IMMUNE NETWORK OPTIMIZATION OF COMPOSITE SAAS FOR CLOUD COMPUTING

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Kevin Kaatz Bauer

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

November 2015

Fargo, North Dakota

North Dakota State University
Graduate School

Title

IMMUNE NETWORK OPTIMIZATION OF COMPOSITE SAAS

FOR CLOUD COMPUTING

By

Kevin Kaatz Bauer

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig
Advisor (typed)

Dr. Changhui Yan

Dr. Yechun Wang

Approved:

November 2, 2015
Date

Dr. Kenneth Magel
Department Chair

ABSTRACT

Serving the needs efficiently for a wide gamut of cloud users is a challenge. One way to address this challenge is to decompose SaaS (Software as a Service) into application components and then consider them as loosely coupled processes that achieve higher functionality. Optimization occurs in efficiently pairing virtual machines to application components in order to lower operating costs for cloud service providers and to lower subscription costs for customers. This thesis explores utilizing an immune network algorithm that mimics antibody activation and antigen and antibody suppression for resource optimization. Experiments are conducted with a series of SaaS configurations, application components placed with virtual machines. Results generated by the proposed algorithm are compared with a previously proposed grouping genetic algorithm. This data reveals that the immune network algorithm outperforms the grouping genetic algorithm in time taken to calculate a resource distribution strategy.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Simone Ludwig for her support, her ideas, for her motivating me, and for her kindness.

DEDICATION

This disquisition is dedicated to my parents, Allen and Cheryl Bauer, my sister, Kara Bauer, my best friend, Mike Zimmer, and to my wife, Maiko.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1. INTRODUCTION

Cloud computing is the provisioning of computer processing, networks, data, and applications to a consumer over the Internet [1]. This is becoming the preferred way for businesses to provision resources so they may outsource part of their workload to reduce costs in labor and in maintaining hardware [2] [3]. These resources are delivered on demand and are scalable in a pay-as-you-go approach.

Cloud computing can also be comprised of a large quantity of physical machines with heterogeneous computing resources distributed around different geographic locations [3]. The consumer is unaware of the infrastructure or platform details of the system, but expects that the service is always operational and that the service performs as if it always has the optimal amount of resources available for it on-demand.

The scalability of cloud computing must address the scaling up or the scaling down of resources. In addition, the balance between proactively scaling resources and reactively scaling resources has to be achieved [3]. For example, when a consumer no longer requires a service, the resources are quickly relinquished. Or, when a consumer requires a service, resources are allocated quickly. These operations must be implemented in a manner that is non-disruptive.

Although cloud computing by definition is still evolving [4] [5], cloud computing usually falls into one or a hybrid of three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

IaaS is the provisioning of computer processing, networks, and storage to the consumer so that their operating systems and software applications may utilize them. An example of IaaS is Amazon's EC2 (Elastic Cloud Computing) [6]. However, consumers must still manage the services that run on IaaS. For example, operating systems must be updated.

Databases must be administered. And, if consumers are administering a web service traffic must be managed by them.

PaaS hides infrastructure details from the consumer that are visible in IaaS: database administration; load balancing; and server configuration. The PaaS provider manages infrastructures for customers. The consumer may then deploy on top of this service applications, libraries, and other tools [6]. A well-known example of PaaS is Google's App Engine.

SaaS, the service that is the focus of this paper, is applications that are available to a consumer via a client. Clients that connect to SaaS are usually browsers [6]. Examples of SaaS are Microsoft's Office 365 and Google Drive.

Optimally provisioning resources for cloud computing services is a difficult problem, because the physical machines that deploy the resources can be in different geographic locations yet together must sufficiently deliver them to the consumer. In addition, Virtual Machines (VMs) that run atop these physical machines appear to be uniform, but usually they are not [7]. The machines that comprise the infrastructure are heterogeneous with storage disks that read/write at different rates or the VMs are sharing physical machine resources with other VMs servicing other consumers. Load balancing is also a difficult problem in cloud computing. Migrating VMs to different physical machines without negatively affecting Quality of Service (QoS) is problematic [8].

Applications that demand resources should be provisioned enough to fulfill the Service Licensing Agreement (SLA), yet no more then is required. Then, service providers can distribute resources economically. There are many methods for determining the best placement for an application among a selection of VMs.

Summarizing a short list of these existing methods, which are later described in greater detail in Related Work, propositions such as genetic algorithm implementations, resource auction systems, ant colony algorithm implementations, service deployment architecture systems, and scheduling tasks in the cloud for timely resource distribution.

Nature inspired algorithms feature prominently in the short list of solutions for complex optimization problems, such as genetic algorithms and ant colony algorithms, and for other problems that exist in engineering and computer science [9]. They are used because nature provides abundant examples of solving complex problems efficiently and effectively.

In this thesis, a nature inspired algorithm is investigated to optimize resource distribution for composite SaaS in a cloud computing environment. An Immune Network algorithm is applied to this task. It is inspired by natural immune systems using the idea of activation, suppression, and memory when antibodies interact with antigens.

The organization of this thesis is as follows: Related Work describes other solutions for resource optimization applied to cloud computing; Problem Formulation; Proposed Approach introduces and describes this thesis’s proposed approach as well as a comparison algorithm; Experiments and Results; Conclusions and Future Work.

2. RELATED WORK

Related work in the area of optimization in a Cloud environment includes the following. One approach uses the genetic algorithm for distributing resources by focusing on obtaining near optimal quality of service from infrastructures rather than finding optimal placement for VMs [10]. SaaSs are modeled as a cell. The cell is composed of application requirements: processing, memory, storage, network latency, and read/write times. These constraints must be fulfilled. VMs that satisfy these requirements are randomly paired with the application components. After the pairings, the roulette wheel selection of the genetic algorithm, the probabilistic application of the crossover operation and the mutation operation are applied. Once these operations finish, if new cells were generated they are added to the population. Then, each cell's fitness is determined. The fitness calculation is the distance from constraints to constraint satisfaction, so the smaller the difference the better the fitness. Thus, the fitness function is to be minimized. The cell with the best fitness is employed and should have near optimal quality of service for its application components.

Another approach is based on an auction system. Markets are considered efficient distributers of resources. Therefore, modeled after market resource allocation, VMs are distributed via an auction system. Traditionally, cloud providers bundle together homogenous VMs that remain static, and then they sell these to the highest bidder. However, this is not always the most efficient use of the infrastructure [11]. This is because customer requirements and the VMs that they require are not homogeneous. Consequently, a method that takes into account the heterogeneous requirements of customers was proposed [12].

For example, for each customer the cloud provider builds a number of VMs from a set of combinations of attributes. Customers bid on these VMs and the top bidder wins.

Yet, this method is susceptible to shill bidding. Shill bidding occurs when a customer impersonates multiple customers to lower the price of VMs. In addition, customers may conspire to bid at lower prices to bring down the overall cost. These problems lead to low profits for the service provider. To make VM auction systems shill bidding proof and to maximize cloud provider revenue, a variation of the previously proposed method was proposed in [13], [14].

Another novel method for efficiently distributing resources in a cloud-computing environment is via the ant colony algorithm [12]. The cloud is modeled as an undirected graph. The vertexes are clusters of VMs, and the edges connect these resources in a cloud-computing environment. The clusters of VMs have boundary conditions on the load that they can accept for processing, memory, storage, and bandwidth. If these boundaries are broken k times out of n, k and n are set arbitrarily, that vertex or rather that cluster of nodes is identified as a hotspot, meaning it is overloaded. To find avail- able resources to offset the overloaded hotspot, the ant colony algorithm is employed to find a node in the undirected graph representing the cloud that can provision more resources.

For example, in the hotspot, VMs belonging to the cluster are searched for idle resources. Idle resources are the minimum amount of resources that are available from all VMs in that cluster. This constraint is enforced so other VMs are not overloaded. If none are found, then neighboring clusters are searched on a path from the hotspot. As neighboring clusters of VMs are searched for idle resources the following operations are performed: the pheromone density is calculated for each node visited on the path from the hotspot, the nodes are added to an avoidance list, and the pheromone evaporation rates are calculated for previously visited nodes. Eventually, a path to a node with idle resources obeying the constraints converges.

Else, there are no nodes that satisfy the constraints to alleviate overloaded hotspots in the cloud-computing environment. One benefit for using this method is that the search process for finding idle resources in node clusters may utilize parallel processing.

For large-scale cloud computing, service deployment architectures may aid in efficiently provisioning resources. In [15], the cloud-computing environment is engineered to be SLA aware. Content Delivery Networks (CDN), like Akamai, contain edge servers that are physically close to clients and deliver content to them. The problem with this architecture is that all content is considered equal and unexpected bottlenecks can occur at these edge servers. For example, data in cloud environments can be active or passive. Active and passive data have different read/write requirements. Thus, [15] suggests providing different tiers of servers for different types of content and the resources that they require. First, the CDN is a tree of service resource providers. The leaf nodes are the edge servers that deliver content to customers.

Next, each Named Node Server (NNS) and its corresponding Block Servers (BS), together they act as the edge servers, have multiple Resource Monitors (RM) and Resource Allocators (RA), the inner nodes of the tree. The RMs monitor the rates of data moving uplink and downlink from the BSs. As needed, the delta of the rate of data transfer that is bottlenecking and overloading the constraints set for the tiered BS can be sent to parent RAs in the CDN, ameliorating the breach in the SLA. The MiniMax algorithm strictly regulates the rates of resource transfer sent from children to parent nodes and from parent to children nodes. These methods coupled with h tiers of service in the BS servers also mitigate overloading the BSs, because each tier can be allotted its required resources that are usually expected for the type of content that they are holding.

The Hadoop Yarn system is commonly used for administering resources for scheduled tasks [16]. Resource managers assign application managers to worker nodes that monitor how many resources the worker nodes are using. The application managers also negotiate with the resource managers for resources to complete their scheduled tasks. Then, the resource managers will deliver the requirements according to the set policies. For example, one simple policy is queue based (FIFO). The tasks are scheduled for resources and Worker Nodes in the order of their arrival. Fair policies evaluate all the tasks and the resource managers attempt to distribute equal amounts of resources to all of the tasks in the job queue or the average equal share of the dominant resource requirements. The last common policy is the capacity policy. For each task the resource manager attempts to schedule an equal share of the resources for each task in the queue. The leftover resources that are still free are given to tasks that are using more than what was scheduled for them. And, some tasks within the queue are prioritized and provisioned resources accordingly. However, according to [16] these policies are not optimal.

3. PROBLEM FORMULATION

Composite SaaSs are composed of application components, ACs, and virtual machines, VMs. The ACs are placed within VMs, and the VMs are placed within cloud architectures, such as computing servers and data storage servers. Resource requirements can vary over time, so scheduled reconfigurations of AC and VM pairings are required. However, every time a reconfiguration is triggered, the optimal placement of ACs with the appropriate VMs is desirable. VMs may derive their resources from heterogeneous hardware so it is not always optimal to place one AC with one VM in each composite SaaS. In many cases it can be optimal to place multiple ACs with one VM. Because, a particular VM might have more processing-, memory-, and storage resources that it is provisioned from its hardware than what other VMs are provisioned from their hardware. In Figure 1 ACs are in the top row and VMs are in the bottom row. In this example some VMs are used for more than one AC.

Application Component ID Numbers

13	12	4	3	8	7	5	10	9	1	6	2	15	14	11
3	5	2	2	5	1	2	5	1	1	3	3	1	2	5

Virtual Machine ID Numbers

Figure 1. An Arrangement of ACs and VMs.

We can ascribe a fitness metric to an assortment of SaaSs, a cell, in a cloud-computing environment. Ideally, ACs are always placed with VMs that have the least amount of processing, memory-, and storage- resources *used* to avoid overloading VMs. Thus, VMs are measured by the load they are bearing. In addition, since ACs are placed with VMs they may also need to be shifted to other VMs as deemed by some schedule.

ACs also require processing resources, data storage, and memory. A metric is applied to them, too. For example, the data storage transfer requires bandwidth. The larger the data store, the more bandwidth is required to do the transfer for optimal placement. And, the process of moving the application component itself has a memory requirement. Ideally, only ACs that are the least burdensome to move to other VMs are shifted so as not to overload the resources that are available in the cloud-computing environment. Based on these heuristics we may use this formula to ascribe a fitness value to an assortment of SaaSs, or what is known as a cell.

$$F = w_1 \times F(TC) + w_2 \times F(MC) \tag{1}$$

w_1 is the weight applied to VMs, and w_2 is the weight applied to ACs. *TC* is the total cost of the SaaSs' VMs in a cell. This is the formula used to normalize the fitness of *TC*:

$$F(TC) = \begin{cases} 0, \text{ if } TC > C_{init}, \\ \frac{C_{init} - TC}{C_{init}}, \quad \text{otherwise}, \end{cases} \tag{2}$$

C_{init} is the initial cost of the VMs before the cost is recalculated for subsequent reconfigurations. *VM* is the set of all VMs used by the SaaSs in a cell and *TC* is calculated with the following formula:

$$TC = \sum_{vm \in VM} C_{vm} \tag{3}$$

A single VM's cost is the sum of its processing-, memory-, and storage- resources used divided by the sum of the VM's processing-, memory-, and storage- resources provisioned, as demonstrated in the following formula.

$$\frac{(Processor\ Used + Memory\ Used + Storage\ Used)}{(Processor\ Available + Memory\ Available + Storage\ Available)} \tag{4}$$

MC is the migration cost of ACs. It is normalized with the following formula:

$$F(MC) = 1 - \frac{MC}{N(AC)} \tag{5}$$

N(AC) is the number of ACs that are in the cell. The formula for determining the *MC* is as follows:

$$MC = \sum_{ac \in AC} \frac{S_{ac}}{max(S_{AC}) \times 2} + \frac{M_{ac}}{max(M_{AC}) \times 2} \tag{6}$$

S_{ac} is the storage requirement of the AC, its size, and M_{ac} is the memory requirement of the AC. S_{AC} and M_{AC} both refer to the set of all ACs in a SaaS.

To demonstrate example calculations using Equations 1 – 6, Table 1 is an example cell that will be ascribed a fitness.

Table 1. Example Cell Arrangement of AC and VM Pairings.

SaaS 1			SaaS 2		
AC	1	2	AC	3	4
VM	2	2	VM	3	4

For each AC and VM identified in Table 1, Table 2 shows processing-, memory-, and storage resources each VM provides.

Table 2. Example VM Resources Provisioned.

VM	1	2	3	4
Processor	2	4	6	8
Memory	3	5	7	9
Storage	4	8	12	16

Table 3 shows each AC's requirements.

Table 3. Example AC Resources Required.

AC	1	2	3	4
Processor	1	2	3	4
Memory	2	3	4	5
Storage	3	4	5	6

Based upon the AC and VM pairings displayed in Table 1, Table 4 shows how the resources provisioned are used.

Table 4. Example VM Resources Used.

VM	1	2	3	4
Processor Used	0	3	3	4
Memory Used	0	5	4	5
Storage Used	0	7	5	6

Using Table 2 for resources provisioned, Table 3 for what resources are required, and Table 4 for how the resources provisioned are used, it is now possible to calculate the cost for each VM using Equation 4. Arithmetic for VM 2 is shown in Equation 7.

$$Resources\ Used = \frac{(3+5+7)}{(4+5+8)} = 0.7083333 \tag{7}$$

Table 5. Example VM Cost Calculation.

VM	1	2	3	4
Cost	0	0.88235294	0.48	0.45454545

Now that the VM costs are calculated, Table 5, the cost per SaaS can be calculated, Figure 2 shows the VMs per SaaS, and subsequently, the Total Cost, *TC*.

Table 6. Example Total Cost Calculation.

SaaS	1	2	Total Cost
Cost	0.88235294	0.93454545	1.8168984

Using Table 5, it it easy to determine the final cost of a cell, which is the sum of the cost per SaaS in Table 6.

Next, to determine the migration cost, *MC*, in Equation 6, AC and VM pairings per SaaS in Figure 2 are used. Arithmetic for AC 1's cost is shown in Equation 8. In the denominators in Equation 8, the AC with the max storage resource for the SaaS, AC 2 is used, and the AC with the max memory for the SaaS, AC 2 is used again, which determines the values of 4 and 3.

$$MC = \frac{3}{4 \times 2} + \frac{2}{3 \times 2} = 0.7083333 \quad (8)$$

Table 7. Example AC Cost Calculations and Example Migration Cost Calculation.

AC Costs	1	2	3	4	Sum	Migration Cost
Storage	0.375	0.5	0.4166667	0.5	1.7916667	3.525
Memory	0.333333	0.5	0.4	0.5	1.7333333	

After both the Total Cost and Migration Cost values are calculated, they must both be normalized to calculate the fitness of the cell. First, the Total Cost from Table 6 is normalized with Equation 2. For the initial Total Cost value, an arbitrary Total Cost of 1.91345 is used.

$$TC = \frac{1.91345 - 1.8168984}{1.91345} = 0.05046 \quad (9)$$

Then, the Migration Cost must also be normalized with Equation 5.

$$MC = 1 - \frac{3.525}{4} = 0.11875 \tag{10}$$

Finally, with both the normalized Total Cost and Migration Cost, and weights of 0.5, Equation 1 is used to ascribe a fitness to the cell.

$$Fitness = 0.5 \times 0.05046 \times 0.5 \times 0.11875 = \ 0.00149803 \tag{11}$$

A fitness with a value of 0.00149803 is ascribed to this cell.

4. PROPOSED APPROACH

Artificial Immune Systems (AIS) [17] [18] are already used to solve problems in computer science and in engineering [19]. What makes them useful are their inherent features of learning-, recognition-, and memory of patterns. For example, AIS has been applied to security systems [20] [21], robotics [22] [23] [24] [25], and towards clustering and classification [26] [27] [28] [29] [30]. For the SaaS cloud resource optimization problem, the Immune Network algorithm is considered.

4.1. Immune Network Algorithm

Neils K. Jerne [17] proposed that antibodies in invertebrates not only regulated foreign antigens inside an immunological system, but also other antibodies that are a part of the body's "self", not only the body's "non-self". Regulatory reactions depend on responses received from the immunological system. When the immunological system reacts positively, it generates new antibodies. When it reacts negatively, it suppresses antibodies. Hoffman [18] states that antibodies are capable of recognizing an almost unlimited amount of substances. This indicates that B-lymphocytes and T-lymphocytes can recognize each other by their variable regions, allowing regulation of production and suppression of antibodies. This is known as Immune Network theory. The properties of Immune Networks, such as pattern recognition, memory, the construction and the destruction of specific biological configurations can be utilized for solving cloud computing resource allocation problems.

The immune network algorithm pseudo-code is provided in Algorithm 1.

Algorithm 1. Immune Network Algorithm.

```
1. Input: populationSize p, numberOfClones n, randomRatio r, affinityThreshold a,
2. mutationParameter m
3. Output: bestCell b
4. population ⇐ InitializePopulation(p)
5. while (stoppingCriterion not met) do
6.     EvaluatePopulation(population)
7.     b ⇐ GetBestCell(population)
8.     initialAverageFitness ⇐ CalculateAverageFitness(population)
9.     do
10.        for (cell ∈ population) do
11.            clones ⇐ GenerateClones(cell, n)
12.            for (clone ∈ clones) do
13.                clone ⇐ Mutate(clone, m)
14.            end for
15.            EvaluatePopulation(clones)
16.            descendent ⇐ GetBestCell(clones)
17.            if descendent is more fit than cell
18.                replace cell with descendent in population
19.            end if
20.        end for
21.    while (CalculateAverageFitness(population) > initialAverageFitness)
22.    SuppressLowAffinityCells(population, a)
23.    cells ⇐ CreateRandomCells(r)
24.    population ⇐ cells
25. end while
26. return b
```

The algorithm begins its main iterative process by randomly initializing a population of cells. Next, a secondary iterative process begins. First, each cell in the population is evaluated and ascribed a fitness. At this step the cell with the best fitness can be selected and recorded. Then, each cell in the population is cloned to create an additional cloned cell population. Each cloned cell is mutated and then evaluated. Original cells are replaced in the population by the mutated ones if the mutated cloned cells have a better fitness. These steps continue in the secondary iterative process until a stopping criterion is met.

After the stopping criterion is met, the immune network interactions described in immune network theory occur. During the network interactions, cells with the lowest affinity are removed from the population. At this point, an extra set of cells is randomly initialized and added to the population. This continues until a stopping criterion for the main iterative process is met. Once the main iterative process terminates, the fittest cell in the population is returned.

4.2. Immune Network Algorithm Applied to the SaaS Optimization Problem

The implemented algorithm based on Immune Network theory is applied to the SaaS optimization problem as follows: First, sets of cells are randomly initialized. Each cell has a configuration of VM and AC pairings from the available pool of VMs and ACs to satisfy the requirements of existing SaaSs. This set will be referred to as the population. Next, the affinity, the fitness, is calculated. The fitness per cell is calculated with Equations 1 – 6. Then, for each cell in the population an arbitrary number of clones per cell are generated. Each cloned cell is passed to a mutation function that randomly alters the cell. This process is explained in the next section and shown in Figure 10. After the mutation operation, if any cell among the clones has a better fitness as a result of the mutation operation than the cell it was cloned from originally is replaced by the mutated cell in the population.

After the clonal selection operations, the network regulatory operations, or as the algorithm refers to as *network suppression*, occurs. For each cell in the population, if the cell's fitness compared with the population's average fitness is worse than the average by a predetermined threshold, then the cell is removed from the population. If the cell is better than the average by at least the threshold factor it is allowed to remain in the population.

After the network suppression operations, an arbitrary number of randomly generated cells are added to the population.

All of these operations repeat in an iterative fashion until the stopping criterion is met. Once the stopping criterion is met, the fittest cell of the population is returned. Please refer to the pseudo-code in Algorithm 1.

4.3. Immune Network Algorithm Implementation

In the SaaS composition problem, randomly configured SaaSs are grouped together to model a cell. The SaaSs are composed of many application components (ACs), each with their own processing-, memory-, storage-, and read/write requirements. The sum of these requirements is the SLA for the SaaS. VMs rest atop distributed physical machines that may reside in different physical locations. These VMs can be paired with any of the application components. VMs may also be the sole resource provider for an application component or the resource provider for many application components. Arbitrary amounts of cells are created with the same SaaSs, ACs, and VMs, each with different pairings and placements.

A fitness evaluation is ascribed to each cell. The lower the fitness the better, so the fitness function minimizes. Costs for VMs are determined by the amount of resources that are being used. The more resources that are used, the more the fitness degrades since then a VM has a larger workload. Therefore, it is better for an AC to be paired with a VM that has a lower workload. Thus, it is more likely the SLA for the AC is adhered to. The fitness is also determined by the storage and memory requirements of ACs. This is because ACs with larger memory and storage requirements are costlier to move to different VMs over a network.

Therefore, the larger the memory and storage requirements are, the more fitness degrades in the cell.

The IN implementation closely follows the algorithm description in Algorithm 1. First populations of cells, assortments of composite SaaSs, are randomly initialized. Refer to Figure 1. After initialization, Step 4, each cell is first inspected to determine if any ACs' resources are not being met, Step 6. Resources not being met imply ACs do not have enough processing-, or enough memory-, or enough storage resources provisioned. If so, then the SLA for the composite SaaS is not being met. The implementation attempts to find a VM with a minimized load of ACs for the purposes of avoiding overloading any one VM. And, the VM must not be a VM the AC is already paired with to avoid duplicate pairings. If there are no more VMs available to satisfy SLAs, then the cloud service provider must provide more VMs for the cloud-computing environment.

After the repair process, for each cell in the population, a fitness metric is ascribed to the cell using Equations 1 through 6. An average fitness of the entire population is also calculated, in Step 8.

Next, each cell in the population is cloned an arbitrary number of times, in Steps 10 and 11, which is set by a variable in Step 1. The cloned cells undergo a mutation operation, Step 13, to add diversity to the search space for finding the fittest cell. In this implementation of the Immune Network algorithm, mutation is the process of randomly selecting an AC in a SaaS and removing its VM. A random search is performed to find any VM in the pool of VMs that satisfies the ACs' requirements. If there is one, it is paired with the AC. If there is no other VM that satisfies the ACs' requirements then the mutation operation is skipped, because the SLA takes precedence over the mutation operation.

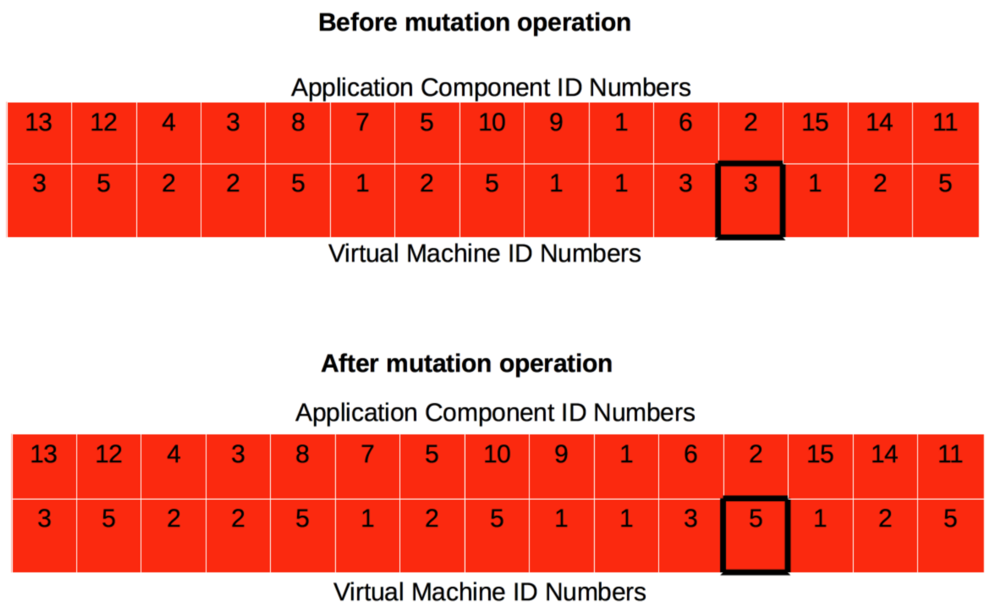


Figure 2. Mutation Operation for IN and GGA.

The new VM may or may not improve the fitness of the cell. This mutation operation is performed an arbitrary number times set by the mutation parameter in Step 2 of Algorithm 1.

After the cloning- and mutation operations complete, the clone is checked to determine that all of its ACs have enough resources to satisfy the SaaS SLAs, Step 15, and if not it is repaired. Next, the clone is evaluated using the fitness equations 1 through 6. From the set of clones per cell, if the clone with the best fitness of the set of clones has a better fitness than the original cell it was cloned from, then the clone replaces the original cell in the population, Steps 16 through 18.

In Step 9 of Algorithm 1, if the predefined average fitness remains greater than the average fitness after Step 10 through Step 17 then the while loop is repeated until a stopping criterion is met.

Determining a cell's affinity with the population of cells, for network suppression, Step 22, is calculated as follows. First, the mean average fitness of the cell population is calculated.

The absolute value of the best fitness, in the cell population, minus the average fitness multiplied by an arbitrary support threshold, set in Step 1, is the affinity threshold.

$$\text{AffinityThreshold} = \text{SupportThreshold} * (|\text{BestFitness} - \text{AverageFitness}|) \qquad (12)$$

For each cell in the population, if its fitness is greater than the best fitness plus the affinity threshold, the cell is removed from the population.

Algorithm 2. Suppress Low Affinity Cells

```
1.  if cellFitness ≥ (AffinityThreshold + BestFitness)
2.      Remove cell from population
```

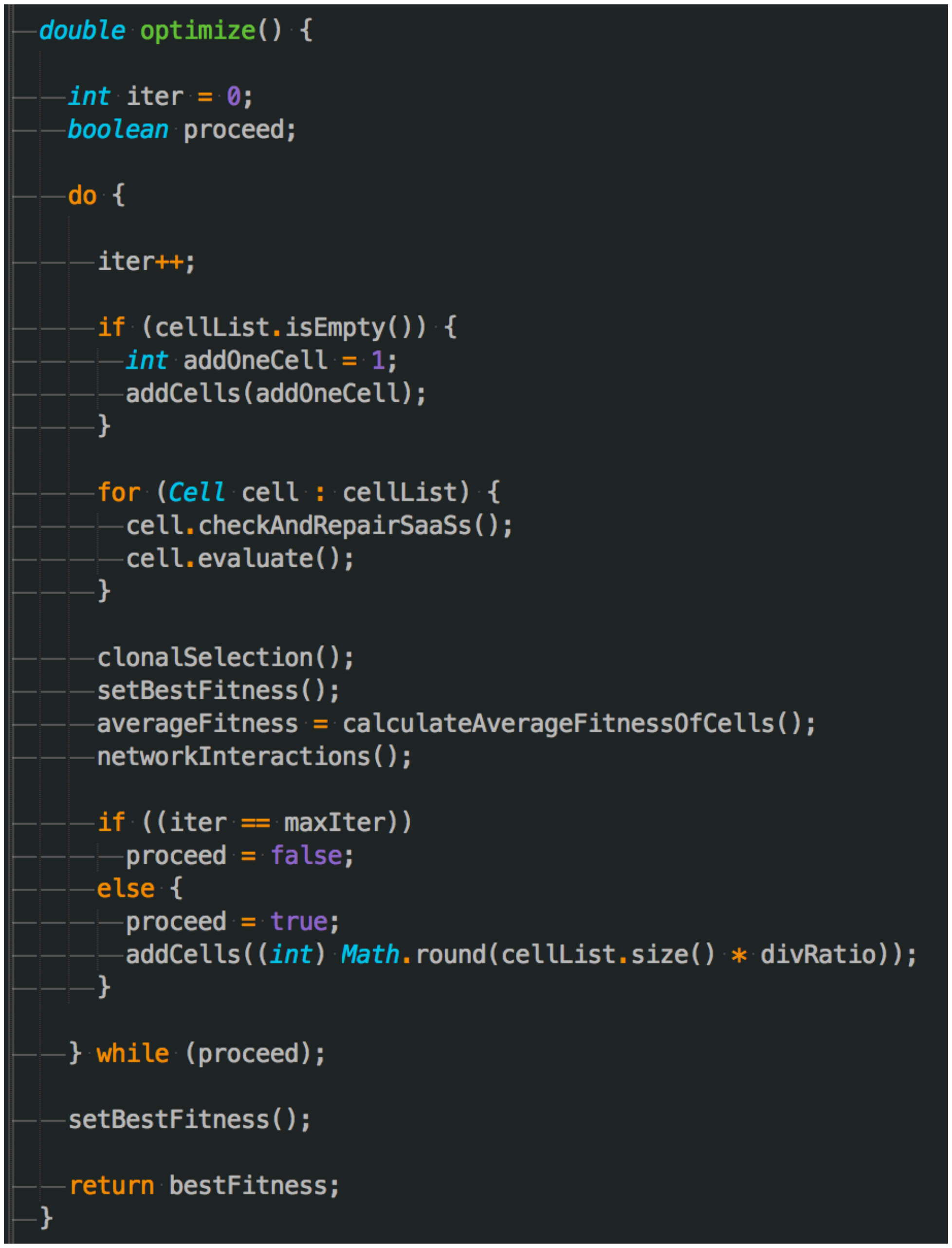
The count of cells to randomly generate is determined with the following formula:

$$\text{NumberOfCellsToAdd} = \text{Round}(\ \text{Count}(\text{cellPopulation}) * \alpha):\ 0 < \alpha \leq 1 \qquad (13)$$

Alpha is set in Step 1.

4.4. Immune Network Algorithm Implementation: Java Code Snippets

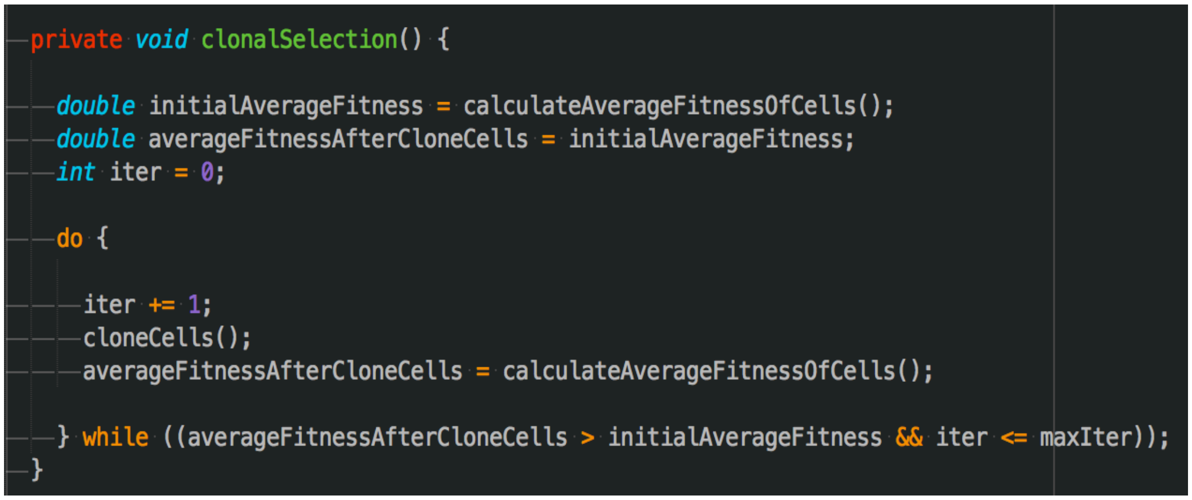
In this section snippets of Java code implementing primary processes of Algorithm 1 are presented. Figure 3 implements the main iterative loop, steps 5 to 26 in Algorithm 1. Sub processes are abstracted into separate methods for software readability. Cell cloning, described in 4.1 and in 4.3 occurs within method clonalSelection(). And, cell suppression occurs within method networkInteractions(). Cell fitness calculations described in Equations 1 – 11 are executed in the cell object's method evaluate: cell.evaluate().



```java
double optimize() {

    int iter = 0;
    boolean proceed;

    do {

        iter++;

        if (cellList.isEmpty()) {
            int addOneCell = 1;
            addCells(addOneCell);
        }

        for (Cell cell : cellList) {
            cell.checkAndRepairSaaSs();
            cell.evaluate();
        }

        clonalSelection();
        setBestFitness();
        averageFitness = calculateAverageFitnessOfCells();
        networkInteractions();

        if ((iter == maxIter))
            proceed = false;
        else {
            proceed = true;
            addCells((int) Math.round(cellList.size() * divRatio));
        }

    } while (proceed);

    setBestFitness();

    return bestFitness;
}
```

Figure 3. Immune Network Main Loop.

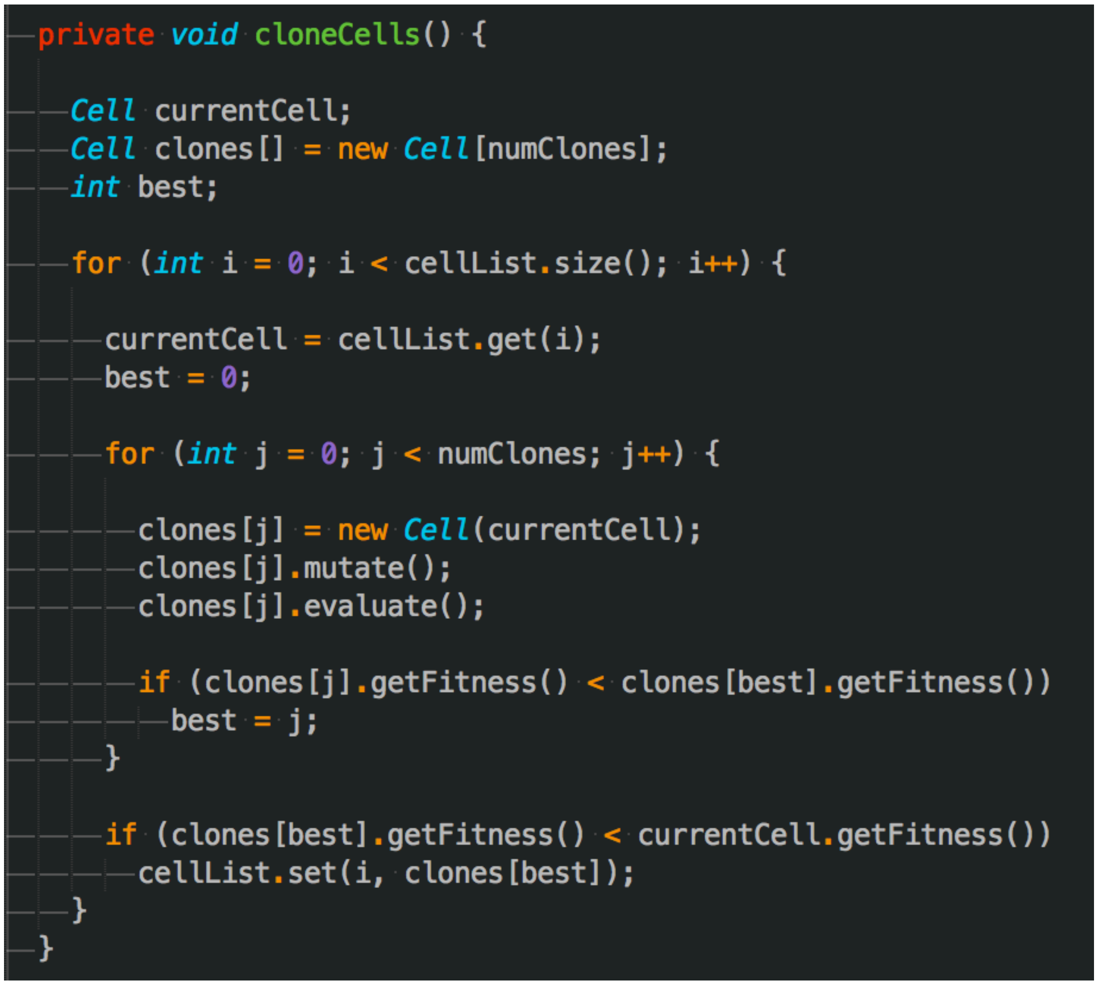
Figure 4 and Figure 5 cover steps 8 to 21 of Algorithm 1. Figure 4 controls the inner iterative process in Algorithm 1, which sets and controls the stopping criterion on Step 9 of Algorithm 1.



```java
private void clonalSelection() {

    double initialAverageFitness = calculateAverageFitnessOfCells();
    double averageFitnessAfterCloneCells = initialAverageFitness;
    int iter = 0;

    do {

        iter += 1;
        cloneCells();
        averageFitnessAfterCloneCells = calculateAverageFitnessOfCells();

    } while ((averageFitnessAfterCloneCells > initialAverageFitness && iter <= maxIter));
}
```

Figure 4. Immune Network Inner Loop.

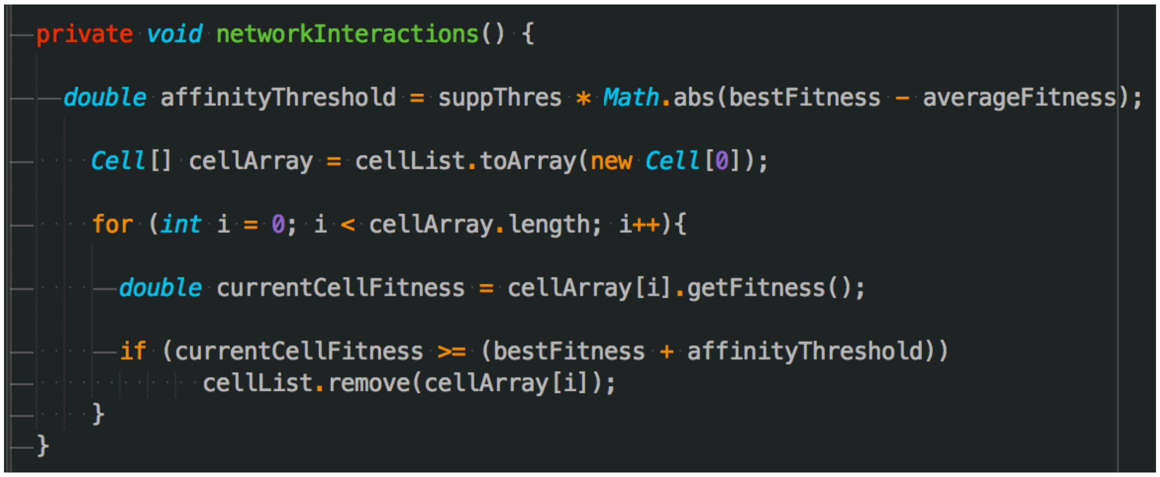
Steps 10 to 20 are abstracted into the method cloneCells(), which is represented in Figure 5. Also in Figure 5, the mutate operation executed on cloned cells is shown in Figure 9. To reiterate the cloning and mutation processes described in prior sections, Figure 5 shows when each cell in the population is cloned, mutated, and then evaluated. Then, out of the population of cloned cells, the cloned cell with the best fitness is selected. If that selected cloned cell has a better fitness than the cell it was cloned from, it replaces the original cell in the population. The cloneCells() method terminates when all the cells in the population have been cloned and evaluated for possible replacement. Cloning of cell objects in the population was done via a copy constructor.



```
private void cloneCells() {

    Cell currentCell;
    Cell clones[] = new Cell[numClones];
    int best;

    for (int i = 0; i < cellList.size(); i++) {

        currentCell = cellList.get(i);
        best = 0;

        for (int j = 0; j < numClones; j++) {

            clones[j] = new Cell(currentCell);
            clones[j].mutate();
            clones[j].evaluate();

            if (clones[j].getFitness() < clones[best].getFitness())
                best = j;
        }

        if (clones[best].getFitness() < currentCell.getFitness())
            cellList.set(i, clones[best]);
    }
}
```

Figure 5. Clone Cells.

Finally, Figure 6 covers step 22 of Algorithm 1 and it covers Algorithm 2. Calculating the double value for variable affinityThreshold closely resembles Equation 12. And, cell suppression resembles the pseudo-code in Algorithm 2.



```java
private void networkInteractions() {
    double affinityThreshold = suppThres * Math.abs(bestFitness - averageFitness);
    Cell[] cellArray = cellList.toArray(new Cell[0]);
    for (int i = 0; i < cellArray.length; i++){
        double currentCellFitness = cellArray[i].getFitness();
        if (currentCellFitness >= (bestFitness + affinityThreshold))
                cellList.remove(cellArray[i]);
    }
}
```

Figure 6. Network Interactions.

For constraints set by Java SE 6, the cell population is copied to an array so that cells can be removed from the population as the population is iterated through. Lists in Java SE 6 are immutable during iteration.

5. EXPERIMENTS AND RESULTS

This section describes how the Immune Network implementation was prepped for its experimental runs: programming language used, parameters set for the experimental runs, SaaS configurations, and the experimental runs' server environment. After listing and explaining these items, the results from the experimental runs are explained. For comparison purposes, results from an existing solution, a Grouping Genetic Algorithm implementation, are also explained. For a proper understanding of the compared results, the Grouping Genetic Algorithm must also be understood.

5.1. Comparison Algorithm: Grouping Genetic Algorithm

To determine how the Immune Network (IN) algorithm fares it was compared with an implemented version of the Genetic Grouping Algorithm (GGA) [1].

Algorithm 3. Grouping Genetic Algorithm.

```
1. bestFitness = 0
2. initialize(Population)
3. while (stoppingCriterion not met) do
4. foreach cell in Population do
5.        if cell violates SaaS resource requirements then
6.              Repair(cell)
7.        end if
8.        Calculate new VMs' costs
9.        Calculate new ACs' costs
10.       Calculate cell fitness
11.       if fitness(cell) > bestFitness then
12.             Replace bestFitness
13.       end if
14.       Select individuals from the Population based on roulette wheel selection
15.       Probabilistically apply the crossover operator to generate new individual
16.       Probabilistically select individuals for mutation
17.       Use the new individuals to replace the old individuals in the Population
18. end
```

The GGA is similar to a standard Genetic Algorithm (GA). What is different between GA and GGA is that the GGA algorithm evaluates a chromosome’s fitness based on a composite set of elements. In this case the VMs and ACs inner components comprise the overall quality of the chromosome and its ability to fulfill service license agreements (SLAs). For example, VMs in a chromosome might contribute to a good fitness for the chromosome. But, enough ACs could weigh the chromosome’s fitness towards less quality. The overall composite chromosome must be measured. Thus the algorithm derives its name *Grouping* Genetic Algorithm.

And, GGA adds an additional operation to the standard steps in the GA. In addition to initialization, selection, crossover and mutate operations, the GGA iterates through the population and determines if all the chromosomes are fulfilling their SLAs. If any of them are not adhered to, then the cells are repaired by pairing additional VMs with ACs that are bearing the least volume. It is after this operation that the cells are evaluated and the selection, crossover and mutate operations are performed.

To quickly summarize how the crossover operation was implemented for this thesis’s research, and to show the crossover operation visually, Figure 7 was taken directly from [1].

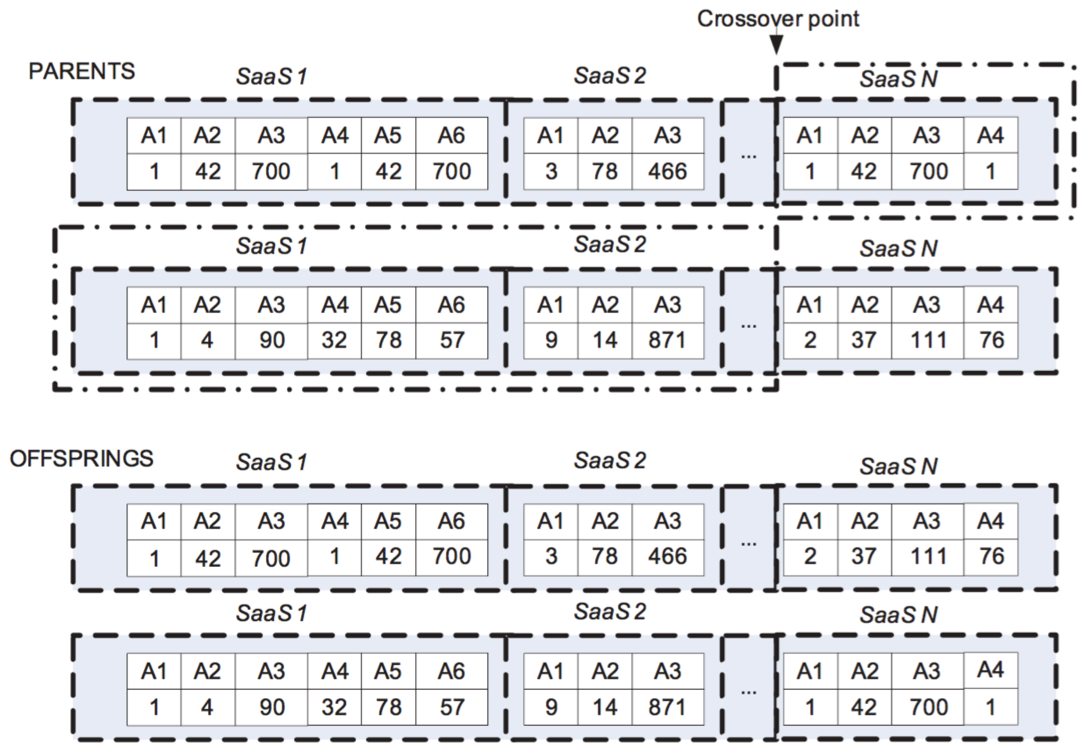


Figure 7. Crossover Operation [1].

For the parent cells, two cells are randomly selected at Step 14 of Algorithm 3. During Step 15, each parent cell is cloned to generate the offspring cells. Then, a SaaS is randomly selected per offspring cell. Next, their AC and VM pairing configurations are swapped, but if and only if the swapping does not break the cells' SLAs. The offspring cells are returned to the population if the SLAs are adhered to.

5.2. Experimental Setup

Table 8 shows the configuration used for the experiments.

Table 8. A Configuration of AC and VM Specifications.

AC			VM		
CPU	**Memory**	**Storage**	**CPU**	**Memory**	**Storage**
6-12 (incr. of 2)	12-32 (incr. of 2)	200-1000 (incr. of 50)	12-6 (decr. of 2)	32-12 (decr. of 2)	1000-200 (decr. of 50)

For this setup, a configuration of resources varies for both the ACs and for the VMs. In Table 8 the ranges for resources are iterated through and repeated in a VM and AC initialization method. The iteration control loops' sentinels are set by the count of VMs and ACs set for the experiments in declared parameters, which are explained shortly. The characteristics of the AC and VM instances in terms of CPU (number of cores; 2.6-2.8GHz), memory (measured in GiB), and storage (measured in GB) were obtained from the Amazon website [31].

Parameters set for the implementation of the IN algorithm were set as follows:

- Number of chromosomes = 100
- Number of clones per iteration = 30
- Number of ACs = 50
- Number of VMs = 50
- Number of SaaSs = 10
- Number of iterations = 50
- Mutation Parameter = 10

- Affinity threshold = alpha * abs(bestFitness-averageFitness), where alpha was set in ranges 0.1 – 0.9 in increments of 0.1.

Parameters set for the comparison GGA algorithm were set as follows:

- Number of chromosomes = 100
- Number of ACs = 50
- Number of VMs = 50
- Number of SaaSs = 10
- Number of iterations = 50
- Crossover probability = 0.7
- Mutation probability = 0.1
- Mutation Parameter = 10

Both the IN and GGA implementations were run 30 times each for every configuration in Table 8. And, for each alpha used to determine the Affinity Threshold in IN, 0.1 – 0.9 in increments of 0.1, each configuration in Table 1 was run with each alpha increment 30 times. From all of these configurations averages of the results are reported.

The algorithms were implemented with Java SE 6. And, they were run with the OpenJDK Runtime Environment (IcedTea6) with the OpenJDK 64-Bit Server VM. OpenJDK was installed on a virtual machine running Debian version 3.2.0-4-amd64. There was one 64-bit CPU with 2,793 MHz. And, the virtual machine running the Debian server was provisioned 15 GB of memory.

5.3. Results

Results are presented and explained based on the configuration in Table 8. First, in the conducted experiment the threshold factor is observed. A threshold factor is used to determine the affinity threshold described in Equation 12. In Table 9 it is discernable that the fitness stops improving at a threshold factor of 0.6. This is so if the associated standard deviation is considered with the fitness values for higher threshold factors. Thus, subsequent results utilizing a threshold factor of 0.6 in this thesis are presented for the IN implementation.

Table 9. Threshold Factor and Achieved Fitness.

Threshold Factor	Fitness and StdDev
0.1	0.0179150276864746±0.010686925
0.2	0.0180501679338058±0.009170124
0.3	0.009715845024539±0.009316006
0.4	0.0102096112172823±0.008126578
0.5	0.00732991875659623±0.007332595
0.6	0.00551042911392754±0.007320064
0.7	0.00183202761733134±0.004094568
0.8	0.00192065387041332±0.004157237
0.9	0.00267278292279459±0.004328752

The second part of the experiment demonstrates how the IN algorithm compares with the GGA algorithm regarding fitness calculation.

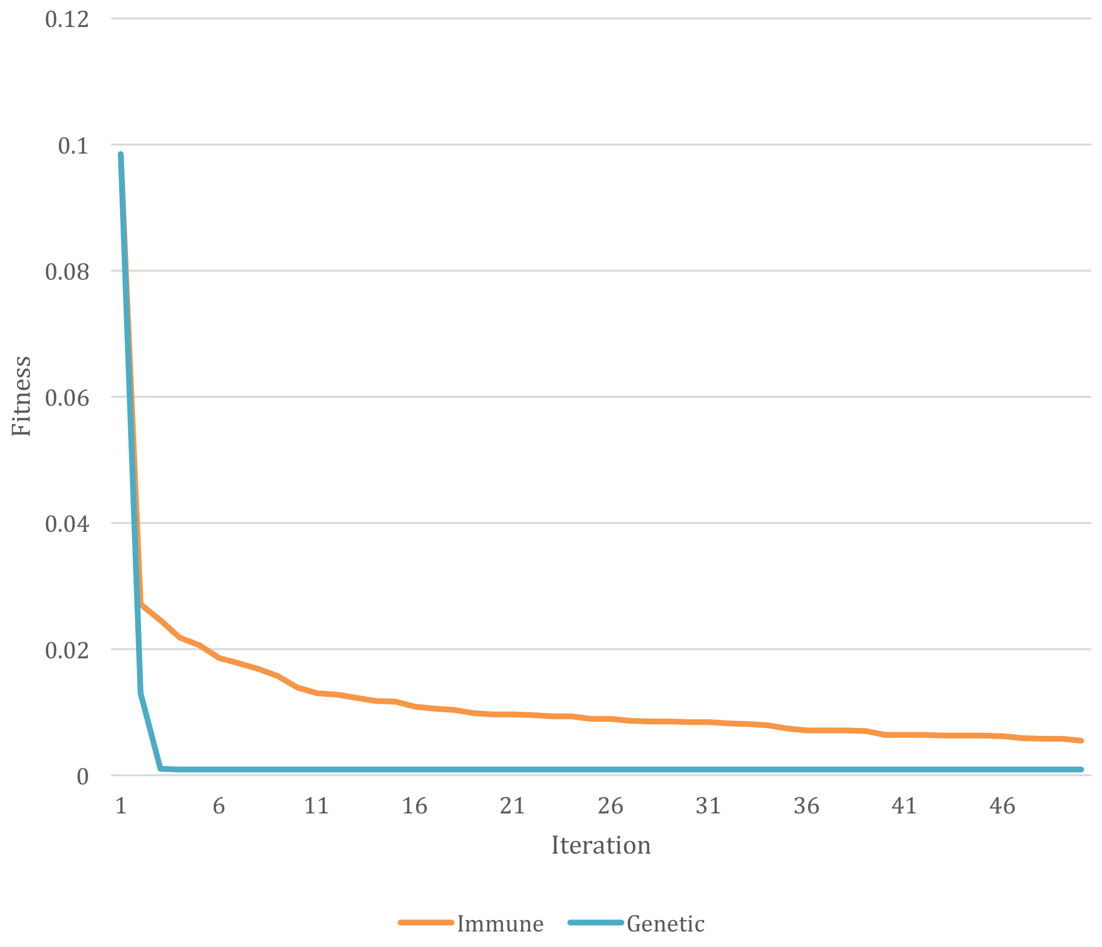


Figure 8. IN and GGA Comparison.

It is clear in Figure 8 that the GGA implementation converges on a fitness in less iterations than the IN implementation. It is also clear in Figure 8 and in Figure 9 that the GGA implementation converges on a more optimal fitness than the IN implementation. The difference between the average achieved fitness obtained from the IN and GGA implementations, after convergence, is 0.004558851.

On average, the fitness recorded during the first iteration for both implementations, before convergence, is 0.098425178.

Comparing the GGA implementation's fitness during initialization with the recorded fitness when the implementation terminates and has converged, there is a 99% improvement. Comparing the IN implementation's fitness when it initiates with the recorded fitness when it terminates and has converged, there is a 94% improvement.

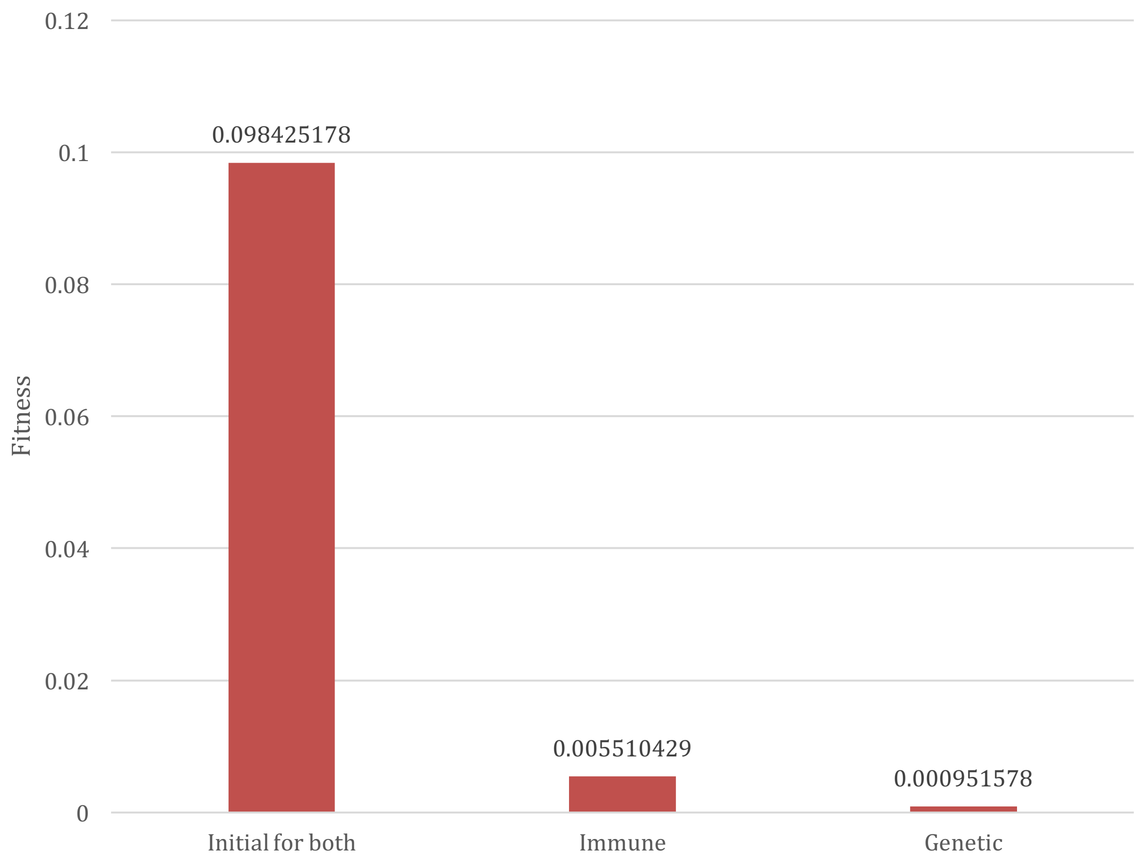


Figure 9. Fitness Optimization.

In Figure 10 the difference, between the IN and GGA, in achieved fitness per iteration is observed.

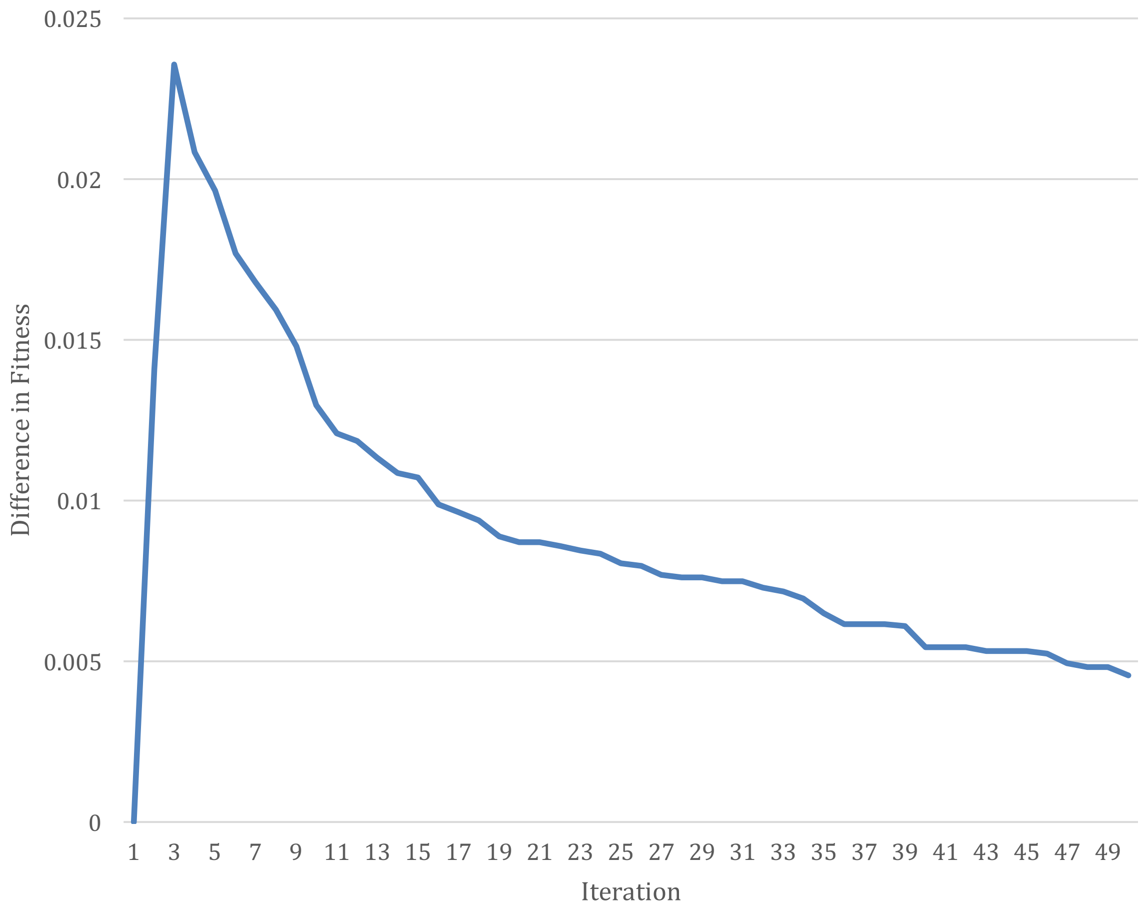


Figure 10. Difference in Achieved Fitness.

It is clear that the GGA implementation will achieve a better fitness on average, yet, at times, it is possible for the IN implementation to converge on a better fitness than the GGA implementation. Figure 11 demonstrates a single run from both the IN implementation and the GGA implementation.

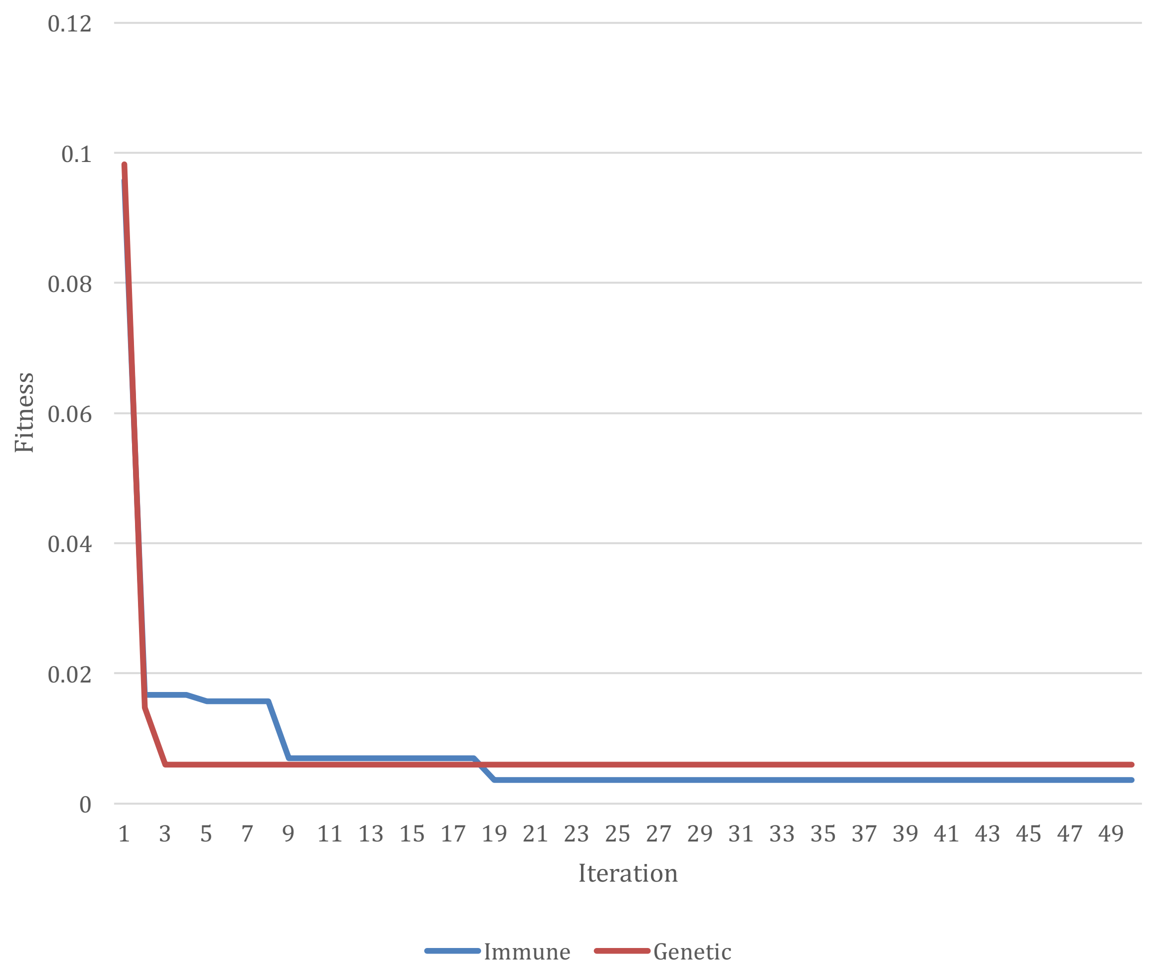


Figure 11. A Single Run Comparison.

However, depending on SaaS SLA requirements the GGA's average achieved fitness may not be enough of an improvement over the IN's average achieved fitness to merit using the GGA instead of the IN. Therefore, the time taken for the GGA and IN implementations is examined next.

By first appearances, when comparing the pseudo code of the IN algorithm to the GGA algorithm it appears as though the IN algorithm will take more time to calculate a fitness than the GGA's runtime. It appears so because with the GGA algorithm a fitness evaluation for each cell occurs only once per iteration.

Examining the IN algorithm, a fitness evaluation occurs once for each cell in the population per iteration on line 6 of Algorithm 1. However, additional fitness evaluations occur in the clonal selection operations of the IN algorithm. In Algorithm 1 starting at line 15, each cell is cloned an arbitrary amount of times, and then each clone is mutated for the possibility of rendering a better fitness. The mutation process is shown in Figure 2. For this experiment each cell in the population was cloned 30 times. After the cloning process, the resulting clones are evaluated. This is an additional 30 evaluation operations per cell in the population per iteration. If one of the evaluated cloned cells has a better fitness than the original cell in the population that it was cloned from, the cloned cell replaces the original.

What keeps the IN algorithm from taking more time to converge on a fitness than the GGA algorithm is the SuppressLowAffinityCells operation on line 22 of Algorithm 1. The operation demonstrated with Equation 12, Algorithm 3, and Figure 6, reduces the population count of cells each iteration. Therefore, during each iteration the count of evaluation functions occur at a diminishing rate. On line 23 of Algorithm 1, after the SuppressLowAffinityCells operation additional cells are added to the population. For this experiment the count of cells in the population was multiplied by 0.6. This calculation is visible in Figure 3 and also shown in Equation 13. The product is then rounded to the closest integer. The rounded result is the number of new cells that are generated and added to the population for subsequent iterations.

Based on the conducted experiment’s results, cells that are added after the SuppressLowAffinityCells operation never exceed the count of cells removed. Since these operations occur each iteration, the count of new cells added to the population each iteration also decreases as shown in Figure 12. The cell count shown is the mean average calculated from 30 runs.

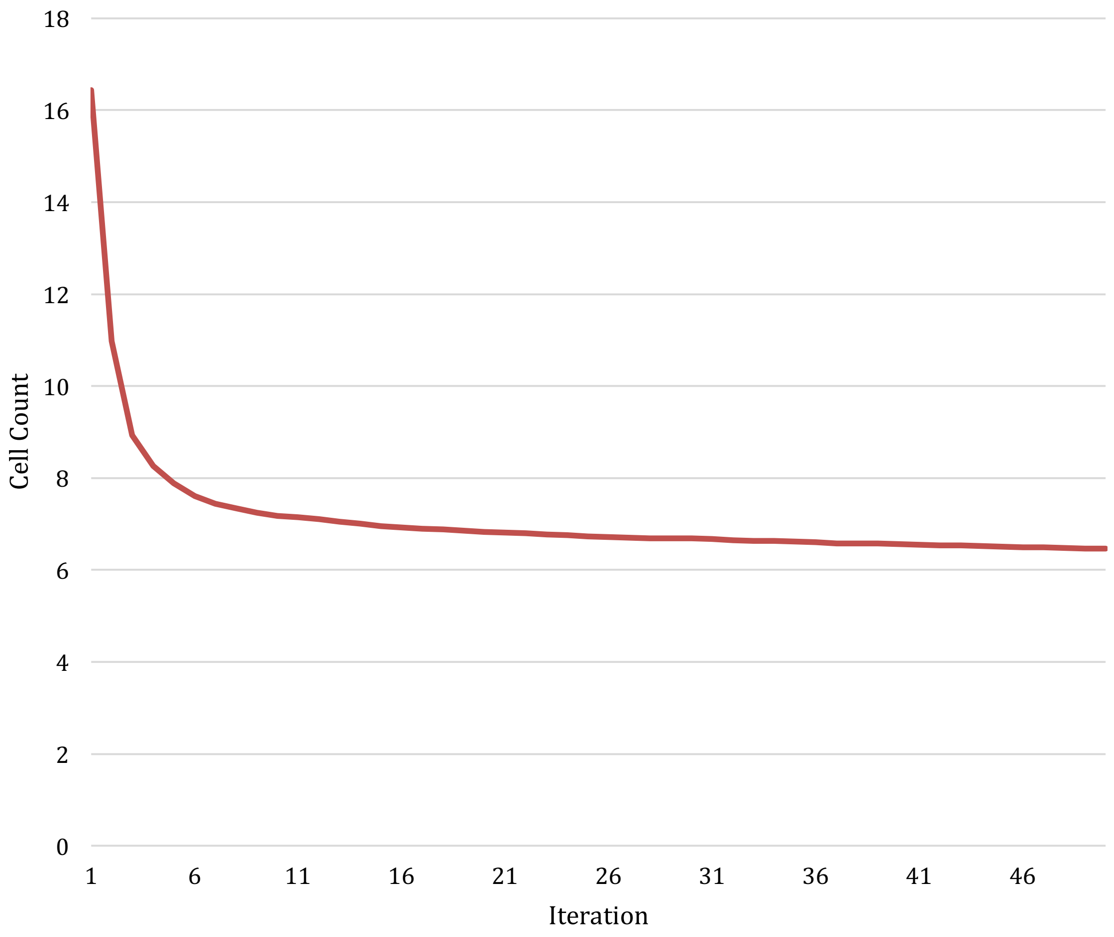


Figure 12. Immune Network's Average Count of Cells.

Both the GGA implementation and the IN implementation are initialized with the same 100 cells, they have the exact same properties, such as VMs and ACs with their attributed specifications, so it is clear that the IN algorithm is efficient in suppressing cells. The IN suppresses 84 cells in the first iteration. Cell counts in the GGA implementation average to a static count as demonstrated in Figure 13.

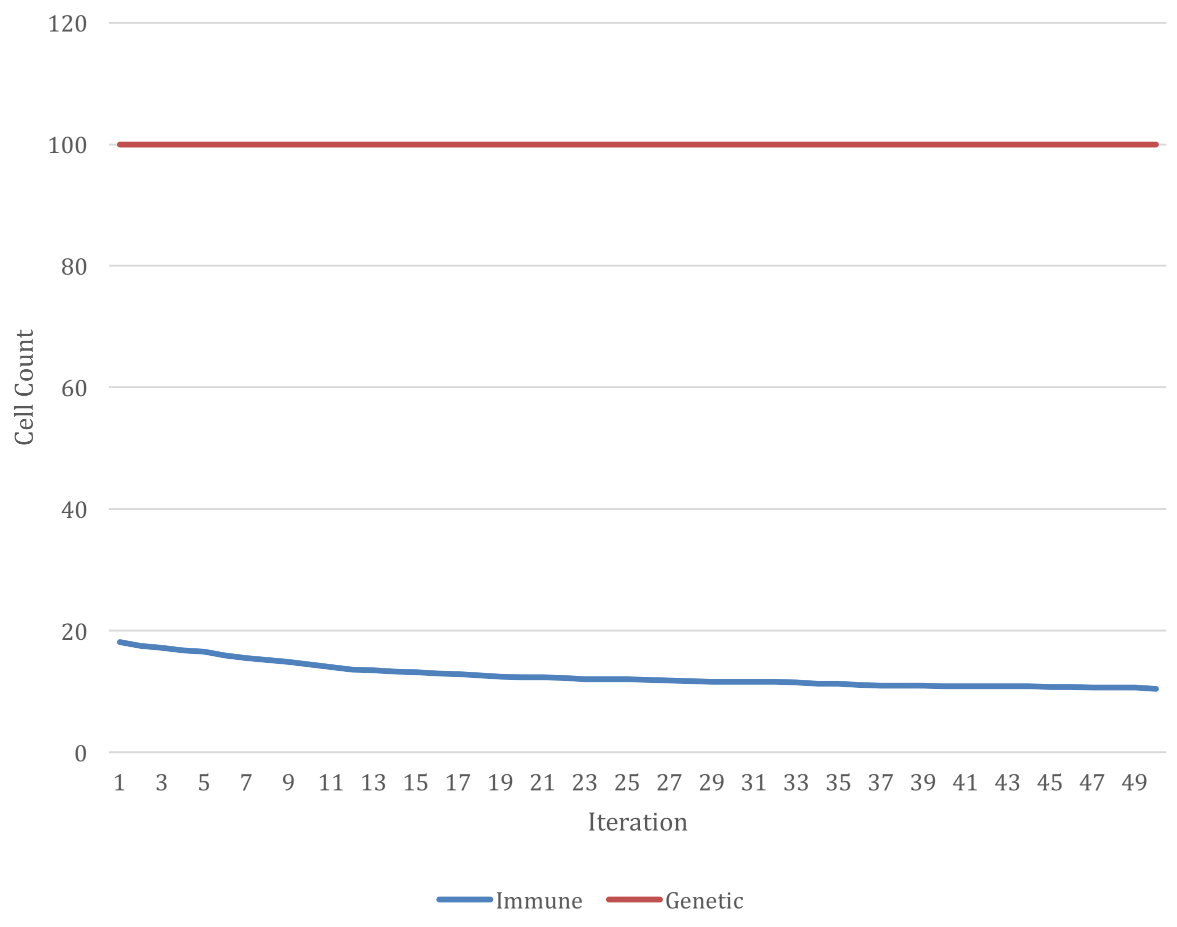


Figure 13. Average Cell Count.

Thus, each subsequent iteration in the IN implementation takes a reduced amount of time to derive a fitness as shown in Figure 14.

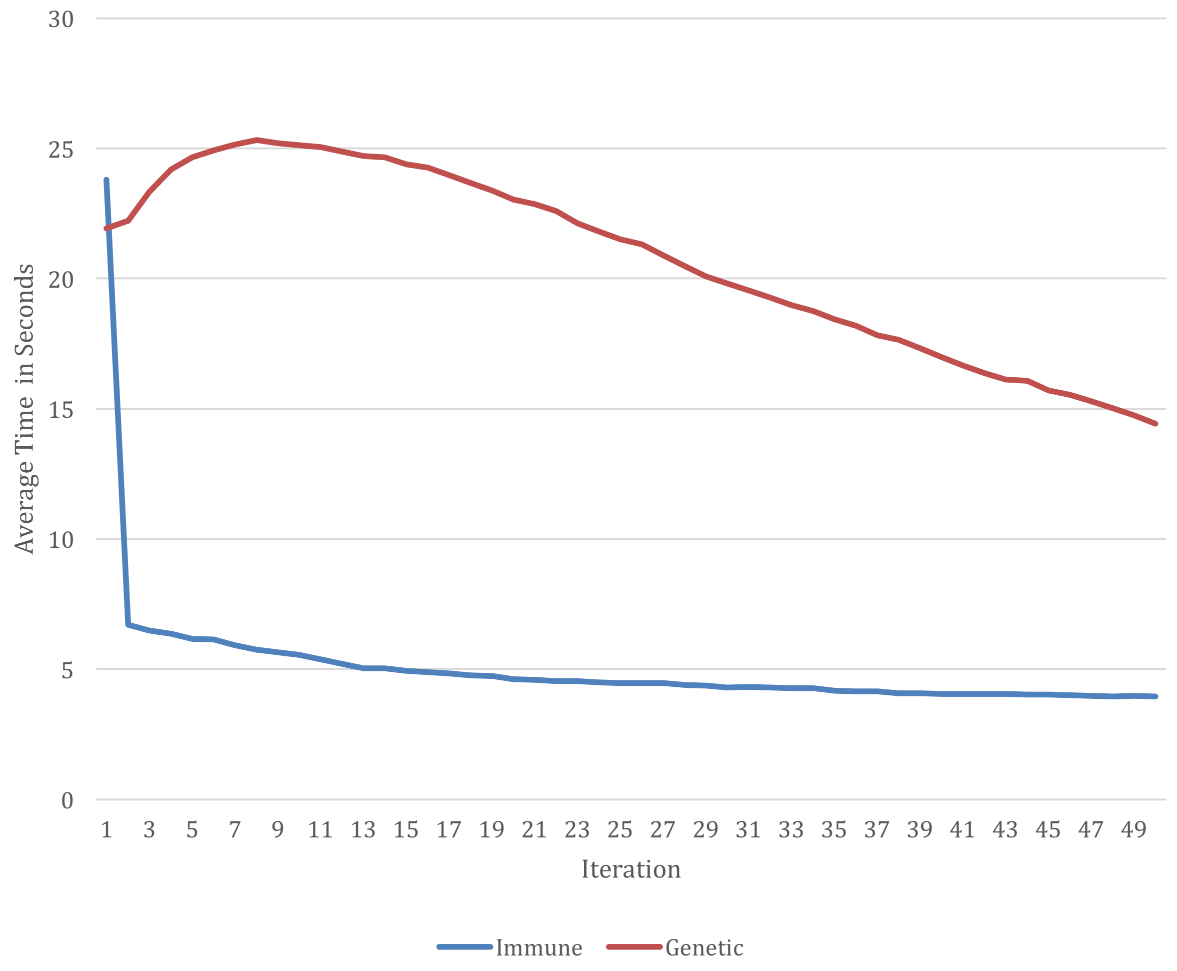


Figure 14. Average Time.

In Figure 15, the overall time taken in seconds for the IN implementation and the GGA implementation are compared.

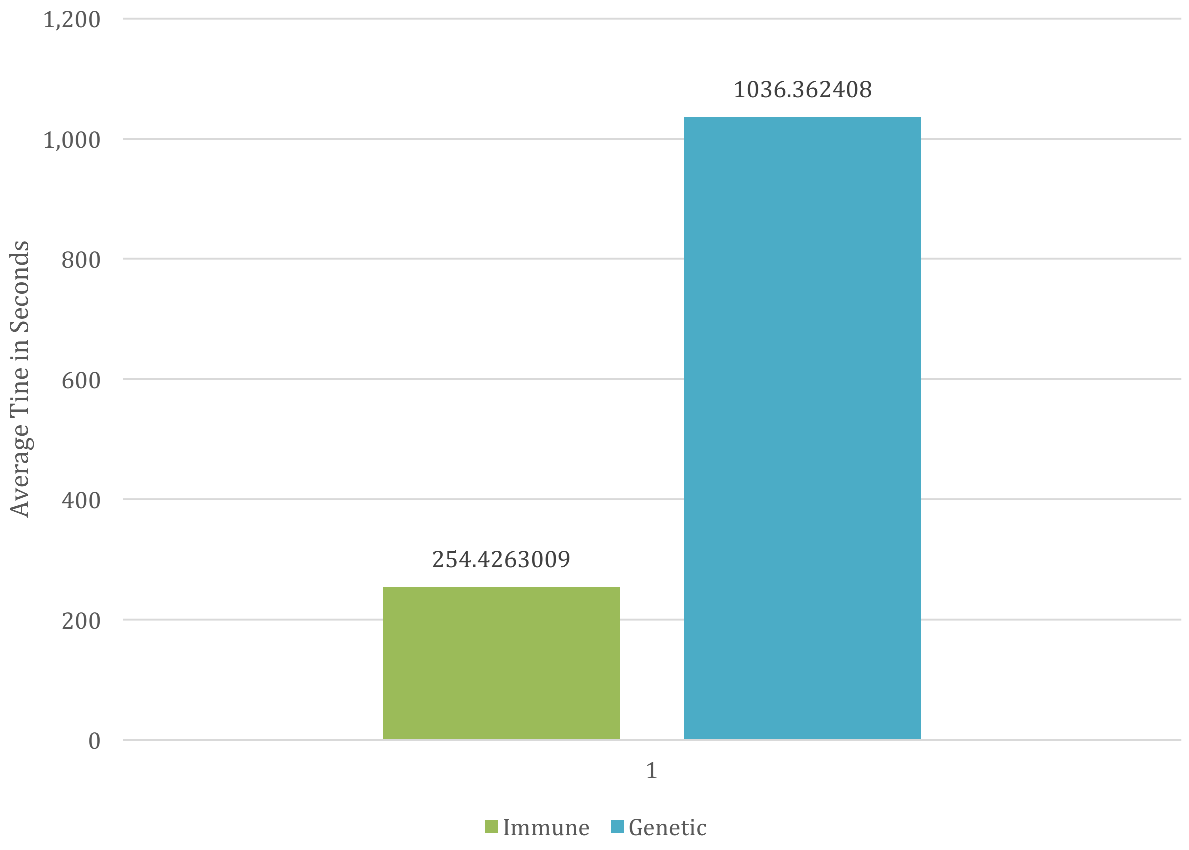


Figure 15. Overall Average Time.

It is clear that the IN implementation takes substantially less time to converge on a fitness if the overall time taken is considered. In fact, it takes 75% less time. To further this point, Figure 16 demonstrates the GGA implementation's additional time taken in seconds per iteration if compared with the IN implementation.

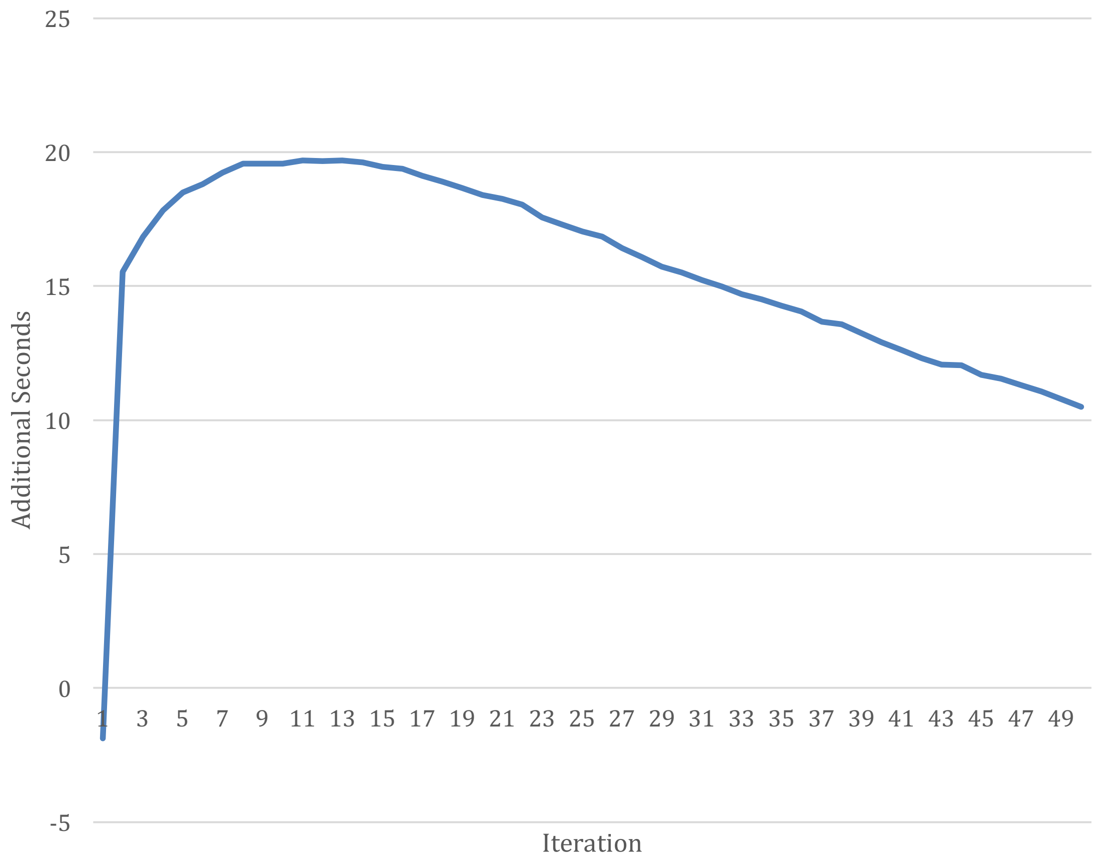


Figure 16. GGA's Average Additional Time.

It is clear in Figures 14, 15, and 16 that the IN implementation takes less time to converge on a fitness than the GGA implementation. However, the GGA implementation on average achieves a better fitness than IN. Cloud service providers must take care in balancing the tradeoffs between using the established GGA algorithm and the IN algorithm for SaaS resource optimization, balancing between optimal fitness achieved and time taken to calculate fitness.

6. CONCLUSIONS AND FUTURE WORK

This thesis examined the possibility of utilizing the Immune Network algorithm to dynamically and optimally manage resources for SaaS in cloud data centers. SaaS are comprised of application components, each having their own requirements, such as processing-, memory-, and storage needs. If any one of these components are not provisioned enough resources to fulfill their requirements, then the cloud SaaS that contains an application component without its required resources is unable to fulfill its service licensing agreement. Resources provided for SaaS are virtual machines with attributed processing-, memory-, and storage capabilities. This thesis demonstrates that the Immune Network algorithm can be applied to dynamically provisioning resources for SaaS in an optimized manner. The algorithm was compared with an existing solution, the Grouping Genetic algorithm. What was discovered was that the Immune Network algorithm implementation does not on average produce better resource strategies than the Grouping Genetic algorithm. However, the Immune Network algorithm does produce resource optimization strategies that are close, fitness wise. The Immune Network algorithm produces a 94% improvement in resource optimization from initialization and the Grouping Genetic algorithm produces a 99% improvement in resource optimization. And, the research conducted for this thesis revealed that the Immune Network algorithm will calculate a resource strategy faster than the Grouping Genetic algorithm by as much as 75%, because the Immune Network algorithm suppresses cells in its population and the Grouping Genetic algorithm does not, which reduces the count of fitness evaluation functions executed.

This thesis's findings provide SaaS providers a tradeoff between calculating a better resource strategy at a cloud data center, but then they are penalized in time taken to do so. Or, a SaaS provider may choose the Immune Network algorithm to generate a resource strategy in less time, but then the provider is penalized in that the resource strategy is not as optimal as the one that is provided by the Grouping Genetic algorithm.

For future work, both the Immune Network algorithm and the Grouping Genetic algorithm take significant time to calculate a resource strategy. To improve the compute time parallel processing might be a solution. The population of cells could be segmented and processed in parallel. Then, subsequently a MapReduce process could merge the results.

REFERENCES

[1] Yusoh, Z.I.M.; Maolin Tang, "Clustering Composite SaaS Components in Cloud Computing using a Grouping Genetic Algorithm," in *Evolutionary Computation (CEC), 2012 IEEE Congress*, Brisbane, Queensland, Australia, 2012, p. 8.

[2] Grandison, T.; Maximilien, E.M.; Thorpe, S.; Alba, A., "Towards a Formal Definition of a Computing Cloud ," in *2010 6th World Congress on Services (SERVICES-1)*, Miami, 2010, pp. 191-192.

[3] Ardagna, C.A.; Damiani, E.; Frati, F.; Montalbano, G.; Rebeccani, D.; Ughetti, M., "A Competitive Scalability Approach for Cloud Architectures," in *In Cloud Computing (CLOUD), 2014 IEEE 7th International Conference*, Anchorage, AK, 2014, pp. 610-617.

[4] Grance, Tim; Mell, Peter, "The NIST Definition of Cloud Computing," September 2011, Official definition of cloud computing.

[5] Vaquero, Luis M.; Rodero-Merino, Luis; Caceres, Juan; Lindner, Maik, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50-55, January 2009.

[6] Foster, I.; Yong Zhao; Raicu, I.; Shiyong Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop*, Austin, TX, 2008, pp. 1-10.

[7] Unuvar, M.; Doganata, Y.; Steinder, M.; Tantawi, A.; Tosi, S., "A predictive method for identifying optimum cloud availability zones," in *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, Anchorage, AK, 2014, pp. 72-79.

[8] Tsakalozos, K.; Verroios, V.; Roussopoulos, M.; Delis, A., "Time-Constrained Live VM Migration in Share-Nothing IaaS-Clouds," in *2014 IEEE 7th International Conference Cloud Computing (CLOUD)*, Anchorage, AK, 2014, pp. 56-63.

[9] Department of Computer Science, University of Surrey. (2015, Oct.) University of Surrey. [Online]. http://www.surrey.ac.uk/cs/research/nice/

[10] Anastasi, G.F.; Carlini, E.; Coppola, M.; Dazzi, P., "QBROKAGE: A Genetic Approach for QoS Cloud Brokering," in *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, Anchorage, AK, 2014, pp. 304-311.

[11] Gulati, Ajay; Holler, Anne; Ji, Minwen; Shanmuganathan, Ganesha; Waldspurger, Carl; Zhu, Xiaoyun, "Vmware distributed resource management: Design, implementation, and lessons learned.," *VMware Technical Journal*, vol. 1, no. 1, pp. 45-64, January 2012.

[12] Xin, Lu; Zilong, Gu, "A load-adapative cloud resource scheduling model based on ant colony algorithm," in *2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 2011, pp. 296-300.

[13] Haoming, Fu; Zongpeng, Li; Chuan, Wu; Xiaowen, Chu, "Core-selecting auctions for dynamically allocating heterogeneous vms in cloud computing," in *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, Anchorage, AK, 2014, pp. 152-159.

[14] Groves, Theodore, "Incentives in teams," *Econometrica: Journal of the Econometric Society*, pp. 617-631, 1973.

[15] Kassa, D.F.; Nahrstedt, K., "SCDA: SLA-aware cloud datacenter architecture for efficient content storage and retrieval," in *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, 2014, pp. 120-127.

[16] Yi, Yao; Jiayin, Wang; Bo, Sheng; Lin, J.; Ningfang, Mi, "HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand," in *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, 2014, pp. 184-191.

[17] Jerne, Niels K., "The generative grammar of the immune system," *Bioscience Reports*, vol. 5, no. 6, pp. 439-451, 1985.

[18] Hoffman, Geoffrey W., *Immune Network Theory*, Vancouver, British Columbia, Canada: The University of British Columbia, 2008.

[19] Hart, Emma; Timmis, Jon, "Application areas of AIS: The past, the present and the future," *Applied soft computing*, vol. 8, no. 1, pp. 191-201, 2008.

[20] Forrest, S.; Perelson, A. S.; Allen, L.; Cherukuri, R., "Self-nonself discrimination in a computer," in *2012 IEEE Symposium on Security and Privacy*, pp. 202.

[21] De Castro, L. N.; Von Zuben, F. J., "The Clonal Selection Algorithm with Engineering Applications," *Proceedings of GECCO*, vol. 2000, pp. 36-39, 2000.

[22] Ayara, M.; Timmis, J.; De Lemos, R.; & Forrest, S., "Immunising Automated Teller Machines," Springer Berlin Heidelberg, 2005.

[23] Khaled, A.; Abdul-Kader, H. M.; Ismail, N. A, "Artificial Immune Clonal Selection Algorithm: A Comparative Study of CLONALG, opt-IA and BCA with Numerical Optimization Problems," *International Journal of Computer Science and Network Security*, vol. 10, no. 4, pp. 24-30, 2010.

[24] Whitbrook, A. M.; Aickelin, U.; Garibaldi, J. M., "Idiotypic immune networks in mobile-robot control," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 37, no. 6, pp. 1581-1598, 2007.

[25] Whitbrook, A. M.; Aickelin, U.; Garibaldi, J. M., "The transfer of evolved artificial immune system behaviours between small and large scale robotic platforms," Springer Berlin Heidelberg, 2010.

[26] Kim, J.; Greensmith, J.; Twycross, J.; Aickelin, U, "Malicious code execution detection and response immune system inspired by the danger theory," *arXiv preprint arXiv:1003.4142*, 2010.

[27] Hunt, J. E.; Cooke, D. E., "Learning using an artificial immune system," *Journal of network and computer applications*, vol. 19, no. 2, pp. 189-212, 1996.

[28] Harmer, P.K.; Williams, P.D.; Gunsch, G.H.; Lamont, G.B., "An artificial immune system architecture for computer security applications," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 3, pp. 252,280, June 2002.

[29] De Castro, L. N.; Timmis, J., *Artificial immune systems: a new computational intelligence approach*, Springer Science & Business Media, 2002.

[30] Watkins, A.; Timmis, J.; Boggess, L., "Artificial immune recognition system (AIRS): An immune-inspired supervised learning algorithm," *Genetic Programming and Evolvable Machines*, vol. 5, no. 3, pp. 291-317, 2004.

[31] (2014, December) Amazon Compute and Storage Instances. [Online]. http://aws.amazon.com/ec2/instance-types/