_HUMAN 9437 CD59_HUMAN 966 0.73 experiments:in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:14635045;sources:HPRD,BioGRID,I2D
LYN_HUMAN 4067 PP1R8_HUMAN 5511 0.67 experiments:in vitro,Biochemical Activity,enzymatic study;pmids:11104670;sources:HPRD,BioGRID,I2D
NPHN_HUMAN 4868 LYN_HUMAN 4067 0.52 experiments:in vivo;pmids:12846735;sources:HPRD,I2D
DLG4_HUMAN 1742 LYN_HUMAN 4067 0.59 experiments:in vivo;pmids:9892651;species:Rattus norvegicus (Rat);sources:HPRD,I2D
BCAR1_HUMAN 9564 LYN_HUMAN 4067 0.9 experiments:in vitro,in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:9581808,9020138;species:Mus musculus (Mouse);sources:HPRD,BioGRID,HomoMINT,I2D
U119A_HUMAN 9094 LYN_HUMAN 4067 0.79 experiments:in vitro,in vivo,Reconstituted Complex,Affinity Capture-Western,affinity chromatography technology,pull down;pmids:12496276;sources:HPRD,BioGRID,I2D
TRAT1_HUMAN 50852 LYN_HUMAN 4067 0.85 experiments:in vitro,Reconstituted Complex,pull down;pmids:9687533,10790433;sources:HPRD,BioGRID,I2D
SKAP1_HUMAN 8631 LYN_HUMAN 4067 0.75 experiments:in vitro,Reconstituted Complex,pull down;pmids:9195899;sources:HPRD,BioGRID,I2D
SKAP2_HUMAN 8935 LYN_HUMAN 4067 0.8 experiments:in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:9837776;species:Mus musculus (Mouse);sources:HPRD,BioGRID,HomoMINT,I2D
LYN_HUMAN 4067 TRPV4_HUMAN 59341 0.77 experiments:in vitro,in vivo,Biochemical Activity,Affinity Capture-Western,enzymatic study,affinity chromatography technology;pmids:12538589;sources:HPRD,BioGRID,I2D
NCTR3_HUMAN 259197 CD59_HUMAN 966 0.74 experiments:in vitro,in vivo,Affinity Capture-Western,affinity chromatography technology;pmids:14635045;sources:HPRD,BioGRID,I2D

Appendix C. MatSim File Format

Taken from MatSim examples repository: https://github.com/matsim-org/matsim-libs/tree/master/examples/scenarios

```
<network>
   <nodes>
      <node id="1" x="665498.5915828889" y="6600721.058093354" ></node>
      <node id="10" x="658803.7360160261" y="6575644.324923517"></node>
      <node id="100" x="566282.7735573719" y="6454558.626888261"></node>
      <node id="10000" x="408774.9462956431" y="6233917.802557087"></node>
      <node id="10001" x="408772.02211252623" y="6234167.648466459"></node>
      ...
        </nodes>
   <links>
      ...
      <link id="pt_99992" from="pt_740057809" to="pt_740057653" length="257.27792215211286" freespeed="8.333333333333334" capacity="500.0"
permlanes="1.0" oneway="1" modes="pt" ></link>
      <link id="pt_99996" from="pt_740057599" to="pt_740057756" length="157.21884660387735" freespeed="8.333333333333334" capacity="500.0"
permlanes="1.0" oneway="1" modes="pt" ></link>
      <link id="pt_99997" from="pt_740057688" to="pt_740057647" length="207.21244492773178" freespeed="8.333333333333334" capacity="500.0"
permlanes="1.0" oneway="1" modes="pt" ></link>
      <link id="pt_99998" from="pt_740057647" to="pt_740057623" length="378.77959250564095" freespeed="8.333333333333334" capacity="500.0"
permlanes="1.0" oneway="1" modes="pt" ></link>
      <link id="pt_99999" from="pt_740057623" to="pt_740057686" length="292.07787848782823" freespeed="8.333333333333334" capacity="500.0"
permlanes="1.0" oneway="1" modes="pt" ></link>
   </links>
</network>
```

Appendix D. MASS Parallel I/O File Format

Upon request of Professor Fukuda I wrote the initial implementation of an importer for a novel file format to support loading more efficiently on multiple nodes. The initial format may change as my understanding of the expectations evolve.

The format is in plain text for now. The format starts with a header of comma separated values indicating the number of neighbors the ordinal vertex has starting with 0.

Following the header is the neighbor data in the form of an adjacency list. With 1 vertex per line starting with vertex 0. The values are output tab-separated with 10 columns per value. This results in an offset of 10 * numberOfPrecedingNeighbors + numberOfPrecedingVertices. The second term perhaps not obviously accounts for the newlines that would be encountered. The first term calculates the number of bytes represented by the lines of the preceding neighbors. This formula gives us an offset of a vertex *i* starting from the beginning of the data (after the header line.) The value of *numberOfPrecedingNeighbors* is available through the header by summing the values before the current index within the list of values. E.G: given a header line

9,9,9,9,9,9

And calculating for vertex # 3 the sum of the preceding neighbors would be 9 * 3 = 27 preceding neighbor values. With the formula above this gives us an offset of 27 * 10 + 3 = 273 byte offset where this vertex's neighbor list begins.

Here is a sample graph in the describe format. It is a complete graph with 10 vertices:

```
9,9,9,9,9,9,9,9,9,9
    1    2    3    4    5    6    7    8    9
    0    2    3    4    5    6    7    8    9
    0    1    3    4    5    6    7    8    9
    0    1    2    4    5    6    7    8    9
    0    1    2    3    5    6    7    8    9
    0    1    2    3    4    6    7    8    9
    0    1    2    3    4    5    7    8    9
```

```
0    1    2    3    4    5    6    8    9
0    1    2    3    4    5    6    7    9
0    1    2    3    4    5    6    7    8
```

Appendix E. Bug Fixes

In addition to the work proposed for and carried out for this project, we came across a number of hidden bugs in the MASS library. Both to complete our work and to improve the quality of the library, these bugs were fixed and significant bugs are documented here.

- Remote nodes do not have the same agent limit as the control node - if the max agent count is modified, it will not be reflected in remote nodes.
    - To resolve this bug we elected to read the value from the control node before initializing the workers. The control node then passes this value as a parameter for initializing the remote nodes.

Appendix F. User Manual

F.1. GraphPlaces

GraphPlaces is the high level interface to using Graph support in MASS. Client applications should use GraphPlaces instead of Places when they would like to benefit from the new graph features of MASS.

Similarly, the Place class parameter of the GraphPlaces constructor should be a class derived from VertexPlace instead of simply Place.

1.1. Details

- *GraphPlaces* - This is a client visible class used to populate the network with VertexPlaces. It is used instead of Places and takes parameters for the file input type and path. This class is responsible for distributing the vertices across the nodes and instantiated VertexPlaces with the appropriate vertexId to populate.

- *VertexPlace* - This is the client visible class to be used as the base class for individual places in the client. As such the class specified in the GraphPlaces structure for place instantiation must be a subclass of VertexPlace. It is inside of this class that the input file is read to populate the neighbors based on the input format and algorithm to use.

F.2. Graph Maintenance

Detailed in this whitepaper is the addition of graph maintenance features to the existing GraphPlaces class. These features are in the form of public methods on GraphPlaces that allow modifying the graph.

2.1. Maintenance Methods

➔ `addVertex` - add a new vertex to the graph

- ➔ removeVertex - remove an existing vertex from the graph. This is a cascading operation that will remove this vertex as a neighbor from all vertices in the graph
- ➔ addEdge - add an edge from *source* to *destination* vertex with an associated weight
- ➔ removeEdge - remove the edge between *source* and *destination* vertices

2.2. Sample Code

Below is a quick example of creating a triangle graph with the new maintenance methods of GraphPlaces

```
MASS.init();

Graph graph = new GraphPlaces(0, VertexPlace.class.getName(), 120);

final String vertexA = "A";
final String vertexB = "B";
final String vertexC = "C";

graph.addVertex(vertexA);
graph.addVertex(vertexB);
graph.addVertex(vertexC);

graph.addEdge(vertexA, vertexB, 0.9);
graph.addEdge(vertexB, vertexC, 0.9);
graph.addEdge(vertexC, vertexA, 0.9);

MASS.finish();
```

F.3. Cytoscape

Cytoscape is the graph visualization package that this project integrates with to allow visualization and modification of graphs running on MASS.

3.1. Installation

The following instructions are from the perspective of linux but should be fairly similar on windows.

1. Retrieve the desktop version of Cytoscape from cytoscape.org (the project plugins were developed with Cytoscape 3.7.2) and install
2. Pull the Cytoscape code from bitbucket:

   https://bitbucket.org/mass_utility_developers/mass_java_utilities
3. Copy the plugins to the Cytoscape plugins directory:

   ~/CytoscapeConfiguration/3/apps/installed
4. Run Cytoscape and verify the plugins are installed
   a. Navigate to Apps -> App Manager
   b. Click the “Currently Installed” tab
   c. There should be at least 2 plugins listed here: (See Figure F.1)
      i. MASS - Import Network
      ii. MASS - Export Network

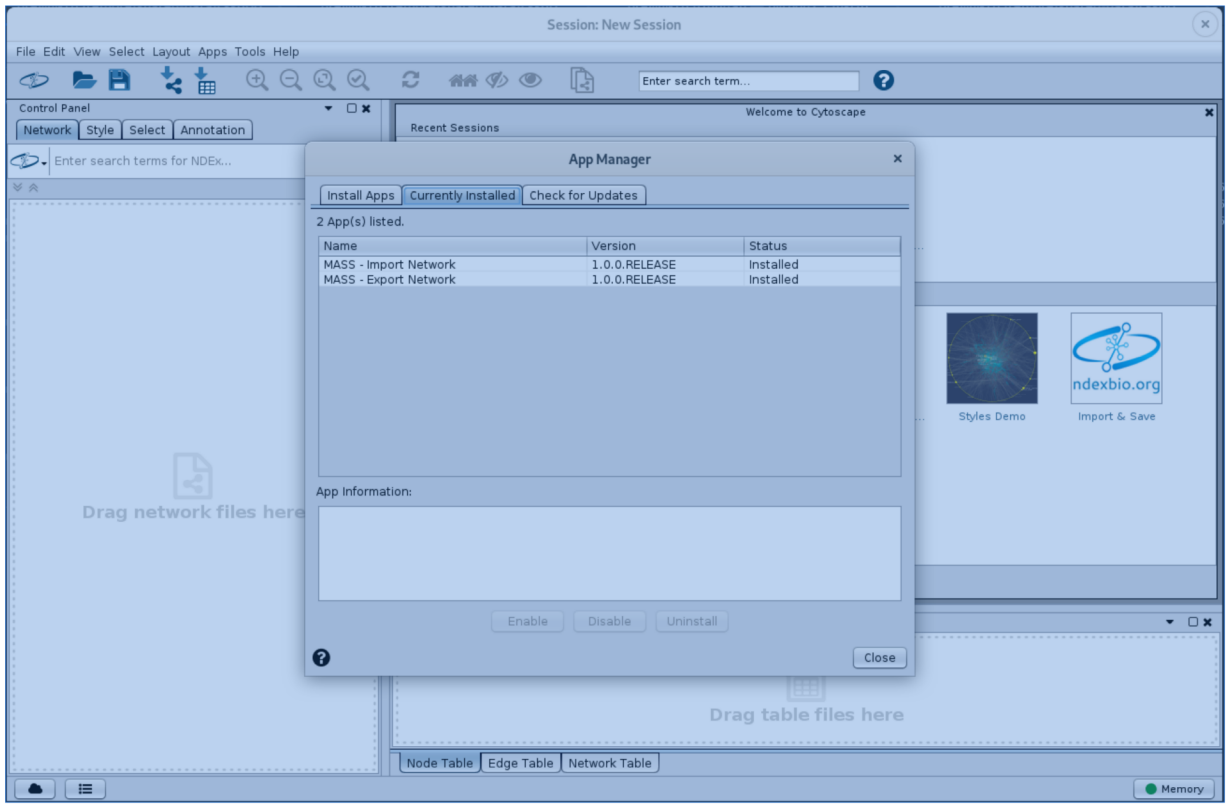


Figure F.1: Cytoscape “App Manager” with MASS plugins listed

3.2. Usage

Using Cytoscape in a MASS client application is accomplished by creating an instance of the CytoscapeListener class. The constructor for CytoscapeListener takes the Graph interface, which GraphPlaces implements, as a parameter.

At the end of your program it is advised to call the CytoscapeListener’s *finish* method to prevent the MASS program from completing execution. This will allow the CytoscapeListener thread to continue serving and processing requests to and from Cytoscape.

3.2.1. Code Snippet for CytoscapeListener

```
MASS.init();
GraphPlaces graph = new GraphPlaces(0, VertexPlace.class.getName(), 120);
MASSListener listener = new CytoscapeListener(graph);
```

```
... program code ...
listener.finish();
MASS.finish();
```

3.2.2. Cytoscape Usage

The current iteration of the Cytoscape integration resulted in two plugins: import-network, and export-network. Import network, from the perspective of Cytoscape, calls out to MASS on the localhost port 8165 and retrieves its current in-memory graph. Of particular note on this feature is that, after importing the graph from MASS, the network layout will have all nodes stacked on top of each other. This can be remedied by pressing F5 or selecting “Apply Preferred Layout” from the Layout menu.

Export network sends the currently selected network to MASS on localhost where MASS replaces the current in-memory graph with the new graph received from Cytoscape. This operation is accessed from the Apps > MASS submenu.

Detailed description of how to use Cytoscape is left up to the authors and cytoscape.org.

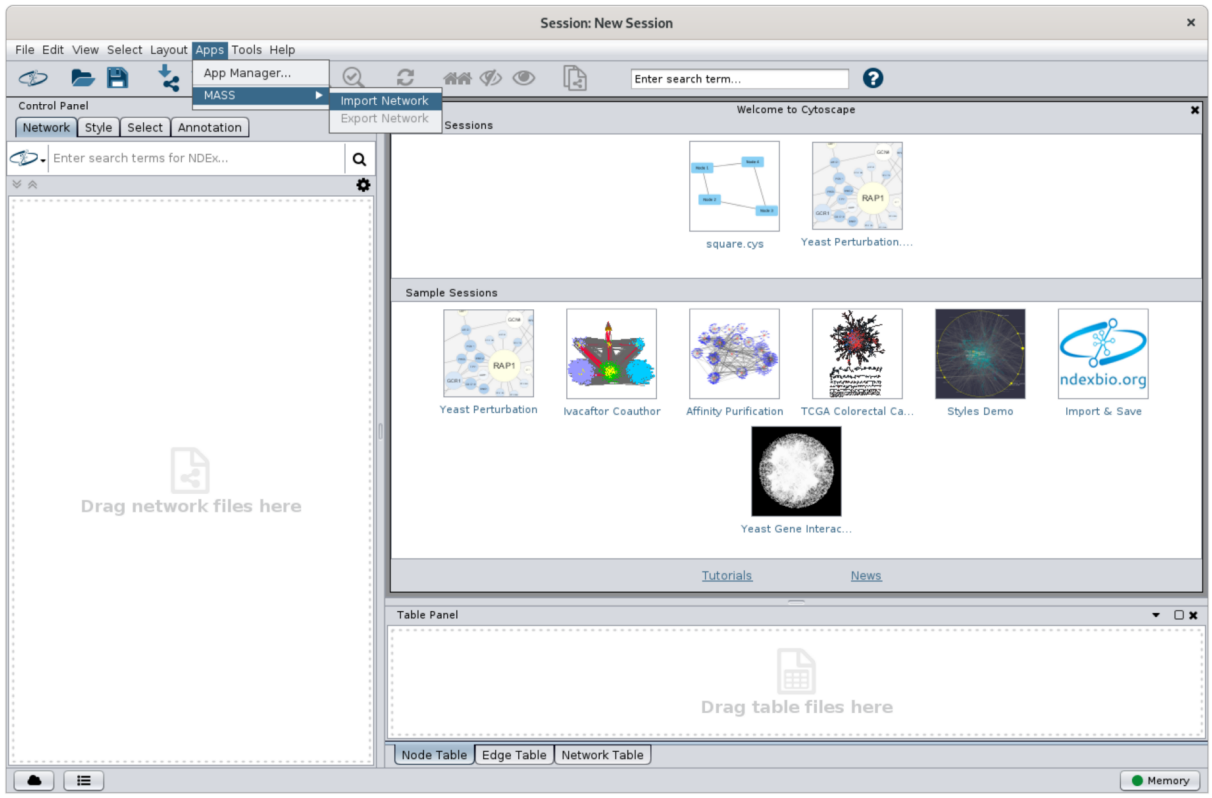


Figure F.2: The import network operation from inside of Cytoscape

3.2. Cytoscape Integration

In case you may be interested in integrating the Cytoscape plugins developed in this project with a different external framework besides MASS there are a few things to consider:

1. The plugins are currently communicating with MASS via Java's ObjectStream classes. This means that direct integration with the plugins currently would require the integration point to be in Java, short of a full reimplementation of Java object deserialization.
   a) My recommendation in the Future Works section is to convert the communication to a JSON message that would alleviate this problem and will more or less be required to integrate with MASS C++
   Alternatives are possible of course but using a well known format such as JSON would allow other systems to utilize existing JSON parsing facilities to

reduce development cost

b) The Java object serialization method can be finicky requiring the classes to be deserialized to be present in the same package as where they were sourced from.

c) It may be possible to create a client class that packages the required GraphModel and VertexModel into a jar, however, when MASS moved from Java 1.8 to Java 11 this solution no longer worked due to Cytoscape being Java 1.8 currently

2. Requests from the plugin are in the form of a simple string and implicit process
   a) “getGraph” expects a response of a GraphModel
   b) “setGraph” tells the receiver to read a GraphModel from the socket
3. The GraphModel is a simple data class to represent a graph with a list of vertices and a name. Each vertex is represented by a VertexModel class that countains an ID and a list of neighbors

Appendix G. CSV File Format

The first format we implemented for this project was a Comma-Separated Values file in the form of an adjacency list. While not explicitly called out in the proposal, this format provided a smooth entry into the work to be done in this project without requiring a lot of digging to define the format. This format also facilitated loading graphs into the implementation for testing and verification.

The format represents a graph in an adjacency list with one vertex per line. The vertex ids are implied by their ordination within the file: the first line represents a vertex with id 0 and contains a list of the ids of its neighbors.

With this structure, `PlacesBase` can calculate the number of vertices in the graph by reading the file line-by-line and counting the total number of lines as the number of vertices. MASS then instantiates a VertexPlace for each vertex it associates each with the implied id with via constructor arguments.

The VertexPlace constructor then calls init which discriminates the input file to determine input function to use for importing the neighbors for this vertex. The CSV init_neighbors function uses its ID to skip the number of lines equivalent to its ID to retrieve the line representing this vertex. From this point it is a simple matter of parsing the line for neighbor ids and associated weights if present.

Appendix H. MASS Parallel I/O

For each of these formats we have considered how best to improve the read performance across a MASS cluster. In particular we were able to devise a strategy to allow random access to the CSV format by padding each vertex edge list with trailing -1 neighbors and separating the fields with tabs instead of commas.

The proposed formats for this project, HIPPIE and MATSim do not allow such simple padding unfortunately. The flexible nature of xml limits the library's ability to predict where expected information will be. On the other hand, the HIPPIE format being a tab-separated format could be modified to be seeked randomly, however, this would require dropping the free-form text field or using a large padding section. Regardless, this modification would require a pre-processing step not ideal for our goals of making the library easy to use.

These shortcomings and the desire to improve the read performance of very large input formats led us to develop a simple yet effective input format that could be randomly seeked for reading. As the format was specifically to assist in a group member's work, it was decided to name the format after the member most benefiting from the format.

The implementation for reading the vertices from MASS's current Parallel I/O Format, or sar format, is similar to the original CSV format. Each vertex's ID attribute is implied from its ordinal position in the file: the first will have ID 0, the second having ID 1, and so on. The VertexPlace objects are created based on the count of vertices determined by the header line of the input file. Each vertex is assigned its ID in order starting from 0.

Inside of the VertexPlace class the vertex's neighbors are initialized with the help of the header. Summing up the neighbor counts of vertices preceding the current vertex, the neighbor information can be seeked directly in this format which allows more efficient parallel execution compared to reading through the entire file for each vertex as seen in other formats.

Appendix I. MASS Graph interface

This interface represents the graph interface to the MASS library for user applications.

```
public interface Graph {
   GraphModel getGraph();
   GraphModel getGraph(boolean all);

   // Graph Maintenance
   boolean addEdge(Object vertexId, Object neighborId,
                double weight);
   boolean removeEdge(Object vertexId, Object neighborId);

   int addVertex(Object vertexId);
   int addVertex(Object vertexId, Object vertexInitParam);
   boolean removeVertex(Object vertexId);

   void setGraph(GraphModel newGraph);
}
```