Project Report

On

"Artificial evolution of ANT"

Submitted in Partial Fulfilment of the requirement for the Award of degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE & ENGINEERING

Under the Supervision of :

Ms. Vandana Dabass

HOD(CSE)

Submitted by :

Tanishq Sharma

Roll No 3067037

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING & TECHNOLOGY

(A Unit Ganga Technical Campus)

SOLDHA, BAHADURGARH

DECLARATION

I am student of B.Tech (CSE) hereby declare that the major project entitled **“Artificial Ant Evolution”** which is submitted to Department of ComputerScience Engineering, School of Engineering & Technology, Soldha Delhi-NCR, affiliated to Maharshi Dayanand University, Rohtak (Haryana) in partial fulfillmentof requirement for the award of the degree of Bachelor of Technology in ComputerScience Engineering, has not been previously formed the basis for the award of anydegree, diploma or other similar title or recognition.

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

Ms. Vandana Dabass	**Tanishq Sharma**
HOD (CSE)	**Roll No. 3067037**
School of Engineering & Technology	

ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

We would like to express our sincere gratitude and heartfelt thanks to **Ms. Vandana Dabass** for her unflinching support and guidance, valuable suggestions and expert advice. Their words of wisdom and expertise in subject matter were of immense help throughout the duration of this project.

We also take the opportunity to thank our Director and all the faculty of the Ganga Technical Campus for helping us by providing necessary knowledge base and resources.

We would also like to thank our parents and friends for their constant support.

TANISHQ SHARMA

Roll No. 3067037

CERTIFICATE

Certified that the Project entitled "**Artificial Ant Evolution**" submitted by **Tanishq Sharma** bearing Roll no. **18BTCSE034** in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (Computer Science and Engineering) of M.D.U Rohtak, is a record ofthe student's own work carried out under my supervision and guidance. To the best of our knowledge, this project has not been submitted to M.D.U or any university or institute for award of a degree. It is further understood that by his/her certificate that undersigned does not endorse or approve of any statement made, opinion expressed or conclusion drawn there in but approve the dissertation only for the purpose for which it is submitted.

Ms. Vandana Dabass	**Dr. Rakesh Rajpal**
HOD(CSE)	**(Director)**
School of Engineering & Technology	**School of Engineering & Technology**

ABSTRACT

Nature is full of miracles, just a little genetic shift can promote the inception of a new species. In this new era of artificial intelligence, we are keen to replicate it on artificial species to thrive in their artificial environment. Ant is a small yet complex creature on earth, different species have different traits and they possess some of the most complex behaviour due to natural selection and survival of the fittest. In our artificial environment, we have produced complex behaviour in ants with the help of artificial intelligence. In our experimental investigations, we have successfully trained the brain (ANN) of ant to feed on food present in our artificial Environment and is now capable of surviving in our artificial environment.

***Keywords*:** Genetic Algorithm, Artificial Neural Network, Evolution, Natural Selection.

LIST OF ACRONYMS

NN	Neural Network
GA	Genetic Algorithm
ANN	Artificial neural network
EA	Evolutionary Algorithms
DNN	Deep Neural Network

LIST OF FIGURES

TABLE OF CONTENTS

Chapter 1
Introduction

1.1 Project Idea

Nature is stuffed with miracles, just a touch genetic shift can promote the inception of a replacement species. During this new era of AI, we are keen to duplicate it on artificial species to thrive in their artificial environment. Ant may be a small yet complex creature onearth, different species have different traits and that they possess a number of the foremost complex behaviors because of action and survival of the fittest. In our artificial earth, we wish to supply complex behavior in ants with the assistance of computing.

Algorithms used for artificial earth:

1) Genetic Algorithm
2) Artificial Neural Network

1.2 Genetic algorithm

In engineering and research, a genetic algorithm (GA) is a metaheuristic algorithm, inspiredby the method of survival that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are preferred algorithms to generate high-quality solutions tooptimization problems and search problems by using biologically inspired operators like mutation, crossover and selection.

The solution of an optimisation problem using the GA methodology involves a stochastic search of the solution space using strings of integers, known as *chromosomes*, which represent the parameters being optimised. Each integer within these chromosomes is known as a *gene* and, for these modelling applications, each gene has a decimal value between 0 and 9. It should be noted that this is not the traditional GA approach where genes are binary quantities. The advantage of the decimal representation for this type of application is that it allows a wider range of possible values in smaller chromosomes and is particularly suitable for both model and design optimisation.

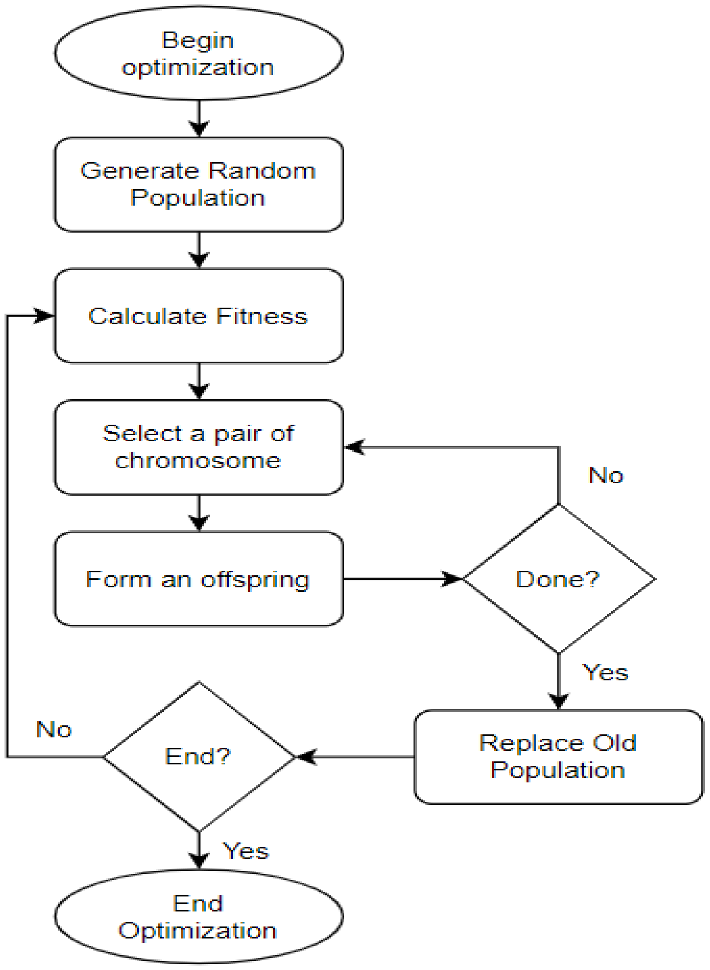


Figure 1.1: Flowchart of genetic algorithm

1.3 Artificial Neural Network

An Artificial Neural Network (ANN) is a type of algorithm that is designed to simulate the neurons in the human brain. This computing system is found to solve some of the most complex problems which have not been solved by simple algorithms. The artificial neural network has self-learning capabilities. The ANN algorithms consist of forward propagation propagation helps the algorithm to learn from the errors that are the mistake done by it and backward propagation

The forward propagation helps the algorithm to find the output through the neurons and backward

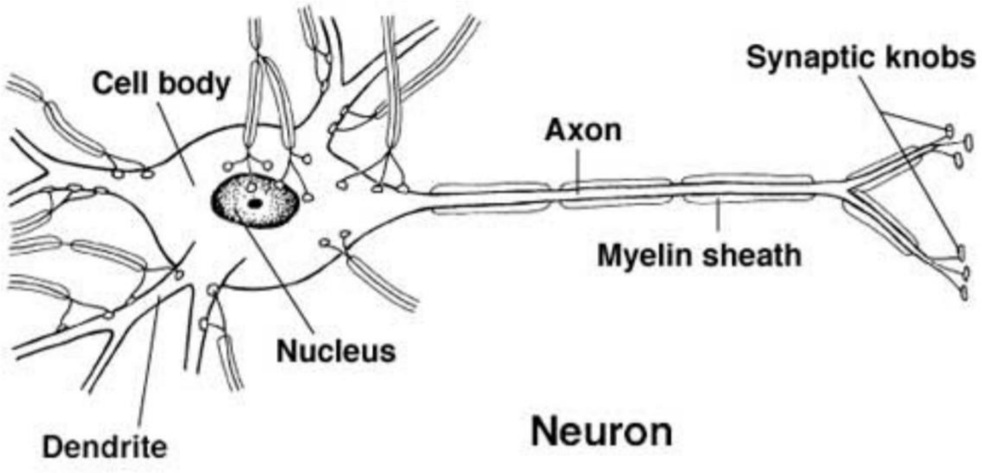


Figure 1.2: Detailed diagram of biological Neuron.[2].

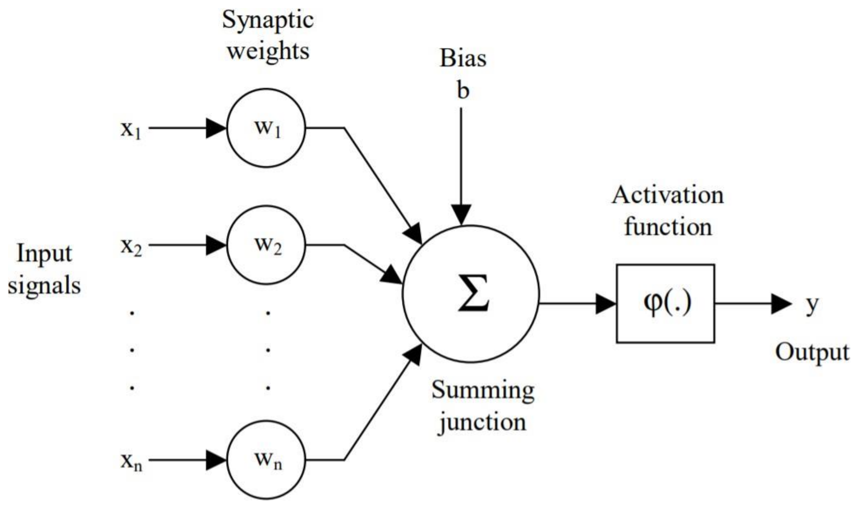


Figure 1.3: Artificial Neuron Architecture.

1.4 Introduction to Neural Networks

The term "**Artificial Neural Network**" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

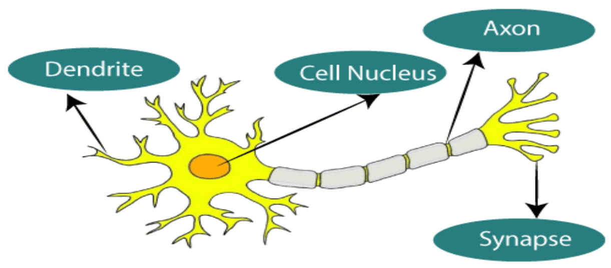


Figure 1.4: Neuron.

The given figure illustrates the typical diagram of Biological Neural Network.

The typical Artificial Neural Network looks something like the given figure.

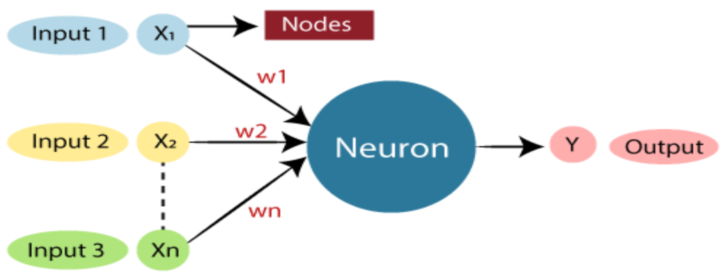


Figure 1.5: Artificial Neuron Architecture.

Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

Biological Neural Network	**Artificial Neural Network**
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

An **Artificial Neural Network** in the field of **Artificial intelligence** where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is designed by programming computers to behave simply like interconnected brain cells.

There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can extract more than one piece of this data when necessary from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

1.5 The architecture of an artificial neural network:

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. In order to define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Lets us look at various types of layers available in an artificial neural network.

Artificial Neural Network primarily consists of three layers:

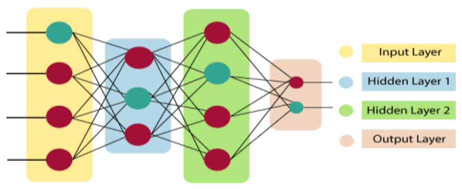


Figure 1.6: Architecture of ANN

Input Layer: As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer: The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

1.6 Advantages of Artificial Neural Network (ANN)

Parallel processing capability:

Artificial neural networks have a numerical value that can perform more than one task simultaneously.

Storing data on the entire network: Data that is used in traditional programming is stored on the whole network, not on a database. The disappearance of a couple of pieces of data in one place doesn't prevent the network from working.

Capability to work with incomplete knowledge:

After ANN training, the information may produce output even with inadequate data. The loss of performance here relies upon the significance of missing data.

Having a memory distribution:

For ANN is to be able to adapt, it is important to determine the examples and to encourage the network according to the desired output by demonstrating these examples to the network. The succession of the network is directly proportional to the chosen instances, and if the event can't appear to the network in all its aspects, it can produce false output.

Having fault tolerance:

Extortion of one or more cells of ANN does not prohibit it from generating output, and this feature makes the network fault-tolerance.

1.7 Disadvantages of Artificial Neural Network:

Assurance of proper network structure:

There is no particular guideline for determining the structure of artificial neural networks. The appropriate network structure is accomplished through experience, trial, and error.

Unrecognized behaviour of the network:

It is the most significant issue of ANN. When ANN produces a testing solution, it does not provide insight concerning why and how. It decreases trust in the network.

Hardware dependence:

Artificial neural networks need processors with parallel processing power, as per their structure. Therefore, the realization of the equipment is dependent.

Difficulty of showing the issue to the network: ANNs can work with numerical data. Problems must be converted into numerical values before being introduced to ANN. The presentation mechanism to be resolved here will directly impact the performance of the network. It relies on the user's abilities.

The duration of the network is unknown:

The network is reduced to a specific value of the error, and this value does not give us optimum results.

Science artificial neural networks that have steeped into the world in the mid-20th century are exponentially developing. In the present time, we have investigated the pros of artificial neural networks and the issues encountered in the course of their utilization. It should not be *overlooked that the cons of ANN networks, which are a flourishing science branch, are eliminated individually, and their pros are increasing day by day. It means that artificial neural networks will turn into an irreplaceable part of our lives progressively important.*

1.8 How do artificial neural networks work?

Artificial Neural Network can be best represented as a weighted directed graph, where the artificial neurons form the nodes. The association between the neurons outputs and neuron inputs can be viewed as the directed edges with weights. The Artificial Neural Network receives the input signal from the external source in the form of a pattern and image in the form of a vector. These inputs are then mathematically assigned by the notations x(n) for every n number of inputs.

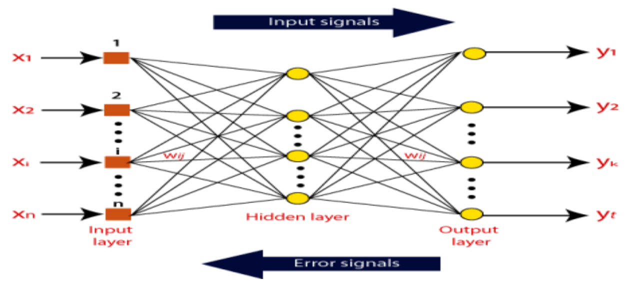


Figure 1.7: Working of ANN

Afterward, each of the input is multiplied by its corresponding weights (these weights are the details utilized by the artificial neural networks to solve a specific problem). In general terms, these weights normally represent the strength of the interconnection between neurons inside the artificial neural network. All the weighted inputs are summarized inside the computing unit.

If the weighted sum is equal to zero, then bias is added to make the output non-zero or something else to scale up to the system's response. Bias has the same input, and weight equals to 1. Here the total of weighted inputs can be in the range of 0 to positive infinity. Here, to keep the response in the limits of the desired value, a certain maximum value is benchmarked, and the total of weighted inputs is passed through the activation function.

The activation function refers to the set of transfer functions used to achieve the desired output. There is a different kind of the activation function, but primarily either linear or non-linear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tan hyperbolic sigmoidal activation functions. Let us take a look at each of them in details:

Binary:

In binary activation function, the output is either a one or a 0. Here, to accomplish this, there is a threshold value set up. If the net weighted input of neurons is more than 1, then the final output of the activation function is returned as one or else the output is returned as 0.

Sigmoidal Hyperbolic:

The Sigmoidal Hyperbola function is generally seen as an "**S**" shaped curve. Here the tan hyperbolic function is used to approximate output from the actual net input. The function is defined as:

F(x) = (1/1 + exp(-????x))

1.9 Types of Artificial Neural Network:

There are various types of Artificial Neural Networks (ANN) depending upon the human brain neuron and network functions, an artificial neural network similarly performs tasks. The majority of the artificial neural networks will have some similarities with a more complex biological partner and are very effective at their expected tasks. For example, segmentation or classification.

Feedback ANN:

In this type of ANN, the output returns into the network to accomplish the best-evolved results internally. As per the **University of Massachusetts**, Lowell Centre for Atmospheric Research. The feedback networks feed information back into itself and are well suited to solve optimization issues. The Internal system error corrections utilize feedback ANNs.

Feed-Forward ANN:

A feed-forward network is a basic neural network comprising of an input layer, an output layer, and at least one layer of a neuron. Through assessment of its output by reviewing its input, the intensity of the network can be noticed based on group behaviour of the associated neurons, and the output is decided. The primary advantage of this network is that it figures out how to evaluate and recognize input patterns.

1.10 Natural Selection

Natural selection, genetic drift, and gene flow are the mechanisms that cause changes in allele frequencies over time. When one or more of these forces are acting in a population, the population violates the Hardy-Weinberg assumptions, and evolution occurs. The Hardy-Weinberg Theorem thus provides a null model for the study of evolution, and the focus of population genetics is to understand the consequences of violating these assumptions.

Natural selection occurs when individuals with certain genotypes are more likely than individuals with other genotypes to survive and reproduce, and thus to pass on their alleles to the next generation. As Charles Darwin (1859) argued in *On the Origin of Species,* if the following conditions are met, natural selection must occur:

1. There is variation among individuals within a population in some trait.
2. This variation is heritable (i.e., there is a genetic basis to the variation, such that offspring tend to resemble their parents in this trait).
3. Variation in this trait is associated with variation in fitness (the average net reproduction of individuals with a given genotype relative to that of individuals with other genotypes).

Directional selection leads to increase over time in the frequency of a favoured allele. Consider three genotypes (*AA*, *Aa* and *aa*) that vary in fitness such that *AA* individuals produce, on average, more offspring than individuals of the other genotypes. In this case, assuming that the selective regime remains constant and that the action of selection is the only violation of Hardy-Weinberg assumptions, the *A* allele would become more common each generation and would eventually become fixed in the population. The rate at which an advantageous allele approaches fixation depends in part on the dominance relationships among alleles at the locus in question (Figure 1). The initial increase in frequency of a rare, advantageous, dominant allele is more rapid than that of a rare, advantageous, recessive allele because rare alleles are found mostly in heterozygotes. A new recessive mutation therefore can't be "seen" by natural selection until it reaches a high enough frequency (perhaps via the random effects of genetic drift — see below) to start appearing in homozygotes. A new dominant mutation, however, is immediately visible to natural selection because its effect on fitness is seen in heterozygotes. Once an advantageous allele has reached a high frequency, deleterious alleles are necessarily rare and thus mostly present in heterozygotes, such that the final approach to fixation is more rapid for an advantageous recessive than for an advantageous dominant allele. As a consequence, natural selection is not as effective as one might naively expect it to be at eliminating deleterious recessive alleles from populations.

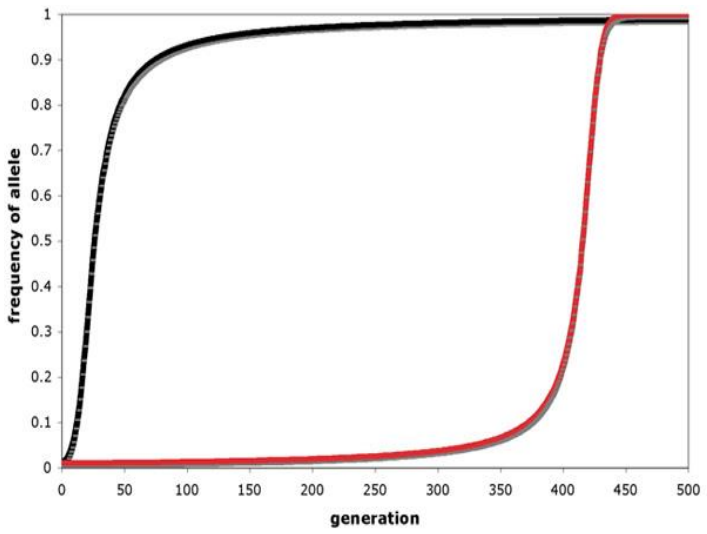


Figure 1.8 : Allele-frequency change under directional selection favouring (a) a dominant advantageous allele and (b) a recessive advantageous allele

Balancing selection, in contrast to directional selection, maintains genetic polymorphism in populations. For example, if heterozygotes at a locus have higher fitness than homozygotes (a scenario known as heterozygote advantage or overdominance), natural selection will maintain multiple alleles at stable equilibrium frequencies. A stable polymorphism can also persist in a population if the fitness associated with a genotype decreases as that genotype increases in frequency (i.e., if there is negative frequency-dependent selection). It is important to note that heterozygote disadvantage (underdominance) and positive frequency-dependent selection can also act at a locus, but neither maintains multiple alleles in a population, and thus neither is a form of balancing selection.

Genetic drift results from the sampling error inherent in the transmission of gametes by individuals in a finite population. The gamete pool of a population in generation t is the total pool of eggs and sperm produced by the individuals in that generation. If the gamete pool were infinite in size, and if there were no selection or mutation acting at a locus with two alleles (*A* and *a*), we would expect the proportion of gametes containing the A allele to exactly equal the frequency of *A*, and the proportion of gametes containing *a* to equal the frequency of *a*. Compare this situation to tossing a fair coin. If you were to toss a coin an infinite number of times, the proportion of heads would be 0.50, and the proportion of tails would be 0.50. If you toss a coin only 10 times, however, you shouldn't be too surprised to get 7 heads and 3 tails.

This deviation from the expected head and tail frequencies is due to sampling error. The more times you toss the coin, the closer these frequencies should come to 0.50 because sampling error decreases as sample size increases.

In a finite population, the adults in generation *t* will pass on a finite number of gametes to produce the offspring in generation *t* + 1. The allele frequencies in this gamete pool will generally deviate from the population frequencies in generation *t* because of sampling error (again, assuming there is no selection at the locus). Allele frequencies will thus change over time in this population due to chance events — that is, the population will undergo genetic drift. The smaller the population size (*N*), the more important the effect of genetic drift. In practice, when modeling the effects of drift, we must consider effective population size (N_e), which is essentially the number of breeding individuals, and may differ from the census size, *N*, under various scenarios, including unequal sex ratio, certain mating structures, and temporal fluctuations in population size.

At a locus with multiple neutral alleles (alleles that are identical in their effects on fitness), genetic drift leads to fixation of one of the alleles in a population and thus to the loss of other alleles, such that heterozygosity in the population decays to zero. At any given time, the probability that one of these neutral alleles will eventually be fixed equals that allele's frequency in the population. We can think about this issue in terms of multiple replicate populations, each of which represents a deme (subpopulation) within a metapopulation (collection of demes). Given 10 finite demes of equal N_e, each with a starting frequency of the *A* allele of 0.5, we would expect eventual fixation of *A* in 5 demes, and eventual loss of *A* in 5 demes. Our observations are likely to deviate from those expectations to some extent because we are considering a finite number of demes (Figure 2). Genetic drift thus removes genetic variation within demes but leads to differentiation among demes, completely through random changes in allele frequencies.

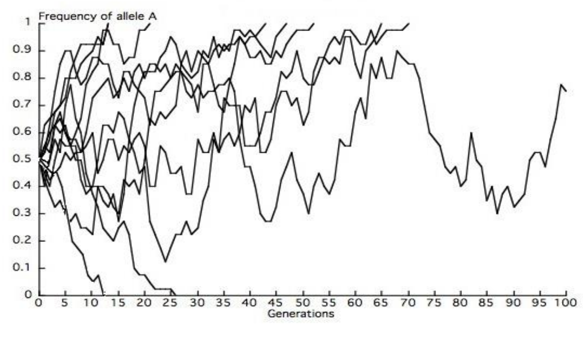


Figure 1.9 : Simulations of allele-frequency change in 10 replicate populations (N = 20)

Since the initial frequency of the A allele = 0.5, we expect A to be fixed in 5 populations and lost in 5 populations, but our observations deviate from expectations because of the finite number of populations. In this run of simulations, we see 7 instances of fixation (p = 1), 2 instances of loss (p = 0), and one instance in which there are still two alleles after 100 generations. In this last population, A would eventually reach fixation or loss.

Gene flow is the movement of genes into or out of a population. Such movement may be due to migration of individual organisms that reproduce in their new populations, or to the movement of gametes (e.g., as a consequence of pollen transfer among plants). In the absence of natural selection and genetic drift, gene flow leads to genetic homogeneity among demes within a metapopulation, such that, for a given locus, allele frequencies will reach equilibrium values equal to the average frequencies across the metapopulation. In contrast, restricted gene flow promotes population divergence via selection and drift, which, if persistent, can lead to speciation.

Natural selection, genetic drift and gene flow do not act in isolation, so we must consider how the interplay among these mechanisms influences evolutionary trajectories in natural populations. This issue is crucially important to conservation geneticists, who grapple with the implications of these evolutionary processes as they design reserves and model the population dynamics of threatened species in fragmented habitats. All real populations are finite, and thus subject to the effects of genetic drift. In an infinite population, we expect directional selection to eventually fix an advantageous allele, but this will not necessarily happen in a finite population, because the effects of drift can overcome the effects of selection if selection is weak and/or the population is small. Loss of genetic variation due to drift is of particular concern in small, threatened populations, in which fixation of deleterious alleles can reduce population viability and raise the risk of extinction. Even if conservation efforts boost population growth, low heterozygosity is likely to persist, since bottlenecks (periods of reduced population size) have a more pronounced influence on Ne than periods of larger population size.

We have already seen that genetic drift leads to differentiation among demes within a metapopulation. If we assume a simple model in which individuals have equal probabilities of dispersing among all demes (each of effective size N_e) within a metapopulation, then the migration rate (m) is the fraction of gene copies within a deme introduced via immigration per generation. According to a commonly used approximation, the introduction of only one migrant per generation ($N_e m = 1$) constitutes sufficient gene flow to counteract the diversifying effects of genetic drift in a metapopulation.Natural selection can produce genetic variation among demes within a metapopulation if different selective pressures prevail in different demes. If N_e is large enough to discount the effects of genetic drift, then we expect directional selection to fix the favored allele within a given focal deme. However, the continual introduction, via gene flow, of alleles that are advantageous in other demes but deleterious in the focal deme, can counteract the effects of selection. In this scenario, the deleterious allele will remain at an intermediate equilibrium frequency that reflects the balance between gene flow and natural selection.

The common conception of evolution focuses on change due to natural selection. Natural selection is certainly an important mechanism of allele-frequency change, and it is the only mechanism that generates adaptation of organisms to their environments. Other mechanisms, however, can also change allele frequencies, often in ways that oppose the influence of selection. A nuanced understanding of evolution demands that we consider such mechanisms as genetic drift and gene flow, and that we recognize the error in assuming that selection will always drive populations toward the most well adapted state.

1.11 Deep Neural Network

A deep neural network (DNN) is an ANN with multiple hidden layers between the input and output layers. Similar to shallow ANNs, DNNs can model complex non-linear relationships.

The main purpose of a neural network is to receive a set of inputs, perform progressively complex calculations on them, and give output to solve real world problems like classification. We restrict ourselves to feed forward neural networks.

We have an input, an output, and a flow of sequential data in a deep network.

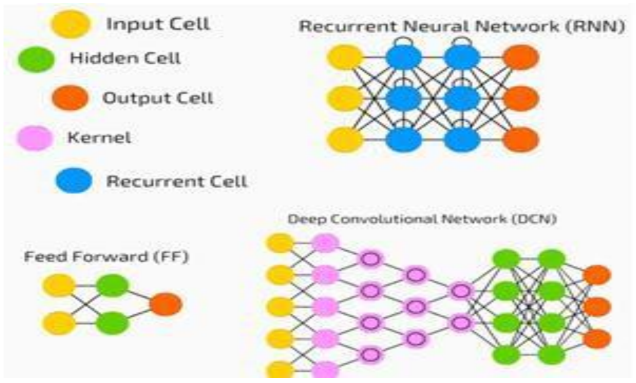


Figure 1.10 : Deep Neural Networks

Neural networks are widely used in supervised learning and reinforcement learning problems. These networks are based on a set of layers connected to each other.

In deep learning, the number of hidden layers, mostly non-linear, can be large; say about 1000 layers.

DL models produce much better results than normal ML networks.

We mostly use the gradient descent method for optimizing the network and minimising the loss function.

We can use the **Imagenet**, a repository of millions of digital images to classify a dataset into categories like cats and dogs. DL nets are increasingly used for dynamic images apart from static ones and for time series and text analysis.

Training the data sets forms an important part of Deep Learning models. In addition, Backpropagation is the main algorithm in training DL models.

DL deals with training large neural networks with complex input output transformations.

One example of DL is the mapping of a photo to the name of the person(s) in photo as they do on social networks and describing a picture with a phrase is another recent application of DL.

The neural network needs to learn all the time to solve tasks in a more qualified manner or even to use various methods to provide a better result. When it gets new information in the system, it learns how to act accordingly to a new situation.
Learning becomes deeper when tasks you solve get harder. Deep neural network represents the type of machine learning when the system uses many layers of nodes to derive high-level functions from input information. It means transforming the data into a more creative and abstract component.
In order to understand the result of deep learning better, let's imagine a picture of an average man. Although you have never seen this picture and his face and body before, you will always identify that it is a human and differentiate it from other creatures. This is an example of how the deep neural network works. Creative and analytical components of information are analyzed and grouped to ensure that the object is identified correctly. These components are not brought to the system directly, thus the ML system has to modify and derive them.

Chapter 2
Genetic Algorithm

2.1 What is Genetic algorithm

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of *mixed integer programming*, where some components are restricted to be integer-valued.

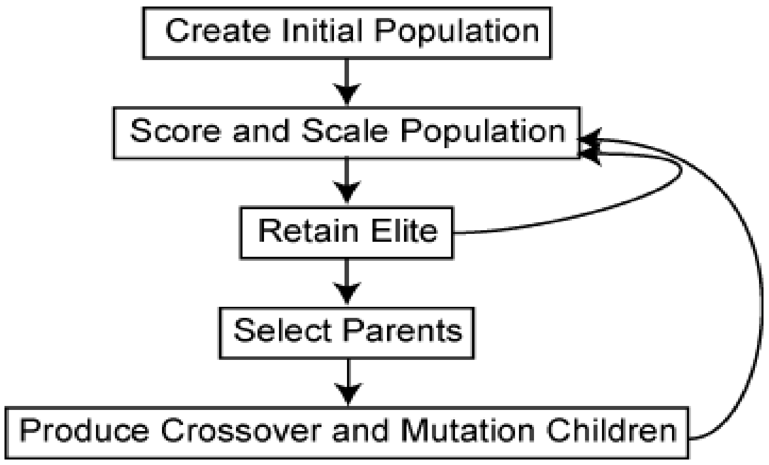

Create Initial Population
Score and Scale Population
Retain Elite
Select Parents
Produce Crossover and Mutation Children

Figure 2.1: Genetic Algorithm Flow

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation. The selection is generally stochastic, and can depend on the individuals' scores.
- *Crossover rules* combine two parents to form children for the next generation.
- *Mutation rules* apply random changes to individual parents to form children.

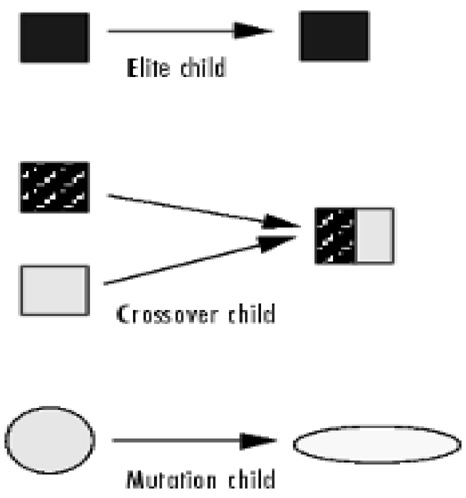


Figure 2.2: Crossing Between Parents Generation

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. **They are commonly used to generate high-quality solutions for optimization problems and search problems.**

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

2.2. Foundation of Genetic Algorithms

Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others

3. Genes from "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

2.2.1. Search space

The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).

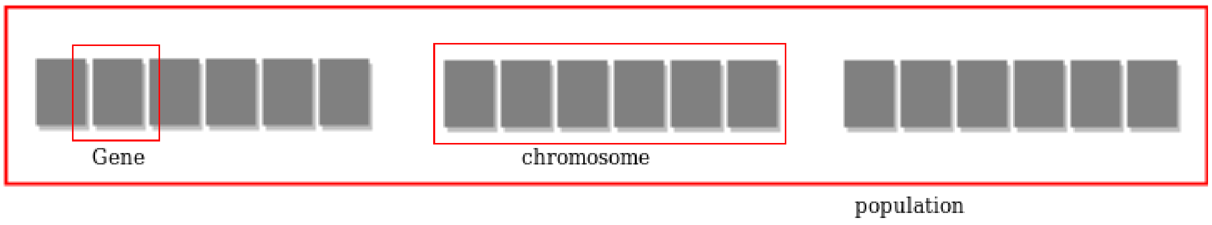


Figure 2.3: Space Search

2.2.2. Fitness Score

A Fitness Score is given to each individual which **shows the ability of an individual to "compete"**. The individual having optimal fitness score (or near optimal) are sought.
The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores. The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce **better offspring** by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.
Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better **"partial solutions"** than previous generations. Once the offspring produced having no significant difference from

offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

2.2.3. Operators of Genetic Algorithms

Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example –

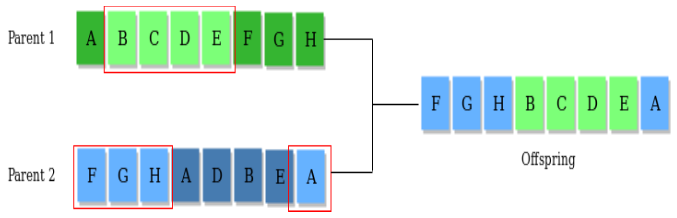


Figure 2.4: Crossover Operation

3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence. For example –

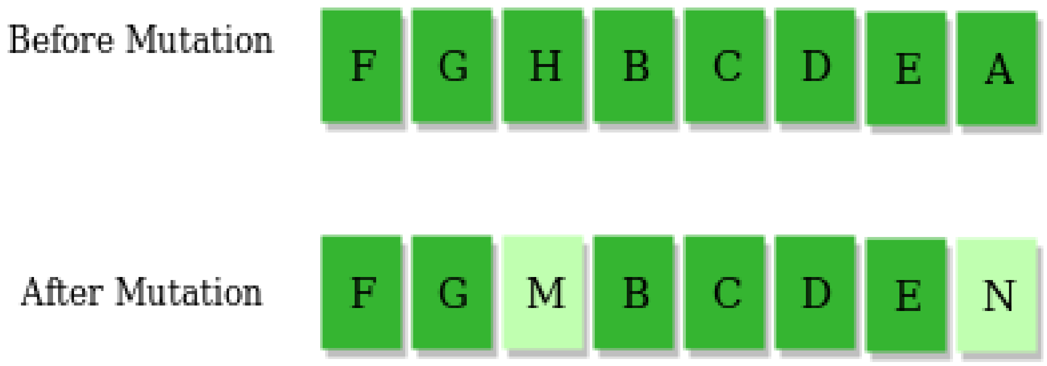


Figure 2.5: Mutation Operation

The whole algorithm can be summarized as –

1) Randomly initialize populations p

2) Determine fitness of population

3) Until convergence repeat:

    a) Select parents from population

    b) Crossover and generate new population

    c) Perform mutation on new population

    d) Calculate fitness for new population

2.3. Why use Genetic Algorithms

- They are Robust
- Provide optimisation over large space state.
- Unlike traditional AI, they do not break on slight change in input or presence of noise

2.4. Application of Genetic Algorithms

Genetic algorithms have many applications, some of them are –

- Recurrent Neural Network
- Mutation testing
- Code breaking
- Filtering and signal processing
- Learning fuzzy rule base etc

Chapter 3
Literature Survey

3.1 Emergent Tool Use From Multi-Agent Interaction

In this paper, they provided various situations to the hide-and-seek agents, Under which they reacted differently and very much humanly and also reflected some out of the box thinking ability like humans.

In this environment, agents play a team-based hide-and-seek game. Hiders are tasked with avoiding line-of-sight from the seekers, and seekers are tasked with keeping the vision of the hiders. There are objects scattered throughout the environment that hiders and seekers can grab and lock in situ, furthermore as randomly generated immovable rooms and walls that agents must learn to navigate. Before the sport begins, hiders are given a preparation phase where seekers are immobilized to provide hiders with an opportunity to run away or change their environment.

As agents train against one another in hide-and-seek, as many as six distinct strategies emerge. Each new strategy creates a previously nonexistent pressure for agents to achieve the following stage. Note that there aren't any direct incentives for agents to interact with objects or to explore; rather, the emergent strategies shown below are a result of the auto curriculum induced by multi-agent competition and therefore the simple dynamics of hide- and-seek.

From few decades' optimizations techniques plays a key role in engineering and technological field applications. They are known for their behaviour pattern for solving modern engineering problems. Among various optimization techniques, heuristic and meta-heuristic algorithms proved to be efficient. In this paper, an effort is made to address techniques that are commonly used in engineering applications. This paper presents a basic overview of such optimization algorithms namely Artificial Bee Colony (ABC) Algorithm, Ant Colony Optimization (ACO) Algorithm, Fire-fly Algorithm (FFA) and Particle Swarm Optimization (PSO) is presented and also the most suitable fitness functions and its numerical expressions have discussed.

Chapter 4
Motivation

Our model is inspired by nature which is quite random. This randomness helps the species to grow in different ways due to which some organisms have dissimilar looking species. The natural selection of survival of the fittest helps the organism to survive in the harsh environment and learn complex behavior which helps them to survive and thrive. In this new age of artificial intelligence, we want to produce similar conditions and results in our artificial environment.

In the research mentioned above agents reacted very intelligently in accordance with the situations. They were able to strategize and execute to win the sport. We want to replicate iton artificial species to thrive in their artificial environment.

Chapter 5
Proposed Work

5.1. Single Ant Interaction and Evolution

In our artificial environment, there is a single ant. The ant's motive is to eat food which is present in the environment.

5.1.1. Environment

The artificial environment consists of a grid of 20 x 20. The ant can't go out of the grid. The environment provides food on the grid, the ant has to eat the food to survive.

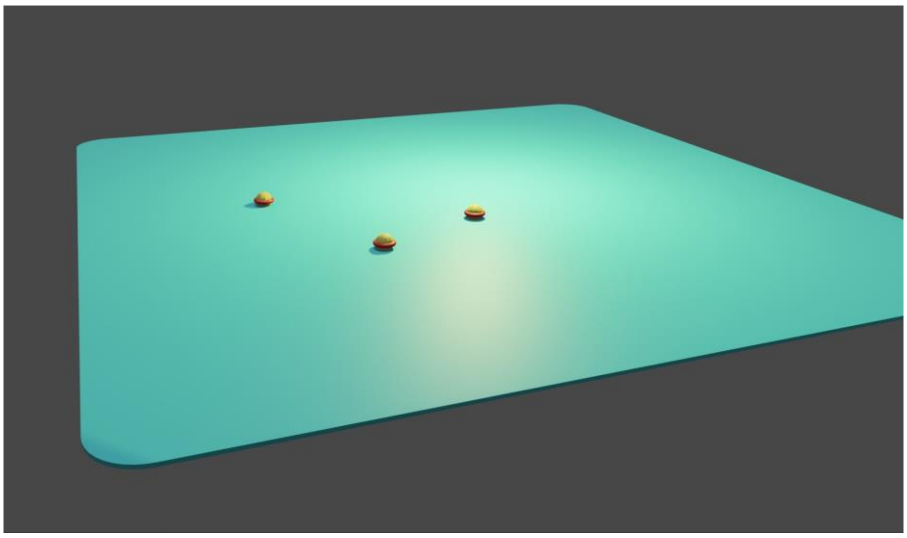


Fig:6.1 Artificial Environment

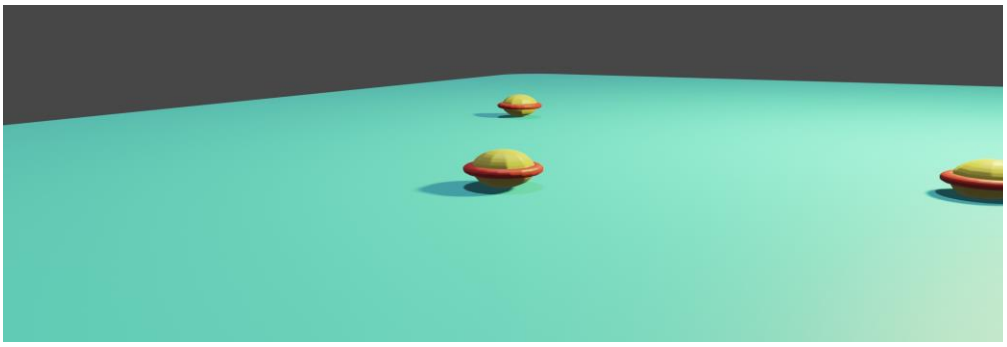


Fig:6.2 Food for ant

These images show the Environment that we provide our artificial ant. The Environment comprises the land area(Grid) on which our ant can roam around and foods which our ant can eat and gain some fitness points.

5.1.2. Ant

The ant resides in the grid in the environment. The ant can move in three directions - left, right and front. The ant can roam freely in the environment, the constraint for the ant is thatif it goes out of board it will die. The ant has to eat the food which the environment provides. The ant has some energy in the start, after moving some distance if the ant does not eat the food then it will have no energy left and it will die.

The ant has a vision, it can look in front of it and the ant has an artificial brain which helpsit to decide the direction to go taking vision as input and direction as output.

The brain of ant consists of a deep neural network with the following specifications -

1. Input has 20 neurons.
2. It has two hidden layers having 20 neurons in the first layer and 12 neurons in thesecond layer.
3. Output has 4 neurons.

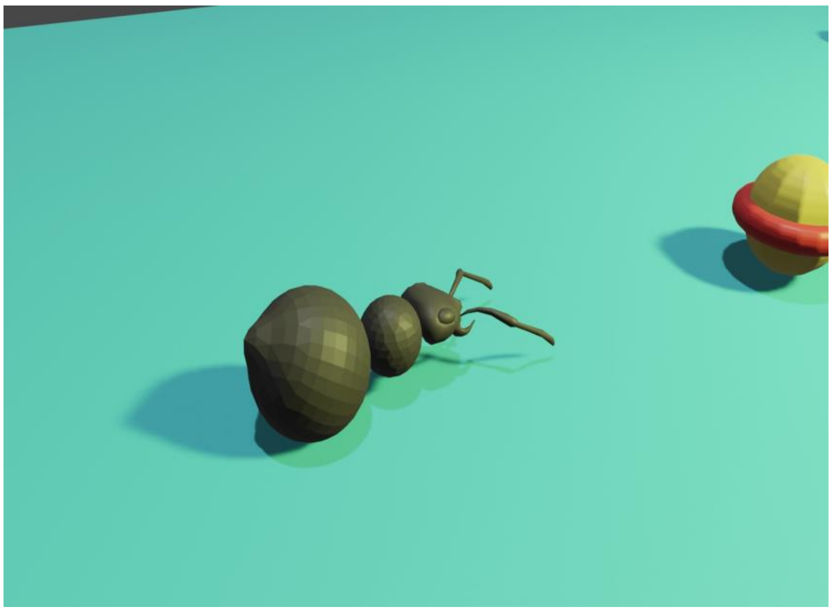


Fig:6.3 Ant

5.1.3. Training Deep Neural Network

We have used a genetic algorithm to train the neural network. The description of every phase of genetic algorithm is as follows -

1. The population of genetic algorithm is taken as a deep neural network with

randomweights.

2. The fitness of the deep neural network is calculated by simulating the decisionsmade by the deep neural network in the environment, the fitness function is proportional to the number of food eaten in the life of the ant.
3. The fitness of the brain (DNN) is sorted in decreasing order, the top fifty per cent ofthe best performing brain of an ant is selected as the parent for making offspring.
4. In our approach, we have not used a crossover operator due to bad performance.
5. The parent is selected from the top fifty per cent and mutation is performed over theweights of the neural network. The mutation rate is taken as five per cent.
6. In our approach, we have taken top one per cent of parent to be passed to nextgeneration so that the fitness in the next generation is not decreased and the complex behavior learnt in previous generations is retained.
7. The old generation is replaced and the whole procedure is repeated to get maximum fitness.

Chapter 6
Conclusion, Future Work & Learning Outcomes

6.1. Conclusion

In our experimental investigations, we have successfully trained the brain (ANN) of ant to get the highest fitness value as 45592083. The maximum food eaten by the ant is 45590. Our trial is based on running the genetic algorithm to 1190 generations after which we have got the best fitness value.

1. The following graph shows the fitness of the best performing ant in each generation.

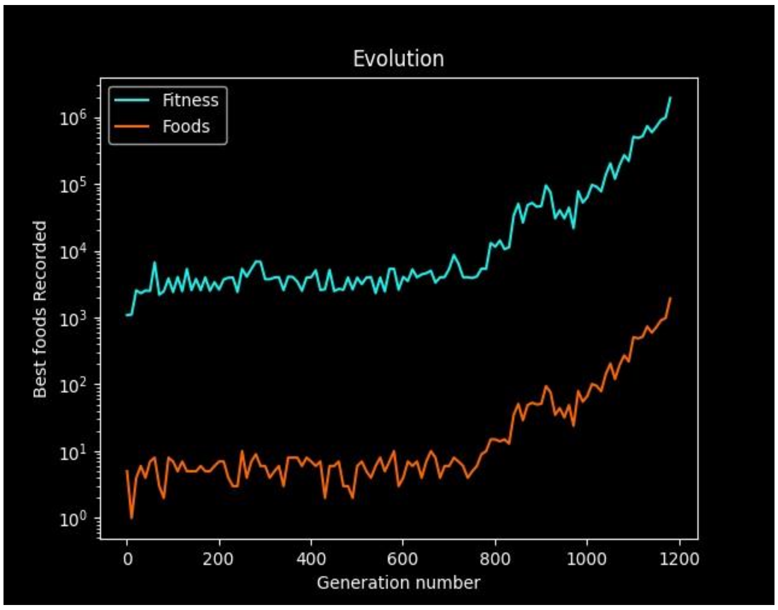


Fig:7.1 Plot of fitness and food eaten by the best performing ant in each generation.

2. The following graph shows the fitness progress in Ants generation vise.

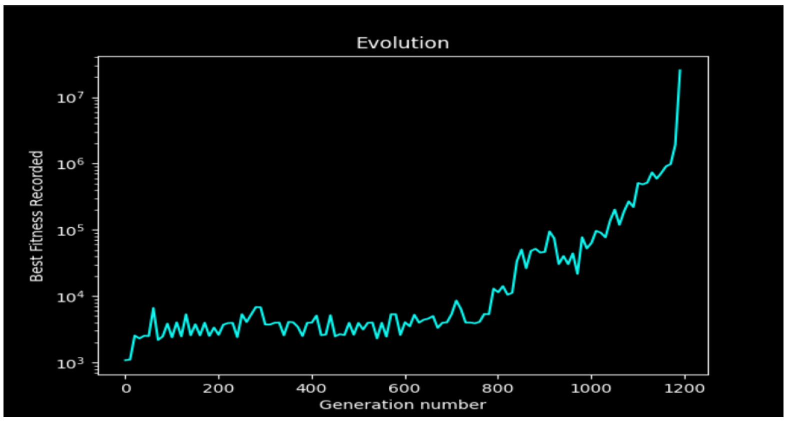


Figure 7.2: Graph for fitness

3. The following graph shows the food capture by Ants generation vise.

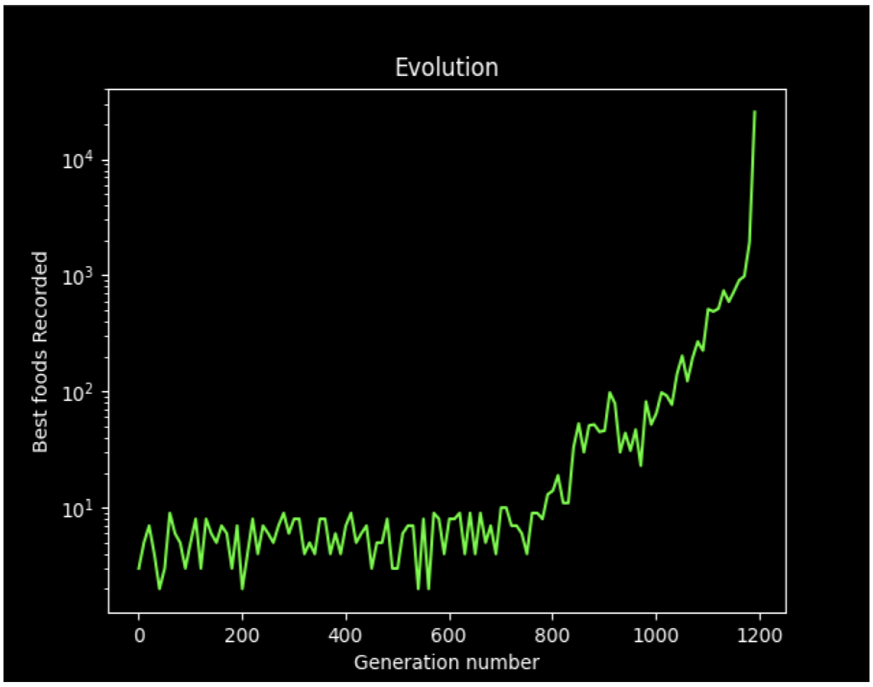


Figure .7: Graph for food

6.2 Future Work

In our model we have taken a single ant to learn the direction it needs to go to get the food, the next goal is to make an artificial environment of multiple ants to interact with each other and the environment to learn some complex behaviors like teamwork, self- organizing, ant colony building, dead reckoning. For training, we have used a genetic algorithm and it has slow learning due to its brute force nature, so for learning reinforcement learning can be used to optimize.

6.3 Learning Outcomes

To accomplish the project's objective we learned about different fields of Artificial Intelligence, Main Algorithms in our project are Genetic Algorithm and Artificial Neural Networks.

Both the Algorithms are combined to replicate the natural selection in our artificial ant. These Algorithms are implemented using python and a 3-D simulation of the outcome is built with the help of blender software.

Chapter 7

CODING

7.1 Ant Class

```
from settings import *
from Vector import *
from math import ceil, floor
import operator
import random
import pygame
import copy


class Ant:
    def __init__(self, dna, food, vector):
        self.dna = copy.deepcopy(dna)
        # self._decision_brain = dna['decision_brain']
        # self.movement_brain = dna['movement_brain']
        # self.offsprint_brain = dna['offspring_brain']
        # direction = [up, up-right, right, right-down, down, down-left, left, left-up]
        self.direction_move = [(0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1)]

        self.vector = vector
        self.food = food
        self.direction = 0

        self.dna['energy'] = 1000
        self.dna['vision_mask'] = self.dna['vision_mask']
        self.dna['vision_distance'] = self.dna['vision_distance']
        self.dna['offspring_range'] = self.dna['offspring_range']
        self.dna['velocity'] = self.dna['velocity']

    def mutate(self, ant):
        ant.dna['vision_mask'][random.randrange(0, 8)] ^= 1
        ant.dna['vision_distance'] = max(1, ant.dna['vision_distance'] + random.randint(-1, 1))
        ant.dna['offspring_range'] = ant.dna['offspring_range'] + random.randint(-1, 1)
        ant.dna['velocity'] = max(1, ant.dna['velocity'] + random.randrange(-1, 2))

        # ant.dna = {'energy': ant.energy, 'vision_mask': ant.vision_mask, 'vision_distance':
ant.vision_distance, 'offspring_range': ant.offspring_range, 'velocity': ant.velocity}

    def energy_calc(self, vector):
        return max(1, (self.dna['energy']/1000)) + self.dna['velocity']*box_distance(self.vector,
vector) + (self.dna['vision_mask'].count(1)*self.dna['vision_distance'])/10

    def x_y_vision(self, x, y):
        if x < 0:
            x = -1
        elif x > 0:
            x = 1
        if y < 0:
            y = -1
        elif y > 0:
            y = 1

        ind = self.direction
        for _ in range(8):
            if (x, y) == self.direction_move[ind]:
                return ind
            ind = (ind + 1)%8
        return 8

    def move(self):

        nearest_food = []
        nearest_food_to_move = None
        nearest_location_to_move = None
        # map_ind = [column, row, right_diagonal, left_diagonal]
        map_ind = [self.vector.x, self.vector.y, self.vector.x + self.vector.y, GRID_WIDTH-
self.vector.x + self.vector.y]
        ind = 0
        for map in self.food.food_map.values():
            for food in map[map_ind[ind]]:
                x = self.vector.x - food.x
                y = self.vector.y - food.y
                vs_msk = self.x_y_vision(x, y)
                if box_distance(self.vector, food) <= self.dna['vision_distance'] and (vs_msk == 8
or self.dna['vision_mask'][vs_msk]):
                    nearest_food.append([box_distance(food, self.vector), food])
            ind += 1



        # move to the nearest food
        if len(nearest_food) > 0:
            nearest_food.sort(key = operator.itemgetter(0))
            nearest_food_len = nearest_food[0][0]

            # move to random food having distance same as nearest food
            ind = 0
            while ind < len(nearest_food):
                if nearest_food[ind][0] > nearest_food_len:
                    break
                ind += 1
            rnd = random.randrange(0, ind)
            nearest_food_to_move = nearest_food[rnd][1]

            # move directly to food position if velocity is greater than distance
            # if box_distance(nearest_food_to_move, self.vector) <= self.dna['velocity'] and
self.dna['energy'] >= self.energy_calc(nearest_food_to_move):
            if box_distance(nearest_food_to_move, self.vector) <= self.dna['velocity']:
                nearest_location_to_move = nearest_food_to_move
                self.vector = nearest_location_to_move
                return nearest_location_to_move, nearest_food_to_move

            # move randomly to any 8 pos which is near to the nearest food
            else:
                min_distance = 1e9
                for move in slopes:
                    x, y = move[0]*self.dna['velocity'] + self.vector.x, move[1]*self.dna['velocity'] +
self.vector.y
                    # if in_grid(x, y) and self.dna['energy'] >= self.energy_calc(Vector(x, y)):
                    if in_grid(x, y):
                        euclid_dist = euclidean_distance(nearest_food_to_move, Vector(x, y))
                        if euclid_dist < min_distance:
                            min_distance = euclid_dist
                            nearest_location_to_move = Vector(x, y)



        # if no food found in vision move randomly
        if nearest_food_to_move == None:
            moves = slopes
            random.shuffle(moves)
            for move in moves:
                x, y = move[0] + self.vector.x, move[1] + self.vector.y
                # if in_grid(x, y) and self.energy_calc(Vector(x, y)) <= self.dna['energy']:
                if in_grid(x, y):
                    nearest_location_to_move = Vector(x, y)
                    break

        self.dna['energy'] -= self.energy_calc(nearest_location_to_move)
        self.vector = nearest_location_to_move



        return nearest_location_to_move, None

      def draw(self, WIN):
        ant_image = pygame.Surface((TILE_SIZE, TILE_SIZE))
        ant_image.fill((0, 0, 0))
        WIN.blit(ant_image, (self.vector.x*TILE_SIZE, self.vector.y*TILE_SIZE))
```

7.2 Food & Gird Class

```
from settings import *
from Vector import *
import random
import pygame

class Food:
    def __init__(self, num_food):
        self.food = []
        self.energy_in_food = 100
        self.food_competition = {}
        self.food_map = {
            'column' : [[] for i in range(GRID_HEIGHT+GRID_WIDTH)],
            'row' : [[] for i in range(GRID_HEIGHT+GRID_WIDTH)],
            'right_diagonal' : [[] for i in range(GRID_HEIGHT+GRID_WIDTH)],
            'left_diagonal' : [[] for i in range(GRID_HEIGHT+GRID_WIDTH)]
        }
        for _ in range(num_food):
            self.add_food()

    def mapind(self, food):
        return [food.x, food.y, food.x + food.y, GRID_WIDTH-food.x + food.y]

    def add_food(self):
        x, y = random.randrange(0, GRID_WIDTH), random.randrange(0, GRID_HEIGHT)
        food = Vector(x, y)
        self.food.append(food)
        map_ind = self.mapind(food)
        ind = 0
        for map in self.food_map.values():
            map[map_ind[ind]].append(food)
            ind += 1

    def add_competition(self, food_on_ant, ant):
        if food_on_ant in self.food_competition:
            self.food_competition[food_on_ant].append(ant)
        else:
            self.food_competition[food_on_ant] = [ant]

    def get_energy(self):
        energy_dict = {}
        for ants in self.food_competition.values():
            l = len(ants)
            for ant in ants:
                energy_dict[ant] = self.energy_in_food/l

        for food in self.food_competition.keys():
            self.food.remove(food)
            map_ind = self.mapind(food)
            ind = 0
            for map in self.food_map.values():
                map[map_ind[ind]].remove(food)
                ind += 1



        self.food_competition = {}
        return energy_dict

    def draw(self, WIN):
        food_image = pygame.Surface((TILE_SIZE, TILE_SIZE))
        food_image.fill((255, 0, 0))
        for food in self.food:
            WIN.blit(food_image, (food.x*TILE_SIZE, food.y*TILE_SIZE))
        def __str__(self):
            return str(self.__class__) + ": " + str(self.__dict__)

        def __repr__(self):
            return str(self.__class__) + ": " + str(self.__dict__)
```

7.3 Genetic Algorithm Class

```
import numpy as np
import random
from NerualNetwork import *
import time, datetime
from numpy import loadtxt
import operator
import copy
from random import randrange
from math import floor
from numpy import loadtxt, savetxt
import pickle
import os

population = []
generationCount = 0
popRanked = {}
def Ga(X, Y, n_h, main, generations=10, popSize=100, eliteSize=10, mutationRate=0.05):

  def initial_population(popSize):
    population=[]

    for i in range(popSize):
        population.append(NeuralNetwork(X, Y, n_h))
    return population
```

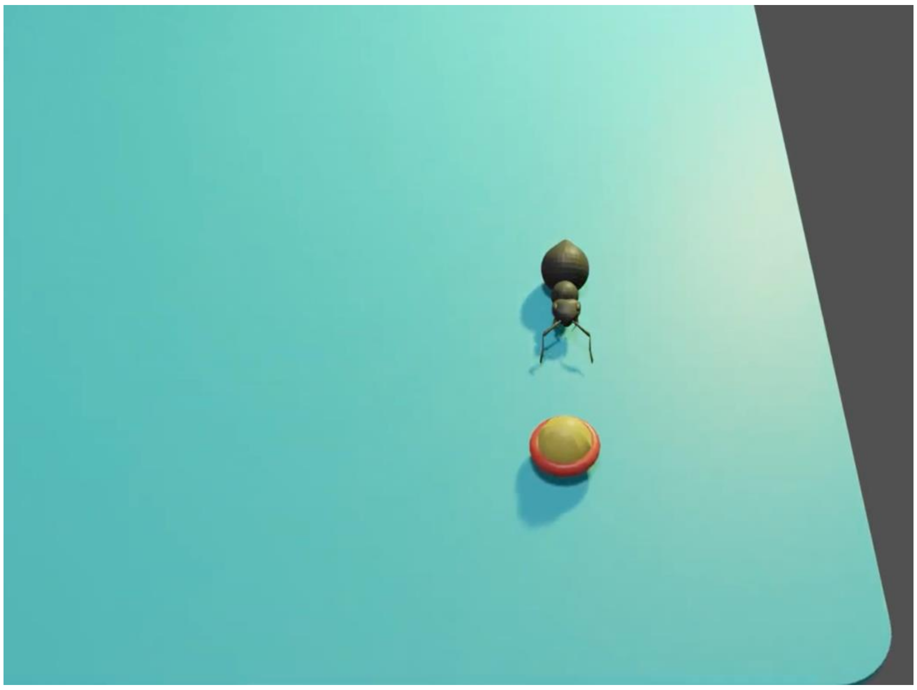
```
def mutation(child, mutationRate):
  global generationCount
  scale = 1
  lscale = 0.0
  # Add decay in scale
  # scale = scale/pow(10, generationCount//10)
  for _, params in child.params.items():
    if random.random() <= mutationRate:
      params += np.random.normal(loc=lscale, scale=scale, size=params.shape)

def rankPopulation():
  global population, popRanked
  popRanked = main(population)
  # fitnessSum=0
  # for i in range(len(population)):
  #   # print(population[i], population[i].food)
  #   fit = population[i].compute_cost(X, Y)
  #   # fitnessSum+=fit
  #   popRanked[i] = fit
  popRanked = sorted(popRanked.items(), key = operator.itemgetter(1), reverse = True)
  # return fitnessSum, rankedPopulation

def random_pick():
  global popRanked
  parentSelectPercentage = 0.5
  l = 0
  r = floor(parentSelectPercentage * len(popRanked))
  return randrange(l, r+1)

def next_generation(eliteSize, mutationRate):
  global population
  global popRanked
  # popRanked = rankPopulation()
  # fitnessSum = popRanked[0]
  newPopulation = []
  for i in range(eliteSize):
    newPopulation.append(population[popRanked[i][0]])
  for i in range(len(population)-eliteSize):
    tmpPop = copy.deepcopy(population[random_pick()])
    mutation(tmpPop, mutationRate)
    newPopulation.append(tmpPop)
  return newPopulation

def genetic_algorithm(popSize, eliteSize, mutationRate, generations):
  global population, generationCount, popRanked
  generationCount = 0
  population = initial_population(popSize)
  # popRanked = rankPopulation()
  # print("Initial fitness: " + str(popRanked[0][1]))
  best_fitness = -1e9
  best_pop = []

  for i in range(generations):
    generationCount += 1
    rankPopulation()
    fitness = popRanked[0][1]
    if best_fitness < fitness:
      best_fitness = fitness
      best_pop = copy.deepcopy(population[popRanked[0][0]])
      with open('best_weight.pickle', 'wb') as handle:
          pickle.dump(best_pop, handle, protocol=pickle.HIGHEST_PROTOCOL)
    print("Generation : {}\t Fitness: {}".format(str(i+1), str(fitness)))

    population = next_generation(eliteSize, mutationRate)
    # if i%100==0:
    #   if not os.path.exists('weights'):
    #     os.makedirs('weights')
      # with open('weights/weights{}.pickle'.format(i), 'wb') as handle:
        #     pickle.dump(best_pop, handle, protocol=pickle.HIGHEST_PROTOCOL)
        # for i in range(1, 100):
        #   with open('weights/weights{}.pickle'.format(i), 'wb') as handle:
        #     pickle.dump(population[popRanked[i][0]], handle,
protocol=pickle.HIGHEST_PROTOCOL)
         # savetxt('{}.csv'.format(name), , delimiter=',')
        # savetxt('antW2.csv', best_pop.ant.brain.W2, delimiter=',')
        # savetxt('antb1.csv', best_pop.ant.brain.b1, delimiter=',')
        # savetxt('antb2.csv', best_pop.ant.brain.b2, delimiter=',')
    return best_pop
  return genetic_algorithm(popSize, eliteSize, mutationRate, generations)
```

7.4 Environment Class

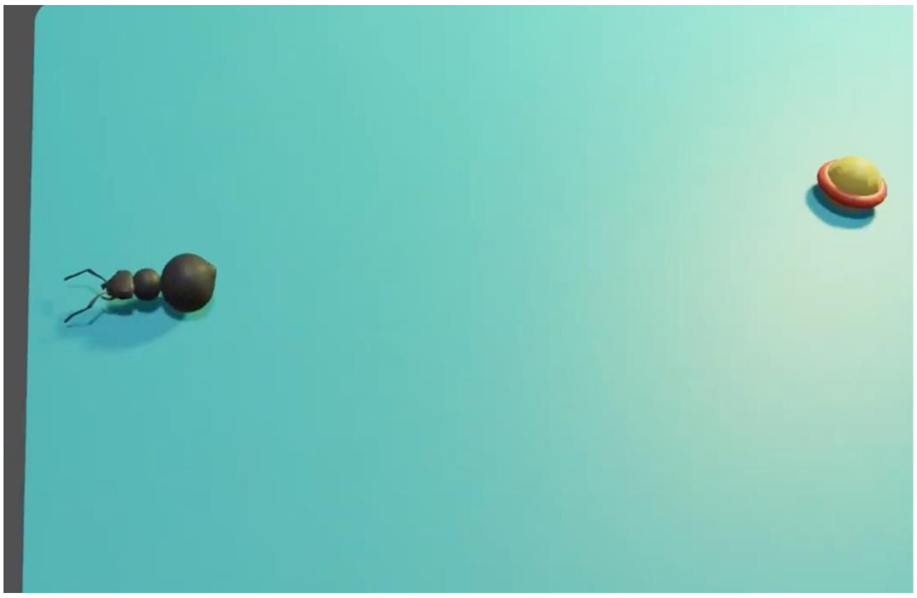
```
from operator import le
from Food import *
from Ant import *
from Vector import *
from settings import *

def custom_ant_comparator(ant):
    return ant.dna['energy']

class Environment:
    def __init__(self, dna, num_initial_ants, num_initial_food, day_length):
        self.base_dna = dna
        self.food = Food(num_initial_food)
        self.ants = []
        self.add_ants([], num_initial_ants)
        self.day_length = day_length
        self.num_initial_food = num_initial_food

    def add_ants(self, ants, num_ants, mutation = False):
        if len(ants) == 0:
            for _ in range(num_ants):
                ant = Ant(self.base_dna, self.food, self.random_location())
                if mutation:
                    ant.mutate(ant)
                self.ants.append(ant)
        else:
            self.ants.extend(ants)

    def random_location(self):
        return Vector(random.randrange(0, GRID_WIDTH), random.randrange(0,
GRID_HEIGHT))

    def rank_population(self):
        self.ants.sort(key=custom_ant_comparator, reverse=True)

    def random_pick(self):
        parent_select_percentage = 0.5
        l = 0
        r = floor(parent_select_percentage * len(self.ants))
        return random.randrange(l, r+1)

    def mating(self):
        parent1 = self.ants[self.random_pick()]
        parent2 = self.ants[self.random_pick()]
        child = Ant(parent1.dna, self.food, self.random_location())
        for key in parent2.dna.keys():
            if random.randrange(0, 2)==1:
                child.dna[key] = parent2.dna[key]
        child.dna['energy'] = 1000
        if random.random() <= 0.5:
            child.mutate(child)
        return child

    def next_generation(self, elitesize):
        ants = []
        self.rank_population()
        # print(self.ants[0].energy, self.ants[0].velocity)
        eng = 0
        for ant in self.ants:
            eng += ant.dna['energy']
        print(eng, self.ants[0].dna)
        self.food = Food(self.num_initial_food)
        for _ in range(elitesize):
            ants.append(Ant(self.ants[0].dna, self.food, self.random_location()))
        for _ in range(len(self.ants) - elitesize):
            ants.append(self.mating())
        self.ants = ants
        # for ant in ants:
        #     print(ant.dna)

    def loop(self, gui):
        if gui:
            FPS = 7
            WHITE = (200, 200, 200)
            pygame.display.set_caption("Ant Game")
            clock = pygame.time.Clock()
            ANT_IMAGE = None
            WIN = pygame.display.set_mode((WIN_WIDTH,WIN_HEIGHT))

        QUIT = False
        while QUIT == False and len(self.food.food) > 0:
            if gui:
                clock.tick(FPS)
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        QUIT = True
                        pygame.quit()
                        quit()
                # print(ant, end='\n\n\n\n')
                WIN.fill((255, 255, 255))
                self.food.draw(WIN)
                for ant in self.ants:
                    ant.draw(WIN)
                pygame.display.update()

            less_energy_ant = []
            for ant in self.ants:
                ant_new_locaion, food_on_ant = ant.move()
                if ant_new_locaion == None:
                    less_energy_ant.append(ant)
                elif food_on_ant != None:
                    self.food.add_competition(food_on_ant, ant)

            energy_dict = self.food.get_energy()
            for ant in self.ants:
                if ant in energy_dict:
                    ant.dna['energy'] += energy_dict[ant]

            # print(ant.energy)
        # for ant in self.ants:
        #     print(ant.energy, ant.velocity)

    def ga(self, gui, num_iter, elitesize):
        while num_iter > 0:
            if num_iter >= 2:
                self.loop(gui)
            else:
                break
                print(len(self.ants))
                self.loop(True)
            # print('\n\n\n')
```

Chapter 8
Ant Generations

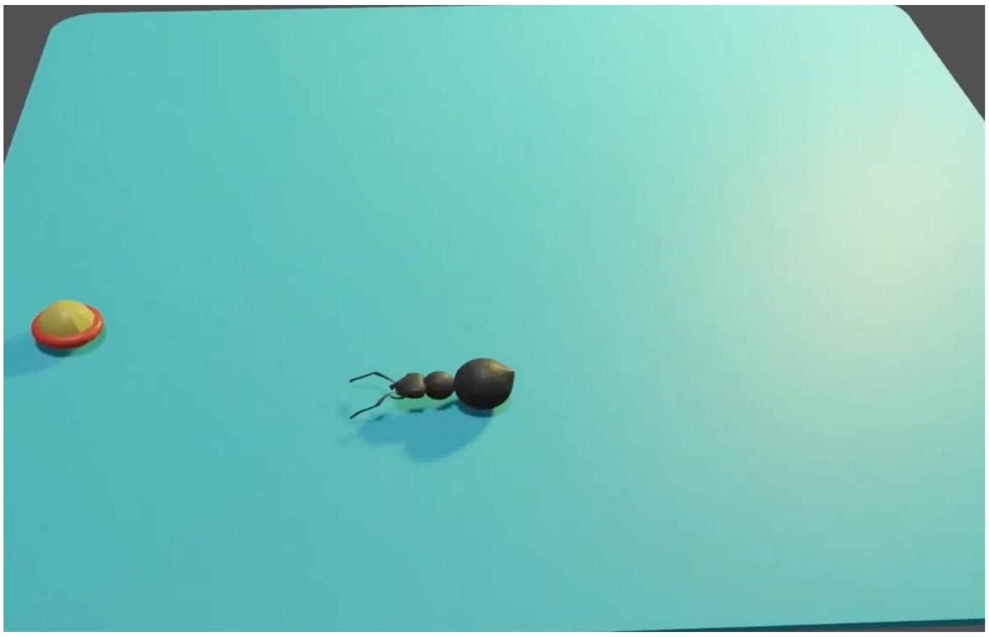
8.1 1st Generation Ant



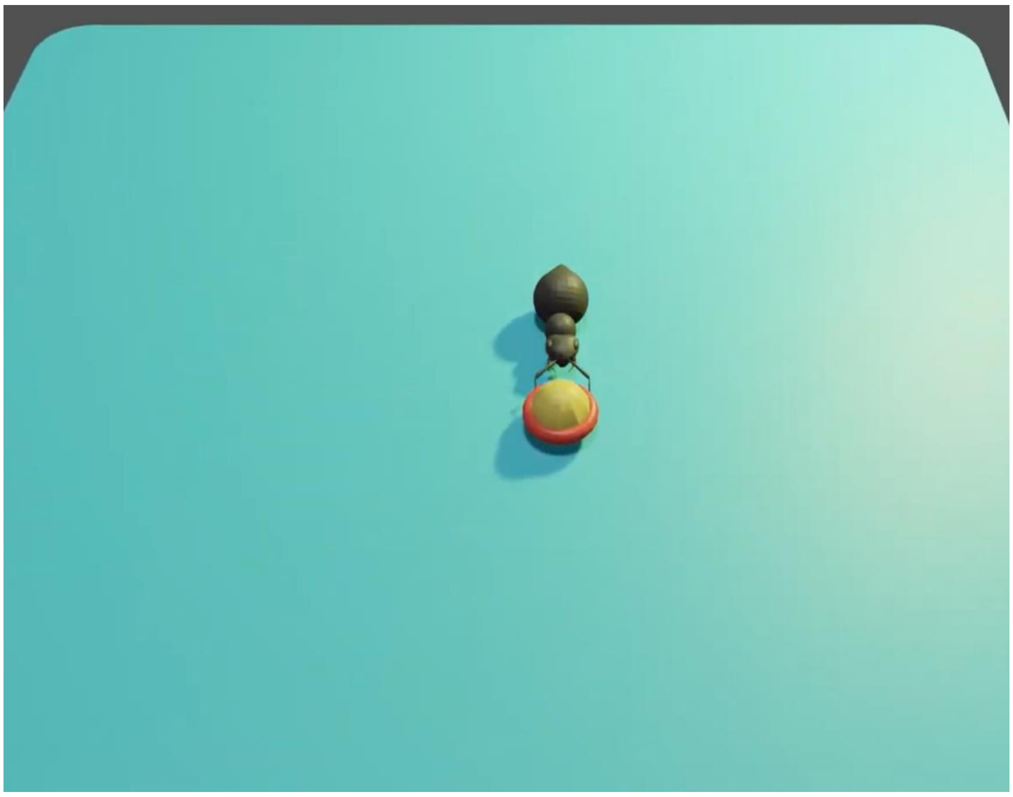
8.2 10th Generation Ant



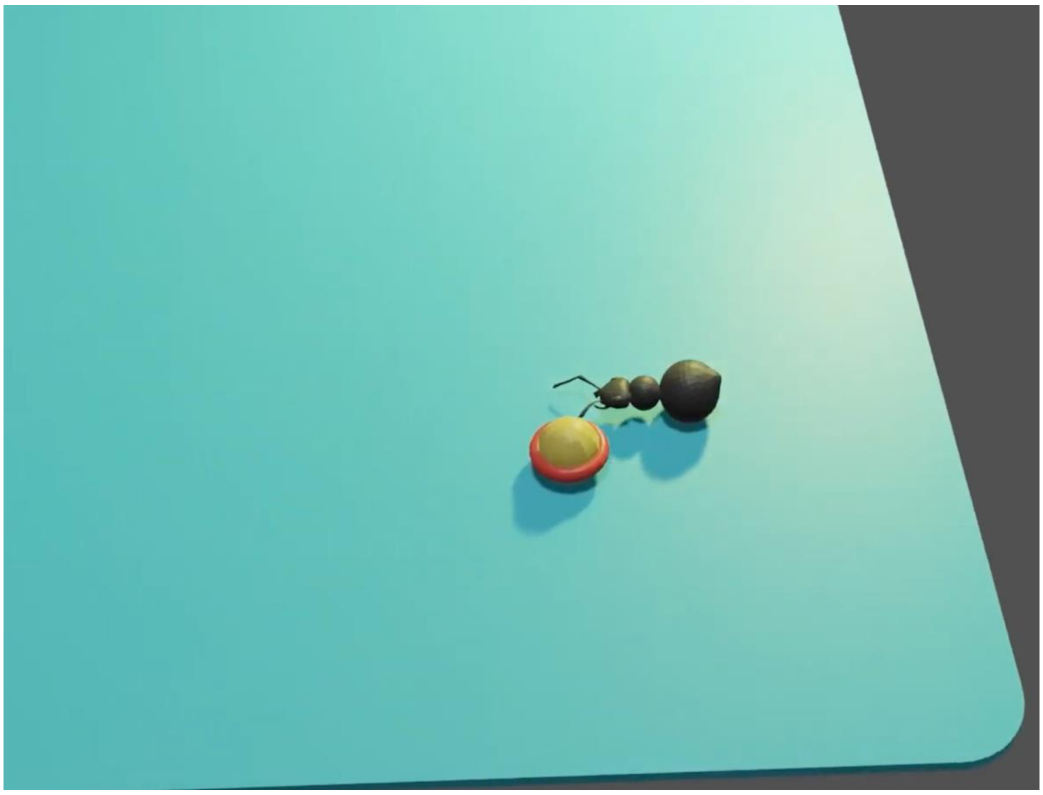
8.3. 50th Generation Ant



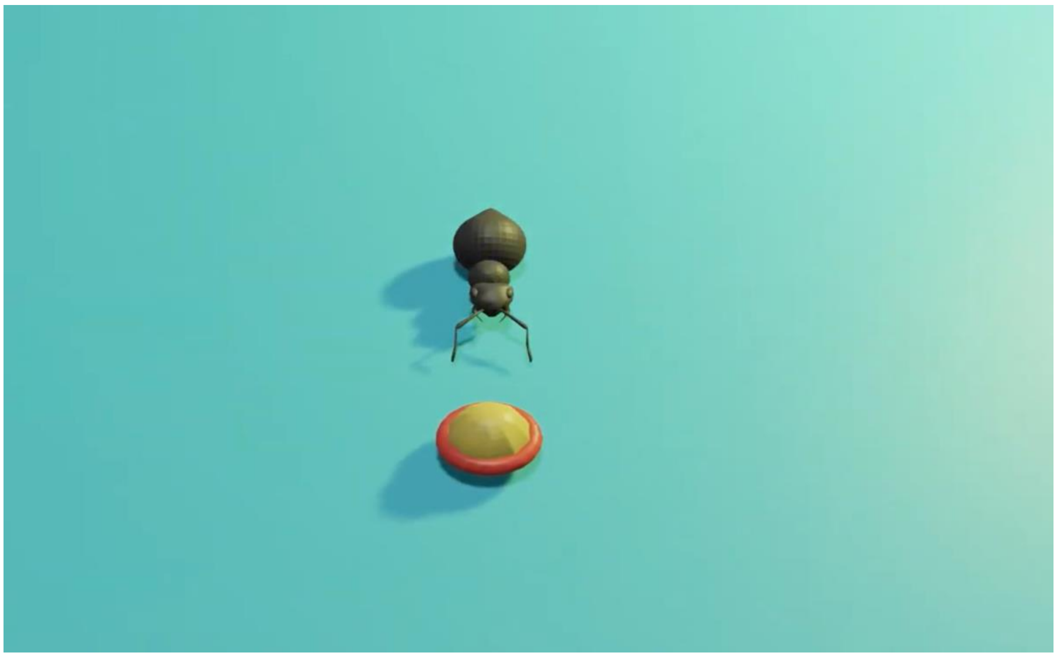
8.4. 100th Generation Ant



8.5 500th Generation Ant



8.6. 1190th Generation Ant



References

[1] Mantas Paulinas, Andrius Ušinskas, "A Survey Of Genetic Algorithms Applications For Image Enhancement and Segmentation", Electronic Systems Department, Faculty of Electronics, Vilnius Gediminas Technical University.

[2] Csáji, B.C., 2001. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, *24*(48), p.7.

[3] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B. and Mordatch, I., 2019. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*.

[4] Omar M. Sallabi and Younis EI-Haddad, "An Improved Genetic Algorithm to Solve the Traveling Salesman Problem", World Academy of Science, Engineering and Technology Volume 3, 2009.

[5] Carroll, S. P. & Fox, C. W., eds. *Conservation Biology: Evolution in Action.* New York, NY: Oxford University Press, 2008.

[6] Darwin, C. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. London, England: John Murray, 1859.

[7] Futuyma, D. J. *Evolutionary Biology*, 3rd ed. Sunderland, MA: Sinauer Associates, 1998.

[8] Gillespie, J. H. *Population Genetics: A Concise Guide*, 2nd ed. Baltimore, MD: The Johns Hopkins University Press, 2004.

[9] Haldane, J. B. S. A mathematical theory of natural and artificial selection, Part

I. *Transactions of the Cambridge Philosophical Society* **23**, 19–41 (1924).

[10] Hedrick, P. W. *Genetics of Populations,* 3rd ed. Sudbury, MA: Sinauer & Associates, 2005.

[11] Wright, S. *Evolution: Selected Papers*. Chicago, IL: University of Chicago Press, 1986.