UNIVERSITY of WASHINGTON

Computer Science & Software Engineering

Genetic Algorithm Applications in Artificial Creativity

Capstone Final Report & Documentation

Conrad Dudziak

Advised by Yusuf Pisan

8/23/2019

TABLE OF CONTENTS

Introduction

This capstone project followed the exploration of genetic algorithms and their applications to artificial creativity. As a final product, this project resulted in an algorithm capable of replicating images using polygons through evolutionary programming. To develop this software, three fundamental questions had to be answered:

1. What is a genetic algorithm?
2. How can a genetic algorithm be applied to artificial creativity?
3. How can a genetic algorithm be deployed?

This project was started with no knowledge in evolutionary programming or artificial creativity. Therefore, the first step in this capstone project was researching the literature of evolutionary programming. Darwinism, fitness functions, breeding strategies, mutation rates, and the principles of evolution became the core concepts by which the first algorithms in this project were constructed.

After researching and recreating some fundamental example projects such as string breeding, I began my research on artificial creativity. Google's Deep Dream is an existing example of artificial creativity that uses neural networks to create stylized images. There are two primary barriers in artificial creativity: Rule breaking and the evaluation of creativity. This capstone project explores the ways that a genetic algorithm can handle these barriers more successfully than a neural network such as Google's Deep Dream.

Next, in order to test the feasibility of a graphics based evolutionary program, I refactored the string breeding algorithm into a shape breeding algorithm. This algorithm converged a population of random shapes to a single target shape by color and vertex locations.

Finally, I extended the shape breeding algorithm into an image breeding algorithm, which uses shapes to construct stylized images. The report that follows includes information about how this algorithm operates, its proposed deployment to Microsoft Azure, an analysis on its outputs, and the consequences that fitness and breeding has on artificial creativity.

Background

What is a Genetic Algorithm?

The traditional genetic algorithm is the process of narrowing a problem space by evaluating the fitness of a guess. Therefore, an answer can be found quickly through a series of evolving guesses. For example, in a situation where an individual is tasked with guessing a number, the individual may have to guess an infinite amount of numbers before ever succeeding. However, if the individual were to receive 'hot' or 'cold' feedback after each of their guesses, they would have an easier time narrowing the possibilities in the problem set.

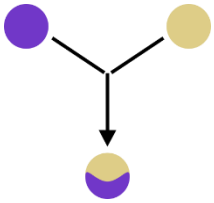
Some additional realms of genetic programming include:

- Interactive Selection
    - Interactive selection explores the application of genetic algorithms in the visual arts. User interaction can influence the way which a visual element evolves. An example of user interaction could be a user choosing their favorite images from a set of images. The computer would then create a new image with the target of producing an image that the user would like based on their prior selections.
- Ecosystem Simulation
    - The realm of ecosystem simulation explores the behaviors of pseudo-living beings, where objects on the screen learn to interact with their environment and each other. As a result, the objects mate and pass their genes on to a new generation. However, limiting factors constrain the population, allowing only the "fittest" individuals to reproduce.

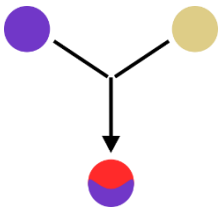
The Principles of Evolution

Darwinism discusses natural selection and evolution under the pretense of three core principles:

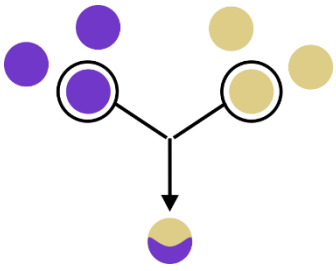
1. Heredity
    a. Proceeding generations must have a means of receiving the properties of their predecessors. The traits of one generation must be passed to the next generation, or else the population set will never converge to an evolved set. Without heredity, the DNA of the population set will only ever contain random noise.



2. Variation
    a. There must always be differences in the individuals within the population set. Individuals in the population set cannot be identical, or else reproduction will not result in new interesting sets of genes. Therefore, DNA must have the ability to mutate to prevent the population from converging to an incomplete answer.



3. Selection
    a. There must be a tool used by the population to reproduce. Parents must breed in order to produce the next generation. However, darwinian evolution states that the most "fit" of the population have the highest chance at reproducing. As a result, individuals with the most desirable genes are also the most likely to be selected for reproduction.



Requirements

This section lists the functionality of the image replicating software.

1. The image replicator must be accessible by a web browser on any device.
2. The image replicator must load with a default input image ready for computation.
3. The image replicator must be able to receive an image of any size as an input image.
4. Input images must be acceptable as types png, jpeg, or jpg.
5. The image replicator must have input fields for the genetic algorithms population size, polygon count, vertex count, mutation rate, and resolution scale factor.
6. The image replicator must perform computations on a smaller scaled version of the input image at a scale factor of the resolution scale factor.
7. The image replicator must display the average fitness of the entire population.
8. The image replicator must display the fitness value of the best fit image.
9. The best fit image must be displayed in a canvas of equal size of the input canvas.
10. The image replicator must implement a genetic algorithm that uses principles of heredity, variation, and selection under an object-oriented model.
11. The image replicator must use only sets of colored semi-transparent polygons to construct the target image.

Constraints

This section lists the restrictions placed on the completion of this project.

1. The image replicator must be completed and available by web URL by August 23, 2019.
2. The image replicator is hosted on a trial Azure account with $200 credit, rendering it invalid after the credits are consumed.
3. The image replicator is also hosted on github pages, but this version of the image replicator runs locally on the client's web browser, which results in runtimes constrained to the user's hardware.

Guidelines

The following guidelines are suggestions for when using or debugging the image replicator software.

1. The image replicator will always perform better when replicating target images that contain contrasting colors and sharp edges. Input images containing soft colors and no edges are likely to struggle converging and take more time.

2. Lower resolution images will always yield results in a faster runtime and in fewer generations than larger resolution images. Therefore, it is recommended not to use images larger than 1024x1024 pixels.
3. Any received inputs will not be included in a computation until the start button is clicked. If the algorithm is already running, then the algorithm must first be stopped with the stop button.
4. The resolution scale factor determines how precise the replicated image will be to the target image. However, a low resolution scale factor will harm performance as more internal computations are required to compare pixel values.
5. If an image is converging too early, the problem likely lies in the mutation rate. However, too high of a mutation rate can result in random noise which is incapable of converging.

Assumptions

The following assumptions are made under the constraints of this project.

1. The user is connecting to the image replicator software using a supported web-browser such as google chrome.
2. Users cannot use more Azure computational space than provided by the Azure trial.
3. Users connecting to the github pages version of the image replicator are not using a mobile device, as a mobile device will be very slow in comparison to a laptop or desktop machine.
4. Users always input non-negative and non-zero values into the input fields.
5. Users always upload an accepted image format file into the input field.

Software Design

Building a Generic Genetic Algorithm

The first step in constructing a genetic algorithm is working the three principles of evolution into an iterative pseudo code loop. Figure 1 shows the activity diagram of a generic genetic algorithm as it performs evolutionary programming with heredity, variation, and selection.

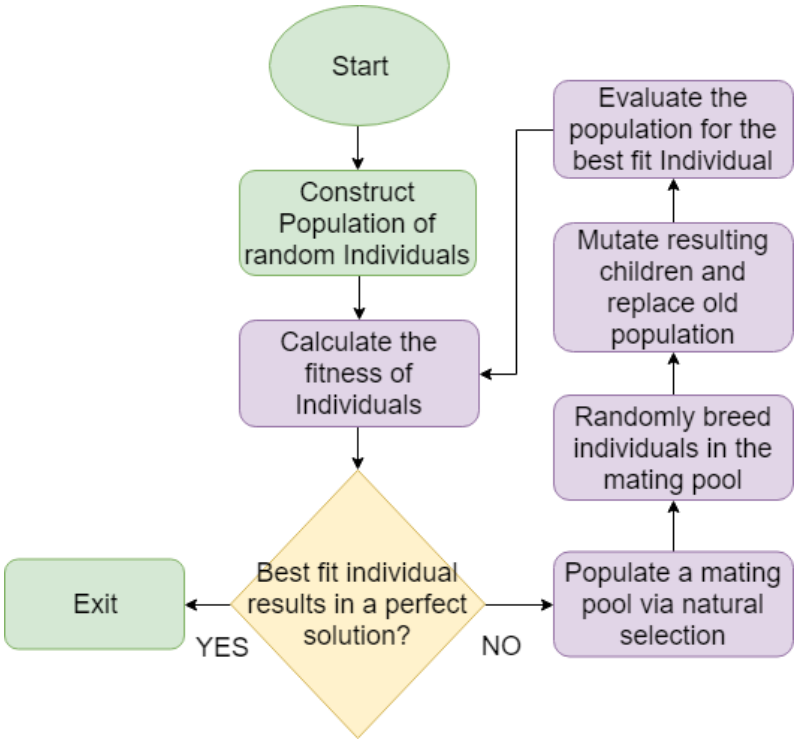
Heredity occurs during mating, when two individuals are bred together to produce an offspring. The resulting child consists of the genes of its parents, so that the current generation always influences and contributes to the DNA of the proceeding generation. In this way, traits are carried on between generations to provide the population with continuity.

Variation occurs when the resulting child is mutated. Mutation provides a population with the necessary change and variance needed to converge the population towards a target.

Otherwise, the population will always be limited by its DNA pallet, where missing DNA components for the solution can never be found because no parent possesses them.

Selection occurs when the mating pool is populated. The method by which the mating pool is populated is undescribed because of the generic nature of this genetic algorithm. However, it is likely to occur in some form of natural selection, where the most fit individual of the population has the most presence in the mating pool. Therefore, when randomly selected for breeding from the mating pool, the most fit individual has the largest percent chance of being chosen.

Figure 1: Activity Diagram of a Generic Genetic Algorithm



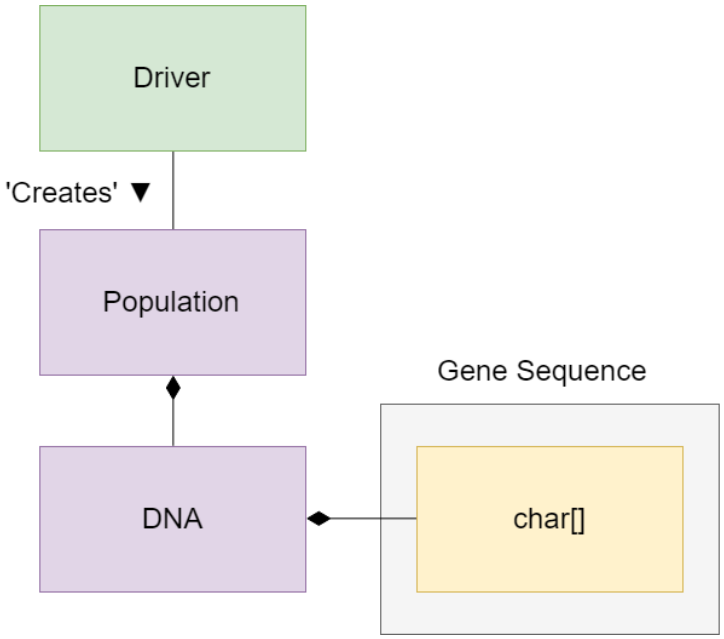
As a generic construction of a genetic algorithm, this iterative sequence can be applied to a wide variety of problems where details of DNA interactions are specified only in the data structure of the gene sequence.

Building a String Breeding Genetic Algorithm

As a first attempt at evolutionary programming, I followed textbook tutorials to create a genetic algorithm capable of converging a population set of random strings to a single target string. The algorithm was written in C# under an object-oriented model consisting of three classes: Driver, Population, and DNA.

Figure 2 shows a diagram describing the composition of the string breeding algorithm.

Figure 2: Composition of String Breeding Genetic Algorithm



Driver

The Driver class defines the target string, population size, and mutation rate that the population will experience. The driver then creates a Population object with these parameters and begins to loop through the evolutionary process for that population.

Population

The Population class behaves as a data structure containing all the individuals (DNA objects) of the population set. Upon construction, the population object populates itself with DNA objects. These DNA objects are randomly initialized. The population object also holds the implementation necessary for the selection process. The selection of breeders is done by weighting all the individuals in the population set against their contribution to a summated total fitness score of the population. An individual who contributed largely to the summated fitness score is more likely to be chosen for reproduction.

DNA

The DNA class behaves as a gene sequence. This class contains the actual character array (string) of each corresponding individual in the population set. Therefore, this class contains the implementation of the variation and heredity components of the evolutionary principles. DNA can crossover with a partner DNA by splicing character arrays at a random midpoint. The resulting DNA is returned to the population set as a new child. DNA can also mutate, which is the process of setting an element in the character array to a new random element.

Lastly, the DNA class is responsible for calculating a fitness score for each DNA object. The DNA object receives a target string and compares its character array values to each value in the target string. If the values match, the fitness score is incremented.

Github link to the string breeding algorithm:

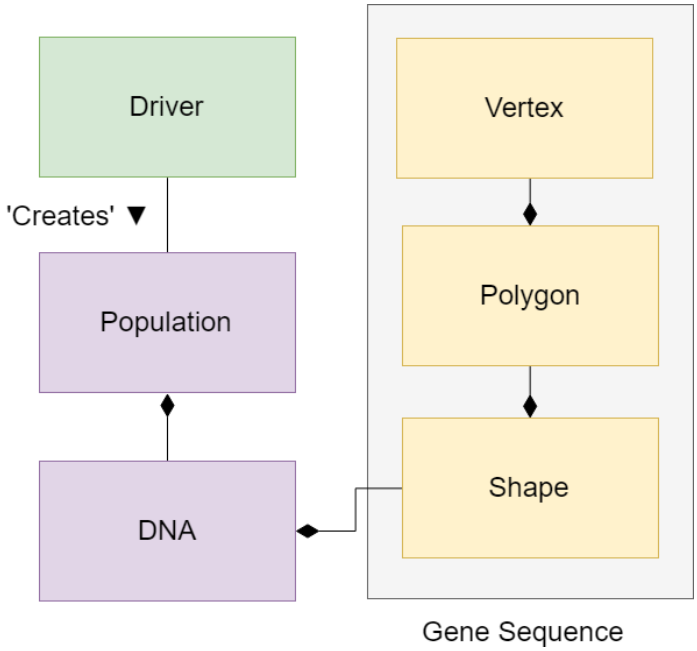
- https://github.com/ConradDudziak/Genetic-Algorithm-Converging-Strings

Building a Shape Breeding Genetic Algorithm

The next step towards constructing an image replicating software was adding graphics to a genetic algorithm. To accomplish this, I created an evolutionary program capable of receiving a target polygon and converging a population of random polygons to the target polygon.

Figure 3 shows a diagram describing the composition of the shape breeding genetic algorithm. The genes of the individuals consist of the polygon's vertices and color RGBA values.

Figure 3: Composition of Shape Breeding Genetic Algorithm



JavaScript was used to create this genetic algorithm with graphics, because of the fluidity of the web-based scripting language and the ability to quickly construct and test possible implementations. The P5.js library was used to draw shapes onto the screen with a framerate-based draw loop and a simple shape API.

The string breeding and shape breeding genetic algorithms were both implemented around the generic genetic algorithm, where changes only occurred in the gene sequence.

The steps involved in refactoring the previous edition of the genetic algorithm into the graphical version were as follows:

1. Created Polygon.js and Vertex.js. These classes are data structures used by the genetic algorithm to store information about shapes within the DNA objects genes.
    - Polygon.js consists of a vertex list and a red, green, blue, and alpha integer value. Polygon.js can also construct random polygons.
    - Vertex.js is an object used by the Polygon.js class to create a set of (x,y) coordinates. Each (x,y) coordinate is constructed as a vertex and added to the Polygons vertex list.
2. Ported DNA, Population, and Driver to JavaScript classes.
    - The overall structure and implementation of these classes is identical to the C# classes, other than syntax and language specific libraries.
3. Created a fitness function within the DNA.js class that scores each DNA (individual) based on its genes (polygon). The function receives a target polygon, which is then used for comparison against the DNA's genes (polygon). Both the RGBA and vertex list of the polygon are used to calculate the fitness score.
    - Color values of the target and current DNA are rescaled from 0-255 to 0-1. The scores of the color values are then subtracted to result in a difference, which is used to determine that specific colors contribution to the total score. The scores across all colors are then summated.
    - To calculate the score of the vertex list, the closest pairing vertices between the target and DNA are calculated with a distance function. The shortest distance between two vertices of the polygons is then rescaled based on the diagonal distance of the canvas. This results in a score for a single vertex pair based on the worst-case scenario of a possible vertex pair distance (where the vertices of the two polygons are on opposite corners of the canvas). Once all the vertices are paired and scored, the score is added to the color score and returned as the polygon's fitness.
4. Created a breeding algorithm that uses a random weighted sample of the population. DNA is randomly selected from the sample and mutated with a random rate, where the color and vertex values of the polygon is slightly adjusted to add some variation. The newly mutated DNA is then placed into a new population for the next generation.

Building an Image Replicating Genetic Algorithm

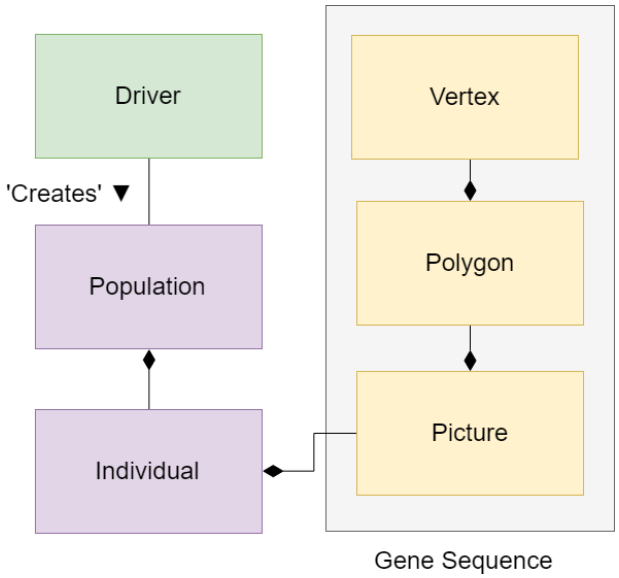
After integrating graphics into the genetic algorithm, there remained one problem: How can sets of polygons be compared to the data of an image? In order to make the final image replicating software, I needed a mechanism capable of comparing image data against polygon

data. P5.js is limited in the way that it can compare polygons. Polygons can only be compared by vertices and color values, which makes it unusable for a genetic algorithm that receives an image as a target rather than a polygon as a target. The solution to this problem was canvases and canvas contexts.

The Canvas 2D API offers graphical support with a html canvas element. A canvas element can draw images and shapes with data describing a set of points and colors. After an image or shape is drawn to the canvas element, the image data can be retrieved as an integer array. This returns the pixel values of the image in an array size width*height*4 (the four RGBA values for each pixel). As a result, this mechanism of drawing and cooking images into data arrays provided the software with the ability to easily convert polygon sets into images.

Figure 4 shows a diagram describing the composition of the image breeding genetic algorithm. The genes of the individuals are pictures which consist of a polygon set.

Figure 4: Composition of Image Breeding Genetic Algorithm



The steps involved in refactoring the previous genetic algorithm into the image constructing version were as follows:

1. Created Picture.js, driver.js, and index.html.
    - Picture.js is the gene data structure of the genetic algorithm. Picture.js is responsible for holding a set of randomly constructed polygons and drawing that set of polygons to a provided canvas.
    - Driver.js is the location of the booting and initialization of the genetic algorithm. This file includes the windows.onload(), and it is the file the contains all

configurations and iterating tools for the genetic algorithm. This file is also responsible for retrieving all the html elements from the webpage.

- Index.html consists of three canvases, an input image, and all the necessary scripts. The first canvas is the inputCanvas, which displays the input image. The second canvas is the outputCanvas, which displays the current most fit individual of the population. The dataCanvas is a hidden canvas which is used for calculations when cooking individuals (which consist of polygon sets) into images (arrays of image data).

2. Refactored Polygon.js, Vertex.js, DNA.js, and Population.js.
    - Polygon.js was simplified to receive only the single parameter of total vertices. Polygon.js also creates a random polygon based on RGBA values with range of 0 to 1. These values are then later scaled appropriately when applied to filling a drawn polygon. In the same way, Vertex.js was changed to operate on coordinate values with ranges from 0 to 1, which are later scaled to the appropriate width and height.
    - DNA.js was changed to be Individual.js. Individual.js serves the same purpose as DNA.js, but the genes of an Individual now consist of a Picture. The fitness of individuals is calculated with a mean squared error between the individual's genes image data array and the target image data array. Breeding between individuals first requires a process of populating a mating pool with a weighted sample. Individuals are then selected from that mating pool at random for breeding. For each child, one of the two parents are randomly selected to provide a polygon. The polygon is then mutated and provided to the child. Lastly, the child replaces an individual in the previous population set.
    - Population.js now receives a width, height, targetData array, polygonCount, vertexCount, and dataContext. These items are used to populate individuals with polygons, and to calculate the fitness of individuals. The width and height specify the width and height of the dataCanvas.
    - Setup a web-GUI version of the running genetic algorithm on github pages.
        - https://conraddudziak.github.io/ImageCreationWithGeneticAlgorithms/

Figure 5 shows the UML class diagram for the interaction between the driver component and the population object. The driver is responsible for collecting inputs and running the iterations of the genetic algorithm, which is exposed to the driver through the population object. The population object is constructed by the driver and called to iterate the generate method at set intervals.

Figure 6 shows the UML class diagram for the entire genetic algorithm, where the population is responsible for constructing Individuals. The genes of Individuals then consist of a Picture,

which in turn consist of Polygons and Vertices. The picture object contains the base level draw method, which handles the actual logic for drawing the polygon data onto a canvas parameter.

Figure 5: UML Class Diagram of Driver and Population Components

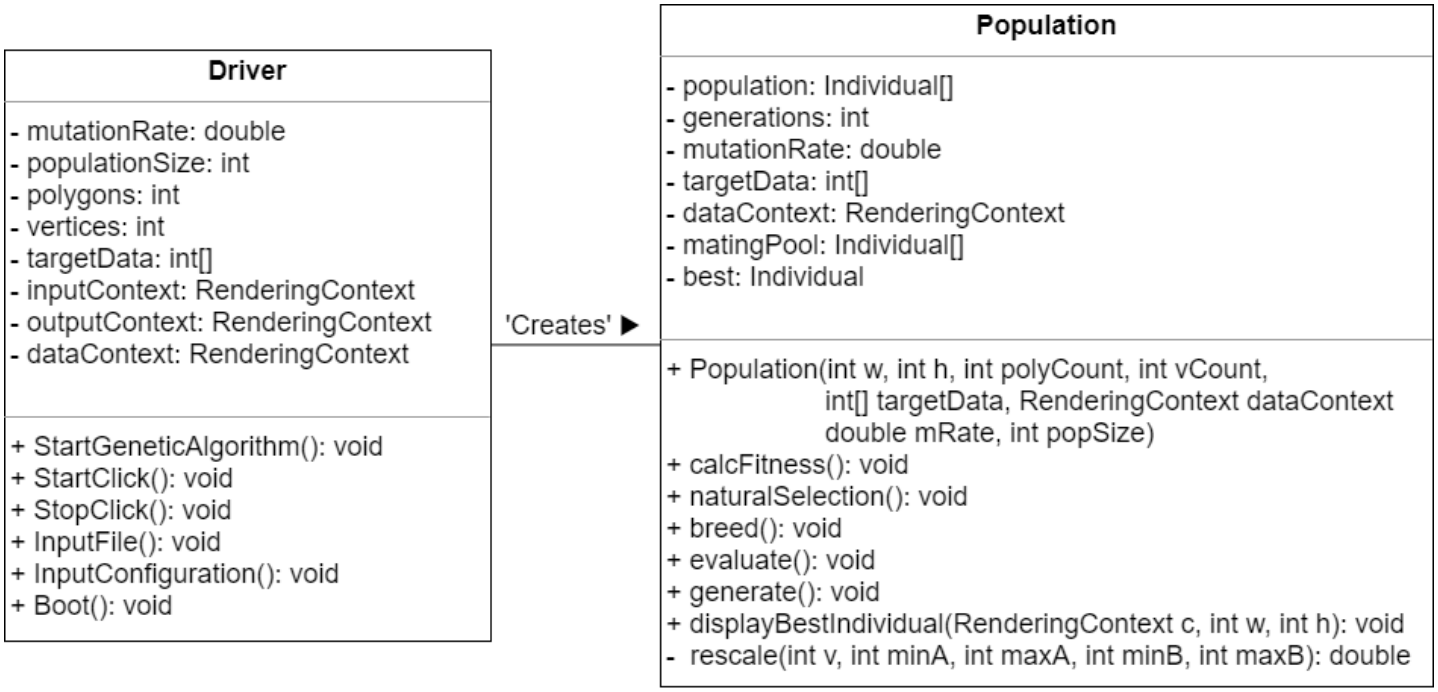
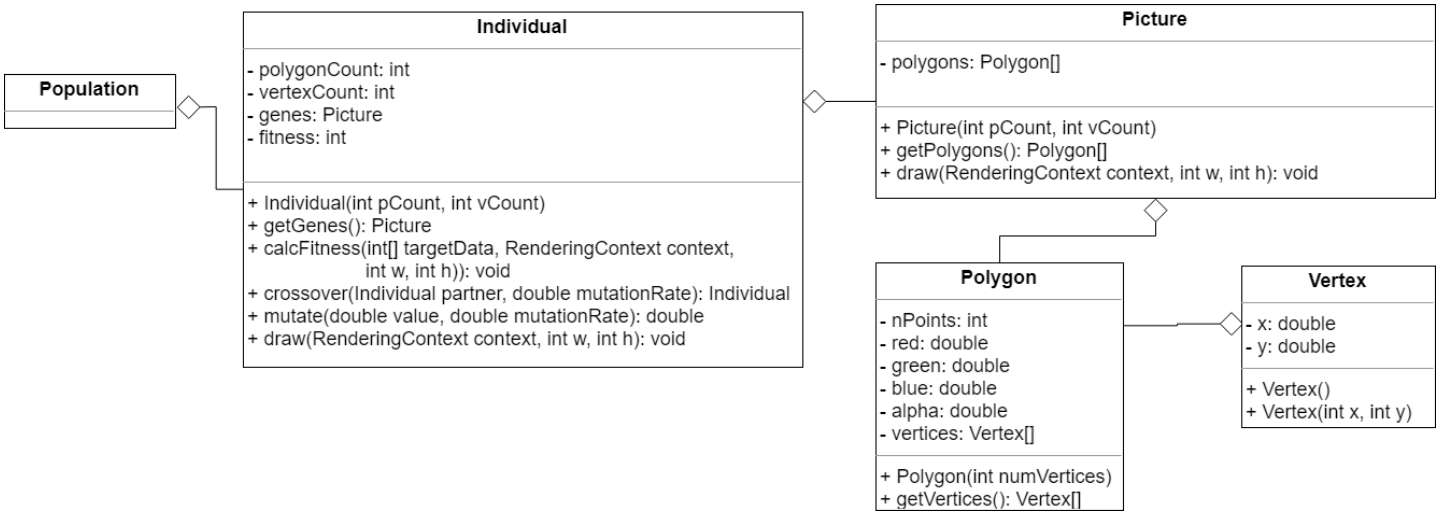


Figure 6: UML Class Diagram of Genetic Algorithm



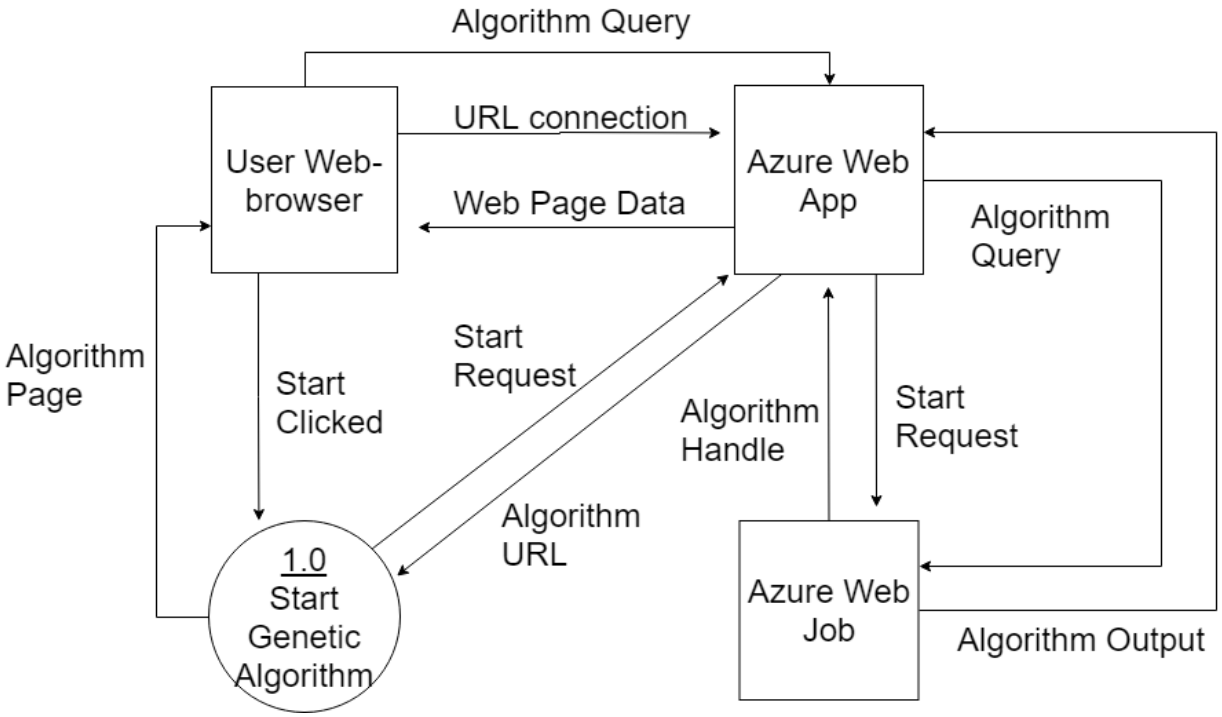
Deploying the Genetic Algorithm

There are many options and methods to deploy a web hosting service for user access. The two web hosting services used in this project were github pages and Microsoft Azure.

Github pages allows for rapid prototyping at no cost and is used to host a client-side version of the genetic algorithm. This means that the computational load of the genetic algorithm is placed onto the user's web-browser. In comparison, Microsoft Azure offers the ability for the computational load of the algorithm to occur on the server side.

Figure 7 shows a proposed dataflow diagram for the integration of Microsoft Azure into the genetic algorithm. Information is passed between three main entities in the diagram: the users web-browser, the azure web app, and an azure web job.

Figure 7: Dataflow Diagram of Proposed Azure Integration

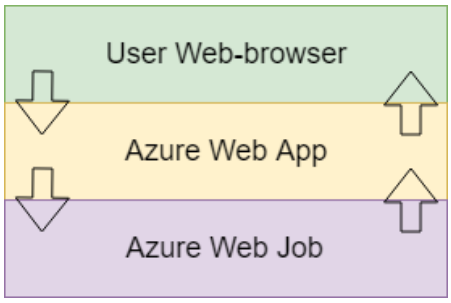


The users web-browser sends a connection request to the Azure Web App, which hosts the index.html page along with the supporting JavaScript items. Upon clicking the start button, an HTTP get request is sent to the Azure Web App to request an Azure Web job process ID. The web job then returns a handle to the Azure Web App which opens a new page in the user's web-browser.

The Azure web job then handles computations of the genetic algorithm and the user web browser can query for results. The query contains a process ID which the Azure web job uses to identify which process is computing the user's algorithm. A data package is then returned to the user's web-browser through the Azure Web App containing the best fit image and the accompanying genetic algorithm data.

Figure 8 shows a stack representing the system levels which information is passed to and from the client side.

Figure 8: System Dataflow Stack



For the final deliverable of this project, only the Azure Web App was completed as part of this system. Azure Web Job was not fully integrated.

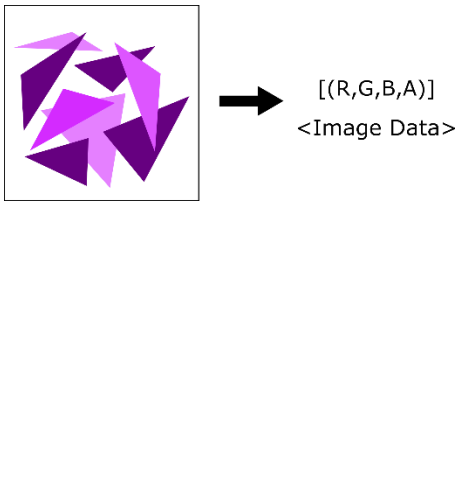
Results

The process of creating an image constructing genetic algorithm required iterative deliverables that slowly built up to a final product. Therefore, the strategies and functions implemented within the final algorithm are as much a result of this project as the actual data outputs collected in the projects analysis.

Fitting, Scoring, and Breeding

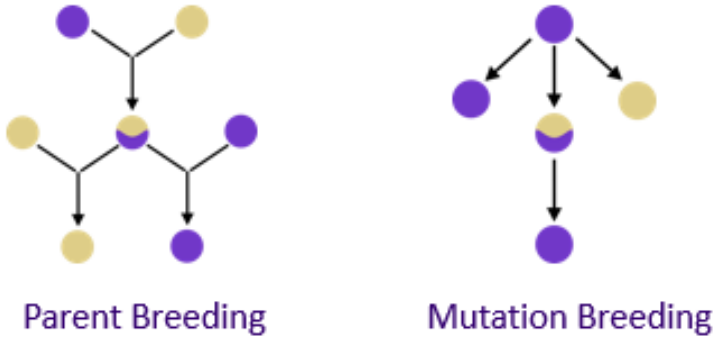
The final scoring method of this project uses JavaScript canvases to convert polygon sets into image data. Figure 9 shows an illustration with pseudo code of the scoring process.

Figure 9: Fitting and Scoring Pseudo Code Illustration



The other key result from the construction of this algorithm were the differences between breeding strategies. Specifically, this project explored the differences between parent breeding and mutation breeding. Figure 10 shows an illustration of the two breeding strategies, where both processes explore different methods of reproducing the next generation.

Figure 10: Illustration of Parent Breeding and Mutation Breeding



Parent breeding reproduces by using two or more individuals to construct a new individual, where each parent involved has a chance to pass DNA to the child. Mutation breeding is the process of a single Individual mutating into multiple individuals, which are then selected for survival based on their fitness. Table 1 shows a comparison of parent breeding and mutation breeding.

Table 1: Comparison of Parent Breeding and Mutation Breeding

Criteria	**Parent Breeding**	**Mutation Breeding**
Technique	Genes are passed to children which replace the population	Genes mutate to replace individuals' old genes
Replacement Method	Children always replace the previous generation	Resulting mutated genes replace old genes only if they are more fit
Population Size	>= 2	1
Mutation Restrictions	Parents must share similar gene structure (number of polygons)	No restrictions
Can Mutate	• Colors • Vertices	• Colors • Vertices • Total polygons • Total vertices

Mutation breeding provides the genetic algorithm with further control of the ways which the population set can be converged. Furthermore, mutation breeding gives the population the ability to start with minimal gene sequence data. This means that the population can begin with individuals only containing a single polygon, which helps improve early runtimes.

Figure 11 shows a comparison of the data collected with the algorithm using both parent breeding and mutation breeding.

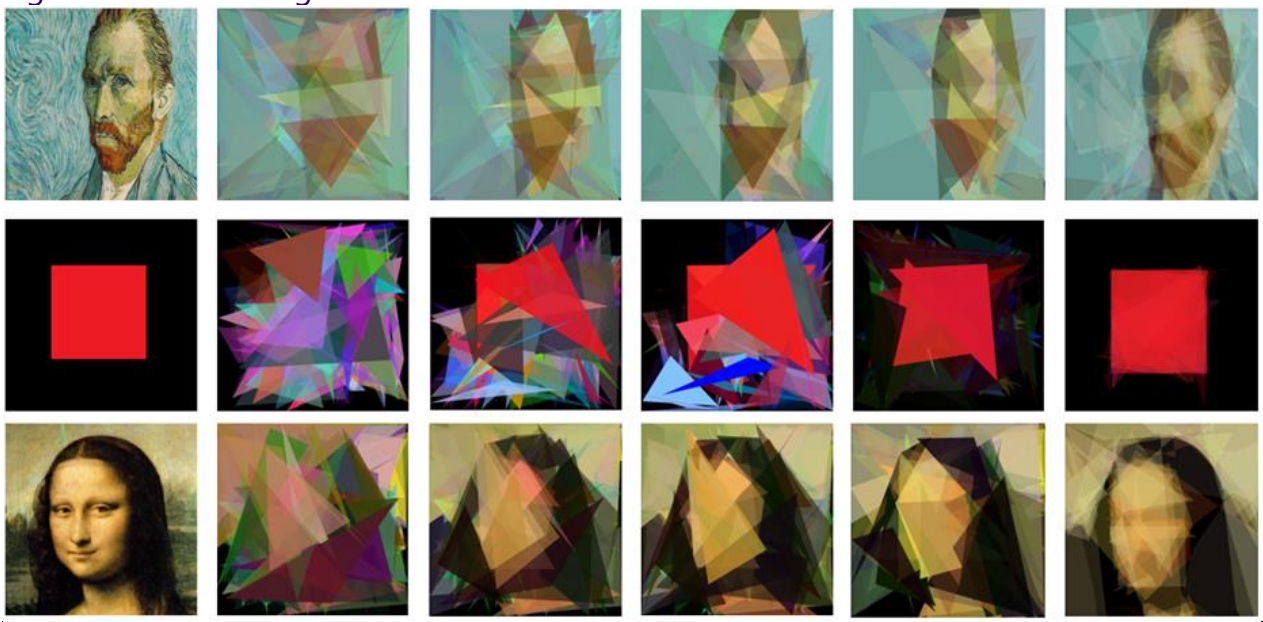
Figure 11: Comparison of Parent Breeding and Mutation Breeding

Mutation Breeding	Generations: 250	Generations: 800	Generations: 12000	Generations: 71000
	Time: 10 seconds	Time: 30 seconds	Time: 5 minutes	Time: 1 hour
	Best fit: 23%	Best fit: 52%	Best fit: 74%	Best fit: 87%

Parent Breeding	Generations: 110	Generations: 320	Generations: 4000	Generations: 22000
	Time: 10 seconds	Time: 30 seconds	Time: 5 minutes	Time: 1 hour
	Best fit: 8%	Best fit: 32%	Best fit: 51%	Best fit: 71%

Figure 12 shows more output results of the genetic algorithm ran with both parent breeding and mutation breeding.

Figure 12: Genetic Algorithm Test Case Results



Target Image	Generations: 250	Generations: 750	Generations: 3500	Generations: 7000	Generations: 50000
	Time: 10 seconds	Time: 1 minute	Time: 5 minutes	Time: 30 minutes	Time: 3 hours
	Avg fit: 51%	Avg fit: 68%	Avg fit: 78%	Avg fit: 83%	Avg fit: 89%
	Best fit: 58%	Best fit: 71%	Best fit: 79%	Best fit: 85%	Best fit: 90%

The use of parent breeding and mutation breeding is situational, where no single breeding strategy is best for every case. For example, the square image in figure 12 uses mutation breeding to subtract unneeded geometry, while the Mona Lisa image uses parent breeding because of its need for detail which a larger gene sequence can provide.

Conclusion & Lessons Learned

Artificial creativity is constrained by two fundamental barriers: Rule breaking and the evaluation of creativity. This project intended to solve these barriers using genetic algorithms. The following is a discussion on the pros and cons of genetic algorithms in artificial creativity.

Pros and Cons of Genetic Algorithms in Artificial Creativity

Google's Deep Dream is an artificially creative program that uses convolutional neural networks to find and enhance patterns in images through algorithmic pareidolia. This solution requires a black box algorithm that has greater freedom in rule breaking but is much harder to evaluate. A block box algorithm prevents the ability for a human to study the functions and strategies that exhibit the creative behavior.

In comparison, the image constructing genetic algorithm created in this research project is much easier to evaluate because of its well-defined fitness function and breeding strategies. As a result, the functions that are producing creative outputs can be observed and tweaked in order to help define creativity and what makes the algorithm creative.

Although the fitness function of a genetic algorithm makes the softwares creative behaviors easy to evaluate, it also inhibits the programs ability to break rules. This is because fitness functions and breeding strategies must be distinctly written by a human, which inevitably limits the creativity of the machine and its ability to handle problems.

Future Work

Future work on this project consists of improving the fitness function so that the algorithm will no longer require a target image for comparisons.

Integrating Image Recognition

In order to produce an algorithm that can create unique images without the need of an input reference image, a neural network is required. Instead of comparing pixel values, the program can pass image data arrays to an external image recognition software. The algorithm will then receive feedback based on what the software possibly recognizes in the polygon set. The recognized object will then be used to score the population and slowly converge the population towards the most recognized version of that object.