A Survey, Model Design, and Benchmark Test Implementation of Agent-Based Applications

By Caleb Yang
Mentor Professor Munehiro Fukuda

Introduction

The survey of simulation applications in the fields of biology, business/industry, and economics/social sciences that rely on agent-based modeling has previously identified key characteristics in the agent-based model. These key characteristics include agent types, agent micro-behaviors, modes of communication, and finally simulation topologies. Previously, one sequential benchmark test was implemented, Brain Grid. This past quarter the following sequential benchmark test have been implemented; financial modeling, social network modeling, and MatSim. The Tuberculosis model's sequential benchmark test is close to completion.

Application Model Overview

In the survey/literature review two family of model classes; static and dynamic agent applications. Static agent applications indicate that the modeled entities stay in place after initialization, whereas dynamic agent applications do have the modeled entities move about its environment after initialization. It was later determined that the clustered agents model can be classified differently when changing the dynamics, such as when the entities (i.e., banks and firms) become environmental elements and messages become the agents in their stead.

Static Agent Applications

Brain Grid <Neural Networks>:

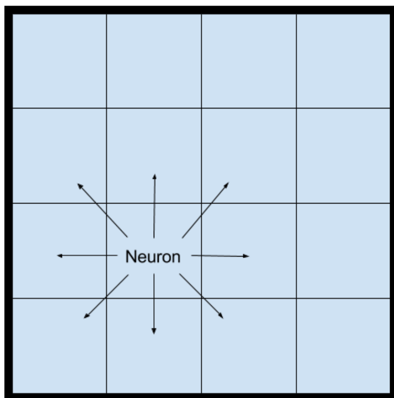

Figure 1: Neural Network Active Neuron Growth

Original Application Summary:
Brain Grid is an application that models the functionality and growth of neural synapses. The simulation environment is comprised of three types of neurons: active, inhibitory, and neutral. Active neurons initiate the growth in Moore's neighborhoods, whereas neutral neurons perpetuate it and inhibitory neutralizes it.

Financial Modeling <Clustered Agents>:

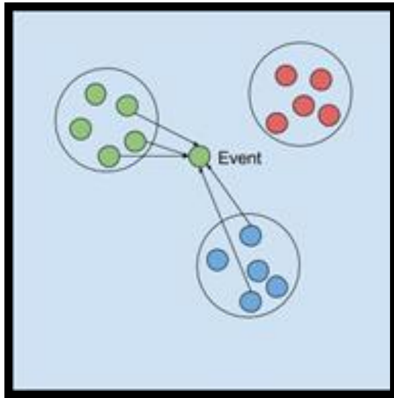

Figure 2: Clustered Agent Multicast Event

Original Application Summary:
The original application models three financial relief solutions: liquidated by a purchase & assumption, bail-out, bail-in. Thus, to model it there are three agents: banks, firms, households. There are seven phases used to model financial relief including loan request, raise liquidity, and dividend payout.

This model was used to better understand the effectiveness of a bail-in solution to the alternative purchase and assumption and especially a bail-out.

Social Network Modeling
<Network of Static Agents>:

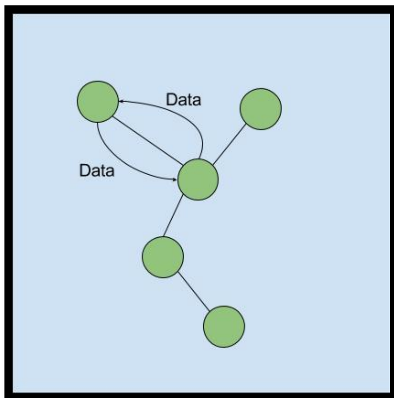

Figure 3: Communication of 1st-degree

Original Application Summary:
The original application models social networks for further analysis. The modeled network initially needs to match the extrapolated data representative of a given social network. To do this messaging rules were established to emulate the environment in the source network. Then the model is given several combinations of network characteristics to establish accurate modeling.

This model was used to better understand developing social networks such as online forums and gaming clans.

Game of Life <Cellular Automata>:

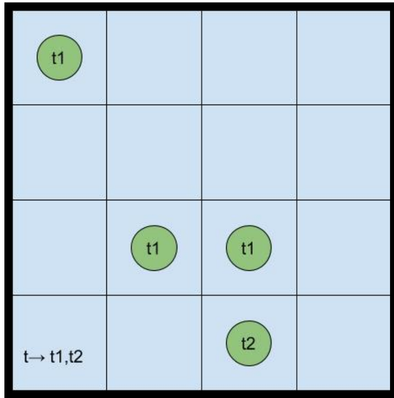

Figure 4: Illustrates a simple propagation

Original Application Summary:
The (Conway's) Game of Life illustrates the propagation and death of cells in its environment.

There are no longer studies being done on this model, because it has been exhausted of mathematical significance.

Dynamic Agent Applications

Tuberculosis:

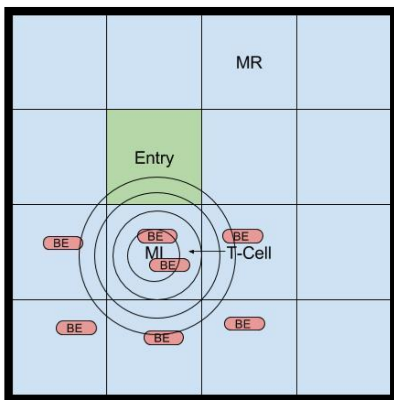

Figure 5: T-Cell response to infected macrophage

Original Application Summary:
The original application models the biological response from the inflammatory system to a Tuberculosis infection.

The model like most models made for biological simulations allowed researchers additional control and reduced

Virtual Design Team:

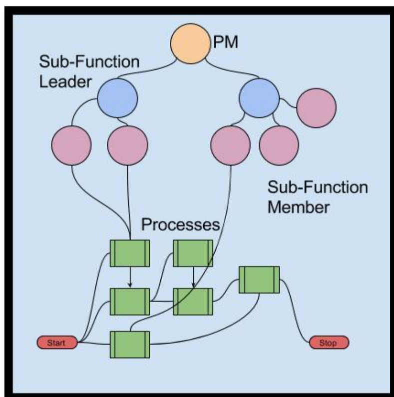

Figure 6: Illustrates VDT's architecture

Original Application Summary:
Virtual Design Team modeling evolved into a system where dynamic work could be done (i.e., surgical procedures). The basics included a generalization of employee hierarchy and a process graph as scene in the image to the left.

This model was developed for industrial and research usage, to identify procedural and personal bottlenecks and other efficiencies.

MatSim:

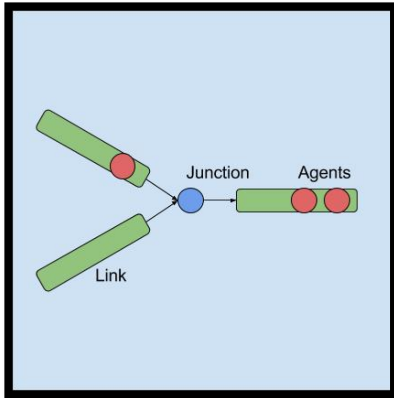

Figure 7: Illustrates converging roads

Original Application Summary:
MatSim is a competitor modeling framework that specializes in modeling traffic.

This framework has been used to understand traffic better in relation to events such as construction and new environmental elements. Though it still needs to encapsulate emergency states such as a car crash.

Model Design

Brain Grid <Neural Networks>

Brief Discussion of Implementation

The neural networks model will require two classes to be made, the grid and the neuron. As stated earlier the neuron will need to come in three types: active, inhibitory, and neutral. Thus, the neuron class should support an enum for the types ACTIVE, INHIBITORY (INACTIVE for brevity), and NEUTRAL. The complete environment will be represented by the grid, who is responsible for enforcing the toroid space, and safety of the synapse growth.

The synapse growth is limited by three conditions: INHIBITORY neurons, point-to-point synapses. and non-redundant synapses. With these condition the neuron class will need to support an enum for stages: VISIT1, VISIT2, ENACTED, CONNECTED, STOPPED, INDEF. The normal growth pattern utilizes the first three enums where VISIT1 and VISIT2 are used to denote alternating growth periods, and ENACTED denotes a grown neuron. The next two enums are used to denote "completed" synapses (i.e. synapses that will not be intercepted). Lastly INDEF will be used to denote the default state of non-active neurons because the growth simulation starts from the active neurons.

Code Snippet

Before a Neuron can grow it must ensure that it is a possible to do so. In the Brain Grid model, there are multiple things that can stop a synapse’s growth. For the method that does, it assumes that the connections between ACTIVEs are already handled, and like noted earlier a destination Neuron that is INACTIVE will stop it. Additionally, a destination Neuron that is completed will stop it. More complexly, redundant synapses cannot exist. To account for redundant synapses the unconfirmed synapse must first be used to identify the ACTIVE Neuron at the other end. At that point the neighbor of this ACTIVE Neuron will be searched through a similar traversal to confirm or reject the “new” synapse.

```
// Assumptions ACTIVE to ACTIVE pre-handled
// Returns -1 not safe, 0 safe, 1 connect
BrainGrid::GrowthFlag BrainGrid::SafeGrowth(Neuron& activeSrc, Neuron& dest)
{
  // Check for Invalid Growth Neighbors
  if (dest.neuronType == Neuron::INACTIVE
    || dest.neuronType == Neuron::ACTIVE
    || dest.neuronStage == Neuron::STOPPED
    || dest.neuronStage == Neuron::CONNECTED)
  return BrainGrid::STOP;
  else if(dest.neuronStage == Neuron::INDEF)
    return BrainGrid::SAFE;
  else
  {
    // Determine Destination Neighbor's Active Neuron
    Neuron* next = dest.neighbors[0];
    while(next->neuronType != Neuron::ACTIVE)
    {
       next = next->neighbors[0];
    } // End traversal of dest Neuron
```

```
      // Iterate through all Dest's Active's Immediate Moore Neighborhoods
      for(int i=0; i < 8; i++)
      {
        // Skip Unused Neighbor
        if(next->neighbors[i] == static_cast<Neuron*>(NULL)) continue;
        // Skip Redundant Path
        if(i == dest.nextCell) continue;

        // Trivial Connection to activeSrc
        if(next->neighbors[i] == &activeSrc)
        {
          return BrainGrid::STOP;
        }
        else if(next->neighbors[i]->neuronStage == Neuron::CONNECTED)
        {
          // Traverse Synapse to Determine Corresponding Active (Again)
          Neuron* tmp = next->neighbors[i];
          while(tmp->neuronType != Neuron::ACTIVE)
          {
            if(tmp->neighbors[2] != static_cast<Neuron*>(NULL))
              tmp = tmp->neighbors[2];
            else
              tmp = tmp->neighbors[1];
          } // End traversal for next->neighbor[i]'s ACTIVE

          // Determine if Corresponding Active is Redundant
          if(tmp == &activeSrc)
           return BrainGrid::STOP;
        } // End check for next->neighbor[i]'s ACTIVE
      } // End CONNECTED check of dest's ACTIVE Neuron
      return BrainGrid::CONNECT;
    }
  } // End Safe Growth
```

Financial Modeling <Clustered Agents>

Brief Discussion of Implementation

The clustered agents model will require two classes (agent, and cluster) and a driver. The reason why the model requires the model is because the simulations phases must be synchronized and centralized. On the other hand, the cluster is made to be generic enough to pass one-to-one and multicast messages, such as being responsible for out and in buffers that notify agents and update messages. Lastly agents need to have an API such that they can model data updates, data transfers, and connection transactions.

Code Snippet

To accommodate communication between agents each need know what are the simulation phases. To initiate the simulation each agent must keep the phase consistent, and through the messages sent during those phases.

```
//------------------------Bank Implementation------------------------------//
std::string Bank::initMsg(const int phase)
{
   std::string retVal = "";
   if(phase == 0)
     TotalProposedLoans = 0;
   return retVal;
}

// Assumes msg was sent correctly
std::string Bank::runAgent(const std::vector<std::string>& msg)
{
   if(msg[0] == "0-0-0") // Recieved Request
   {
      std::stringstream ss;
      ss << "0-1-1_" << msg[2] << "_C" << clusterId << "A" << agentId << "_" + msg[3];
      // Check
      if((NetAssetValue - TotalProposedLoans )/2 > atoi(msg[3].c_str()))
      {
         TotalProposedLoans += atoi(msg[3].c_str());
         std::cout << "Loan Recieved: " << ss.str() << "\n";
         return ss.str();
      }
      else
        return "";
   }
   else if(msg[0] == "0-2-1") // Recieved Confirmation
   {
      int firmLoan = atoi( (msg[3].substr(0,msg[3].length() -2)).c_str() );
      if(msg[3].at(msg[3].length() - 1) == 'N') //Check if Firm has already been serviced
      {
        TotalProposedLoans -= firmLoan;
      }
      else
      {
        std::size_t agentIdOffset = msg[2].find('A');
        int firmId = atoi( (msg[2].substr(agentIdOffset+1)).c_str() ); // CHECK

        // Update Assests/Loan
        NetAssetValue -= firmLoan;
        FirmAccounts.reserve(firmId+1);

        // Finialize Loan
        FirmAccounts[firmId] = firmLoan;
        std::cout << agentId << " Loan Accepted: " << firmLoan << "\n";
        std::cout << agentId << " Net Asset Value " << NetAssetValue << "\n";
      }
     return "";
   }
}
```

Social Network Modeling <Network of Static Agents>

Brief Discussion of Implementation

The model will require two entities; a graph and agent. Since the topology of the environment is a bidirectional graph, storing the connection between nodes will be done in a triangular adjacency matrix to cut down on half the memory. Additionally, note that in a social network on a preexisting system like Facebook will require that "friends" (first-degree neighbor) not be limited, therefore a node can have 0 "friends" to even 500 "friends". The original application required statics to be kept throughout the simulation to gauge the communication of the network, including a network's tendency to communicate with nodes outside of its immediate network. Therefore, after constructing the adjacency matrix, each node must also know what nodes are n-degree neighbors for future usage since a node can communicate with any other node without traversing the graph. For each node to know each degree of neighbor, the initialization phase must include a modified breadth-first search algorithm to be run on all nodes.

Code Snippet

The modified breadth-first search algorithm determines the nth-degree of neighbor for a node by utilizing two queues. The elements within the process queue is representative of the same degree. Additionally, the next queue are the corresponding elements, to the elements within the process queue. The process queue removes an element when it places that element's neighbors into the next queue. Once the process queue is emptied, it is swapped with the next queue.

```
void Graph::degree_BFS()
{
  std::cout << "Begun BFS \n";
  // Run per Node
  for(int i = 0; i < graphSize; i++)
  {
    std::cout << "Agent: " << i << "\n";
    int degree = 1;
    std::queue<int> process;
    std::queue<int> next;
```

```
        // Initialization process
        process.push(i);
        // BFS
         while(!process.empty())
         {
            int current = process.front();
            process.pop();

            // Search for Adjacent Agents
            for(int m = 0; m < graphSize; m++)
            {
               // Adjacent Agent and Redundancy Check
               if(getVal(current,m) == 1)
               {
                  if(current == i)
                  {
                     next.push(m);
                  }
                  else if(isValid(i,m))
                  {
                     // Update for Agent i
                     setVal(i,m,degree);
                     next.push(m);
                  }
               }
            } // Finished Adjacenct Search

            // Check for next degree
            if(process.empty())
            {
               process.swap(next);
               degree++;
            }
         }
      } // Completed BFS on all Agents
   }
```

MatSim

Brief Discussion of Implementation

The model will require three entities; graph (junction information), agent (car), place (road). Unlike the topology for social network modeling, MatSim represents roads which includes one-way roads. Therefore, each portion of the road will be represented by a one-way road and a value in a adjacency matrix will represent the junction between roads. To simply model the motion of traffic, each agent (car) will have a predetermined path that is calculated via Dijkstra's Algorithm and stored within each agent.

Code Snippet

Instead of having an agent (car) class, the test maintains a list of paths, where paths are representative of a route. Any road holding a car, is holding an id of agents for a graph maintained structure mentioned earlier that holds all car paths. The simulation starts out with each road and attempt to move every car based off the stored paths. If the car can move to the next road is pops off a road id from its path. To visualize; "- "symbolizes completion of a car's route, "*" symbolizes a moving car, and "$" symbolizes a stopped car. Because of the sequential nature one car can move faster than intended if the following road ids are greater than the one it currently sit upon.

```
// Run the simulation
void Graph::runAgents(int TIME_MAX, int maxAgents) {
   // Initializies Agents before simulation
   initAgents(maxAgents);
   int numAgents = maxAgents;

   // Printing Output Header
   std::cout << "t" << std::setw(5);
   for(int printHeader = 0; printHeader < numAgents; printHeader++)
     std::cout << "v" << printHeader << std::setw(5);
   std::cout << "\n";

   // Runs Simulation
   for(int time = 2; time < TIME_MAX || numAgents > 0; time+=2)
   {
     // Print Output Body
     std::cout << time/2 << std::setw(5);
     for(int printOut = 0; printOut < maxAgents; printOut++)
       std::cout << agents[printOut].back() << std::setw(6);
     std::cout << "\n";

     // Go through each link
     for(int i = 0; i < size; i++)
     {
       int curSize = (links[i].agentIds).size();
```

```
        // Go through each link's agentId
        for(int j = 0; j < curSize; j++)
        {
          std::vector<int> tempAgent = agents[(links[i].agentIds).front()];
          std::cout << (links[i].agentIds).front();
          // Finished Traversal
          if(tempAgent.size() == 1)
          {
             (links[i].agentIds).pop();
             std::cout << "-";
             numAgents--;
             continue;
          }
          int nextLink = tempAgent.back();
          // Check next link
          if((links[nextLink].agentIds).size() != Link::MAX_CAPACITY_AGENTS)
          {
            std::cout << "*";
            // Move Agent's id to the next Link
            (links[nextLink].agentIds).push((links[i].agentIds).front());
            // Update Agent's path
            agents[(links[i].agentIds).front()].pop_back();
            // Remove that Agent's id from the current queue
            (links[i].agentIds).pop();
          }
          else // Unable to move
          {
            std::cout << "$";
           break;
          }
        }// Completely traversed Link's Queue
      } // Finished One Iteration
      std::cout << "\n";
    } // Finished Simulation
  }
```

Tuberculosis

Brief Discussion of Implementation

The model will require another toroidal graph environment with generic agents. The reason why generic agents were chosen is because they simplify the simulation greatly. Instead of having T-Cells and Macrophage, there are inactive/active and infected/noninfected flags. Because there are several place elements and flags within the system, the dispersal method was made generic to accommodate bacterial growth, chemical diffusion, and bacterial elimination.

References

Bryan C. Thorne, Alexander M. Bailey, “Multi-cell Agent-based Simulation of the Microvasculature to Study the Dynamics of Circulating Inflammatory Cell Trafficking, Annals of Biomedical Engineering, 2007

Bryan C. Thorne, Alexander M. Bailey, Shayn M. Peirce, "Applying Agent--Based Modeling to Studying Emergent Behaviors of the Immune System Cells", Briefings in Bioinformatics Vol 8 (No. 4) 245-257, 2007

Caroline C. Krejci, Benita M. Bearmon, “Modeling Food Supply Chains Using Multi-Agent Simulation”, Winter Simulation Conference, 2012

Chee Siang Ang, Panayiotis Zaphiris, “Simulating Social Networks of Online Communities: Simulation as a Method for Social Design”, Human-Computer Interaction, 2009

Eric Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems", PNAS, 2002

Fukuda, M., Stiber, M., Salathe, E., & Kim, W., “CDS&E: Small: Multi-Agent-Based Parallelization of Scientific Data Analysis and Simulation.”, 2013

Herbert Gintis, "Long--range Research Priorities in Economics, Finance, and the Behavioral Sciences", Santa Fe Institute, 2010

Jose L. Segovia-Juarez, Suman Ganguli, “Identifying control mechanisms of granuloma formation during M. tuberculosis infection using an agent-based model”, Journal of Theoretical Biology, 2004

Kent D.Miller, “Agent-Based Modeling and Organization Studies: A critical realist perspective”, Organization Studies, 2015

Luis F.O. Jacintho, Andre F.M. Batista,Terry L. Ruas, Maria G.B. Marietto, Fabio A. Silva, “An Agent-Based Model for the Spread of the Dengue Fever: A Swarm Platform Simulation Approach”, Spring Simulation Multiconference, 2010

M.-H. Chang, J.E. Harrington Jr, "Chapter 26: Agent-Based Models of Organizations", Handbook of Computational Economics, 2006

MASS C++, Parallel-Computing Library for Multi-Agent Spatial Simulation in C++, February 24th, 2015, < http://depts.washington.edu/dslab/MASS/docs/MassCpp.pdf >

Olivier Kooy, "Understanding the causal relations in organizational structures of project teams", System Engineering, Policy Analysis and Management, TU Delft, Master Thesis, 2012

Rebuilding the MOSAIC, Fostering research in the social, behavioral, and economic science at the national science foundation in the next decade, 2011

Thomsen, Jan, Raymond E. Levitt, John C. Kunz, Clifford I. Nass, Douglas B. Fridsma, “A Trajectory for Validating Computational Emulation Models of Organizations” Journal of Computational & Mathematical Organization Theory, 1999