DNAgents 3.0: Genetically Engineered Mobile Agents for a Dynamic Network Topology

John Bovatsek
University of Southern Mississippi

The University of Southern Mississippi

DNAgents 3.0: Genetically Engineered Mobile Agents for a Dynamic Network Topology

by

John Bovatsek

A Thesis

Submitted to the Honors College of

The University of Southern Mississippi

in Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science in

the Department of Computer Science

May 2012

Approved by

Dia Ali, Ph. D.
Professor of Computer Science

David R. Davies, Ph. D., Dean
Honors College

ABSTRACT

DNAgents 3.0: Genetically Engineered Mobile Agents for a Dynamic Network Topology

by John Bovatsek

May 2012

Mobile agents are a relatively new topic in the realm of computer science research. They are being researched throughout the world by many scientists in order to ascertain their viability in real world applications. They have many disadvantages that keep them from widespread adoption. With his graduate dissertation *Genetically Engineered Intelligent Mobile Agents*, Kackley opened a whole new area of this research by combining mobile agents with genetic algorithms. These algorithms model the natural process of evolution to evolve solutions for problems.

This thesis is part of a research group effort to expand on Kackley's work in the Database Research Lab for Intelligent Agents at the University of Southern Mississippi. The goals of this research were to first gain a thorough understanding of both mobile agents and genetic algorithms and to augment DNAgents 2.0 with new capabilities. Currently, in Kackley's implementation of DNAgents 2.0, there is no mechanism to facilitate a changing network topology. This behavior is not suited for a live network environment in which computers could be connecting or disconnecting to the network throughout the mobile agent's task. In order to take advantage of the full potential of mobile agents, one needs to give them ability to adapt to an ever-changing network.

In this research we propose an addition to Kackley's dissertation project to allow adding arbitrary nodes to the network. In our DNAgents 3.0, the simulation is now able to add new nodes with either random or specific links to other nodes. The agents are immediately able to seamlessly move to the new node by reaching it through one of these linked neighbors. This allows for the genetically engineered agents to operate on a dynamic network topology.

TABLE OF CONTENTS

List of Figures

1. Introduction

An agent, defined as a static agent, is a set of algorithms that has the ability to communicate with other agents, change based on the environment (proactive and reactive), and to perform tasks at its own command. A mobile agent, therefore, is an agent encompassing all of these traits that has the added ability to stop its execution, save all the data it has accumulated, and move across the network to resume execution on another host [1].

Networks between computers are essentially nodes connected via channels of communication. However, these networks are now becoming more complex. They are becoming larger and more complex due to the proliferation of web-connected computers, cell phones, servers, televisions, etc. These devices, however, are not all connected to a reliable or motionless connection. Cell phones, for example, are constantly in movement and need to connect to different nodes of the network as they move. In order for a mobile agent to adjust to rapidly changing networks, the topology that the agent uses must dynamically change with the network [2, 3].

Dynamic network topologies in a mobile agent system is necessary in applications where every node of the network does not have the ability, or desire, to be connected to the network at all times. This could mean a simple deletion of one node from the network during the agent's lifespan or a large amount of moving nodes throughout the covered space. The main focus of research in this area is on the latter.

In his graduate dissertation [1], Kackley combined mobile agents with genetic algorithms in order to create what he calls genetically engineered intelligent mobile agents. A genetic algorithm is one that is modeled to act like natural evolution. With this combination of mobile agents and genetic algorithms, the agent can spawn new algorithms to do tasks more efficiently. It can also choose to stop execution while others in its population continue to evolve or mutate while moving across the network. Kackley's dissertation resulted in DNAgents 2.0. This program is the first part of a larger goal to create agents that can evolve virus protection. In his implementation, however, problems can arise when one wants to change the network topology.

The network is made up of multiple nodes that are linked together. The agents can then traverse the network using these links. Currently, in Kackley's implementation of DNAgents 2.0, there is no mechanism to facilitate a changing network topology.

This behavior is not suited for a live network environment in which computers could be connecting or disconnecting to the network throughout the mobile agent's task. In order to take advantage of the full potential of mobile agents, one needs to give them ability to adapt to an ever-changing network.

1.1. Mobile Agents

Mobile agents are agents that have a "behavior, state, and location" [4]. The authors of [4] go on to describe two categories of mobility for Mobile Agents. The category with the highest degree is called *Strong Migration* while its counterpart is *Weak Migration*. In strong migration, the agent controller grabs the whole state of the mobile agent (data and execution), packs it up with the code, and sends it all to the next location. Weak migration, on the other hand, only sends the data state, leaving out the state of execution. This is easily overcome by setting variables inside the data to represent the execution state. After the state is grabbed from the variable, a method in the execution needs to decide where to go to resume execution based on that state variable.

In [1], Kackley describes many advantages to using mobile agents. The ones that stand out the most are the reduction of network traffic and scalability. Mobile agents are more efficient than client-server applications due to their very nature. This can be seen in the figure 1.

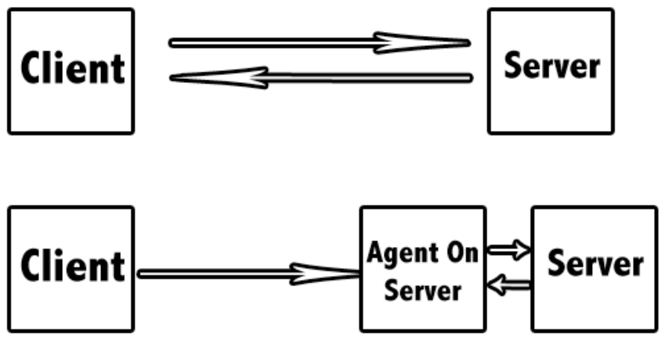


Figure 1: Client-Server vs. Client-Agent-Server

In the above figure, it can be seen that the client-server application must send many messages back and forth. Mobile agents, however, avoid this by sending the whole agent to the server and sending the messages locally. In their experiments, the authors of [5] came to the conclusion that the overall efficiency of mobile agents vs client-server applications was dramatically in favor of mobile agents when utilizing large networks. They found the mobile agents, being able to hop directly to the server, were able to be less effected by network bandwidth. The nature of mobile agents not only increases performance in applications where there are many messages to be sent, but it also ensures that all messages are received by the server and not lost due to connectivity issues.

Even with the many advantages, there are disadvantages to mobile agents that are holding them back from widespread use. One such disadvantage is security. In the IBM research report *Mobile Agents: Are They a Good Idea?* the authors provide an overview of security concerns regarding mobile agents [6]. The first concern the authors list is authentication. Some servers may wish to check that the agent comes from a trusted server. With the agents moving across the network, how can the accepting server know the agent originated from an authorized user?

The second security issue is determining whether or not the code the agent wishes to execute will not harm the host. If an agent has been manipulated to inflict some harm on the host server, how does the host catch it? The very idea of mobile agents is the server allowing a program to download onto it and executing the program.

The final security concern the authors describe is “the agent’s ability or willingness to pay for services provided by the server (unless these are free).” The concern is the agent could come in and access whatever it needs without paying for the service. It also could have been modified to fake currency in order to fool the server into thinking the services were paid for.

Another disadvantage, that really halts the wide adoption, is the complexity of mobile agents [7]. There comes a point when the question arises if the added complexity is worth the boosts in efficiency. When a client-server application can do the task well enough with significantly less complexity, why use mobile agents?

1.2. Genetic Algorithms

Genetic algorithms were developed in the 1950s and 1960s in order to use evolution as a model for optimization [8]. Genetic algorithms are modeled after the natural process of evolution. Figure 2 shows a general diagram for a genetic algorithm.

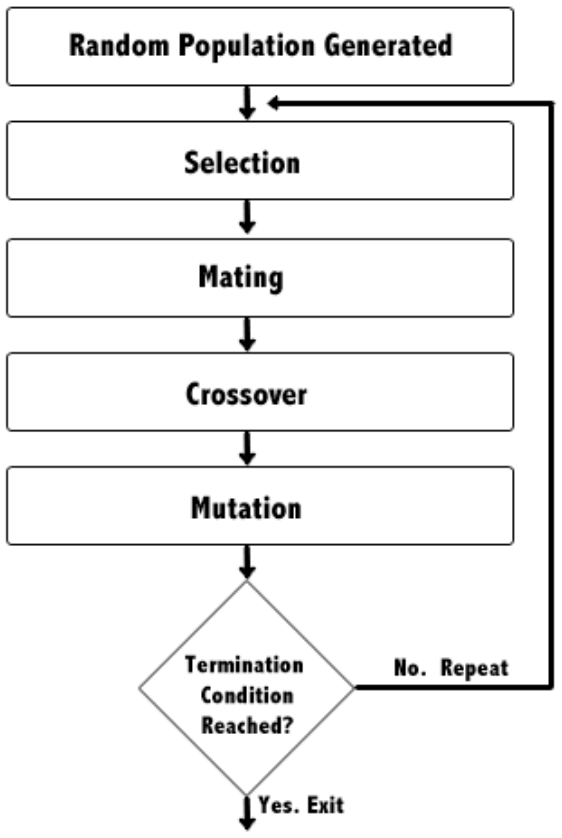


Figure 2: Diagram of Genetic Algorithms

Typically, the algorithm starts with a randomly generated "solution" to a problem, called the chromosome. This solution is then sent to the selection phase. This process intends to give the most fit of the population higher chances to move on to the next generation [9]. The selection phase improves the population's average quality by giving the most fit the best chances to survive. To achieve this, some method is used to determine the best candidates to mate [1].

There are several ways to design a selection algorithm. Three such ways are elitist, tournament, and roulette-wheel. In the elitist algorithms, only solutions that reach a certain level of fitness are selected [1, 9]. In tournament, however, groups of solutions are selected at random and the fittest of that group is selected. Roulette-wheel assigns a probability to be selected based on the fitness of the solution [1, 9].

These types of selection affect the diversity of solutions in the population. The elitist type of selection provides the least amount of diversity, as it gives no chance for the less-fit solutions to move on. With tournament and roulette-wheel algorithms, less fit solutions have a chance, albeit smaller, to be selected. The elitist algorithm is also less efficient as it requires the solutions to be sorted before selection happens. The authors of [9] state the complexity to be O(N ln N) where N is the number of solutions in the population.

In general, the best solutions are selected for mating and sent to the mating process. After the top chromosomes are chosen, they need to create a new generation of chromosomes in order to attempt finding a better solution. In the mating section of the algorithm, it is decided how many, if any at all, "offspring" the solution will have. If the probability function states the two solutions are going to mate, they are sent to the recombination, or crossover, section.

In crossover, the solutions are randomly merged. There are different ways to merge the parent solutions in order to get the child. Two common ways are the single-point crossover and cut-and-splice crossover [1]. The first way is single-point crossover, in which the parent chromosomes are the same length, and are split at the same point. The first half of the split is taken from the first parent, and the second half from the second parent. This results in the offspring's chromosome being the same length as the parent chromosome. The other way is multi-point crossover. It is the same basic idea, but multiple crossover points are selected. This could result in a chromosome being a different size than the parent.

In the final stage of the genetic algorithm, the chromosomes go through mutation. In this process, "individuals in the population are randomly altered" [1]. Mutation ensures that the population doesn't eventually become stagnant by merging into the same essential solution. It also allows diverse features to be reintroduced that might have been cut out during crossover [10]. Once this stage is complete, the program will either check here, or in the fitness function, whether or not a suitable solution has been found. If not, it will do the whole process again.

As stated in [8], genetic algorithms are used in many areas of scientific engineering. One way, as mentioned above, is to use them for optimization. The algorithms can find the best solution to a problem over many generations by checking thousands of different solutions and stitching together pieces of the better solutions. They can also be used in automatic programming, in which instruction sets are randomly evolved to some end. Other uses include machine learning, economic models, immune system models, and ecological models.

2. Problem Statement

As mentioned in the introduction, the mobile agents need functionality to adapt to dynamic topologies. In a live network, computers are not available at all times. It might be as simple as a computer being turned off temporarily for maintenance. In another case, the network could consist of home computers which are not connected to the network or even powered on at all times. No matter the reason, a computer being added to or removed from the network should not halt the operation of the agent system as a whole. Kackley's DNAgents 2.0 simulation did not include this functionality. In this research we propose an addition to Kackley's dissertation project to allow adding arbitrary nodes to the network.

3. Related Work

A large quantity of related work on dynamic networks involving mobile agents mostly resides in the field of creating routing tables for multi-hop network traversal.

In [2], the authors discuss the use of routing agents to discover and create routing information for a network. The authors first create a network of nodes to be traversed. The nodes have radio ranges that allow the node to connect to any other node within that range. Then, mobile agents are created that traverse node to node in order to discover the network. Each agent will take into account all the links, or neighbors, of its current node and decide to move to one of them. As it migrates to the next node, it captures information about the link between the two nodes. Presumably, the authors are capturing the latency between the two nodes. The agent then saves this data into its personal history and updates the routing table of the current node. Using these routing tables, future agents can find the shortest path across the network.

The authors, however, do not cover adding arbitrary nodes to the network, only discovering nodes that are already there and linked to other nodes. The authors explicitly state, “Every node knows who its neighbors are.”

In [11], the authors reference [2] and note the use of a static network. They recreate the results of [2] by setting up a network of nodes that have radio ranges. However, these nodes are now dynamic in that the position and radio ranges can change. In addition they add stigmergy to the agents. Stigmergy is a common concept in insects in which a trace of a past action that influences a future action. The authors use a method they call leaving a footprint in which the agent tells the node where it is going next. The following agent will read this location from the node and know not to connect to the same node as the previous agent. This allows the network topology to be discovered faster due to less redundancy. However, when it comes to the fixing the "unrealistic assumption" of a static network, the authors do not seem to address the creation of new nodes or changing links. Obviously, links are changing as nodes go out of their radio range, but this is not discussed in the paper.

Kackley's DNAgents 2.0 simulation attempts to combine genetic algorithms and mobile agents. The implementation first creates a set of mobile agents. At birth, the mobile agents create a random instruction set in an attempt to evolve a hopping behavior. Instructions include the typical computer instructions such as add, subtract, and etc. The important instruction in this experiment, however, is send. The instruction is generated with random addresses so that agent will be sent to some node (if it exists) when it reaches that instruction.

After instruction generation, the mobile agent goes through the basic process of genetic algorithms. The agent's "hops" are tracked and the agent goes through crossover and mutation. The fitness function takes into account the amount of hops the agent has successfully completed in order to evolve an agent that will traverse the network efficiently.

The agents utilize a simulation environment in which a pseudo-network is created containing nodes, which only host agencies, and links, which connect nodes. The simulation steps through each iteration of the genetic algorithm and mobile agent process and allows the viewer to keep track of the population's status.

In [1], Kackley states the motivation behind DNAgents 2.0 is to eventually protect networks against intrusion and attack. Kackley describes the possibility of a genetically engineered mobile agent that uses genetic algorithms to discover and exploit vulnerabilities on a computer network. Since the current method of protection is reactive, standard protection methods would be useless since the entities would constantly evolve new exploits and not be the same as the previous generation of entities. Kackley discusses a "benign counterpart" to this threat which uses genetically engineered mobile agents which are also evolving to discover exploits and viruses in computers connected to the network. The agents then pass its newly found exploits to fellow agents through message passing and genetics. The agents then try to continuously attack any virus or exploit it discovers.

Kackley is not the only researcher looking to use Genetic Algorithms to protect networks. In an article about his current research, Dr. Errin Fulp is quoted saying many security problems are due to poor configuration [12]. Seeing this as the problem, he and a graduate student named Michael Crouse are attempting to use genetic algorithms to constantly change the network environment in order to ward off attacks [13]. In [13], the authors propose using genetic algorithms to create a “moving target” configuration system. This type of system periodically changes the configuration of computers in order to reduce the risk of a whole network being infected by malicious code on one connected computer. The author’s system uses genetic algorithms by modeling the computer configuration as a chromosome. The chromosome is then combined with other chromosomes to create a whole new configuration. The newly evolved configuration is tested based on its resistance to an attack and ranked among other configurations. This creates diverse configurations on the network so in the event that a computer on the network is compromised, other connected computers are likely to not have the same vulnerability.

Kackley’s DNAgents 2.0 project was later picked up by researcher Andrew Cordar [14]. Cordar’s work on this project was in two parts. His first project was to convert DNAgents 2.0 to JAVA from its original C++ code. After this conversion, he was then tasked with adding the ability to delete arbitrary nodes from the network.

In order to achieve this, Cordar first checks if there are any agents active on the node. If there are none, the node’s “live” variable is set to false. In the network’s list of nodes, the node is replaced with a new node with an ID of -99. The ID -99 is later used to check if a node is just a deleted node or not. This is done so the indexes on the node list, which are used in other parts of the simulation, are not changed.

If the node has living agents, however, they must first be killed before the node can be deleted from the network. Looping through each agent in the Node’s agency, Cordar checks if the agent is in transit. If they are currently leaving the node, he ignores them. However, if they are still executing commands on the node, they are killed off. He then does the same process as if the node had no agents and replaces it with a node with a ID of -99. By replacing the node with a dummy node to keep the indexing consistent is the best option within the current architecture of the simulation.

4. Methods

In modifying Kackley's DNAgents 2.0, adding new nodes to the network required a few steps. First, the new node must be created. Then, any links from the new node to other nodes in the network had to be added. To do this I created a function in the simulation that accepted an array of links. This array contains the IDs of the nodes the new node is to be linked to. The method is shown in Figure 3.

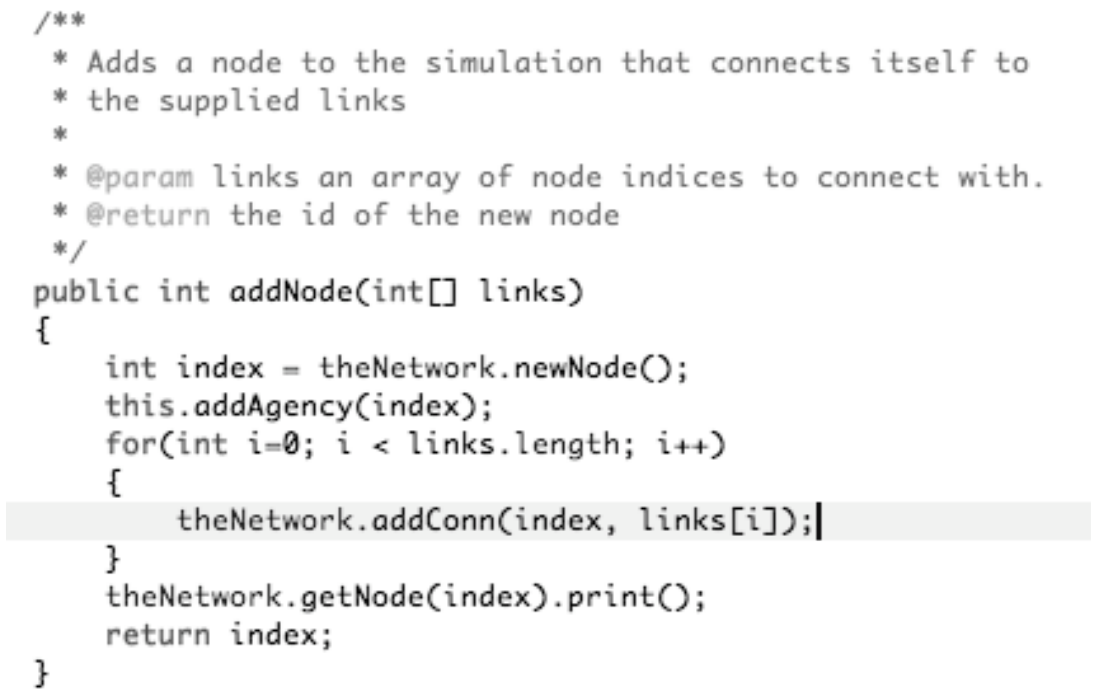
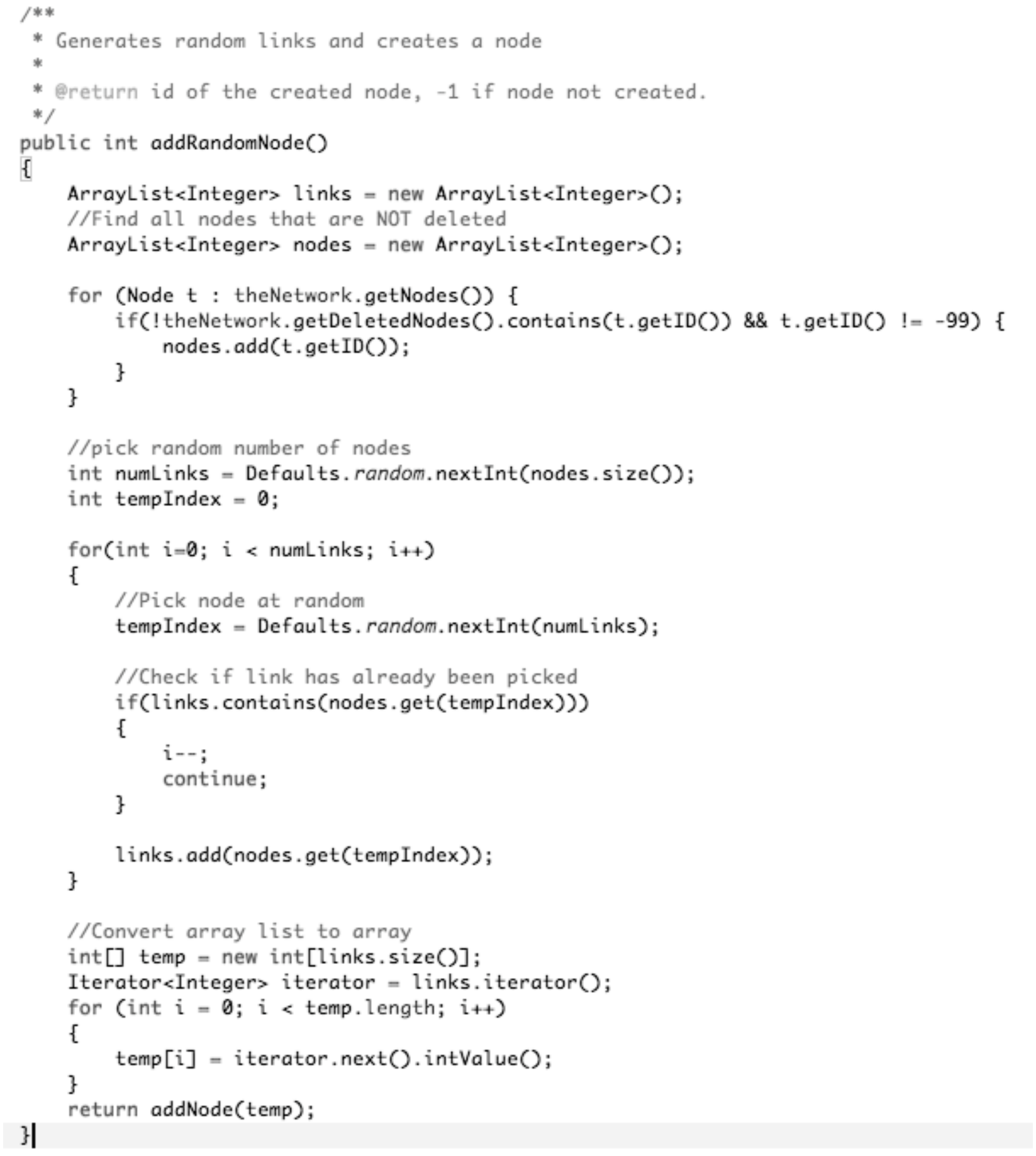

```
/**
 * Adds a node to the simulation that connects itself to
 * the supplied links
 *
 * @param links an array of node indices to connect with.
 * @return the id of the new node
 */
public int addNode(int[] links)
{
    int index = theNetwork.newNode();
    this.addAgency(index);
    for(int i=0; i < links.length; i++)
    {
        theNetwork.addConn(index, links[i]);
    }
    theNetwork.getNode(index).print();
    return index;
}
```

Figure 3: addNode method

Next, the way the simulation sends agents needed to be modified.

Kackley's genetically engineered mobile agents generate random instructions and random addresses for the parameters to each instruction. In the case of the send instruction, a random address is generated detailing which node to send the agent to. In an ideally generated instruction set, that address would be generated using the getNeighbor instruction. However, as the instructions are randomly generated, this is most likely not the case so the send function for the simulation must check if the destination is a neighbor to the current node. Kackley's implementation did not perform this check so any node could connect to any other node, regardless of the network topology.

After nodes could be successfully added, I then added methods to randomly add nodes with an arbitrary amount of links to an arbitrary set of nodes and also to delete a random node. For example, the code for adding random node is shown in Figure 4.



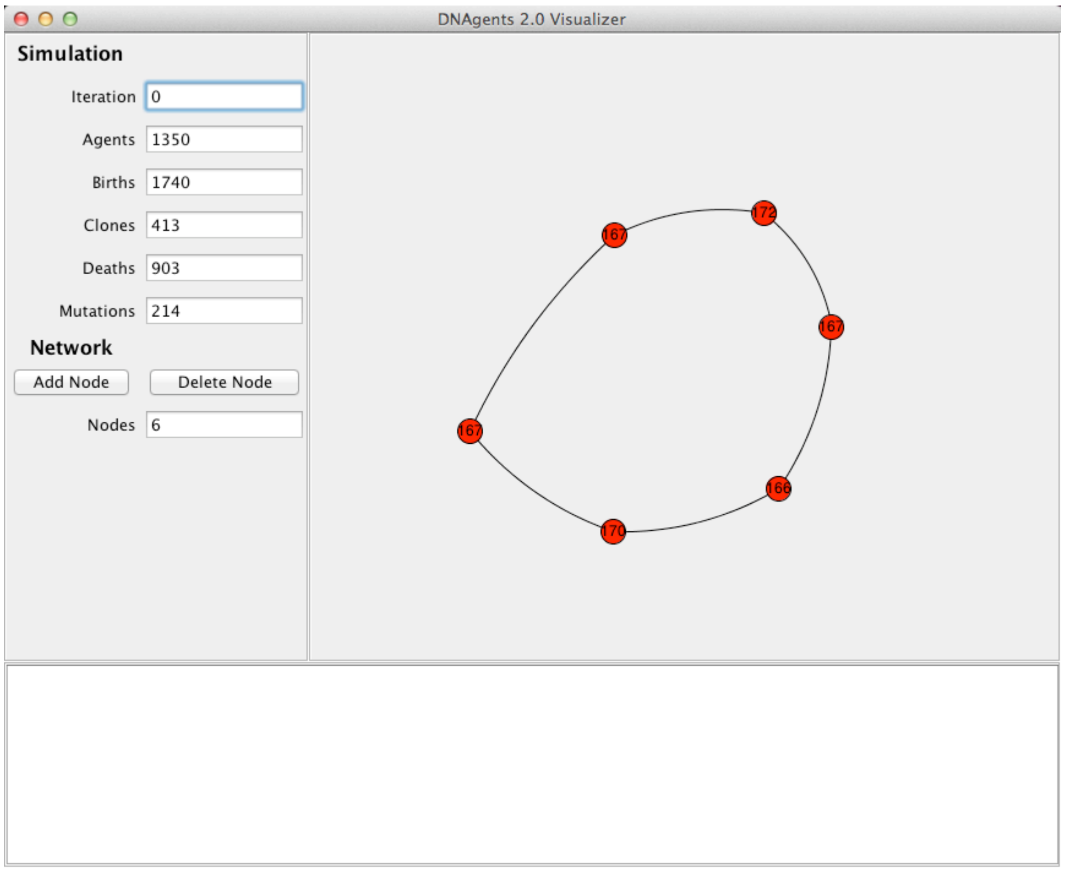
```
/**
 * Generates random links and creates a node
 *
 * @return id of the created node, -1 if node not created.
 */
public int addRandomNode()
{
    ArrayList<Integer> links = new ArrayList<Integer>();
    //Find all nodes that are NOT deleted
    ArrayList<Integer> nodes = new ArrayList<Integer>();

    for (Node t : theNetwork.getNodes()) {
        if(!theNetwork.getDeletedNodes().contains(t.getID()) && t.getID() != -99) {
            nodes.add(t.getID());
        }
    }

    //pick random number of nodes
    int numLinks = Defaults.random.nextInt(nodes.size());
    int tempIndex = 0;

    for(int i=0; i < numLinks; i++)
    {
        //Pick node at random
        tempIndex = Defaults.random.nextInt(numLinks);

        //Check if link has already been picked
        if(links.contains(nodes.get(tempIndex)))
        {
            i--;
            continue;
        }

        links.add(nodes.get(tempIndex));
    }

    //Convert array list to array
    int[] temp = new int[links.size()];
    Iterator<Integer> iterator = links.iterator();
    for (int i = 0; i < temp.length; i++)
    {
        temp[i] = iterator.next().intValue();
    }
    return addNode(temp);
}
```

Figure 4: addRandomNode method

This code begins by finding all of the nodes that are not deleted, putting the IDs of each node into an ArrayList. It then decides on a random number of links to create. Next, it picks random nodes, being sure they have not already picked before, and adds those to an ArrayList of links to add. Once that is converted to an array, it is sent to the add method.

5. Results

In order to show results, a visualizer was created that utilizes Java Swing elements for the GUI and the Java Universal Network/Graph Framework (JUNG) to visualize the network. Figure 5 illustrates the basic setup showing a small network.

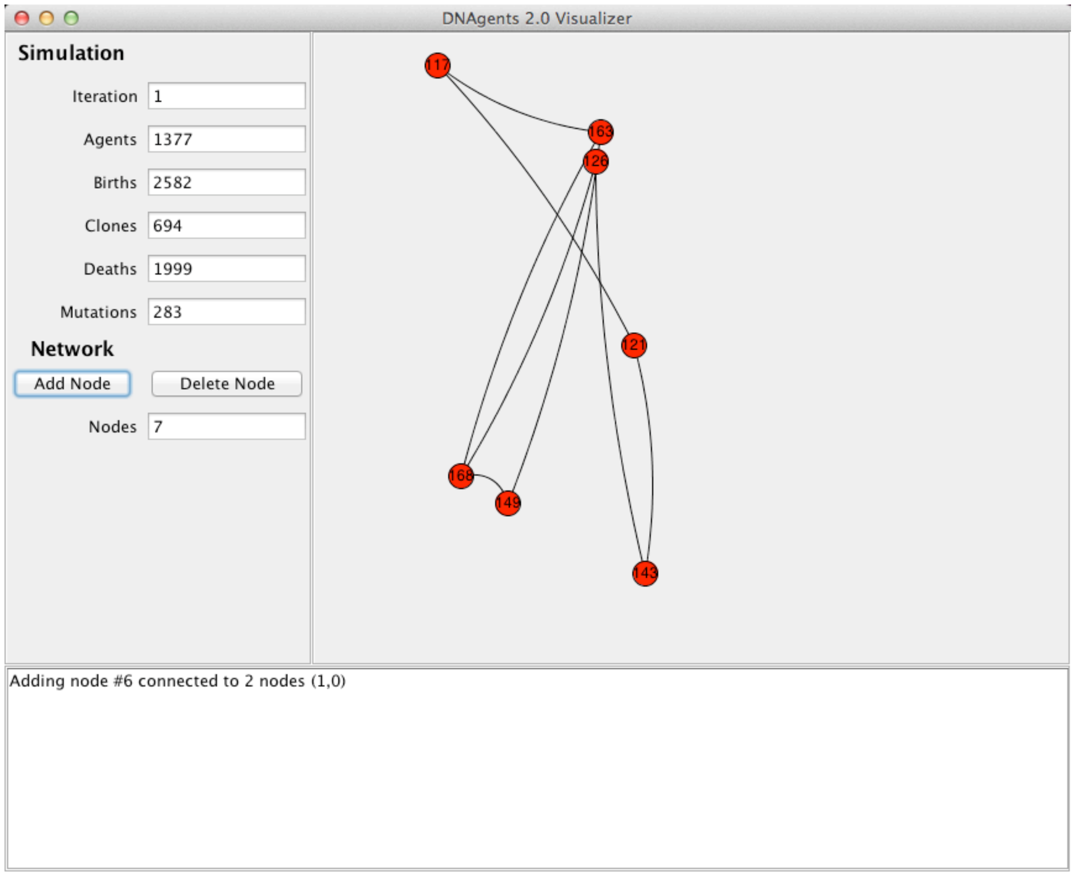


Iteration 0
Agents 1350
Births 1740
Clones 413
Deaths 903
Mutations 214
Nodes 6

Figure 5: Basic screen at program start

The visualizer shows the simulation's iteration number and the number of agents, births, clones, deaths, and mutations. It also gives the options to add and delete a node with a counter for the amount of nodes. The actual visualization of the network using JUNG shows the nodes utilizing a self-organizing layout algorithm. Each node shows the number of agents residing on the node. It is important to note these numbers do not add up to the number of total agents due to the number of agents in transmission.

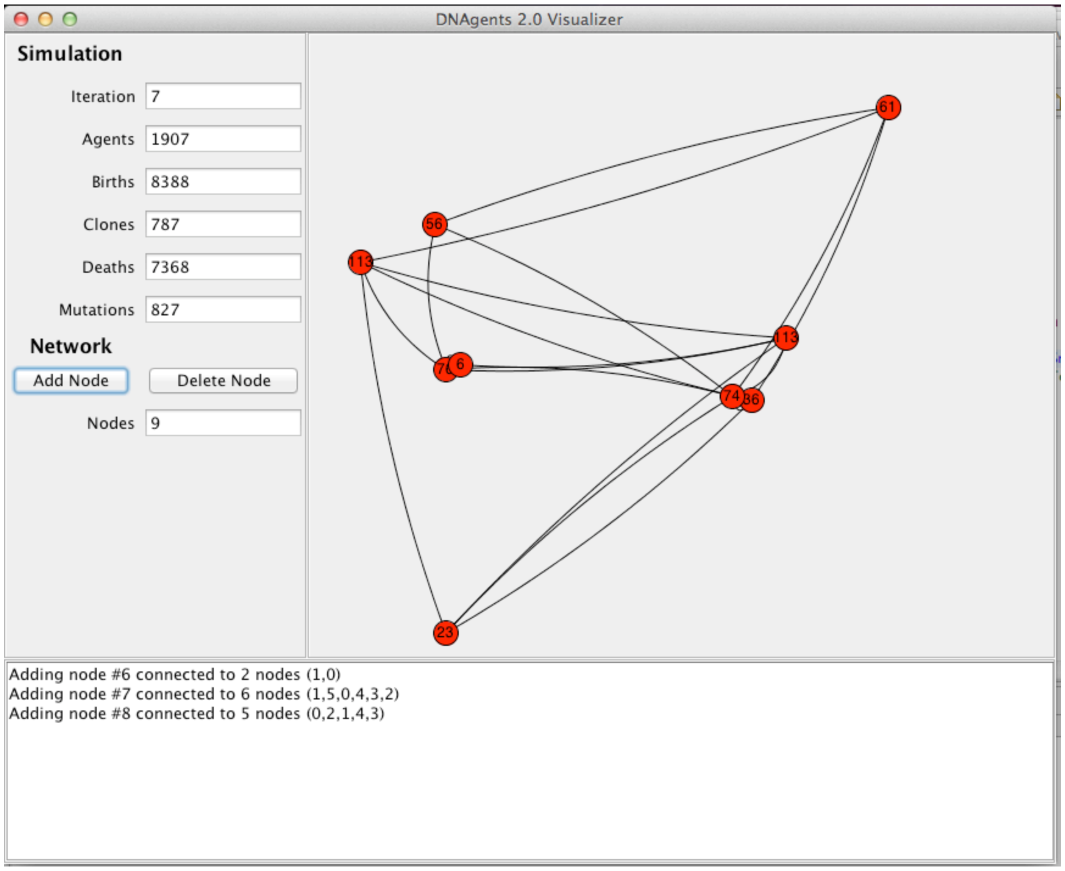
Adding a node can be done at any point of the simulation. Once the button is pressed, or the function is called in the program, the node is added to the network with the supplied links. Agents can take advantage of the new nodes immediately. The updated visualizing is shown in Figure 6 and Figure 7.



Iteration	1
Agents	1377
Births	2582
Clones	694
Deaths	1999
Mutations	283
Nodes	7

```
Adding node #6 connected to 2 nodes (1,0)
```

Figure 6: A node added to the network



Iteration 7

Agents 1907

Births 8388

Clones 787

Deaths 7368

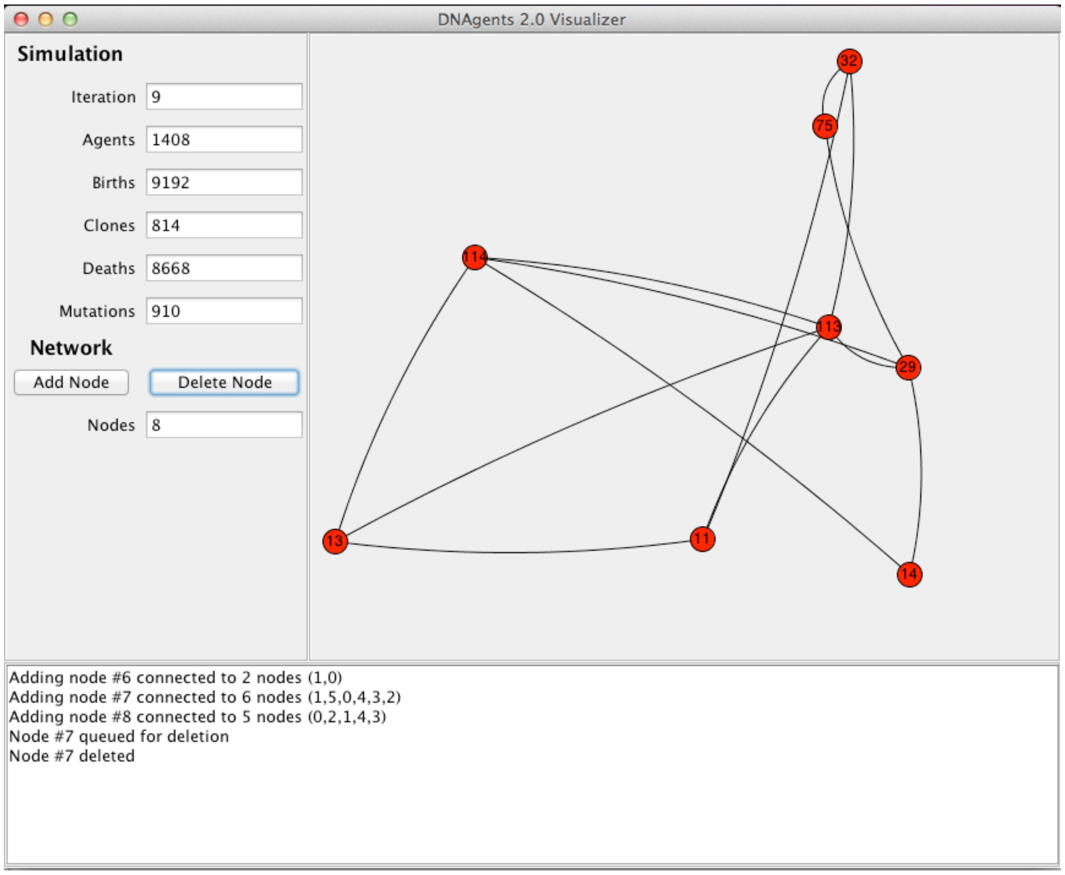
Mutations 827

Nodes 9

```
Adding node #6 connected to 2 nodes (1,0)
Adding node #7 connected to 6 nodes (1,5,0,4,3,2)
Adding node #8 connected to 5 nodes (0,2,1,4,3)
```

Figure 7: Two more nodes added to the network

To delete nodes, Cordar's node deletion is utilized. Since the nodes cannot be deleted mid-iteration, the nodes to be deleted are queued for deletion and deleted at the beginning of the next iteration. This can be seen in action in Figure 8.



Iteration 9
Agents 1408
Births 9192
Clones 814
Deaths 8668
Mutations 910
Nodes 8

```
Adding node #6 connected to 2 nodes (1,0)
Adding node #7 connected to 6 nodes (1,5,0,4,3,2)
Adding node #8 connected to 5 nodes (0,2,1,4,3)
Node #7 queued for deletion
Node #7 deleted
```

Figure 8: A node being deleted from the network

6. Conclusion and Future Work

Mobile agents are a relatively new concept to computing research and have a place in many future works. With his combination of genetic algorithms and mobile agents [1], Kackley has opened up a new area for research in the mobile agent field. Now that the agents can evolve over the course of their tasks, they can possibly adapt better to changing conditions of the network or even evolve new ways to solve their tasks.

In DNAgents 3.0, the simulation is now able to add new nodes with either random or specific links to other nodes. The agents are immediately able to seamlessly move to the new node by reaching it through one of these linked neighbors. This allows for the genetically engineered agents to operate on a dynamic network topology.

Since this research is so new, there are many potential ways to expand on this work. One problem that needs looking into is overpopulation. Since the agents are cloned in order to overcome extinction, they eventually breed without any control over the numbers. In [1], Kackley states that different selection mechanisms could be the solution. If an agent is given set amount of energy and runs until it is depleted, over population will not be as much of a problem. Other research could be in the area of optimizing the current framework by experimenting with different selection, crossover, and mutation algorithms.

The motivation of this work is to ultimately create a framework for autonomous agents to test a network for vulnerabilities and solve any problems.

To take DNAgents 3.0 to the next step, the current experiment to evolve hopping behavior should be replaced with new functionality to detect vulnerability within nodes of the network. This will lead to future functionality in which solutions to these vulnerabilities can be evolved by the agents as they traverse the network and help realize the end goal of this work.

REFERENCES

[1] J. Kackley, “Genetically Engineered Intelligent Mobile Agents,” Ph.D. dissertation, School of Computing, University of Southern Mississippi, Hattiesburg, MS, 2010.

[2] N. Minar, K. H. Kramer, and P. Maes. Cooperating Mobile Agents for Dynamic Network Routing. In *Software Agents for Future Communications Systems,* Chapter 12. Springer-Verlag, 1999.

[3] K. Rothermell and M. Schwehn. *“Mobile Agents”. Encyclopedia for Computer Science and Technology*, New York: M. Dekker Inc., 1998.

[4] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole - Concepts of a Mobile Agent System. *World Wide Web*, 1(3):123-137, 1998.

[5] R. S. Gray, D. Kotz, R. A. Peterson, J. Barton, D. Chacon, P. Gerken, M. Hofmann, J. Bradshaw, M. Breedy, R. Jeffers, N. Suri. Mobile-Agent Versus Client / Server Performance: Scalability in an Information-Retreval Task. Technical Report, Dartmouth College Hanover, NH, USA, 2001.

[6] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer science,* 1222:25-45, 1997.

[7] S. Fischmeister. Mobile code paradigms, Lecture Notes, 2002. Available: http://www.embeddedcmmi.at/fileadmin/src/docs/teaching/WS02/VS/mobile_agents.pdf

[8] M. Mitchell and S. Forrest. Genetic Algorithms and Artificial Life. *Artificial Life,* 1(3), 267-289, 1994.

[9] T. Blickle and L. Thiele. A Comparison of Selection Schemes used in Genetic Algorithms. Technical Report, Computer and Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1995.

[10] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Pralleles, Reseaux Et Systems Repartis,* 10, 1998.

[11] H. Khazaei, J. Misic, and V. Misic. “Mobile Software Agents for Wireless Network Mapping and Dynamic Routing,” presented at the ICDCSW ’10 Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops. Washington, DC, USA, 2010.

[12] F. Daniel, (2012, Feb. 15). Genetics and nature provide models to prevent cyber attacks. Winston-Salem Journal [Online]. Available: http://www2.journalnow.com/business/2012/feb/15/4/wake-forest-university-researchers-working-on-way--ar-1933784/

[13] M. Crouse and E. Fulp. A Moving Target Environment for Computer Configurations Using Genetic Algorithms. *Proceedings of the 4th Symposium on Configuration Analytics and Automation,* 1-7, 2011.

[14] A. Cordar. private communication and source code. University of Southern Mississippi, 2010-2011.