Guess the Picture

By: Max DeMelo – 100937368

Comp 4106 – Final project

# Problem Domain

Guess the picture has a very simple concept and applies AI techniques in order to complete the task. The task set out by this program is given an original image, attempt to recreate it using a base population of either random pixels in the same dimensions of the picture or similar pictures predetermined by the user. It takes the images in the population mixes them all up and tries to see how accurate it can get using what it has.

# Motivation for the problem

The motivation for this problem is less what it does, more so how it does it. I wanted to do a problem that used a genetic algorithm in order to complete its task but I also wanted to see the progression of the algorithm as it made new generations of pictures. I came across an article that referred to artificial creativity which was defined as teaching a computer to paint (or make a picture) by teaching it what style you wanted it to follow. This was exactly what I was looking for as it allowed me to use genetic algorithms as well as see the progression generation by generation seeing the pictures it saves evolve and adapt into what I was trying to make it look like.

Once I had what I was doing I need to determine how it was going to do it. Initially I thought it would be fun to give the computer the final image and give it a population of random pixels to start. I thought that with the diversity of the random pixels it would be able to converge onto the final picture. But I also had the idea to start trying to guess the image based on images that would be classified in the same regard. For example, if I wanted the program to guess a picture of a face I would give it other faces, possibly one similar to the one in the final image.

With both of these ideas in place I decided to see which base population would give the best final outcome.

# AI Techniques used

The AI techniques used in this project were Genetic algorithms that try to implement artificial creativity and classification. However, the classification of the project is done by hand due to the fact that it is out of the scope of the project. The selection process for deciding which parents (images) would be used for mating was a tournament system which took K potentials from the population and selected the strongest of them. That was done twice, one for each parent and those two-mated creating a new offspring.

# Design Choices

## Population Initialization

The pictures themselves are stored as BufferedImages, this allowed me to access every pixel with both getters and setters, get the width and height of the picture easily. I stored the initial picture as its own separate image due to the fact that I referred to it so much to determine the strength of all pictures in the population.

The population of the program was stored in an array of BufferedImages which was the size of a predetermined variable called **POPULATION\_COUNT** which could be changed in order to use more or less population. The population was generated in one of two ways, first was making each image in the population from scratch using randomized colors for the pixels which were named Population\_i.jpg where the i was the number in the population that image was. The other way that the population can be generated is by filling the correct directory (\MiniCapStone\Pictures\population) with images that the user feels is the same classification of the image trying to be made by the program. These images must be named the same way as the auto generated images.

## New Generations

Once the population has been determined the first generation can get under way. A generation’s first step is to determine if it wants to mutate the runt of the litter (the image that is the least similar to the final image), this allows the program to keep diversity as well as make sure the program is heading in the right direction. The mutation chance currently is set to 1 in 100 chance of mutating and will be discussed later. Once it has done that it makes the rest of the new generation by determining who will mate. This pair of images will produce a new image which will be added to the new population. This pair of images is selected using a tournament selection in which K images are taken from the population and fitted against each other. The image that is the closest to the final image wins the right to move on and mate. This is done once more for the other parent, with a quick check to make sure it’s not the same parent. Once this is done enough to fill a new population it is saved overtop of the previous generation and the whole process can start a new with the new generation ready to see who will pass on their pixels.

## Mutations

Mutations are done in one of two spots. First spot that mutations can happen are in every new generation the runt has the chance to mutate the full image and pass it on to the new generation. This image is given to the Mutate class that takes the image and mutates it in currently one of three ways. It can either shift all the red, green or blue of the image by 50 in the positive direction looping back around after passing 255. This brings diversity back into the population possible adding the needed shades of certain colors to get the population closer to the final product.

The second place that mutations can happen is in every single pixel of all the children. Currently a random number is rolled between 1 and 100 which determines which parent it takes any particular pixel from but there is also another number rolled and if these two numbers end up being the same the pixel gets mutated. That means that any given pixel has a 1 in 10 000 chance of being mutated. Mutations in this sense mean getting re-randomized.

## Strength Calculation

The strength calculation compares the two images it is given and gives a percentage as to how close to each other they are. This calculation has two steps, first of which is determining how similar the color of both pixels is:

Followed by:

With this calculation, the higher the % the stronger that picture was in the population.

## Mating

When the two parents are picked for a new child image the process of mating goes through each pixel in the new image and at each pixel there are two main stages. The first stage is generating two random numbers, the first being the mating value which ultimately determines which parent it will get a particular pixel from and the second being its mutating value. If these two values are the same the pixel will mutate, if they aren’t if the number is under half of the mutation threshold then I choose the pixel at the same position of the first parent and vice versa for over half of the mutation threshold.

# Possible Enhancements

Some potential enhancements to the code would be:

* UI that would make it user friendly and make it so that you don’t have to program every new iteration of the program
* Different mutations, both full mutations for runts and single pixel mutations
* More complex mating between two different images allowing them to prefer the stronger pixel or parent

# Appendix

As the program was described in depth during design choices this will show two examples that were generated using the project.

## Beaut.jpg

This simulation was done with a population count of 21, K was set to 5 and was done over 10 generations. The reason this was done with so few generations and population size is because with a 1920x1080 picture with these stats the simulation took 1h 30min. This was also done with pictures similar to this such as some with different faces or in different rooms of my house and a couple of my roommate.



Figure 1: Final image aiming for (sorry for the face)

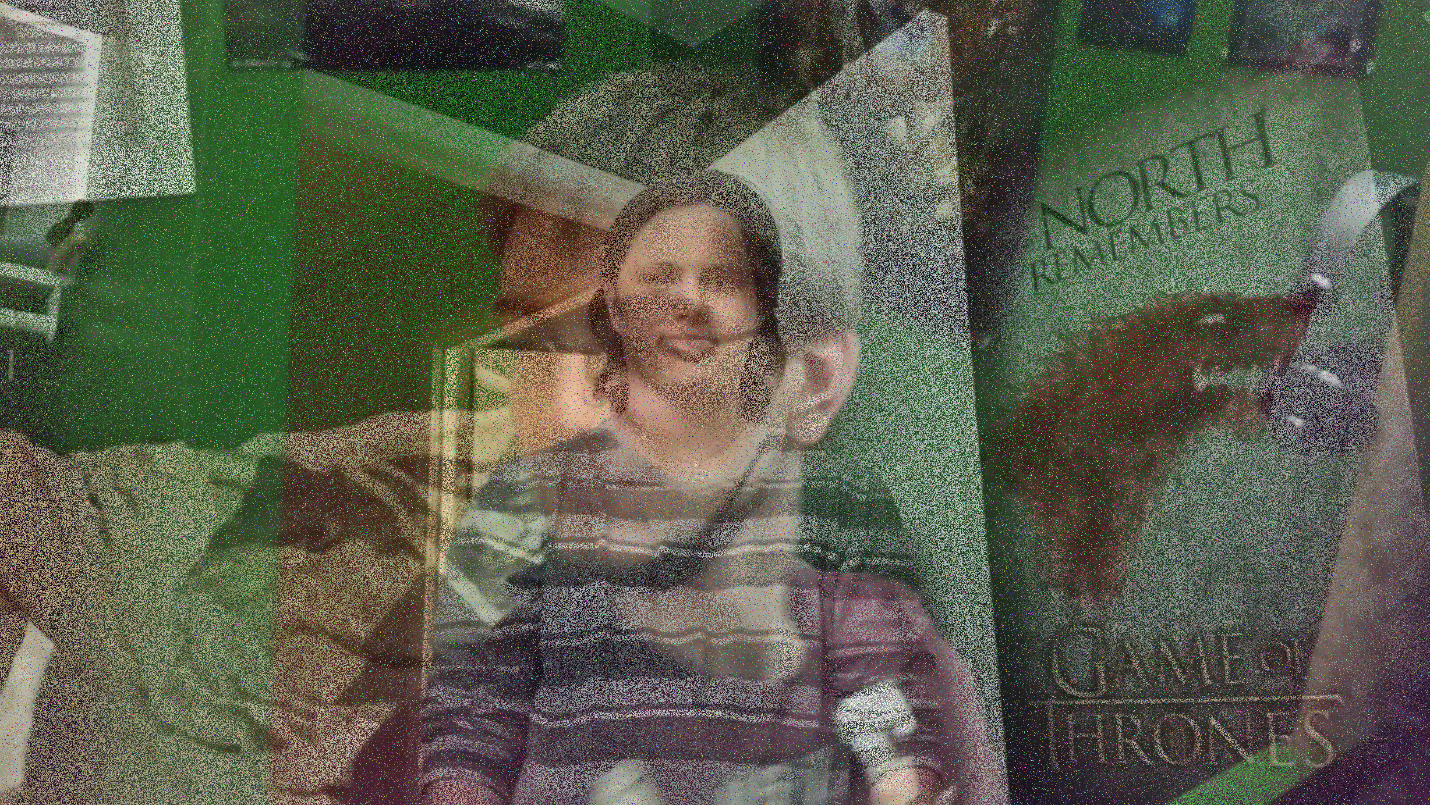


Figure 2:Final Result (predetermined population)

## Puppy.jpg

This simulation was done with a much bigger sample size with a population count of 100, with a K set to 5 done over 1000 generations. This photo is only about 160x160 which allowed me to do more population and generations. This simulation took about half an hour and was done with random pixels as the initial population.



Figure 3: Puppy that was the final image aiming for



Figure 4: Final Result (random Pixels)

## Which is better random or predetermined

Personally, I like the final result that came from the randomized pixels as it was more what I was aiming for when I set out to do artificial abstraction and I feel with some of the possible enhancements mentioned in chapter 5 it could lead to a far more accurate image.

# Code

## guessThePicture.java

**package** Main;  
  
**import** javax.imageio.ImageIO;  
**import** java.awt.\*;  
**import** java.awt.image.BufferedImage;  
**import** java.io.File;  
**import** java.io.IOException;  
**import** java.util.Random;  
  
**import static** java.awt.image.BufferedImage.***TYPE\_INT\_ARGB***;  
  
*/\*\*  
 \* Created by Max on 4/17/2017.  
 \*/***public class** guessThePicture {  
  
 **private final int POPULATION\_COUNT** = 100;  
 **private final int K** = 5; *//number picked during tournament selection* **private final int MUTATION\_THRESHOLD** = 100; *// 1 in a 100 to mutate* **private final** Random **random** = **new** Random();  
 **private final** Mutator **mutator** = **new** Mutator(**random**);  
  
 **private** String **directory**;  
 **private** BufferedImage **initialPicture**;  
 **private** BufferedImage[] **population**;  
 **private int height**, **width**;  
  
 **public** guessThePicture(**boolean** randomData, String photoName){  
 **directory** = System.*getProperty*(**"user.dir"**);  
 **directory** = **directory** + **"\\Pictures"**;  
 **try**{  
 **this**.**initialPicture** = ImageIO.*read*(**new** File(**directory** + **"\\intialPictures\\"** + photoName));  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }  
  
 **this**.**height** = **initialPicture**.getHeight();  
 **this**.**width** = **initialPicture**.getWidth();  
  
 **if**(randomData)  
 randomPopulationInit();  
 **else** choosenPopulationInit();  
  
 }  
  
 **private** BufferedImage correctSize(BufferedImage image){  
 **int** height = image.getHeight(), width = image.getWidth();  
 BufferedImage newImage = **new** BufferedImage(**this**.**width**, **this**.**height**, ***TYPE\_INT\_ARGB***);  
 **if**(height > **this**.**height**)  
 height = **this**.**height**;  
 **if**(width > **this**.**width**)  
 width = **this**.**width**;  
 **for**(**int** x = 0; x < width; x ++){  
 **for**(**int** y = 0; y < height; y ++){  
 newImage.setRGB(x, y, image.getRGB(x,y));  
 }  
 }  
 **return** newImage;  
 }  
  
 **public void** choosenPopulationInit(){  
 **population** = **new** BufferedImage[**POPULATION\_COUNT**];  
 BufferedImage buffer;  
 **for**(**int** i = 0; i < **POPULATION\_COUNT**; i++){  
 **try**{  
 buffer = ImageIO.*read*(**new** File(**directory** + **"\\population\\Population\_"** + (i + 1) + **".jpg"**));  
 **population**[i] = correctSize(buffer);  
 }**catch**(IOException e){  
 **break**;  
 }  
 }  
 savePopulation();  
 }  
  
 **public void** randomPopulationInit(){  
 Color color;  
 **population** = **new** BufferedImage[**POPULATION\_COUNT**];  
 **for**(**int** i = 0; i < **POPULATION\_COUNT**; i ++){  
 **population**[i] = **new** BufferedImage(**width**, **height**, ***TYPE\_INT\_ARGB***);  
 **for**(**int** x = 0; x < **width**; x ++){  
 **for**(**int** y = 0; y < **height**; y ++){  
 color = **new** Color(**random**.nextInt(256), **random**.nextInt(256), **random**.nextInt(256));  
 **population**[i].setRGB(x, y, color.getRGB());  
 }  
 }  
 }  
 savePopulation();  
 }  
  
 **private void** savePopulation() {  
 **for** (**int** i = 0; i < **POPULATION\_COUNT**; i++) {  
 **try** {  
 File outputfile = **new** File(**directory** + **"\\population\\Population\_"** + (i + 1) + **".jpg"**);  
 ImageIO.*write*(**population**[i], **"png"**, outputfile);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 **public** BufferedImage mate(BufferedImage father, BufferedImage mother){  
 BufferedImage child = **new** BufferedImage(**width**, **height**, ***TYPE\_INT\_ARGB***);  
  
 **for**(**int** y = 0; y < **height**; y ++){  
 **for**(**int** x = 0; x < **width**; x++){  
 **int** mating = **random**.nextInt(**MUTATION\_THRESHOLD**);  
 **int** mutate = **random**.nextInt(**MUTATION\_THRESHOLD**);  
 **if**(mutate == mating)  
 child.setRGB(x, y, **mutator**.mutantPixel());  
 **else if**(mating < **MUTATION\_THRESHOLD**/2)  
 child.setRGB(x, y, father.getRGB(x,y));  
 **else** child.setRGB(x, y, mother.getRGB(x,y));  
 }  
 }  
  
 **return** child;  
 }  
  
 **public void** newGenerations(**int** numberOfGenerations){  
 **for**(**int** i = 0; i < numberOfGenerations; i ++) {  
 newGeneration();  
 System.***out***.println(**"Gen: "** + (i+ 1));  
 }  
 savePopulation();  
 }  
  
 **public void** newGeneration(){  
 BufferedImage[] newPopulation = **new** BufferedImage[**POPULATION\_COUNT**];  
  
 **int** start;  
 **int** superMutation = **random**.nextInt(**MUTATION\_THRESHOLD**) + 1;  
 *//runt mutated into new population* **if**(superMutation > (**MUTATION\_THRESHOLD** - 1)){  
 start = 1;  
 **int** runtIndex = findRunt();  
 newPopulation[0] = **mutator**.mutate(**population**[runtIndex]);  
 }*//no full mutations* **else** start = 0;  
  
 **for**(**int** i = start; i < **POPULATION\_COUNT**; i ++){  
 **int** father = tournamentSelection();  
 newPopulation[i] = mate(**population**[father], **population**[tournamentSelection(father)]);  
 }  
  
 **population** = newPopulation;  
 }  
  
 **private int** findRunt(){  
 **int** runtIndex = 0;  
 **double** runtStrength = percentDifferencePicture(**population**[0]);  
 **for**(**int** i = 1; i < **POPULATION\_COUNT**; i ++){  
 **if**(runtStrength > percentDifferencePicture(**population**[i])){  
 runtIndex = i;  
 runtStrength = percentDifferencePicture(**population**[i]);  
 }  
 }  
 **return** runtIndex;  
 }  
  
 **private int** tournamentSelection(**int** firstParent){  
 **int**[] selected = **new int**[**K**];  
 **boolean** cantSelect;  
 **do** {  
 selected[0] = **random**.nextInt(**POPULATION\_COUNT**);  
 }**while**(selected[0] == firstParent);  
  
 **for**(**int** i = 0; i < selected.**length**; i++){  
 **do**{  
 cantSelect = **false**;  
 selected[i] = **random**.nextInt(**POPULATION\_COUNT**);  
 **for**(**int** k = 1; k < i; k ++){  
 **if**((selected[i] == selected[k]) && (selected[i] == firstParent)){  
 cantSelect = **true**;  
 **continue**;  
 }  
 }  
 }**while**(cantSelect);  
 }  
  
 **int** alphaIndex = 0;  
 **for**(**int** i = 1; i < selected.**length**; i ++){  
 **if**(percentDifferencePicture(**population**[selected[alphaIndex]]) < percentDifferencePicture(**population**[selected[i]]))  
 alphaIndex = i;  
 }  
  
 **return** selected[alphaIndex];  
 }  
  
 **private int** tournamentSelection(){  
 **int**[] selected = **new int**[**K**];  
 **boolean** cantSelect;  
 selected[0] = **random**.nextInt(**POPULATION\_COUNT**);  
 **for**(**int** i = 0; i < selected.**length**; i++){  
 **do**{  
 cantSelect = **false**;  
 selected[i] = **random**.nextInt(**POPULATION\_COUNT**);  
 **for**(**int** k = 1; k < i; k ++){  
 **if**(selected[i] == selected[k]) {  
 cantSelect = **true**;  
 **continue**;  
 }  
 }  
 }**while**(cantSelect);  
 }  
  
 **int** alphaIndex = 0;  
 **for**(**int** i = 1; i < selected.**length**; i ++){  
 **if**(percentDifferencePicture(**population**[selected[alphaIndex]]) < percentDifferencePicture(**population**[selected[i]]))  
 alphaIndex = i;  
 }  
  
 **return** selected[alphaIndex];  
 }  
  
 **public double** percentDifferencePicture(BufferedImage picture1){  
 **return** percentDifferencePicture(picture1, **initialPicture**);  
 }  
  
 **public double** percentDifferencePicture(BufferedImage picture1, BufferedImage picture2){  
 **double** total = 0;  
 **for**(**int** y = 0; y < **height**; y ++){  
 **for**(**int** x = 0; x < **width**; x ++){  
 total += percentDifferencePixel(x, y, picture1, picture2);  
 }  
 }  
 **return** total / (**height**\***width**);  
 }  
  
 **private double** percentDifferencePixel(**int** x, **int** y, BufferedImage picture1, BufferedImage picture2){  
 Color c1 = **new** Color(picture1.getRGB(x,y)), c2 = **new** Color (picture2.getRGB(x,y));  
 **double** pctDiffRed = ((Math.*abs*((**double**)c1.getRed() - (**double**)c2.getRed())) / 255);  
 **double** pctDiffGreen = (Math.*abs*((**double**)c1.getGreen() - (**double**)c2.getGreen())) / 255;  
 **double** pctDiffBlue = (Math.*abs*((**double**)c1.getBlue() - (**double**)c2.getBlue())) / 255;  
 **return** (100 - ((pctDiffRed + pctDiffBlue + pctDiffGreen) / 3 \* 100));  
 }  
  
 **public int** getRGB(**int** x, **int** y){  
 **return initialPicture**.getRGB(x,y);  
 }  
  
 **public int** getPOPULATION\_COUNT(){  
 **return POPULATION\_COUNT**;  
 }  
  
 **public** BufferedImage getInitialPicture(){  
 **return initialPicture**;  
 }  
  
 **public** BufferedImage getPopulation(**int** index){  
 **return population**[index];  
 }  
  
 **public int** getHeight() {  
 **return height**;  
 }  
  
 **public int** getWidth(){  
 **return width**;  
 }  
}

## Mutator.java

**package** Main;  
  
**import** java.awt.\*;  
**import** java.awt.image.BufferedImage;  
**import** java.util.Random;  
  
*/\*\*  
 \* Created by Max on 4/19/2017.  
 \*/***public class** Mutator {  
  
 Random **random**;  
  
 **public** Mutator(Random random){  
 **this**.**random** = random;  
 }  
  
 **public** BufferedImage mutate(BufferedImage victim){  
 mutations mutation = mutations.*values*()[**random**.nextInt(mutations.*values*().**length**)];  
 **switch**(mutation){  
 **case *redShift***:  
 victim = shiftColor(0, victim);  
 **break**;  
 **case *blueShift***:  
 victim = shiftColor(1, victim);  
 **break**;  
 **case *greenShift***:  
 victim = shiftColor(2, victim);  
 **break**;  
 }  
 **return** victim;  
 }  
  
 **private int** shift50(**int** currValue){  
 **if**((currValue + 50) <= 255)  
 **return** currValue + 50;  
 **else  
 return**((currValue + 50) - 255);  
 }  
  
 **public** BufferedImage shiftColor(**int** shift, BufferedImage victim){  
 Color currColor, newColor;  
 **for**(**int** x = 0; x < victim.getWidth(); x ++){  
 **for**(**int** y = 0; y < victim.getHeight(); y ++){  
 currColor = **new** Color(victim.getRGB(x,y));  
 **switch**(shift){  
 **case** 0:  
 **int** newRed = shift50(currColor.getRed());  
 newColor = **new** Color(newRed, currColor.getGreen(), currColor.getBlue());  
 victim.setRGB(x,y,newColor.getRGB());  
 **break**;  
 **case** 1:  
 **int** newGreen = shift50(currColor.getGreen());  
 newColor = **new** Color(currColor.getRed(), newGreen, currColor.getBlue());  
 victim.setRGB(x,y,newColor.getRGB());  
 **break**;  
 **case** 2:  
 **int** newBlue = shift50(currColor.getBlue());  
 newColor = **new** Color(currColor.getRed(), currColor.getGreen(), newBlue);  
 victim.setRGB(x,y,newColor.getRGB());  
 **break**;  
 }  
 }  
 }  
 **return** victim;  
 }  
  
 **public int** mutantPixel(){  
 Color color = **new** Color(**random**.nextInt(256), **random**.nextInt(256), **random**.nextInt(256));  
 **return** color.getRGB();  
 }  
  
 **public enum** mutations{  
 ***redShift***,  
 ***blueShift***,  
 ***greenShift*** }  
}