Modified Huffman Code for Lossless Compression and Bandwidth Optimization and Applying Genetic Algorithms to Generating Paintings Based on Images

Alexander Hansen

Abstract

This thesis contains two projects. A modified Huffman code is presented as a lossless method to compress common traffic types. We posit the usage of compression benefits instead of just frequency of occurrence, as is common in Huffman codes, as the priority of each node when constructing the Huffman tree. We show the effectiveness of this method on common data transmission types and describe what would be needed for adoption of this new algorithm.

We explore genetic algorithms as a method to create paintings based on images. We find a balance between computational work required and visually pleasing results to the algorithm, prioritizing aspects of the parameter space based on their impact on the painting and how they impact computational workload.

Acknowledgments

Thanks to Dr. Mark C. Lewis, my thesis advisor, and Dr. Paul Myers, my academic adviser, for being ever-present entities in my education at Trinity University.

Modified Huffman Code for Lossless Compression and Bandwidth Optimization and Applying Genetic Algorithms to Generating Paintings Based on Images

Alexander Hansen

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 04, 2018

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Modified Huffman Code for Lossless Compression and Bandwidth Optimization and Applying Genetic Algorithms to Generating Paintings Based on Images

Alexander Hansen

Contents

List of Tables

List of Figures

Chapter 1

Modified Huffman Coding for Lossless Compression and Bandwidth Optimization

1.1 Introduction

This project is a compression algorithm based on the Huffman code. We maximize the amount of tokens in the tree in exchange for a smaller compressed message. We avoid having to store the entire Huffman tree with the compressed message by storing it separately, giving an identification number to it and prepending all messages compressed using that code with that identification number, so it can be used to decode it later. Instead of constructing a Huffman code for every single file we compress, we construct a number of generalized Huffman codes based on a category of data and assign each an identification number. These codes and identification numbers together become a "data context". We

implement a program that holds many data contexts and assigns the best one to each input message and observe how much we can compress messages and how quickly the messages can be decoded

1.1.1 The Problem

Cisco estimates that in 2016, over 96,000 petabytes of data were transferred across networks around the world.[18] This is up from 72,000 petabytes in 2015 and for an even more extreme comparison, 12 petabytes in 1998. Clearly, internet traffic is increasing at a great pace. As internet traffic increases, consumers are sending more and more web requests from their client devices every day, with more and more devices to send from. This creates a challenge for those who maintain servers that handle these requests. Savings of kilobytes or even just bytes can add up to have a significant impact on total bandwidth usage when requests are being processed en masse. If there were a way to more effectively compress data before sending it to the client, provided it is quick at both compressing and decompressing and is relatively easy to implement, the savings in bandwidth could be huge, thus providing benefit to any implementing networks. We suggest a compression algorithm and implementation that meets these criteria.

1.1.2 Common Transmission Types

The majority of traffic on the internet is either text, image, video, or audio. This paper will focus on a text implementation due to the ease of representing text on paper. However, it has already been shown that Huffman code techniques can be used to compress images, video, and audio as well.[1][22][2] The algorithm we present can therefore also be used on these formats.

Recently, the concept of minimizing Javascript of other web languages has become more

and more important. If a large company with a website that is accessed a lot is able to minimize their payload, or filesize of their webpage, they could save lots of bandwidth. Our solution can be used to compress code very efficiently, so this is another reason for focusing on text in our implementation and examples.

What is important is that all common network traffic types can be tokenized, and when tokenized, there is at least some redundancy present. This is when minimum redundancy codes such as the Huffman code are most useful and effective. Note that there is a lot of redundancy in Javascript code (and most code) in the form of variable names, bracketing, keywords, etc., making it a great target for Huffman-based compression. Even if there is not that much redundancy, a Huffman code can be used to hold sets of equivalently occurring tokens to save memory.[15] A Huffman code based solution is therefore very adaptable to the terrain of modern data transmission.

1.1.3 Proposed Solution

Bandwidth is clearly the core concern. We could minimize bandwidth with all sorts of lossy compression, but lossy compression is not acceptable in an algorithm designed for all kinds of web traffic and could result in corrupted data. We seek to minimize bandwidth usage with only lossless techniques.

As memory and data storage device prices decrease every year, it is becoming less and less important to be efficient in space complexity.[13] The traditional Huffman code is fairly conservative in its use of space. We seek to take advantage of this modern trend by inflating the size of the compression and decompression programs themselves in exchange for a smaller compressed file size and thus better compression ratio. Transmitting this smaller file will then use less bandwidth.

From a high level perspective, the version of the Huffman code we present has two major

deviations from the original specification. The addition of a data context and the offloading of the Huffman code tree itself into the implementation specification, saving the output file from having to store the entire tree. The addition of a data context has been discussed before, but it has not been as large or offloaded as ours is.[22] We also introduce the concept of a compression benefit. That is, the benefit of compressing something big is more than compressing something small. This will be discussed more later.

A Huffman code is constructed by breaking apart an input message into tokens and constructing a tree based on those tokens and how frequently they occur, with the most frequently occurring tokens at the top of the tree. This tree can then be traversed to encode or decode messages. An example of a traversal represented as a binary string is shown in Figure 1.1. In this example, 0 represents going to the left child and 1 represents going to the right child. The tree itself and the traversal are then written to the disk in an efficient manner, providing a compressed file. If the tree grows too large, then writing it to the output file could make the output file larger than the original input. This is why the Huffman code's most common used method of tokenizing an input is by the letters and symbols used in English, ensuring that no more tokens than the English alphabet and some other symbols are encoded. As this sort of memory conservation is not important in our implementation, we want to choose a less granular token. In our example of English text, we could choose a word or small phrase. We will use a word, as implementing a decision algorithm for determining what constitutes a small phrase is not the focus of this paper, but such an algorithm could be a great addition.

With these less granular symbols, in our case words, we can then construct a much larger Huffman code tree based on the text. This tree should not be based on one individual file, rather, a corpus based on some genre of text. These subsets should be similar in genre, providing a more accurate tree. As an example, if one were to construct a Huffman code tree

on all papers submitted to a computer science journal and then use that code to compress the next submitted paper, it would probably do a decent job. If that code was then used to compress all tweets made in the past hour, it would probably not compress that file very well, barring some extreme coincidence. This genre-generalized Huffman code tree is the "data context", or context of the data being transmitted.

We then take this very large data context, created from a corpus of computer science academic papers, and store it in the encoding/decoding program with an identification number. The identification number is a fixed-length message before the actual encoded message that specifies which data context was used to encode this message. The output file then does not have the large Huffman code tree in it taking up space, rather, it has an identifier for a data context that it will be paired up with when it is received and decoded.

There is also the consideration of benefits. If the word "a" occurs extremely frequently in the input, it will be in one of the top levels of the Huffman code and be compressed down to a very small size. However, if we compress the word "the" down to that same small size, even if "the" is less frequent, we are potentially (depending on the frequency of occurrence of "the"), going to be compressing more efficiently. For the sake of example, assume a one-byte letter size and the token is on the third level of the tree, meaning a three-bit compressed token size. We would be removing 5 bits from the representation of the token "a" and 21 bits from the representation of the token "the". The benefit of compressing a symbol is the frequency at which it occurs times the size of that token. We then use these benefit values instead of the normal frequency value used by a standard Huffman code.

As decoding a compressed file using a Huffman code is very fast, this file can be decompressed quickly.[9] We also considered using different branching factors to minimize the depth of the tree, but this turns out to have no real impact on output file size.

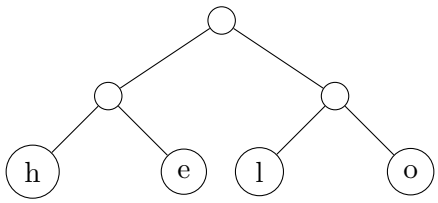


Figure 1.1: A rudimentary Huffman code tree with a corresponding binary string that encodes the string "hello". A "0" in the string corresponds to going to the left child of a node, and a "1" corresponds to going to the right.

1.2 Implementation and Methods

1.2.1 Specification

The implementation of this algorithm has two parts: a client and a centralized server that keeps track of data contexts. The client needs to have access to a set of standard data contexts (Huffman code trees), their identifiers, and have the ability to traverse them to decode or construct a lookup table to encode a message. The server needs to store all data contexts and their identifiers and provide updates to the clients when needed.

The client as described here is not the typical web definition of a client as a browser, it is an implementation of our algorithm to encode or decode messages. The clients will requests new contexts from the server if they come across a context identifier for which they do not have the context. The client will also be tasked with determining which context, given its current inventory, is most efficient. The simple, and slowest, way to do this is to compress the message with every potential data context and pick the best one. There are much better ways to do this using genre classification, which is an entire field of its own. What is important here is that the process of deciding which context would be optimal does not have to be a brute force solution like what we are using. We used that in our implementation that for simplicity's sake, though. Because the decision of which data context to use must

be done when encoding, the process of encoding will indeed be slower than decoding. This is okay, as it could be done in advance (not on the fly as requests are made) to all files being sent. For dynamically generated files, a smarter classification algorithm would be needed to assign an optimal data context on the fly.

An encoded file consists of the context identifier, the compressed data, and any tokens that were not found in the data context at the end. This is necessary because, as the contexts are not generated based on the input file itself, it is possible for an input file to have tokens in it that are not in the context. There will be a node in the data context that is in the least likely position, a leaf at the farthest level of the tree, that will be a placeholder. The tokens not present in the data context are then stored in their original form at the end of the file, in the same order that the placeholders occurred. This does mean that if a very malicious input message were to somehow circumvent all available contexts and contain a large amount of tokens that no context contained, it would be possible for the message to actually expand instead of being compressed. This is, however, both extremely unlikely and could be checked by comparing the output message size to the input message size and opting not to compress in this case.

The context identifier at the beginning of the file would be however long the implementers feel necessary for it to cover all contexts they would desire. This sort of decision has been historically difficult to make correctly, as the popularity of a particular system is hard to gauge at its inception (e.g. IPv4). In our implementation, it was three bytes, with an option to check the end of the file for more version info if all three bytes were 255.

1.2.2 Advantages

The solution to the problem of bandwidth optimization is the primary advantage. This algorithm can cut down on file sizes in transmission by immense amounts. This algorithm

also can be implemented easily, without much more difficulty than a regular Huffman code. This is important, as the adoption of this algorithm would require browsers to implement decoders and servers to implement encoders (clients and servers here referring to the web sense of the words, not our specification's definition). It is also adaptable to many different data formats and allows for a lot of data contexts, so a data context can fit a specific input message very well.

1.2.3 Disadvantages

Corner cases do exist. It is hypothetically possible for a file to expand, or get abysmal compression. We do avoid this as much as possible, but there will theoretically always be some heinously complicated input that could not be compressed very efficiently. This would not happen in general, though.

The disk space required to store so many data contexts could become very large, especially when accounting for different data types (audio, video, etc.). We assume disk space is not an issue, however, and disregard this.

Lastly, this algorithm is not very good at handling random data or noise. If a person is browsing their computer late at night and drifts off to sleep, their head could hit the keyboard and generate the string "ugibigbnhdfvbsmdofu". This would most likely not belong to many contexts, and if it did, that context would not fit the rest of the text very well. This is not too big of a problem, though, as it will just remain uncompressed via the placeholder nodes in the context. The implementer also has the choice to be more smart about these situations and perhaps resort to a different compression algorithm.

1.2.4 Our Implementation and Strategy

Our implementation used the programming language Rust, for its speed and ease of implementing low-level data structures safely (i.e. without memory leaks or things of that nature).

The construction of this tree can be seen in the appendix as Figure A.2. First, the corpus is split up on spaces. In a more efficient implementation, punctuation would also be its own token, but I have omitted this from the appendix for brevity. These tokens are then stored as tuples with their benefits, calculated as their length times their size. These tuples are then inserted into a priority queue/min heap strucure which is popped repeatedly to constructed the Huffman code. A lookup table for quicker encoding is constructed from the code for all tokens it contains. Finally, some identifier (arbitrary at this point) is attached to the code to form a data context.

For encoding, each token is found in the lookup table with its corresponding encoded form. For decoding, the data context traverses the Huffman tree following the traversal prescribed by the binary encoded file. This means that encoding is $O(n)$ where n is the number of tokens in the input and decoding is $O(m)$ where m is the size in bits of the encoded message. Both are very fast operations.

1.3 Results

We constructed a few data contexts that are representative of data frequently transmitted over the internet, namely minified Javascript, English text sourced from novels, and social media English text (a corpus of tweets). We compressed another sampling of texts from the same genre as the data context and averaged the compression ratios to get the values seen in Table 1.1.

One particular advantage of using a Huffman coding in this manner is the bit-level output of the algorithm. As the actual implementation of characters varies by application (some applications will have single-byte char sizes, some have two-byte char sizes, systems that support Unicode have four-byte char sizes). The compressed size will always be the same number of bits, but the ratio can depend on how big a character or token is in the original file. The encoding and decoding part of this program took under one-fiftieth of a second total on my (somewhat slow machine, even with extremely large data contexts (for example, the English Novels context contained over $50,000$ tokens).

Great compression results can be had with this compression algorithm, especially in the realm of highly redundant data like code (e.g. Javascript). It would just need to be adopted by both servers and clients for it to be practical.

Table 1.1: Some compression results using data contexts. The ratios are the compressed size over the original size, both in bits. In the "Benefits Ratio" column is the result for the calculation using benefits, and in the "No Benefits Ratio" column, a standard Huffman code using just the frequency of token occurrence was used.

	Benefits Ratio	No Benefits Ratio
English Novels	0.29214	0.41200
English Social Media	0.38830	0.42221
Javascript	0.23819	0.38194

1.4 Future Work

We hope to construct many more data contexts for varying data types and investigate exactly how many contexts would be needed to adequately cover all common data transmission types. We also hope to implement the encoding and decoding programs as browser and server plugins, enabling transmission of data compressed in this manner and supporting adoption of this algorithm.

Chapter 2

Applying Genetic Algorithms to Generating Paintings Based on Images

2.1 Introduction

This project explores the application of genetic algorithms to generating paintings based on photos. We find various combinations of parameters for our genetic algorithm that strike a balance between computational work and visually pleasing and accurate results.

2.1.1 Background Information

The subject of generating paintings based on algorithmic operations applied to computers has been studied before in great depth.[12][23][19] These approaches are highly effective and expressive, but some can be stylistically limited due to their deterministic nature of

analysis. In the case of both [12] and [23], they analyze features from the original image, such as color or edge location, and abstract them into something that can become a stroke or collection of strokes and represent them in a table. The painting algorithm draws from this table to ensure the feature is represented in the final rendered painting. This is the approach of most painting algorithms that are based on an original image. Those other algorithms which paint images based on no original reference image, such as [19], are a different problem than this project, so they will be referred to sparingly.

As it is hard to say in art whether one approach is better or worse, this project seeks not necessarily to improve upon the previous work but to provide another entirely different tool for generating paintings based on a reference image. We utilize machine learning, specifically a genetic algorithm, to create a large amount of paintings and then evolve them into one, particularly accurate, painting. This accuracy is determined by a fitness function which is discussed later.

The concept of a genetic algorithm was originally proposed by A.S. Fraser with the intent of being able to computerize and simulate evolution.[7] Originally, it was much more focused on the mutation and mating aspects of it, and less on the fitness definition and iteration. In applying genetic algorithms to art, however, a creative use of a fitness function and a mutation function are required.

Much work has been done on genetic algorithms, and they seem to contain an inherent quality that makes them good at optimization problems. This can be seen in the amount of publications across many diverse fields using genetic algorithms to optimize their problems. Some examples are molecular geometry calculations,[5] electromagnetic design tools,[21] course scheduling.[4] There is much more work out there exemplifying this optimization characteristic. [16][14][10][6][20]

Genetic algorithms are often applied to optimization problems in this way because they

are able to find optimal solutions without necessarily knowing why they are optimal. In problems where there are features that can be evaluated and weighted based on how good or bad they are with respect to the final product, genetic algorithms are able to find optimal arrangements of these features. We apply this to paintings, as every stroke can be viewed as a feature and evaluated in this way. Some work has been done on applying genetic algorithms to art and in very interesting ways. Karl Sims has published a paper and created an art exhibit using one.[17] He, too, struggles with large parameter spaces in his work and discusses some ways around them, but his process is not based on a reference image as ours is, so we cannot take advantage of these approaches.

Sims also has an exhibit where there are sixteen different paintings, and visitors to the exhibit can select the most visually appealing ones. Their features are then mixed to create the next generation of paintings. This is a good demonstration of the idea behind genetic algorithms.

Other work on using genetic algorithms with art include Roger Johansson's implementation in Javascript of a genetic algorithm to approximate the best arrangement of fifty polygons to replicate the Mona Lisa with great success.[11] This project can be viewed online. [1]

2.2 Implementation

2.2.1 Applying a Genetic Algorithm to Painting

There are three core parts to this project: the generation function, the genetic algorithm, and the rendering. Paintings are a struct represented by a collection of strokes, which is a struct with a start point, an end point, a width, and a color.

[1] It can be viewed here: https://chriscummins.cc/s/genetics/

The genetic algorithm itself has five parts: the selector function, which selects which of the population to mate; the fitness function, which defines the fitness of a painting; the crossover function, which defines how two paintings interact to create a child; the mutation function, which randomly mutates a painting; and the simulator, which iterates the genetic algorithm according to some given properties and kills off stochastically.

The structure of the program, from a high level, is as follows:

1. The generator function generates n initial random paintings.
2. The selector function selects m of the initial random paintings to mate and produce child paintings.
3. The crossover function takes two selected paintings and combines their strokes, half from one and half from the other, to create a child painting. It repeats this process until all of the selected paintings have been mated.
4. Another selector function selects o random members of the population.
5. The mutation function mutates these o members.
6. The simulator then stochastically kills off m paintings from the population.
7. Go back to step 2 and repeat until a predetermined number of iterations has been met.
8. Render the most fit, according to the fitness function, of the population and output it as the final result.

2.2.2 The Fitness Function

This paper will frequently refer to the "fitness" of a painting. This is a value determined by the fitness function. It is therefore crucial to explain first, as the rest of the discourse in this paper relies on this concept. The fitness function gives us an integer value for how "good" a painting is with respect to the original photo. This allows the various parts of the simulator to address photos by how fit they are and make decisions accordingly. It also provides us with a quantitative metric with which to compare the effectiveness of various approaches and tweaks to our program.

In our problem, we want to take an image and turn it into a painting. One of the most basic concepts of how accurate a painting is to a given reference photo is how far off the color is of every pixel. For example, given the RGB values of original photo pixel located at $(1, 1)$ as $(255, 254, 255)$ and the corresponding painting pixel located at $(1, 1)$ as $(255, 255, 255)$, we can see that the G value is one pixel off from the original. This gives it an "unfitness" of 1. We can take the inverse of this to get the fitness. The inverse is the maximum possible difference $(255 + 255 + 255 = 765)$ minus the actual difference (1), so $765 - 1$. Therefore, this particular pixel in the painting has a fitness of 764. The fitness value of a painting is the sum of the fitness of every rendered pixel in comparison to the original photo.

Note that the fitness function does not rely on mutating any data, only referencing the two images. This means that it is easily implemented in parallel, a feature that is important in executing genetic algorithms in a timely manner. The actual code for this function can be seen in the appendix as Figure **??**.

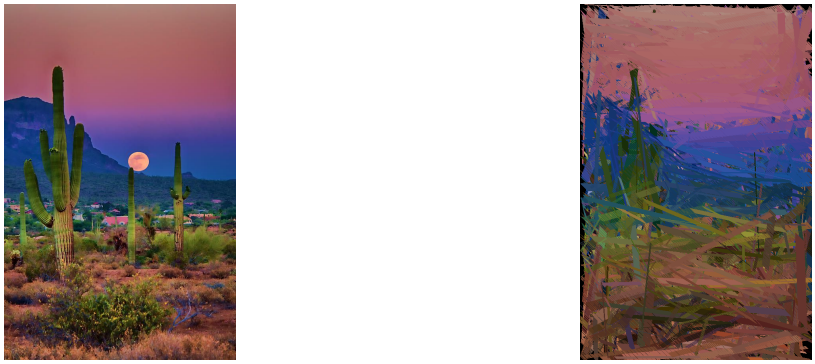


Figure 2.1: A rendered painting with its original reference image to the left of it

2.2.3 The Rendering Function

Rendering a painting is as minimalist and intuitive. From the start point to the end point of every stroke, a line is drawn with the stroke's color. This line has the width of the stroke's width. There is no actual paint simulation or anything particularly advanced happening here. This is because the discoloration of paint or lighting could interfere with the fitness function's ability to accurately compare the color of a stroke to the original photo.

The rendering function is important to this project because it is used in the calculation of the fitness and therefore the assessment of the genetic algorithm's progress and quantitative results.

Figure 2.1 shows an example of a rendered painting with it's original image next to it.

2.2.4 The Generator Function

The generator function is the function that generates random paintings for the initial population of the genetic algorithm. Originally, it was truly random, generating completely random strokes with completely random parameters and combining them into a painting. This was not good enough. The parameter space of the problem is too large. Strokes have

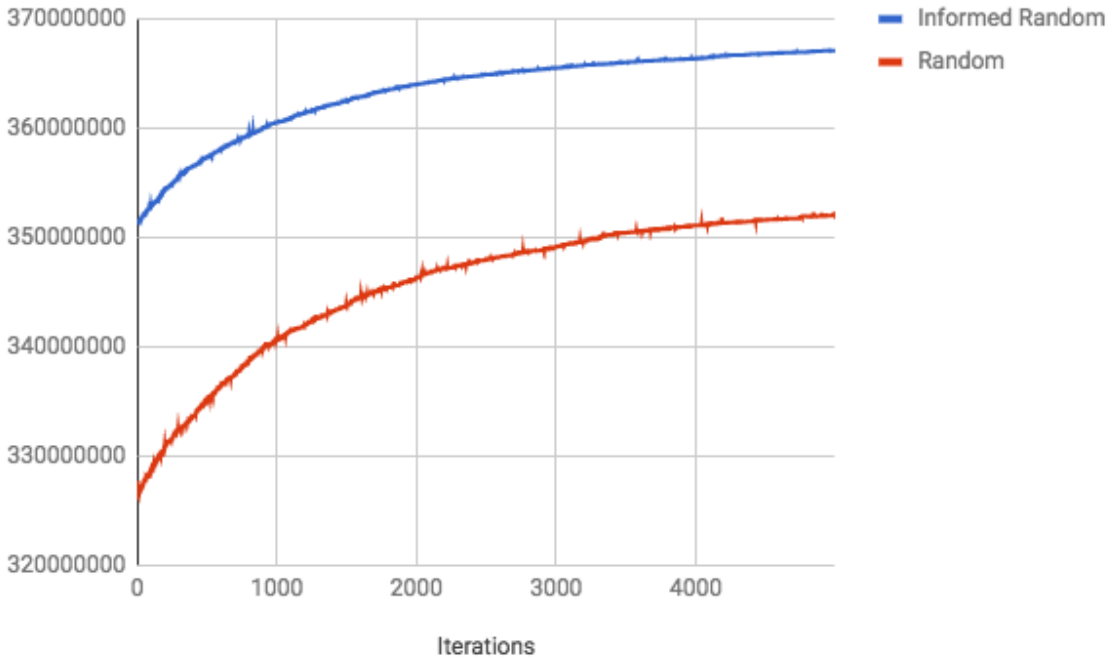
color, length, position, width, etc., and starting with truly random strokes puts us so far away from an optimal solution that it takes the genetic algorithm far too many iterations to get a desirable result. We instead created an "informed" random generator. In this improved generator, most aspects are still kept random but certain things are set to be more accurate by taking some features from the reference image. We limited the minimum and maximum length of strokes to be set by an argument passed in by the user, which minimizes the parameter space greatly. We also set the color of every stroke to be the color of the original image at its start position, and thus further shrank the parameter space. This proved to greatly increase starting fitness. Note that the fitness increase did not sacrifice the effectiveness of the genetic algorithm. The final fitness is a function of the starting fitness, as seen below. In Figure 2.2, we introduce the "fitness graph" which is a graph of the maximum fitness at every iteration through an execution of our genetic algorithm, to show the difference in both fitness and final image when an informed random generation function is used versus a purely random one.

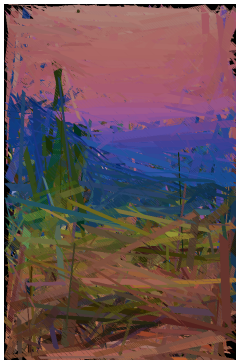
2.2.5 The Selector Function

The selector function's job is to select which parents will mate to produce the children for the next iteration.

There are different kinds of selectors that are used in genetic algorithms. There is stochastic, which is random and is effectively useless for this application; there is tournament selection, which chooses x members of the population at a time to participate in a tournament and picks the top 2 most fit participants from that tournament (multiple tournaments can be run per iteration); fitness maximal selection, which selects the absolute most fit members to mate; and some other more obscure types.

The amount parents being selected to "mate" each iteration is an input to these selector





(a) Informed random final result.

(b) Random final result.

Figure 2.2: The maximum fitness of every iteration's population using the informed random generation approach versus the non-informed approach, and their respective final results from the genetic algorithm after 5000 iterations.

functions. So, the maximize selector picks the top x to mate, producing $\frac{x}{2}$ children. This is a parameter that we fixed to $\frac{n}{3}$, where n is the total population size, producing $\frac{n}{6}$ of the population size children every iteration, and killing off $\frac{n}{6}$ as well. We tested a few other values, but ultimately fixed it to $\frac{n}{3}$ as it seemed optimal. This is discussed more in the results section. Fixing this value, although semi-arbitrary, provided consistency as we tested the other elements of the genetic algorithm's parameter space, which was already far too large. The maximize selector was our primary choice as it picks the most fit paintings and then uses them to create children. This can cause a phenomenon of a local maxima, but it still resulted in the highest fitness. That is, the local maxima level is higher in the maximize function than the highest fitness able to be achieved by other selectors in the same amount of time. This will be discussed more in the results section.

The tournament selector was the other selector we experimented with. It takes two parameters: the number of tournaments and the number of participants in the tournament. We fixed these values, again somewhat arbitrarily, to $\frac{n}{6}$ tournaments of size $\frac{n}{10}$. For a population size of 1000, this would be 166 tournaments of size 10. This results in 1660 fitness calculations per iteration, which is more than the maximize selector, but is often more efficient as it will select the same population member multiple times and access the cached fitness value. The scaling of the tournament selector is independent of the fitness increase, meaning it is not just a slower maximize selector.[8] This means it could be resistant to the local maxima that the maximize selector can sometimes experience.

2.2.6 The Crossover Function

After a subset of the population has been selected to become "parents", they are passed into a crossover function. In the original definition of a genetic algorithm, this was a function that took the two parents' binary string representations and split them, combining part

from one parent and part from the other.[7]

We approximate this process; instead of using a binary representation, we take half of the strokes from one painting and half of the strokes from the other, combining them. To maximize the increase in fitness, we compare the two potential combinations (parent one's first half, and then parent two's, or vice versa), and pick the best combination. This worked, but also made us more susceptible to local maxima, also known as Hamming walls. This will be discussed later in the results section. Because of this, this feature was removed. Some other versions of the crossover function exist, such as not taking exactly half from each painting and instead taking a random percentage from each that totals one hundred percent, but no major improvement in performance was shown using this method. This new set of child paintings are then added into the total population.

2.2.7 The Mutation Function

After the child paintings are added to the population, a stochastic selector function selects a predetermined amount of paintings to mutate. We decided that we would pick a random parameter of a stroke and modify it by a predetermined "mutation strength". The code of our original implementation of the mutation function is available as Figure A.1 in the appendix. You will notice that in the end, this function, like the original crossover function, also makes a comparison to ensure a better fitness. The painting compares itself to its pre-mutated state and picks the more fit painting to return. This also made us more susceptible to Hamming walls, and was removed. The mutation strength is better as a relatively large value to help avoid Hamming walls.

After the mutation occurs, a number of paintings equivalent to the amount of child paintings that were just added are stochastically selected to be "killed", or removed from the population.

2.3 Results

2.3.1 Maximizing Fitness

Some of the most fit images generated based on the original reference image are below:

More paintings can be seen in the appendix as Figure **??**.

Stroke Properties

These two paintings ran for the same number of iterations. They show the increase in "difficulty" of arranging longer strokes versus shorter strokes. Through many tests, it has become apparent that a feature in a painting only shows up if the stroke length is less than that feature itself. The longer strokes tend to obscure the smaller details, like the moon in the background and some shrubbery in the foreground. In a much shorter time, a shorter-

stroke painting is able to achieve a much higher fitness than another painting of longer strokes. So, one way to maximize fitness is to just have shorter strokes. Thinner strokes also have this effect but it is not as dramatically as shorter strokes. This "impressionist" style of painting strikes a great balance of computational work to good results, requiring much less computation for much more fit results. For the sake of exploring other styles, we then chose to move on to the more challenging longer-stroke version and see how much improvement can be made.

Selectors

When trying to maximize fitness, it is intuitive to use the maximize selector. This is the selector that selects the most fit members of the population to mate for the next generation. It does increase the run-time dramatically, as at every iteration it must find the most fit members to mate, but the increase in fitness makes this trade-off worth it. Using the maximize selector to run a genetic algorithm for some amount of time hours will result in a higher fitness than using a tournament selector for the same amount of time. Technically, the tournament selector can get through many more iterations in the same time, but each individual iteration has a lesser positive impact on the fitness.

2.3.2 Hamming Walls

When looking at the fitness graph of an execution of our genetic algorithm, it takes a logarithm, or inverse-exponential, shape. This can be seen in Figure 2.3. We postulate that this is due to a local maxima, or Hamming wall as they are called in genetic algorithms. This postulated maxima can be seen in Figure 2.4. This is a place where an increase in fitness would require many changes to happen simultaneously.[3] It could also be due to approaching the most optimal arrangement of strokes, but as a perfect fitness value is much

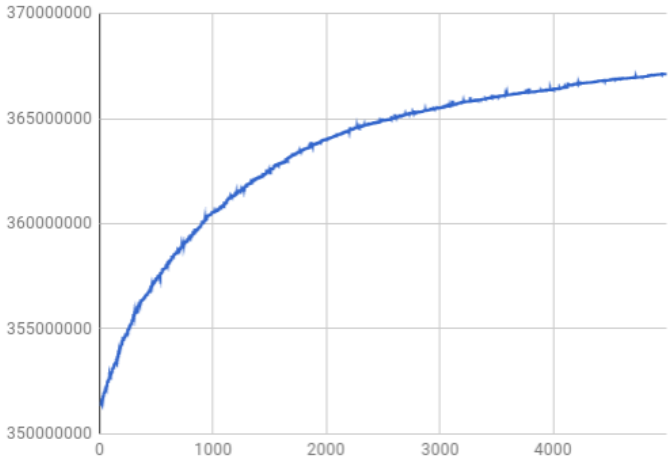


Figure 2.3: A graph of the maximal fitness throughout one execution of the genetic algorithm.

higher than those values these lines are asymptotically approaching, this seems unlikely.

More evidence for the existence of a Hamming wall can be seen in Figure 2.5. If the posited hamming wall was actually an optimal arrangement of strokes, then the fitness graph for the random generator execution should approach the same optimal arrangement as the informed random generator execution, around $368,000,000$, and then start to flatten out. Instead, it reaches its own Hamming wall around $351,000,000$.

If one is familiar with Hamming walls, they may have heard of Gray coding. Hamming walls can be combated with something called Gray coding[3] when using binary strings to represent members of the population. As our approach to using genetic algorithms on paintings does not use a binary-based representation, these methods of Gray coding are unfortunately not applicable. There may be ways to apply the strategies of Gray coding to our application, but they are not obvious at this point.

Hamming walls, or local maxima in general, occur when an algorithm cannot see far enough in the future to make a change that will temporarily distance itself from its goal

but eventually get it closer. When thinking about them in this sense, any part of our algorithm that greedily grabs the best option could be causing this phenomenon. The most obvious candidate is the maximize selector, which always grabs the most fit members of the population. For many iterations, that most fit member may not actually change at all,

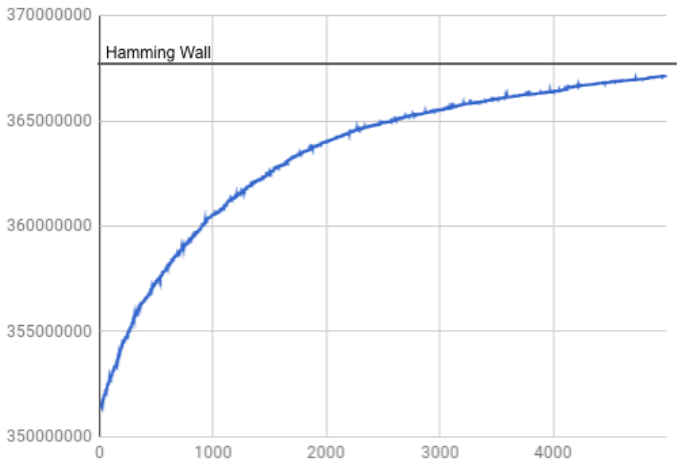


Figure 2.4: A fitness graph with a Hamming wall visualized. The Hamming wall is around fitness $= 368,000,000$.

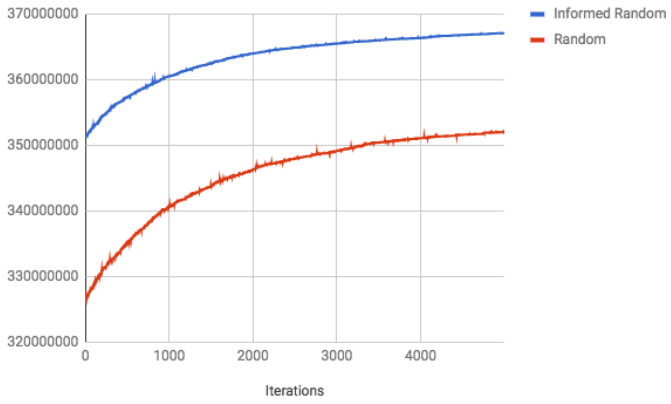


Figure 2.5: The fitness graph of two executions using an informed random and a random generator function.

resulting in a lot of children with the same or extremely similar characteristics being added over and over again. Over a longer time, it is theoretically possible that a sudden mutation and more fit child could cause a spike and an overcoming of a Hamming wall.

Increasing the previously mentioned mutation strength would increase the probability of being able to overcome a Hamming wall. We tested raising the mutation strength until it was detrimental to the overall fitness and it never was able to overcome Hamming walls.

Overcoming Hamming Walls with Selector Choice

Discouraged by this Hamming wall, we turned to the tournament selector. It was touched on before, but for the sake of detail: the tournament selector works by selecting tournaments out of the total population. These tournaments each have x participants, where x is less than half of the population. It then picks the most fit two participants to make a child for the next iteration. This selector can sometimes be quicker than the maximize selector as you will only need to calculate the fitness of the tournament participants, not the entire genetic algorithm. This can also help avoid Hamming walls, as it will often not select the most fit or most optimal path for the next iteration. This results in a more flat, but perhaps more consistent, slope.

Figure 2.6 shows the maximal fitness of every iteration using the tournament selector. Notice that it is more linear shaped, and less logarithmic, which is good when hoping to avoid asymptotically approaching a Hamming wall.

This optimistically linear slope is immediately overshadowed when the tournament selector graph is plotted along with the maximize selector graph. This can be seen in Figure 2.7. Clearly, the maximize selector is far more effective. The postulated Hamming wall is so far above anything the tournament selector achieved in the same number of iterations. The genetic algorithm with a tournament selector would have to run for approximately

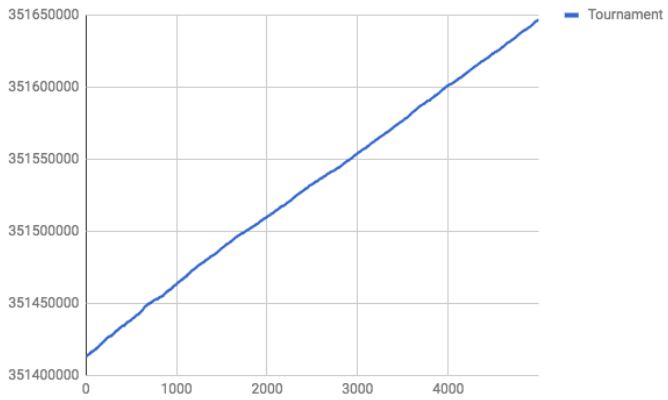


Figure 2.6: The maximum fitness of every iteration using the tournament selector.

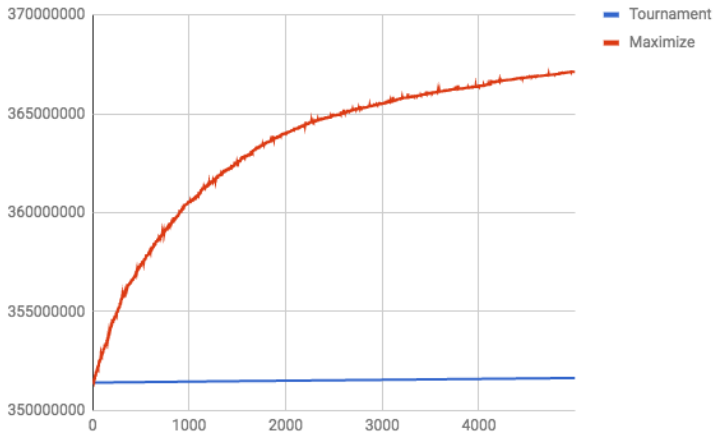


Figure 2.7: The tournament selector's maximal fitness at every iteration graphed with the maximize selector's.

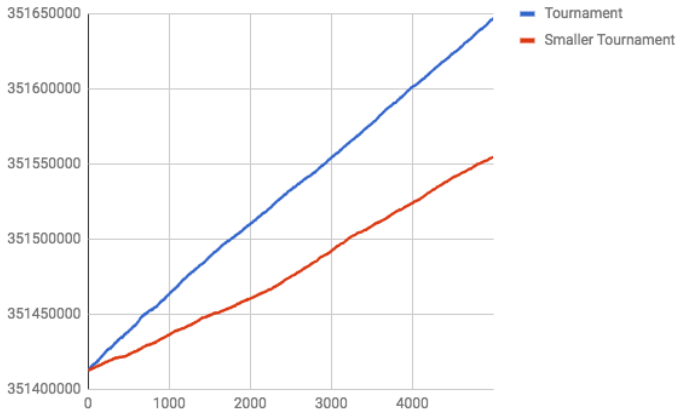


Figure 2.8: The fitness graphs of our tournament size and a smaller tournament size.

190, 503 iterations in order to overcome the one using a maximize selector's fitness. As the maximize selector took about 17 hours to run 5000 iterations and the tournament selector took about 13 hours to do 5000 iterations. The tournament selector would have to run for roughly 494 hours, or 41 days straight on our systems to even approach the Hamming wall of the maximize selector, assuming the linear slope would continue and it would not encounter its own Hamming wall at an earlier point. In that same time, the maximize selector would still be either slowly approaching the Hamming wall, or could be even higher than it. The amount of time required to achieve a comparable result with the tournament selector makes this approach not worth it unless running the genetic algorithm for a very long time. Lowering the tournament selector's parameters to do less tournaments or less participants in the tournaments does make it run faster but has a dramatically negative impact on the its slope. The increase in speed is not big enough to compensate for this negative impact on the slope. This can be seen in Figure 2.8. The parameters of these two executions can be seen in Table 2.1.

From this test and similar ones, we deduced that it is not worth it to pursue smaller

Table 2.1: Parameters of the tournament selector executions

	Number of Tournaments	Tournament Size	Runtime	Final Fitness
Tournament	$\frac{n}{6}$	$\frac{n}{10}$	13:12:52	351646879
Smaller Tournament	$\frac{n}{10}$	$\frac{n}{12}$	11:47:50	351554728

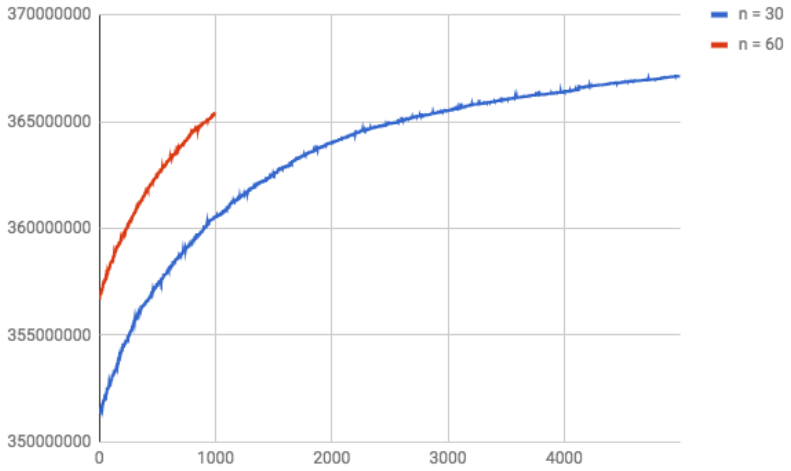


Figure 2.9: The fitness graph of two executions of the genetic algorithm: one with the population set to $n = 30$ and one with the population set to $n = 60$.

tournament parameters for more quick execution. The better balance of computational work and fitness is therefore in the larger tournament selector, but the best balance is still found in the maximize selector.

Due to these above factors, we moved on from tournament selection and used maximize selection for all tests after that.

2.3.3 The Impact of Population Size

The size of the population in each iteration turns out to have a significant impact on the shape of the fitness graph. Unfortunately, it also increases the time it takes to execute one iteration greatly.

For all executions mentioned up until this point, the population value has been set to $n = 30$. Figure 2.9 shows what happens when we increased the population size to $n = 60$, using our maximize selector. Unfortunately, it took so long to run, we were only able to run it out to a thousand iterations. But clearly the impact is massive. The slope at every point is higher and the initial fitness is higher. This makes sense from a conceptual perspective, with more population being generated in the beginning, there are more combinations of strokes and therefore a higher likelihood of more fit paintings being generated. Unfortunately, to run these 1000 iterations took 69.18 hours. Recall that running 5000 iterations with $n = 30$ took only around 17 hours. We found that doubling the population size increases the amount of time required to run one iteration by a factor of 20.347, on average. This means that increasing the population size is not as efficient as running a lower population for a longer time.

Some more examples of paintings and their reference images can be seen in the appendix as Figures **??**, **??**, **??**, and **??**.

2.3.4 The Impact of Iterations

The amount of iterations an execution has exerts a very predictable effect on the fitness graph. Shown in Figure 2.10, the number of iterations merely determines how far along this logarithmic curve it goes. Note that even when a random generator is used, giving a lower fitness to start with, the pattern persists.

2.4 Future Work

In the future, we hope to look into how to implement Gray coding for a project like this to avoid Hamming walls. New methods of combating Hamming walls may be needed in order

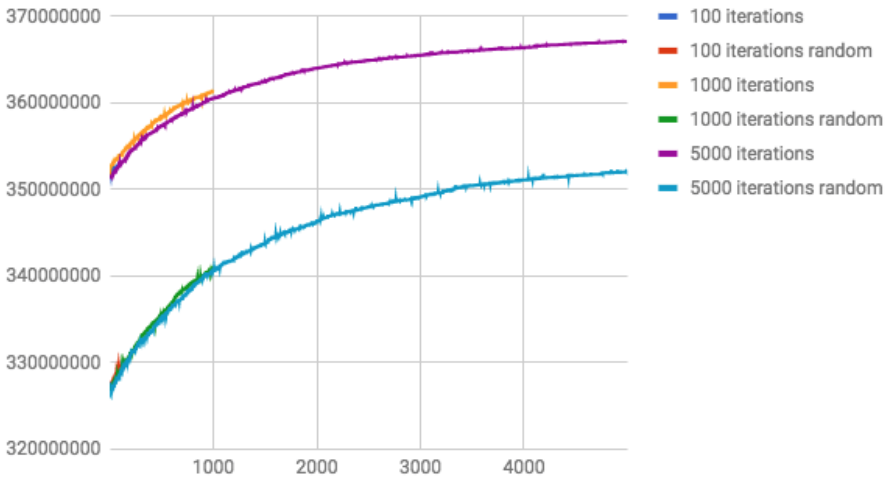


Figure 2.10: The fitness graphs of six executions, three of which used a random generator function and three didn't, of varying lengths (amounts of iterations).

to solve the problem in this context. We also hope to test more parameter configurations for our selectors, finding more optimal configurations. Lastly, we feel that implementing a more intelligent fitness function, maybe something that uses edge detection or something along those lines, could result in a more accurate evolution.

Bibliography

[1] S. Ashida, H. Kakemizu, M. Nagahara, and Y. Yamamoto. Sampled-data audio signal compression with huffman coding. In *SICE 2004 Annual Conference*, volume 2, pages 972–976 vol. 2, Aug 2004.

[2] M Atheeshwari and K Mahesh. Efficient and robust video compression using huffman coding. *INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY*, 2(8), Aug 2011.

[3] Paul Charbonneau. An introduction to genetic algorithms for numerical optimization. *NCAR Technical Note*, page 74, 2002.

[4] Peter Cowling, Graham Kendall, and Limin Han. An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1185–1190. IEEE, 2002.

[5] D. M. Deaven and K. M. Ho. Molecular geometry optimization with a genetic algorithm. *Phys. Rev. Lett.*, 75:288–291, Jul 1995.

[6] Mark Erickson, Alex Mayer, and Jeffrey Horn. The niched pareto genetic algorithm 2 applied to the design of groundwater remediation systems. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 681–695. Springer, 2001.

[7] Alex S Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences*, 10(4):484–491, 1957.

[8] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[9] Chen Hong-Chung, Wang Yue-Li, and Lan Yu-Feng. A memory-efficient and fast huffman decoding algorithm. *Information Processing Letters*, 69(3):119 – 122, 1999.

[10] Jeffrey Horn, Nicholas Nafpliotis, and David E Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 82–87. Ieee, 1994.

[11] Roger Johansson. Genetic programming: Evolution of mona lisa, 2008.

[12] Atsushi Kasao and Masayuki Nakajima. Synergistic image creator? a picture generation system with consideration of the painting process. 30:13–21, 09 1999.

[13] John C McCallum. Memory prices (1957-2017).

[14] Heinz Mühlenbein, M Schomisch, and Joachim Born. The parallel genetic algorithm as function optimizer. *Parallel computing*, 17(6-7):619–632, 1991.

[15] S. Pigeon and Y. Bengio. A memory-efficient adaptive huffman coding algorithm for very large sets of symbols. In *Data Compression Conference, 1998. DCC '98. Proceedings*, pages 568–, March 1998.

[16] Colin R Reeves. A genetic algorithm for flowshop sequencing. *Computers & operations research*, 22(1):5–13, 1995.

[17] Karl Sims. *Artificial evolution for computer graphics*, volume 25. ACM, 1991.

[18] Cisco Systems. Visual networking index.

[19] Corinna Vehlow, Fabian Beck, and Daniel Weiskopf. Painting with flow. *Proceedings of the IEEE VIS Arts Program (VISAP)*, pages 117–126, 2014.

[20] Matthew Bartschi Wall. *A genetic algorithm for resource-constrained scheduling*. PhD thesis, Massachusetts Institute of Technology, 1996.

[21] D. S. Weile and E. Michielssen. Genetic algorithm optimization applied to electromagnetics: a review. *IEEE Transactions on Antennas and Propagation*, 45(3):343–353, Mar 1997.

[22] M. J. Weinberger, G. Seroussi, and G. Sapiro. Loco-i: a low complexity, context-based, lossless image compression algorithm. In *Data Compression Conference, 1996. DCC '96. Proceedings*, pages 140–149, Mar 1996.

[23] Yili Zhao and Dan Xu. Monet-style images generation using recurrent neural networks. In Abdennour El Rhalibi, Feng Tian, Zhigeng Pan, and Baoquan Liu, editors, *E-Learning and Games*, pages 205–211, Cham, 2016. Springer International Publishing.

Appendix A

Code Appendix

Figure A.1: The original mutation function.

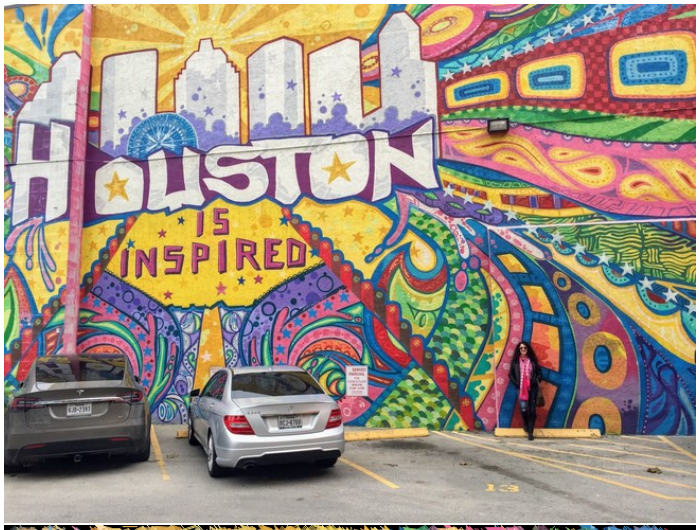
```
match rng.gen::<i32>() % 3 {
    0 => {
        to_modify.start.x = (to_modify.start.x
                             + rng.gen::<u32>() % mutation_strength) % self.width;
        to_modify.start.y = (to_modify.start.y
                             + rng.gen::<u32>() % mutation_strength) % self.height;
    }
    1 => {
        to_modify.end.x = (to_modify.end.x
                           + rng.gen::<u32>() % mutation_strength) % self.width;
        to_modify.end.y = (to_modify.end.y
                           + rng.gen::<u32>() % mutation_strength) % self.height;
    }
    2 => {
        to_modify.width = to_modify.width + rng.gen::<u32>() % mutation_strength;
    }
    _ => (),

}

s.strokes.remove(to_modify_index);
s.strokes.push(to_modify);
let post = s.fitness();
if post > pre {
    return s;
} else {
    return self.clone();
}
```

Figure A.2: The function for creating a data context.

```
/// Creates a DataContext based on a corpus.
pub fn new(corpus: String) -> DataContext {
    // Gather vector of pointers to individual words in the corpus.
    let mut tokens: Vec<&str> = corpus.split(" ").collect();
    // Create unique tuples of tokens and their benefit.
    let mut tokens_with_benefit: Vec<(&str, u64)> = tokens
        .par_iter()
        .map(|x| {
        (*x,(tokens.iter().filter(|&y| y == x).count() * x.len()) as u64)})
        .collect();
    // Add the "not contained" token with a benefit of zero
    tokens_with_benefit.push(("token not contained", 0u64));
    tokens_with_benefit.sort();
    tokens_with_benefit.dedup();
    tokens_with_benefit.sort_by_key(|x| x.1);
    // Create the Huffman tree. At this point, tokens_with_benefit is sorted
    // by lowest benefit to highest benefit. Note that the compare
    // function for this node type is inverted to make this BinaryHeap a
    // MinHeap, useful for creating a Huffman code.
    // Start with all singletons
    let mut forest: BinaryHeap<Node> = tokens_with_benefit
        .iter()
        .map(|x| {
            Node::Leaf {data: x.0.to_string(), benefit: x.1,}}).collect();
    // Merge the singletons one by one into a tree.
    while forest.len() > 1 {
    let left = forest.pop().expect("heap pop didn't work");
let right = forest.pop().expect("heap pop didn't work");
let benefit = left.benefit() + right.benefit();
        let new_tree = Node::Interior {
            left: Box::new(left),
            right: Box::new(right),
            benefit: benefit,
        };
        forest.push(new_tree);
    }
    // There should be only one node in the forest left, the root.
    assert!(forest.len() == 1);
    return DataContext {
        _context_id: "Test".to_string(),
        root: forest.peek().unwrap().clone(),
        encoding_table: forest.peek().unwrap().make_table(tokens),
    };
}
```

Figure A.3: The function for calculating the fitness of a painting. Here, "self" is a painting.

```
 pub fn fitness(&self) -> i32 {
        let mut fitness = 0f64;
        // The image we are trying to approximate.
        let goal = load_image(&self.filename);
        let rendered_strokes_buffer = self.render_strokes();
        for x in 0..goal.width() {
            for y in 0..goal.height() {
                let grgb = goal.get_pixel(x, y).data;
                let rrgb = rendered_strokes_buffer.get_pixel(x, y);
                let unfitness = (grgb[0] as i32 - rrgb[0] as i32).abs() +
                    (grgb[1] as i32 - rrgb[1] as i32).abs() +
                    (grgb[2] as i32 - rrgb[2] as i32).abs();
                fitness += 765.0 - unfitness as f64;
            }
        }
        return fitness as i32;
    }
}
```


HOUSTON
IS
INSPIRED