2016

Optimizing Website Design Through the Application of an Interactive Genetic Algorithm

Elijah Patton Mensch

Optimizing Website Design Through the Application of an Interactive Genetic Algorithm

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Elijah Mensch

Annandale-on-Hudson, New York
May, 2016

Abstract

The goal of this project was to determine the efficacy and practicality of "optimizing" the design of a webpage through the application of an interactive genetic algorithm. Software was created to display a "population" of mutable designs, collect user feedback as a measure of fitness, and apply genetic operations in an ongoing evolutionary process. By tracking the prevalence of design parameters over multiple generations and evaluating their associated "fitness" values, it was possible to judge the overall performance of the algorithm when applied to this unique problem space.

Contents

Dedication

This project is dedicated to my friends and family. I know that's a huge cliché, but it's really true—without their support, I never would've gotten this far.

Acknowledgments

The full list of people deserving of my thanks is altogether too massive to include here; I suspect it would occupy an overwhelming majority of this paper's length. For all of you that spoke to me, made me laugh, or even just tolerated my behavior over the course of the last few months, please know that I am grateful beyond words. You made this endeavor possible.

That being said, there are a few I'd like to mention. My project adviser Bob McGrail provided plenty of useful guidance and advice, always steering me in the right direction. In a similar vein, my academic adviser Sven Anderson shared machine learning wisdom with me that proved fundamental in shaping this project. Sven also deserves credit for introducing me to genetic algorithms, which I first explored in his course "Introduction to Artificial Intelligence." I'd also like to thank John Belk and Ethan Rogers for struggling alongside me; misery *does* love company. Finally, I'd like to thank my family: Melissa, Peter, Hannah, and Jona. Their love and support has gotten me through far more than just this project.

1
Introduction

1.1 Motivation

The growth of the Internet has been nothing short of unprecedented. Since the turn of the millennium, global Internet use has grown sevenfold, escalating from 6.5% of households to 45%. In the developed world, this number expands to more than 80% [1]. The modern world is increasingly characterized by a digital "connectedness" which comes with a commensurate increase in the importance of an effective online presence for businesses, corporations, and even government agencies [2].

A cornerstone of this presence lies in presentation. For a website to properly serve its purpose, an appealing design may be the most impactful tool available. A study on factors contributing to the development of visitor trust with online health websites suggested that a significant majority of users found design to be overwhelmingly more important than content when establishing the trustworthiness of the site [3]. There is no reason to believe this relationship would not hold true in other domains: generating "conversions," e-commerce, branding, or increasing effectiveness in any of the wide range of situations websites are used in today.

As such, the role of the designer is in high demand. Indeed, their job is far from easy: identifying "good" designs and design elements is an ongoing task without a clear right answer. Larger websites, potentially including tens or hundreds of pages, present an even more monumental challenge. This conundrum provides the chief motivation for the experiment described within this paper. Based on previous work done in applying evolutionary computation to art and user interfaces, it was believed that a similar approach could be taken in procedurally generating appealing website designs [28] [4]. Ideally, a well-designed algorithm could augment a web designer's workflow by identifying key features of a desirable design and "suggesting" an overall prototype as a framework for further improvement.

1.2 Project Goals

The main goal of this project was the implementation and evaluation of a genetic algorithm applied to the creation of webpage design. By treating design as an "optimization problem," it was hoped that said algorithm could procedurally improve on a "population" of designs in response to user feedback, eventually converging on a design or set of designs that best embodied overall preference. This approach raises many questions that the experiment hopes to answer, including:

- Can design be translated to an optimization problem?
- Will a genetic algorithm applied to this problem demonstrate convergence on a "good" design?
- How can the performance of a genetic algorithm be evaluated in this context?
- What is the most effective method of gathering user feedback to drive design evolution?

A secondary objective was the creation of server software to drive the experiment. Though necessarily a "means to an end," developing the software provided some insight into the difficulties of translating genetic algorithms to web design.

1.3 Related Work

A significant amount of research has already been done regarding the development of methodologies for the continual improvement of website design. Large sites like Amazon, Google, Facebook, and others run thousands of experiments a year aimed at improving all facets of their user experience [5]. This section discusses the two major forms these experiments take: A/B testing and multivariate testing.

1.3.1 A/B Testing

A/B testing is, in essence, a straightforward application of the scientific method to website design. Initially, a designer generates a "hypothesis" in the form of a tweaked layout: this may contain different copy text, images, colors, page layout, or any number of other design elements and combinations thereof. The next step is the execution of a controlled test. Further traffic to the webpage in question is partitioned into two groups: the first group receives the original, "control" page, while the second group receives the tweaked, "treatment" page. Ideally, these groups are split evenly for maximum statistical power [5]. Finally, observations are made: user interactions from both groups are analyzed and compared. Metrics gathered vary depending on the scope and goals of the experiment, but typically focus on a single, overarching criterion. For example, an e-commerce website might compare the ratio of site visitors to sales for both the treatment and control. If the treatment generates higher sales, the tweaked design will be used for all future visitors.

A/B testing is often expanded to include more than one "treatment" page, known as A/B/n testing. In A/B/n testing, multiple designs are served out to smaller fractions of the website's visitors. While this method is potentially more effective than standard A/B testing, it is not without risks: as each page receives fewer views, it might take longer to reach a statistically significant conclusion.

1.3.2 Multivariate Testing

Multivariate testing provides another method for evaluating and improving page design. Instead of evaluating entire pages as atomic units, an implementation of multivariate testing will decompose a page into multiple elements, each with its own design variations. For example, a generic sign-up page might be broken down into a header, footer, form, and background image. Visitors would then be funneled to pages containing all possible combinations of these elements. Following the test run, the design variables of each page are correlated with each other and to the overall performance of their respective pages.

In contrast to A/B testing, multivariate testing can provide more complex insights into which design elements carry the most impact and how they interact with each other. This can assist in targeting redesign efforts towards elements shown to be the most meaningful to users, both for future test runs and on entirely new pages. However, the increased complexity of multivariate testing requires significantly higher traffic before a meaningful conclusion can be reached. Additionally, the element-focused nature of the test means it is not suitable for radical redesigns—a limitation not shared by A/B testing.

Oftentimes, the best testing strategy may combine both the A/B and multivariate technique. Multivariate tests can be run to better understand which elements are

driving user behavior; that information is then used to focus larger and more radical redesigns that will be evaluated via A/B testing [6].

2
Background

2.1 CSS

The "World Wide Web," in its initial implementation, lacked an important feature: a method for controlling the presentation of a page beyond its general structure. As the web was originally intended as a platform for the sharing of scientific documents, *content* took priority over *presentation*: the results of a research report were of far more importance than the choice of typeface. This paradigm began to shift as the web broadened beyond scientific environments [7]. The increase in scope and variety of content provided the impetus for a standardized method of controlling document presentation: this manifested as the first proposal for "Cascading Style Sheets" in 1994 [8]. Since then, the CSS "spec" has been iteratively revised and expanded. The current version, CSS3, provides a comprehensive set of standards for controlling typography, layout, graphical elements, animations, and more.

CSS specifies a fairly straightforward syntax. A stylesheet includes a list of *rules*, which are composed of a *selector* specifying an element or elements the rule will affect and a matching declaration block containing one or more *declarations*. Each

declaration is split into a *property* and a *value* that together modify the style of the element specified by the selector. This format is demonstrated in figure 2.1.1 as an example stylesheet consisting of two rules.

```
1  p {
2      font-family: 'Times New Roman', serif;
3      font-size: 18px;
4      color: #070707;
5  }
6
7  .important {
8      font-size: 24px;
9      font-weight: bold;
10 }
```

Figure 2.1.1. CSS Structure

The first rule spans lines 1 through 5. The first line of a rule is always its selector. The selector in this case, "p," signifies the beginning of a block (wrapped in curly braces) that will affect all HTML "p" (paragraph) elements. The first declaration in the block, occupying line 2, is composed of the property "font-family" and the value "'Times New Roman', serif". This declaration indicates that the font used for text within elements specified by the rule's selector will be "Times New Roman," with a fallback to a generic serif font if the first is not available. The two remaining declarations in this block, occupying lines 3 and 4, specify a font size of 18px and a dark grey color, respectively. The color is written in the common hexadecimal format: three pairs of characters, each a hexadecimal value between "00" and "FF," correspond to the red, green, and blue values of a color triplet. Altogether, this rule will style all text within HTML "p" tags to be 18px Times New Roman with a dark grey coloration.

The second rule, spanning lines 7 through 10, utilizes a "class" selector. This is indicated by the leading dot: ".important" defines a selector that will affect all HTML elements with the class "important." The two declarations within specify a font size of 24px with a bold weighting.

An actual stylesheet in use by a webpage will likely be significantly more complex than the demonstration in figure 2.1.1. There are a multitude of selector types and an enormous variety of properties that affect everything from an element's margins to its behavior when hovered by a cursor. However, the basic syntax remains the same: a selector followed by a list of property-value pairs that define how an element will be styled. Visiting any modern-day site provides a glimpse of how much control is afforded by CSS: complex layouts, animations, typography, and more can all be determined within a stylesheet.

2.2 Genetic Algorithms

2.2.1 Overview

A genetic algorithm is, simply put, a mapping of classic Darwinian evolution onto the set of problems known to computer scientists as "optimization": the selection of a "best" element from a set of alternatives. The concept of applying biological phenomena to computational problem solving is nothing new: researchers have been exploring evolutionary methods since as early as the 1950s, with the first formal "genetic algorithms" appearing in the 1970s [9]. Today genetic algorithms have found utility in a wide array of applications, from protein folding to weather prediction. [10].

Before delving into the specifics of genetic algorithms, it is worth understanding the nature of the aforementioned "optimization" problems they are intended to solve. In the simplest case, an optimization problem consists of, for a given func-

tion, repeatedly selecting an input value and computing the output, then selecting whichever input produced the maximum (or minimum) value. More broadly, optimization is the search for the best solution among a range of feasible solutions. Figure 2.2.1 shows one example of an optimization problem: determining the global maximum of a paraboloid.

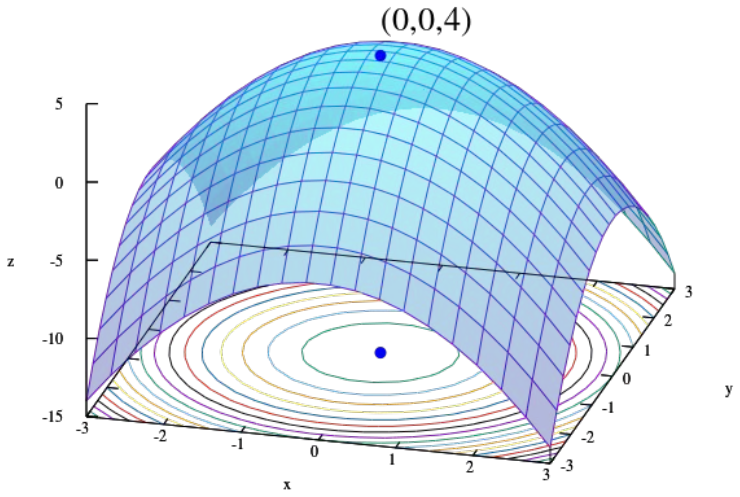

Figure 2.2.1. Global Maximum of a Paraboloid

Genetic Algorithms solve optimization problems by imitating the biological processes of natural selection and sexual reproduction. However, for these processes to be applicable in a computational environment, a suitable representation must be chosen for the given problem. To this end, the classical GA represents possible solutions as bit-string "chromosomes," with each bit (or each group of bits) representing a variable in the solution. For example, a GA tasked with "designing" tires for a vehicle might encode each tire as a string of 8 bits: bits 1-3 encode eight possible

diameters, bits 4-6 encode eight possible tread widths, and bits 7-8 encode four different tread styles (none, diamond, etc). 10100100, 11010001, and 11001101 would all represent different tires. After generating an initial randomized "population" of chromosomes, GAs begin a cycle of selection, crossover, and mutation, continuing until a terminating criterion is met [11].

The "selection" step is intended to mirror the principle of "survival of the fittest." Implicit in this process is an evaluation of every potential solution in a given population to determine which among them are the best. This introduces an important concept of genetic algorithms: the fitness function. Each chromosome must be evaluable by a fitness function that produces a "score" representing the performance of that solution towards a specific goal. In the tire example presented above, this function might take the form of a simulation that evaluates the tire's traction in inclement weather. Better tires should be assigned higher values.

After every chromosome in a population has been given a fitness score, selection may occur. Typically, chromosomes are chosen with a probability proportionate to their fitness: this process is known as fitness proportionate selection. These chromosomes are the "parents" that will be used to create the next "generation" of solutions via crossover and mutation.

Parents (typically two) are selected by the aforementioned process for use in the crossover process. Crossover is the first of two "genetic operators" intended to replicate sexual reproduction. During crossover, parent chromosomes swap data, creating unique children. This process is represented by Figure 2.2.2.

A crossover point is randomly chosen and each "gene" beyond that point in the bit-string is swapped with its partner. As represented above, this process is known as "one-point crossover." Though the most closely related to its biological analog, one-point crossover is typically ignored in favor of "uniform crossover," in which

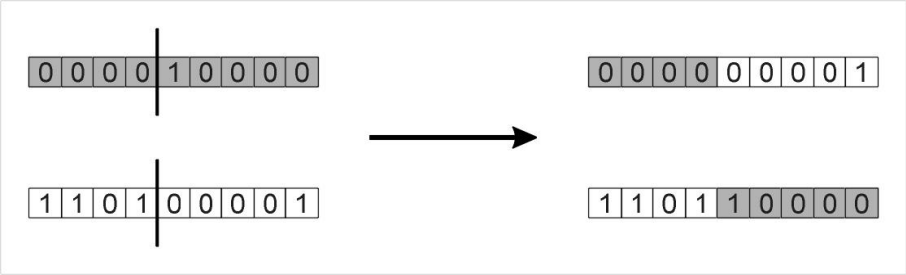

Figure 2.2.2. Genetic Crossover Applied to two Parent Chromosomes

each bit is evaluated individually and swapped according to a fixed probability. It is generally accepted that uniform crossover produces better results than one-point crossover [12].

The second genetic operator, "mutation," is applied to the "offspring" chromosomes produced in the previous step. In this step, each chromosome undergoes a probabilistic mutation procedure where one or more of its bits are "flipped" to a different value. For example, the chromosome 01101**1**10 might become 01101**0**10—a single bit has been selected and flipped. Mutation probability is defined before the GA is run, typically set to a low value. Setting this value too high will cause the GA to behave as a random search. However, it is important to have *some* chance of mutation. Mutation keeps solutions diverse, allowing the GA to avoid a "stagnation" scenario in which chromosomes homogenize and converge on a local minimum.

The steps of selection, crossover, and mutation are performed until the offspring chromosomes constitute a population of a set size. This population forms the next "generation" of the genetic algorithm, with a (hopefully) improved average fitness. The algorithm is run continually until a set termination condition is reached: typically either a generation limit or the discovery of a solution that satisfies minimum criteria.

2.2.2 Interactive Genetic Algorithms

The archetypal genetic algorithm utilizes a fitness function that can be computed efficiently and without human intervention. However, this is not always possible. In certain cases, the "form" of a fitness function may be unknown. This is most commonly encountered when investigating matters of "taste" or aesthetics: for example, determining the best design for a poster cannot easily be done in a computational manner. Design—and, by extension, art—is a subjective process, requiring human input to progress. Interactive genetic algorithms attempt to bridge this gap by either replacing or augmenting a fitness function with human evaluation.

Exploration of IGAs and their parent field, "interactive evolutionary computation," began shortly after the development of "standard" genetic algorithms. In 1986, Richard Dawkins described a computer program that models evolution by displaying a complex, two-dimensional shape (known as a "biomorph") based on a set of rules defining its geometry. Following these rules allowed the procedural generation of new shapes, one of which would then be selected by a user and used as a basis for further generations [13]. Figure 2.2.3 provides an example of one of these "biomorphs."

Dawkins' work was closely followed by the exploration of IGAs in a context more directly related to art. Utilizing a human preference-driven evaluation process, genetic algorithms were shown to generate intriguing and aesthetically pleasing computer graphic images, animations, and three-dimensional sculptures [14].

Interactive genetic algorithms come with unique considerations not required by a standard genetic algorithm. One of these considerations arises from the nature of human psychology: when a human user is unable to distinguish between different outputs, they will be regarded as the same from an optimization perspective. This

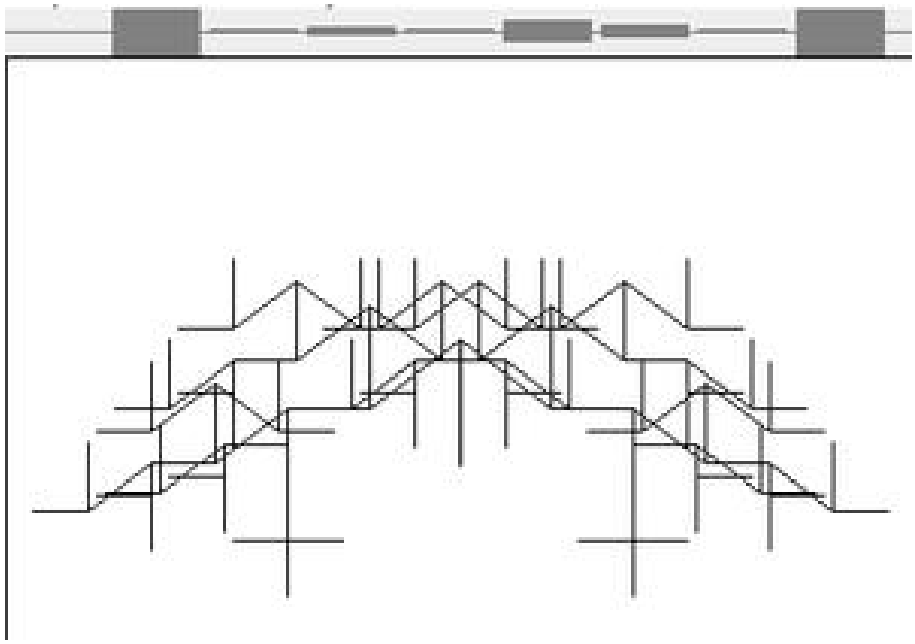

Figure 2.2.3. "Biomorph" after multiple stages of evolution

results in an algorithm that seeks an optimum "area" rather than a true global optimum "point." This is further compounded by the natural fluctuation in human preference: if an IGA has multiple evaluators, their differing tastes create a "fuzzier" optimization target [15]. However, this is not necessarily an issue. If an IGA is used to develop a user interface, it may be beneficial to provide a general impression of a good design rather than a singular optimum. As it is unlikely that a single best design exists, multiple good designs would have more utility in informing a designer's decisions and acting as a starting point for a final candidate.

A second consideration unique to IGAs is that of fatigue. From the perspective of a traditional genetic algorithm, humans make for a very slow and inefficient fitness function. This, combined with a natural limit on the number of solutions a single human can reasonably evaluate before experiencing fatigue, lends itself to a low number of search generations: research suggests an upper limit between 10 and 20 generations for most applications [15]. This necessitates experimental design deci-

sions focused on allowing convergence in this scenario. Again, many tasks involving IGAs do not need a large number of generations to converge: as the optimization target is an area, it is very possible that satisfactory results can be achieved well within the limits imposed by human fatigue.

3
Experiment and Methods

3.1 Page Design

Focal to this experiment was the creation of a generic "webpage" that the genetic algorithm would "evolve" over time in response to user feedback. It was considered important for the design to include a number of mutable design parameters that could manifest in a variety of visually appealing combinations while limiting the appearance of "broken" design elements, such as overflowing content or extreme text-wrapping.

3.1.1 Basic Template

In accordance with the design goals, the sample page that would be judged by visitors was envisioned as a hypothetical news website. The basic layout consists of a top-aligned navigation bar and a "content" section, presenting a grid of news articles that are themselves broken down into a photograph and a title. Figures 3.1.1 and 3.1.2 demonstrate two possible design permutations generable by the algorithm.

The "content" of the page, the news article photos and titles, are a random sampling of 12 from the New York Times Website [16]. In order to avoid the introduction

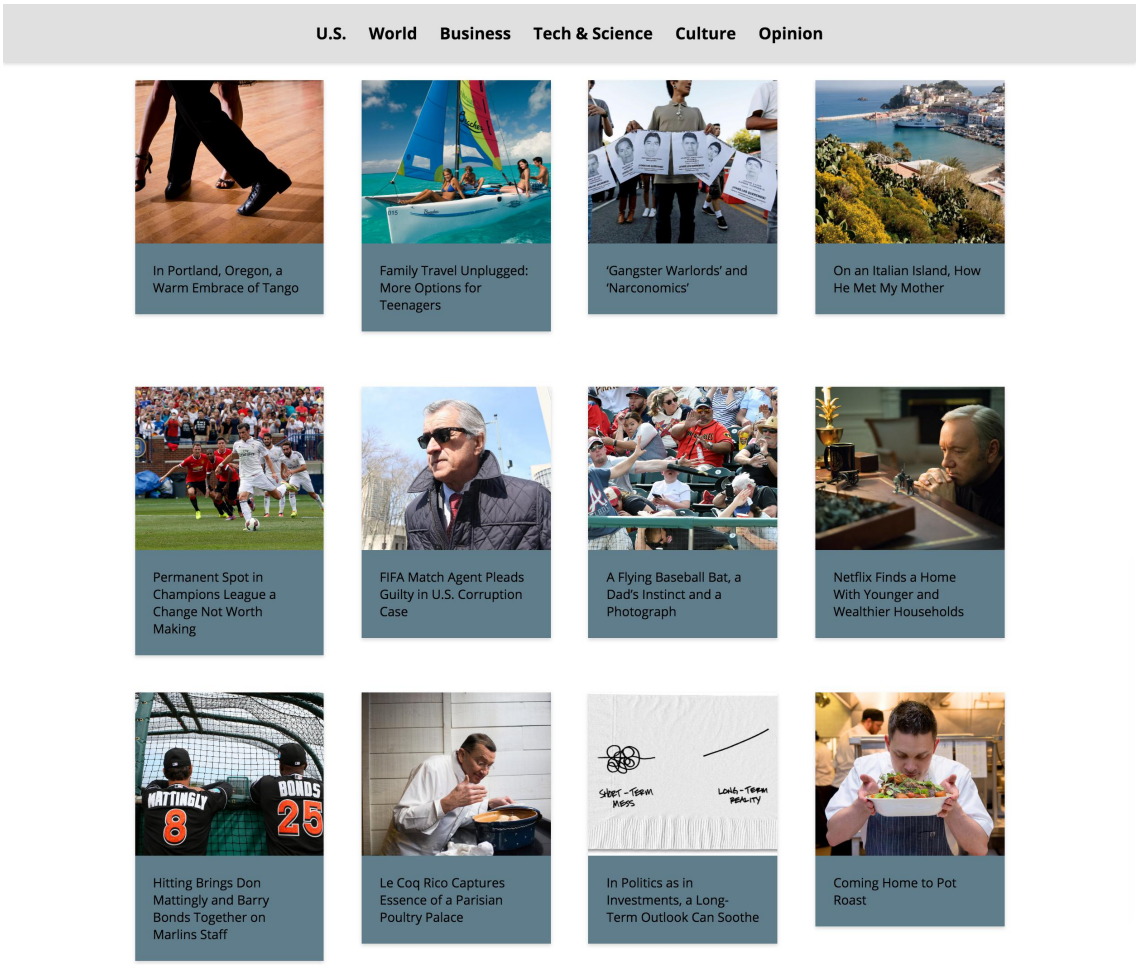

Figure 3.1.1. Possible template design permutation

of extraneous variables in the design evaluation process, the content did not change between layouts: each permutation displayed the same 12 articles. It is worth noting that none of the elements on the page are interactive: articles cannot be "opened" and the navigation bar cannot be used to navigate between sections. While a larger version of this experiment might incorporate multiple pages with interactivity, limitations of scale necessitated a focus on the design of a single page.

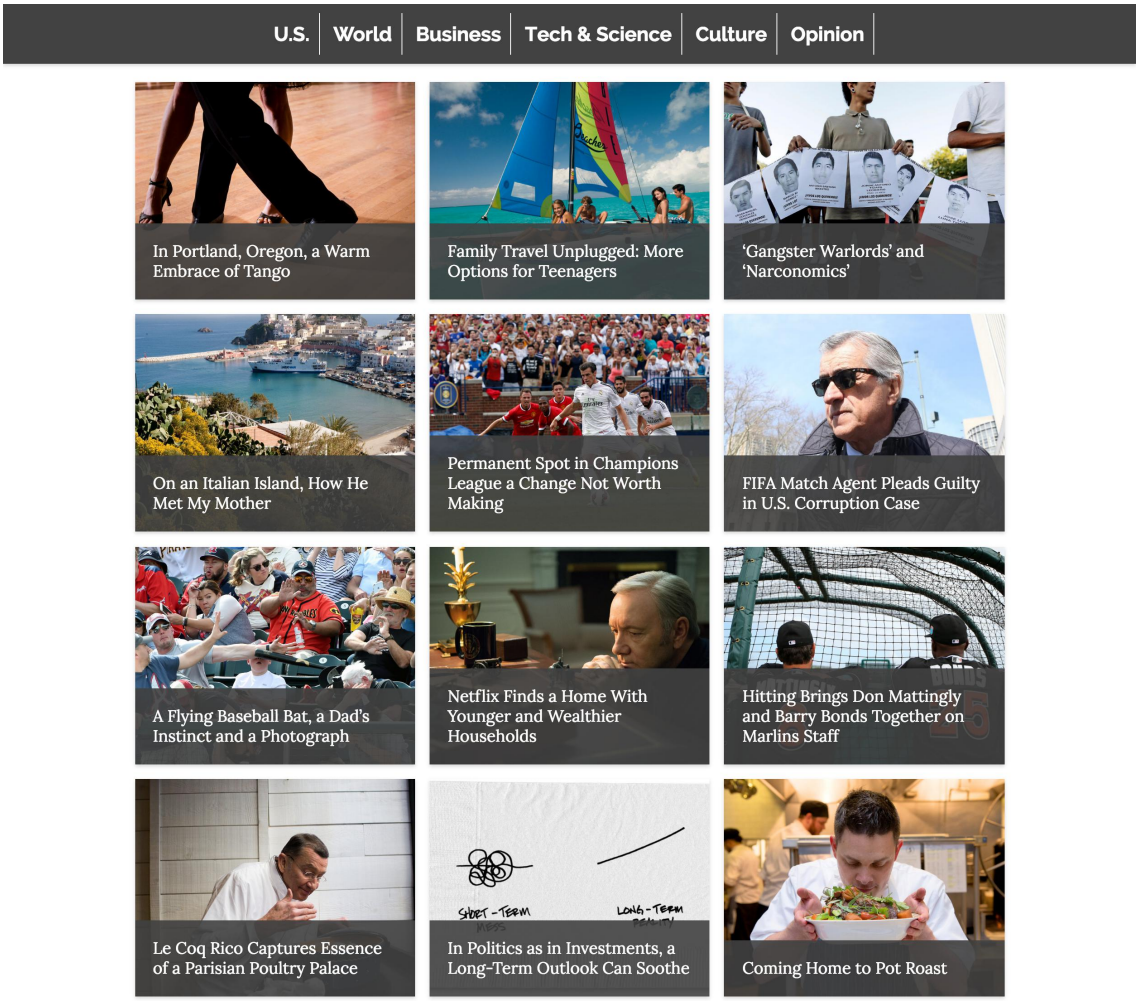

Figure 3.1.2. Alternate template design permutation

3.1.2 Mutable Elements

In total, there are 14 mutable elements that comprise each design. Table 3.1.1 details each parameter along with the possible values it can assume.

Generally, properties are divided between the navigation bar and the the article title boxes ("content"). The first two properties listed in table 3.1.1, "grid gutters" and "element width," refer to the parameters of the grid the articles are laid out on. The internal CSS grid system, discussed in more depth in section 3.4, is based on 12 hidden "columns." As defined by the "element width" property, each article

Property	Possible values
Grid gutters	1/4, 1/2, 1
Element width	3, 4, 6
Navigation background color	#607D8B, #C1362C, #FFFFFF, #E0E0E0, #424242, #99E0FF, #E37222
Navigation element borders	1px, 0px
Navigation bar font	Open Sans, Raleway, Slabo 27px, Lora
Navigation bar font size	14px, 18px, 22px
Navigation bar font weight	Normal, bold
Navigation bar shadow	True, false
Content background color	#607D8B, #C1362C, #FFFFFF, #E0E0E0, #424242, #99E0FF, #E37222
Content style	Overlap, normal
Content font	Open Sans, Raleway, Slabo 27px, Lora
Content font size	14px, 18px, 22px
Content font weight	Normal, bold
Content shadow	True, false

Table 3.1.1. Mutable design parameters and possible values

can occupy three, four, or six of these columns. As such, smaller element widths correspond to more articles per row: for example, figure 3.1.1 uses a width of three for each element, allowing four per row, while 3.1.2 uses a width of four, creating rows of three articles. A width of six would create rows of two articles. The "grid gutters" property refers to the space between grid elements, defined as a fraction of one of the internal columns.

The "navigation" properties are primarily associated with the color of the bar and its typography. The color can assume seven different values: a white, two shades of grey, and four non-grey colors. Typography variables include font, size, and weight. Fonts, split up into two serif and two sans-serif, were chosen from the most popular available in each category on the Google Fonts website [17]. Three possible values for font size were chosen: 14px, 18px, or 22px. The intention was to create design

variations that would be easily noticeable to the human eye. The "step" size, 4px, was considered to provide a balance of reasonable variety and differentiability. A step size of 2px might create typography that was too visually similar, whereas a larger step size would quickly reach unreasonably large values. Finally, fonts could assume either a normal or bold weight.

There are two additional properties affecting the navigation bar: "element borders" and "shadow." Element borders refer to a small separator placed between navigation bar categories. The values for this parameter, either 1 pixel or 0, refer to the thickness of the border (essentially corresponding to "on" or "off"). Shadow refers to a slight drop shadow drawn under the bar, encoded as a boolean value.

The "content" properties are largely the same as those affecting the navigation bar. Typography, background color, and drop shadow parameters are identical. The only other property, "style," refers to the positioning of the content box in the article element. A value of "normal" would cause the box to be placed underneath the photo, whereas a value of "overlap" would shift the box into a bottom-aligned overlapping position that partially covered the article photo. Additionally, the background color is altered to be slightly transparent. Figure 3.1.2 demonstrates this property set to the "overlap" value.

3.2 Outline of Software

The software is broken down into two parts: the client and server. The server is dedicated to interfacing with the database, serving pages to the client, and running the genetic algorithm when necessary to create new generations. The client, running in a browser, provides an interface for a user to request pages maintained by the server and vote for their preferred design.

The majority of the code is written in JavaScript. Traditionally the "language of the web," due to its inclusion in every browser [18], writing server-side code in JavaScript has become possible with the advent of Node.js, a standalone JavaScript runtime intended for network applications [19]. 33 different open-source libraries and utilities were used in the development of the software. While many of these are outside the scope of this paper, some will be discussed in the sections pertaining to their use.

Presentational components of the client-side software are written in HTML and CSS, the latter of which is explained in detail in section 2.1. It is of note that the dynamic layout acted on by the genetic algorithm does not use "vanilla" CSS, instead relying on Sass, a "CSS extension language" with support for variables, functions, and other complex features that build on CSS functionality [20]. The layout discussed in section 3.1.1 utilizes Susy, a Sass library intended to simplify the creation of grid layouts [21]. The complex Sass stylesheet that bridges the gap between genetic algorithm and CSS will be discussed in section 3.4.

Finally, the database software selected is RethinkDB, an open-source "NoSQL" database chosen for its complex built-in administrative interface and real-time analytics capabilities that permitted comprehensive monitoring of the experiment in-progress [22]. The object-relational mapper library Thinky acts as an interface between the Node.js server and database [23].

3.3 Server

3.3.1 General Details

The server software comprises the majority of this project's code complexity. Its primary functionality is twofold: exposing an API interface for the client-side software and managing the population of style chromosomes in the database. The overall

structure of the server is based on the "model-view-controller (MVC) architecture common in web development. In accordance with this pattern, all the data-handling logic and database interfacing is abstracted into separate "models." For this project, three models were created to encapsulate core functionality: "style," "token, and "visitor."

The "style" model contains all the functionality for managing the chromosome population. This includes selection of individuals for evaluation (when requested by the client), handling the receipt of votes for a given individual, summing the fitness of a population, and creating a new generation when a minimum fitness threshold has been reached.

Selection is a non-trivial process. To ensure good performance of the algorithm, it was important that each individual in a generation be viewed roughly the same number of times while still being presented in a random order. Simply selecting individuals in a "round-robin" style fashion (where order is maintained) leads to a condition in which an individual can only be compared to its "adjacent" members in the database. Considering the relative nature of the human-driven fitness function, it was desirable to display as many different combinations of individuals to users as possible, ensuring a wide range of comparisons take place over the lifetime of a generation. To this end, a semi-random solution to selection was devised.

Each individual in a generation maintains a "visited" counter (more information is available in section 3.3.2) that increments whenever it is selected for evaluation. The selection algorithm begins by determining the minimum visited count in the current generation, then filtering out all individuals with a count greater than that value. From this subset of the generation, an individual is randomly chosen, incremented, and sent to the client. However, this gives rise to an edge case in which the minimum-visited subset contains only a single individual. After this individual is se-

lected, the subset necessarily broadens to include every individual in the generation (as they now all share the same visited count). In this scenario, a client that has requested an individual from the previous "round" may randomly re-select an individual that it is already displaying. To avoid showing duplicates, the client sends a list of its currently-maintained individuals with each request: these individuals are then temporarily pre-filtered out of the subset so they cannot be chosen in the random selection process.

The remaining functionality contained by the "style" model is relatively simplistic. Receipt of a vote triggers a database operation that increments the fitness value of the specified individual, followed by a summation operation to determine the overall fitness of a population. When a minimum fitness threshold is reached, a function responsible for creating a new generation is run. The function is discussed in more detail in section 3.4.

The "token" model serves an important function in maintaining client state and preventing malicious tampering of results. When a user begins a new voting session, the server creates and sends a "JSON Web Token" that the client stores and sends with every further request to the server [24]. The token is hashed with a "secret key" that is known only to the server. As such, clients only receive and store an indecipherable string that cannot be altered: any modification will cause the server-side decryption mechanism (using the aforementioned key) to reject the token as invalid. Through this method, all client-side actions are validated by the server, obviating attempts to "cheat" by voting early, submitting a vote for an individual not requested by the client, or any other invalid action.

The token contains all information about client state: email address, the list of all style individuals currently being viewed (referenced by a unique ID), and the number of requests for individuals that have been made so far. The number of

requests is used to validate and enable the "vote" button on the client: to help ensure experimental integrity, the button does not enable until all three possible designs have been viewed. When the server decrypts the token, the number of requests made is counted and compared against a preset value: if the values are the same, the client is determined to have made the correct number of requests and allowed to enable the "vote" button.

Finally, the "visitor" model is responsible for tracking user participation. If an email address is submitted by the client at the beginning of the survey process, a new entry is created in the database if that address does not already have a corresponding entry. The only additional information tracked is the number of votes each user has accrued: when a vote is submitted, the token attached to the request is decrypted and the email it contains is used to reference the relevant database entry and have its "voted" field incremented.

The second component of the MVC pattern is the "view." The purpose of the view is, generally, to provide a visual representation of a model to a human user and provide controls for interacting with that model. In this project, the view takes the form of a sophisticated JavaScript application that is sent to a user and loaded in their browser. The details of this application are discussed in section 3.5.

The third component of the MVC pattern, the "controller," provides a bridge between models and the view. More specifically, the controller receives user input from the view, calls the necessary functions in the models, and transmits a response back to the view. Little if any complex logic occurs in the controller beyond this coordinating process.

3.3.2 Database Design

The database provides persistent storage of all the data handled by the application. Each of the models discussed in 3.3.1 has a corresponding table in the database. It is of note that the software chosen for this application, RethinkDB, does not use the standard SQL data types. As a "NoSQL" database, it stores documents in "JSON," a data-interchange format that allows a range of data types, including arrays and complex nested objects [25]. The structures of each database table utilized in this project are described in figures 3.3.1, 3.3.2, and 3.3.3. Each table has two common fields: "id" and "createdOn." The "id" field is an alphanumeric string that uniquely identifies each document. The "createdOn" field is a date value that is automatically set to the current date when the document is inserted into the table.

Field name	Type	Default
id	String	
generation	Integer	
styles	Object	
fitness	Integer	1
Parents	Array - String	
Visited	Integer	0
createdOn	Date	Current date

Table 3.3.1. Schema for "Style" table

Table 3.3.1 represents the structure of the "Style" table. This table maintains the populations of style chromosomes that will be evaluated by users and acted on by the genetic algorithm.

The "generation" field refers to the generation a given individual is a part of. It is of note that this table contains every individual from *every* generation, not just the current generation: this allows for easy analysis of styles over time during the

post-survey data collection phase. However, all of the queries the server runs when the survey is in process filter out all individuals in previous generations.

The "styles" field holds a JSON object containing the style chromosome represented by the individual. The key-value pairs within are discussed in section 3.1.2. This object is the focus of the genetic algorithm: when a new generation is created, it is this data that is affected by the crossover and mutation operations.

The "fitness" field maintains a count of the votes accrued by the individual, used as the overall measure of the represented design's quality.

The "parents" field is an array containing the IDs referencing the individual's genetic "parents:" the chromosomes that were selected for the crossover operation that provided the style parameters for this individual. This array is empty in all members of the first generation: as they are randomly generated, no parents exist. This field allows a given design to be traced backwards, providing a better understanding of the parent designs that gave rise to it.

Finally, the "visited" field stores the number of times the individual has been selected for evaluation. This is critical in ensuring an even selection of individuals in a given generation. The selection method is discussed in section 3.3.1.

Field name	Type	Default
id	String	
email	String	
numReqs	Integer	0
htmlIDs	Array - String	
createdOn	Date	Current date

Table 3.3.2. Schema for "Token" table

Table 3.3.2 represents the structure of the "Token" table. This table is responsible for tracking the existence and maintaining the decoded states of all tokens currently in use by clients. This table's most significant role is in vote validation. If tokens are

not stored in a table, it is impossible to determine if a given token has already been "used," potentially allowing the same request to be sent to the server and processed multiple times. To prevent this, the token associated with a request is checked for request in the database. If the token exists, the vote is processed and the token is removed from the database. If the token does not exist (e.g. a vote request has already been processed), the request is considered to be invalid and is rejected.

The "email" field contains the email associated with the token. This email is used to update the relevant user information whenever a vote is cast. If a participant does not submit an email, this field is blank.

The "numReqs" field tracks the number of requests made during a given session. This is used to validate client state before a vote is allowed to be processed. As defined by experimental parameters, each user must view a specific number of designs before a selection can be made and a vote submitted. Every time a request is made for a new style, the "numReqs" counter is incremented. When a vote request is received, this value is verified to be at the specified minimum and the vote is processed.

The "htmlIDs" field maintains an array of IDs referencing documents in the "Style" table. This array tracks the styles currently being viewed in a given session. This data is primarily used to prevent duplicate styles being selected to send to a client.

Field name	Type	Default
id	String	
email	String	
votes	Integer	0
createdOn	Date	Current date

Table 3.3.3. Schema for "Visitor" table

Table 3.3.3 represents the structure of the "Visitor" table. This table stores data pertaining to study participants. If a user specifies an email address before beginning

the survey, an entry is created in this table that tracks the number of votes they submit and associates it with their email address. This information is used only for a rough profiling of study participation statistics and in the post-study reward draw.

3.4 GA Implementation

3.4.1 General Details

The genetic algorithm implemented for this project is written in JavaScript (seamlessly integrated with the Node.js server) based on object-oriented principles. It is of note that a non-traditional chromosome representation was chosen for this experiment. In most cases, chromosomes are encoded as "bitstrings" [11]. However, in the interest of human readability and in consideration for the complex values encoded within, the chromosome representation described in this section maintains a list of "gene" objects that store a value (along with all the possible values that gene can express) in lieu of a classic binary representation. This has the added benefit of allowing for easy changes to the chromosome "schema": adding a new property or extending existing ones is a trivial matter. Figure 3.4.1 shows the code for "gene.js" in its entirety.

The gene object is relatively simple. The constructor method (run every time a gene object is created), on lines 2-12, takes two arguments: "possibleValues" and "value," which defaults to null. PossibleValues is an array storing all the values the gene may assume, while value is an optional argument that sets the gene to a specific value on creation. If this argument is not provided, the "value" member variable is set to a random item from the possibleValues array. Lines 14-16 define a generic "getter" that returns the current value of the gene.

Additionally, the gene object includes a "mutate" method on lines 18-25 that performs the genetic mutation operation. This method takes a single argument,

```
export default class Gene {
    constructor(possibleValues, value = null) {
        this.possibleValues = possibleValues;

        if(value) {
            this.value = value;
        } else {
            let idx = Math.floor(Math.random() *
                this.possibleValues.length)
            this.value = possibleValues[idx];
        }
    }

    getValue() {
        return this.value;
    }

    mutate(p = 0.02) {
        if(Math.random() < p) {
            let newPossibles = this.possibleValues.filter(
                val => (val != this.value));
            this.value = newPossibles[Math.floor(Math.random() *
                newPossibles.length)];
        }
    }
}
```

Figure 3.4.1. Code in "gene.js"

"p," that defines the likelihood of a mutation taking place. A random number is generated and compared to p: if it is below the threshold, a mutation occurs. During a mutation, a copy of the possibleValues array is created that does not include the gene's current value, from which a new value is randomly selected. This filtering process prevents a mutated gene from manifesting the same value.

The "Chromosome" object maintains a list of Gene objects and sets their possible values. Figures 3.4.2, 3.4.3, 3.4.4 and 3.4.5 show "chromosome.js" in its entirety.

Figure 3.4.2 shows the "constructor" method for the chromosome class, which is run whenever a new chromosome is created. The constructor takes a single optional argument "vals," which is an object containing values that will be used to pre-set the chromosome's constituent genes. The chromosome class has a single member variable: a "genes" object (in this case, essentially an associative array) that maintains a number of key-value pairs referencing "gene" objects corresponding to the properties discussed in section 3.1.2. The possible values each parameter can assume are hard-coded in. For example, the "gridgutters" member variable on line 6 is set to a gene object which itself takes an array corresponding to these values: "1/4", "1/2", and "1". The second argument passed to the gene object, used for setting a specific value, is the relevant key in the vals argument.

The code in figure 3.4.2 defines a method, "getObjectRepresentation," that translates a chromosome into a simplistic JavaScript object of key-value pairs. This method is called when a chromosome needs to be stored in the database: only the property names and their current values are required.

The code in figure 3.4.4 shows the code responsible for the genetic crossover operator, encapsulated in the static method "uniformCrossover." This method takes three arguments: two chromosomes to be operated on and a crossover probability. Essentially, a new chromosome is built by iterating through each pair of genes in the parent chromosomes and randomly selecting either the value of the gene from "cr1" or "cr2," based on the probability defined by "p."

The code shown in figure 3.4.5 defines the "mutate" method on the chromosome. The purpose of this method is fairly straightforward: when called, each gene in the chromosome will have its "mutate" method called. In essence, the chromosome-level mutation method provides a shortcut for calling mutate on all of its constituent

genes. The single parameter "p," defining the mutation chance, is passed down to the gene-level mutate method.

The "selection" step of the genetic algorithm is encapsulated in the "style" model discussed in section 3.3.1. The code for this process is shown in figure 3.4.6.

This method, "rouletteSelection," implements a standard fitness proportionate selection algorithm. In FPS, parent chromosomes are repeatedly chosen with a probability proportionate to their fitness and used to create offspring chromosomes [11]. Two arguments are given when this method is called: an array of all the members of a parent population, "pop," and a desired number of individuals in the to-be-generated offspring population, "num." Initially, an array containing only the fitness values of individuals in the parent population is created and the total fitness is stored in a variable. After a random number is generated (capped at the total fitness), the "fitnesses" array is traversed, subtracting the value at the current index from the stored random number. If this subtraction process reduces the number to 0, the individual referenced by the current index has been selected. This process is repeated, selecting pairs of individuals and performing crossover until an offspring population of the requisite size has been created.

```
import Gene from './gene';

export default class Chromosome {
    constructor(vals = {}) {
        this.genes = {
            gridgutters: new Gene(['1/4', '1/2', '1'],
                vals.gridgutters),
            elementwidth: new Gene([3, 4, 6],
                vals.elementwidth),
            navbgcolor: new Gene(['#607D8B', '#C1362C', ...],
                vals.navbgcolor),
            navelementborders: new Gene(['1px', '0px'],
                vals.navelementborders),
            navfont: new Gene(['Open Sans', 'Raleway', ...],
                vals.navfont),
            navfontsize: new Gene(['14px', '18px', '22px'],
                vals.navfontsize),
            navfontweight: new Gene(['normal', 'bold'],
                vals.navfontweight),
            navshadow: new Gene(['true', 'false'],
                vals.navshadow),
            contentbgcolor: new Gene(['#607D8B', '#C1362C', ...],
                vals.contentbgcolor),
            contentstyle: new Gene(['overlap', 'normal'],
                vals.contentstyle),
            contentfont: new Gene(['Open Sans', 'Raleway', ...],
                vals.contentfont),
            contentfontsize: new Gene(['14px', '18px', '22px'],
                vals.contentfontsize),
            contentfontweight: new Gene(['normal', 'bold'],
                vals.contentfontweight),
            contentshadow: new Gene(['true', 'false'],
                vals.contentshadow)
        };
    }
```

Figure 3.4.2. Constructor method in "chromosome.js"

```
getObjectRepresentation() {
    let obj = {};
    Object.keys(this.genes).forEach(name => {
        return obj[name] = this.genes[name].getValue();
    });
    return obj;
}
```

Figure 3.4.3. getObjectRepresentation method in "chromosome.js"

```
static uniformCrossover(cr1, cr2, p = 0.5) {
    let newProps = {};

    Object.keys(cr1.genes).forEach(name => {
        newProps[name] = (Math.random() < p ? cr1.genes[name] :
            cr2.genes[name]).getValue();
    });

    return new Chromosome(newProps);
}
```

Figure 3.4.4. UniformCrossover method in "chromosome.js"

```
mutate(p = 0.02) {
    let newChromosome = new Chromosome(this.getObjectRepresentation());

    Object.keys(newChromosome.genes).forEach(name => {
        newChromosome.genes[name].mutate(p);
    });

    return newChromosome;
}
```

Figure 3.4.5. Mutate method in "chromosome.js"

```
function rouletteSelection(pop, num) {
    const fitnesses = pop.map(obj => obj.fitness);
    const totalFitness = fitnesses.reduce((total, current) => {
        return total + current;
    });

    let indices = [];

    while(num--) {
        let randVal = Math.floor(Math.random() * totalFitness)
        for(let idx = 0; idx < fitnesses.length; idx++) {
            randVal -= fitnesses[idx];
            if(randVal <= 0) {
                indices.push(idx);
                break;
            }
        }
    }

    return indices;
}
```

Figure 3.4.6. Roulette selection code

3.4.2 Mapping GA to CSS

Given that the target domain of the genetic algorithm in this experiment is a CSS file that can be received and processed by a web browser, additional work must be done in translating the chromosome (represented as a JavaScript object) to valid CSS. Additional functionality is provided by "Sass," mentioned briefly in section 3.2. Sass provides support for variables, conditional expressions, functions, and more, while compiling into "vanilla" CSS. Variables, in particular, are heavily used in this project. Figure 3.4.7 demonstrates the use of variables in Sass code.

```
1 $red: #FF0000;
2 $fontsize: 24px;
3
4 div {
5     background-color: $red;
6     font-size: $fontsize;
7 }
```

Figure 3.4.7. Sass variable usage

The first two lines of the code snippet declare two variables: "red," which stores a hex color, and "fontsize," which stores a px value. These variables are then used within the "div" selector, where the compile step will replace all instances of variables with the associated value.

When a client requests a new design for evaluation, the chromosome-to-CSS translation process begins with the programmatic generation of a Sass "snippet" containing variables representing the key-value pairs contained within the chromosome object. The code responsible for this conversion is shown in figure 3.4.8.

The "genVariables" method takes a JavaScript object as an argument and then iterates through each key-value pair, performing the necessary string formatting and

```
1 function genVariables(vars) {
2     return Object.keys(vars).map(name => {
3         return '$' + name + ': ' + vars[name] + ';';
4     }).join('\n')
5 }
```

Figure 3.4.8. JS-to-Sass variable code

concatenation operations to convert it into a Sass variable before inserting a newline character. For example, the JavaScript object:

```
{
    key1: "value1",
    key2: "value2"
}
```

Would be converted into the following Sass code:

```
$key1: value1;
$key2: value2;
```

This Sass snippet is then prepended to the front of a larger Sass file that contains the styling rules for the dynamic page that will be evaluated by the client. Though the entire file is too large to include in this paper, an example rule is shown in figure 3.4.9.

This set of nested rules defines the style of the navigation bar. There are a number of variables in use: any value beginning with a "$" is a Sass variable that will be replaced with the "actual" CSS value during the compilation step. For example, the "background" property on line 2 will receive one of the 7 possible colors defined by the chromosome schema. Of note in this snippet is the custom "calc-font-color" Sass function in use on lines 19 and 24, the code for which is shown in figure 3.4.10.

This function determines the ideal color for a font printed over a background of the color passed in an argument. If the given color is either transparent or beyond an experimentally-determined lightness threshold, the function will return the hex value for black: "#000000." Otherwise, the hex value for white will be returned: "#FFFFFF." This function reduces overall code complexity by obviating the need to store a paired font color value for each color defined in the chromosome schema.

After the Sass variable snippet and main file are combined, the resultant string is compiled into a standard CSS file. In the interest of simplifying client-side code, an additional step is taken: the HTML file defining the structure of the dynamic page (discussed in section 3.1.1 is loaded and the CSS file is "inlined," producing an HTML string that has styles defined internally. In this way, the client need only make a single request for HTML with style information included, rather than two separate requests for HTML and CSS.

```
.nav-container {
  background: $navbgcolor;
  text-align: center;
  padding: 10px;
  margin-bottom: 20px;

  @if $navshadow == true {
    @extend .shadow;
  }

  .nav {
    display: inline-block;
    list-style: none;
    padding: 0;
    margin: 0;
    font-family: $navfont;
    font-size: $navfontsize;
    font-weight: $navfontweight;
    color: calc-font-color($navbgcolor);

    li {
      display: inline-block;
      padding: 10px;
      border-right:
        $navelementborders solid calc-font-color($navbgcolor);
    }
  }
}
```

Figure 3.4.9. Sass rules for navigation bar

```
@function calc-font-color($color) {
  $_computedcolor: #FFFFFF;

  @if opacity($color) == 0 or (lightness($color) / 100% > 0.26) {
    $_computedcolor: #000000;
  }

  @return $_computedcolor;
}
```

Figure 3.4.10. “calc-font-color” function code

3.4.3 Fitness Function

In this experiment, the fitness of individuals is determined by the number of "votes" given to them by human evaluators. In the interest of minimizing fatigue, a "relative" rather than "absolute" voting method was used. It was believed to be easier to select a favorite design among a small set of possibilities than independently assign a rating to a single individual (i.e. on a 1-10 scale). Indeed, past work with IGAs suggests that a simple "Boolean" rating system over small numbers of individuals leads to reduced fatigue [27] [28].

This leads to an evaluation process where a user will request three possible designs (selected semi-randomly, detailed in section 3.3.1 before submitting a vote for their favorite among the three. This individual's fitness is incremented by 1 while the other two individuals are left unmodified. A new set of individuals is made available for display and the process repeats until the user decides to stop.

3.4.4 GA Parameters

Important to any successful use of GAs is the selection of ideal algorithm parameters: crossover rate, mutation rate, population size, and termination criteria. While it is generally accepted that there is no universal "best" set of parameters, typical starting points suggest a crossover rate of 0.5, a mutation rate of 0.02, a population size of 50 (or more, depending on search space size), and a terminating criteria that allows for many hundreds of generations [26].

While these numbers may provide good performance in traditional GAs, interactive GAs necessarily require additional consideration when selecting parameters. The speed and realistic quantity of human evaluation creates a significant limitation in scale: whereas a computational fitness function is likely to be capable of evaluating hundreds of individuals a second, humans may only be able to evaluate a few

individuals per minute. However, this is offset by the inherent "fuzziness" in an IGA optimization target, which suggests that satisfactory solutions may be reached in far fewer generations [15].

For this experiment, the following parameters were chosen: 0.5 crossover rate, 0.02 mutation rate *per gene*, and a population size of 20 individuals. No termination criterion was set, with the intention that the experiment would be run until user participation began to drop off. Additionally, the round-robin style voting system "fitness function" requires an inter-generational termination condition to define a point at which a new generation is created. This was chosen as a total fitness of 300 across all individuals in a population, allowing a large number of evaluations to occur before selection. The 0.5 crossover rate implies an even mixing of genes among parents: it was not considered necessary to favor one parent of another.

The 0.02 mutation rate is a per *gene* value, not per *chromosome*. This means that the actual probability a chromosome (containing 14 genes) will undergo at least one mutation is quite high: roughly 0.25. This value was considered reasonable based on the small size of the experiment and the belief that under-represented features in the initial population could, through "bad luck," be evolved out of the population. A high mutation rate would occasionally re-introduce "forgotten" features that might be more appealing in newer designs.

The population size of 20 is considerably lower than a standard GA. Again, this decision is based primarily on the scale limitations imposed by IGAs. A population size that was too large would have a "spreading" effect on the fixed quantity of votes distributed through a generation, potentially reducing the search performance by "hiding" more fit individuals amongst their less fit peers.

3.5 Client-side app

The second software component of this experiment is the client-side application running in the browser. This application provides an interface for a user to submit an optional identifying email address, view designs for evaluation, and vote on a favorite. Hosted on a publicly-accessible server, visiting the site presents a form requesting an email and providing consent information and instructions. Figure 3.5.1 illustrates the initial "gate" form.

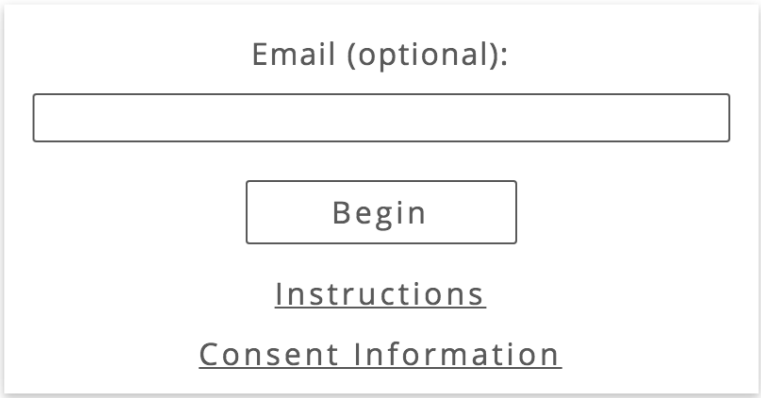

Figure 3.5.1. Initial client application "gate"

Pressing "Begin" requests a session token from the server along with the raw HTML data of the first design to be evaluated. The design is then presented as illustrated in figure 3.5.2. A minimalistic interface with muted colors was deliberately chosen to focus users on the evaluation task without extraneous features affecting judgment.

The "Previous" and "Next" buttons at the top of the page are used to navigate between a set of three designs. The application maintains a cache of pages so unnecessary load is not placed on the server: a request for a new page is only made

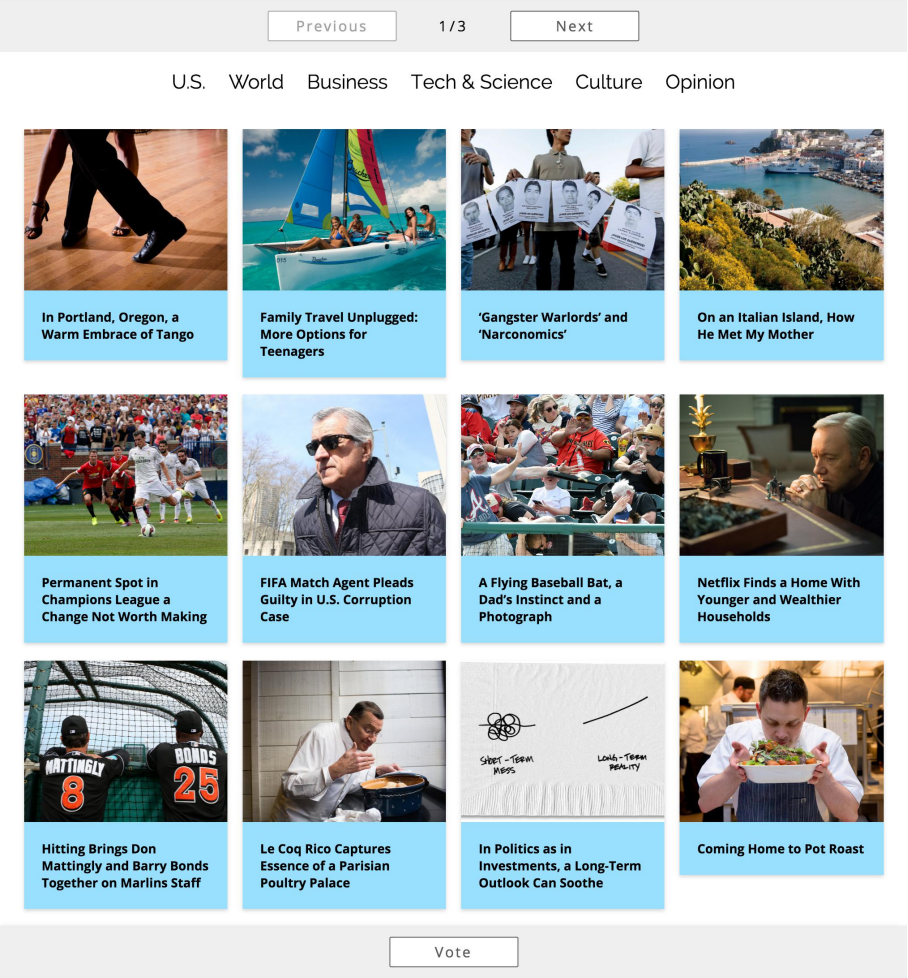

Figure 3.5.2. Client application page carousel view

when "Next" is pressed and the cache is empty. When all three designs are viewed, the "Vote" button enables, allowing the user to navigate to their preferred design and cast a vote. A new session automatically begins by clearing the page cache and requesting a fresh design from the server. The client also stores and attaches the session token to each request, allowing for server-side validation (details in section 3.3.1). The user may choose to participate in as many voting cycles as they wish.

3.6 Experimental Setup

For this experiment to be successful, a large number of votes needed to be collected. Hosting the experiment website at a public address allowed for easy participation: anyone with access to an internet-connected device was able to evaluate designs and submit votes. Interest was generated through Bard College-centric social media and word-of-mouth, along with a $50 Amazon.com Gift Card reward draw done at the conclusion of the study.

The experiment was planned and designed to be monitored over the entirety of its run to ensure the software and genetic algorithm performed without error. The decision to terminate the study would be made after a significant decrease in user participation, at which point the database would be downloaded for local analysis and the server shut down.

4
Results

4.1 Feature Representation Graphs

The dataset collected and presented in this section attempts to demonstrate convergence on a specific design phenotype (or phenotypes) by plotting the number of individuals in each generation that express a given feature. 14 graphs display the frequency, from 0 through 20, of every design parameter and its possible values over 10 generations.

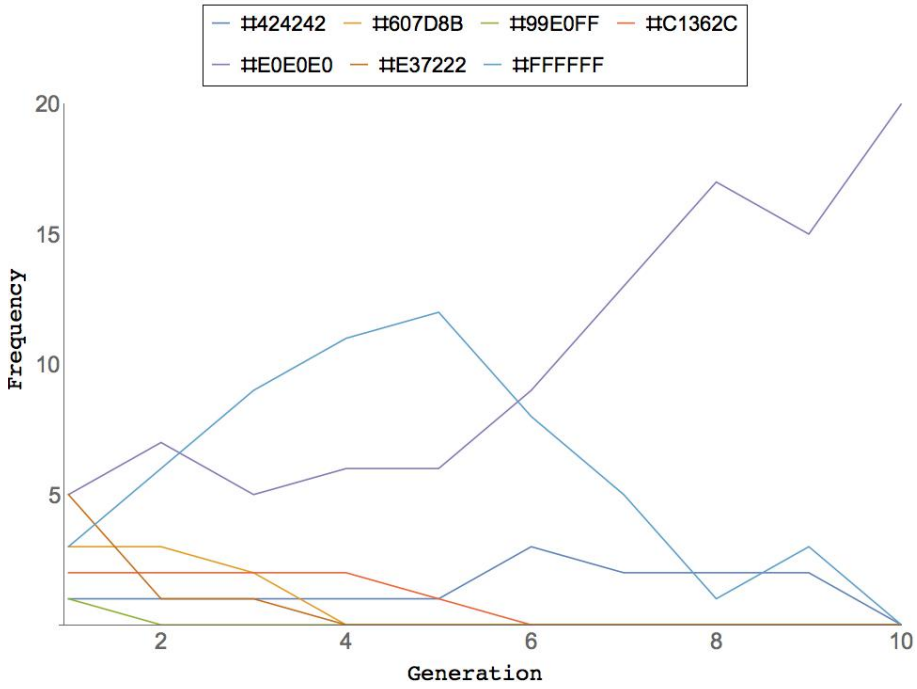

Figure 4.1.1. Frequency of content background color values over 10 generations

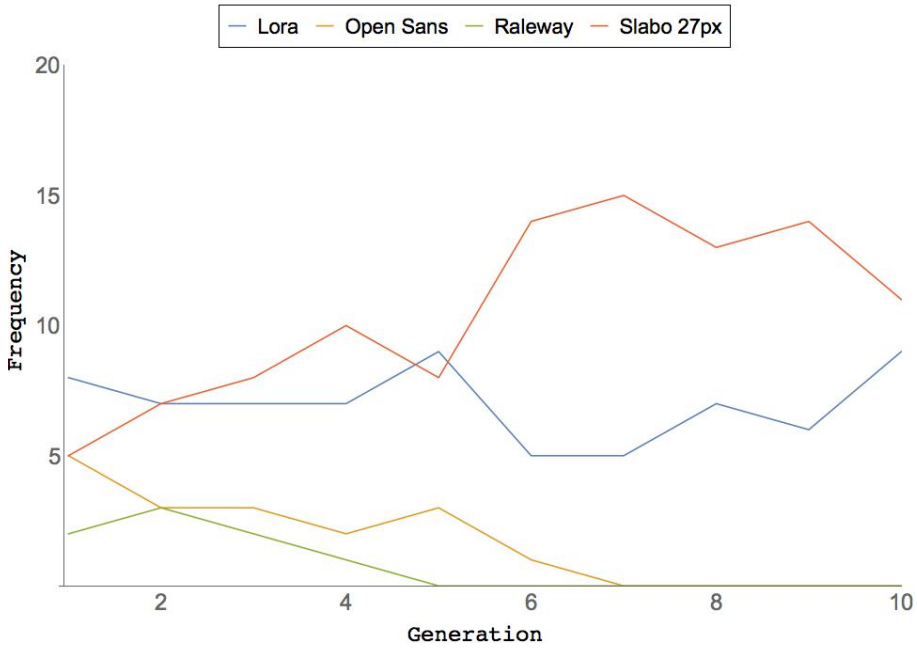

Figure 4.1.2. Frequency of content font values over 10 generations

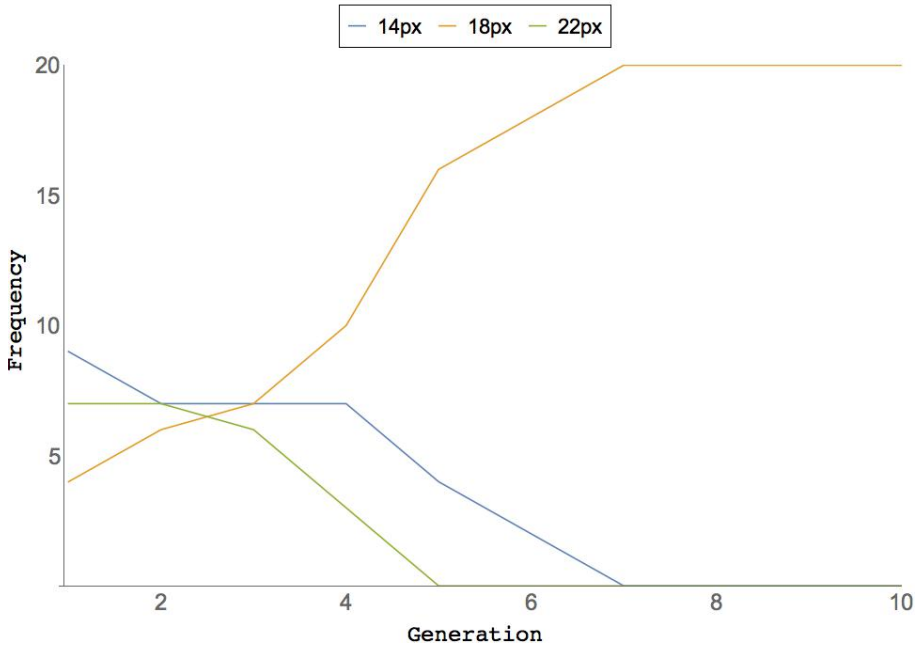

Figure 4.1.3. Frequency of content font size values over 10 generations

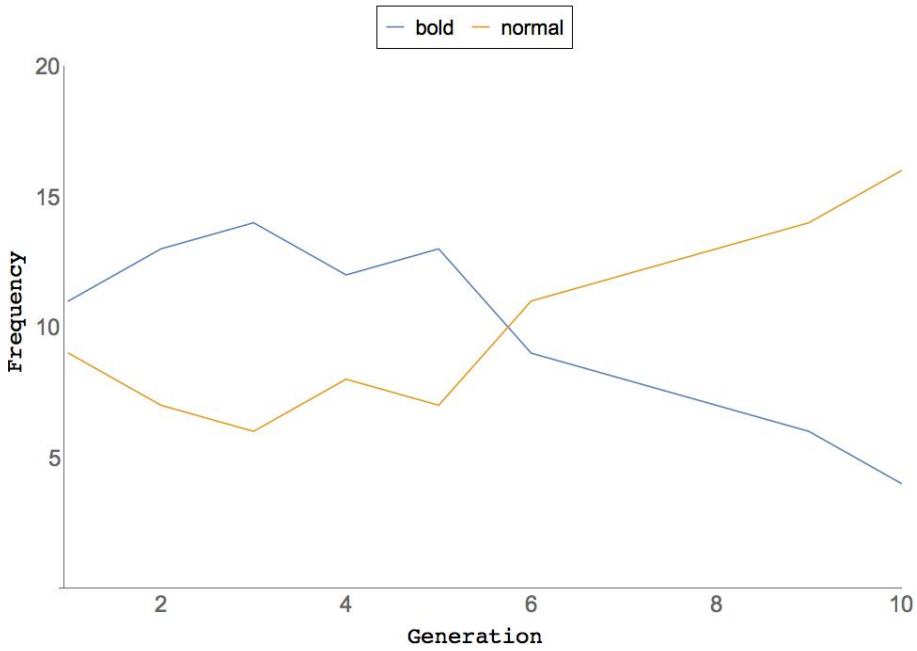

Figure 4.1.4. Frequency of content font weight values over 10 generations

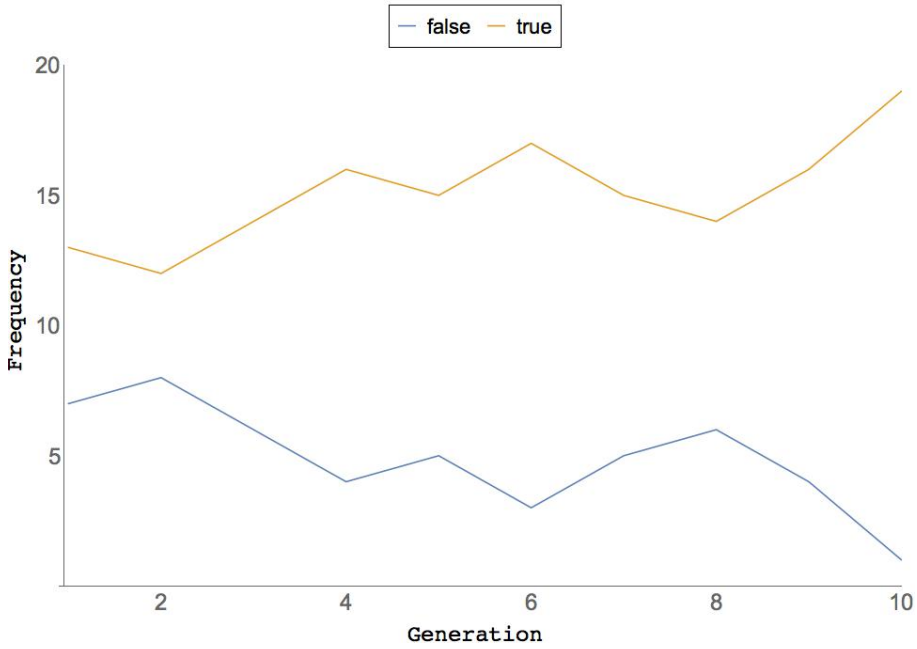

Figure 4.1.5. Frequency of content shadow values over 10 generations

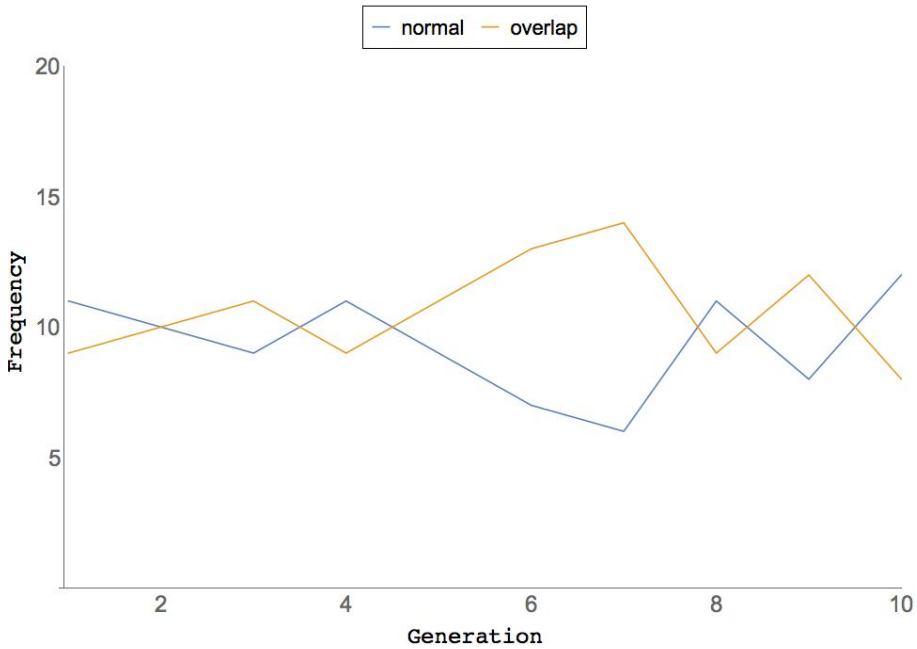

Figure 4.1.6. Frequency of content style values over 10 generations

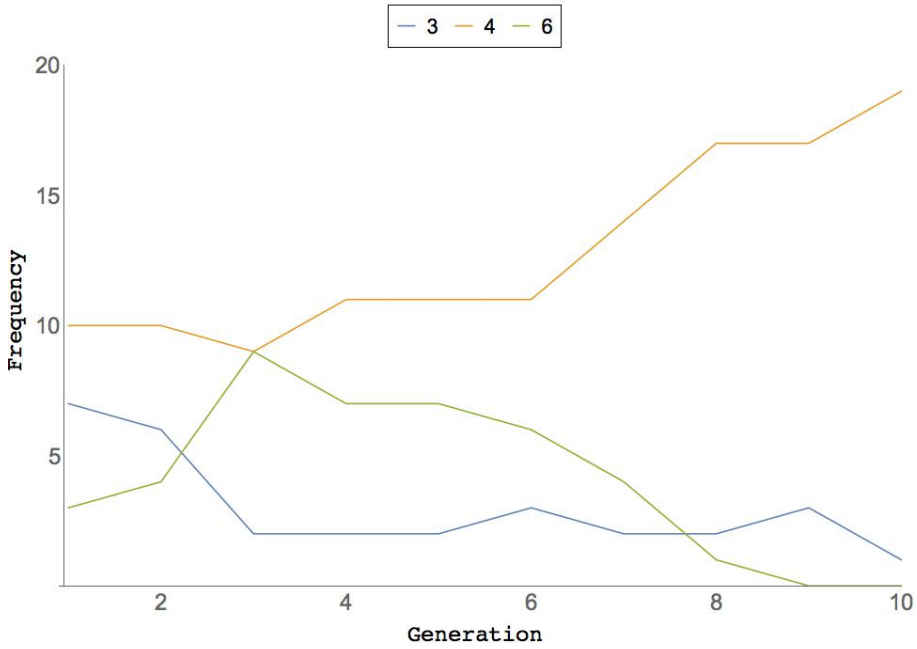

Figure 4.1.7. Frequency of element width values over 10 generations

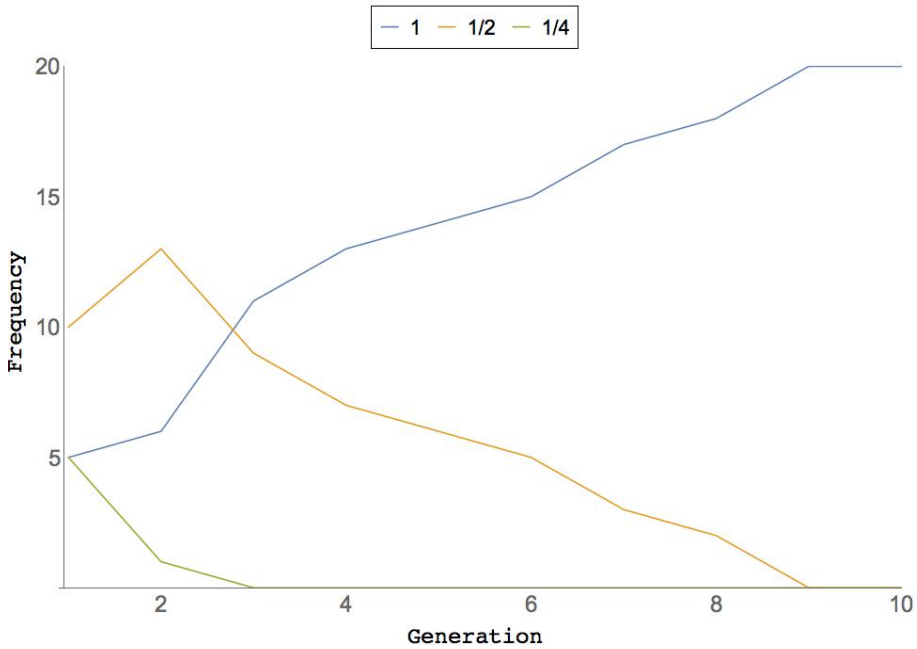

Figure 4.1.8. Frequency of grid gutter values over 10 generations

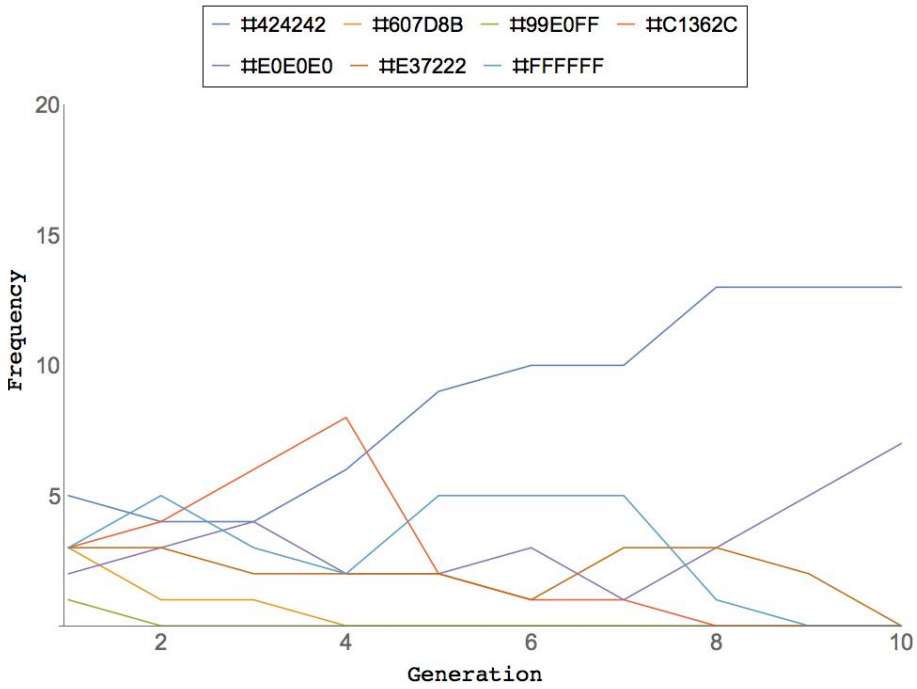

Figure 4.1.9. Frequency of navigation bar background color values over 10 generations

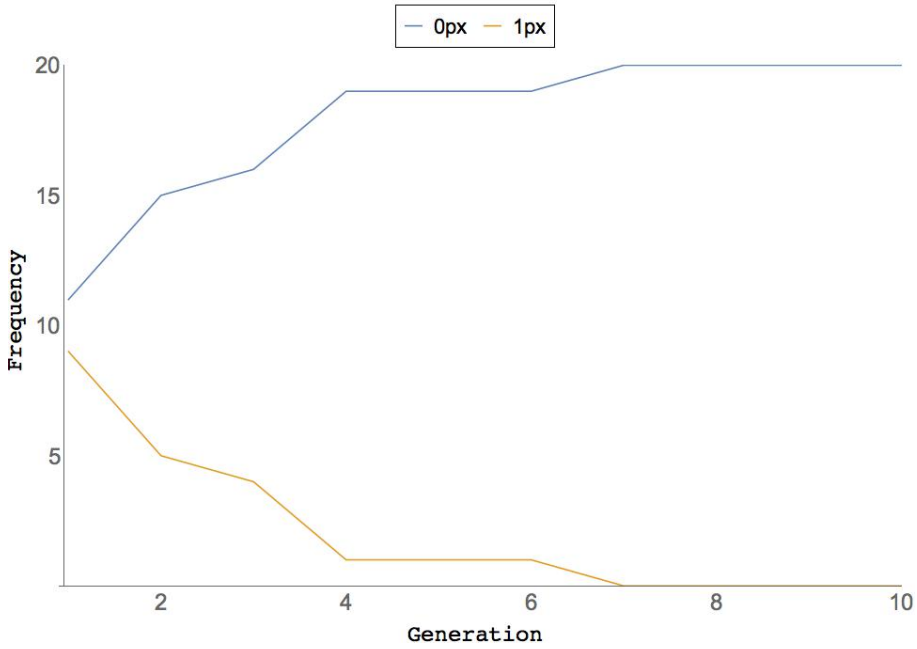

Figure 4.1.10. Frequency of navigation bar element border values over 10 generations

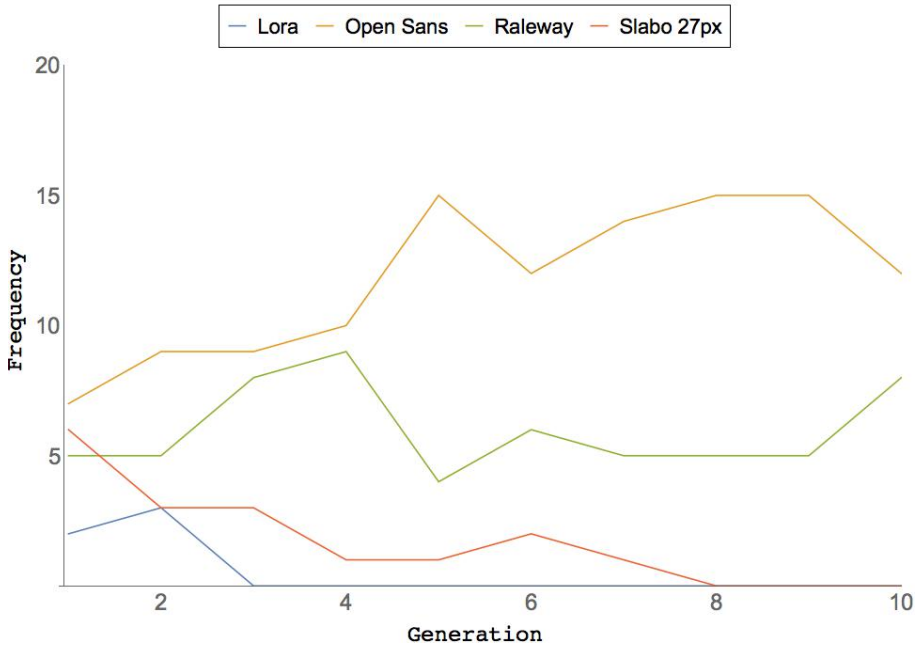

Figure 4.1.11. Frequency of navigation bar font values over 10 generations

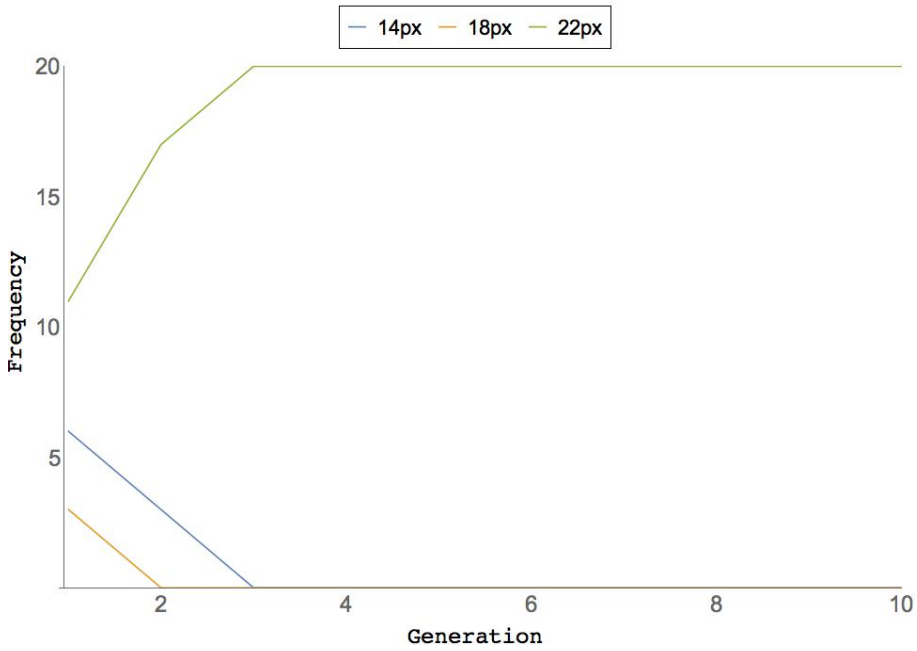

Figure 4.1.12. Frequency of navigation bar font size values over 10 generations

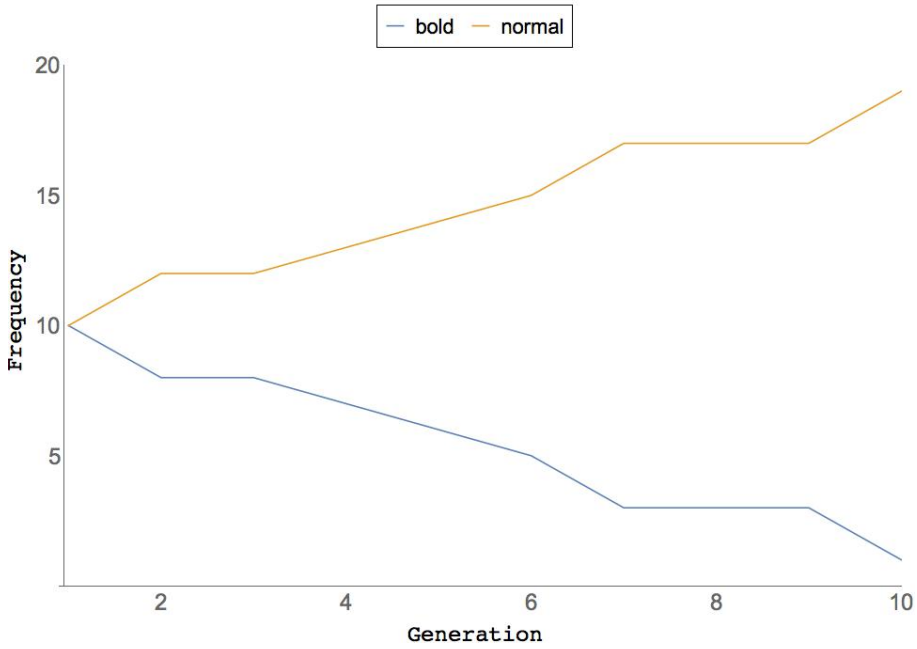

Figure 4.1.13. Frequency of navigation bar font weight values over 10 generations

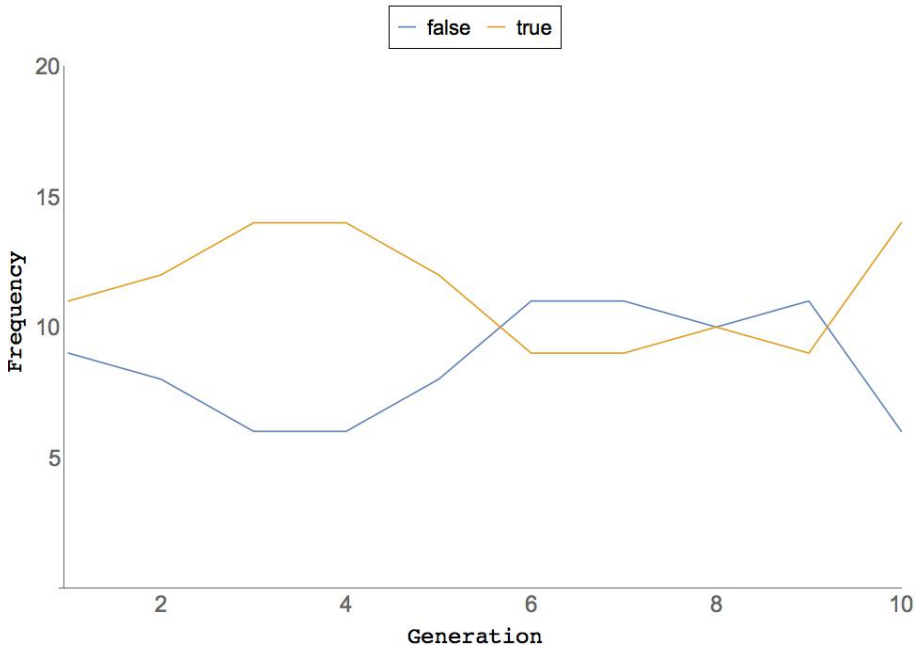

Figure 4.1.14. Frequency of navigation bar shadow values over 10 generations

4.2 Fitness Area Graphs

The dataset presented in this section examines the total fitness "share" occupied by certain features. As all generations have a maximum fitness of 300, grouping individuals by feature, summing their fitness, and "stacking" the data on an area plot provides a visual representation of the percentage of the total fitness "budget" allotted to that feature. It is of note that these graphs only include the first nine generations, as these are the only generations with complete fitness data: the 10th generation did not undergo a complete user evaluation process. Additionally, this section includes a table of the average correlation values between a feature's total fitness and its frequency over nine generations.

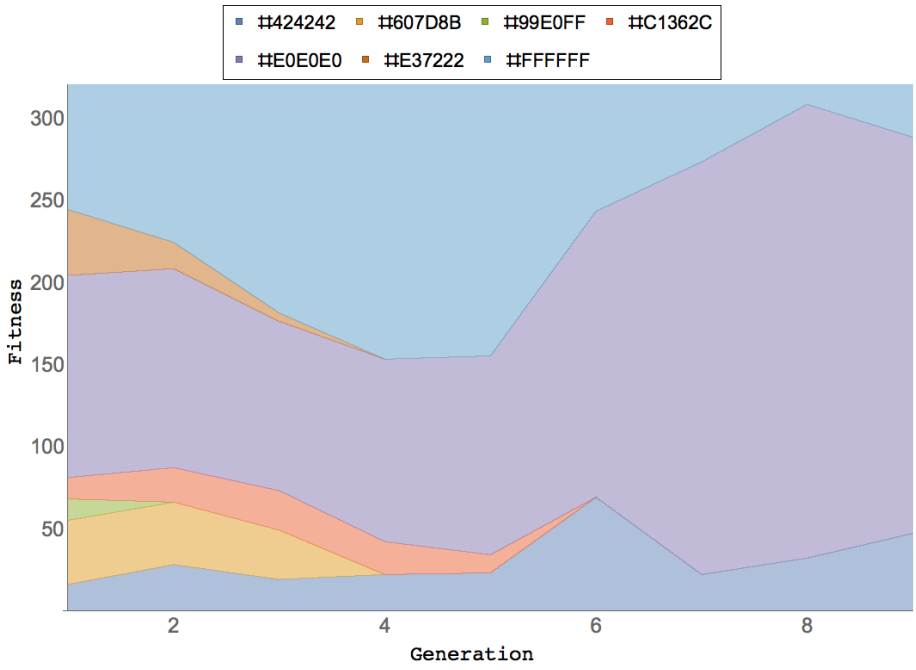

Figure 4.2.1. Fitness "share" of content background color values over nine generations

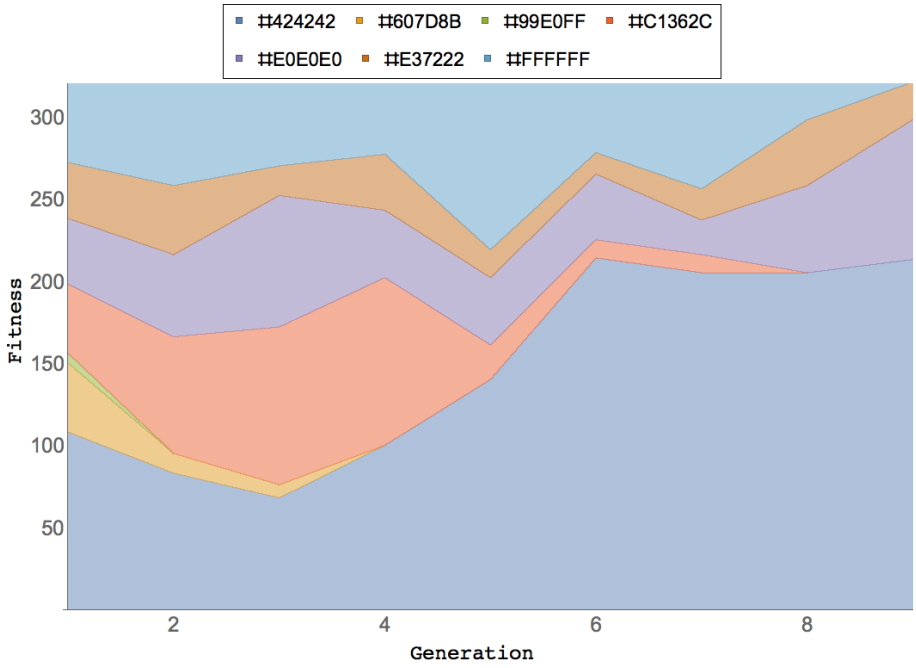

Figure 4.2.2. Fitness "share" of navigation background color values over nine generations

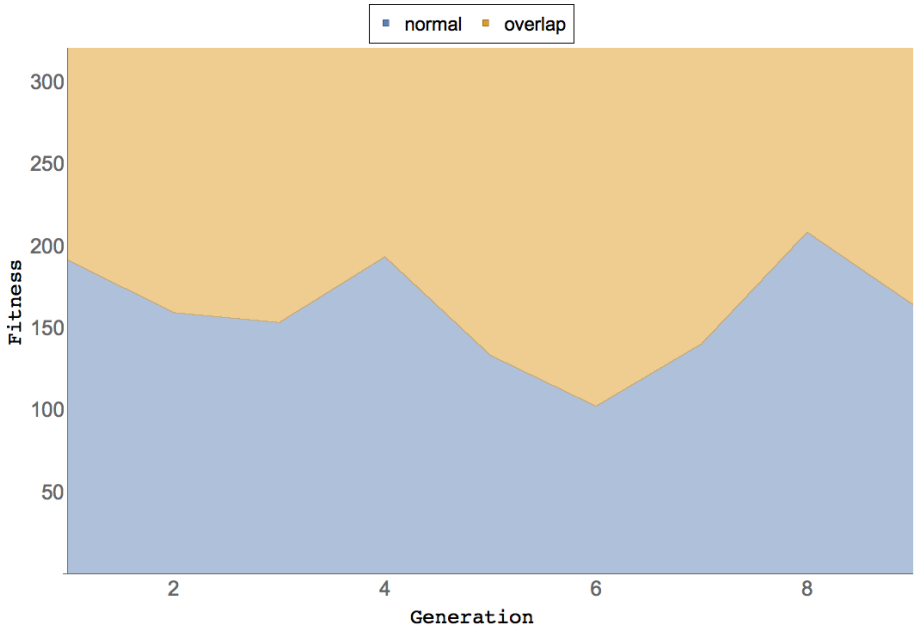

Figure 4.2.3. Fitness "share" of content style values over nine generations

Feature	Correlation
Grid gutters	0.99
Element width	0.95
Navigation background color	0.91
Navigation element borders	0.95
Navigation bar font	0.89
Navigation bar font size	0.98
Navigation bar font weight	0.95
Navigation bar shadow	0.85
Content background color	0.96
Content style	0.80
Content font	0.85
Content font size	0.98
Content font weight	0.95
Content shadow	0.85

Table 4.2.1. Average correlation value between total fitness and frequency per feature

4.3 Voting data

Figure 4.3.1 depicts the voting tendencies of all non-anonymous users that submitted at least one vote. Voting data on non-anonymous users was not collected.

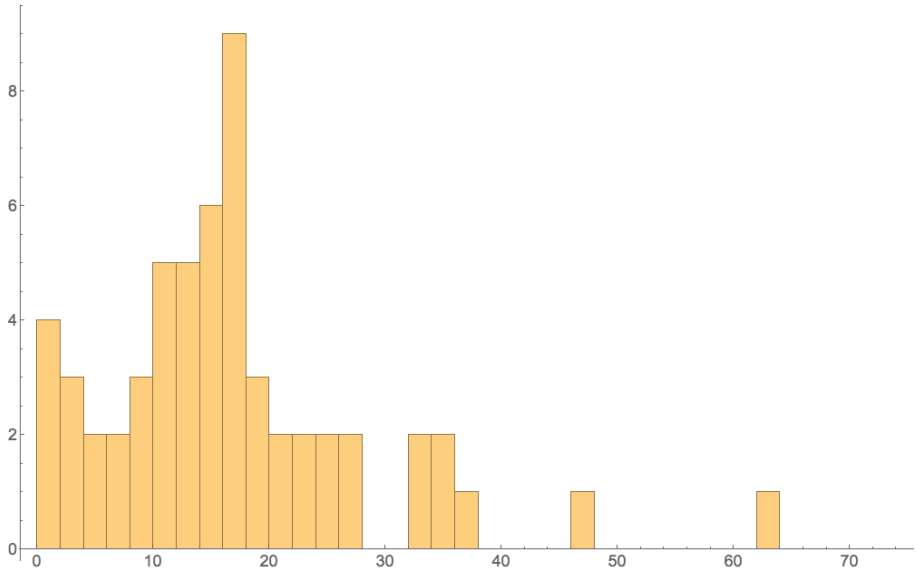

Figure 4.3.1. Histogram of vote distribution

4.4 Selected Design Examples

This section presents designs that illustrate the performance of the algorithm. Figures 4.4.1 and 4.4.2 show the worst and best designs in generation one, whereas figures 4.4.3 and 4.4.4 depict the worst and best designs in generation nine. The true last generation cannot be used because no fitness data was collected.

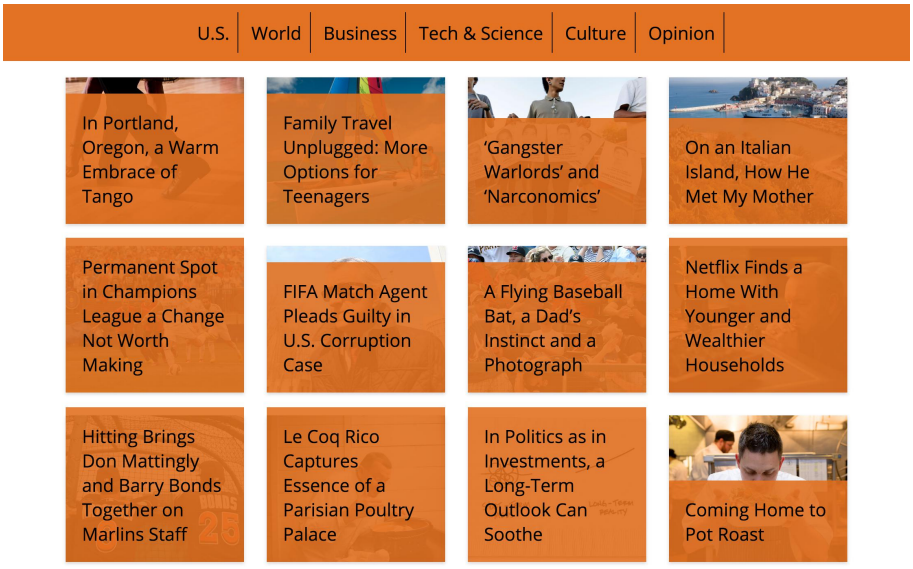

Figure 4.4.1. The lowest-fitness design in generation one

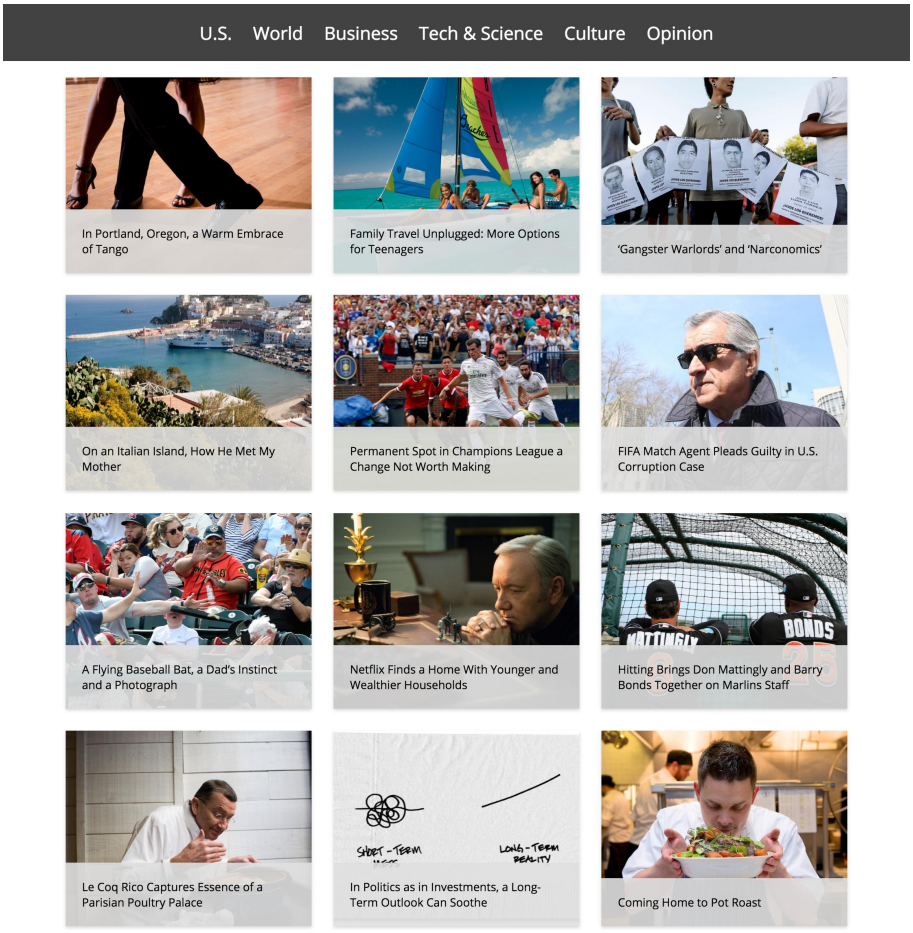

Figure 4.4.2. The highest-fitness design in generation one

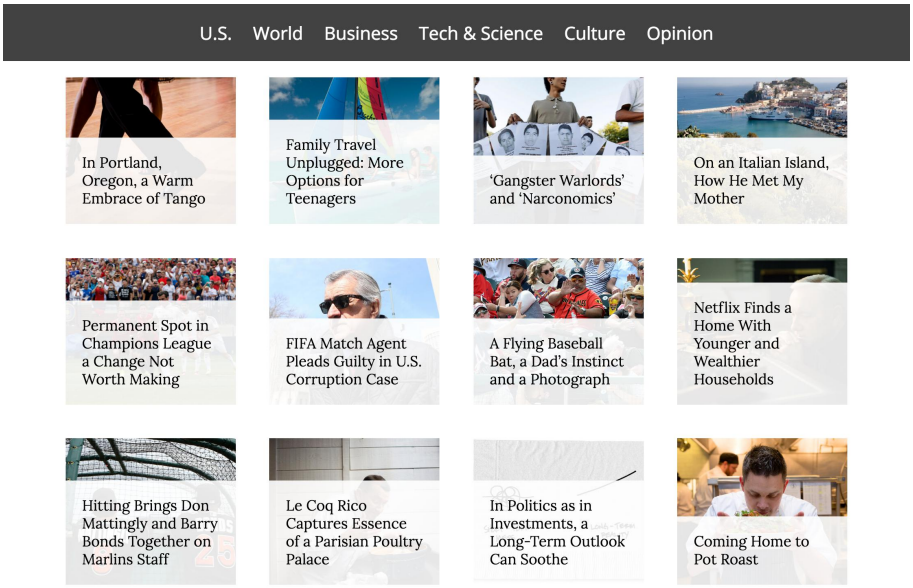

Figure 4.4.3. The lowest-fitness design in generation nine

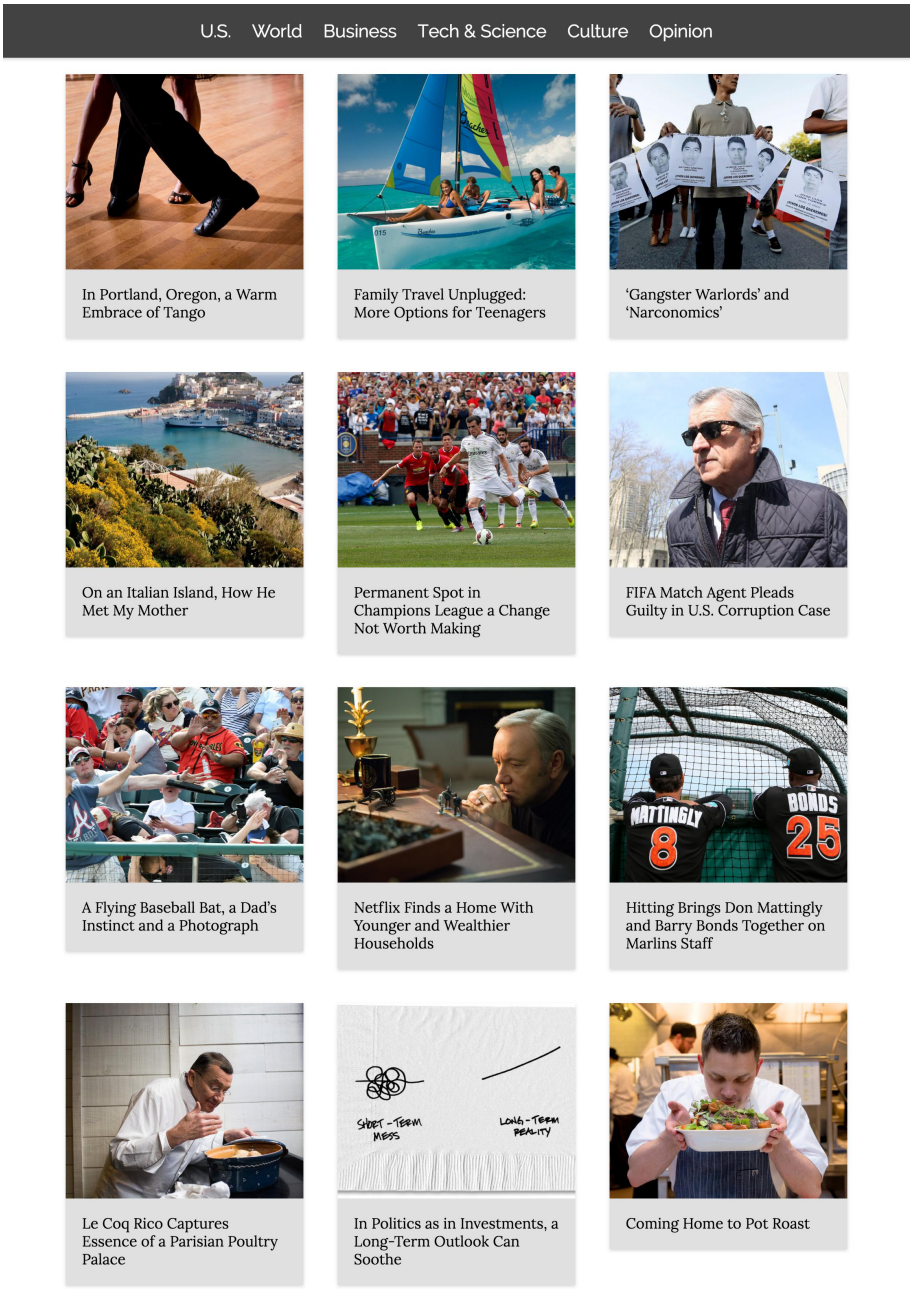

Figure 4.4.4. The highest-fitness design in generation nine

5
Discussion

5.1 User Participation

User participation exceeded expectations. Over the course of four days, 63 registered users submitted an average of 32 votes each. Based on the confidentiality agreement, information was not collected for anonymous users. However, examining total fitness data provides some information on vote distribution between registered and anonymous users: 2047 votes came from registered sources and an additional 690 came from anonymous users. The 2737 total votes were good for 9 "full" generations and a final 10th generation that presents the last designs generated by the algorithm.

Figure 4.3.1 illustrates the vote distribution among 59 users. The most common range was between 10 and 20: this is not unexpected, considering the minimum count for reward qualification was 10. However, there are surprising outliers. There are four users not included on the histogram that each submitted over 100 votes. Most remarkably, two users voted over 300 times.

Feedback collected at the end of the study suggests fatigue was not an issue. On the contrary, some users went as far as reporting a near *addictive* quality to the

evaluation process. As many visitors went beyond the 10 vote "minimum," it seems likely that the three-design comparative evaluation process was easy and engaging. Furthermore, there were many reports of multiple voting "sessions" spread out over a long period of time, driven by a desire to see the algorithm's evolutionary progress over multiple generations.

5.2 Efficacy of Algorithm

The first stage in evaluating the performance of the algorithm is based on the postulation that there exists one or more combinations of features that would result in a preeminent design or "class" of designs expressing similar traits. If the algorithm performs correctly, this design *should* eventually emerge as its constituent features are selected over the course of multiple generations and begin to increase in frequency until they "overwhelm" less desirable features. In essence, it was believed that an attractive design existed as a optimization target for the algorithm to converge on.

Examining the figures in section 4.1 lends credence to the notion that the algorithm was able to converge on a design target. Many of the 14 features showed a clear trend in the expression of specific values. For example, figure 4.1.1 demonstrates a preference for light gray (#E0E0E0) as a content background color: despite being expressed in only 25% of the initial population, the final generation was comprised entirely of individuals presenting that feature. Additionally, this graph illustrates a more general trend towards selecting away from achromatic (non-gray) colors. Of the seven possible color values, four are chromatic: #607D8B, #99E0FF, #C1362C, and #E37222. None of these colors "survived" past generation six. Figure 4.1.9 is indicative of a similar trend in color selection. By the final generation, the only two remaining navigation bar background colors in the population are a light and dark

gray. The last chromatic color, orange (#E37222), was selected out of the population following generation 9.

Multiple other graphs also suggest the existence of higher-valued traits that "win" within their respective features. For instance, figure 4.1.3 depicts a trend in content font size. By generation seven, every individual in the population expresses a font size of 18px. Some graphs are of note because they demonstrate a trend in a "class" of values rather than a singular one, exemplified by the two font feature graphs: figures 4.1.11 and 4.1.2. The former shows a preference for sans-serif fonts (Open Sans, Raleway) in the navigation bar while the latter indicates a bias towards serif fonts (Lora, Slabo 27px) in content presentation. In total, 11 of the 14 features composing a design were overwhelmingly represented by a single value or class of values in the final generation.

Also worthy of mention are features that did not converge on a specific value. Figure 4.1.6 is representative of this. Across all 10 generations, neither of the two values showed a clear trend in selection. Based on the assumption that some features simply do not have a consensus "good" or "bad" value, this is not unexpected. Indeed, the lack of a trend within these features may reveal a certain robustness to the algorithm, indicating it can "support" conditions where no preference exists.

The second stage of evaluation is focused on an investigation of the underlying fitness values driving the algorithm. Typically, fitness data provides a reliable metric for judging the performance of a GA: a well-performing algorithm should demonstrate an increasing average fitness over all generations. However, peculiarities of the IGA fitness function used in this experiment render an analysis of average fitness impossible. As a new generation is created after the current population reaches a total fitness of 300, the average fitness of each generation is identical.

The inability to analyze average fitness complicates the assessment of performance. In lieu of this, further examination is based on the principle of GAs that fitter individuals will be selected more often. It is important to show that traits appearing most frequently in a population belonged to the fittest individuals of the previous generation, as this implies those individuals were routinely selected to contribute their traits to the new generation.

To this end, the graphs in section 4.2 provide a visual representation of the relative fitness of each trait. As the summed fitness of all individuals in a population is invariant, the values can be grouped by trait and "stacked" to create an area plot illustrating how a given trait performs over every generation. Figures 4.2.1 and 4.2.2 show the fitness landscape of two complex features with seven possible values. Compared with their corresponding figures 4.1.1 and 4.1.9, a visual correlation is immediately obvious. As the total fitness "share" of a feature increases, so does its frequency in the population. Table 4.2.1 lists the average correlation value between a feature's frequency and total fitness over nine generations. Even the smallest correlation coefficient, .80 for "content style," is large enough to indicate a significant trend. Figure 4.2.3 shows the fitness landscape for this feature.

This data suggests that the algorithm is performing at a basic level: better features, judged by the fitness value of the individuals expressing them, are preferentially selected for and increase in representation over time. Based on results from section 4.1, the algorithm shows a long-term convergence process. An examination of the designs presented in section 4.4 further supports this idea. The best design in generation one (figure 4.4.2) is very similar to the best design in generation nine (figure 4.4.4): the colors and page structure are nearly identical, with the most notable difference being the change from "overlapping" to "normal" article titles. These features evidently maintained a high value over the course of the experiment, as they

consistently garnered a large quantity of votes and, as a consequence, remained well-represented in the population. On the other hand, almost *none* of the features expressed in generation one's worst individual (shown in figure 4.4.1) survived. Furthermore, the visual difference between the best and worst individual in generation one is much greater than the difference between the pair in generation nine. This suggests that, as is intended, the algorithm is optimizing towards a homogeneous population consisting entirely of the best feature combinations.

5.3 Possible Issues

Though it can be said with some confidence that the underlying IGA is functioning at a fundamental level, it is difficult to make a substantiated claim that the designs in the final generation are more appealing than the designs in the first. It may be the case that the overarching phenotype the algorithm converged on was not representative of the ideal combination of the available features. This can best be likened to a known pitfall of optimization algorithms in general: the search process may become "stuck" in a local optimum of significant magnitude.

The nature of the problem domain may exacerbate this pre-existing trait. Evaluating the quality of a design is a subjective process, often involving many competing and correlated variables. For instance, two colors may be attractive on their own, but unappealing when used together. In the context of a genetic algorithm, this could easily manifest as an issue: features in undesirable combinations are likely to be pushed out of a population before they are given a chance to be evaluated alongside elements that demonstrate synergy with them. Additionally, mutation-driven re-introduction of lost features may not be effective in resolving this issue. If only a single individual receives the necessary mutation to create the desirable combination of features, there is no guarantee it will survive the current generation. This is

especially true if the individual expressing the mutation possesses a number of other undesirable features that make its selection unlikely.

The aforementioned issue illuminates another concern with this experiment: the fitness function and associated method of evaluation. The main considerations when the fitness function was designed were the minimization of fatigue and a general ease of evaluation. While ostensibly successful in these goals, the comparative evaluation of a random subset of the population creates a scenario in which the maximum fitness a given *phenotype* can attain is directly proportional to the number of times it occurs in the population. For instance, a mutated individual that re-introduced a "dead" feature in an appealing combination cannot be voted on more times than it is selected for viewing. This quantity is necessarily limited by the evaluation-selection algorithm discussed in section 3.3.1 based on the fact that the mutated individual, solely representative of its phenotype, only appears once in the population. In this scenario, the final vote tally may not be indicative of the true relative quality of the design.

The minimal user participation data collected and discussed in section 5.1 reveals an issue in experimental design: there was no cap on user voting. Two users voted more than 300 times each, effectively guiding the algorithm through entire generations based solely on their preference. When the experiment was planned, it was believed that the search process would naturally settle on a "compromise" set of designs that reflected a mix of every participant's preferences, smoothing out outliers. The lack of a voting cap allows a user to significantly impact the hypothetical optimization target the algorithm is attempting to move towards. Minimally, this creates noisier data that makes more complex analysis difficult to perform. In the worst case scenario, some features may be selected out of a population entirely, despite being appealing to a majority.

5.3.1 Suggested Improvements

Future work in this experiment could focus on improving on all three of the issues presented in 5.3. The first issue—a too-rapid removal of features that may be desirable in later generations—is the most challenging to resolve, as it is fundamentally connected with the behavior of genetic algorithms. Beyond conducting larger scale studies to refine mutation rate, a possible solution might involve an active and continual "tweaking" process performed by a human overseer. Based on observed homogenization within a generation, new features could either be manually introduced for "testing," or the mutation rate could be raised to naturally produce a greater variety of phenotypes.

The fitness function could potentially be modified towards a more granular implementation. The current evaluation method only allows for a single winner among a set of three possibilities, even in scenarios where a second design is almost as appealing. If the system were altered to allow for designs in a subset to be ranked and assigned a proportional fitness value (rather than 1 or 0), fitness data over a generation might better reflect user preference and improve algorithm performance.

Preventing user "over-participation" has an easy fix: introduce a per-user voting cap. In general, voting was loosely controlled and unmonitored. Future work would benefit from the collection of more complex metrics on user participation that could provide insight on evaluation trends and overall algorithm performance. In a similar vein, the inclusion of a post-study survey would provide a formal, controlled interface for the collection of feedback on multiple aspects of the experiment.

6
Conclusion

6.1 Summary

This paper described the design of a user-driven interactive genetic algorithm intended to optimize website design and outlined the creation of a software implementation. The software was used as the basis of a multi-day study where users were asked to visit a public web address and submit votes on their favorite design among a subset of possible designs within a population, the data from which was used to drive further generations of designs. The software performed admirably: no errors were recorded over the course of more than 10,000 fulfilled requests.

Two main types of data were collected and analyzed: the frequency with which each design feature's traits appeared in a generation and the fitness values of the associated individual that contained them. The first dataset was used to judge the algorithm's convergence on a specific design phenotype, with the belief that this was indicative of a basic level of performance by the algorithm. An examination of the frequency of traits occurring in generation 10 versus generation one supports this hypothesis: the designs in the final generation express a much smaller variety of

traits and, consequently, share a similar appearance. The second dataset was used to provide further support for the algorithm's performance. A strong correlation was observed between trait fitness and frequency, indicating that the algorithm was almost certainly performing optimization correctly.

However, it is more difficult to claim that the optimization target was actually representative of better designs. Though anecdotal user feedback received and qualitative analysis of the final designs suggests that the designs were almost certainly better overall, the experiment did not formal method to test this. Future work in this area might include a post-survey study to gather a large quantity of feedback on final design quality.

The implementation also provides many avenues for further improvements and experiments. Different fitness functions could be tested, genetic algorithm parameters (crossover rate, mutation rate, population size, etc) could be tweaked, and more data could be collected to better judge the algorithm's performance.

6.2 Future Work

What this paper details is merely a proof of concept. Plenty of work has already been done in the field of computationally-assisted design augmentation, with plenty more still to explore. Multivariate and A/B testing are already used in combination with sophisticated usage data to assist web designers in creating effective designs. In a future experiment performed at the necessary scale, this data could potentially replace the fitness function described within this paper. Rather than requiring a deliberate survey process, the quality of a design could be derived from metrics like conversion rate or time spent on a page.

Indeed, any practical application of genetic algorithms to page design optimization would require a computationally-derived fitness function. However, if the relevant

data were to be collected, an IGA-driven process could be run over a long time span, steadily performing optimization in a "behind-the-scenes" manner. The result is a constantly-running "smart" multivariate testing system where viable combinations are continually narrowed through the genetic algorithm, potentially resulting in better performance with less required website traffic.

Bibliography

[1] Brahima Sanou, *ICT Facts & Figures*, `http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf`. Accessed May 27, 2016.

[2] Thom File and Camille Ryan, *Computer and Internet Use in the United States: 2013*, American Community Survey Reports (2013).

[3] Elizabeth Sillence, Pam Briggs, Lesley Fishwick, and Peter Harris, *Trust and Mistrust of Online Health Sites*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (2004).

[4] Dimitri Masson, Alexandre Demeure, and Gaelle Calvary, *Examples Galleries Generated by Interactive Genetic Algorithms*, Proceedings of the Second Conference on Creativity and Innovation in Design (2011).

[5] Ron Kohavi and Roger Longbotham, *Online Controlled Experiments and A/B Tests*, Encyclopedia of Machine Learning and Data Mining (2015).

[6] *Comparing a Multivariate Test to an A/B Test*, `https://www.optimizely.com/resources/multivariate-test-vs-ab-test/`. Accessed April 5, 2016.

[7] Håkon Lie and Bert Bos, *Cascading Style Sheets: Designing for the Web*, 3rd ed., Addison-Wesley, Harlow, England, 2005.

[8] Håkon Lie, *Cascading HTML Style Sheets - A Proposal*, CERN, 1994.

[9] Kenneth De Jong, *Genetic algorithms: a 30 year perspective*.

[10] Nikhil Padhye, *Evolutionary Approaches for Real World Applications in 21st Century*, Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (2012), 43–48.

[11] Erik D. Goodman, *Introduction to Genetic Algorithms*, Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (2012), 671–692.

[12] Keki M. Burjorjee, *Explaining Optimization in Genetic Algorithms with Uniform Crossover*, Proceedings of the Twelfth Workshop on Foundations of Genetic Algorithms XII (2013), 37-50.

[13] Richard Dawkins, *The Blind Watchmaker: Why The Evidence of Evolution Reveals a Universe Without Design*, Norton, New York, 1996.

[14] Karl Sims, *Artificial Evolution for Computer Graphics*, SIGGRAPH Comput. Graph. **25** (1991), 319–328.

[15] Hideyuki Takagi, *Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation*, Proceedings of the IEEE **89** (2001), 1275–1296.

[16] *The New York Times - Breaking News, World News & Multimedia*, `http://www.nytimes.com/`. Accessed May 16, 2016.

[17] *Google Fonts*, `https://www.google.com/fonts`. Accessed May 16, 2016.

[18] David Flanagan, *Javascript: The Definitive Guide*, O'Reilly, Beijing, 2011.

[19] *Node.js*, `https://nodejs.org/en/`. Accessed April 26, 2016.

[20] *Sass: Syntactically Awesome Stylesheets*, `http://sass-lang.com/`. Accessed April 26, 2016.

[21] *Susy*, `http://susy.oddbird.net/`. Accessed April 26, 2016.

[22] *RethinkDB: The Open-source Database for the Realtime Web*, `https://www.rethinkdb.com`. Accessed April 26, 2016.

[23] *Thinky – Node.js ORM for RethinkDB*, `http://thinky.io/`. Accessed April 26, 2016.

[24] *JSON Web Tokens - jwt.io*, `https://jwt.io/`. Accessed April 27, 2016.

[25] *JSON*, `http://www.json.org/`. Accessed April 27, 2016.

[26] Kenneth De Jong and William Spears, *An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms*, Parallel Problem Solving from Nature (1990), 38–47.

[27] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi, *Interactive Evolution for the Procedural Generation of Tracks in a High-end Racing Game*, Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (2011), 395–402.

[28] Juan Quiroz, Sergiu Dascalu, and Louis Sushil, *Human Guided Evolution of XUL User Interfaces*, CHI '07 Extended Abstracts on Human Factors in Computing Systems (2007), 2621–2626.