Genetic Programming Made Easy: Graphical Evolution

Research Project with Jan van Eijck

Olim Tuyt - 10814175

1 Introduction

Quoting David Goldberg, one of the founders of genetic algorithms, a genetic algorithm is a search algorithm “based on the mechanics of natural selection and natural genetics” [1]. Based on the evolution theory posed by Charles Darwin, they are randomized search algorithms simulating the survival of the fittest within a population of some data points, where the fittest are determined by a *fitness function*. Using the genetic principles of mutation and crossover, diversity is forced into the population. By tweaking the amount in which these techniques are used for a particular example, the often big data sets do not need to be fully fully searched. Therefore, a genetic algorithm can be a lot faster than a search of the full data set.

In general, a genetic algorithm starts with an initial population, consisting of points from some data set. These points we will call organisms from now on. The data set could consist of any objects you would like to optimize, for example bit strings of some length n. Now, following Goldberg, there are three phases to forming the next generation.

The first phase is called *reproduction*. In this phase, we pick organisms from the current population. This can be done in a variety of ways, as is explained further in a short treatise that Philip Michgelsen and I wrote on this subject [3]. However, most of the reproduction techniques are based on the fitness of the individual organisms, meaning the higher the fitness of an organism, the higher its chance of surviving the reproduction phase.

The second phase is the *crossover* phase. The phase is accompanied by a crossover parameter $p_c \in [0, 1]$, fixed at the start of the genetic algorithm. The operation of crossover takes two organisms and returns two new ones and is easiest to explain when considering bitstrings. When considering two bitstrings $x_1 \ldots x_n$ and $y_1 \ldots y_n$ of some length n, crossover picks some index $1 \leq i < n$ and exchanges the ends of the bitstrings after this index i. The result would therefore be strings $x_1 \ldots x_i y_{i+1} \ldots y_n$ and $y_1 \ldots y_i x_{i+1} \ldots x_n$. Sadly, a lot of data representations do not lend themselves directly for such an operation of crossover. However, the important principle here is that information is being exchanged between two organisms and this information does not get lost. It means that the ‘good parts’ of an organism can be merged with the good parts of another one, thus creating an even better organism. Taking this principle into account usually gives some form of crossover, even though the direct approach does not work directly. The crossover parameter p_c is the fraction of the organisms from the reproduction phase that will actually undergo crossover.

The third and final phase is *mutation*. This operation is applied to all organisms that came out of the first two phases. Also this phase is accompanied by a mutation parameter $p_m \in [0, 1]$, which is fixed at the start of the algorithm. Again, it is easist explained when looking at bitstrings. The mutation operation applied to some bitstring $x_1 \ldots x_n$ goes through the entire string. At every bit x_i, it flips a biased coin with probability p_m of getting heads. If it lands heads, the bit is flipped. If not, we do not flip and continue to the next bit. As with crossover, not every data representation lends itself directly for this operation. However, the important principle to take away from this operation is the fact that at specific places in the organism, information can be changed, without changing the structure of the organism itself. The information that was there before is gone forever.

This small report will cover my algorithm made for the project Genetic Programming Made Easy. During the project, we have devised our own genetic algorithm and played around with a lot of examples. The algorithm that I will explain in the following section tries to approximate a given image using certain shapes.

2 Graphical genetic algorithm

Whenever you search for genetic algorithms on the internet, you will find a lot of fancy, graphically pleasing representations of genetic algorithms of a variety of sorts. Designing a car crossing a circuit, making a stick figure walk or replicating a certain picture. The replication of pictures drew my attention. The algorithm gets a picture as input and tries to replicate that image as closely as possible using certain shapes. The shapes can be cirles, squares, triangles or any polygon you can come up with and every generation these shapes can be moved or coloured to resemble the original picture even more.

In the following subsections, I will explain my version of such an image replicating algorithm implemented in Haskell. I used the package JuicyPixels to read and write images within Haskell, as this is not possible in Haskells Prelude. This package can get the information from each individual pixel and recognizes various different image types. The main type I cover in my algorithm is the 8-bit grayscale image without an alpha channel. This is the easiest type to look at as it only contains one numerical value, namely the grayscale. The organisms we will use will be a list of shapes together with the pixel properties for that shape. For example, in case of a circle in an 8-bit grayscale image, the organism will consist of a list of 4-tuples that contain the x and y coordinate of the top left of the cirlce, the radius of the circle and the (8-bit) grayscale of the circle. This organism can be converted into an image quite easily by just drawing these circles onto a white canvas. These images created from an organism this way will be the basis of the fitness function.

2.1 Fitness function

In the algorithm I wrote, we will want the fitness to be as close to 0 as possible, where a fitness of 0 will express a perfect replica of the picture. The fitness function is easiest to define when we deal with a pixel type with only a single numerical value attached. We look at the 8-bit grayscale image. The fitness function takes an organism and as explained above, converts it into an image by drawing all shapes on an empty canvas (of the same size as the original image). Now we look at each coordinate (x,y) in the image and let $p_{x,y}$ and $p'_{x,y}$ be the pixel in the original image and the organism respectively. We define the fitness for this pixel as the absolute difference of their grayscale, denoted as $|p_{x,y} - p'_{x,y}|$. Then we sum over all the coordinates and this gives the fitness of the organism. More mathematically written:

$$f(\text{organism}) = \sum_{x,y} |p_{x,y} - p'_{x,y}|.$$

There are a few remarks to be placed here.
Firstly, this fitness function is quite easy to define as we are dealing with pixels that only

have a single component, namely the grayscale. For colour images, where each pixel has three components (RGB), this fitness function wouldn't work. There are easy ways around this, by for example summing up the differences in the R, G and B values. However, I do not know how well this will work, as I concentrated on getting the grayscale case working. But this is something that could be explored further.
Secondly, we might want to strengthen parts of this function with what Goldberg calls fitness scaling ([1], p.76). As we are dealing with relatively large organisms (64×64 pixels already gives over 4000 values to add up), individual pixel differences might cancel each other out. Trying to avoid these cancellations and to increase the difference in good and bad individuals and thus accelerate convergence, we might want to enlarge the fitness for bad individuals even further. There are generally two ways to do this. One of them is to enlarge the overall fitness of the organism (f_1 below), the other is to enlarge the fitness of each individual pixel (f_2 below). One could use a whole range of functions for this, but raising to a certain power or exponentiation are often the most straightforward.

$$f_1(\text{organism}) = (\sum_{x,y} |p_{x,y} - p'_{x,y}|)^k$$
$$f_2(\text{organism}) = \sum_{x,y} |p_{x,y} - p'_{x,y}|^k$$

In my algorithm, I have chosen for f_2 with $k = 3$, but many experimentation can be done here to see what works well.
Thirdly,

2.2 Reproduction

I won't spend a lot of words on this, as most of the techniques are already explained in the treatise mentioned earlier. In my algorithm, I have gone for Roulette Wheel Reproduction, where fitness approaching 0 is the best. I have however modified a single thing, which is that in my algorithm, the best of a generation *always* survives. It has a probability of 1 of ending up in the next generation and does not even have to go through mutation. The reason for this is that I found that these images are very easily broken, in the sense that they might be on the right track of convergence, crossover once and are back to square one again. Letting the best organism survive guarantees that there is at least no relapses during the run of the algorithm.

2.3 Crossover

Crossover in this case can be done fairly simple, as the organisms are actually lists of tuples and thus it is fairly easy to perform crossover. One takes two organisms, picks an index of the list and performs crossover as if it was a bitstring with the tuples the individual bits. Note that as we are dealing with tuples, the shapes themselves will never be cut up, the cuts only happen in between shapes. The reason for this is Haskell related, because the different elements of the tuple might have different types and otherwise one would have to play around with type management. Getting rid of this restriction could add some diversity and as before, experimentation can be done here.

2.4 Mutation

The way I did mutation requires a bit of an explanation. If we are given a mutation parameter p_m, we mutate every element of every tuple in the organism with probability p_m. However, as we're not dealing with bit strings, mutate is a rather vague term. In the most broad definition, it could mean that any numerical value could be uniformly changed into any other relevant numerical value. Relevant here means that, for example, we would want the x coordinate to be changed into something that falls within the bound of the image. This would come in handy early on in the algorithm, as we would need to make some pretty big jumps in order to come close to the picture we are trying to recreate. However, once we have arrived in the proximity of that picture, we might not want to make such big jumps in terms of placement or colour of the shapes. The solution to this is to restrict the size of the jumps we take to something a bit smaller. For example, the x coordinate in a 64×64 image can only be increased or lowered by maximally 20 pixels at a time. This way, we can more accurately converge further once we come somewhat close to the image. The downfall is that it probably takes longer to get there in the first place. I have chosen for the latter option in my algorithm.

3 Implementation

As said before, the graphical genetic algorithm is implemented in Haskell and can be seen in the accompanied file. This section will cover the main functions in this file and how to use it.

I tried to make the implementation of this algorithm as general as possible. When you are dealing with the replication of images, there's a lot of variables to take into account. The extension of the image file (.png, .jpeg, .gif, etc.), the colour type of the image (black-and-white, grayscale, colour, etc.) and many more. Here I made the explicit choice to only work with .png files, because they were quite easy to handle within the JuicyPixels package that I used. This means that the image you'd want to replicate needs to be in this format and the result will be a .png file.

However, when looking at the different colour types, I didn't want to restrict myself to one certain colour type. This is why I chose to define an image type class, which encompasses more than one such colour type. This is why, in the accompanied Haskell file, you'll see multiple instances of this type class. Currently, there are four colour types defined: black-and-white (ImageBW), grayscale (ImageY), grayscale with an alpha channel (ImageYA) and RGB colours (ImageRGB). Of these four types, the black-and-white and grayscale types are most fleshed out. The other two have more of a dummy fitness function implemented, not fully using the data actually available. This means that when using this algorithm, preferably black-and-white or grayscale images are used. If you would want to use other colour types, I suggest looking at their corresponding fitness function. However, this is quite easy to change and won't cause you much trouble. The big advantage of the defined type class is that it's quite easy to define new colour types to work with. One only needs to 5 functions for a possibly new type: `mutation, crossover, orgToSeq, fitness` and `randomCircles`, each of them explained in the Haskell file. The main genetic algorithm does not need to be rewritten for this new type, which is the big advantage.

The main function in the file is the `writeResult` function. This function takes a great number of arguments to specify all arguments forthe genetic algorithm itself and takes two filepaths. One filepath to specify the location of the to be replicated image and one filepath to specify the location where to save the resulting image of the algorithm. The arguments are as follows:

1. *seed*: this is the seed which is used for the random part of the algorithm. The advantage of this is that we can replicate results of the algorithm directly, because we know with which

random number the algorithm started. If we wanted to truly randomize the algorithm, we could randomize this first seed.

2. *generation size*: this is an integer representing the size of each generation.

3. *number of generations*: this is an integer representing the number of generations we want the algorithm to run.

4. *crossover parameter*: this is the crossover parameter for the genetic algorithm, as explained in the introduction.

5. *mutation parameter*: this is the mutation parameter for the algorithm, as explained in the introduction.

6. *radius*: this is the radius of the shapes (circles in this case) used to replicate the image.

7. *number of shapes*: this is the number of shapes that each organism will consist of. In this implementation, this number is fixed. This means that this argument and the radius determines somehow the 'resolution' of the resulting image.

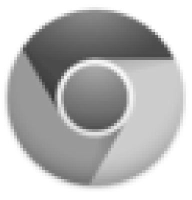
8. *filepath for output*: this is the string representing the filepath where the output of the genetic algorithm will be printed. This output will the best organism after the algorithm has run its course converted into its corresponding image.

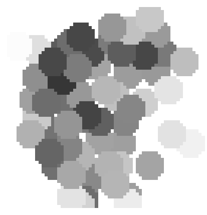
9. *filepath for input*: this is the filepath that should lead to the to be replicated image.

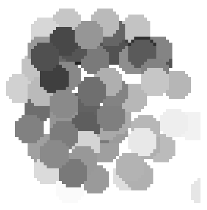
Note that there is also a `writeResultBW` function in the file. This function does exactly the same as the `writeResult` function and takes exactly the same arguments, but it works only when the image you want to replicate is a pure black-and-white image.

4 Results

One of the great disadvantages of my current implementation is that it takes quite a while to run. I have done a few runs for a 64×64 picture. Here it takes about 800 to a 1000 generations to come somewhere close to the original image. However, these runs take about 12 hours on my laptop to run. For this reason, it's quite tricky to do some really impressive testing. However, below I have outlined a few of the runs I have done on a grayscale image.


Original picture, 64×64 pixels


Result after 800 generations, with $p_c = 0.02$, $p_m = 0.01$, 50 organisms per population, 150 circles per organism


Result after 1000 generations, with $p_c = 0.02$, $p_m = 0.01$, 50 organisms per population, 200 circles per organism

Bibliography

[1] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, 1998.

[2] Melanie Mitchell. *An Introduction to Genetic Algorithms.* 1999.

[3] Olim F. Tuyt Philip W.B. Michgelsen. Genetic programming made easy: Reproduction techniques. *small treatise outling reproduction techniques.*