Automated Agent Migration Over Structured Data

Vishnu Mohan

A report

submitted in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2022

Reading Committee:

Professor Munehiro Fukuda, Chair

Professor Robert Dimpsey

Professor Michael Stiber

Program Authorized to Offer Degree:

Masters of Science

University of Washington

Abstract

Automated Agent Migration Over Structured Data

Vishnu Mohan

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
Computer Science and Software Engineering

Agent-based data discovery and analysis views big-data computing as the results of agent interactions over the data. It performs better onto a structured dataset by keeping the structure in memory and moving agents over the space. The key is how to automate agent migration that should simplify scientists' data analysis. We implemented this navigational feature in multi-agent spatial simulation (MASS) library. First, this paper presents eight automatic agent navigation functions, each we identified, designed, and implemented in MASS Java. Second, we present the performance improvements made to existing agent lifecycle management functions that migrate, spawn and terminate agents. Third, we measure the execution performance and programmability of the new navigational functions in comparison to the previous agent navigation. The performance evaluation shows that the overall latency of benchmark applications improved with the new functions. Programmability evaluation shows that new implementations reduced user line of codes (LOC), made the code more intuitive and semantically closer to the original algorithm. The project successfully carried out two goals: (1) design and implement automatic agent navigation functions and (2) make performance improvements to the current agent lifecycle management functions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

1.1 BACKGROUND

Big Data computing came to prominence more than a decade ago. It utilized a cluster of computing nodes to analyze a large number of data and to draw meaningful conclusions from the data within a reasonable time. Popular tools such as Map Reduce [8] and SPARK [7] have emerged over time and provides customers a framework for analyzing big data. The programming frameworks provided by these tools attracts physical scientists analyzing structured data. For all these frameworks, data is flattened into a stream, broken to partitions and fed through multi-threaded analyzing units. To achieve parallelization, multiple analyzing units spin up and execute in parallel in multi-threaded fashion over a cluster of computing nodes. However, these tools cannot easily analyze structured in-memory datasets [1] such as graphs and requires the data to be flattened into data streams. The new tools such as SciHadoop and GraphX facilities have emerged that supports analysis of structured data but doesn't allow incremental modification to data and support visualization [2]. Agent-based data discovery approach [1] that uses an analyzing unit to navigate over a distributed dataset, collects and modifies the information can be an easier alternative for analysis of structured data.

Agent-based modelling and simulation (ABMS) is an approach to modelling complex systems, based on interaction among autonomous agents that are self-contained with a set of attributes and behaviors and its own decision-making capabilities. Advances in computational capabilities has resulted in development of agent-based models across a spectrum of domains from stock market, supply chains, consumer markets, and to predict the spread of pandemics [3]. While ABMs have been used primarily to model complex systems and observe collective behavior, they can also be used for data hunting and discovery.

Multi-Agent Spatial Simulation (MASS) [18] is a parallel computing library for multi-agent and spatial simulation over a cluster of computing nodes. MASS follows a data discovery methodology for analysis and discovery of big data. MASS applies agent-based modeling for

structured data analysis where agents navigate through the data-structure and find its attributes/shapes using emergent collective agent behavior.

1.2 MOTIVATION

Prior to my work, MASS users wrote application specific custom agent navigation functions to achieve their data analysis goals. Users write *OnArrival* and *OnDeparture* functions that determine the agent behavior and decision making based on the application and structured data being analyzed. These custom functions utilize basic functions - *Migrate (), Spawn (), Kill ()* - built in MASS to manage the agent navigation and lifecycle. For example, to perform Breadth First Search (BFS) on a graph, a user will write custom agent navigation function to move the agent to the source vertex, to determine the neighboring vertices, to move the agent to one of the neighbors, to spawn children, and to migrate the child agents to the remaining neighbors. Writing code to manage this agent navigation is cumbersome for users who are primarily focused on data analysis. Physical scientists who perform big data analysis needs simple programming frameworks to support their analysis that that can be used out of the box.

My research seeks for generalizing agent navigational patterns used to analyze structured data, proposes optimal design, and implements out the box functions to support the agent navigational patterns in MASS Java, while making improvements to the existing agent lifecycle management functions. As a part of my research I analyzed benchmark programs to identify common agent navigational patterns in MASS and proposed functions for supporting the navigational patterns.

1.3 PROJECT GOALS

The goal of this project is to identify, design and implement automated agent navigation to perform analysis on structured data. The auto-agent navigation methods were then measured for overall performance and programmability. The strategy we adopted to perform was:

1) **Identify generalized agent navigational patterns to analyze structured data.**

   Agent based data analysis over structured data is superior to data streaming based methods. We chose the following benchmark programs as they involve analysis of graphs or are computational geometry problems have been implemented in MASS:

   a. Breadth First Search
   b. Triangle Counting
   c. Range Search
   d. Closest pair or points in Space
   e. Voronoi Diagram in Space
   f. Convex Hull
   g. Closest pair of points using Quad tree
   h. Voronoi Diagram using Quad tree
   i. Ant colony optimization (ACO)

2) **Design and implement automated agent navigation.**

   For each of the identified generalized agent navigational pattern, we designed and implemented eight new navigational functions in MASS java.

3) **Improve the existing agent lifecycle management functions to improve performance.**

   MASS currently has basic built-in functions - *Migrate (), Spawn (), Kill (), and manageall ()* – that support agent navigation and lifecycle management. *Migrate (), Spawn (), and Kill ()* set attributes on agents to migrate to different location, spawn children and terminate agents, while *manageAll()* commits the life cycle management. These functions have inefficiencies that significantly impact the performance of applications when executed

over large datasets. We made improvements to these functions to improve the overall performance of the applications.

As a part of our work, we measured the overall performance improvements by comparing the overall latency for analyzing large datasets and measured programmability of the new methods using both quantitative and qualitative measurements. We used overall reduction in user lines of code (LOC) for quantitative measurements while we used the semantic gap between original algorithms and their MASS implementation for qualitative measurements.

2 RELATED WORK

This section compares the MASS prior to my work, with two-related ABM systems: Netlogo and Repast Simphony from the view point of agent migration. At the end, we clarify their programing challenge which is our motivation for automating agent migration.

Netlogo and Repast Simphony are two agent-based modelling tools that provides users the ability to model complex systems. Modelers give behavioral instructions and navigational inputs to agents which determines the agent behavior with the environment and with other agents [4]. This makes it possible to explore relationships between micro-level behavior of individual agents with macro-level patterns.

Netlogo supports agent migration in a 2D continuous space and provides pre-defined behaviors to agents which can be utilized by the modelers [14]. It includes migration functions such as FORWARD, BACKWARD, RIGHT, LEFT, HATCH, DIE, JUMP and MOVE-TO [13]. FORWARD/BACKWARD enables agents to move forward and backward from its current position in the environment. RIGHT/LEFT enables agents to change the direction of movement. DIE terminates agents and removes it from the environment and HATCH spawn new agents that inherits properties from its parent agent. MOVE-TO moves agents to a specified x and y coordinate.

Repast Simphony also provides pre-defined functions that support agent migration and behavior. This includes agent migration functions [9][10][11][12] such as moveByDisplacement, moveByVector, moveTo, VNContains, and MooreContains. MoveByDisplacement moves agents from its current location by a specified amount. MoveByVector moves agents by a specified distance from its current position along a specified angle. MoveTo moves agents from its current location to a new location specified as input. VNContains determines whether or not a particular agent is in the Von Neumann[1] neighborhood of a particular source. MooreContains determines whether or not a particular agent is in the Moore neighborhood of a particular source.

Both Netlogo and Repast Simphony provides basic agent migration methods but doesn't provide functions that support complex agent propagation and migration. To perform complex agent propagation, users have to write custom functions utilizing the basic predefined functions. Knowledge of these limitations motivated us to upgrade MASS, our own agent-based modelling library to support complex agent migration and propagation out of the box.

MASS [17] has two main entities that are used for data analysis - *Agents* and *Places*. MASS users represent their dataset across the places which is then analyzed using agents. MASS currently supports basic agent-navigation and life-cycle management functions. *Migrate ()* function moves agents to a specified place index. *Spawn ()* creates new agents that inherit properties from its parent agent and are spawned at the same place as parents. *Kill ()* function terminates agents and removes it from the environment. Users require significant programming capabilities to successfully perform data analysis. Users have to build custom agent migration functions that would use these basic functions. Inability to support complex agent navigational functions natively in MASS gives big burden to physical scientists who hopes to conduct big data computing using agent-based modelling.

[1] Appendix A provides details of the Van-Neumann and Moore's Neighborhood

3 DESCRIPTIVE ENHANCEMENT OF AGENT MIGRATION

3.1 MASS LIBRARY

MASS contains two main classes - *Places* and *Agents*. Places is a multi-dimensional array of place objects allocated over a cluster of nodes, each having a globally unique array index. Each place can host agents and is capable of exchanging information with other places in the system. Agents are analyzing units that reside in a place and navigates across places distributed over a cluster of nodes. Parallelization is achieved by multi-threaded MASS processes that execute on agents and places, and run across on cluster of nodes, communicating using Java Secure Channel connected by Transmission Control Protocol (TCP) sockets.

Figure 1 shows the architecture of MASS across multiple computing nodes with each node containing multiple threads to perform the data analysis. The number of threads in a computing node relies on the number of CPU cores.

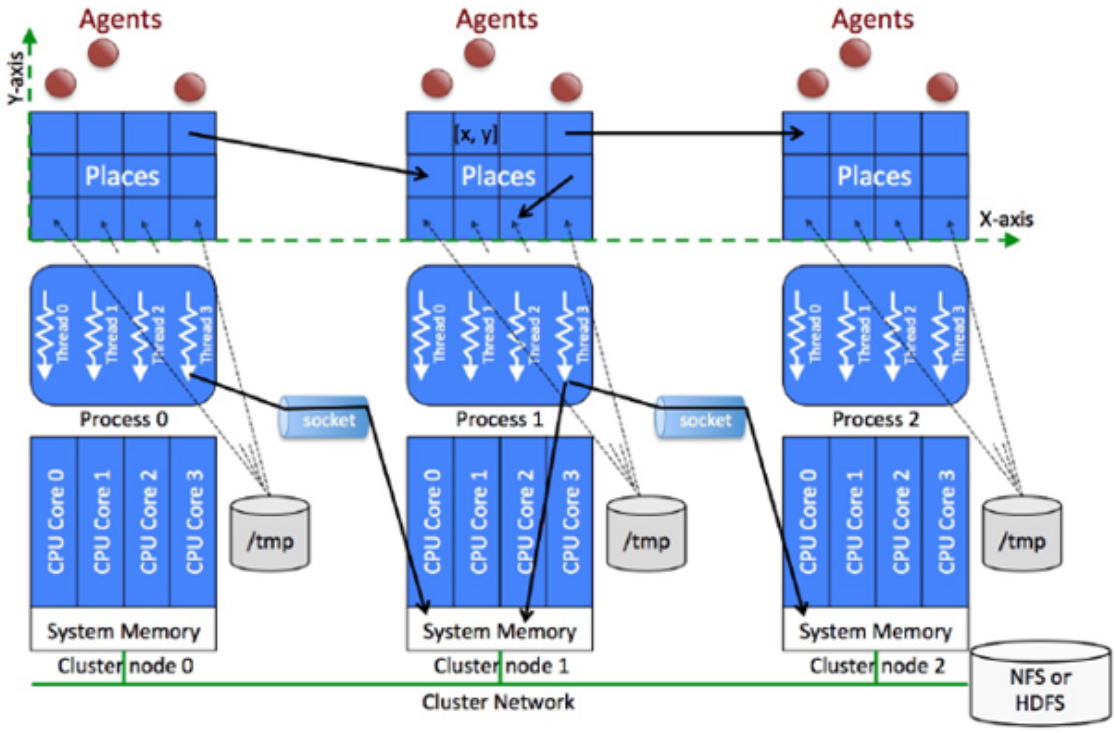

Figure 1: Mass Library Architecture

Listing 1 shows the control flow of a user program using MASS. The user program calls *MASS.init()* to start the MASS processes across a cluster of nodes. It then creates place objects across the computing nodes, where places are assigned a global index. It then creates agents with, each agent assigned to a place and a bag of agents. The user program can then implement algorithms using the built-in functions such as: *Places.callAll(func)* for concurrently invoking a function on all places; *Place.exchangeAll(func)* for each place to collect data from its neighbors; *Agents.callAll (func)* for concurrently invoking a function on all agents in the system; *Agents.manageAll()* to commit spawning child agents, terminate agents and migrate agents from one place to another.

```
public static void main( String[] args ) {
    MASS.init();
    Places network = new Places( 1, Node.class.getName( ), null, nNodes );
    network.callAll( Node.init_ );
    Agents crawler = new Agents( 2, Crawler.class.getName( ), ( Object )args2agents, network, 1 );

    for (int time = 0; nAgents > 0; time++)
    {
        crawler.callAll(Crawler.departure_);
        crawler.ManageAll();
        crawler.callAll(Crawler.onArrival_);
        crawler.ManageAll();
    }

    MASS.finish( );
}
```

Listing 1: User Program

Figure 2 shows the main Java classes that constitute MASS. *Agents* and *AgentsBase* classes represent a collection of agents. The *Agents* Class is the interface to a user program for creating and manipulating the Agents while the AgentsBase class contains the implementation details for creating agents, *callAll (),* and *Manageall (). Places* and *PlacesBase* classes represent a collection of place objects over computing nodes. The *Places* Class is the interface to a user program while the *PlacesBase* class contains the implementation details for creating places, *callAll (),* and *Exchangeall ()*. The *MASS* class is responsible for construction and deconstruction of the computing cluster and has references to all places and agents within the cluster. The *MProcess* class is responsible for message passing between remote and master computing nodes.

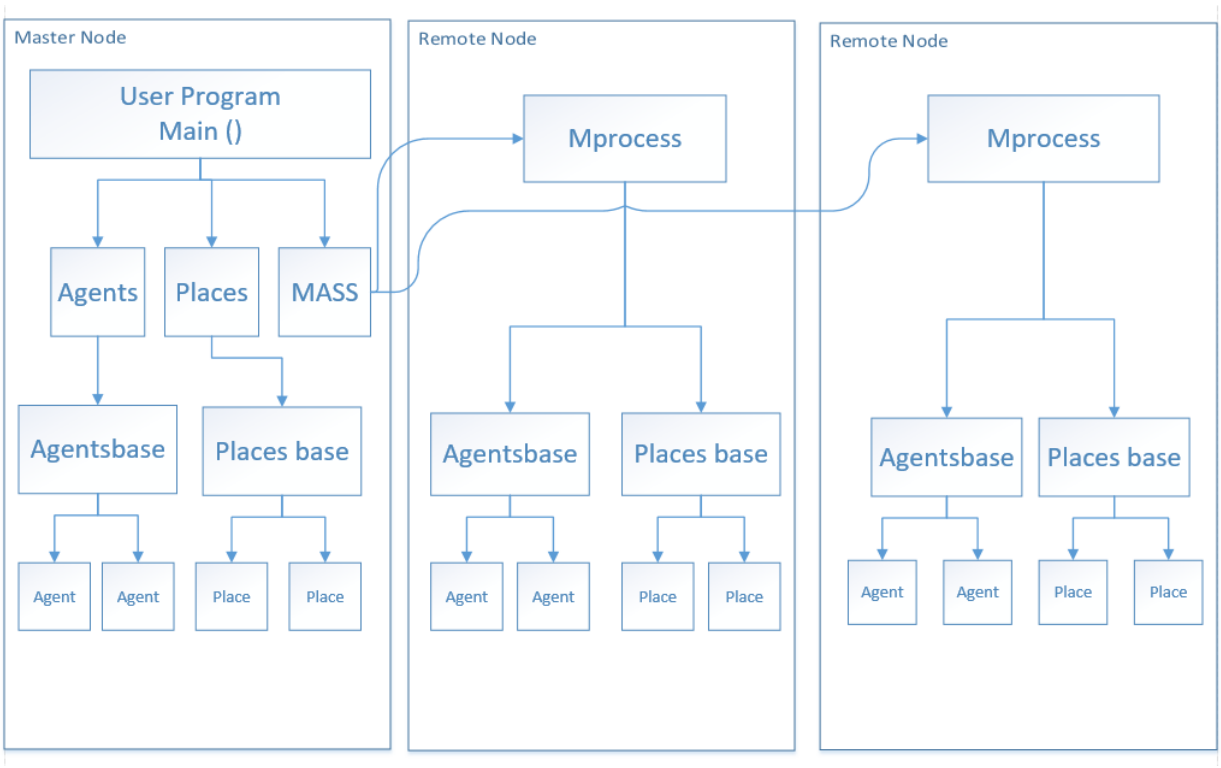

Figure 2: Class Structure of MASS

3.2 High Level Agent Migration Function

Currently MASS users write custom agent navigation functions to implement algorithms required for analysis. Custom functions use basic built-in MASS functions - *Migrate (), Spawn (),* and *Kill ()* – to manage agent navigation. A user program invokes the custom functions from the main program using *callAll (function)*. This puts a big burden on the users requiring their significant programming capabilities to build custom agent navigational functions and successfully perform data analysis. In this paper we identify eight agent navigation patterns used in benchmark programs and develop built-in MASS functions to support these navigational patterns.

3.2.1 MigratePropagate

MigratePropagate (see figure 3) performs breadth first search of a graph. The function fetches all the neighboring vertices associated to a vertex where the agent is located, excludes the

previous vertex from where the agent migrated from, moves the agent to one of the neighbors, spawn child agents and moves them to the remaining neighbors. The function will manage the agent traversal through all the nodes in the graph with minimum latency.

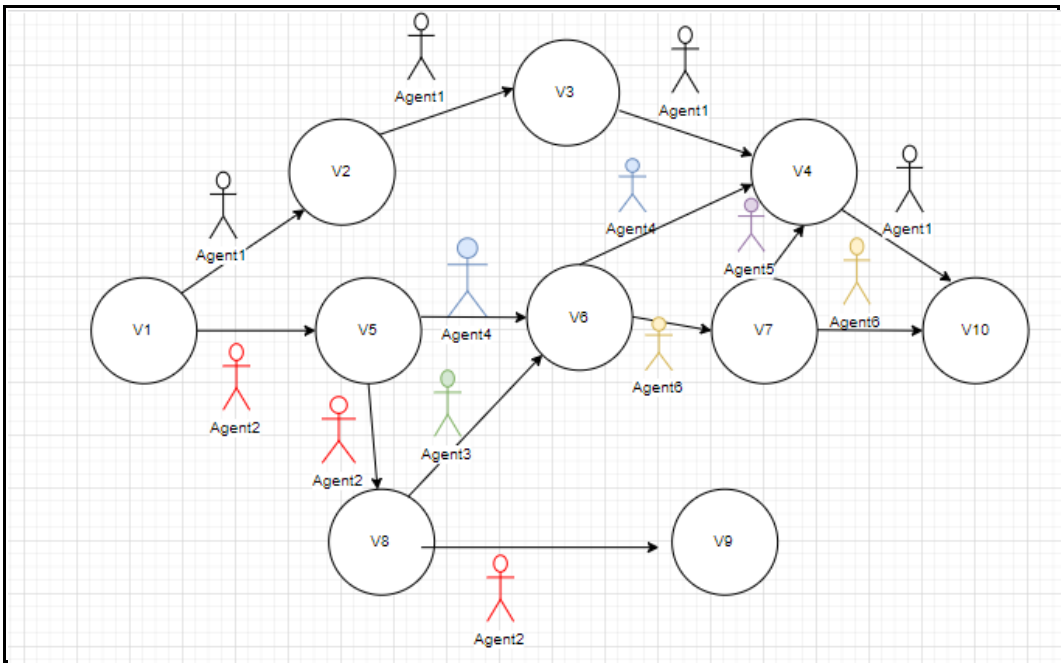

Figure 3: MigratePropagate

3.2.2 MigratePropagateDownstream

MigratePropagateDownstream (see figure 4) performs downstream propagation of an agent on a graph. The function fetches all the neighboring vertices which are downstream (lower index values), moves the agent to one of the neighbors, spawn child agents and dispatches child agents to the remaining neighbors. *MigratePropagateDownstream* will support user algorithms to perform triangle counting and connected component counting on a graph.

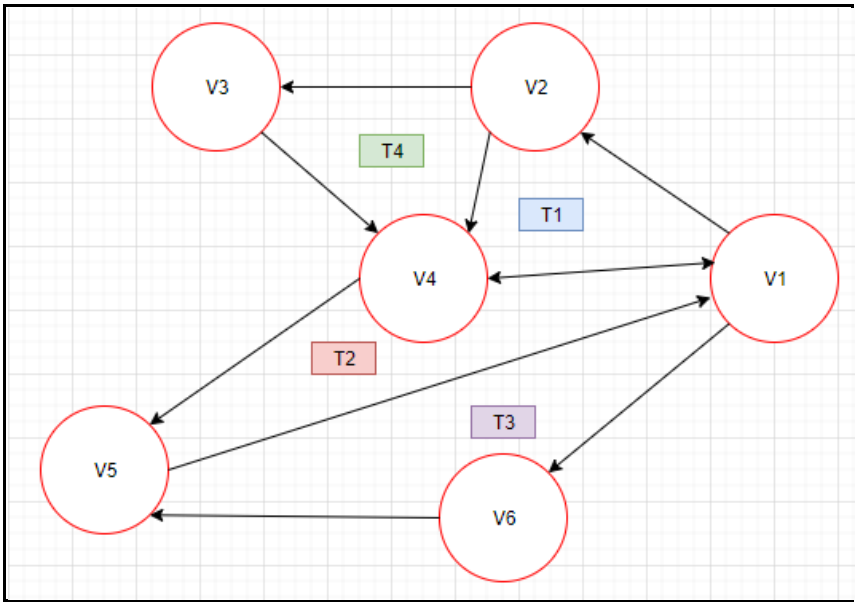

Figure 4: MigratePropagateDownstream

3.2.3 MigrateOriginalSource

MigrateOriginalSource migrates agent to the graph vertex where agent was first assigned to. The function fetches all the neighboring vertices associated to a vertex where the agent is located, and moves to the originating vertex if it is one of the neighbors. MigrateOriginalSource will support users executing algorithms to perform triangle counting and connected component counting on a graph.

3.2.4 MigratePropagateTree

MigratePropagateTree (see figure 5) propagates agents through a distributed binary tree. The function can be invoked with three parameters: *BothBranches*, *LeftBranch*, and *RightBranch*. *BothBranches* will propagate an agent on both branches of a tree node. *LeftBranch* and *RightBranch* will migrate the agent to the left and right branches of a tree node respectively. To implement tree propagation, we extended the *VertexPlace* class in MASS to store the left and

right branches associated to a tree node. For *BothBranches* the function will migrate the agent to the left neighbor, spawn a child agent and migrate the child agent to the right neighbor. For *LeftBranch* and *RightBranch,* the function will migrate the agent to the corresponding branches. MigratePropagateTree will be used in algorithms such as range search where agents propagate through the tree to determine if data present on a tree node is within a given range of values.

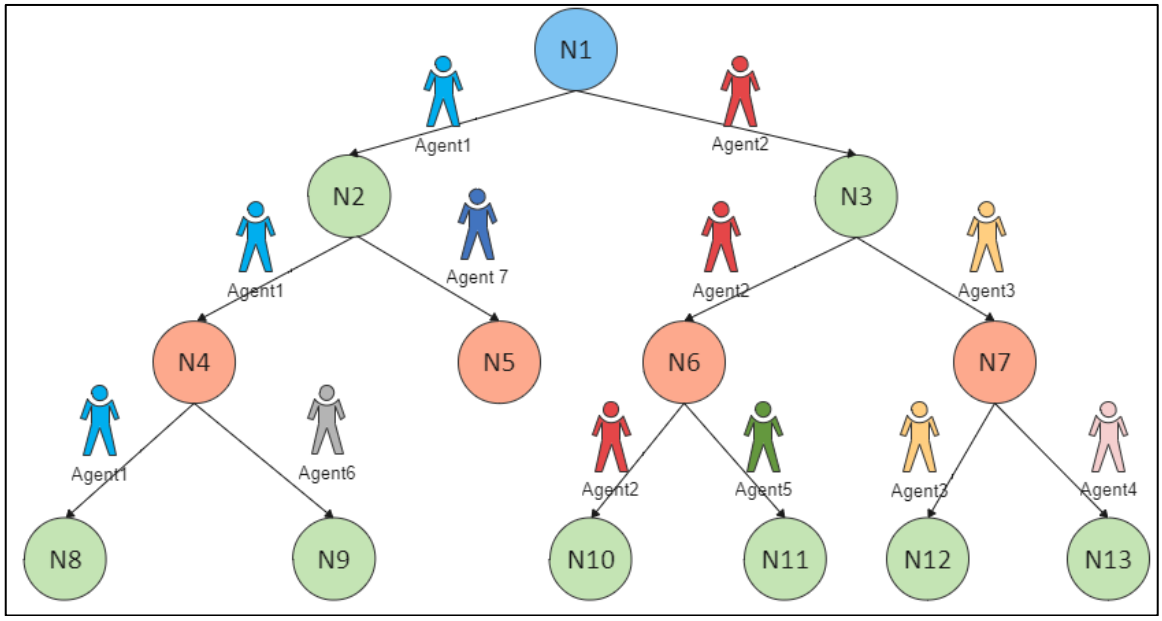

Figure 5: MigratePropagateTree

3.2.5 MigratePropagateRipple

MigratePropagateRipple (see figure 6) propagates agents to the Von-Neumann and Moore's neighborhood[2] of the agent's current location forming a ripple. The function fetches the neighboring places in Von-Neumann and Moore's neighborhood, spawn child agents and moves them to the neighboring places. This function will support users in executing the Closest pair of Points in contiguous space, Closest pair of Points using Quad tree, Voronoi diagram in contiguous space, Voronoi diagram using Quad tree and K-Nearest neighbors.

[2] Appendix A provides details of the Van-Neumann and Moore's Neighborhood

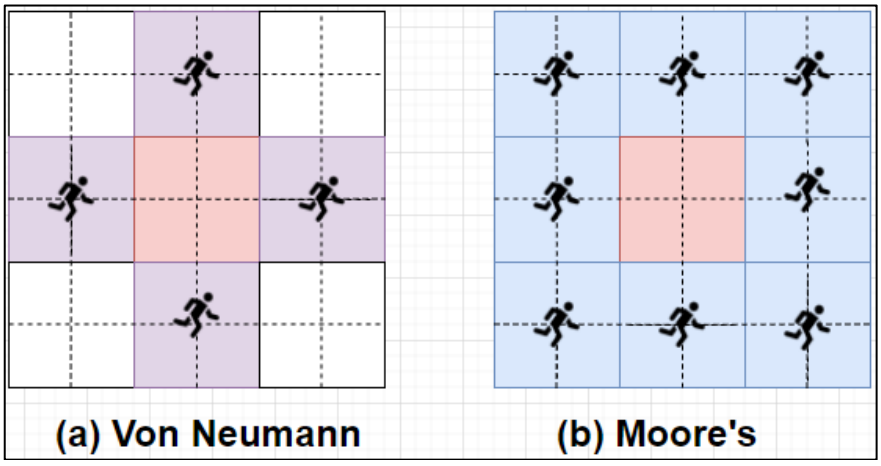

Figure 6: MigratePropagateRipple

3.2.6 MigrateMin and MigrateMax

MigrateMin and *MigrateMax* (see figure 7) will support algorithms that require agents to traverse through a graph based on the weight of the edges. *MigrateMin* will move an agent to the neighboring graph vertex with minimum edge weight. *MigrateMax* will move an agent to the neighboring graph vertex with maximum edge weight.

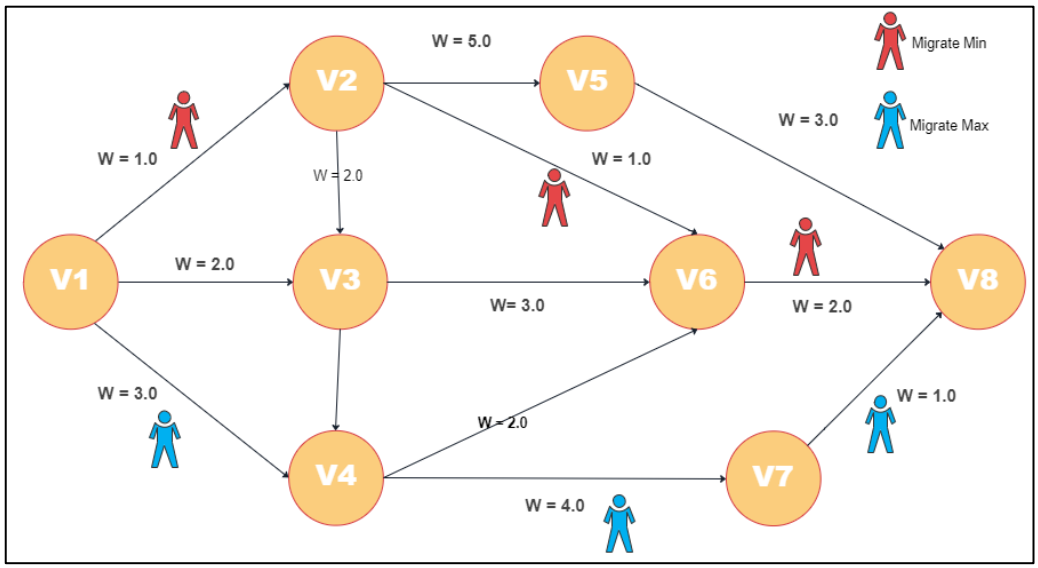

Figure 7: MigrateMin and MigrateMax

3.2.7 Migrate Random

MigrateRandom (see figure 8) will support algorithms that requires agents to traverse through a graph by randomly pick a neighboring vertex in a Graph for migration. This function will support users in executing Random Walk and Ant Colony optimization (ACO).

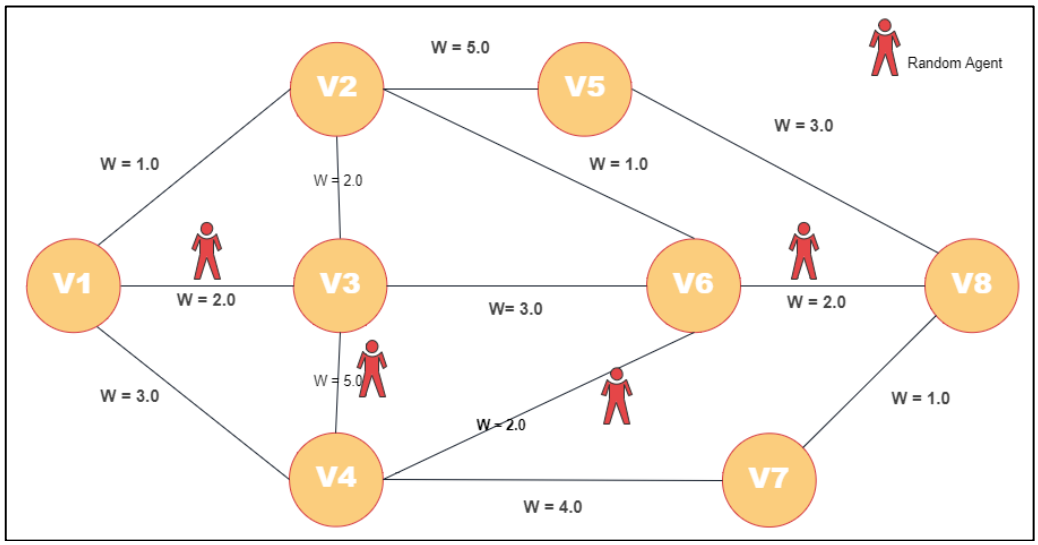

Figure 8: MigrateRandom

3.3 SMART AGENT AND SMART PLACE IMPLEMENTATION

We implemented two new classes *SmartAgent* and *SmartPlace* in MASS core library to support the high-level agent migration functions (which were listed above).

3.3.1 SmartAgent

The automated agent navigation methods will be part of the *SmartAgent* class. MASS applications have user-defined custom agent class to perform application specific actions. The user agents will extend *SmartAgent* to gain access to the automated migration methods. *SmartAgent* will in turn extend the *Agent* base class. In addition to the automated agent navigation methods, SmartAgent will contain properties: *itinerary* representing the list of places the agent navigated through,

nextNode representing the next destination for the *SmartAgent* and *prevNode* representing the previous place from where the agent migrated from.

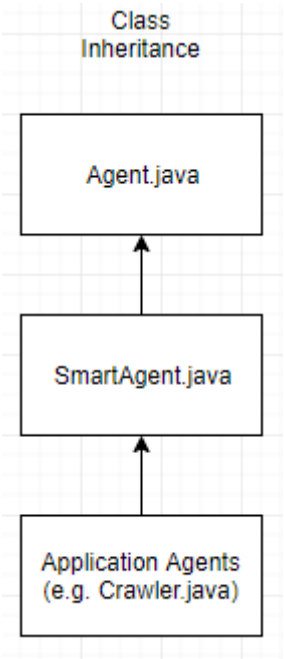

Figure 9: SmartAgent Class structure

3.3.2 SmartPlace

SmartPlace will have place properties that support automated agent navigation. For example, in case of migratePropagate, *SmartAgent* located at a place will need to know the neighboring places to spawn and migrate. *SmartPlace* will encapsulate the place properties such as neighbors and distances to the neighbors to support the automated agent navigation. The user-defined *place* class that is part of the MASS application will extend *SmartPlace* to gain access to the place properties used for automated migration methods. *SmartPlace* will extend the place base class.

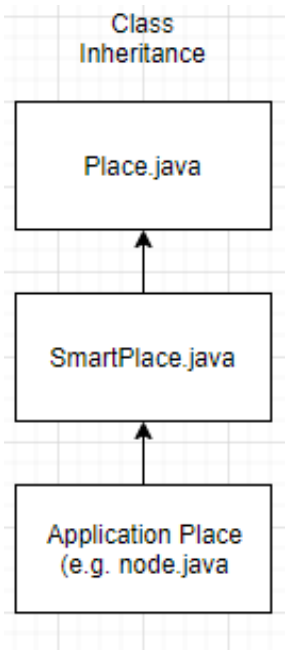

Figure 10: SmartPlace Class structure

Applying *SmartAgent* in a Mass Application: To demonstrate the use of SmartAgent in a MASS application, we describe how a MASS user building an application that requires agent propagation through a graph and changing an attribute on each of the graph vertex will utilize *SmartAgent*. The user-defined agent class will have a custom function annotated with *@Onarrival* that sets the desired attributes on the graph vertex when the agent arrives. The user agent class will extend the *SmartAgent* class to have access to the *MigratePropagate* function. The user program will then use the built-in *callAll* function that concurrently invokes the *MigratePropagate* function on all agents in the system.

3.4 AGENT LIFE CYCLE MANAGEMENT

MASS currently supports three basic agent-navigation and life-cycle management functions: *Migrate ()* moves agents to a specified place index, *Spawn ()* creates new agents that inherit properties from its parent agent and are spawned at the same place as the parent. *Kill ()* function terminates agents and removes it from the environment. *Migrate (), Spawn (), and Kill ()* set

attributes on agents, respectively. However, to migrate to different location, spawn children and terminate agents, the user's main program has to call the *ManageAll ()* function.

ManageAll () function (see figure 11) in the *AgentsBase* class operates on a bag of agents /*AgentsList* present in a computing node. It performs three primary agent lifecycle management functions - spawning child agents, terminating agents, and moving agents. Multiple threads pick up agents from the bag of agents in a thread-safe manner and evaluate each agent to perform the three-step life-cycle management.

Step1: Spawning child Agents: For each evaluated agent, check if a child agent should be spawned. If yes, new child agents are created, added to the bag of agents/*AgentsList* and at the same place as the parent agent. The agent is then registered with the messaging provider, and *OnArrival* and *OnCreation* events for agents, and *OnArrival* event for place are queued.

Step2: Termination/Kill Agents: For each evaluated agent, check if the evaluated agent should be terminated. If yes, the agent is removed from the place and from the bag of agents/*AgentsList.* Any frozen agent is re-instantiated, added to the *bag* of agents/*AgentsList* and at the same place as the parent agent. The agent is then registered with the messaging provider, and *OnArrival* and *OnCreation* events for agents, and *OnArrival* event for place are queued.

Step3: Migration of Agents: For each evaluated agent, this step checks if the agent should be moved. If yes, it checks for the destination place coordinate, removes from the agent from the current place and adds to the destination place.

As can be seen from the figure 11, *ManageAll* function does not perform migration (Step 3) for child agents spawned during Step 1 as a part of the same *ManageAll* execution. Algorithms that require child agents to be spawned and dispatched to different destinations require *ManageAll* to be executed twice. First for spawning the child agents and a second time for moving the agent to the desired destination. *ManageAll* is performance intensive, and executing *ManageAll* twice

for spawning and dispatching child agents has severe performance implications while executing algorithms over large datasets.

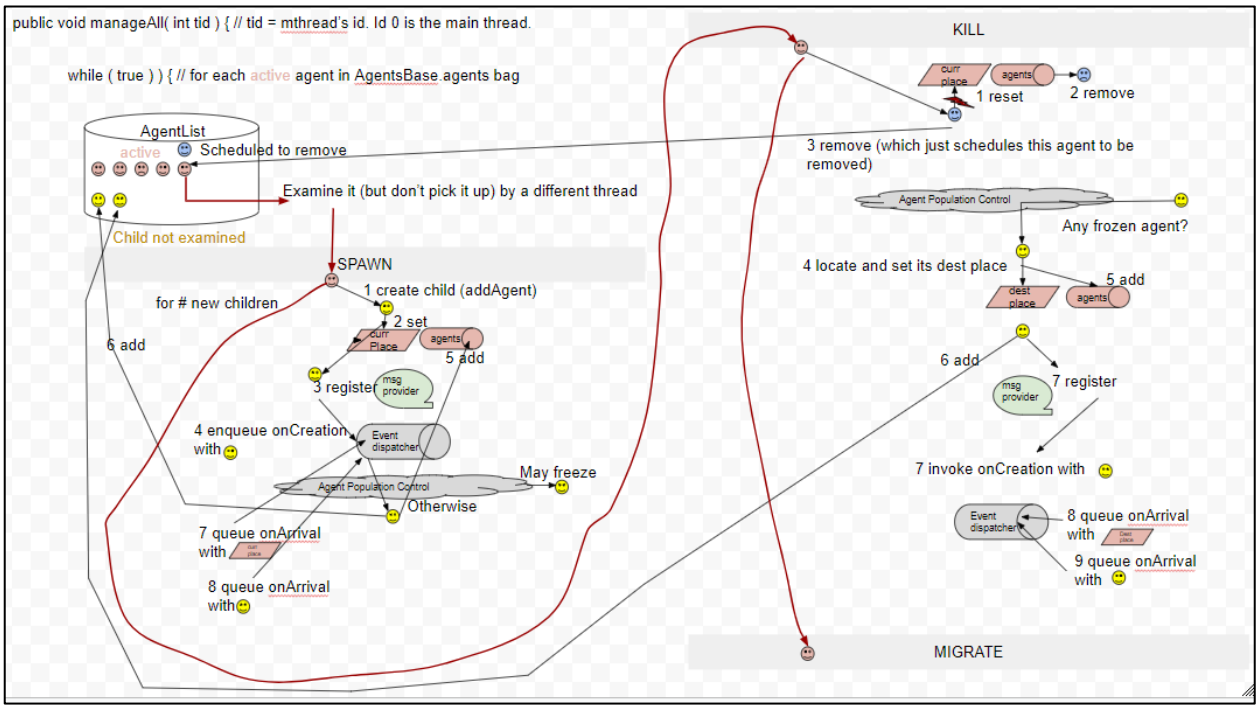

Figure 11: Current Spawn and Kill Functions in ManageAll()

In order to mitigate the poor performance and make automatic agent migration smoother, we revised *ManageAll* to de-couple agent lifecycle management from agent migration. In this revision (see figure 12), we created a separate helper function for agent lifecycle management which will evaluate the agents in the bag of agents/*AgentList*; (1) perform agent population control to see if a new agent can be created, (2) create child agent and add child agent to the place of parent agent and bag of agents, (3) terminal agents and finally (4) reset the bag of agents after agent addition and removal. In the new agent lifecycle management, the spawned child agents will be placed in the same bag of agents and *ManageAll* will continue to perform the agent migration for all agents including the child agents spawned. This design will ensure that agent cloning and migration will be carried out with only one *ManageAll* execution.

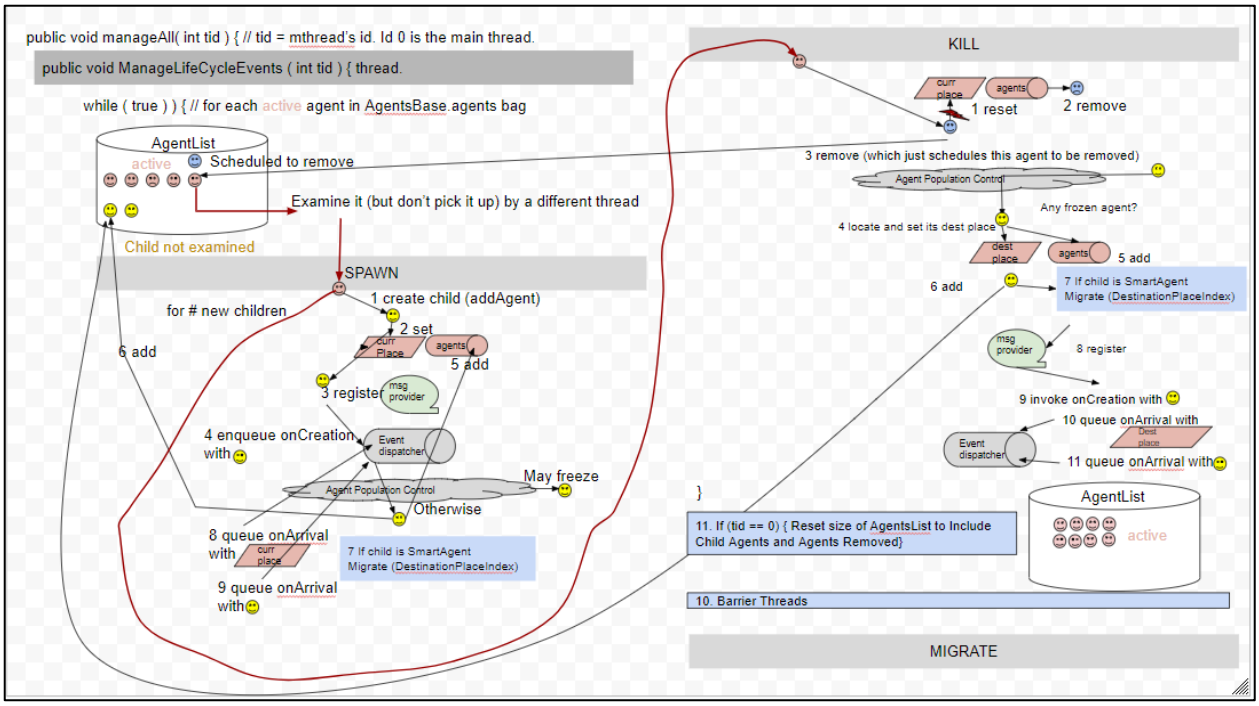

Figure 12: ManageLifeCycleEvents Function in ManageAll()

3.5 KD Tree Construction

In addition to updating benchmark programs with new automated agent migration methods, and improving agent lifecycle management, we also extended MASS core classes to support algorithms requiring tree traversal: The *VertexPlace* class was extended to store left and right branch references; the *GraphPlaces* class was extended to update properties of graph vertices distributed across cluster of computing nodes. Currently MASS users create application specific class for graph vertex with tree-traversal properties, write custom code to instantiate an agent, and migrate the agent to all the vertices in the graph to update it with tree-branch references. This is high-inefficient since the agent has to be migrated to the vertex when a new vertex is added to the graph. With this extension, user do not require: (1) custom code to hold tree-branch properties required for tree traversal and (2) agent migration to update to vertices distributed across cluster of computing nodes.

4 EVALUATION

This section measures the execution performance and programmability improvement brought by this automated agent migration, using four benchmark programs.

4.1 BENCHMARK PROGRAMS

To evaluate the automated agent migration, we updated four benchmark programs to utilize the new agent migration methods and measured it for accuracy, performance and programmability. Table 1 below shows the agent migration methods and the corresponding benchmark programs used for evaluation. MigrateMin, MigrateMax, MigrateRandom is not applicable to our existing benchmark set and will be used for future application development.

Automated Agent Navigation Methods	Benchmark programs
MigratePropagate	Breadth First Search
MigratePropagateDownStream and MigrateOriginalSource	Triangle Counting
MigratePropagateTree (BothBranches) MigratePropagateTree (LeftBranch) MigratePropagateTree (RightBranch)	Range Search
MigratePropagateRipple	Closest Pair of Points in Space

Table 1: Agent Propagation and Benchmark program

To measure accuracy of the automated agent migration, we tested benchmark programs against a range of inputs. We then compared the output against the output from legacy benchmark program implementation for the same set of inputs. All benchmark programs with automated agent migration provided accurate results with no deviation from the legacy benchmark implementation output. Table 2 below shows the automated agent migration method, corresponding benchmark program and the tests performed.

Method	Application	Tests Conducted
MigratePropagate	Breadth First Search	Tested with graph containing 100, 500 and 1000 and 2000 vertices
MigratePropagateDownStream MigrateOriginalSource	Triangle Counting	Tested with graph with 1000, 3000 and 10000 vertices
MigratePropagateTree (BothBranch) MigratePropagateTree (LeftBranch) MigratePropagateTree (RightBranch)	Range Search	Tested with trees containing 100, 100000, and 200000 datapoints
MigratePropagateRipple	Closest Pair of Points in space	Tested with 2D continuous space containing 64, 100, 100000, and 200000 datapoints

Table 2: Agent Propagation, Benchmark program and test performed

4.2 PERFORMANCE EVALUATION

4.2.1 MigratePropagate Performance

We implemented BreadthFirstSearch (BFS) using *MigratePropagate* and evaluated it on graphs with 100, 500, 1000 and 2000 vertices, running across four computing nodes. As can be seen from figure 14, the execution time for *MigratePropagate* is lower than the time taken by the legacy migration methodology and the difference in performance increases with the number of vertices in the graph. The performance improvement for *MigratePropagate* is primarily due to usage of the new *ManageAll* function which will spawn and migrate child agents as a part of a single execution. With the increase in number of vertices more agents are required to be spawned and migrated and there is increased efficiency from improvements to *ManageAll*. Table 3 shows the performance details including average, minimum and maximum execution times for triangle counting after five executions of each test case. Both legacy and new migrations were executed when the operational load on the computing nodes were comparable. Table 4 shows the BFS performance range at a 95% confidence interval.

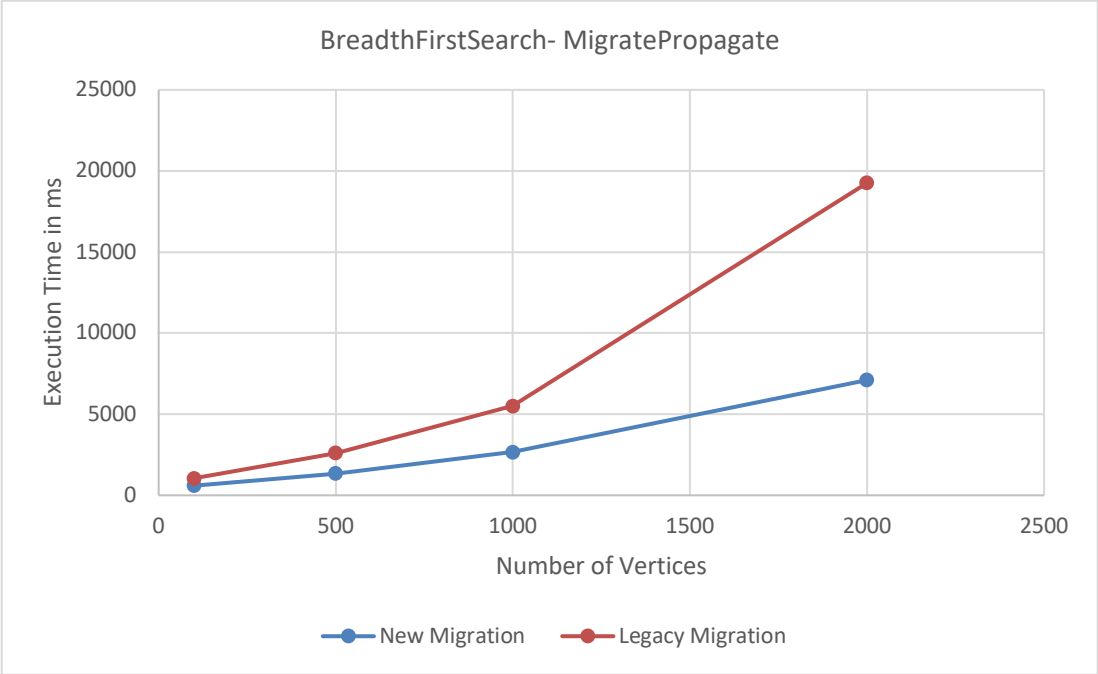

Figure 13: Overall Latency - Breadth First Search

	New Migration (ms)				Legacy Migration (ms)			
	100 Vertices	500 Vertices	1000 Vertices	2000 Vertices	100 Vertices	500 Vertices	1000 Vertices	2000 Vertices
Average	596	1351	2668	7104	1054	2608	5514	19253
Min	537	1023	2451	6763	966	2531	5272	17972
Max	685	1962	2869	7430	1194	2758	5839	20532

Table 3: BreadthFirstSearch Performance Details

95% Confidence	Population Mean for New Migration (ms)			
	100 Vertices	500 Vertices	1000 Vertices	2000 Vertices
Low	548	1024	2532	6869
High	644	1678	2804	7339

Table 4: Breadth First Search Performance at 95% confidence

4.2.2 MigratePropagateDownstream and MigrateOriginalSource Performance

We implemented triangle counting using *PropagateDownStream* and *MigrateOriginalSource* and evaluated it on graphs with 1000, 3000 and 10000 vertices, running across four computing nodes.

To perform triangle counting, agents are first allocated to all graph vertices, *MigratePropagateDownStream* is executed twice to propagate the agent from the source vertex to downstream vertices and then *MigrateOriginalSource* is executed to check if the source node is one of the neighbors and migrate to the source node. As can be seen from figure 15, the execution time for new agent migration methods is lower than the time taken by the legacy migration methodology and the difference in performance increases with the number of vertices in the graph. The performance improvement is due to the usage of new *manageAll ()* that spawns of child agents and dispatches them to their destinations in a single execution. Table 5 shows the performance details including average, minimum and maximum execution times for triangle counting after five executions of each test case. Both legacy and new migrations were executed when the operational load on the computing nodes were comparable. Table 6 shows the triangle counting performance range at a 95% confidence interval.

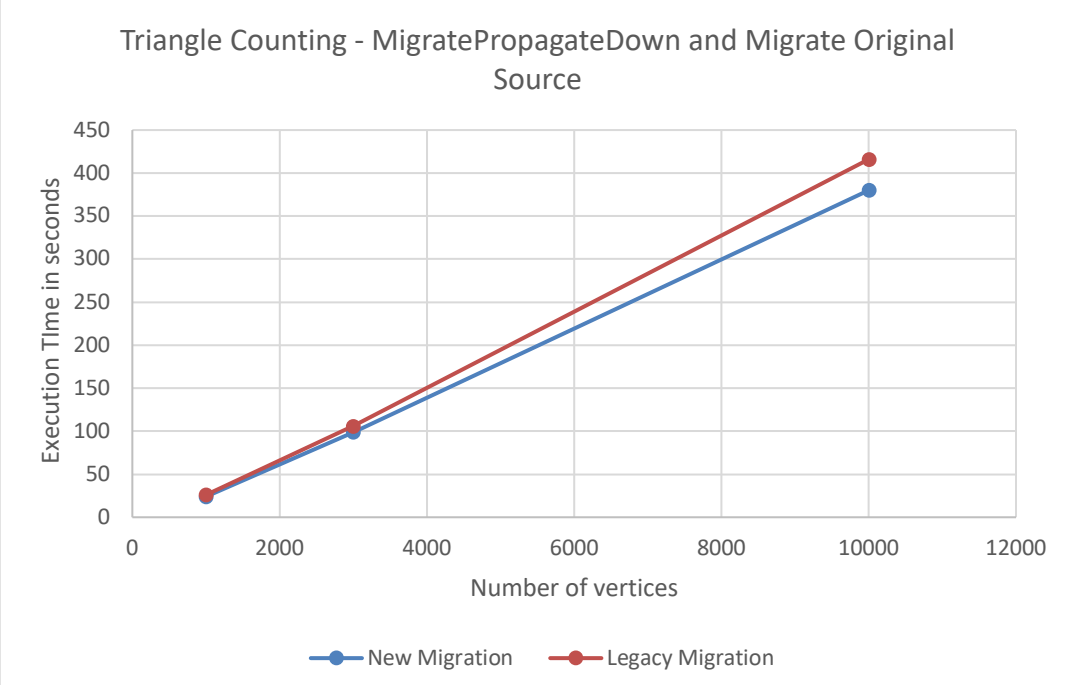

Figure 14: Overall Latency - Triangle Counting

	New Migration (s)			Legacy Migration (s)		
	1000 Vertices	3000 Vertices	10000 Vertices	1000 Vertices	3000 Vertices	10000 Vertices
Average	24	99	380	26	106	416
Min	21	94	367	25	103	400
Max	27	102	404	28	112	440

Table 5: Triangle Counting Performance Details

95% Confidence	Population Mean for New Migration (s)		
	1000 Vertices	3000 Vertices	10000 Vertices
Low	22	97	367
High	26	102	393

Table 6: Triangle Counting Performance at 95% confidence

4.2.3 MigratePropagateTree performance

We implemented range search using *MigratePropagateTree* and evaluated it by running across a distributed tree containing 100, 100K, and 200K datapoints over four computing nodes. Figure 15 shows the overall execution performance for different input datasets. The overall execution time for range search is dependent on both the input range and the number of input data points. We were unable to compare the performance against a legacy implementation as there is no comparable agent migration-based implementation for range search. Additionally, KDTree construction for large datasets do not complete in a reasonable timeframe with the legacy implementation and hence we don't have a comparable reference for performance. Table 7 shows the performance details including average, minimum and maximum execution times for range search after five executions of each test case. Both legacy and new migrations were executed when the operational load on the computing nodes were comparable. Table 8 shows the range search performance range at a 95% confidence interval.

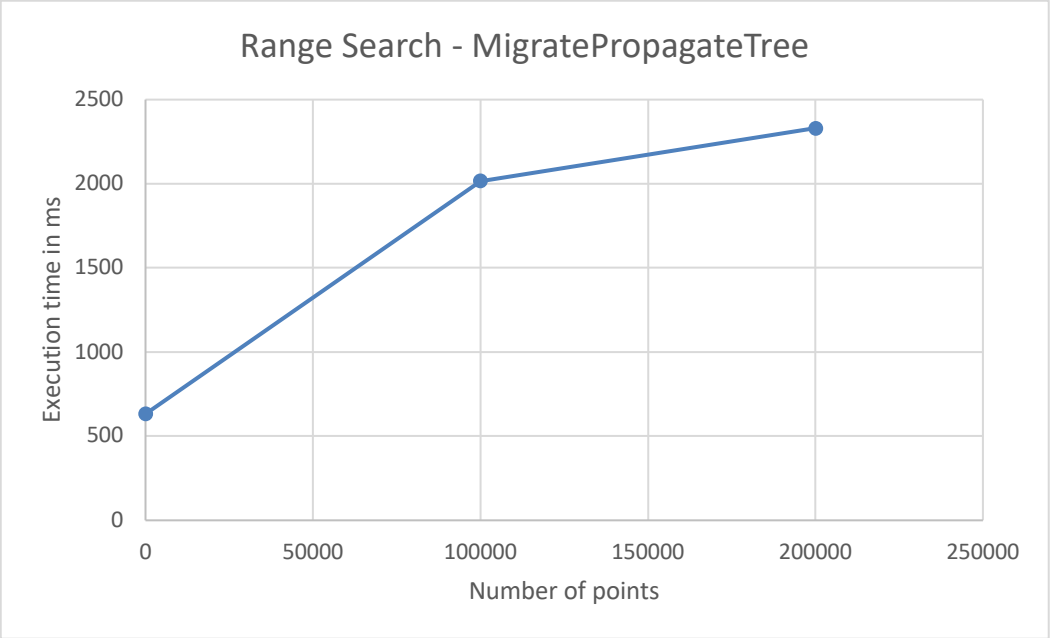

Figure 15: Overall Latency - Range Search

	MigratePropagateTree (ms)		
	100	100K Points	200K Points
Average	633	2016	2331
Min	588	1539	2155
Max	679	2229	2508

Table 7: Range Search Performance Details

95% Confidence	Population Mean for New Migration (ms)		
	100	100K Points	200K Points
Low	602	1777	2207
High	664	2255	2454

Table 8: - Range Search Performance at 95% confidence

4.2.4 KD Tree construction performance

Currently KDTree construction for large datasets (100K and 200K points) do not complete in a reasonable timeframe. This is because: (1) graph vertex don't have properties such as branch references required for tree traversal, and (2) there is no efficient way (whose reason is described

below) to update a graph vertex distributed across cluster of computing nodes. MASS users create application specific class for graph vertex with branch reference properties, write custom code to instantiate agent, migrate agent to the graph vertex and then update the graph vertex with the branch references. This is highly inefficient since each update requires agent migration and execution of *manageAll ()* which is performance intensive. We implemented improvements to the KD tree construction described earlier in the document and evaluated the performance by constructing distributed tree containing 100, 100K, and 200K datapoints. As can be seen from figure 16, we were able to construct trees with up to 200K datapoints within reasonable timeframe. Table 9 shows the performance details including average, minimum and maximum execution times for KDTree construction after five executions of each test case using both legacy and new methods. Both legacy and new methods were executed when the operational load on the computing nodes were comparable. Table 10 shows the KDTree construction performance range at a 95% confidence interval.

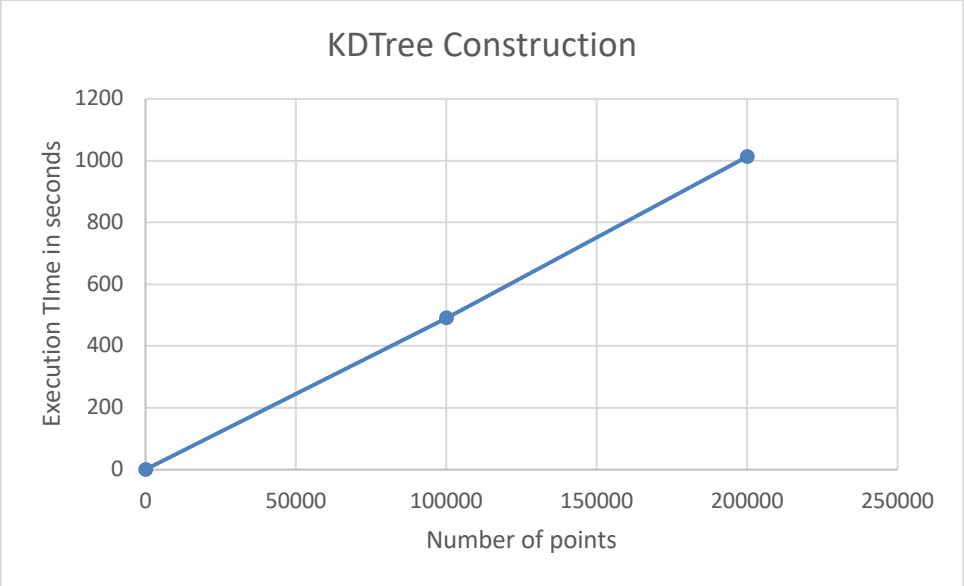

Figure 16: Overall Latency - KDTree Construction

	KDTree Construction (seconds) - New Method					KDTree Construction (seconds) - Old Method				
	100 Points	1000 Points	3000 Points	100K Points	200K Points	100 Points	1000 Points	3000 Points	100K Points	200K Points
Average	1	4	12	490	1013	22	270	992	-	-
Min	1	4	12	442	955	19	263	968	-	-
Max	1	4	13	505	1053	26	277	1023	-	-

Table 9: KD Tree Construction Performance Details

95% Confidence	Population Mean for New Method (s)				
	100 Points	1000 Points	3000 Points	100K Points	200K Points
Low	1	4	12	466	980
High	1	4	13	513	1046

Table 10: KD Tree construction Performance at 95% confidence

4.2.5 PropagateRipple performance

We implemented closest pair of points using PropagateRipple and evaluated it by running in 2D continuous space with 64, 100, 100K, and 200K datapoints over four computing nodes. As can be seen from figure 18, the execution time for new agent migration methods is at par with the time taken by the legacy migration methodology. Table 11 shows the performance details including average, minimum and maximum execution times for closest pair of points in space after five executions of each test case. Both legacy and new migrations were executed when the operational load on the computing nodes were comparable. Table 12 shows the KDTree construction performance range at a 95% confidence interval

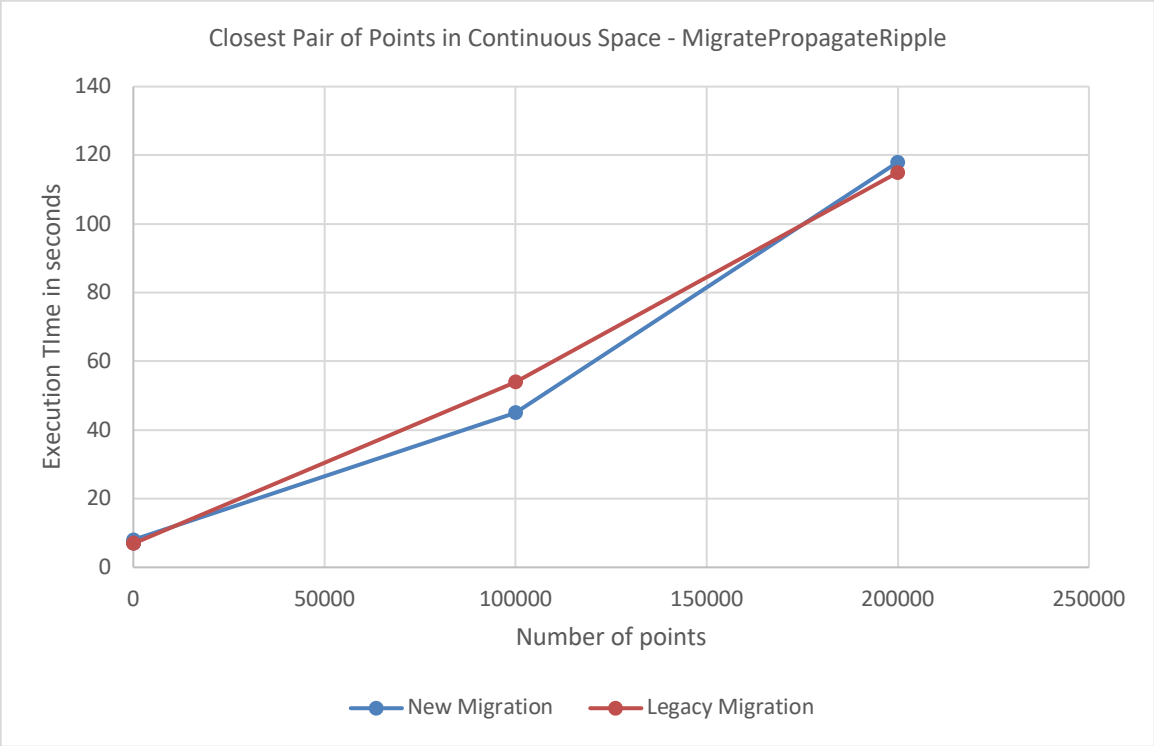

Figure 17: Overall Latency - Closest pair of points

	New Migration (s)				Legacy Migration (s)			
	64 Points	100 Points	100K Points	200K Points	64 Points	100 Points	100K Points	200K Points
Average	7	8	45	118	7	7	54	115
Min	6	7	44	116	7	7	52	113
Max	7	10	47	123	8	7	57	116

Table 11: Closest Pair of Points Performance Details

95% Confidence	Population Mean for New Migration (s)			
	64 Points	100 Points	100K Points	200K Points
Low	7	6	44	116
High	7	9	47	121

Table 12: Closest Pair of points Performance at 95% confidence

4.3 PROGRAMMABILITY EVALUATION

We examined programmability using quantitative and qualitative measurements. For quantitative measurements we examined the number of lines of custom code. For qualitative measurements we examine the ease of programming and the semantic meaning of the new agent migration methods.

Quantitative Measures. We tallied the custom lines of code (see Table 13) removed as a percentage of the total lines of code in the user defined agent class for the benchmark application. Triangle counting and Breadth First Search has the highest percentage of lines removed while closest pair of points has the lowest.

Method	Application	% LOC Removed
MigratePropagate	Breadth First Search	83%
MigratePropagateDownStream and MigrateOriginalSource	Triangle Counting	60%
MigratePropagateTree (BothBranch)	Range Search	44%
MigratePropagateTree (LeftBranch)		
MigratePropagateTree (RightBranch)		
MigratePropagateRipple	Closest Pair of Points	22%

Table 13: Number of lines of custom code removed

Triangle counting and Breadth first search currently utilize *OnArrival* and *OnDeparture* methods to simulate agent travel and propagation through the datasets. Users write all decision-making logic to move agent along the graph edges after examining the neighbors, spawning child agents and terminating agents. *MigratePropagate, MigrateOriginalSource* and *MigratePropagate DownStream* fully abstracts the agent navigation and propagation from the user and supports them as built-in MASS functions. This has helped reduce the user-maintained code by 83% and 60% respectively. Closest Pair of Points utilizes the *SpaceAgent* class that supports propagation in the *Von-Neumann* and *Moore's* neighborhood. However, it does not support agent life-cycle

management including agent termination and preventing duplicate agent propagation. *MigratePropagateRipple* handles propagation in *Von-Neumann* and *Moore's* neighborhood along with agent life cycle management.

Qualitative Measures. We believe that the new agent migration method is easy to program and is semantically closer to the original algorithms being performed. For example (see listing 2), to perform triangle counting, user will execute *MigratePropagateDownStream* twice to propagate the agent from the source vertex to downstream vertices and then execute *MigrateOriginalSource* to check if the source node is one of the neighbors and migrate to the source node. This is easier than writing *OnArrival* and *OnDeparture* methods to examine the graph edges, spawn agents to downstream vertices, and finally migrate back to the source vertex.

```
For (int i = 0; i < 3; i++)
{

    if (i < 2)
    {
        crawlers.callAll(Crawler.propagateDown_);
    }
    else{
        crawlers.callAll(Crawler.migrateOriginalSource_);
    }

    crawler.ManageAll();

}
```

Listing 2: User program for TC using automated agent migration methods

Additionally, *MigratePropagateDownStream* and *MigrateOriginalSource* are more intuitive than the *OnArrival* and *OnDeparture* methods (see listing 3) which we believe do not convey the semantic meaning of the algorithm being performed.

```
For (int i = 0; i < 3; i++)
{
    crawler.callAll(Crawler.departure_);
    crawler.ManageAll();
    crawler.callAll(Crawler.onArrival_);
    crawler.ManageAll();
}
```

Listing 3: User program for TC in current MASS

We also compared the availability of automated agent migration methods with other competing products such as Netlogo and Repast Simphony. As can be seen from Table 14, with the introduction of the new automated agent navigation, MASS now has the greatest number of advanced agent migration methods. Repast Simphony supports certain agent navigational methods such as *Shortestpath*, *MoveAgentByDisplacement* and *MoveAgentbyVector* that is currently not supported in MASS. However, *SmartAgent* in MASS can be easily extended to incorporate these agent navigational patterns.

	NetLogo	Repast Simphony	Mass Old	Mass New
MigratePropagate	No	Yes[3]	No	Yes
MigratePropagateDownStream	No	No	No	Yes
MigrateOriginalSource	No	No	No	Yes
MigratePropagateTree	No	Yes	No	Yes
MigratePropagateRipple	Yes[4]	Yes[5]	No	Yes
MigrateMin	No	No	No	Yes
MigrateMax	No	No	No	Yes
MigrateRandom	No	No	No	Yes
ShortestPath	No	Yes	No	No
MoveAgentByDisplacement	No	Yes	No	No
MoveAgentByVector	No	Yes	No	No

Table 14: Availability of automated agent migration across products

Based on the discussion in this section we can conclude that the new automated agent migration along with the improvements to the agent lifecycle management has brought significant performance improvements to MASS while executing the benchmark programs. The new

[3] Repast simphony provides method to perform breadth first search
[4] Netlogo supports models such as Voronoi diagram, K-Nearest Neighbor.
[5] Repast simphony has methods to check if an agent is present in the Von Neumann or Moore's neighborhoood

automated agent migration has also made MASS easier to use and has improved the programmability when compared to other competing products such as Repast simphony and Netlogo.

5 CONCLUSION

5.1 SUMMARY

To pursue our research focused on automated agent navigation over structured data, we identified, designed, and implemented eight automated agent navigation functions in MASS java. We improved the execution performance of agent propagation through structured data with two major achievements: (1) improvements to existing agent lifecycle management functions that migrate, spawn, and terminate agents and (2) improvements to graph construction in MASS java by building the support for tree traversal and the ability to update properties of graph vertices distributed across a cluster of computing nodes. The performance improvements to the benchmark programs demonstrated the efficiency of the new agent navigation methods and the agent lifecycle improvement. The performance improvement to KD Tree construction demonstrated the efficiency of improvements made to graph construction in MASS java. Programmability evaluation shows that new implementations reduced user line of codes (LOC), made the code more intuitive and semantically closer to the original algorithms. We successfully achieved our project goal by identifying the generalized agent navigational patterns, designing and implementing automatic agent navigation functions, and making performance improvements to the current agent lifecycle management functions.

5.2 FUTURE WORK

To further extend the work we have completed, we see the following opportunities:

1) Introduce additional agent navigation functions including:

- *PropagateRippleWithBouncing*: This method will support the calculation of Euclidean shortest path between two points in contiguous space. It will propagate a ripple from the source point in contiguous space and bounce off opaque obstacles until the ripple detects the destination point.
- *MigrateLowestCoordinatePoint and MigrateUnboundedRegion***:** These methods will be used for constructing a Convex Hull. MigrateLowestCoordinatePoint will move an agent to the starting Coordinate point and MigrateUnboundedRegion will be used to move the agent to the Voronoi site present in the unbounded Voronoi region.
- *MoveAgentByDisplacement* and *MoveAgentByVector*: *MoveAgentByDisplacement* will move an agent from its current location by the specified amount of displacement in a continuous space. MoveAgentByVector will move an agent from its current location by the specific amount of displacement along the specified angle.

2) **Implement benchmark programs to evaluate MigrateMin, MigrateMax and MigrateRandom**. We have re-implemented benchmark programs to evaluate Migrate Propagate, MigratePropagate DownStream, Migrate Original Source, Migrate Propagate Tree, MigratePropagateRipple. In future we will implement benchmark programs such as Dijkstra's algorithm to verify MigrateMin, MigrateMax and MigrateRandom.

3) **Re-implement more benchmark programs in MASS using the automated agent navigation methods.** We have re-implemented four benchmark applications including breadth first search, triangle counting, range search, and closest pair of points in continuous space using the new agent navigation functions. We will utilize *MigratePropagateRipple* to re-implement, Closest pair of Points using Quad tree, Voronoi diagram in contiguous space, Voronoi diagram using Quad tree and K-Nearest neighbors. We will utilize *MigratePropagateDownStream* to re-implement connected components in graph.

BIBLIOGRAPHY

1. M. Fukuda, C. Gordon, U. Mert and M. Sell, "An Agent-Based Computational Framework for Distributed Data Analysis," in *Computer*, vol. 53, no. 3, pp. 16-25, March 2020, doi: 10.1109/MC.2019.2932964.
2. J. Gilroy, S. Paronyan, J. Acoltzi and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2957-2966, doi: 10.1109/BigData50022.2020.9378298.
3. Charles M. Macal and Michael J. North. 2009. Agent-based modeling and simulation. In Winter Simulation Conference (WSC '09). Winter Simulation Conference, 86–98.
4. Uri Wilensky; William Rand, "Analyzing Agent-Based Models," in *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*, MIT Press, 2015, pp.283-310.
5. Uri Wilensky; William Rand, "The Components of Agent-Based Modeling," in *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*, MIT Press, 2015, pp.203-282.
6. M. Kipps, W. Kim and M. Fukuda, "Agent and Spatial Based Parallelization of Biological Network Motif Search," *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 2015, pp. 786-791, doi: 10.1109/HPCC-CSS-ICESS.2015.222.
7. Spark. Accessed on: Apr. 15, 2022. [Online]. Available: http://spark.apache.org/
8. Hadoop. Accessed on: Apr. 17, 2022. [Online]. Available: http://hadoop.apache.org/
9. Repast Simphony Shortest Path. Accessed on: Apr. 30,2022. [Online]. Available: https://repast.github.io/docs/api/repast_simphony/repast/simphony/space/graph/ShortestPath.html/
10. Repast Simphony Continuous Space. Accessed on: Apr. 30,2022. [Online]. Available: https://repast.github.io/docs/api/repast_simphony/repast/simphony/space/continuous/ContinuousSpace.html/
11. Repast Simphony Graph Utilities. Accessed on: Apr. 30,2022. [Online]. Available: https://repast.github.io/docs/api/repast_simphony/repast/simphony/engine/graph/EngineGraphUtilities.html/

12. Repast Simphony Nary Traverser. Accessed on: Apr. 30, 2022. [Online]. Available: https://repast.github.io/docs/api/repast_simphony/repast/simphony/engine/graph/NaryTreeTraverser.html
13. Netlogo Dictionary, Accessed on: May 4, 2022. [Online]. Available: https://ccl.northwestern.edu/netlogo/docs/dictionary.html/
14. Netlogo Models, Accessed on: May 4,2022. [Online]. Available: https://ccl.northwestern.edu/netlogo/models/
15. Von-Neumann Neighborhood, Accessed on: May 5,2022. [Online]. Available: https://mathworld.wolfram.com/vonNeumannNeighborhood.html
16. Moore Neighborhood, Accessed on: May 5,2022. [Online]. Available: https://mathworld.wolfram.com/MooreNeighborhood.html
17. University of Washington. MASS Java Manual, 2016. https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf
18. MASS: A Parallelizing Library for Multi-Agent Spatial Simulation, Accessed on: April 2,2022. [Online]. Available: http://depts.washington.edu/dslab/MASS/index.html

APPENDIX A

Von-Neumann and Moore's Neighborhood

Von-Neumann and Moore's neighborhood represent neighboring spaces in a 2D contagious space. Von -Neumann Neighborhood [15] is a diamond shaped neighborhood surrounding a give cell as shown in the figure 3.

	(x,y+1)	
(x-1,y)	P (x,y)	(x+1,y)
	(x,y-1)	

Figure 18: cells in the Von-Neumann Neighborhood

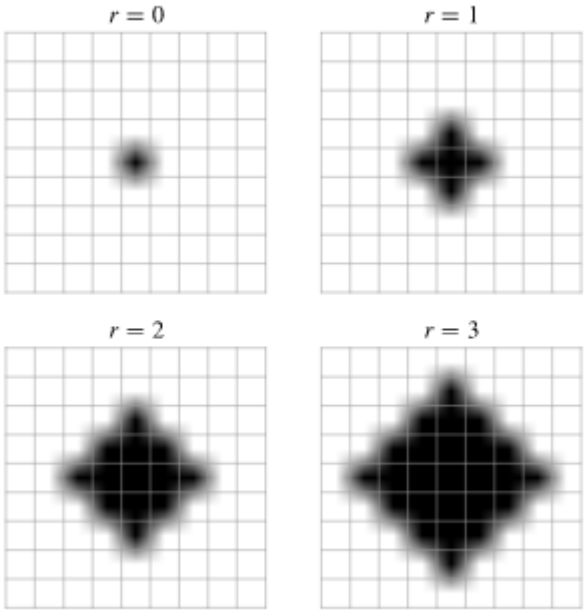

Figure 19: von Neumann neighborhoods for ranges r 0, 1, 2, and 3

Moore's Neighborhood [16] is a square shaped neighborhood surrounding a give cell as shown in the figure 5.

(x-1,y+1)	(x,y+1)	(x+1,y+1)
(x-1,y)	P (x,y)	(x+1,y)
(x-1,y-1)	(x,y-1)	(x+1,y-1)

Figure 20: cells in the Moore's Neighborhood

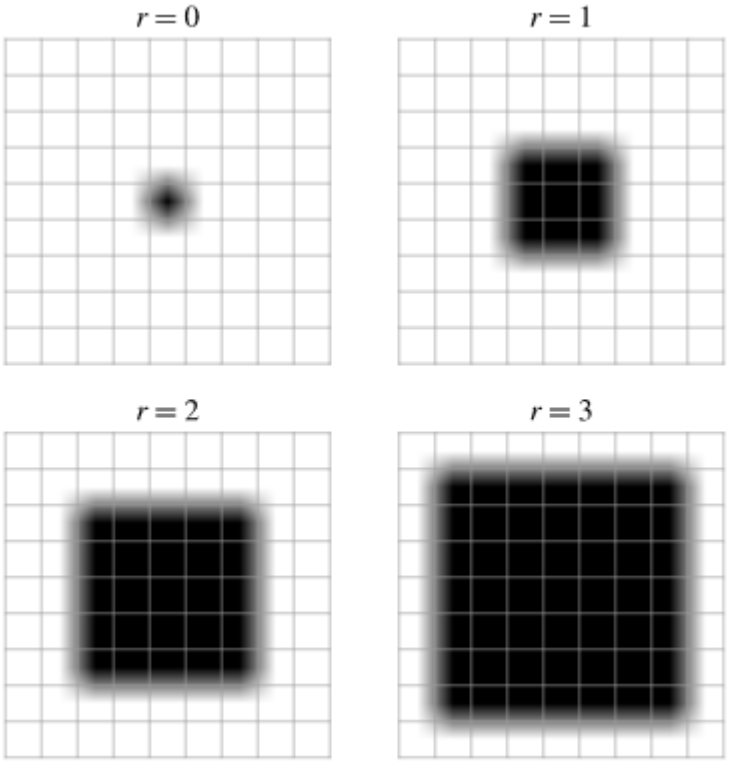

Figure 21: Moore neighborhoods for ranges r 0, 1, 2, and 3

APPENDIX B1

MigratePropagate

```
public Object migratePropagate( Object arg )
{    SmartPlace smartPlace = (SmartPlace) getPlace( ); //Get the place where the agent is
     if ( smartPlace.footprint == -1 ) {
        int[] neighbors = smartPlace.neighbors;
        int[] distances = smartPlace.distances;
        if (neighbors.length == 0 || (neighbors.length == 1 && prevNode == neighbors[0]))
        {
           smartPlace.footprint = 1; //This place has been visited
           kill( );
           return null;
        }
        nextNode = ( neighbors[0] != prevNode ) ? neighbors[0] : neighbors[1];
        migrate( nextNode ); //Migarte to the next Node

        SmartArgs2Agents[] args
            = new SmartArgs2Agents[( getAgentId( ) == 0 && getPlace( ).getIndex( )[0] == 0 && ( ( SmartPlace
           )getPlace( ) ).footprint == -1 ) ? neighbors.length - 1: neighbors.length - 2];

        if (args.length == 0) {
           prevNode = getPlace( ).getIndex( )[0];
           smartPlace.footprint = 1; //This place has been visited
           return null; //if there are no neighbours to spawn, just return
        }

        for ( int i = 0, j = 0; i < neighbors.length; i++ ) {
           if ( neighbors[i] == nextNode || neighbors[i] == prevNode )
              continue;
           args[j++] = new SmartArgs2Agents( neighbors[i],getPlace( ).getIndex( )[0]);
        }

        spawn( args.length, args );

        prevNode = getPlace( ).getIndex( )[0];
        smartPlace.footprint = 1; //This place has been visited

     }else {
        kill( );
     }
     return null;
}
```

APPENDIX B2

MigratePropagateDown

```
public Object propagateDown( Object arg )
  {
    int currStep = ((Integer) arg).intValue();
    if (getPlace() != null && !(getPlace() instanceof VertexPlace)) {
                return null;
    }

    Object [] neighbors = ((VertexPlace) getPlace()).getNeighbors();
    int currNodeGlobalIndex = getPlace().getIndex()[0];

    int availableEdges = 0;
    for (int i = 0; i < neighbors.length; i++) {
      int neighborGlobalIndex = (Integer)neighbors[i];
      if (neighborGlobalIndex < currNodeGlobalIndex)
        // Going to a neighbor with a lower id
        availableEdges++;
    }

    if (availableEdges == 0) {
      // No more edges to explore. I'm done
      kill();
    } else {
      // Prepare arguments to be passed to children
      SmartArgs2Agents[] args = new SmartArgs2Agents[availableEdges -1];
      int argsCount = 0; // eventually reaches # children
      for (int i = 0; i < neighbors.length; i++) {
        int neighborGlobalIndex = (Integer)neighbors[i];

        if (neighborGlobalIndex < currNodeGlobalIndex) {
          if (--availableEdges == 0) {
            // Parent takes the last available edge and also immediately migrates
            itinerary[currStep + 1] = neighborGlobalIndex;
            migrate(itinerary[currStep + 1]);
          } else {
            int[] childItinerary = itinerary.clone();
            childItinerary[currStep + 1] = neighborGlobalIndex;
            args[argsCount++] = new SmartArgs2Agents( childItinerary,neighborGlobalIndex);
          }
        }
      }

      if (args != null) {
        spawn(args.length, args);
      }
    }

    return null;
  }
```

APPENDIX B3

MigrateOriginalSource

```
public Object migrateSource( Object arg )
  {
    int currStep = ((Integer) arg).intValue();
    if (getPlace() != null && !(getPlace() instanceof VertexPlace)) {
      return null;
    }

    // Retrieve the current node's information
    Object [] neighbors = ((VertexPlace) getPlace()).getNeighbors();

    // Check if the current node has my original node as a neighbor
    for (int i = 0; i < neighbors.length; i++) {
      int neighborGlobalIndex = (Integer)neighbors[i];

      if (neighborGlobalIndex == itinerary[0]) { // YES
        itinerary[currStep + 1] = neighborGlobalIndex;
        migrate(itinerary[currStep + 1]); //TO DO Check why this was missed
        break;
      }
    }
    if (itinerary[currStep + 1] == -1) { // NO
      MASS.getLogger().debug("Step " + currStep +
          ": agent(" + getAgentId() + ") can't go home at " +
          itinerary[0] + " and thus gets terminated at " +
          getPlace().getIndex()[0])   ;
      kill();
    }

    return null;

  }
```

APPENDIX B4

MigratePropagateTree

```
public Object propagateTree( int path, Object arg )
{
   if (getPlace() != null && !(getPlace() instanceof VertexPlace)) {
      return null;
   }
   int currNodeGlobalIndex = getPlace().getIndex()[0];
   int left = ((VertexPlace)getPlace()).left;
   int right = ((VertexPlace)getPlace()).right;
   if (left == -1 && right  == -1) {
      kill();
   } else if(((VertexPlace)getPlace()).footprint == 1){
      kill();
   }
   else {
      switch (path){
         case BothBranch_:
            if (left != -1 && right  != -1){
               migrateAndSpawn(arg, left, right);
            }
            else if (left != -1)
               migrate(left);
            else if (right != -1)
               migrate(right);
            break;
         case LeftBranch_:
            if (left != -1)
               migrate(left);   //Migrate the Agent to Left Branch
            break;
         case RightBranch_:
            if (right != -1){
               migrate(right);   //Migrate the Agent to Left Branch
      }
      ((VertexPlace)getPlace()).footprint = 1; //Setting the footprint since place is visited
   }
   return null;
}

private Object migrateAndSpawn( Object arg, int left, int right)
{
   migrate(left); //Migrate Parent Agent to left
   Object [] arguments = new Object[2];
   arguments[0] = arg;
   arguments[1] = level;
   SmartArgs2Agents[] args = new SmartArgs2Agents[1];
   args[0] = new SmartArgs2Agents(SmartArgs2Agents.rangeSearch_, arguments, right, -1);
   spawn(args.length, args);
   return null;
}
```

APPENDIX B5

MigratePropagateRipple

```
public Object propagateRipple(Object argument) {
   if (getPlace() != null && !(getPlace() instanceof SpacePlace))
      return null;
   boolean hasThePlaceBeenAlreadyVisited = checkandUpdateFootPrint();
   if (hasThePlaceBeenAlreadyVisited)
   {  kill();
      return null;
   }

   if (generation % 2 == 0) {
      // if generation is even, spawn to N, W, E, S
      spawnAgentinNeighborhood("vonNeumann", argument, currentCoordinates);

   } else {
      // if generation is odd, spawn to N, W, E, S, NW, SW, NE, SE
      spawnAgentinNeighborhood("moore", argument, currentCoordinates);
   }
   kill(); //Kill Parent Agent;
   return null;
}

//agent spawns in 'moore' manner or "vonNewmann" manner
 private Object spawnAgentinNeighborhood(String neighborhoodPattern, Object args, double[]
   currentCoordinates) {
   int numOfAgents = 0;
   switch(neighborhoodPattern){
      case "moore":
         numOfAgents = 8;
         break;
      case "vonNeumann":
         numOfAgents = 4;
         break;
   }

   Object[] arguments = new Object[numOfAgents];
   int[] curIndex = getIndex();

   // new coordiantes where agent is migrating
   double[] subInterval = ((SpacePlace) getPlace()).getSubInterval().clone();
   Vector<double[]> neighbors = SpaceUtilities.getNeighborPatterns(neighborhoodPattern, subInterval);
   for (int i = 0; i < neighbors.size(); i++ ) {
      for (int j = 0; j < subInterval.length; j++) {
         neighbors.get(i)[j] += currentCoordinates[j];
      }
   }
   generation++;
   for(int i = 0; i < numOfAgents; i++){
      SpacePlace sp = (SpacePlace)getPlace();
      Object[] finalArgs = new Object[2];
      SpaceAgentArgs spaceAgentArgs = new SpaceAgentArgs(currentCoordinates, neighbors.get(i),originalCoordinates,
            getIndex(), subIndex, generation, originalId);
      finalArgs[0] = (Object) spaceAgentArgs;  //argument for SpaceAgent
      finalArgs[1] = args;  //argument for customer agent class
      arguments[i] = (Object) finalArgs;
   }
   spawn(numOfAgents, arguments);
   return null;
}
```

APPENDIX B6

MigrateMin

```
public Object migrateMin( Object arg )
{
   if (getPlace() != null && !(getPlace() instanceof VertexPlace) && !(getPlace() instanceof SmartPlace)) {
      return null;

   Integer [] neighbors; //Holds the indexes of neighbors
   Integer [] weights; //Holds weights of the the neigbors

   if (getPlace() instanceof VertexPlace)
   {
      neighbors = Arrays.asList(((VertexPlace) getPlace()).getNeighbors())
               .toArray(new Integer[0]);

      weights = Arrays.asList(((VertexPlace) getPlace()).getWeights())
            .toArray(new Integer[0]);

   }else {
      return null;
   }

   //If neigbors or weights are null return null
   if (neighbors == null || weights == null)
      return null;

   int minimumWeightIndex = getMinimumWeightIndex(weights);
  migrate( neighbors[minimumWeightIndex] ); //Migrate to Node with minimum weight
  return null;

}

private int getMinimumWeightIndex(Integer [] weights)
{
   int min = Integer.MAX_VALUE;
   int MinimumWeightIndex = 0;

   for (int i = 0; i < weights.length; i++)
   {
      if (min > weights[i].intValue())
      {
         min = weights[i].intValue();
         MinimumWeightIndex = i;
      }
   }

   return MinimumWeightIndex;
}
```

APPENDIX B7

MigrateMax

```
public Object migrateMax( Object arg )
  {
    if (getPlace() != null && !(getPlace() instanceof VertexPlace) && !(getPlace() instanceof SmartPlace))
        return null;
    Integer [] neighbors; //Holds the indexes of neighbors
    Integer [] weights; //Holds weights of the the neigbors

    if (getPlace() instanceof VertexPlace)
    {
      neighbors = Arrays.asList(((VertexPlace) getPlace()).getNeighbors())
          .toArray(new Integer[0]);

      weights = Arrays.asList(((VertexPlace) getPlace()).getWeights())
          .toArray(new Integer[0]);

    }else {
      return null;
    }

    //If neigbors or weights are null return null
    if (neighbors == null || weights == null)
      return null;

    int maximumWeightIndex = getMaximumWeightIndex(weights);
    migrate( neighbors[maximumWeightIndex] ); //Migrate to Node with maximum weight
    return null;
  }

private int getMaximumWeightIndex(Integer [] weights)
  {
    int max = Integer.MIN_VALUE;
    int MaximumWeightIndex = 0;

    for (int i = 0; i < weights.length; i++)
    {
      if (max < weights[i].intValue())
      {
        max = weights[i].intValue();
        MaximumWeightIndex = i;
      }
    }

    return MaximumWeightIndex;
  }
```

APPENDIX B8

MigrateRandom

```
public Object migrateRandom( Object arg )
  {
    if (getPlace() != null && !(getPlace() instanceof VertexPlace) && !(getPlace() instanceof SmartPlace))
      return null;

    Integer [] neighbors; //Holds the indexes of neighbors
    Integer [] weights; //Holds weights of the the neigbors

    if (getPlace() instanceof VertexPlace)
    {
      neighbors = Arrays.asList(((VertexPlace) getPlace()).getNeighbors())
          .toArray(new Integer[0]);

      weights = Arrays.asList(((VertexPlace) getPlace()).getWeights())
          .toArray(new Integer[0]);

    }else {
      return null;
    }

    //If neigbors or weights are null return null
    if (neighbors == null || weights == null)
      return null;

    int randomIndex = getRandomIndex(weights);

    migrate( neighbors[randomIndex] ); //Migrate to Node with maximum weight

    return null;
  }
```