VISUAL DEMONSTRATION OF GENETIC ALGORITHM

A design project in the domain of genetic algorithms for the course of Artificial Intelligence (IC3022)

UNDERTAKEN BY

ANAYA PAWAR

(T. Y. Q 5, G. R. 1710263)

ANEESH PODUVAL

(T. Y. Q 6, G. R. 1710045)

SARTHAK CHUDGAR

(T. Y. Q 16, G. R. 1710202)

JOHNATHAN FERNANDES

(T. Y. Q 37, G. R. 1710168)

UNDER THE GUIDANCE OF **PROF. KAPIL MUNDADA** & **PROF. (DR.) KULKARNI JAYANT V.**

TABLE OF CONTENTS

PREFACE

A **Genetic Algorithm** is a high-level search technique inspired by the process of **natural selection**. It is used to generate solutions through the implementation of biological processes such as **mutation**, **crossover**, and **natural selection**.

Originally based on **Darwin's theory of evolution**, the genetic algorithm has been extensively researched and is used to this very day in various applications such as search engines and logistic problem-solving.

The aim of this course project is to simulate a genetic algorithm in an easy to visualize program, in order to further understand the concept, given minimal prior experience in the field.

PROBLEM STATEMENT

For this simulation, we consider the case of several **flies** that want to navigate a room to reach an **apple**. We aim to develop a system to demonstrate how these flies, through genetic algorithm processes, evolve to avoid the obstacles and reach the apple in an **efficient** manner.

In our simulation, (pictured in Figure 1) we represent the flies as black dots, the obstacles as red rectangles, and the goal as a green circle.

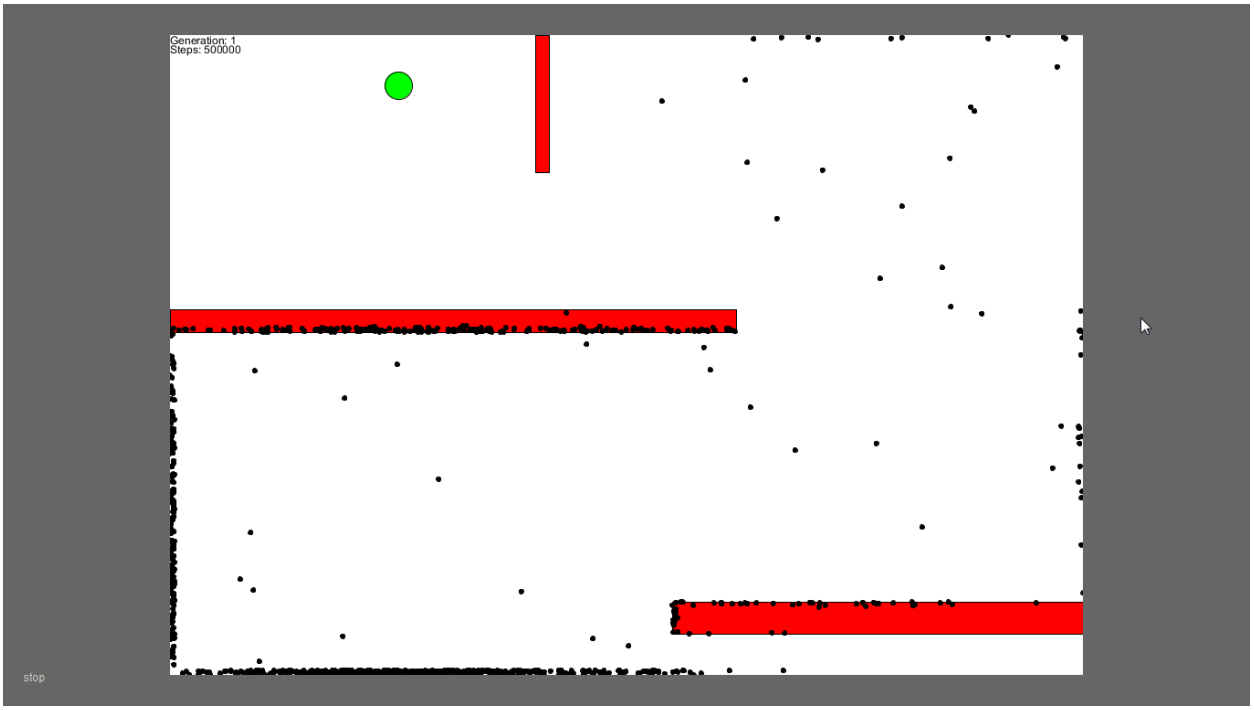

Figure 1: The simulation field

PROGRAMMING TOOLS

For this project, we have used the **Processing language**, which is based off of Java, in the **Processing compiler**, based off of the Arduino compiler. Processing maintains a focus on easy graphical applications while also being able to solve complex problems like programming languages, allowing us to focus on **algorithm implementation and graphic design** side by side.

ALGORITHM

We first initialize the flies, the obstacles, and the goal (apple).Each fly has **chromosomes** (represented as arrays of integers) formed of multiple **genes** (each element of that array). The length of this array dictates the amount of steps a fly can take before dying of starvation. The fly follows the directions in its gene until it runs out of steps, touches an obstacle, or reaches the goal.

Once all the flies have stopped, the program calculates the **fitness** of each fly. This is calculated by a **fitness function**, defined as:

$$fitness = (Distance\ to\ goal)^{-2}$$

Equation 1: Fitness function 1

Here, fitness is inversely proportional to the straight line distance between the fly and the goal. We take the square of the distance so that a small step towards the goal makes a big difference to the fitness of the individual.

If the fly is already at the goal, then fitness is calculated as:

$$fitness = 10^5 \times (Steps\ taken)^{-1}$$

Equation 2: Fitness function 2

If the fly has reached the goal, then the fitness is simply a function of how many steps they took to reach the goal. This becomes the new minimum number of steps for future generations and is the key to **improving efficiency**.

After the fitness of each fly has been evaluated, we have to choose the **parent** for the next **generation** of flies via the **roulette wheel process**. To implement this, we take the running sum of all the fitness values, and then choose a fitness value (and its associated fly) at **random**. Due to the nature of this selection process, flies with **a higher fitness** are more likely to be selected.

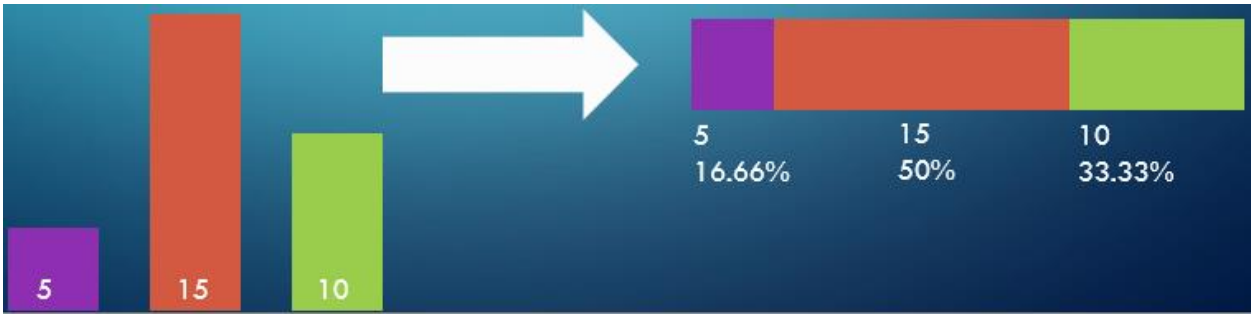

Figure 2: Roulette wheel selection process

After finding the "best" fly, we implement a system for it to "**Reproduce**". We place this fly directly into the next generation with no changes to its instructions. This is to ensure that in case the whole population mutates **negatively**, the species as a whole does not end up **devolving**. The other flies in the generation receive the same set of instructions, but we implement a **mutation** function using a random number generator to change a few directions in each of the new flies' instruction set. Our "best" fly is highlighted **blue** in forthcoming generations.

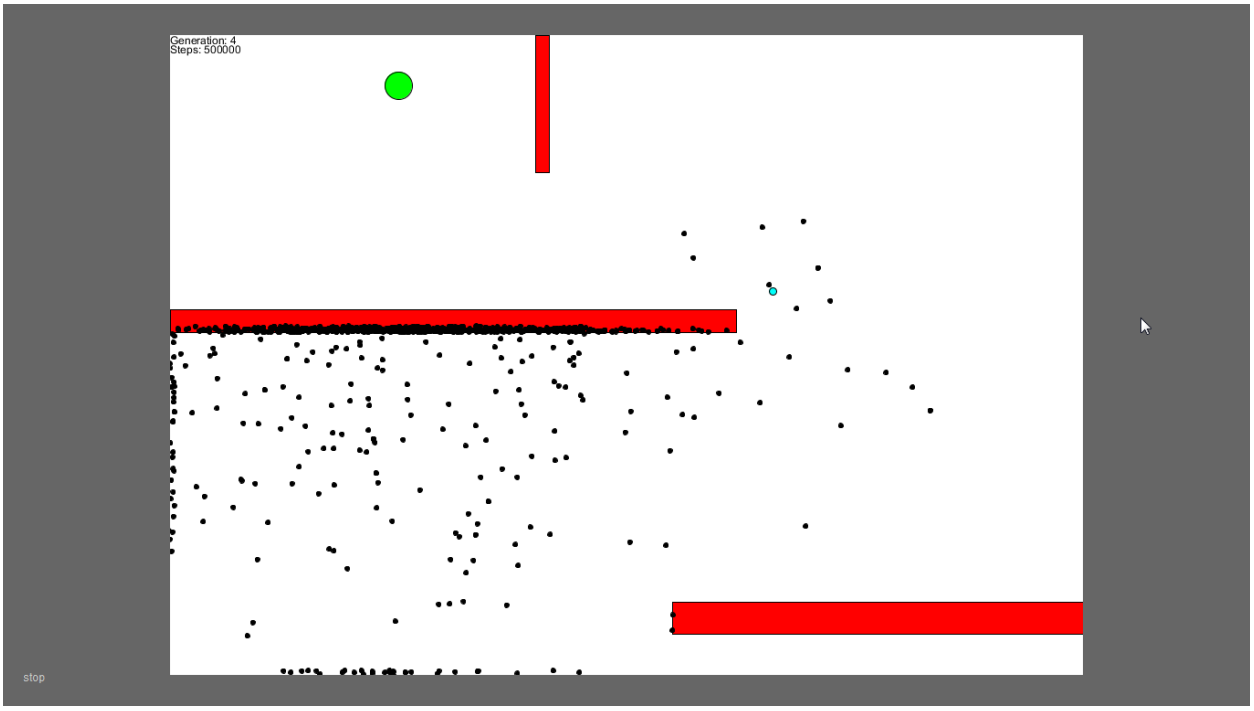

Figure 3: Establishing baseline performance

Given the scope of this project we only consider a **single parent** and hence there do not implement a gene **crossover** feature, which is where we would splice together the instruction set of two parents to create a child with traits from both.

OBSERVATIONS

When first started, the flies all travel in **random** directions, most of them colliding with the obstacles almost instantly. The flies that manage to survive the longest are usually the ones that get the farthest.

After a few generations (average 3.5), we notice the flies tend to "**swarm**" together since they all have a common path with only a few mutations.

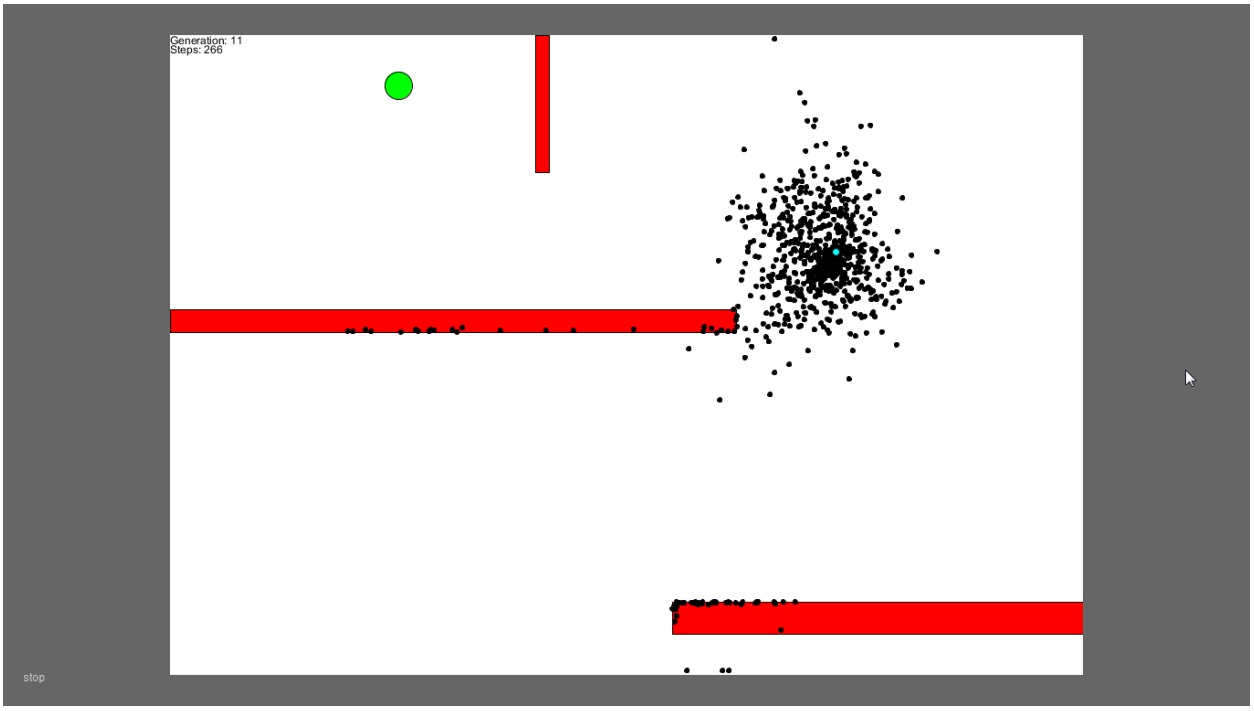

Figure 4: Flies "swarming" together after a few generations

Once the flies reach the goal, it becomes a **competition** to reach the goal in the **least number of steps**.

Given the scale of this program, the model tends to reach its most **efficient within a few generations**.

On average, the flies reach the goal within 7 generations, and after 13 generations, reach peak efficiency.

FUTURE SCOPE

While the field of Artificial Intelligence continues to grow, so do the applications of each algorithm developed. The Genetic Algorithm in particular can be applied to further **optimize** search engines and recommendation systems, while also being applied to solve computer science problems such as the infamous travelling salesperson problem.

CONCLUSION

Developing this program helped us understand the **underlying nuances** of working with the genetic algorithm, and how much work goes into developing robust **toolboxes** which accomplish the same task in a matter of minutes. This **visual approach** also assisted in

educating our fellow colleagues about the genetic algorithm during our **in-class seminar** as well.

ACKNOWLEDGEMENT

This project would not have been possible without the ever present guidance from both professors who were always ready to assist us with a constant stream of ideas and suggestions.

BIBLIOGRAPHY

Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert; Francone, Frank (1998). Genetic Programming – An Introduction. San Francisco, CA: Morgan Kaufmann. ISBN 978-1558605107.

Goldberg, David (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley Professional. ISBN 978-0201157673.

Khemani , Deepak - A First Course in Artificial Intelligence-McGraw-Hill Education (2014)

Koza, John (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press. ISBN 978-0262111706.

Michalewicz, Zbigniew (1996). Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag. ISBN 978-3540606765.

Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press. ISBN 9780585030944.

APPENDIX

FIGURES

EQUATIONS

CODE

```
//Artificial Intelligence course poject - Genetic Algorithm


class Brain //Brain class to store instructions
{
  PVector[] directions; //Array of vectors to store directions. Same values used for acceleration
  int step=0; //Count number of steps taken
  Brain(int size) //Initialize instruction set
  {
    directions=new PVector[size]; //Initialize instructions array with number of directions = size of population
    randomize(); //Insert random instructions into array
  }
  void randomize() //randomize function definition
  {
    for (int i=0; i<directions.length; i++)
    {
      float Angle=random( 2*PI); //Generate random angle from 0 to 360
      directions[i]=PVector.fromAngle(Angle); //Assign angle value to instructions array
    }
  }
  Brain clone() //Clone brain for next generation
  {
    Brain clone=new Brain(directions.length); //Create new brain with same number of instructions
    for (int i=0; i<directions.length; i++)
```

```
  {
    clone.directions[i]=directions[i].copy(); //Copy instructions from current brain to new brain
  }
  return clone; //Return new cloned brain
 }
 void mutate() //Mutation function to randomly change instructions
 {
  float rate=0.01; //Mutation rate
  for (int i=0; i<directions.length; i++)
  {
   float rand=random(1); //Generates random floating point number from 0 to 1
   if (rand<rate) //If random number is less than 0.01
   {
    float Angle=random( 2*PI); //Generate random angle
    directions[i]=PVector.fromAngle(Angle); //Insert into instruction set
   }
  }
 }
}
class Fly //Class for fly properties
{
 PVector pos; //Position Vector
 PVector vel; //Velocity Vector
 PVector acc; //Accelleration Vector
```

```
Brain b; //Instruction set
boolean isdead=false; //Flag to keep track of whether fly is dead or alive
float fitness=0; //Keep track of fitness value
boolean atgoal=false; //Flag to check if fly has reached goal
boolean isbest=false; //Flag to check if fly is best in generation
Fly() //Initiate flies at bottom-middle of screen, with zero initial velocity and acceleration
{
  b=new Brain(1000); //Generate 1000 instruction sets
  pos=new PVector(250, 620);  //Initial position at 250, 620 i.e. bottom-middle of screen
  vel=new PVector(0, 0);  //Initial velocity set to 0
  acc=new PVector(0, 0); //Initial acceleration set to 0
}
void show() //displays flies on screen
{
  if (isbest) //If curreny fly is best out of previous generation
  {
    fill(0, 255, 250); //blue color for best fly
    ellipse(pos.x, pos.y, 8, 8); //draw big ellipse at position
  } else //For all other normal flies
  {
    fill(0); //black color
    ellipse(pos.x, pos.y, 5, 5); //draw ellipse at position
  }
}
```

```
void move() //Function to calculate motion
{
  if (b.directions.length>b.step) //If further instructions exist
  {
    acc=b.directions[b.step]; //Set acceleration to value of instruction array stored at step index
    b.step++; //Increment number of steps taken
  } else //If flies run out of instructions
  {
    isdead=true; //Fly dies if they run out of valid instructions (starvation)
  }
   //Simulating forces being applied to flies
  vel.add(acc); //Adding generated acceleration to velocity
  vel.limit(7); //Limit max speed to 7 pixels per frame
  pos.add(vel); //Change position by velocity amount
}
void update() //Function to check if after moving, fly should die or not
{
  if (!isdead && !atgoal) //If fly is not dead or at goal
  {
    move(); //Call function to calculate new position
     //Calculating if fly intersects obstacles
    if (pos.x<5||pos.x>width-5||pos.y<5||pos.y>height-5)
    {
      isdead=true;
```

```
    } else if (pos.x>0 && pos.x<620 && pos.y>300 && pos.y<325)
    {
      isdead=true;
    } else if (pos.x>550 && pos.x<1000 && pos.y>620 && pos.y<655)
    {
      isdead=true;
    } else if (pos.x>400 && pos.x<415 && pos.y>0 && pos.y<150)
    {
      isdead=true;
    } else if (dist(pos.x, pos.y, goal.x, goal.y)<12.5) //Checking if fly has reached goal
    {
      atgoal=true;
    }
  }
 }
 void calcfit() //Function to calculate fitness value
 {
  if (atgoal) //If fly reached goal
  {
    fitness=0.25+10000/(b.step*b.step); //high fitness inversely proportional to square of steps
taken
  } else //If fly dies due to obstacle or starvation
  {
    float goaldist=dist(pos.x, pos.y, goal.x, goal.y); //Calculate Euclidean distance to goa
```

```
    fitness=1/(goaldist*goaldist); //fitness inversely proportional to square of distance from goal
   }
  }
  Fly getchild() //Function to create child from parent fly
  {
   Fly child=new Fly(); //New fly object
   child.b=b.clone(); //Child has same instructions as parent
   return child; //Return child object
  }
}
class Population //Class to maintain entire population
{
  Fly[] f; //Array of fly objects
  int gen=1; //Count of generations
  float fitsum; //Running suum of fitness valuse
  int best=0; //Keep track of least number of steps
  int min=500000; //USed to find minimum
  Population(int size) //Initialize population of flies
  {
   f=new Fly[size]; //Inistialize array of fly objects
   for (int i=0; i<size; i++)
   {
    f[i]=new Fly(); //Generate fly object for each element in array
   }
```

```
}
void show() //Updates display with new location of fly
{
  for (int i=1; i<f.length; i++)
  {
    f[i].show();
  }
  f[0].show(); //Parent (best from previous generation) is shown separately due to color syntax
}
void update() //UPdate location of fly
{
  for (int i=0; i<f.length; i++)
  {
    if (f[i].b.step>min) //Check if fly is taking more steps than current best
    {
      f[i].isdead=true;
    } else
    {
      f[i].update();
    }
  }
}
void calcfit() //Calculate fitness for fly
{
```

```
  for (int i=0; i<f.length; i++)
  {
    f[i].calcfit();
  }
}
boolean extinct() //Flag to check if all flies are dead or at goal
{
  for (int i=0; i<f.length; i++)
  {
    if (!f[i].isdead && !f[i].atgoal)
    {
      return false;
    }
  }
  return true;
}
void calcfitsum() //Calculates fitness using running sum to simulate roulette wheel
{
  fitsum=0;
  for (int i=0; i<f.length; i++)
  {
    fitsum+=f[i].fitness;
  }
}
```

```
 Fly selectparent() //Randomly choose fly from generation using roulette wheel principle
 {
  float rand=random(fitsum); //Take random value from running sum of fitnesses to act as threshold
  float runningsum=0; //Initialize running sum as 0
  for (int i=0; i<f.length; i++)
  {
   runningsum+=f[i].fitness; //Add fitness of flies to running sum
   if (runningsum>rand) //If running sum is greater than fitness threshold
   {
    return f[i]; //Chose fly as parent
   }
  }
  println("RETURNING NULL");
  return null;
 }
 void mutate() //Mutation function
 {
  for (int i=1; i<f.length; i++)
  {
   f[i].b.mutate();
  }
 }
 void getbest() //Function to find best performing fly i.e. highest fitness
```

```
 {
  float max=0;
  int index=0;
  for (int i=0; i<f.length; i++) //Loop to find maximum fitness value
  {
   if (f[i].fitness>max)
   {
    max=f[i].fitness;
    index=i;
   }
  }
  best=index;
  if (f[best].atgoal)
  {
   min=f[best].b.step; //If fly has reached goal, set number of steps taken as maximum number of steps for all other flies to improve efficiency
  }
 }
 void naturalselect() //Combine previous functions to select a fly as parent for next generation
 {
  Fly[] newf=new Fly[f.length]; //Generate new fly object for parent
  getbest(); //Get fly with best fitness from previous generation
  calcfitsum(); //Calculate fitness sum of flies
  newf[0]=f[best].getchild(); //Place best fly with highest fitness directly into new generation without mutation
```

```
   newf[0].isbest=true; //Declare best fly of previous generation as best for current generation
as reference line
   for (int i=1; i<newf.length; i++) //For each fly in new generation
   {
     Fly parent=selectparent(); //Randomly select parent
     newf[i]=parent.getchild(); //Generate instructions for child
   }
   f=newf.clone(); //Create opy of new fly object
   gen++; //Increment number of generations
  }
}
Population Flies; //Declare population of flies
PVector goal=new PVector(250, 55); //Define goal location vector
void setup() //Setup function
{
  frameRate(60); //Set frame rate for display, ie number of screen updates per second
  size(1000, 700); //Set windows size to 100 x 700
  Flies=new Population(1000); //Initialize population of flies to 1000 flies
}
void draw() //Graphics function
{
  background(255); //White background
  fill(0); //Black color for text
  text("Generation: "+Flies.gen, 0, 10); //Display current generation
```

```
 text("Steps: "+Flies.min, 0, 20); //Display current best step count
  //Draw rectangles for obstacles
 rectMode(CORNERS);
 fill(255, 0, 0); //Red color obstacles
 rect(0, 300, 620, 325);
 rect(550, 620, 1000, 655);
 rect(400, 0, 415, 150);
 fill(0, 255, 0); //Green color for goal
 ellipse(goal.x, goal.y, 30, 30); //Draw goal at specified position


 if (Flies.extinct()) //If entire population is dead
 {
   Flies.calcfit(); //Call function to calculate fitness
   Flies.naturalselect(); //Call function to select best fly
   Flies.mutate(); //Call function to mutate flies
 } else //If flies are still alive
 {
   Flies.update(); //Update flies with instructions
   Flies.show(); //Display flies in new positions
 }
}
```