Additions to MASS Java – Agent Tracing and Agent Distributions

1 Introduction

The Multi-Agent Spatial Simulation (MASS) library is a parallel computing platform in which two main entities are created, Places and Agents, and are distributed over a cluster of computing nodes. The Places are maintained in an n-dimensional array, mapped to processes, and are dynamically allocated over the cluster. Agents are mapped to threads and are able to migrate from Place to Place, interacting with both Places and other Agents as they go. This is the foundation of Agent-Based Simulations within MASS. [1]

Potential applications of MASS continue to grow as new features are added. The additions of recent graduate students, Justin Gilroy and Nasser Alghamid, are of particular interest in my work this quarter and help provide the foundation for my future graduate study. Justin’s work has incorporated support for graphs and the visualization of those graph structures through the use of 3rd-party bioinformatics software, Cytoscape. While Nasser has enabled interaction with running simulations through development of the MASS Monitoring tool and integration of Java’s JShell. The goal of my work this quarter, and the remainder of my graduate program, is based on extending these new features to improve the programmer’s experience when using MASS. Ultimately, this work enables them to spend less time on adjusting MASS to fit their simulation and, instead, to spend their time and effort on solving new and interesting problems.

The features I’ve worked on this quarter and discuss in this report are: Agent Tracing and Agent Distribution. Each feature section begins with a discussion of the new functionality as well as the motivation behind its development. This leads to discussions of the feature’s implementation, including data structures and changes to the MASS library that enable the new functionality. Then we cover what the programmer will need to know to utilize the new features effectively in their applications. Finally, the limitations section covers in what ways the features’ current
implementations are lacking and discusses how these shortcomings may be addressed in my future work.

2 Agent Tracing

During a simulation one or more Agents are usually employed to traverse over the array of Places and perform some function as they go. Although sometimes the interesting part of these programs is how the Agents manipulate the Places, we often have the most interest in the path the Agents took as they traverse through the simulation space. For this reason, the Agent Tracing functionality has been developed.

Agent Tracing provides the programmer with an easy and efficient way to query the MASS library for a complete history of any agents that are registered.

2.1 Implementation

Agent History is stored in a hash table within the Place class, in which the key is the string concatenation of the Agent's class name and the Agent's identification number while the corresponding value is an ArrayList of integer pairs – each integer pair represents the time at which the agent was seen and the Place's linear index\(^1\), respectively. The signature for the agentHistory hash table is shown below:

```java
// HashTable<"AgentClass_AgentID", ArrayList<int[]>> = "Time", int[] = "PlaceLinearIndex"
private Hashtable<String, ArrayList<int[]>> agentHistory = null;
```

This structure initializes to null and is then created automatically upon registering the first Agent to trace. During execution, each Place maintains its hash table independently and is only aware of its own visiting Agents.

The “time” represented in agentHistory is maintained within each AgentsBase instance as an iteration counter that defaults to 0 upon instantiation and increments each time the `manageAll()` method is invoked. It is important to note this time is instance-specific. When reviewing the history from multiple AgentsBase instantiations, similar time values will not mean the Agents moved at the same point in the program's execution because the AgentsBase instances could have been created at different times and have a different schedule of calls to `manageAll()`.

Additionally, each time an Agent is instantiated, spawned, or migrated a call is sent to the corresponding Place to check if that Agent is being tracked and, if so, to increment that Place's agentHistory accordingly.

2.2 User Guide

At this point in development, there are three main functions supported in the Agent Tracing functionality: register, delist, and get traces. All of these functions make use of the Places’ existing `callAll()` methods and use new public variables `AGENT_TRACE_REGISTER`, `AGENT_TRACE_DELIST`,

\(^1\) The “linear index” is a one-dimensional representation of the n-dimensional array of Places. This is used to simplify management of the Place array and allows for consistent reference between applications using differing number of dimensions.
and AGENT_TRACE_GET as the parameter function IDs. Note that these new variables use values in the negative integer range and, therefore, should not conflict with any user-defined method IDs.

**Registering new agents** can be done with iterative calls similar to the function below. Here we are adding tracing for the first agent (ID = 0) of the Agent class.

| Single | places.callAll( places.AGENT_TRACE_REGISTER, ( Object ) Agent.class.getName() + ",0"); |

If working on a *single node*, the user also has the ability to add an array of agents with a single call. (As will be discussed in more detail in the next section, this is limited to a single node due to current limitations on the size of the message that can be passed between the computing nodes.)

| Multiple (Single Node) | Object[] agentsToTrace = new Object[] { Agent.class.getName() + ",0",
Agent.class.getName() + ",1",
Agent.class.getName() + ",2";
places.callAll( places.AGENT_TRACE_REGISTER, agentsToTrace); |

These calls do not currently return a value to indicate success or proper formatting, so it is on the programmer to ensure additions are correct. Further, the programmer is able to successfully register improperly formatted agent strings or agents whose ID numbers are beyond the scope of the application, though nothing will be recorded during the program's execution.

**Delisting agents** is accomplished in the same fashion as registering new agents, the only difference being the parameter function ID.

| Single | places.callAll( places.AGENT_TRACE_DELIST, ( Object ) Agent.class.getName() + ",0"); |

| Multiple (Single Node) | Object[] agentsToTrace = new Object[] { Agent.class.getName() + ",0",
Agent.class.getName() + ",1",
Agent.class.getName() + ",2";
places.callAll( places.AGENT_TRACE_DELIST, agentsToTrace); |

If the programmer attempts to delist an agent that was never registered, then the call is sent and ignored by each Place.

**Getting the results** is a little different than registering and delisting in that the `callAll()` function must collect the results from each computing node and return those results to the program in the form of an Object array. To accomplish this, the programmer must provide an Object array to use as the second parameter and another Object array to catch the results. As shown below, the programmer has the option of querying for all results or for a single agent:

| Single | Object[] agentsCallAllObjects = new Object[] { Agent.class.getName() + ",0"; Object[] agentTraceResults = ( Object[] ) places.callAll( places.AGENT_TRACE_GET, agentsCallAllObjects); |

| All | Object[] agentsCallAllObjs = new Object[1]; Object[] agentTraceResults = ( Object[] ) places.callAll( places.AGENT_TRACE_GET, agentsCallAllObjs); |

When requesting all results, the Object array passed in the parameter may be of any size – as long as the first position is `null` then all agent history results will be returned.
**View the results** by first casting the output from the AGENT_TRACE_GET call back to the native agent history structure. The full history will always be in the first position (e.g., agentTraceResults[0]) of the returned array and the rest of the array will be null. The following code snippet demonstrates this conversion as well as use of the resultant hash table to print all histories for Agents that were recorded by one or more Places during program execution:

```java
Hashtable<String, ArrayList<int[]>> results = (Hashtable<String, ArrayList<int[]>>) agentTraceResults[0];
set<String> keys = results.keySet();
for (String key : keys) {
    if (results.get(key).size() > 0) {
        System.out.println("Agent Trace Results: * + key + * :: " + Arrays.deepToString(results.get(key).toArray()));
    }
}
```

Note that the `Arrays.deepToString()` method, or a similar method from another library, must be used to reach the actual values. Otherwise, the `toArray()` method will only print the memory addresses of the integer pair arrays.

Finally, the result will be similar to output below – taken from a trial run of the Triangle Counting benchmark application:

```
Agent Trace results: edu.ubc.bctt.cs6.11.111.111.Tringles.Crawler_743 :: [0, 743], [2, 820], [4, 1049]
Agent Trace Results: edu.ubc.bctt.cs6.11.111.111.Tringles.Crawler_742 :: [0, 742], [2, 95], [4, 10]1
Agent Trace Results: edu.ubc.bctt.cs6.11.111.111.Tringles.Crawler_741 :: [0, 741], [2, 588], [4, 524], [6, 7411]
Agent Trace Results: edu.ubc.bctt.cs6.11.111.111.Tringles.Crawler_740 :: [0, 740], [2, 55], [4, 67], [6, 7491]
```

For reference, the Triangle Counting application uses a three-step process to determine how many triangles are present in the simulated graph. The Agents are first instantiated on the Places, each Agent then moves twice to Places with lower values and, finally, attempts to return to their original Place. If successful, the Agent has found a triangle. In the above execution, we can see that Crawlers 740 and 741 successfully found triangles.

### 2.3 Performance

The following trials were completed using the Triangle Counting benchmark application and represent the times taken to register, delist, and get the history for 100 Agents instantiated on an array of 500 Places. Each value is an average of 5 trials with that configuration. These tests controlled for the number of nodes to determine if the size of the cluster has any effect on performance.

**Registering (100 Agents)**

<table>
<thead>
<tr>
<th>Node Count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Add (ms)</td>
<td>18</td>
<td>150.6</td>
<td>156.7</td>
<td>155.7</td>
</tr>
<tr>
<td>Group Add (ms)</td>
<td>28</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Delisting (100 Agents)**

<table>
<thead>
<tr>
<th>Node Count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Remove (ms)</td>
<td>37.4</td>
<td>113</td>
<td>104.5</td>
<td>113.7</td>
</tr>
<tr>
<td>Group Remove (ms)</td>
<td>22.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Shown above, a clear pattern emerges with two key takeaways. First, expectedly, the performance is significantly better on a single, local node. Second, the performance is consistent
regardless of the number of remote nodes used in the simulation because the process to add or remove is $O(n)$ and all processes are synchronized after each `callAll()` invocation. Both of these observations help illustrate the additional overhead of communicating over the network.

### Get Traces

<table>
<thead>
<tr>
<th>Node Count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get All (ms)</td>
<td>101.8</td>
<td>112.8</td>
<td>166</td>
<td>133.3</td>
</tr>
<tr>
<td>Get Single (ms)</td>
<td>48</td>
<td>341</td>
<td>316</td>
<td>415.3</td>
</tr>
</tbody>
</table>

When submitting a request for the agent history to be collected and returned to the program there are some interesting aspects to consider. As shown above, on a single node, the results are as you would expect – getting a single item takes less time than returning all items. However, when working with multiple nodes the inverse is true.

First, the performance of requesting a single item deteriorates when using multiple nodes because the item being requested is converted to a byte array by the master node before transmission and then must be decoded by the receiving remote node. Conversely, there is much less overhead when only sending a null parameter.

Intuitively, however, the performance of requesting a single entry will remain relatively consistent regardless of the program while the performance of requesting all records will be positively correlated with the number of agents tracked and the number of Agent migrations. The more agent history captured, the longer it will take to retrieve all records. This is in large part due to the cleaning and organizing automatically done by the master node when entries are returned. (See Appendix A: `cleanAgentHistory()` code snippet for the current code.)

### 2.4 Limitations and Future Work

**Other formats are needed for querying the results.** Current implementation only allows for pulling by agent. However, there will likely be times, such as when preparing data for visualization in Cytoscape, that we will need to find the history by Place and not by Agent. For example, when trying to show the Agents that are currently or have been on a specific Place in the visualization – the current results format would require the entire agent history be pulled and then manually reoriented by the programmer.

A simple solution to this may be to expand the `cleanAgentHistory()` method and utilize additional positions in the returned object array (i.e., array[0] is the agent-oriented view and array[1] is the place oriented view). This would add some additional processing time, but improve overall flexibility.

**Functionality to add a group of agents in one `callAll()` is limited to a single node due to the number of allowable bytes in the java.io.ObjectInputStream used in the underlaying MProcess class's inter-process communication.** By default, the `callAll()` uses the MProcess class for communicating with the remote computing nodes, but there may be an opportunity to utilize functionality similar to how the remote nodes return their values to the master node.

**Return user-define values instead of, or in addition to, the Place's linear index.** In an effort to allow this functionality to replace the need for custom implementations, it may also be necessary to allow for variations of the integer pairs. As a reminder, in the current implementation the pairs
represent the current iteration, “time”, and the Place being visited, in the form of its “linear Index”. However, some programs may find a different value more interesting when assessing the results or correctness of their simulations. For example, in the Triangle Counting benchmark the agents are required to move to places with lower values and then to return home. In this case, it may be useful to know what those values are to determine if the program is working as expected.

To allow for this expansion, a new method may be required. By default, this method would query MASS base for the linear index and return that value. However, the programmer would have the ability to override this method and return a value of their choosing instead.

3 Agent Distribution

When designing agent-based simulations in MASS, one of the first challenges for the programmer is to determine where the Agents are initially placed on the array, or graph, of available Places. To this end, however, the MASS Java library defaults to filling the Places sequentially and does not have other options available without requiring the programmer to write their own custom implementation.

The objective of this Agent Distribution functionality is to make some of the most common distributions available by default in MASS Java. Ideally, this should help reduce the need for custom implementations and, again, allow the programmer to focus on the more interesting parts of their simulation.

3.1 Implementation

The key to this implementation is the map() method in the Agent class. This method takes in the number of Agents that are being populated, the size of the Places array, and the index of a particular Place in the array for which to determine how many Agents need to be populated. See the signature below:

```java
public int map( int initPopulation, int[] size, int[] index )
```

This method is then called iteratively by AgentsBase whenever a new set of Agents is initialized. The fact that this method is contained in the Agents class means that the programmer is easily able to override the method when extending the Agent class for their implementation.

Previously, this method dynamically calculated the number of agents to create each time it was called. However, to enable custom distributions, this methodology had to be changed. Fundamentally, there needed to be a way for the method to know the location of all other allocated Agents and not just those on the Place being queried. For this, a new method has been created, `setMapDistribution()`.

The `setMapDistribution()` method contains all logic required for the new distributions. However, instead of simply returning the number of Agents required for a single place, the method saves the output in the DistMap array. The DistMap array is one-dimensional and simply holds the number of Agents to be mapped to each Place such that each index in the array represents the linear index of a Place in the Places array.
Now, when the `map()` method is called, it first checks to see if a DistMap has been created for the given population and Places array. If not, then a default sequentially filled map is generated. Finally, it returns the value from the DistMap for the parameter index Place. This also means that existing applications will not need to be reworked as a result of the added functionality.

### 3.2 User Guide

At this point in development, four distributions are available for use: SEQUENTIAL, RANDOM, RANDOM_EVEN, and NORMAL_STANDARD. The following is a simple illustration of each of these distributions on a 5-by-5 grid of Places with 15 agents allocated:

![sequential](sequential.png) ![random](random.png) ![random_even](random_even.png) ![normal_standard](normal_standard.png)

The only difference between the RANDOM and RANDOM_EVEN distribution is that the latter ensures that all places have the same number of agents before allowing additional agents to be populated on a Place.

The NORMAL_STANDARD option creates a standard normal distribution where the mean is the middle of the array and the standard deviation of the curve for each dimension is equal to 1.

**Using these distributions** is done by adding additional parameters to the Agents constructor: the desired distribution, which is stored in the Agent class as an enumeration, and an optional seed value. The following constructors illustrate the options available and the default values if a specific distribution or seed is not requested:

```java
public Agents(int handle, String className, Object argument, Places places, int initPopulation) {
    this(handle, className, argument, places, initPopulation, Agent.DISTRIBUTION.SEQUENTIAL, System.currentTimeMillis());
}

public Agents(int handle, String className, Object argument, Places places, int initPopulation, Agent.DISTRIBUTION dist) {
    this(handle, className, argument, places, initPopulation, dist, System.currentTimeMillis());
}

public Agents(int handle, String className, Object argument, Places places, int initPopulation, Agent.DISTRIBUTION dist, long seed) {
    super(handle, className, argument, places.getHandle(), initPopulation, dist, seed);
}
```

So, the user code will look something like this:

```java
Agents agents = new Agents(1, Nomad.class.getName(), null, places, x * y, Nomad.DISTRIBUTION.RANDOM, 0);
```

With this call a new set of agents of the Nomad class have been initialized randomly over the Places array and the seed used in the random number generator was 0.
3.3 Limitations and Future Work

Additional flexibility is required for this functionality to replace the need for programmers to override the default options in their simulations. For example, the current implementation of NORMAL_STANDARD assumes that the distribution needs to be centered in the Place array, that all available places should be used in the distribution, and that only one distribution is needed. In simulations like the Sugarscape benchmark application, all three of these assumptions are incorrect for the need (i.e., two smaller non-overlapping distributions are needed to simulate separate mounds of sugar).

To resolve this issue, I plan to add a third argument to the constructor in the form of an Object array. This approach is ideal because different distributions are going to require different tuning parameters and each parameter may take a different format. This creates great flexibility while not having to create too many overloaded Agents constructors.

Additional distributions are also needed to address a greater range of possible simulations. I'm currently planning on adding support for multivariate normal distribution and Poisson distribution options though this list may also grow as my graduate work continues.

Apache Commons contains support for calculating these distributions and may be helpful when creating these distributions in MASS. [2][3]

 Modifications are needed to support Graph and MatSim applications. The distributions at this point have been designed around the use of an n-dimensional array in which the relationship between each Place is uniform (e.g., the distance between Place A and Place B is equal to the distance between Place B and Place C if they are adjacent to one another in the Place array). However, in Graphs and MatSim applications the relationship between each Place is more complex. This will not affect distributions like RANDOM, but will significantly affect distributions like NORMAL_STANDARD in which the probability of an Agent being populated on a Place is relative to the probability of the Agent being populated on the neighboring Places.

4  Looking Forward

Summer – Volunteer Work

My plan for the summer break is to address the current limitations of the two features discussed in this paper. My goal is to have these features complete and merged into the “develop” branch in Bitbucket by the time classes resume in the Fall.

Fall and Winter – CSS 595

In the Fall, I plan to use the functionality discussed in this paper and begin my work extending and integrating the Graph and InMASS features developed by Justin and Nasser. To begin, I'll work on the Cytoscape integration and visualizing the Agents traversing over the graphs. This will likely require some simultaneous extension of the InMASS functionality to allow the user to walk through and view the Agent migrations at each step during program execution.
5 References


6 Appendix

Appendix A: cleanAgentHistory() function