UiT NORGES ARKTISKE UNIVERSITET

Genetic algorithms (GA) for adaptable design

Faculty of Engineering Science and Technology
Andreas Dyrøy Jansson

Title: Genetic algorithms (GA) for adaptable design

Author: Andreas Dyrøy Jansson

Date: June 6, 2017
Classification: Open
Pages: 69
Attachments: Zipped folder with executable program and demonstration video

Department: Department of Computer Science and Computational Engineering

Study: Master of Science, Computer Science

Student number: 501760/aja073

Course code: SHO6264 Diploma Thesis - M-IT

Supervisor: Bernt A. Bremdal

Principal: UiT - The Arctic University of Norway (Campus Narvik)

Principal contact: Bernt A. Bremdal

Keywords: Genetic algorithm, classification, K-nearest neighbor, web design

Abstract (English):
This thesis discusses the process of implementing and testing of a creative web element design system using a combination of genetic algorithms and K-nearest neighbor classification. Classification will be used as a learning mechanism to give the system the ability to absorb quality measures in regards to visual aesthetics, and utilize this knowledge to evaluate generated designs. In addition, some basic concepts regarding web design will be covered, along with an introduction to artificial intelligence-based design. The results of the implementation will be discussed and compared to related state-of-the-art systems.

Abstract (Norwegian):
Denne oppgaven tar for seg utviklingsprosess og testing av et kreativt web-elementdesignsystem ved bruk av genetiske algoritmer og K-nearest neighbor-klassifisering. Klassifiseringen vil bli brukt som en læringsmekanisme for å gi systemet evnen til å tilegne seg kunnskap om visuell estetikk, og bruke dette til å evaluere genererte design. I tillegg vil grunnleggende konsepter rundt webdesign bli tatt opp, sammen med en introduksjon til kunstig intelligens-basert design. Resultatene av utviklingen vil bli drøftet og sammenliknet med relevante "State-of-the-art"-systemer.

Acknowledgment

I would like to thank my supervisor Bernt A. Bremdal for providing the baseline for the project, and giving me the opportunity to create my own spin on it. I would also like to thank him for his suggestions and feedback throughout the thesis work. I would like to thank Øivind Winge and the developers at Geelmuyden-Kiese for giving me valuable insight and feedback for further development of the program during my work.

Andreas Dyrøy Jansson
Narvik 2017

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

1.1 Problem description

This thesis focuses on the exploration of creative artificial intelligence systems, which implies systems able to create new design and develop unprecedented solutions to problems presented to them. The area of emphasis will be on the application of genetic algorithms combined with classification algorithms. The utilization of artificial intelligence in web design will be the main area of application of the project.

The conducted work should be in respect to state-of-the-art implementations for artificial intelligence based design, with the main focus being on literature applying genetic algorithms alone or as a part of a suite of methods. How to represent and evaluate solutions for visual design will be discussed. A simple prototype program to demonstrate the implementation will be developed and tested, and the results will be discussed. Due to the visual and human preference nature of the task, the program will be highly interactive, giving the user greater control over the genetic algorithm compared to traditional methods. In addition, different ways of terminating the algorithm will be implemented. Additional features to allow users of the program to manually inspect and fine-tune the produced designs will also be added. Furthermore, some basic HTML and CSS will be discussed. Finally, the use of classification algorithms to evaluate and rank tentative designs based on collected vote data will be examined and tested, and the results of this will be discussed.

Based on this summary, the main goals of the thesis will be:

1. Creation of a web page design system able to demonstrate independent creativity while learning to absorb quality measures in regards to human preference
2. Implementation and testing of a dynamic genetic algorithm fitness function able to change over time based on user feedback
3. Improve performance compared to previous work of similar nature

1.2 Artificial Intelligence-based design

Artificial Intelligence (AI) is being used to solve a plethora of problems in the modern world, ranging from beating the world champion at the complex board game Go[1], optimizing design of electronic circuits[2], predicting the price of electricity based on historical data, and even drive cars in challenging conditions[3]. A common denominator is that computers and intelligent systems are excelling at fields typically associated with human experts throughout history. AIs are particularly good at tasks involving large-scale data analysis, and suggest an appropriate response. With the amount of available data regarding design combined with the raw power of modern supercomputers, producing aesthetically pleasing results might be expected to be easily achieved. However, one reason AI implementations are succeeding at the aforementioned activities is due to the fact that they all are able to be broken down into a set of rules learning algorithms are able to pick up. Defining good design is a subjective matter, which varies between cultures, personality and context. According to an article by R. Girling from 2016 [4], creative and social intelligence are skills needed to produce good designs. In his article, Girling discusses the future of job automation, and concludes with creative professions being less susceptible to automation compared to simple routine work. The keyword here is predictability. If a task is able to be broken down into rules, and learned by a learning algorithm, an AI will be both cheaper and more efficient than a human worker. Design is considered a less predictable task, as trends along with the public preference are constantly changing. Human designers are also able to understand and relate to the problem on an emotional level.

In a more recent article[5], he brings up the idea of personalized AIs that learn from their owner's preferences, and thus augment the skills of the designer. He brings up the point of being able to generate and explore a large set of variations of a design, boosting productivity. This may in turn lower the entry point for aspiring designers, as AI generated design could serve as a platform for inspiration for further work. Granted, these generated designs are still based on established rules, created by humans. Girling suggests that until AIs are capable of producing completely novel ideas, elite human designers will still dominate the industry.

In order to give a machine creative capabilities, appropriate representation of the problem is key. As mentioned, intelligent systems are successful at tasks like path finding and prediction due to their ability to be broken down into rules. As visual design preference often is a matter of subjective taste, being able to define rules and representations able to be interpreted by the system is of key importance. Such a representation needs to be simple for intelligent systems to manipulate and evaluate, yet still able to capture all associated nuances. To sum up, providing human feedback during the learning process as discussed above is imagined to yield the best results at the current state of technology.

1.3 The importance of good web design

In a world where an increasing number of interactions between people happens through web pages, good design is becoming more important than ever. Statistics show a yearly increase in internet traffic worldwide, some even claim a rise of up to 10% during the last year[6]. Cisco claims that the amount of data transferred globally over the Internet will exceed 2.3 zettabytes per year (2,300,000,000 terabytes) in the year 2020[7]. According to the May 2017 Web Server Study[8], there were approximately 168 million active sites on the web, a number which has been steadily rising since the year 2000. As the number of web pages grow, being able to stand out in a positive way to keep the reader's attention should be the goal of all web sites. This is especially true for commercial sites who want to stay competitive. First impression is key, and if the visitors do not like what they see, plenty of other sites are available in most cases. Even if there are no alternatives to a particular site, poor design may harm the credibility of the company represented by the page in the long run.

From a web developer's point of view, spending time and effort researching user preferences is imagined to be quite the task, as design preferences and trends often change over time, and for different contexts. But what is considered good design? Experts may agree on certain features to be present in aesthetically pleasing designs. On the other hand, the average user may disagree or not even care. Being able to capture the different preferences for different users, or groups of users regarding design of web page elements is imagined to serve as a platform for web developers ranging from professionals to hobbyist bloggers. Using collected data presented in an intuitive way may save time and effort spent on searching for the next web page design trend.

1.4 Use of genetic algorithms to explore design solution spaces

The genetic algorithm (GA), first described by John Holland [9], is an evolutionary algorithm which is part of the domain of artificial intelligence. Solutions are represented as genes, which are decoded using some mapping function. Appropriate representation is especially important in genetic algorithm implementations, in order to properly explore the solution space and evaluate tentative solutions. Solutions may be represented by binary strings, real values or by using a high level representation. Operators depend on the type of representation used. Using knowledge about the problem to apply appropriate representation allows for problem-specific operators, crossover and mutation. This allows the use of GAs to solve problems ranging from function optimization, design of neural networks, and data mining.

At the core, the genetic algorithm is a guided search often used in optimization problems in a vast span of fields. At the start, the algorithm begins its search in a general set of solutions, and after several iterations, homes in the global optimum. Several mechanisms to avoid local optimums have been introduced, and will be looked into later. After each iteration, the algorithm gets one step closer to convergence. This happens as a result of a selection function, which validates and ranks each produced solution according to a target, which for some problems may be defined as the lowest cost, the highest speed and so on. A new solution is produced as the result of crossover, which is the recombination of highly ranked features from the previous iterations. Similar to the process of natural selection as described by Darwin, the solutions best fitted to solve the proposed problem according to the test will flourish, while the less fit individuals eventually disappear from the set of solutions.

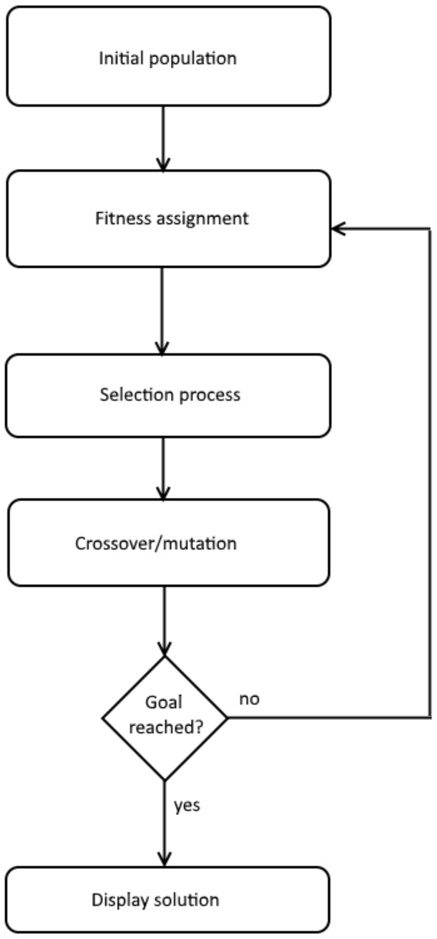


Figure 1: Flow chart of basic genetic algorithm

The flow chart of fig. 1 shows the major steps of the genetic algorithm. At first, a population of randomly generated solutions is created. Individuals are then assigned a fitness score using some object function representing the problem. Using one of several processes of selection, crossover is performed, mixing features of each parent. Selection methods include elitist selection and roulette wheel selection. Both methods will be looked into and discussed later. Furthermore, mutation may also be performed. Mutation is used as a way of avoiding local optimums, in addition to making the algorithm able to adapt to a changing goal. Mutation may also serve as a way of expanding the representation of the solution space in a given population, introducing new features. New individuals are added to the population, or replace less fit, depending on the implementation, and the process continues. Or, if any of the produced offspring solves the problem, the algorithm terminates. The solution may then be displayed.

The initial population of a genetic algorithm may be virtually infinite. The number should be large enough to cover a reasonable amount of permutations, and small enough for the hardware to efficiently run the algorithm with minimal delay. This is because one of the strengths of this type of search is the ability to explore quickly. The complexity of the problem also determines speed. Naturally, complex problems with hundreds of variables will run slower than simple 2D optimization problems. If the initial population is too small, convergence could take longer, as the desired properties of the solution may be absent, and have to evolve from nothing.

2 State-of-the-art systems

2.1 In literature

Using genetic algorithm implementations for design has been performed and documented in various fields to a certain extent. An example is the work regarding the design of mobile phones by C. K.Y Fung et al. [10]. In their work, the fitness function was based on set of survey data, reflecting the public's opinion regarding good design. The data was used to produce a set of rules, and penalizing tentative designs not obeying said rules. However, as the rules were based on already collected data, they were static during run-time. As a result, the fitness function was not able to change over time, as design preferences tend to do. Their work is a particularly good example of this, as most of the designs used were older-style cell phones with small screens and numerical keyboards. Today, these designs would be considered less attractive than large touch-screen phones. Furthermore, the collected data was based on already existing conventional designs, which is expected to rule out radical new designs. This was due to the method of representation, which will be looked at later.

The use of an interactive fitness function is one way of introducing a dynamic element to the selection process. In the work of M. Khajeh et al.[11], an interactive genetic algorithm was proposed to let the user combine various fabrics and clothing designs to form a full outfit. This is a fully interactive process, where the user takes an active role in evaluating every design of each generation. A limiting factor in this system is the number of presented designs in each generation. As the user has to evaluate every design, this number has to be kept relatively low. As seen in figure 4b of [11], this is set to 8 designs per generation. In addition, the algorithm is set to terminate after four generations, "to avoid user fatigue"[11]. According to the authors, convergence time is user dependent and may vary "extremely". The resulting program does not store the feedback provided during run-time, making the user start from scratch each time when designing clothes. The program does however let the user export the resulting design as an image for further use. Results suggest a potential to save time and money during the design process compared to manual work, which is interesting. The use of a distributed database of designs is also suggested. As further development, one point is the addition of a 3D rendering system, to give the user a more true-to-life impression of the produced designs.

The concept of adaptive web sites has been discussed by M. Perkowitz and O. Etzioni[12]. They discuss the navigation flow of a dynamic web site in response to the ever changing browsing habits of visiting users. Using the server's access logs, an algorithm for managing related pages as seen by the users is presented. Although not directly related to genetic algorithms, the concept of using dynamic user feedback during run-time to produce evolving solutions was used as inspiration during development. The proposed algorithm uses data stored on the server, implying learning. In addition, there is an interactive element: proposed solutions are presented to the web master before implementation. To group related pages, the authors used an algorithm called the COBWEB clustering algorithm. However, no visual design is discussed.

Visual design of web pages has been discussed in a limited format by A. Oliver et al. in [13]. In this paper, interactive design similar to the work of Khajeh et al. is used to optimize both page design and layout. A limited set of CSS properties are used to generate designs for all elements on the page, including background color, text color and font size. The iterative process works by the user selecting one or more pleasing designs, driving the algorithm towards convergence. The resulting implementation displays a number of 12 individuals per generation for similar reasons as above. One thing worth noting is the time spent on generating and displaying each new generation. The results discussed in the paper show an average time of 48.9 seconds, "where

[...] about half of this time is devoted to the network and to the display." [13]. Even though this implies around 20 seconds for the user to select designs, there is room for improvement regarding speed. The programming language and run-time environment used, which may impact performance, is not discussed. Furthermore, the article was 14 years old at the time of writing, which may be another factor in regards to the resulting performance. Similar to the interactive design of clothes mentioned above, there were no implemented rules allowing for automation during the evaluation process. Such rules were discussed and suggested for further development only. Being able to base the initial population on an existing design is implied, but not explicitly stated: "When the user wants to define or modify the style/layout characteristics of his site (or of a given page), he may use the IGA [...]" [13]. Another consequence of the age of the paper is the use of dated web standards. The new HTML5 and CSS3 standards offer new features not present in this implementation. Finally, the feedback provided by the user during the iterative process is not saved. Only the final designs are kept.

2.2 Representation

In the work by Fung et al., designs are represented using a combination of categorical and quantitative attributes. For mobile phones, categorical attributes are represented as integers, which are mapped to specific predetermined features. Examples include the shape of the buttons, form factor and body shape. The quantitative attributes take the form of real numbers, and represent various physical properties, including width and thickness. Representing values this way is logical in regards to the problem of optimizing design of mobile phones. Features like shape and form factor need to be represented via discrete values mapped to preset options, as giving the algorithm complete freedom to explore new shapes is imagined to involve unnecessary complexity. Doing this would also most likely produce a high number of unusable solutions. On the other hand, features like size are able to be represented as continuous values. This is because such a value is straight forward to encode, and all values in the specified range are valid solutions.

Khajeh et al. use a similar approach, representing designs through integer mapping only. The reason for this is similar to the one discussed above. In order to use real coded values, the implementation would need to know how to shape and texture clothes, adding an extra layer of complexity to the approach. The system would first need to generate designs based on learned knowledge about fashion, and then recombine features to be used in the interactive genetic algorithm. Breaking clothes down to simple parts and textures makes the representation simple to implement, allowing the relatively complex task of designing a full set of clothes to be solved by the GA.

As the adaptive web site problem does not utilize a genetic algorithm, representation is somewhat different. However, there are some points that may be relevant. Pages are represented on a high level, to use GA terminology. They are interpreted in plain text, hand-tagged by the web master. This information is used by the clustering algorithm in order to group pages together. Relating this to GA would involve creating a specialized set of genetic operators supporting the representation used. Evaluation might then have been performed by using a data mining approach.

Finally, the design of a web page is represented in a similar fashion to the mobile phone example, using a combination of integers mapping to features and real numbers in the implementation discussed by Oliver et al. In their work however, real values are limited to integers, as they are used to represent colors in the RGB color space, which in CSS are limited to integer values. The reasons for this are the same as for the ones above, as CSS requires correct syntax for several properties. Giving the GA the task of generating valid strings would increase the solution space to a point where convergence would be virtually impossible, due to the amount of possible string permutations.

2.3 Other systems

Examples of systems applying artificial intelligence and learning algorithms to produce designs include the Prisma app[14] and The Grid[15]. The Prisma app utilizes neural networks trained on classic art samples and applies these styles to photographs and videos taken by the user. The Grid is an online web page designer, which generates the page based on content provided by the user. However, both examples require a starting point provided by the user, and do not create anything ”new” in the sense of creativity. If the user is a less experienced individual, bad designs may be the result[16]. This leads to a point discussed by Y. Vetrov[17], in which he brings up the idea of ”creative collaboration”, where designers work in tandem with various generative algorithms to explore and solve design problems. Combining intelligent systems with human preference and ”gut feeling” shows great potential and should be explored, according to Vetrov.

3 Basic CSS overview

In this chapter, some basic CSS properties and conventions in respect to the thesis will be discussed.

Using pure HTML only to make a web page may be considered less professional compared to using even a minimum of styling. Design is of course optional, meaning the developer has to devote extra time and effort to do this. Maybe they don't have the skills, as HTML and CSS are two different paradigms. Pure HTML pages are recognizable by their black and white color scheme, and default system fonts combined with poor spacing. In short, they appear bland. HTML offers little to none in regards to design beyond the bare minimum. Cascading Style Sheets (CSS) is the format used to design web pages. Styling code determines the appearance of elements on a web page, including colors, alignment and text size. Appropriate styling is dependent on the context of the page, naturally. Element styling may be defined in an external file, as part of the HTML file, or inline for each element. To achieve a cleaner and more readable code, separating design and layout definition is considered good practice. By doing this, it becomes possible to reuse the same design aesthetics on different pages. Using the same design language gives a more professional overall impression of the site. Each element of the page is associated with a class defined in the external file. The class is a set of properties which are applied to the element. Classes may be named after the element they belong to, or given any name. If the class has a custom name, the element it belongs to has to be explicitly tagged with the class name in HTML[18].

Most CSS properties may be used on any element, with varying results. In this thesis, the main focus will be on the following:

Property name	Legal values	Description
background-color	rgba, hex color code, name	Sets the color of the element
border-color	rgba, hex color code, name	Sets the color of the element outline
border-radius	positive integer values	Sets the radius of an element's corners
border-style	dotted, dashed, solid, double, groove, ridge, inset, outset, none, hidden	Sets the style of the element outline
border-width	positive integer values	Sets the thickness of the element outline
color	rgba, hex, name	Sets the color of text inside the element
font-family	string argument list of font names	Sets the font.
font-size	positive integer values	Sets the size of the text inside the element
padding	positive integer values	Clears an area around content inside an element
text-align	left, center, right	Aligns the text inside an element
text and box-shadow	vertical position, horizontal position, blur radius (optional) and color	Creates a shadow effect around text or an element

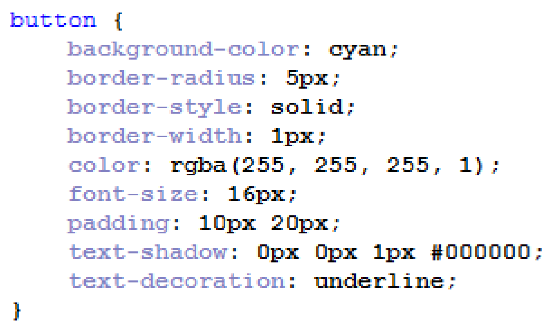
Table 1: Commonly used CSS properties in thesis

These properties may then be applied to most HTML tags. This thesis will focus on the more common tags:

Tag name	Description
button	Crates a clickable button, used to perform some action
input	Crates a text field, where the user may enter text to submit
h1	Defines a header text. Larger font and bold
p	Defines a paragraph, often used for page content text
table, th, td	Creates a table. Header text is tagged using th, and cell content is tagged with td
div	May contain any other element, used in this thesis to create containers

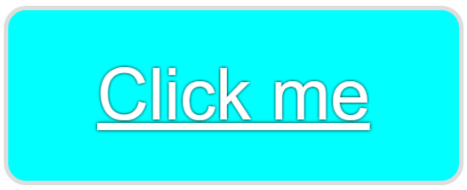
Table 2: Commonly used HTML tags in thesis

As an example, the below code produces the following design:



```
button {
    background-color: cyan;
    border-radius: 5px;
    border-style: solid;
    border-width: 1px;
    color: rgba(255, 255, 255, 1);
    font-size: 16px;
    padding: 10px 20px;
    text-shadow: 0px 0px 1px #000000;
    text-decoration: underline;
}
```

(a) CSS code for simple button

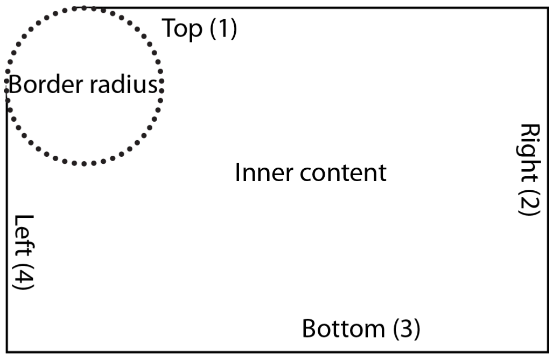


(b) Simple button rendered in browser

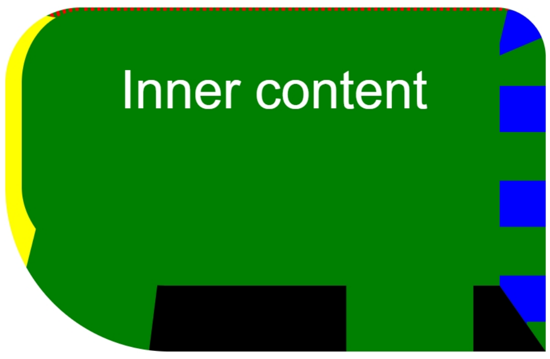
Figure 2: CSS code and resulting button

As seen in fig. 2a, some features of design are represented as integer values, while others are predefined strings.

Each element may have one or more unique border styles, colors, radii and widths:



(a) CSS border properties are interpreted clockwise



(b) Example of CSS button with individual border styles

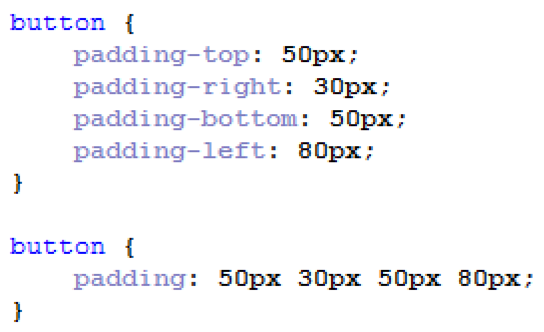
Figure 3: Each side of an element may have its own style

As seen in fig. 3, CSS allows the top, right, bottom and left edge of an element to have individual colors, styles, radii and widths.

For web developers creating CSS, several methods of approach exist ranging from beginner to professional. Graphical user interface tools using drag and drop, along with the "What you see is what you get" concept, allow quick generation of CSS. Due to their simple nature, generated code is often hidden from the user, leading to less control over fine tuning features not offered by the interface. Clicking different buttons and menus and adjusting sliders trying to get the right design might prove time consuming in the long run. Relying on such tools also requires continuous updates. If the tool is older, newer features will be out of reach of users inexperienced with CSS code generation.

On the other hand, advanced web designers often prefer to write CSS manually using pure text editors or other tools with minimal support to access and control advanced features. As one may understand from table 1 and fig. 2a, styling a simple button may involve extensive typing using such tools. If one wants even more control, for instance a simple gradient or making the button change color when clicked, even more typing would be necessary. Also, when using text editors to generate CSS, the code is prone to syntax errors which may not be apparent when viewing the resulting web page, leading to time spent troubleshooting the code at later stages. Another point is the need to know all applicable properties by heart, or spending time researching reference sheets. Time constraints, along with established design paradigms, is imagined to limit an advanced user experimenting with irregular compositions.

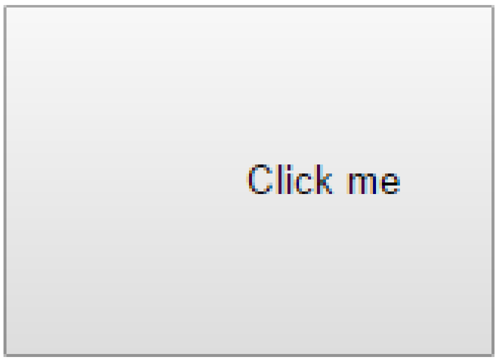
Furthermore, there is no standard way of structuring properties. As an example, the below code samples produce the same result:



```
button {
    padding-top: 50px;
    padding-right: 30px;
    padding-bottom: 50px;
    padding-left: 80px;
}

button {
    padding: 50px 30px 50px 80px;
}
```

(a) CSS code for padding using different syntax



(b) Resulting button

Figure 4: Different CSS syntax producing same result

The first code of fig. 4a sets the padding on all sides using the standard syntax, while the lower uses the shorthand syntax. The standard syntax is easier to read, but requires more typing.

The new CSS3 standard allows for more design features, including rounded corners, gradient colors and to create transformations and animations without the use of additional tools or libraries. This allows for a more uniform cross-platform user experience, which in turn reduces the time needed to develop and test a design in multiple browsers. The new features also need time learning, which could lead to developers not taking the time investigating the new possibilities, and just keep designing using old standards.

A complete reference to CSS may be found at W3schools.com[19].

4 Approach

As discussed earlier, using GAs to explore solutions spaces regarding design requires special attention when representing the information. As mentioned in section 2.2 and explained in section 3, CSS requires precise syntax, meaning that features should be coded as integers, and then mapped to the correct commands. Web page elements need to be both visually appealing and easily accessible, meaning some degree of human interaction throughout the iterative process is required. Therefore, a graphical user interface will be needed.

The values themselves may be represented using binary format, real values like in the work by Fung et al., or some form of high level representation. When using binary format, it becomes possible to perform crossover and mutations on a lower level, giving more control to the algorithm. Furthermore, binary operations may be conducted using very simple operations, and when decoded, results may vary vastly from the original value. This is imagined to allow the algorithm more freedom to explore unexpected combinations of features.

To get a random population, using whatever method provided by the platform of choice is considered sufficient in the context of this thesis.

Different methods of crossover will produce various results, meaning that more than one should be implemented. Commonly used crossover schemes include the elitist selection, and roulette wheel selection. In addition, given the interactive nature of the algorithm, letting the user manually select individuals for crossover will be investigated.

As mentioned, using binary representation allows for lower level manipulation of values. Small changes to the binary string representing one design could lead to something radically different when converting back, and is simple to implement.

Measuring fitness is another important aspect of GAs. As mentioned above, there are multiple ways of determining fitness of a design. Does the design work in conjunction with the other elements on the page? Is the text readable? Does the design appear professional in terms of color usage? One way of determining this is by collecting a set of data from existing web pages where good design is present. The designs could then be broken down into a set of rules, and applied in the fitness function using a similar approach to the design of mobile phones by Fung et al. However, taking the time to search the web for designs, decoding the CSS codes and manually create a solid set of rules covering all possibilities would be beyond the scope of this thesis. As seen in [20], creating such a system with acceptable performance is an entire study in itself. Using an interactive approach eliminates the need of collecting data up front of the fitness evaluation. The population would then be measured against an elite of good designs, chosen by users during run-time. There are two major ways of doing this: The first one could be to generate a new design for the same element for each unique user of a live web page. By doing this, it would be possible to use the intelligence of crowds to improve a design over time. The main flaw of this approach however is the need for the users to provide constant feedback in some form. Furthermore, randomly generated designs may be unreadable, causing the user to loose interest. Constantly changing the design of the page is also imagined to serve as a source of confusion. The other way of performing this interactive fitness evaluation is the use of a client program, where the user is presented with a selection of randomly generated designs similar to Khajeh et al. The user would then select designs in each generation based on their personal preference, driving the design process towards the goal. Based on these advantages, an interactive fitness function will be used in this thesis.

Due to its interactive nature, the resulting program would need a graphical user interface. Several such frameworks for a variety of programming languages exist, including the Qt framework for C++[21], JFC/Swing for Java[22] and Windows Forms for C#[23]. Python could also be an alternative. Since the focus is web design, using PHP might appear as the obvious choice for the implementation. However, PHP requires a web server to run, which in turn needs extra time to configure. In short, each of these platforms come with their distinct advantages and disadvantages beyond the scope of this report, and in the end, C# with Windows Forms was chosen as the programming language for the project. Visual Studio 2017 was used as the development environment. The reason for this was a plan to reuse the code at a later stage of development, as the original idea was to make an entirely web-based application using the asp.net platform, which is C#-based. In addition, previous projects using Windows Forms to create interactive user interfaces meant that existing, working solutions could be retrofitted for this new project. This would mean less time needed to create a working system, and more time spent on the implementation of the genetic algorithm itself.

4.1 Major steps of implementation

The implementation of the interactive genetic algorithm-based web element designer consisted of the following major steps:

- Creating a working interactive genetic algorithm implementation using a single property
- Implementing different crossover types and mutation
- Extending the working implementation to include more properties
- Selecting a design not present in the population as goal
- Generating CSS code based on representation
- Previewing designs as CSS in the program
- Adding support for more elements
- Introducing a set of basic rules in the form of helper variables
- Investigating ways to increase population diversity
- Giving the user more control over the population
- Normalization of weights in fitness function
- Introducing an element of learning based on usage

Getting something simple up and running, and being able to verify its functionality was the first major step. This created a solid foundation which could be expanded throughout the project. After the initial concept of the implementation was verified, adding different crossover types and mutation was the next step, as they are vital parts of GAs. It would also be necessary in order to compare performance and achieve different results. With all of this in place, the chromosome was extended to support multiple properties, allowing for more feature-rich designs to be generated. Testing to see if a randomly generated design not present in the initial population could be approximated was used as a way of testing the mutation performance. Using the new properties to generate proper CSS was a natural next step, as the task revolves around web design. The need for this will also be discussed in greater detail later. Using the existing implementation on other web elements only required minor modifications, as most CSS code may be used on any element. Rules was a necessary addition in order to restrain the initial population, in addition to produce subjectively better looking designs. Due to the rules, population diversity became somewhat limited. Investigating methods to counter this was a direct result of the addition of the rules. To reduce convergence time, various ways of controlling the population during run-time was implemented. Due to the large differences between values in CSS, normalization of values in the fitness function was required. Finally, in order to absorb quality measures embraced by humans, a learning system was added.

4.2 The first implementation

In the first iteration of the solution, the goal was to implement a simple interactive genetic algorithm similar to the work of Khajeh et al. The user would be presented a set of randomly generated colors. The user would then click on a color, which would be flagged as the fittest individual. The genetic algorithm runs until a termination condition is met, in this case, the fitness sum of the population.

Colors were represented by integer values for Red, Green and Blue (RGB) coded as binary strings:

Property	Range
Background color R	$[0, 255]$
Background color G	$[0, 255]$
Background color B	$[0, 255]$

Table 3: Representation of colors

As an example, the following binary string produces the following color:

$$011110110011111110001011_2 = (123, 63, 139)_{10} \tag{1}$$

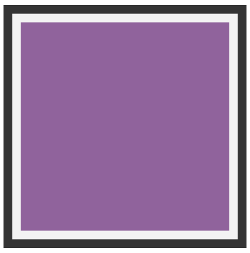


Figure 5: Color produced by eq. 1

The random RGB values were generated using C#'s *Random*-class and stored as separate class member variables. Crossover was conducted using single point crossover:

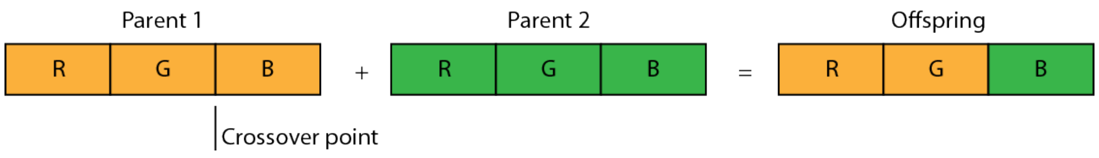


Figure 6: Single point crossover on RGB colors

In fig. 6, a crossover point is randomly selected on the first parent, and genes before this point are copied to the offspring. The rest are copied from the other parent. Mutation was carried out by iterating through the binary strings, and flipping bits according to a mutation chance. In this thesis, mutation chance was set to 5%.

To compute the fitness of each individual, the difference between each RGB component was used:

$$f = |R_{fittest} - R| + |G_{fittest} - G| + |B_{fittest} - B| \tag{2}$$

As seen in equation 2, a higher value of f corresponds to a lower fitness value. This inverse fitness measurement was consistently used throughout the project.

The fittest individual of each generation was selected by the user through an interactive process. To avoid having the user to keep clicking the same individual for every generation, the ability the let the algorithm run for itself until termination was added. The implementation of the solution consisted of three major parts: A chromosome class, a helper class for population operations, and a user interface class. The population helper class, dubbed the "Genetic Engine", generated the initial population, managed the selection process, and kept track of population fitness.

The complete genetic algorithm used in this implementation:

Algorithm 1 Basic interactive GA implementation

```
 1: Generation $g \leftarrow 0$
 2: Generate random initial population $p$
 3: User selects fittest individual $i$
 4: while $p$.fitness $>$ threshold do
 5:     for each Individual $c$ : $p$ do
 6:         $c$.fitness $\leftarrow c - i$ (eq. 2)
 7:     end for
 8:     Run selection process using algorithm 2 or algorithm 3 to create offspring $o$
 9:     for each Binary string : $o$.Chromosome do
10:         Flip bits according to mutation chance
11:     end for
12:     $g \leftarrow g + 1$
13: end while
```

The first selection scheme to be implemented was the elitist selection, with elements from the Genitor implementation similar to the one described by D. Whitley in[24]. This means that a new offspring replaces a less fit individual, which in turn leads to evolutionary pressure.

Algorithm 2 Elitist selection scheme

```
1: Sort $p$ based on fitness
2: $p' \leftarrow$ top 50% of $p$
3: for j:=1 to 10% of $p$.Size do
4:     Select random parent $a$ from $p'$
5:     Crossover($p[p.\text{Size}-j]$, $a$)          ▷ The least fit are replaced through crossover
6: end for
```

After this, the roulette wheel selection scheme was implemented:

Algorithm 3 Roulette wheel selection scheme

```
1: List $L$
2: Sort $p$ based on fitness
3: $M \leftarrow p$.Last.fitness
4: for each Chromosome $c$ : $p$ do
5:     $L \leftarrow M - c$.fitness
6: end for
7: for j:=1 to 10% of $p$.Size do
8:     Parents $a$, $b$
9:     for k:=1 to $p$.Size do                ▷ This loop is run one time for each parent
10:        Generate random number $r \in [0, 1)$
11:        if $r \geq L[i] \ / \sum L$ then $a, b \leftarrow c$
12:        end if
13:    end for
14:    $p[p.\text{Size}-k] \leftarrow$ Crossover($a$, $b$)
15: end for
```

To convert genes to and from their binary form, C#'s *System.Convert* class was used. This class is able to convert integer values to any specified base. Using this method, a problem regarding binary strings quickly appeared: Strings representing RGB values range from 0 - 255, and when small values are converted from decimal to binary, it becomes impossible to mutate them into higher values later, due to the string's short length. To remedy this, each property has an associated expected binary string length. This value is passed to a helper function, which adds leading 0's to the binary string after conversion. This way, small values mutating into a larger value becomes possible. Example:

$$20_{10} = 10100_2 \rightarrow mutate() \rightarrow 11111_2 = 31_{10} \tag{3}$$

$$20_{10} = 00010100_2 \rightarrow mutate() \rightarrow 11111111_2 = 255_{10} \tag{4}$$

3: Resulting binary string produced by best-case mutation using only C#'s converter. As one can see, small randomly generated numbers will get stuck in this range. 4: Resulting binary string after best-case mutation with manually added leading 0's. The full numerical range is within mutation reach.

Results were rendered using a *UserControl*-panel with a custom *OnPaint()* method. *OnPaint* allows geometric shapes, colors and text to be created and rendered in a window. Each chromosome was associated with a panel rendering the color. The panels were presented in a 10×10 grid, meaning that the population size was 100. At this point of development, no CSS was involved.

4.3 Extension of chromosome properties

With a simple, single property implementation of the interactive genetic algorithm working, the next step was to extend the code to include elements one might expect to find on a web page. Gene strings representing the placement of a navigation bar, background color and a front page image were added to the chromosome class. The location of the navigation bar was encoded as an integer with values 0 - 3, where each integer corresponded to left, right, top or bottom of the page. In short, the goal of this iteration was to see of the working simple implementation could be extended by adding new properties to the basic chromosome.

Representation of new properties:

Property	Range	Remark
Background color R	$[0, 255]$	
Background color G	$[0, 255]$	
Background color B	$[0, 255]$	
Navbar color R	$[0, 255]$	
Navbar color G	$[0, 255]$	
Navbar color B	$[0, 255]$	
Navbar location	$[0, 3]$	0 = left, 1 = right, 2 = top, 3 = bottom
Front page image	$[0, 3]$	4 predefined images were available

Table 4: Representation of implemented web page elements

As the amount of properties grew, a list structure was used instead of separate class member variables to hold the binary strings. By doing this, extension simply became a matter of adding new genes to the list, and iterating through the list to extract the properties. Because of this, the Genetic Engine underwent minor updates to accommodate the list structure.

Two new crossover schemes were implemented:

Algorithm 4 Arithmetic crossover

```
1: for each Gene $g$ in Chromosome $A$ and $B$ do
2:     $d_A, d_B \leftarrow g_{10}$                ▷ Converts from binary to decimal
3:     $g' \leftarrow Average(d_A, d_B)$
4:     Replace $g$ in $A$ with $g'$
5: end for
```

The arithmetic crossover works by averaging each value of the parents.

Algorithm 5 Multi-point crossover based on the Fisher-Yates modern shuffle algorithm

```
1: Chromosome $A$, $B$
2: $L \leftarrow A$.Genes
3: $n \leftarrow L$.Size
4: for i:=0 to $n$ do
5:     Array $d \leftarrow i$
6: end for                                   ▷ As described in[25]
7: for i:=1 $\leq n$ do
8:     Generate random number $r \in [0, n - i)$
9:     Swap($n - i$, $r$) in $d$
10: end for
11: Generate random number $k \in [1, n)$
12: for i:=0 to $k$ do
13:     $A$.Genes[$d[i]$] $\leftarrow$ $B$.Genes[$d[i]$]
14: end for
```

Multi-point crossover works in a similar way to the single point version. The main difference is that multiple genes are selected at random from each parent, and copied to the offspring:

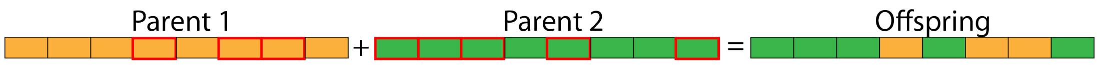


Figure 7: Multi-point crossover

Mutation remained untouched. The fitness function was extended to accommodate for the additional properties, but still uses the sum of differences:

$$f = \sum_{i=0}^{n}(|p_i^{fittest} - p_i|) \tag{5}$$

The fittest individual was still selected by the user. A new addition at this point was the possibility to select a random design not present in the initial population as the fittest individual. Only selecting the fittest individual as an existing member of the population meant that the desired genetic material was already present. By allowing the GA to approximate this unknown design, mutation was put to the test.

No changes to the GA implementation itself were necessary at this point, as it still supported the new features. The new properties added were based on those found in CSS and HTML, but no actual CSS code was added to the implementation.

The featured front page image was a bitmap loaded at program initialization, and rendered in the *UserControl* panel using the *DrawImage()*-method in the Windows Forms Graphics library.

4.4 Generation of CSS based on chromosomes

Using a list structure to hold binary gene strings made the chromosome easy to extend, and the class was extended to include support for rounded corners and element hover and active states. These new values were represented in an identical way to the ones discussed in the previous section.

The first element supported was the button-tag. To support other popular HTML-tags, polymorphism was introduced in the class structure. Each element inherits from an abstract base class with common properties, examples include fitness and background color, and unique properties for each tag. The chromosome list was split in two parts, one for all common genes, and one for any unique genes an element may require. The list of common genes is always the same length for any element, while the unique list may vary in size depending on the inherited class.

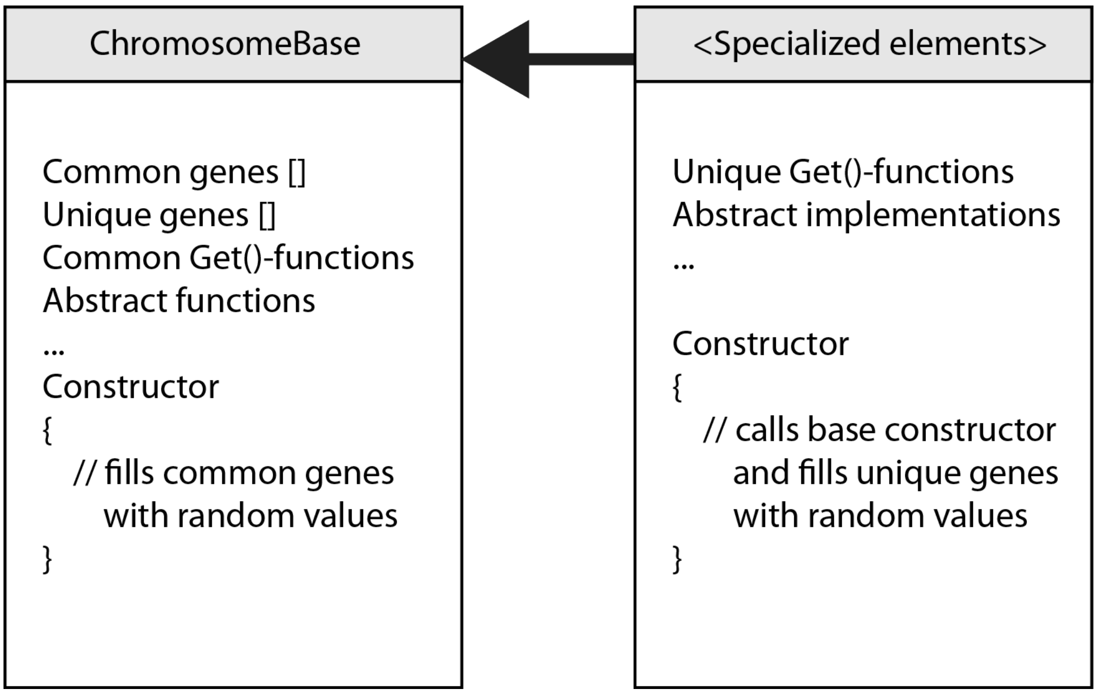


Figure 8: Simplified UML-diagram regarding chromosome inheritance

After the introduction of polymorphism, h1-, p-, input-, and table-tags were implemented, each with their unique genes in addition to the basic features.

Base chromosome		
Background color		
Border colors[]		
Border styles[]		
Border widths[]		
Border radii[]		
Font size		
Paddings[]		
Table	h1	p
Border collapse	Text transform	Text shadow
Header text color	Text shadow	Word spacing
Cell text color	Text align	Line height
Header font size	Text decoration	
Striped row color	Letter spacing	
Header text align		

Table 5: Simplified list of common and unique properties for each implemented element

Table 5 describes inheritance. All chromosome variants inherit the properties of the base chromosome, in addition to providing their own unique features. As an example, the p-tag chromosome has a Background color, Border colors, Border styles Border widths, Border radii, Font size and Paddings, in addition to the element-specific properties Text shadow, Word spacing and Line Height. The button- and the input-elements are the exceptions, as they do not have any additional properties. A graphical example:

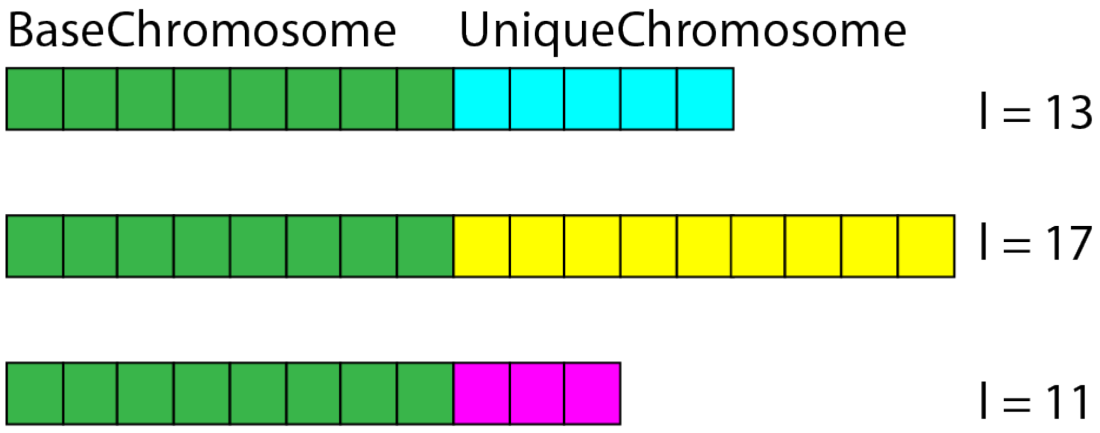


Figure 9: Graphical representation of common and unique properties

Common properties are found in BaseChromosome, while the unique properties are in the UniqueChromosome-part. Together they make up the whole design representation.

Border style requires correct syntax, and as discussed a special mapping function would be needed. This was done by representing the style as an integer corresponding to an index in an array of strings:

Index	Value
0	dotted
1	dashed
2	solid
3	double
4	groove
5	ridge
6	inset
7	outset
8	none
9	hidden

Table 6: Integer - string mapping

Due to the additional properties, the seeding method was changed. The *GetHashCode()* of the *Guid*-class in the *System*-namespace was used as a seed. The Globally Unique Identifier (GUID) has a very low probability of being duplicated, according to Microsoft[26]. The reason for this will be discussed in the results-section. The initial population size was also reduced to 60. Crossover, mutation and the fitness function were all untouched during this iteration.

Using Windows Forms graphics did not do justice to the new additions, and the need for a more true-to-life rendering system arose. By extracting the properties from each design, and mapping them to their representative element, it would be possible to export a CSS-file and view the designs in any browser. Using a new helper class, the integer values of the chromosome were mapped to their respective CSS-string counterparts.

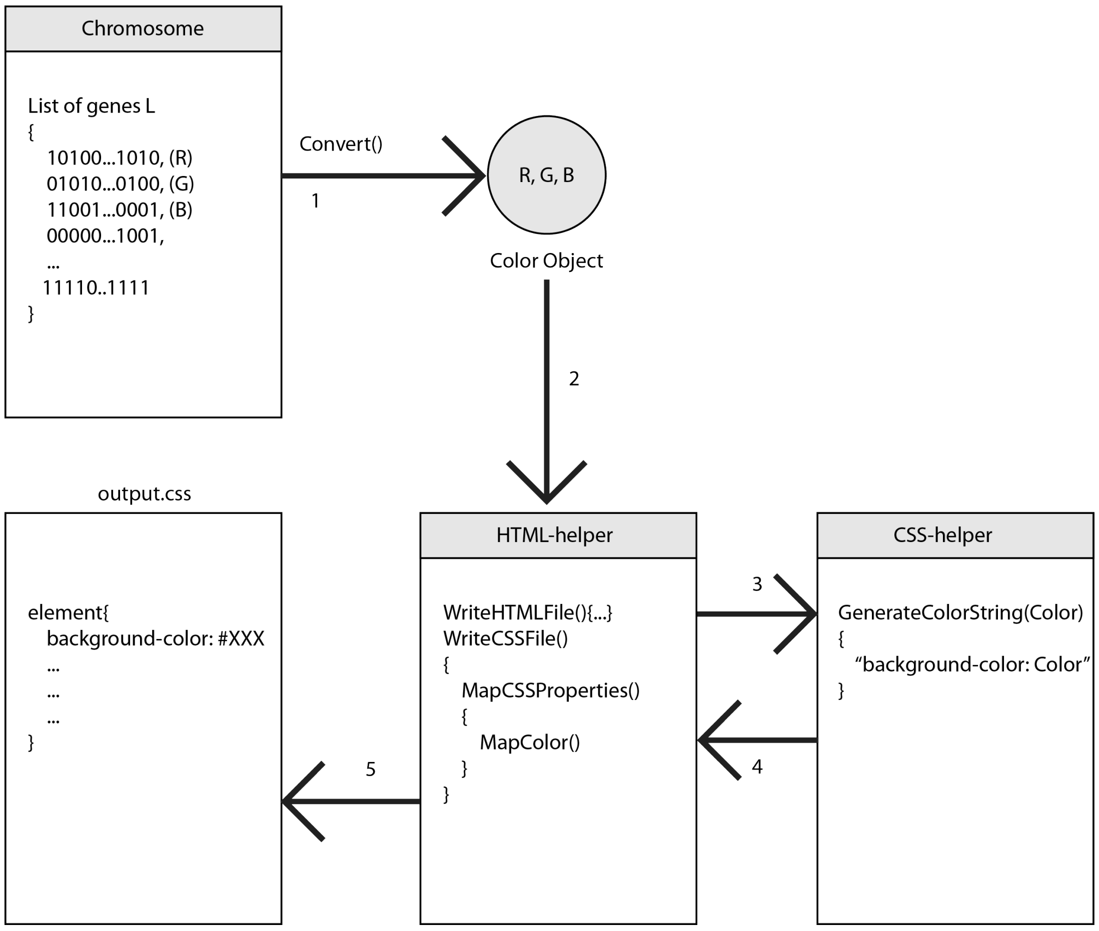
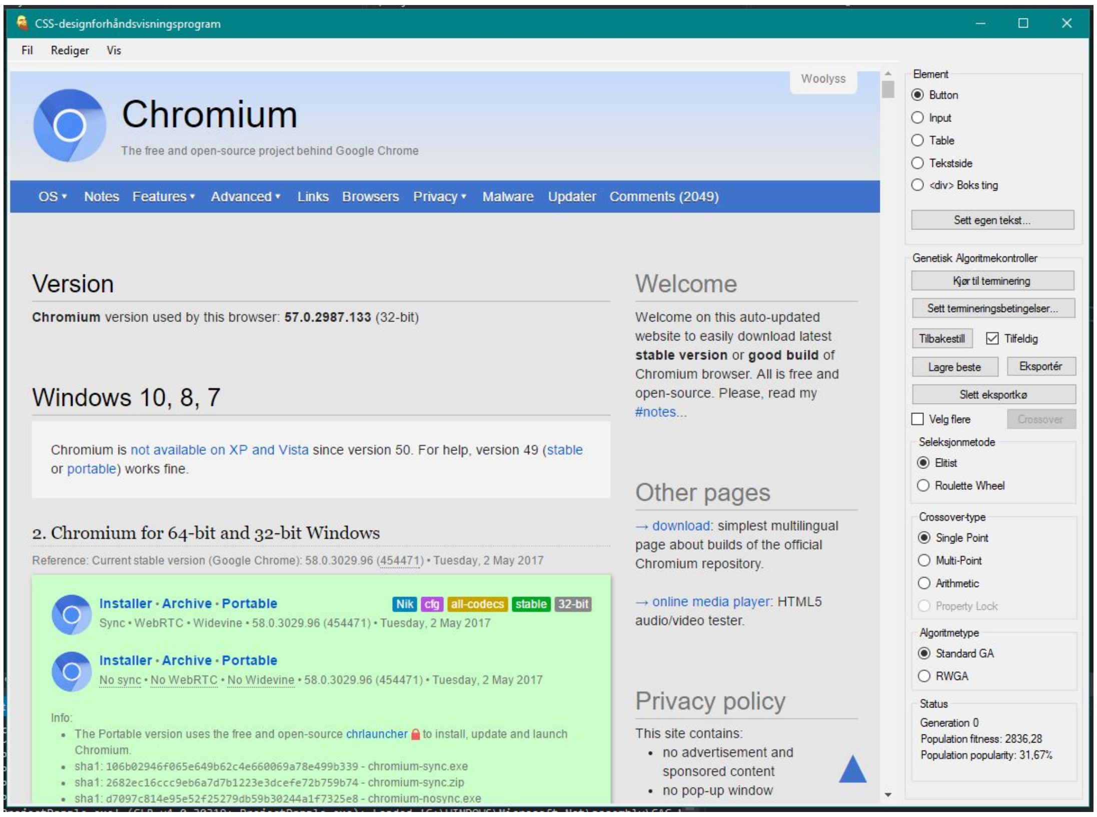


Figure 10: Mapping of the background color property from chromosome to final CSS-file

The above figure shows the basic steps of mapping a single property to CSS using the helper classes. In step 1, binary strings are converted to their decimal, or if applicable, C# color class equivalent. The converted value is passed through to the HTML helper class in 2, which based on the index in the gene list, calls the appropriate string generator method in the CSS helper in 3.

The complete string is passed back to the HTML helper, 4, which finally writes the CSS class file with all its properties using the *StreamWriter*-class found in the *System.IO*-namespace of C# (step 5). The user may then open the preview HTML-file in any web browser to review the results.

The goal was to integrate the browser preview into the application itself. Achieving this would allow the user to monitor the status of the genetic algorithm and review each generated design without the need to switch between multiple applications. Windows Forms offers an embedded web browser component, allowing web content to be displayed within the program. According to the Microsoft Developer Network, the browser engine is the current version of Internet Explorer installed on the user's system[27]. However, after implementing the CSS preview using the web browser component, elements based on the newer CSS3 standard did not render correctly. By consulting W3schools.com's CSS3 Browser Support Reference[28], it became apparent that the browser uses an older version of Internet Explorer. Running browser version verifiers within the program, like the acid test[29], confirms this. As a result of this limitation of the built-in system, the search for an alternative web browser began. There exist multiple projects which aim to replace the default web browsing capabilities of Windows Forms[30, 31, 32], and in the end, the chromium-based CefSharp library was chosen for the solution. The library is well-documented, easy to implement and supports modern features like CSS3 and HTML5, as it is based on the latest release of the chromium embedded framework, which at the time of writing is version 57, as seen in fig. 11.



Chromium

The free and open-source project behind Google Chrome

Version

Chromium version used by this browser: **57.0.2987.133** (32-bit)

Windows 10, 8, 7

Chromium is not available on XP and Vista since version 50. For help, version 49 (stable or portable) works fine.

2. Chromium for 64-bit and 32-bit Windows

Reference: Current stable version (Google Chrome): 58.0.3029.96 (454471) • Tuesday, 2 May 2017

Installer · Archive · Portable
Sync • WebRTC • Widevine • 58.0.3029.96 (454471) • Tuesday, 2 May 2017

Installer · Archive · Portable
No sync • No WebRTC • No Widevine • 58.0.3029.96 (454471) • Tuesday, 2 May 2017

Info:
- The Portable version uses the free and open-source chrlauncher to install, update and launch Chromium.
- sha1: 106b02946f065e649b62c4e660069a78e499b339 - chromium-sync.exe
- sha1: 2682ec16ccc9eb6a7d7b1223e3dcefe72b759b74 - chromium-sync.zip
- sha1: d7097c814e95e52f25279db59b30244a1f7325e8 - chromium-nosync.exe

Welcome

Welcome on this auto-updated website to easily download latest **stable version** or **good build** of Chromium browser. All is free and open-source. Please, read my #notes...

Other pages

→ download: simplest multilingual page about builds of the official Chromium repository.

→ online media player: HTML5 audio/video tester.

Privacy policy

This site contains:
- no advertisement and sponsored content
- no pop-up window

Element
[✓ Button] Input Table Tekstside <div> Boks ting

Genetisk Algoritmekontroller
[✓ Tilfeldig]
Velg flere

Seleksjonmetode
[✓ Elitist] Roulette Wheel

Crossover-type
[✓ Single Point] Multi-Point Arithmetic Property Lock

Algoritmetype
[✓ Standard GA] RWGA

Status
Generation 0
Population fitness: 2836,28
Population popularity: 31,67%

Figure 11: Page displaying the client's version of Chromium inside the application

Up until this point, the goal of the algorithm was to design both the page layout (HTML), and the appearance of each element on the page (CSS). However, it became clear during this phase of development that combining both page layout and element design led to an incomprehensible solution space. Furthermore, layout optimization may only be done so many ways, and it has already been examined by Oliver et al. in [13]. As discussed, the introduction of new properties in CSS3 allows for expansion upon their work regarding element design. Due to this, all genes representing page layout were removed, and the algorithm would be used to design individual elements exclusively. Another helper class was introduced, which purpose was to generate the preview HTML-file. The preview page consisted of a table, with cells for each element with a unique CSS-class.

To make the user able to click on an HTML-element inside the web browser component, JavaScript was used. CefSharp allows JavaScript function calls to be passed through and handled in the C#-code. Using this feature, each cell was given an *onClick(**id**)* in HTML, which passes the index of the selected design to a handler telling the Genetic Engine which design is selected.

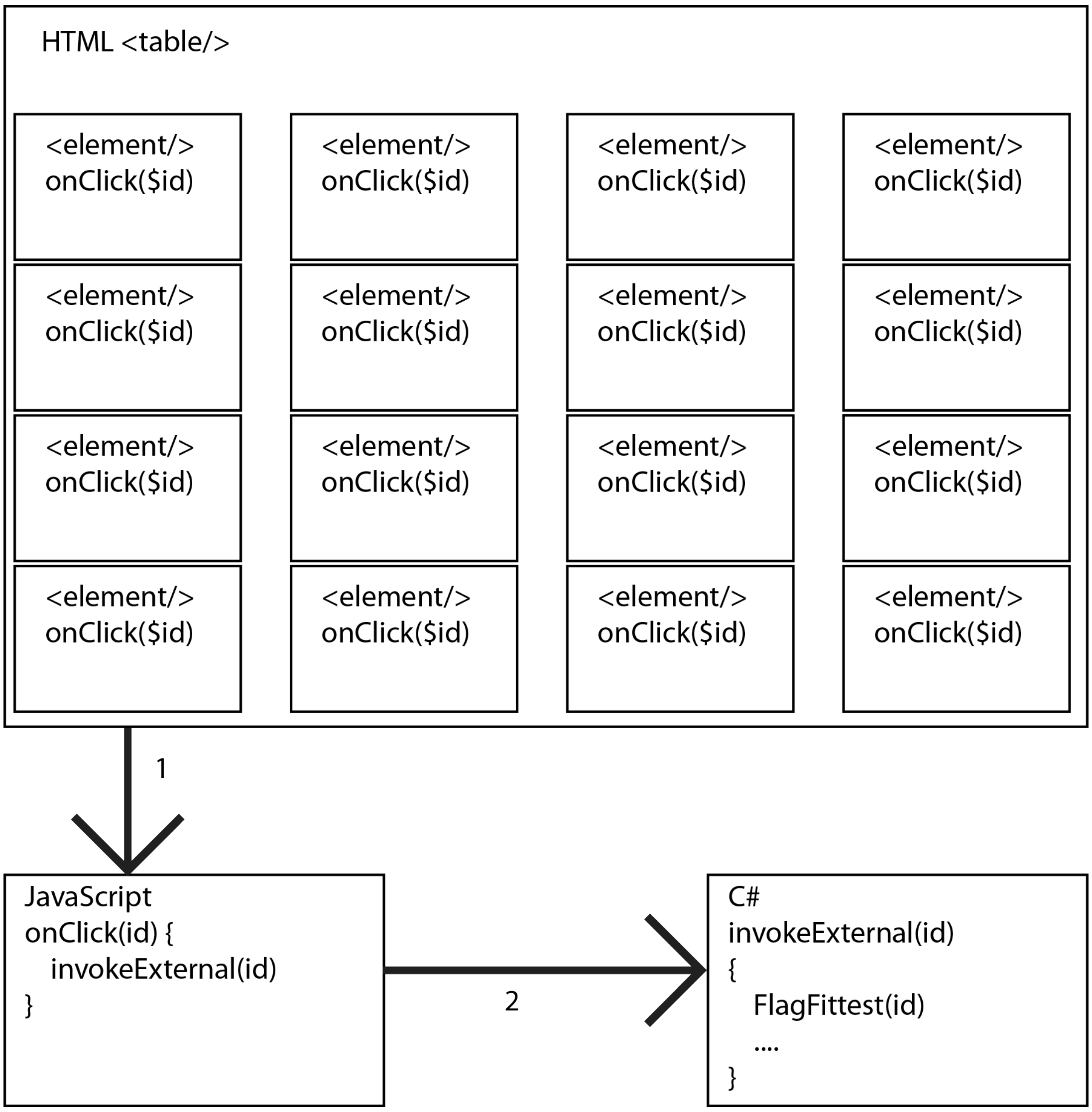

HTML <table/>
<element/> onClick($id)
1
JavaScript onClick(id) { invokeExternal(id) }
2
C# invokeExternal(id) { FlagFittest(id) .... }

Figure 12: Structure of HTML preview file

When the HTML file illustrated fig. 12 is generated, each cell is given an id based on its index in the list of chromosomes held in the Genetic Engine. Using the same interactive code as in the previous iterations, the selected id is set as the most fit individual. To avoid hard-coding of HTML code within the source code, most of the content of the HTML preview page is defined in separate text files, which are loaded and combined at run-time. The exception is the table itself, which is a string generated in a series of for-loops.

4.5 Implementation of basic rules using helper variables

Giving the algorithm complete freedom to explore all possible recombinations of design features in the solution space often resulted in arguably sub-optimal solutions with respect to aesthetics. This was especially true for the table-tag, as seen in fig. 13.

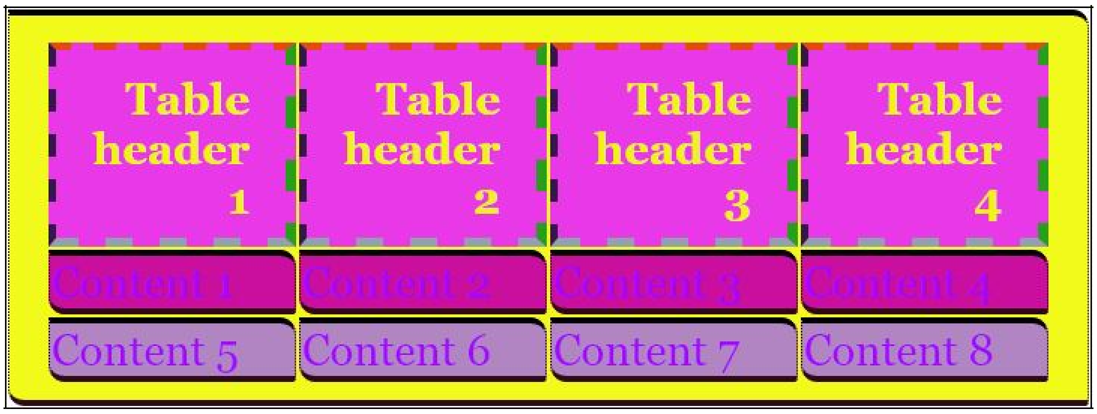


Figure 13: Example of generated table design without any constraints

As mentioned in section 3, each element may have one or more unique border styles, colors, radii and widths, and the initial population was generated with this in mind. However, by observing existing web design, uniformly shaped elements are dominant. Introducing rules in the form of helper variables, determining how many of the border properties to be unique, was a way of limiting the possible combinations. The rules made it possible to reduce the number of combinations of unique properties, without limiting the solution space. Such designs may still occur in the population, albeit in smaller numbers. The helper variable was a randomly generated number with values 0 - 3, and encoded like any other gene in the chromosome, similar to the navbar location gene present in the earlier implementation.

Property	Range
Border colors[]	$([0,255] \times 3)^4$
Border color limiter	$[0,3]$
Border radii[]	$[0,255]^4$
Border radius limiter	$[0,3]$
Border widths[]	$[0,15]$
Border width limiter	$[0,3]$
Border styles[]	$[0,9]$
Border style limiter	$[0,3]$

Table 7: Individual border properties and their limiter variables

As seen in table 7, each CSS property with multiple attributes has an associated helper variable, giving each design a more refined appearance. In short, the variable tells the CSS-helper class how to structure the final string. When converting from binary strings, properties with multiple values are returned as arrays of length 4. The helper value is used as the limit when iterating through each array of properties. Another benefit of introducing the helper variables was faster convergence, which will be discussed further in the results section of this paper.

Another change was the introduction of expected lengths for each binary gene string. As previously mentioned, most genes of the chromosome represent different properties of a CSS-class, which all have varying numerical ranges. Linking each gene with an expected length was a necessary addition as the number of genes increased. Hard-coding each index of the list of binary strings with its associated length was becoming impractical and cumbersome. By adding the length-variable, arithmetic crossover became more flexible, as it simply became a matter of iterating through the list of strings, carry out operations, and still know the appropriate length of binary string when converting back into binary form. Elitist and roulette wheel crossover remained unchanged, in addition to the mutation process.

The interactive aspect of the algorithm was improved in the form of a new selection scheme. This allowed the user to manually select two or more designs for crossover. The crossover scheme used is the arithmetic crossover, taking the average of each design when producing offspring. It was assumed that the user would select two or more designs, and expect the produced offspring to be a mix between the parents analogous to nature. As an example, selecting blue and red colored designs for crossover produces a purple child. The fitness function was not altered.

Furthermore, adding more properties while working with binary strings introduced another challenge regarding the numerical range of each property. Up until this point of development, all encoded properties utilized the full range of their binary values. For instance, each component of the RGB color property range from 0 to 255, which translates to 0 - 11111111 on binary form. On the other hand, the border style property only has a total of 10 legal values (table 1), which needs a total of 4 bits to be represented, as $10_{10} = 1010_2$. As one may imagine, mutation is likely to violate this constraint, a worst-case scenario resulting in $1111_2 = 15_{10}$. When mapping from the integer values to CSS-strings in the helper class, illegal values were handled and the property's default CSS value was returned. An unintentional side effect of this way of handling overflow was a larger probability of default values occurring in the population. In the case of the border style property, there was now a $\frac{6}{15}$ chance of the default value being returned from the mapping function.

Rendering and preview was conducted using the same method as the previous implementation.

4.6 Population diversity exploration and population control

In order to explore a wider variety of design variations in a shorter amount of time, population diversity had to be increased. The introduction of rules also contributed to reduced diversity. Applying the Random Weighted Genetic Algorithm concept [33] was deemed a possibility. Optimizing a design in the context of this project was considered a multi-objective problem, as there are multiple properties including shape, color and font size to be optimized simultaneously. Minor changes were needed to support the introduction of the weights-system, as the Genetic Engine already utilized a list data structure for each gene. Assigned weights were stored in a list with corresponding indices to the list of genes, and each gene was multiplied with its weight while computing the fitness.

$$\vec{f} = \left(\begin{bmatrix} g_1^1 \\ g_2^1 \\ g_3^1 \\ \vdots \\ g_n^1 \end{bmatrix} - \begin{bmatrix} g_1^2 \\ g_2^2 \\ g_3^2 \\ \vdots \\ g_n^2 \end{bmatrix} \right) \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} \tag{6}$$

An example:

$$\vec{f} = \left(\begin{bmatrix} 12 \\ 131 \\ 5 \\ 4 \\ 42 \end{bmatrix} - \begin{bmatrix} 0 \\ 54 \\ 27 \\ 19 \\ 41 \end{bmatrix} \right) \times \begin{bmatrix} 7 \\ 1 \\ 3 \\ 21 \\ 138 \end{bmatrix} = \begin{bmatrix} 84 \\ 77 \\ 66 \\ 315 \\ 138 \end{bmatrix} \tag{7}$$

Thus, as seen in 7, properties with higher weights would impact the fitness function more, even if the difference itself is relatively small. Consequently, by giving important properties larger weights, the algorithm will converge towards solutions where desired values are present. The above examples consider common weights for all chromosomes in the population. However, at this stage of development, each chromosome had its own randomly generated vector of weights. Doing this meant that each chromosome had its own "idea" of which properties were important, leading to increased population diversity.

For consistent testing and benchmarking of the performance of the algorithm, a way to generate a control population had to be implemented. By providing a static seed to the random number generator, it was possible to generate an identical initial population for each run of the program. Since the initial population was generated in a for-loop, the seed used was the iterator variable of the loop.

Rules regarding readability according to W3[34] were also introduced and enforced on the initial population using the following formula:

$$brightness = \frac{R_{text} \times 299 + G_{text} \times 587 + B_{text}}{1000} \tag{8}$$

$$difference = |R_{text} - R_{background}| + |G_{text} - G_{background}| + |B_{text} - B_{background}| \tag{9}$$

To pass the test according to W3, brightness and difference must be greater than 128 and 500, respectively. When generating the initial population, individuals failing the test are not added, and a new individual is generated and tested.

Inspired by the work regarding design mining in [20], a way to parse CSS-files and extract the design features was implemented. The goal was to be able to base an initial population on a known design, and allow the user to explore variations of this. However, as earlier demonstrated, there is no standard way of structuring a CSS-class, and as a result, the parser only works with simple CSS-classes.

Alongside basing an initial population on a known design, it could also be expedient to manually set the parameters of the random number generation. Another window with input fields for minimum/maximum values for each property was added. Using a similar approach to the property parser, the values were stored as lists in a simple data structure, along with expected length for each property. This way, initializing a new chromosome became a matter of simply looping through the lists, and passing the values to the random number generator already present in the chromosome class.

As the number of design features of each chromosome grew, arriving at the "perfect" solution became less and less likely. Giving the user the ability to keep certain, subjectively pleasing features of each design, and leave the algorithm free to explore the rest was imagined to reduce convergence time, and reach better results. This implementation is similar to the wild-card scheme described by Whitley in [24]. The implementation works by the introduction of a new crossover scheme: genes corresponding to properties flagged by the user may not be overwritten by crossover, mutation chance is increased, and the unlocked properties are given weight 0 in the fitness function. This was dubbed the "Prop-lock crossover".

Property	Locked?
Background color	Yes/no
Shape	Yes/no
Font size	Yes/no
Text color	Yes/no
Padding	Yes/no
Hover color	Yes/no
Border colors	Yes/no
Border widths	Yes/no
Border styles	Yes/no

Table 8: List of lockable properties

In table 8, genes are grouped together to form more comprehensible properties of each design, in an effort to make the selection process easier for the user. The chromosomes share a common vector of locks, similar to the implementation of weights mentioned earlier. Crossover is conducted using the following steps:

Algorithm 6 Property locks crossover

```
1: Mutation chance $m \leftarrow 100\%$
2: Define target $T$                              ▷ Target is the design selected by the user
3: Select two random parents $A$, $B$ using algorithm 2 or algorithm 3
4: if $T = A$ OR $T = B$ then
5:     Copy all locked properties from $T$ to $A, B$
6: end if
```

This way, locked properties are ensured to be transmitted throughout the population, leaving unlocked properties free to be explored using the already existing population diversity to fill in the wild-cards. Wild-cards are given weight = 0, using the existing weights vector. As such, these genes are ignored by the fitness function, and may hold any value. A consequence of this was the removal of separate weights vectors for each chromosome. It would have been possible to keep the existing implementation of separate weights alongside the common weights, however, this was not prioritized, as this has already been discussed in [33]. Population diversity is still maintained through the use of wild-cards and increased mutation rates. The fitness function was changed to accommodate the addition of weighted properties:

$$f = \sum_{i=0}^{n}((|p_i^{fittest} - p_i|) \cdot w_i) \tag{10}$$

where w_i is the associated weight. In addition, individuals with low readability according to equation 8 and 9 are given a fitness penalty of 100. The exception is if the user deliberately selects a low-scoring individual as the fittest. The test is ignored, as it is assumed the user truly wants the selected design, disregarding readability.

Up until this point, the algorithm was programmed to terminate after a hard-coded population fitness was achieved. After the introduction of weights, this value was seldom reached, as the fitness sum became severely offset. Furthermore, the user may even want the algorithm to terminate after a certain amount of generations. The need for user-defined termination conditions led to the addition of a window where the user may select either population fitness or the number of generations as the condition to check, in addition to adjust the threshold value itself. This manual control, along with the property-locking crossover scheme described above, increased the degree of interactivity in regards to the GA.

The user was also given the option to set the type of weighting to use through the GUI. Initially, all weights were set to uniform, with value $w_i = 1$, and could be randomized as described in 7.

The parser works by reading all lines of the file, and looks for predefined properties and their values using various techniques. The values are stored in a simple data structure, and passed to a special constructor for the chromosome class. The parsed data is mapped into chromosome form on the same format used internally by the program. The parser loads all CSS-files in a predefined folder, and displays the designs in a separate window using the same renderer as the population preview. The user may then select one of the presented designs, and base the population on this. Multi-point crossover with the selected design is conducted on each member of the initial population to introduce the genetic material.

Based on feedback, viewing a wall of designs may overwhelm the user. A static preview page was introduced to remedy this, where the user is able to view the generated design in the context of a full web page with other design elements. The selected design is written to a separate CSS-file, and the embedded browser is redirected to the pre-made web page. Each element on the page have their own hard-coded id, which is used to link the CSS-class to the element in question. In addition, the user may set a custom text string as the element's inner content from the GUI. Using a custom string of varying length lets the user see the resulting design in various scenarios, since elements scale depending on content.

Lastly, the chromosome class was extended. Changes included the additions of transparent and gradient backgrounds using CSS3 standards. The h1- and p-tags were joined to a single element, combining design of header text and page text using a selection of Google's free fonts available at [35].

4.7 Normalization of weights

Using the difference to calculate fitness worked sufficiently when working with properties of similar numerical range, however further investigations revealed that using this approach, colors were automatically given more weight. This was because of their larger values compared to the other properties, which are typically in the range 0 - 30 (see table 7). Using the already implemented gene data structure, it was possible to add a max value variable to each gene. Dividing each property by its respective maximum value in the fitness function gives a number between 0 and 1, meaning that all properties are weighted equally when computing the difference.

An example (unscaled):

Property	Value 1	Value 2	Difference
Color R	212	18	194
Color G	32	114	82
Color B	128	0	128
Font size	16	18	2
Border style	1	5	4
Border width	0	8	8

Table 9: Example of different property value ranges

An example (scaled):

Property	Value 1	Value 2	Max value	Scaled difference
Color R	212	18	255	0.76
Color G	32	114	255	0.32
Color B	128	0	255	0.5
Font size	16	18	32	0.01
Border style	1	5	9	0.4
Border width	0	8	15	0.5

Table 10: Example of different property value ranges (scaled)

As one might imagine from examining the above tables, not scaling the larger values associated with colors makes the fitness function biased to these properties, giving them precedence over equally important, albeit lower weighted properties. Scaled values mean that all properties compete on equal grounds in the fitness function. In the above example, Color B and Border width are both given 0.5 as difference, as they are both equally different relative to their max values. The consequences of this change will be further looked upon in the results section of this paper. Random weights generated by the multi purpose-inspired implementation were also normalized, meaning the random range was set to $[0, 1)$.

Crossover and mutation were not altered during these changes.

The fitness function was changed to normalize the values in the following way:

$$f = \sum_{i=0}^{n} \left(\frac{|p_i^{fittest} - p_i|}{MAX_i} \cdot w_i \right) \tag{11}$$

No other changes were done during this iteration.

4.8 Introduction of learning and classification

With the interactive genetic algorithm implementation working for a set of HTML-tags and CSS design elements, the time was come to implement the learning element as described in the introduction. The goal was to combine crowd-sourced human feedback with machine-based exploration of the solution domain. This was done by using the K-nearest neighbor classification algorithm.

The K-nearest neighbor algorithm (K-NN) is a simple, yet effective classification algorithm. One of its main strengths is that little or no prior knowledge of the distribution of data is needed. The algorithm is trained using a set of already classified data samples. Features of the data are mapped to a multi-dimensional geometric space, and new elements are classified based on the geometric distance to the training data. K is the number of neighbors to check when classifying the new sample [36].

As an example, consider the following: Given a set of 15 points in 2D representing some feature, where 8 are classified as "Red" and 7 are classified as "Cyan". A new sample is to be classified using $K = 7$, meaning that the 7 closest neighbors (enclosed in circle) are to be considered:

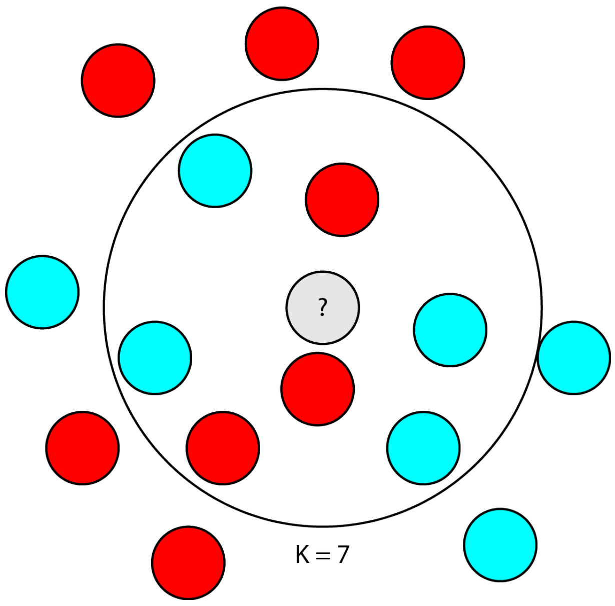


Figure 14: K-NN classification in 2D

As one may see from the example in fig. 14, the new member will be classified as Cyan, with match percent $p = \frac{4}{7} = 57\%$. This is because the most frequent class among the 7 nearest neighbors is "Cyan".

No changes were done to the generation of the initial population in this iteration. The crossover schemes and mutation method were also untouched.

The fitness function was changed to the Euclidean distance formula (with scaling and weights), as it is also used during K-NN classification:

$$f = \sqrt{\sum_{i=0}^{n} \left(\frac{|p_i^{fittest} - p_i|}{MAX_i} \cdot w_i \right)^2} \tag{12}$$

Liked designs were also given a fitness boost $f' = \frac{f}{p}$ during classification based on the match percent discussed above. Designs classified as Liked with the highest percentage match are thus displayed higher up on the list, due to the inverted way of calculating fitness mentioned earlier.

A completely new fitness function was also added. Instead of the Euclidean distance, fitness was assigned based on the aforementioned match percent. Designs classified as Liked are given fitness $f = \frac{1}{p}$, whilst Disliked designs are assigned $f = p$. This means that designs with a high Dislike percentage match are less fit than designs with lower percentage matches. Each generation is sorted based fitness as before, and selection is performed until the set percentage of the population is classified as Liked.

Being able to Like and Dislike designs over time introduced a new level of interactivity, as the user is able to provide more detailed feedback by actively Disliking bad designs. This helps the algorithm to avoid non-optimal solutions.

At first, votes were collected in the form of "likes" only - liked designs would then be displayed further up on the list of previews. To do this, the chromosome class was extended to include a vote variable. In the first implementation, this was a boolean variable, representing liked/not liked. Later, the vote system was extended to include the ability to dislike a bad design. The helper variable was changed to accommodate for this addition. This variable may hold three values: liked, disliked or unrated, unrated being its default value. To allow the user to vote on designs, each cell of the HTML preview table was given a set of like/dislike-buttons, which when clicked, add the design to one of two lists, depending on the vote. Each new design of each new generation is thereafter evaluated and classified according to K-NN.

For each generation, tentative design solutions were classified using the following steps:

Algorithm 7 K-NN as fitness function

```
1: Combine lists of liked/disliked to single list $L$
2: if $L$.Size $\geq K$ then                          ▷ K is the number of neighbors to use
3:     for each Chromosome $d$ in population do
4:         for each Chromosome $V$ in $L$ do
5:             $V$.fitness $\leftarrow f$ (eq. 12)
6:         end for
7:         Sort $L$ based on fitness
8:         for i:=0 to $K$ do
9:             Count number of Liked, Disliked and Unrated
10:        end for
11:        if Liked $\geq$ Disliked AND Liked $\geq$ Unrated then
12:            Classify $d$ as Liked
13:        else if Disliked $\geq$ Liked AND Disliked $\geq$ Unrated then
14:            Classify $d$ as Disliked
15:        end if
16:    end for
17: end if
```

For the first time since the start of development, the algorithm was no longer solely dependent on user input, and could use previously collected vote data to determine the best designs. When the program is initialized, a helper variable keeping track of user input is set to false. If the user does not interact with the program, designs are always sorted based on their K-NN classification, and the individuals with the highest predicted match percent are selected for crossover. This way, the design process becomes automatic. To back this automation up, an additional termination condition was implemented. The condition triggers on the percentage of the population which is classified as Liked by the K-NN algorithm.

Computing the distance between each member of the population in relation to the collected vote data came at a significant performance penalty, especially as the number of collected votes increased. This was solved by splitting the population into equally sized partitions, and performing the classifications for each partition on a separate thread. This parallel implementation brought the performance back up to similar levels as before the classification implementation, which will be discussed in the results-section.

Algorithm 8 Parallel K-NN classification

```
1:  Combine lists of liked/disliked to single list $L$
2:  if $L$.Size $\geq K$ then                    ▷ K is the number of neighbors to use
3:      Divide population into equal partitions $s$, $s + e$
4:      for each Chromosome $d$ in $s$, $s + e$ do
5:          for each Chromosome $V$ in $L$ do
6:              $V$.fitness $\leftarrow f$ (eq. 12)
7:          end for
8:          Sort $L$ based on fitness
9:          for i:=0 to $K$ do
10:             Count number of Liked, Disliked and Unrated
11:         end for
12:         if Liked $\geq$ Disliked AND Liked $\geq$ Unrated then
13:             Classify $d$ as Liked
14:         else if Disliked $\geq$ Liked AND Disliked $\geq$ Unrated then
15:             Classify $d$ as Disliked
16:         end if
17:     end for
18: end if
```

The CSS generation was unchanged in this iteration. The rendering system was updated to display the voting buttons and the classification:

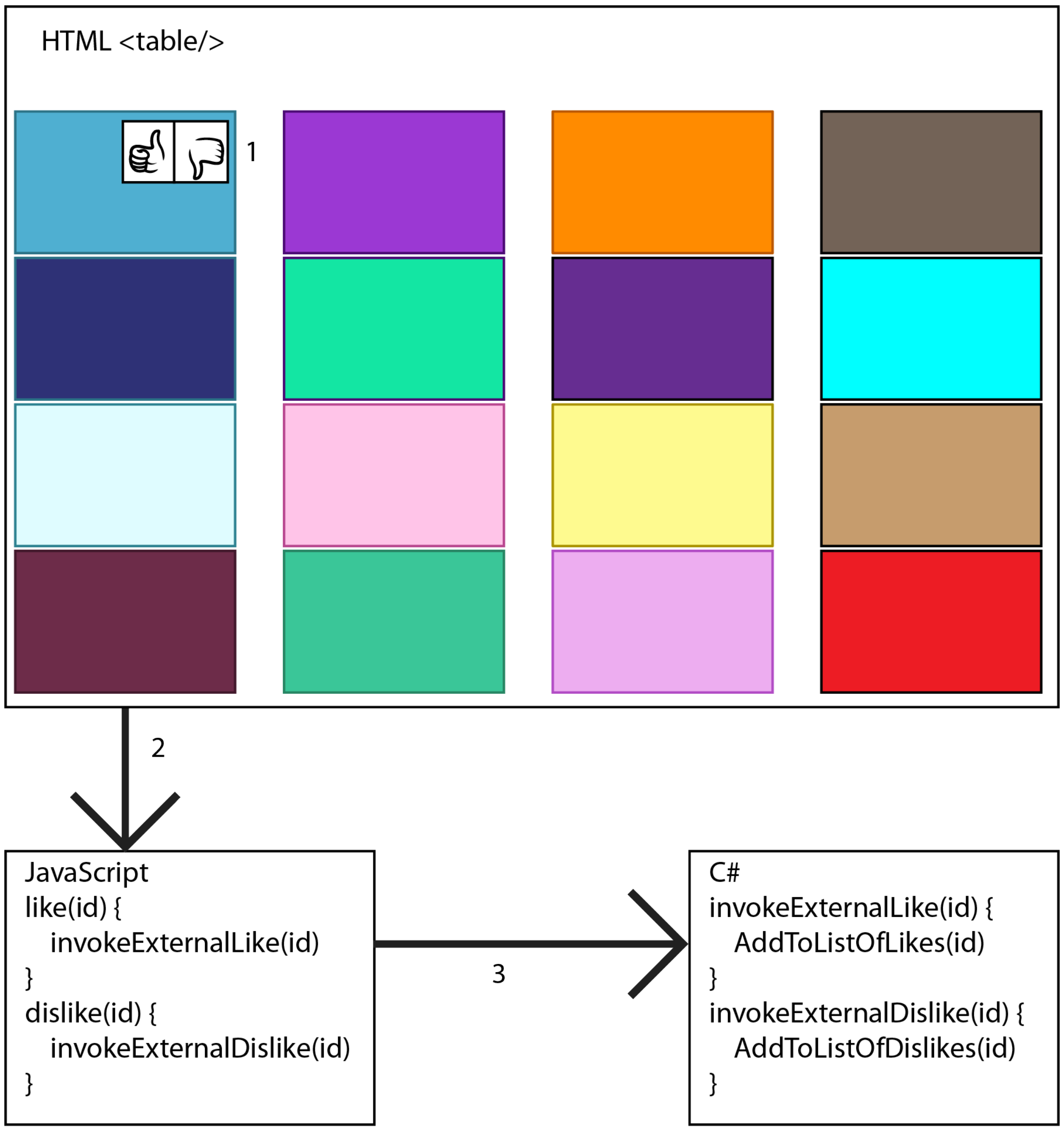


Figure 15: Voting process in application

In fig. 15, Like/Dislike buttons are displayed as the user hovers over a design (1). Each button triggers the design's like/dislike function in JavaScript (2), which is then passed through to the C# code (3), in the same way as the existing interactive selection process. The HTML code for the buttons are stored in a separate text file, and added to the preview page when it is generated. Liked/disliked designs are serialized and saved to two separate files when the program is closed using the *FileStream*-class found in the *System.IO*-name space of C#. Similarly, the files containing voting data are loaded and deserialized at program startup.

4.9 Summary of approach

Design of HTML-elements was encoded using the following values in the final implementation:

Property	Range	CSS Unit
Background color	$[0,255] \times 4$	RGBA
Hover color	$[0,255] \times 4$	RGBA
Shadow/text color color	$[0,255] \times 4$	RGBA
Border colors[]	$([0,255] \times 4)^4$	RGBA
Border color limiter	$[0,3]$	Helper
Border styles[]	$[0,9]^4$	Integer to string mapping
Border styles limiter	$[0,3]$	Helper
Border widths[]	$[0,15]^4$	Pixels
Border widths limiter	$[0,3]$	Helper
Border radii[]	$[0,255]^4$	Pixels
Border styles limiter	$[0,3]$	Helper
Shadow H pos	$[0,15]$	Pixels
Shadow V pos	$[0,15]$	Pixels
Shadow blur	$[0,15]$	Pixels
Font family	$[0,31]$	Integer to string mapping
Font size	$[8,31]$	Pixels
Paddings[]	$[0,31]^4$	Pixels
Paddings limiter	$[0,3]$	Helper
Use shade/tint as hover color	$[0,1]$	Mapping function
Transparent background	$[0,7]$	Mapping function
Gradient background	$[0,1]$	Helper
Secondary gradient color	$[0,255] \times 4$	RGBA

Table 11: Representation of design features in base chromosome

Most values in table 11 are used directly when generating the CSS-file, while other are passed to special mapping functions, similar to the representation discussed by Fung et al. and Oliver et al. in section 2.2. These include Border styles and Font family, which are represented as integer values in the chromosome corresponding to array indices as described in table 6. These arrays contain predefined CSS-strings for said properties.

Some values are represented as integers, but interpreted by the program as true/false. This was done to manipulate the probability of the condition being true, as a probability other than 50% was needed. An example is the transparent background variable, which works by overriding the element's existing background color. Overriding 50% of the population's background color was not considered acceptable, and the current implementation produces a transparent design with a $\frac{1}{8}$ chance. This is because the mapping function is set to return *true* if the value is 1, and *false* otherwise. Similarly, to get a larger number of square designs, the border radius variable was offset by subtracting 128. If this was not done, only $\frac{1}{255}$ of designs would be perfectly square, since $r = 0px$ corresponds to no roundness. Offsetting and returning $MAX(0, r)$ will produce square designs approximately 50% of the time. Another special case is the hover color. At first, the hover color was generated independently of the element's regular color. Basing the hover color on the color of the element was added as an alternative, which computes the shade of a "light" or tint of a "dark" background color and returns it as the hover color of the element if the condition is true using the following formula:

$$\overrightarrow{RGB}_{shade} = [R, G, B] \times 0.8 \tag{13}$$

$$\overrightarrow{RGB}_{tint} = [R + (0.3 \times (255 - R)), G + (0.3 \times (255 - G)), B + (0.3 \times (255 - B))] \tag{14}$$

Alpha values were not modified.

5 Results

5.1 Conceptual architecture

The development resulted in the following conceptual architecture:

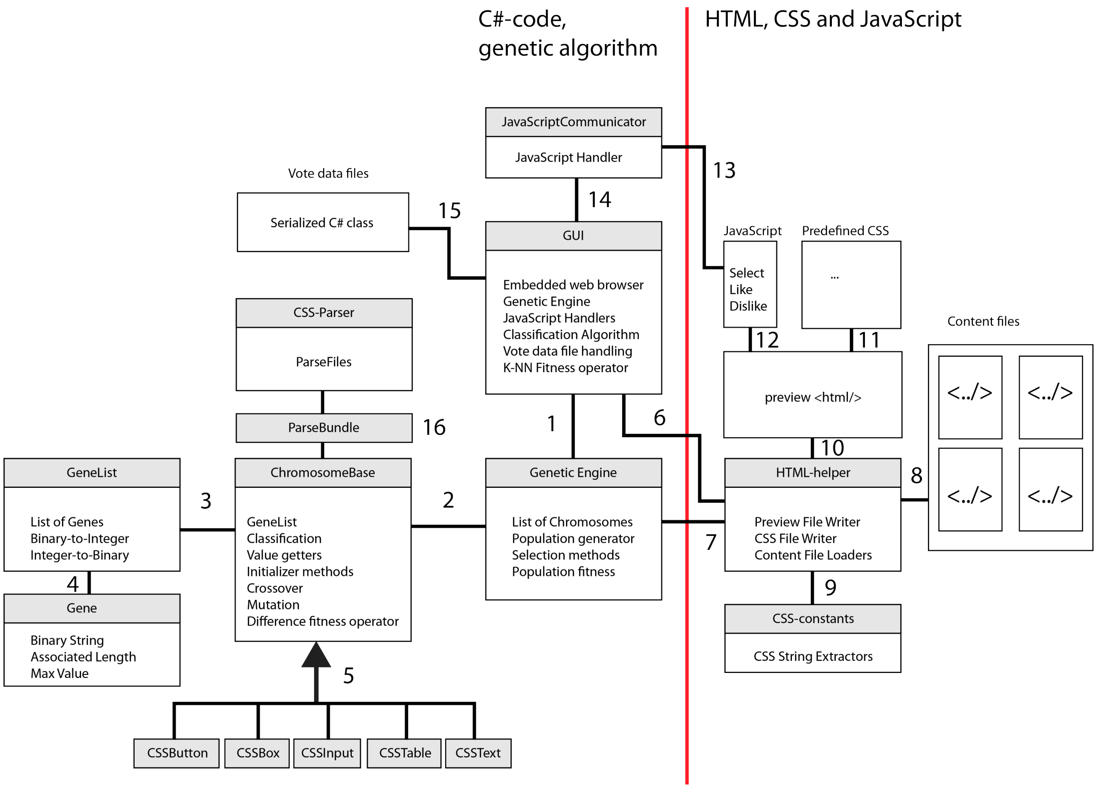


Figure 16: Simplified UML-diagram over main parts of solution

1. The GUI class is responsible for rendering and previewing the designs using the embedded browser
2. The Genetic Engine generates and manages the population using the methods discussed above
3. Designs are represented as integer values stored in the GeneList-class
4. The Gene data structure holds the actual values
5. As explained in section 4.4, additional features are added using inheritance
6. When a termination condition is triggered, the GUI instructs the HTML-helper to write the preview-file
7. The HTML-helper pulls the population of chromosomes from the Genetic Engine
8. The preview file is created by combining content files
9. Encoded values are mapped using the CSS-constants helper-class, which are written to a temporary CSS file
10. The finished preview file is loaded into the browser

11. The preview page is styled using an external static CSS-file
12. JavaScript is used to handle user interactions
13. The interaction handler class in C# relays the command from JavaScript to the GUI
14. User interaction in this case may be in the form of selecting an individual as fittest, voting or manually selecting individuals for crossover
15. In order for the system to remember learned feedback, votes are saved to disk, and read back on program startup
16. The initial population may be based on an existing design using the parser and the Parse-Bundle data structure

All previous versions of the program follow a similar structure in regards to the GUI, the Genetic Engine and the Chromosome class.

5.2 Benchmark information

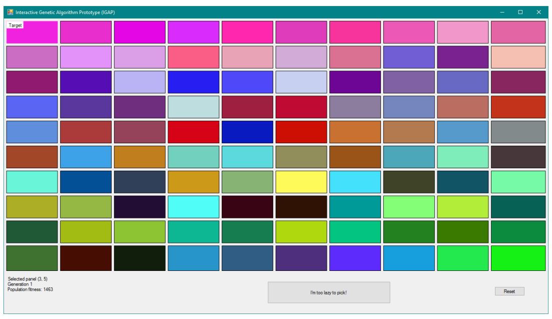
All benchmarks were performed on a computer with the following specifications:

- AMD FX-8350@4.34 GHz
- NVIDIA GTX 970
- Windows 10 Pro 64bit
- 24 GB RAM

Time was measured using the *Stopwatch*-class of the *System.Diagnostics*-namespace in C#.

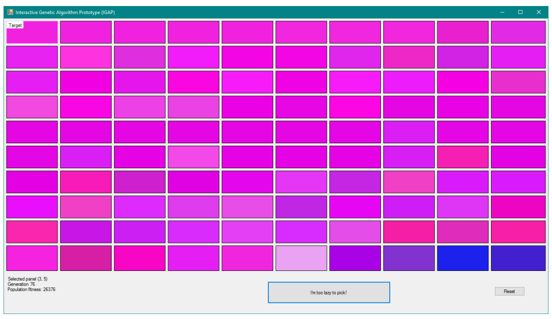
5.3 Single property interactive genetic algorithm

As mentioned, the very first implementation of the interactive genetic algorithm let the user select a single color:



```
Selected panel (3, 5)
Generation 1
Population fitness: 1463
```

(a) Initial population of the first version of program. Top left corner is fittest individual



```
Selected panel (3, 5)
Generation 76
Population fitness: 26376
```

(b) Convergent population of the first version of program

Figure 17: First iteration of program

Using the built-in *Random*-class of C# resulted in a fair random distribution, as seen in fig. 17a. Elitist selection and single-point crossover was used to produce the results observed. The user may click on a color, and over time, the whole population will converge towards that color, as seen in fig. 19b. This is a result of using the difference as a fitness function, as colors closer to the one selected by the user are more frequently selected. Using Windows Forms to render this simple implementation works well.

To benchmark the number of generations required for convergence, the program was run 10 times, selecting a random color each time. The algorithm was set to terminate when the population fitness sum reached 5000 or lower. This produced the following results:

Run	Elitist selection	Roulette wheel
1	56	8
2	47	8
3	56	9
4	57	9
5	55	9
6	74	11
7	63	12
8	68	11
9	72	10
10	52	9

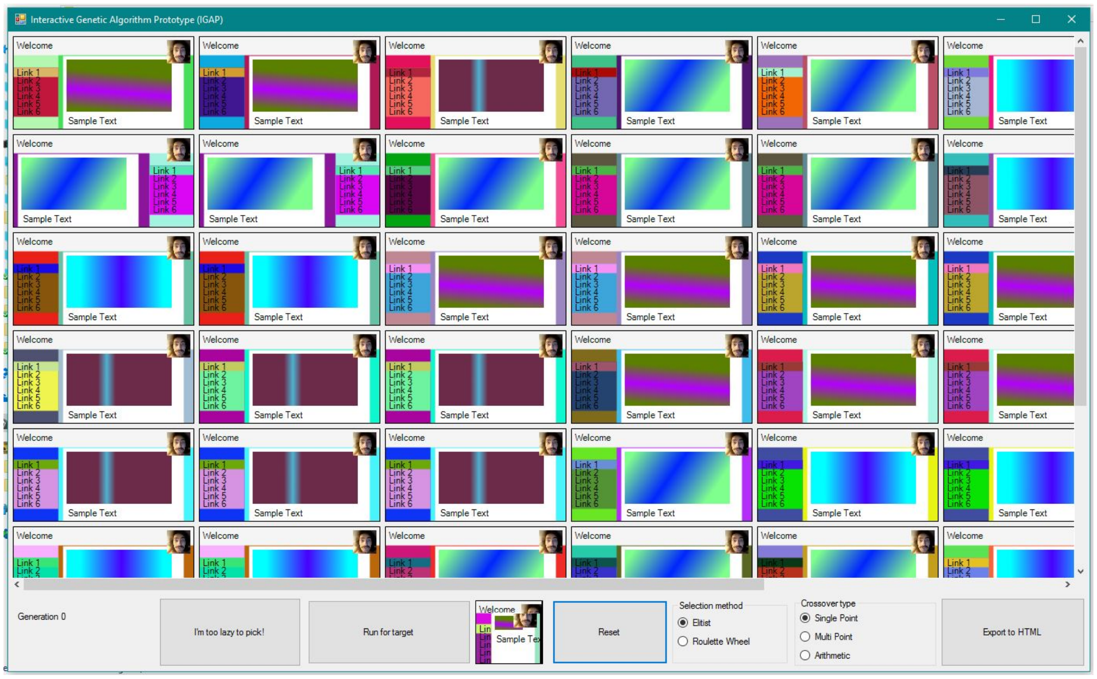
Table 12: Convergence results of first implementation

which corresponds to an average of 60 generations for elitist, and 9.6 generations for roulette wheel.

A potential use case for this simple implementation is to find a new color not thought about by the user. The user is presented with a variety of randomly generated colors, and the user keeps clicking on subjectively pleasing colors until the goal is reached. Alternatively, by clicking the automate-button, the program runs for a set number of iterations using based on the individual with index 0.

5.4 Extension of the working implementation

The addition of more properties made each individual more reminiscent of a simple web page:



Generation 0

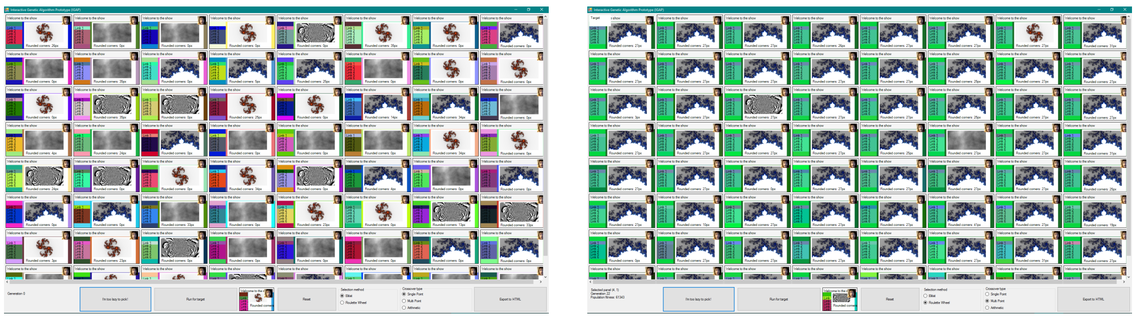
Selection method
(● Elitist) ○ Roulette Wheel

Crossover type
(● Single Point) ○ Multi Point ○ Arithmetic

Figure 18: Second version of program

Adding more properties to each individual means that the random number generator has to generate more values in a shorter time. As a result, one is able to observe duplicate designs in fig. 18.

Similar to the previous version, selecting a target and running selection and crossover using the difference fitness function eventually led to a converged population:



(a) Initial population of the second version of program

(b) Convergent population of the second version of program

Figure 19: Second iteration of program

This version also gave the user more control with the genetic algorithm's parameters using the controls in the lower right (fig. 18). From now on, due to the extensive possibilities of combinations of crossover types and selection schemes, benchmarks will only be run once per combination. Same as before, termination occurred at population fitness < 5000. This produced the following number of generations:

Crossover/selection	Elitist selection	Roulette wheel
Single point	1409	16
Multi-point	1263	17
Arithmetic	2735	17

Table 13: Convergence results of second implementation

The interactive implementation is identical to the last, in which the user may either keep clicking for generations, or let the program run for itself.

As seen in fig. 18, using Windows Forms graphics to render the preview has reached its limit in regards to displaying the new features added in this iteration.

The addition of simple web page elements lets the user explore variations of a set of web page layouts and colors simultaneously. This may help the user imagine their vision for the layout of the web page.

5.5 Using a design not present in the initial population as goal

As a means of benchmarking the mutation operator, the algorithm was set to converge towards features not present in the initial population, measuring the number of generations. Doing this led to the following results:

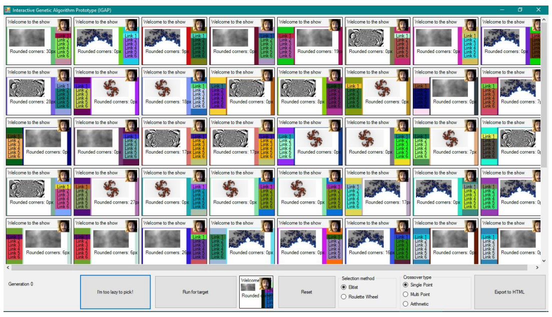
Crossover/selection	Elitist selection	Roulette wheel
Single point	6288	535
Multi-point	5768	216
Arithmetic	10450	444

Table 14: Convergence results of second implementation, unknown design

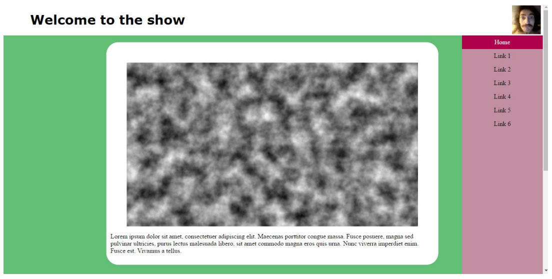
The variable performance seen in table 14 will be discussed further. Results regarding other aspect should not differ from the last, as no changes were made. Similarly, the existing rendering and preview system was kept unchanged.

5.6 Results of generating CSS from chromosomes and browser preview

As stated earlier, Windows Forms did not do justice to the new features added. Exporting the CSS-file and opening it in a browser gave a much better idea of how the final web page would appear:



(a) Render of designs using Windows Forms graphics



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus.

(b) Render of fittest design using browser

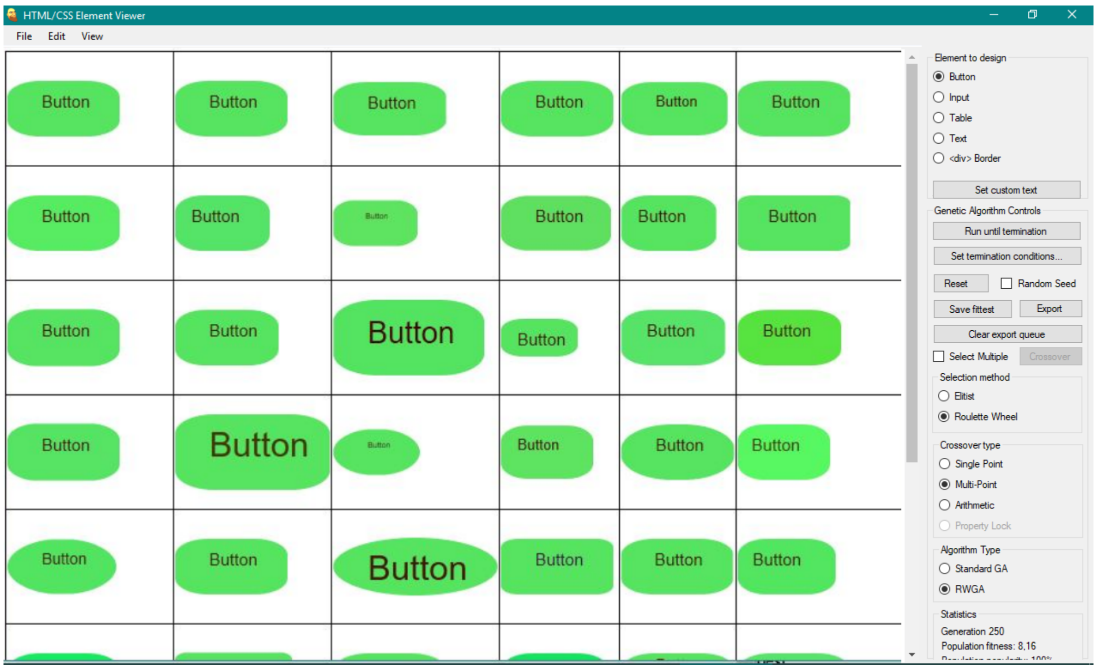
Figure 20: Windows Forms graphics and browser preview comparison

For instance, rounded corner were "visualized" using text only within the application in fig. 20a, but were able to be viewed in full effect using a browser (fig. 20b).

Also, as seen in fig. 20a, changing the initial seeding resulted in a better random distribution compared to fig. 18.

As this iteration was more oriented towards improving the rendering system, no changes were made to the genetic algorithm related parts of the project.

The next major part of this iteration was the implementation of a completely new rendering/preview system:



Element to design: [✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls

☐ Random Seed

☐ Select Multiple

Selection method: Elitist [✓ Roulette Wheel]

Crossover type: Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type: Standard GA [✓ RWGA]

Statistics
Generation 250
Population fitness: 8,16

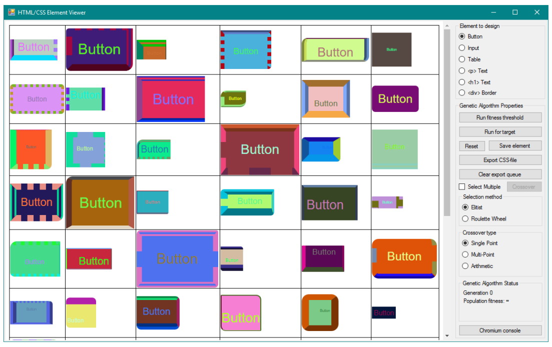
Figure 21: Render of different button designs inside the program using embedded browser

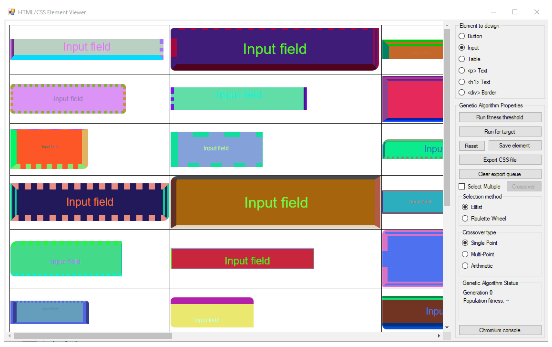
In the above figure, various button designs were created using a combination of randomly generated properties, including corner roundness, color, text size and border style. This particular population was the result of several generations of interactive selection, and has converged towards a green, border-less design. The background color was the quickest property to converge due to its weight, as discussed earlier. As one may see, text size and roundness are less converged due to lower weight. The experience of viewing each new generation became more unified, as the user may review both the visual design and the status of the genetic algorithm simultaneously.

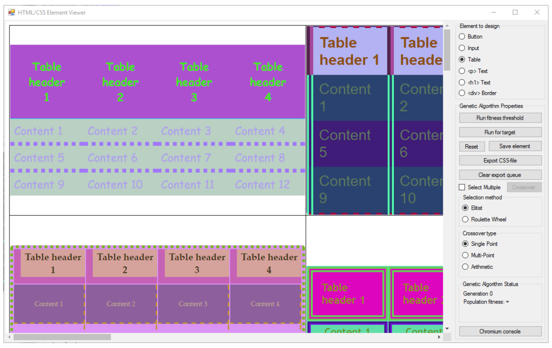
Allowing the user to export generated design to file meant that each design could be further refined manually before introduction to a web page. As an example, a user may want to try out various design for buttons on their web page. Starting with a randomly generated population, the user follows the same interactive process of selection until the best remain. The randomly generated initial population, along with the somewhat unpredictable nature of genetic algorithms, may serve as a source of inspiration in regards to overall web page design. When the population has converged towards a pleasing design, the user may then export the final product for further review and implementation. Exploring a palette of varying design is imagined to reduce time spent on brainstorming a completely new design from scratch, in addition to the time and effort spent implementing and testing each design by hand.

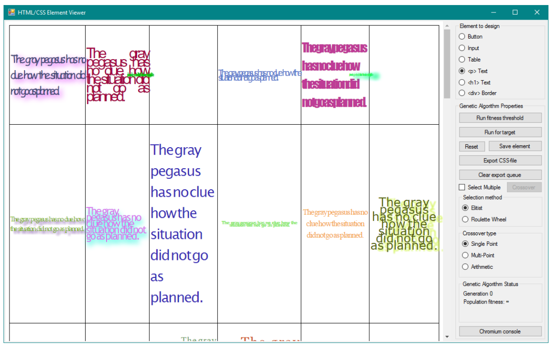
5.7 Additional elements

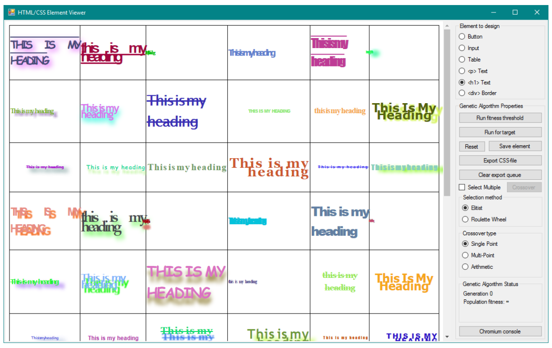
This update focused on the addition of more HTML elements.


(a) Button design preview


(b) Input field design preview


(c) Table design preview


(d) Text design preview


(e) Heading design preview

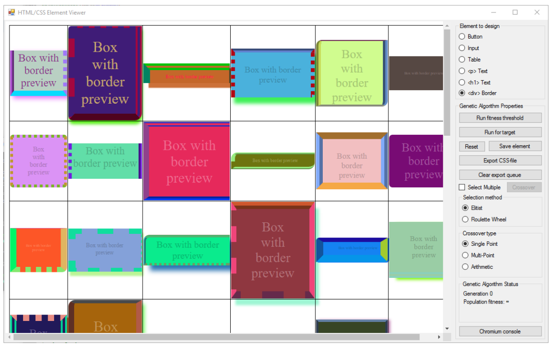
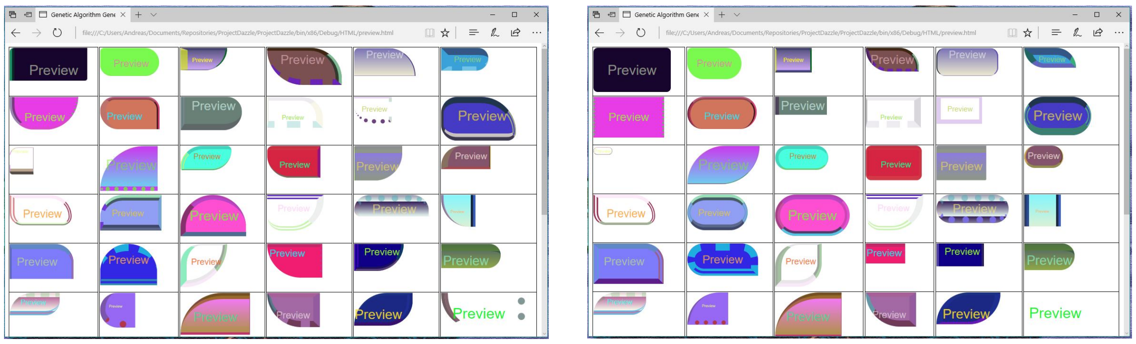

(f) Container design preview

Figure 22: Render of supported elements

As seen in the above figures, using the same representation for designing different elements works in the scope of this project.

5.8 Using helper variables as basic rules

The introduction of helper variables to limit the number of unique styles for each border resulted in more uniform shapes:



(a) Initial population without limiter variable (b) Initial population with limiter variable

Figure 23: Helper variable in effect

As seen from fig. 23b, using random numbers to limit the unique styles of each border does the job. As the helper variable was encoded just like any other gene, mutation was possible. A static seed was used to produce identical initial populations in fig. 23. In addition, the helper variable resulted in slightly faster convergence:

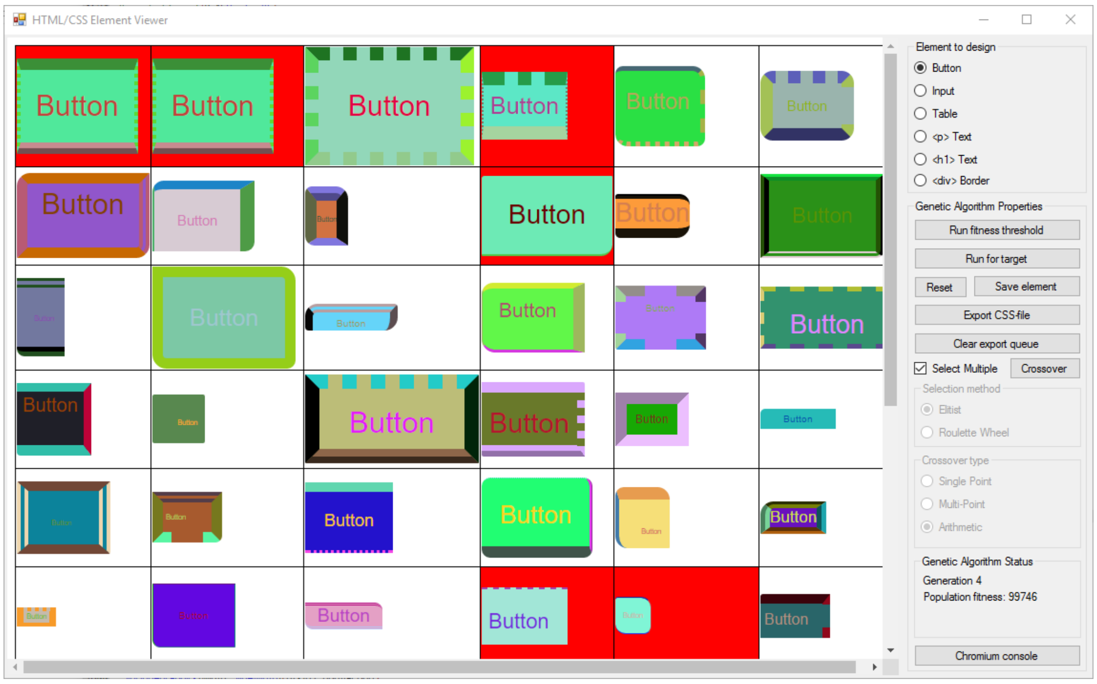
Crossover/selection	Elitist selection	Roulette wheel
Single point	1021	55
Multi-point	568	32
Arithmetic	1731	61

Table 15: Convergence results without helper variable

Crossover/selection	Elitist selection	Roulette wheel
Single point	897	51
Multi-point	391	26
Arithmetic	1301	59

Table 16: Convergence results with helper variable

Furthermore, the introduction of manual selection of individuals for crossover gave the user more control. Being able to more directly guide the algorithm in the desired way resulted in faster convergence compared to letting the algorithm run free:



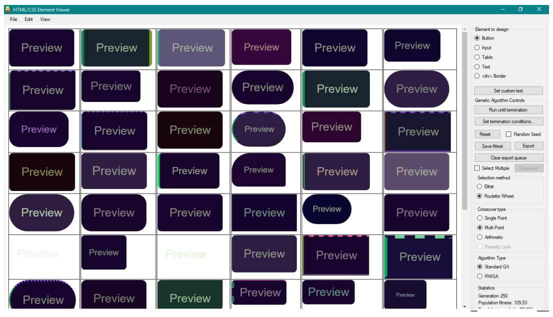
Generation 4
Population fitness: 99746

Figure 24: Manual crossover

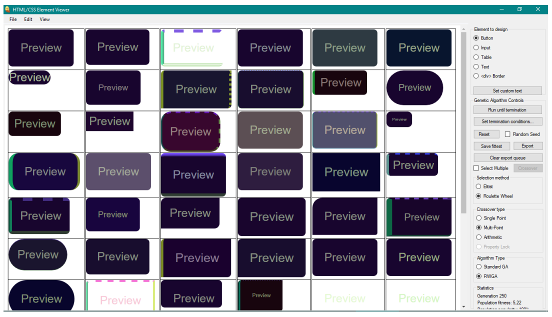
Using manual crossover allowed the user to explore recombination of designs in a noticeably different way than before. By selecting multiple similar designs, the desired genetic material became reinforced in the population, leading to faster convergence, as seen in fig. 24. Additionally, the inspiration process described above became more extensive, as more of the fine-tuning could be done within the program itself, without the need of external editors, due to the increased control. As an example, the user may want a design shaped like a rounded box. Up until this point, the user would have to rely on such a design occurring in the wild, either in the initial population, or by random mutation. Being able to manually combine features of designs, the user is able to select a square box and a circle, and get a mix of the distinct features.

5.9 Increasing population diversity using weights

The introduction of randomly generated weights led to marginally increased population diversity:

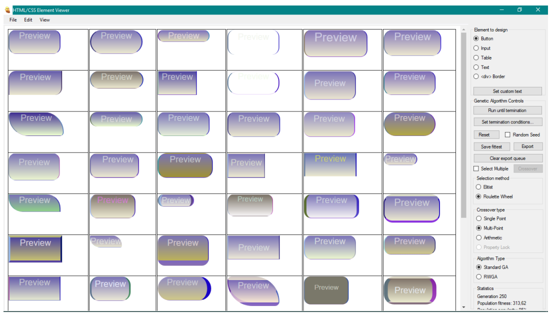


(a) Convergent population after 250 generations, equal weights



(b) Convergent population after 250 generations, random weights

Figure 25: Uniform weights and random weights compared, design A


(a) Convergent population after 250 generations, equal weights

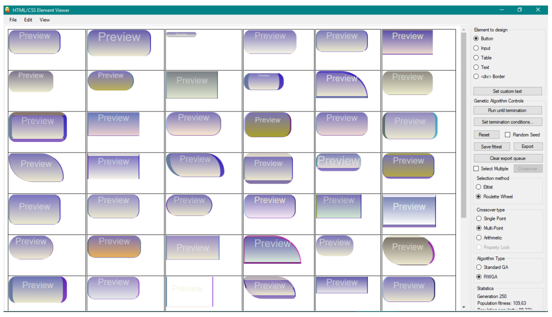
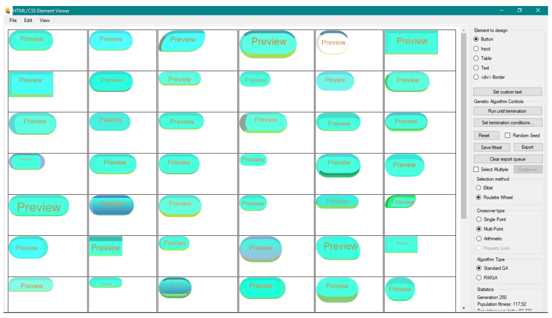

(b) Convergent population after 250 generations, random weights

Figure 26: Uniform weights and random weights compared, design B


(a) Convergent population after 250 generations, equal weights

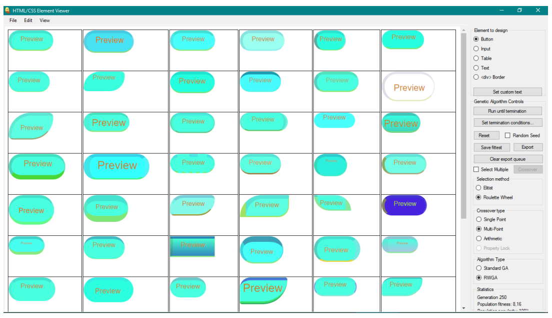

(b) Convergent population after 250 generations, random weights

Figure 27: Uniform weights and random weights compared, design C

As the investigation of the effect of random weights with regards to population diversity is not the main focus of this thesis, only a limited selection of results are considered. However, population diversity is still beneficial to avoid getting stuck in a local optimum. As previously mentioned, population diversity is also important when exploring design solutions. An example of this would be if the user changes their mind during the iterative process. Instead of generating a completely new population, the user may consciously select deviating designs to move the population towards a different part of the solution space.

5.10 Controlling the population

The use of a static seed when generating the initial population paved the way for more consitent testing, as the initial population was constant for every run. An example of this may be seen in fig. 23, as both populations are the same, even after the code was changed an recompiled to accomodate for the helper variables.

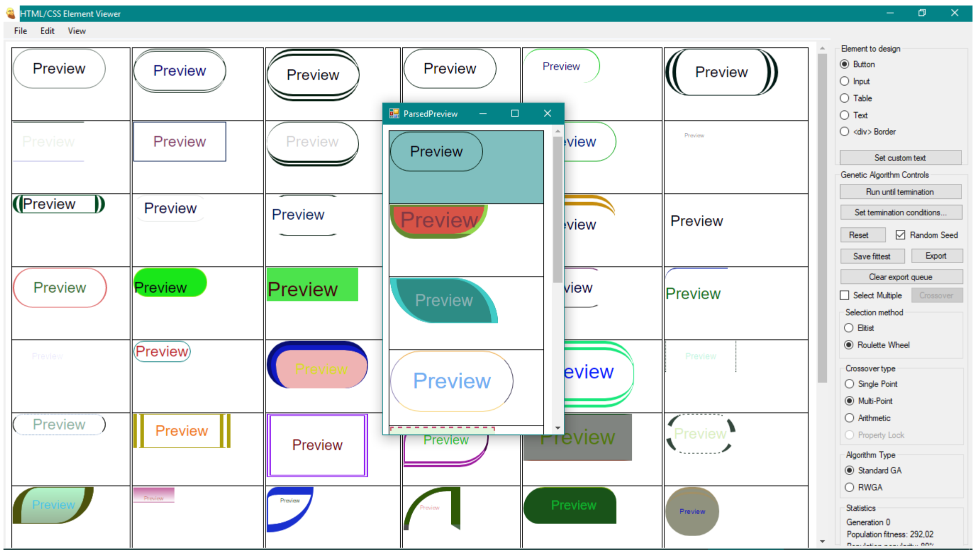
As one might expect, the introduction of readability rules led to slower convergence, as potentially good designs are penalized when failing the test:

Crossover/selection	Elitist selection	Roulette wheel
Single point	931	60
Multi-point	509	30
Arithmetic	1423	63

Table 17: Convergence results with readability test

These values are still better than the ones in table 15.

The other way of controlling the initial population was the parser feature. This proved to be quite useful, as it gave the user the opportunity to explore variations of known good designs. This way, the algorithm does the mundane task of generating the files, and the user may simply click and choose. In the below screenshots, the parsed designs are displayed in a separate window. The design selected as population base is marked with blue.



Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls

[✓ Random Seed]
Select Multiple

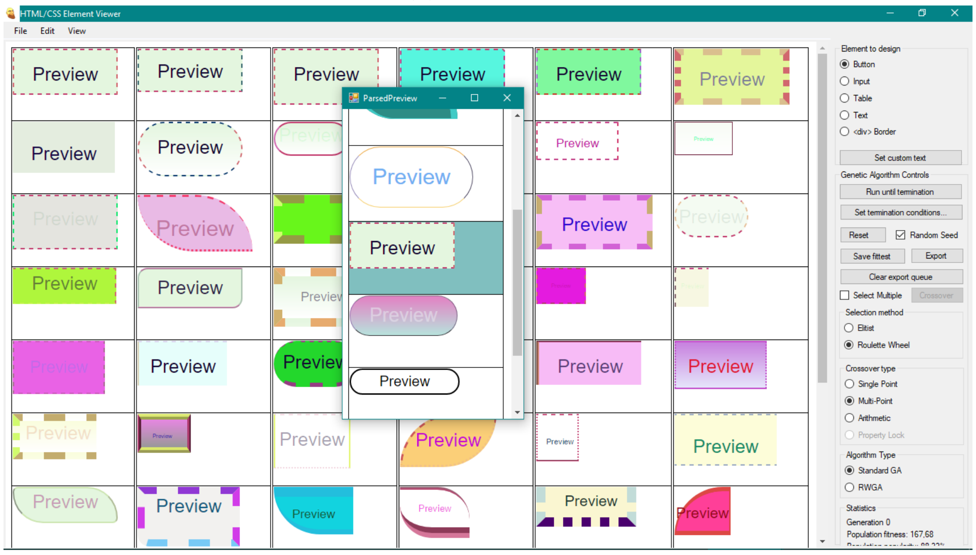
Selection method
Elitist [✓ Roulette Wheel]

Crossover type
Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type
[✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 292,02

(a) Example 1



Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls

[✓ Random Seed]
Select Multiple

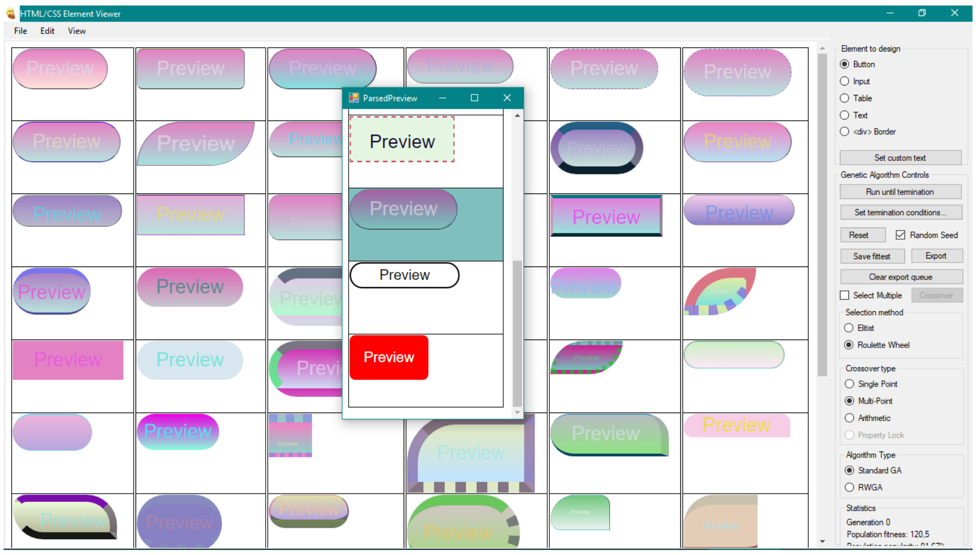
Selection method
Elitist [✓ Roulette Wheel]

Crossover type
Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type
[✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 167,68

(b) Example 2



Element to design: [✓ Button] Input Table Text <div> Border

Random Seed [✓]

Select Multiple [ ]

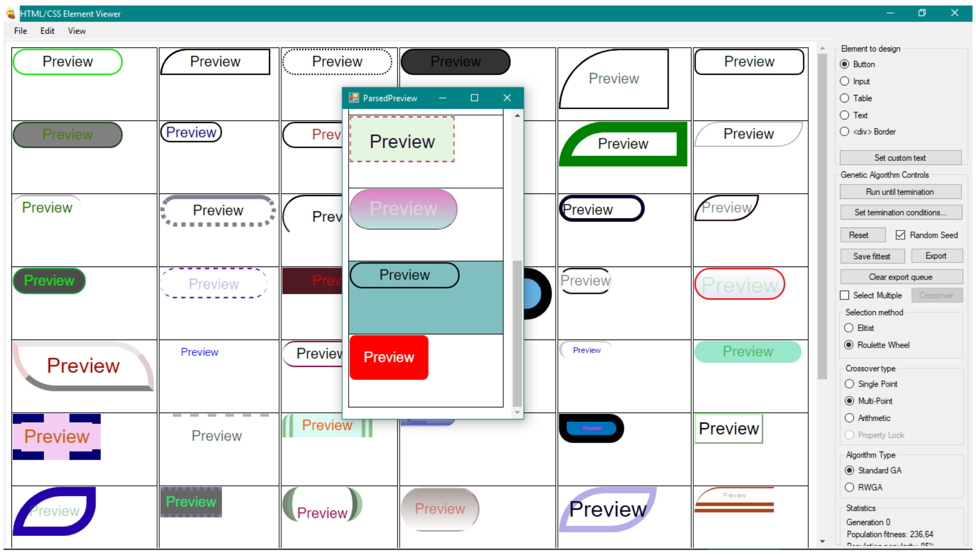
Selection method: Elitist [✓ Roulette Wheel]

Crossover type: Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type: [✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 120,5

(c) Example 3



Element to design: [✓ Button] Input Table Text <div> Border

Random Seed [✓]

Select Multiple [ ]

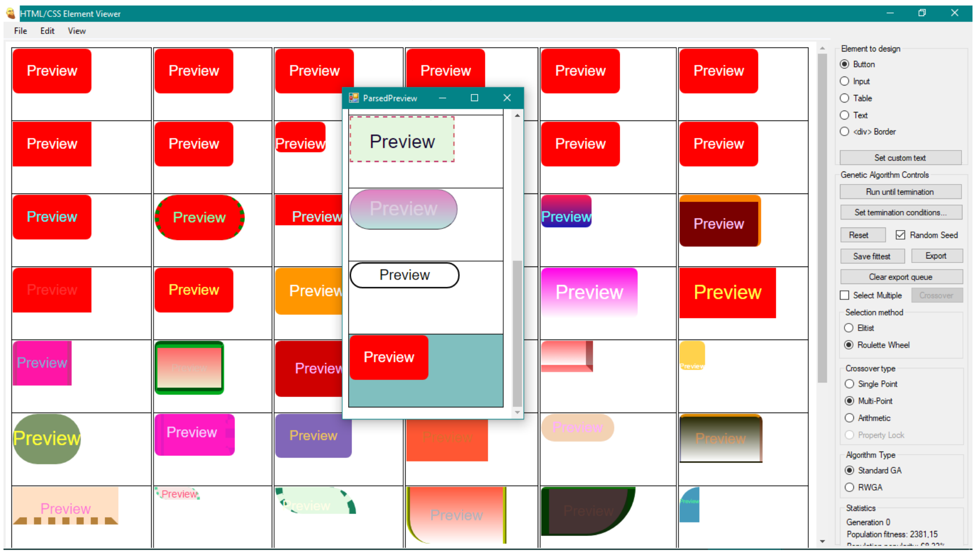
Selection method: Elitist [✓ Roulette Wheel]

Crossover type: Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type: [✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 236,64

(d) Example 4



Element to design: [✓ Button] Input Table Text <div> Border

Random Seed [✓]

Select Multiple [ ]

Selection method: Elitist [✓ Roulette Wheel]

Crossover type: Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type: [✓ Standard GA] RWGA

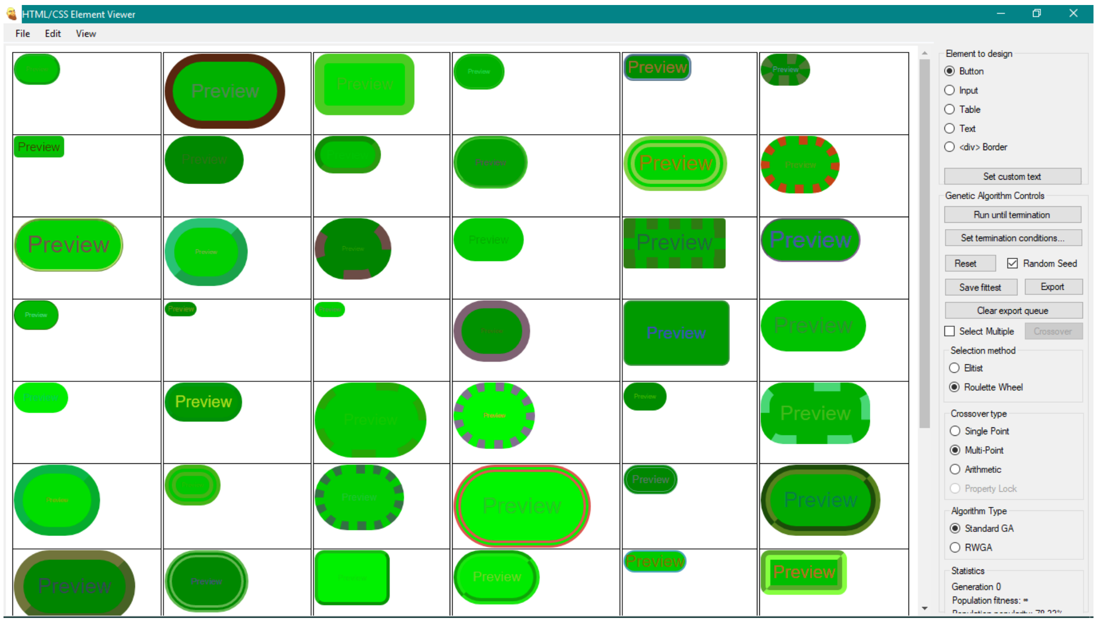
Statistics
Generation 0
Population fitness: 2381,15

(e) Example 5

Figure 28: Examples of initial populations based on parsed designs

The initial populations in the above figures are the direct result of using multi-point crossover to introduce the genetic material to the initial population.

The final way of controlling the initial population was by specifying the range for the random number generator. Depending on the range, the initial population is closer to convergence, resulting in shorter convergence times. As an example, the following initial population was produced when limiting the background color range to 128 - 255 for green only:



Element to design: [✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls

[✓ Random Seed] Select Multiple

Selection method: Elitist [✓ Roulette Wheel]

Crossover type: Single Point [✓ Multi-Point] Arithmetic Property Lock

Algorithm Type: [✓ Standard GA] RWGA

Statistics

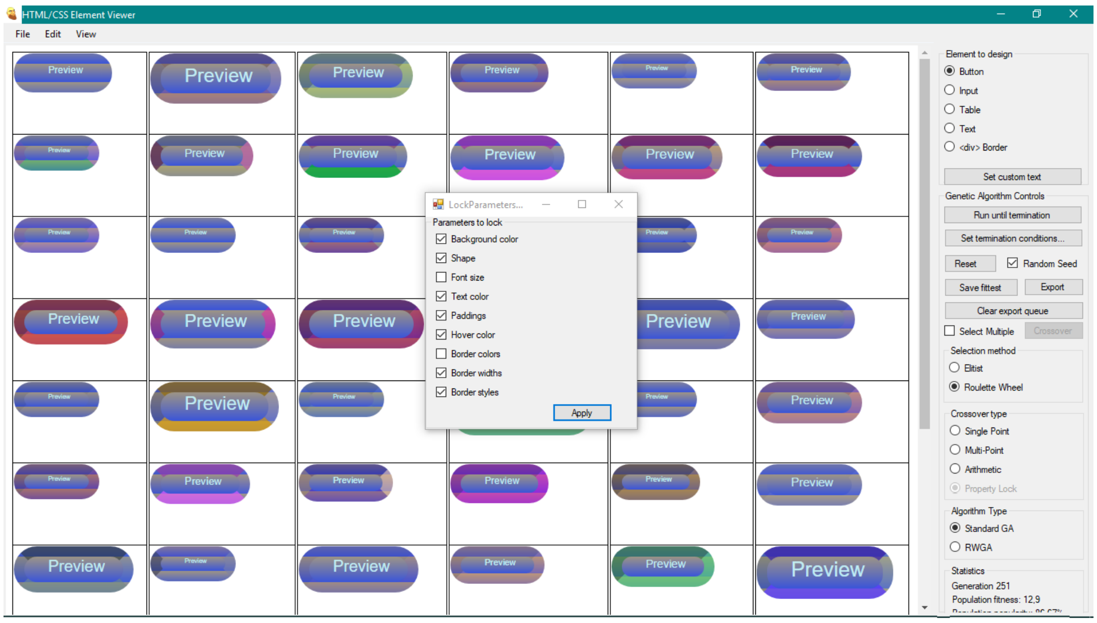
Generation 0

Population fitness: ∞

Figure 29: Initial population limited by initial conditions

As a practical example, a web developer may have an idea of which colors to use, which fonts and which shape. However, the user may want to try out various combinations within a set of parameters. Giving the user this control over the initial population enables more goal-oriented exploration of the solution space, without limiting it in the long run. Mutations still allow for divergence.

Using the implemented prop-lock crossover scheme allowed the user to manually control the population in finer detail during run-time:



Parameters to lock
- [✓] Background color
- [✓] Shape
- [ ] Font size
- [✓] Text color
- [✓] Paddings
- [✓] Hover color
- [ ] Border colors
- [✓] Border widths
- [✓] Border styles

Element to design: [✓ Button] Input / Table / Text / <div> Border

Genetic Algorithm Controls: [✓] Random Seed; [ ] Select Multiple

Selection method: Elitist / [✓ Roulette Wheel]

Crossover type: Single Point / Multi-Point / Arithmetic / Property Lock

Algorithm Type: [✓ Standard GA] / RWGA

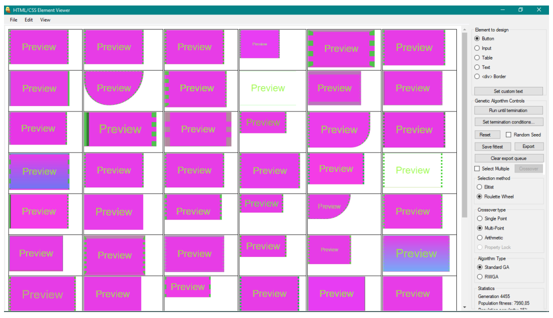
Statistics
Generation 251
Population fitness: 12,9

Figure 30: Converged population with locked/unlocked properties

In fig. 30, it is possible to see the effect of locking some parameters of the design, while leaving others free for the algorithm to explore. If the user finds a design with potential, it is possible to leave properties with undesired values open for further exploration, while gradually locking each good parameter during the iterative selection process. Doing this lets the user arrive at a much more controlled result, compared to complete random exploration.

5.11 The effect of normalizing weights

The effect of normalizing the difference between properties in the fitness function may be observed in the below figures:



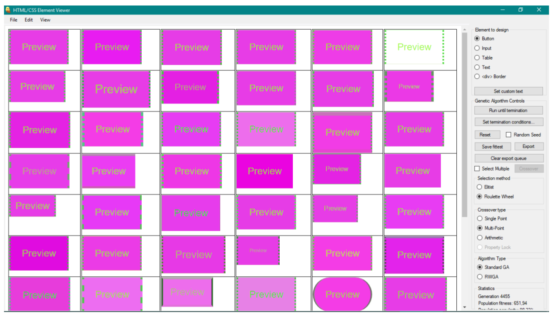
Selection method: Elitist / [✓ Roulette Wheel]

Crossover type: Single Point / [✓ Multi-Point] / Arithmetic / Property Lock

Algorithm Type: [✓ Standard GA] / RWGA

Statistics
Generation 4455
Population fitness: 7990,85

(a) Convergent population after 4455 generations, no normalization



Selection method: Elitist / [✓ Roulette Wheel]

Crossover type: Single Point / [✓ Multi-Point] / Arithmetic / Property Lock

Algorithm Type: [✓ Standard GA] / RWGA

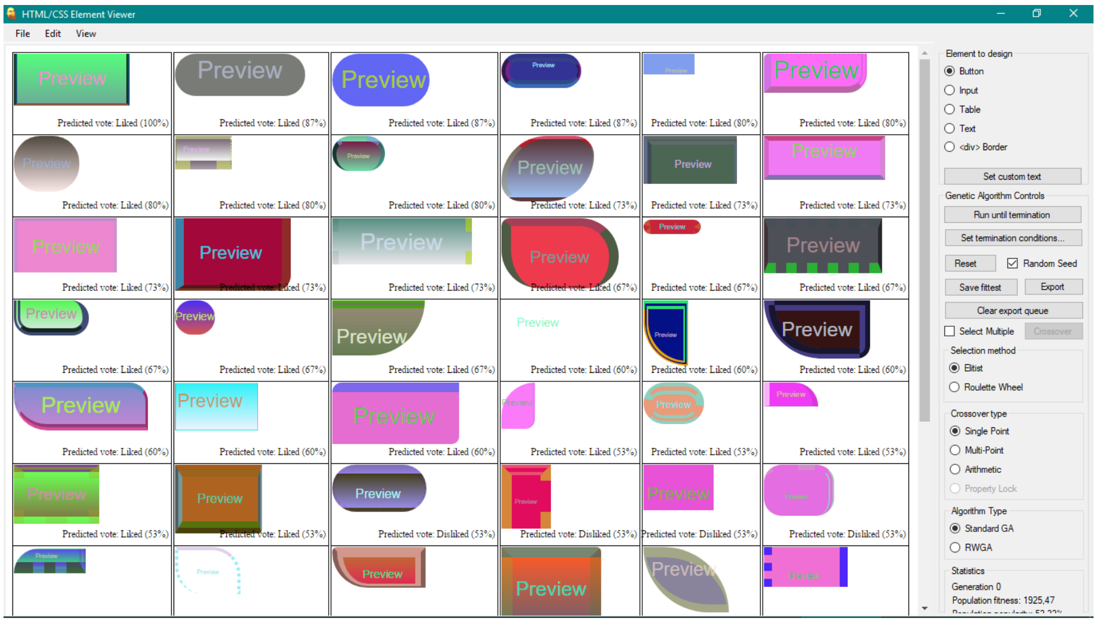
Statistics
Generation 4455
Population fitness: 651,94

(b) Convergent population after 4455 generations with normalization

Figure 31: Distribution of features in population with and without normalized properties

5.12 Introduction of learning and classification

Finally, the implementation of K-NN as a fitness function led to several interesting results. Firstly, by keeping track of designs marked as fittest by the user, and using collected data to classify each new population. Each element is presented with an associated classification match percent, as seen in fig. 32.



```
Predicted vote: Liked (100%)   Predicted vote: Liked (87%)   Predicted vote: Liked (87%)   Predicted vote: Liked (87%)   Predicted vote: Liked (80%)   Predicted vote: Liked (80%)
Predicted vote: Liked (80%)    Predicted vote: Liked (80%)   Predicted vote: Liked (80%)   Predicted vote: Liked (73%)   Predicted vote: Liked (73%)   Predicted vote: Liked (73%)
Predicted vote: Liked (73%)    Predicted vote: Liked (73%)   Predicted vote: Liked (73%)   Predicted vote: Liked (67%)   Predicted vote: Liked (67%)   Predicted vote: Liked (67%)
Predicted vote: Liked (67%)    Predicted vote: Liked (67%)   Predicted vote: Liked (67%)   Predicted vote: Liked (60%)   Predicted vote: Liked (60%)   Predicted vote: Liked (60%)
Predicted vote: Liked (60%)    Predicted vote: Liked (60%)   Predicted vote: Liked (60%)   Predicted vote: Liked (53%)   Predicted vote: Liked (53%)   Predicted vote: Liked (53%)
Predicted vote: Liked (53%)    Predicted vote: Liked (53%)   Predicted vote: Disliked (53%)   Predicted vote: Disliked (53%)   Predicted vote: Disliked (53%)   Predicted vote: Disliked (53%)
```

Element to design: [✓ Button] Input Table Text <div> Border
Set custom text

Genetic Algorithm Controls
Run until termination
Set termination conditions...
Reset [✓ Random Seed]
Save fittest Export
Clear export queue
Select Multiple Crossover

Selection method: [✓ Elitist] Roulette Wheel

Crossover type: [✓ Single Point] Multi-Point Arithmetic Property Lock

Algorithm Type: [✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 1925,47

Figure 32: Initial population with classification match

Here, elements with the highest classification match are placed higher up, while less liked and disliked appear further down, as discussed in section 4.8. As the user selects more designs, the list of selected designs grows, and the classification algorithm becomes dynamic over time. Combining this with the ability to manually Like/Dislike multiple designs of each generation creates the foundation of the K-NN based fitness function. Using the collected data in the fitness function creates dynamics, and is thus able to capture shifts in preference. Further improvements to this will be discussed later.

When the program is launched for the first time, no data is available:

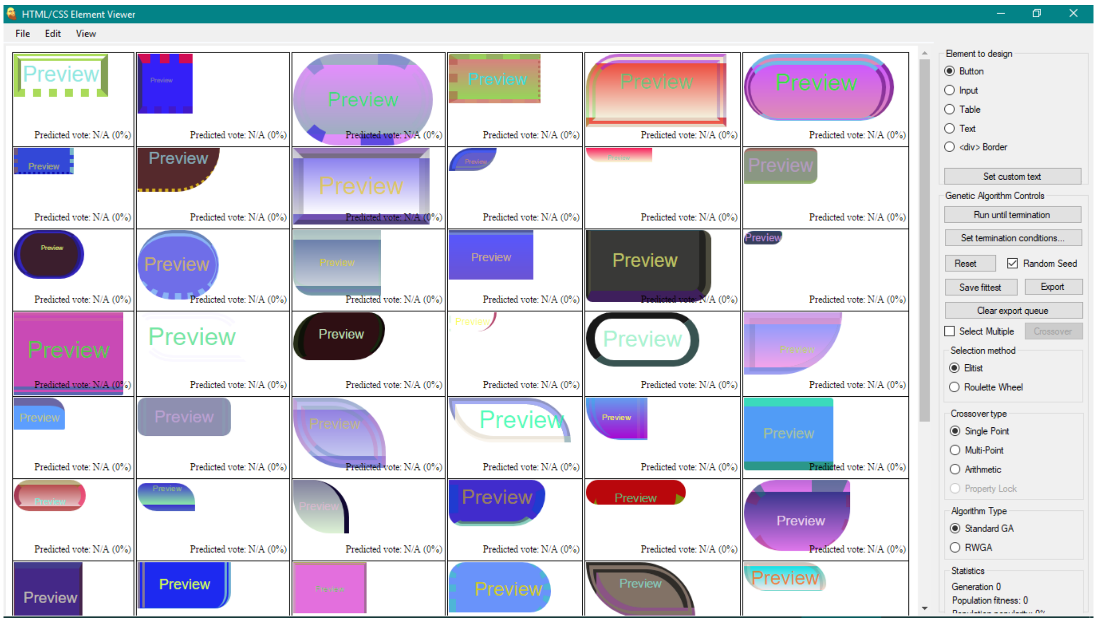


Figure 33: Initial population without classification data

The user may then start voting:

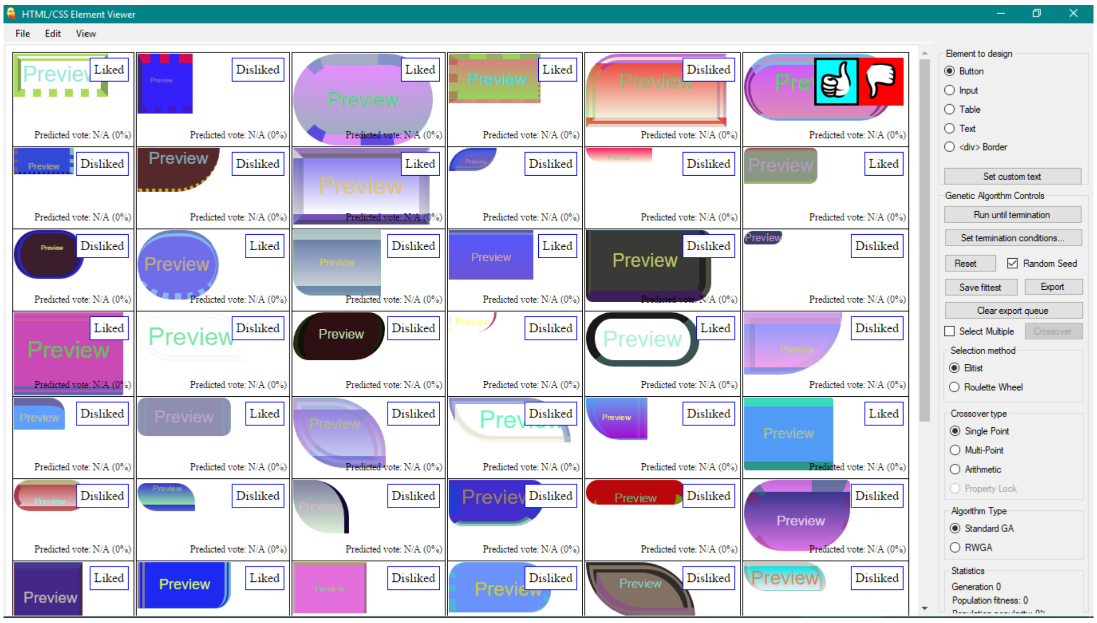
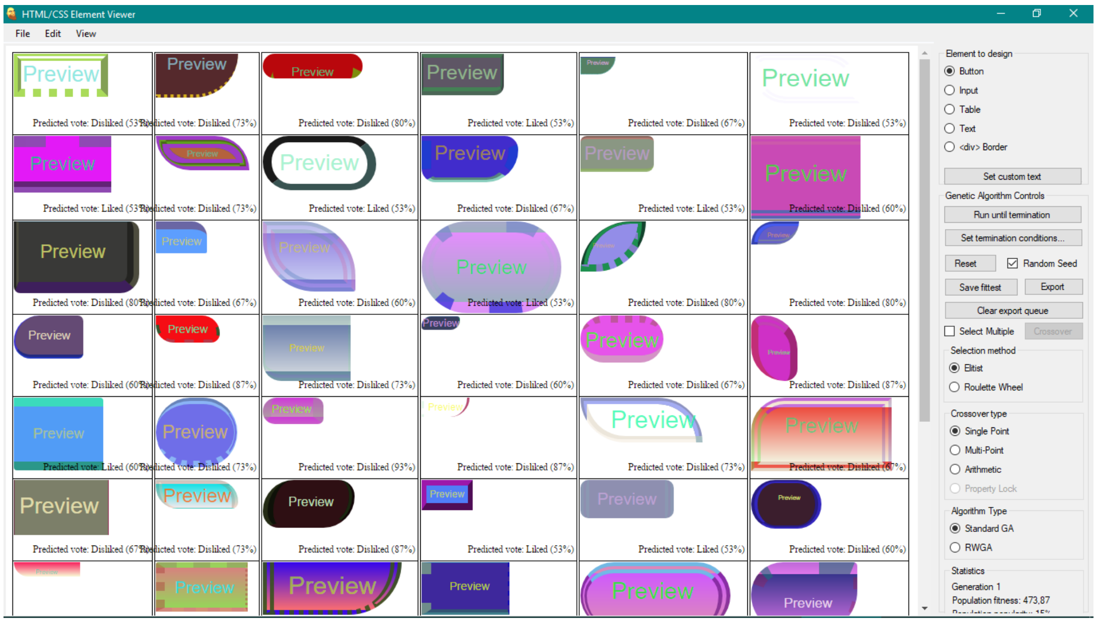


Figure 34: Initial population without classification data, voted

After sufficient vote data is collected (K = 15), classification is performed:



Predicted vote: Disliked (53%) Predicted vote: Disliked (73%) Predicted vote: Disliked (80%) Predicted vote: Liked (53%) Predicted vote: Disliked (67%) Predicted vote: Disliked (53%)
Predicted vote: Liked (53%) Predicted vote: Disliked (73%) Predicted vote: Liked (53%) Predicted vote: Disliked (67%) Predicted vote: Liked (53%) Predicted vote: Disliked (60%)
Predicted vote: Disliked (80%) Predicted vote: Disliked (67%) Predicted vote: Disliked (60%) Predicted vote: Liked (53%) Predicted vote: Disliked (80%) Predicted vote: Disliked (80%)
Predicted vote: Disliked (60%) Predicted vote: Disliked (87%) Predicted vote: Disliked (73%) Predicted vote: Disliked (60%) Predicted vote: Disliked (67%) Predicted vote: Disliked (87%)
Predicted vote: Liked (60%) Predicted vote: Disliked (73%) Predicted vote: Disliked (93%) Predicted vote: Disliked (87%) Predicted vote: Disliked (73%) Predicted vote: Disliked (87%)
Predicted vote: Disliked (67%) Predicted vote: Disliked (73%) Predicted vote: Disliked (87%) Predicted vote: Liked (53%) Predicted vote: Liked (53%) Predicted vote: Disliked (60%)

Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls
[✓ Random Seed]
Select Multiple

Selection method
[✓ Elitist] Roulette Wheel

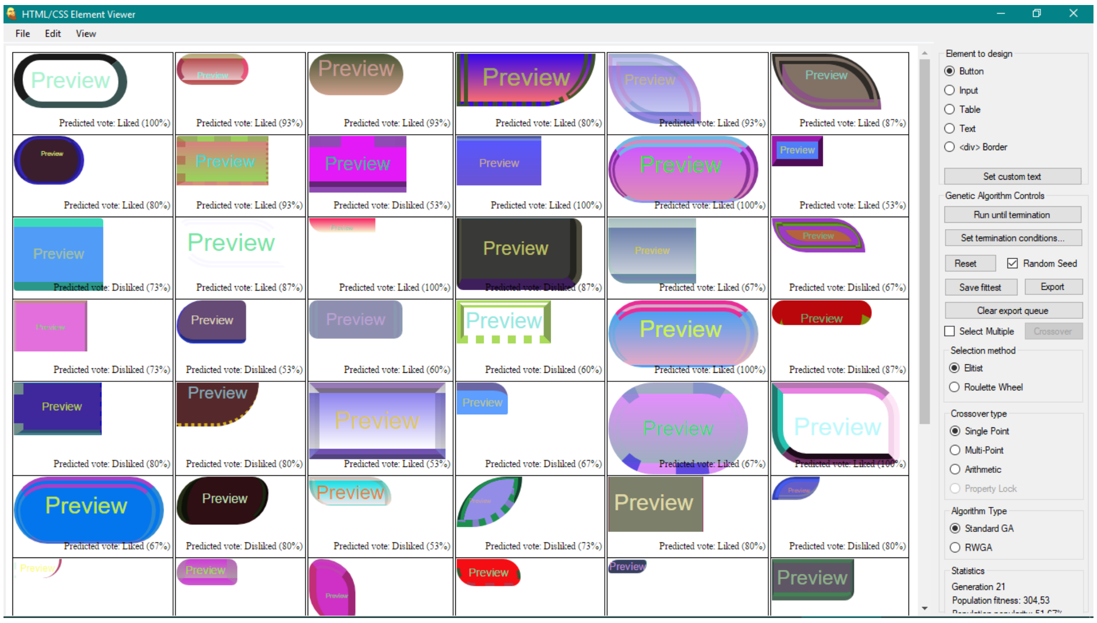
Crossover type
[✓ Single Point] Multi-Point Arithmetic Property Lock

Algorithm Type
[✓ Standard GA] RWGA

Statistics
Generation 1
Population fitness: 473,87

Figure 35: Population classification based on collected vote data, low accuracy

However, the results observed in fig. 35 may seem contradictory to the data provided in fig. 34. After collecting some more data, the results start to become more consistent:



Predicted vote: Liked (100%) Predicted vote: Liked (93%) Predicted vote: Liked (93%) Predicted vote: Liked (80%) Predicted vote: Liked (93%) Predicted vote: Liked (87%)
Predicted vote: Liked (80%) Predicted vote: Liked (93%) Predicted vote: Disliked (53%) Predicted vote: Liked (100%) Predicted vote: Liked (100%) Predicted vote: Liked (53%)
Predicted vote: Disliked (73%) Predicted vote: Liked (87%) Predicted vote: Liked (100%) Predicted vote: Disliked (87%) Predicted vote: Liked (67%) Predicted vote: Disliked (67%)
Predicted vote: Disliked (73%) Predicted vote: Disliked (53%) Predicted vote: Liked (60%) Predicted vote: Disliked (60%) Predicted vote: Liked (100%) Predicted vote: Disliked (87%)
Predicted vote: Disliked (80%) Predicted vote: Disliked (80%) Predicted vote: Liked (53%) Predicted vote: Disliked (67%) Predicted vote: Liked (67%) Predicted vote: Liked (100%)
Predicted vote: Liked (67%) Predicted vote: Disliked (80%) Predicted vote: Disliked (53%) Predicted vote: Disliked (73%) Predicted vote: Liked (80%) Predicted vote: Disliked (80%)

Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls
[✓ Random Seed]
Select Multiple

Selection method
[✓ Elitist] Roulette Wheel

Crossover type
[✓ Single Point] Multi-Point Arithmetic Property Lock

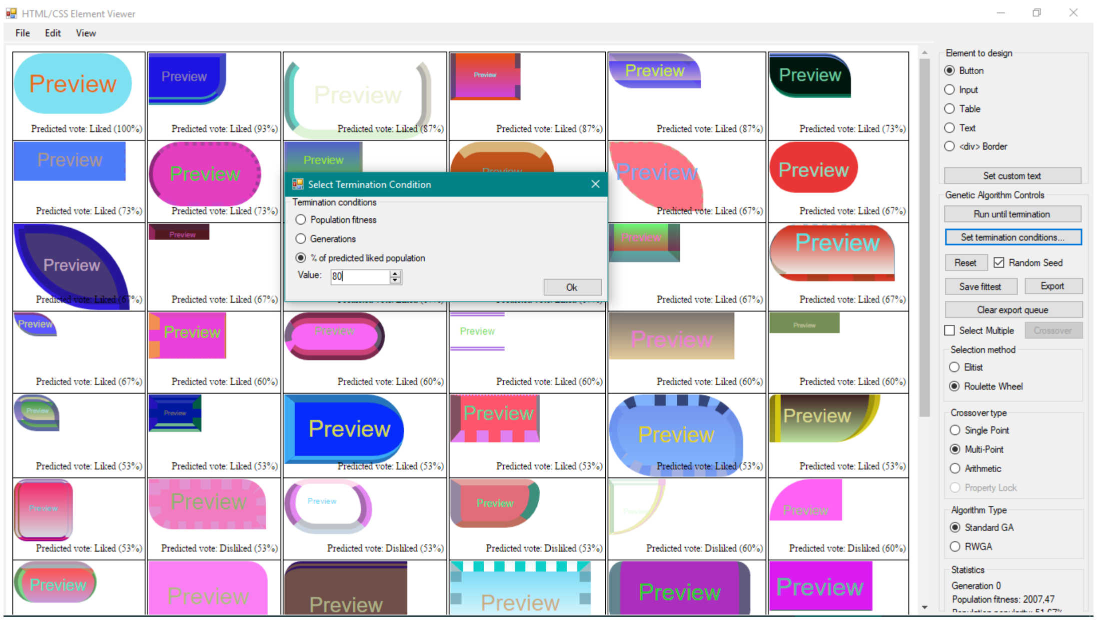
Algorithm Type
[✓ Standard GA] RWGA

Statistics
Generation 21
Population fitness: 304,53

Figure 36: Population classification based on collected vote data, higher accuracy

This shows that the new fitness function described in equation 12 works in classification, even with normalization and weights.

Using the last implemented termination condition lets the algorithm explore on its own using collected data only:


HTML/CSS Element Viewer

Select Termination Condition

Termination conditions

( ) Population fitness

( ) Generations

(●) % of predicted liked population

Value: 80

Element to design: Button; Selection method: Roulette Wheel; Crossover type: Multi-Point; Algorithm Type: Standard GA

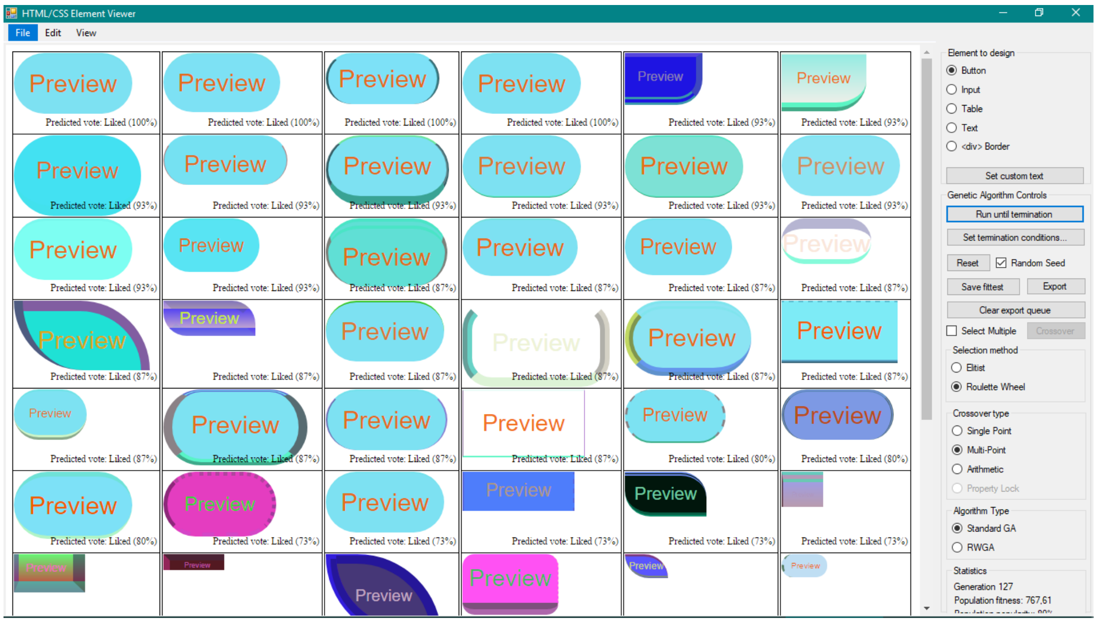
Statistics

Generation 0

Population fitness: 2007,47

Figure 37: Termination condition based on population classification

Here, the algorithm is set to terminate when at least 80% of the population is classified as Liked:


HTML/CSS Element Viewer

Element to design: Button; Random Seed ✓; Selection method: Roulette Wheel; Crossover type: Multi-Point; Algorithm Type: Standard GA

Statistics

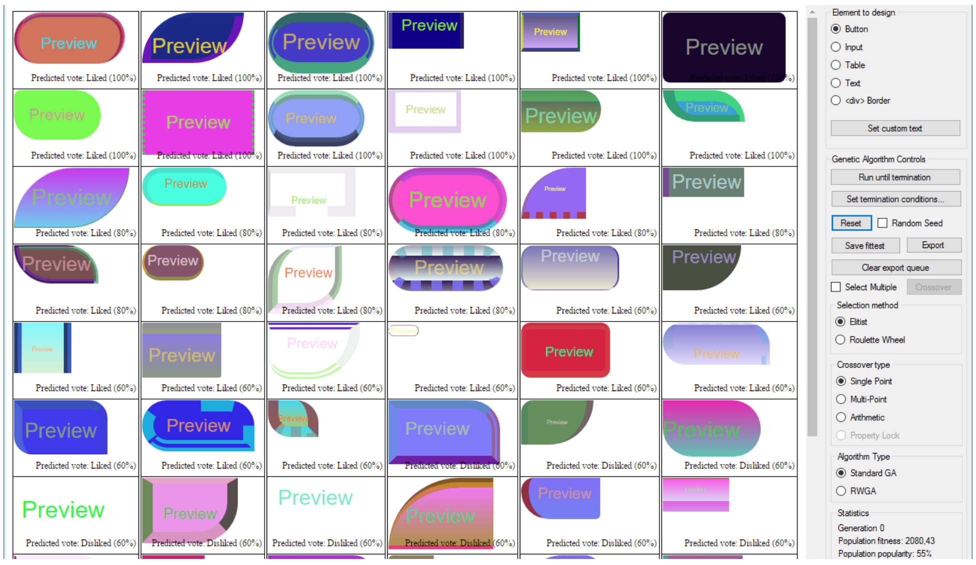
Generation 127

Population fitness: 767,61

Figure 38: Convergent population classified as Liked

In fig. 38 one may observe the result of letting the algorithm run without human interaction.

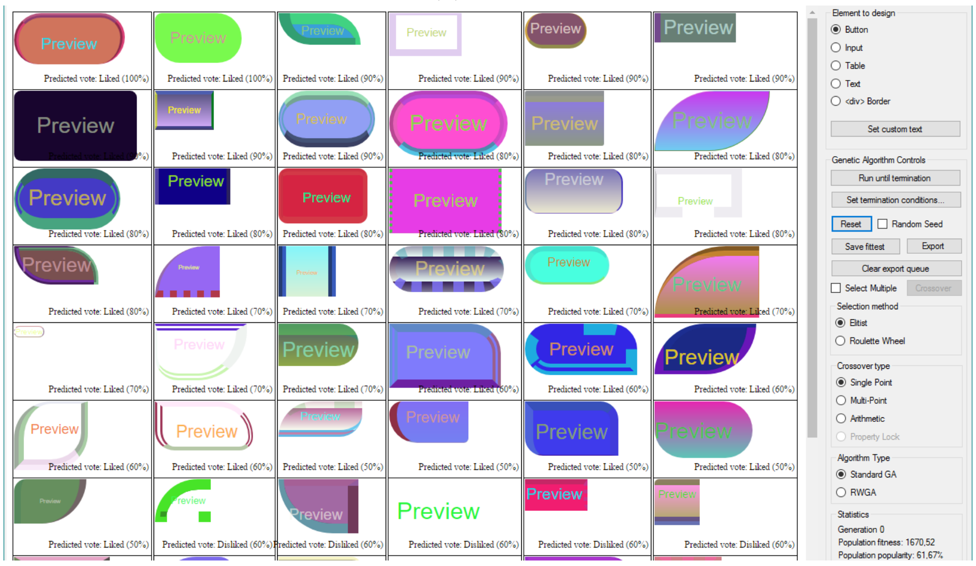
The results in the above experiments were produces by setting K = 15 in the classification algorithm. By changing this value, different results were observed:



Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%)
Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (100%)
Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%)
Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%)
Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%)
Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%)
Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%)

Element to design: [✓ Button] Input Table Text <div> Border
Random Seed
Select Multiple
Selection method: [✓ Elitist] Roulette Wheel
Crossover type: [✓ Single Point] Multi-Point Arithmetic Property Lock
Algorithm Type: [✓ Standard GA] RWGA
Statistics
Generation 0
Population fitness: 2080.43
Population popularity: 55%

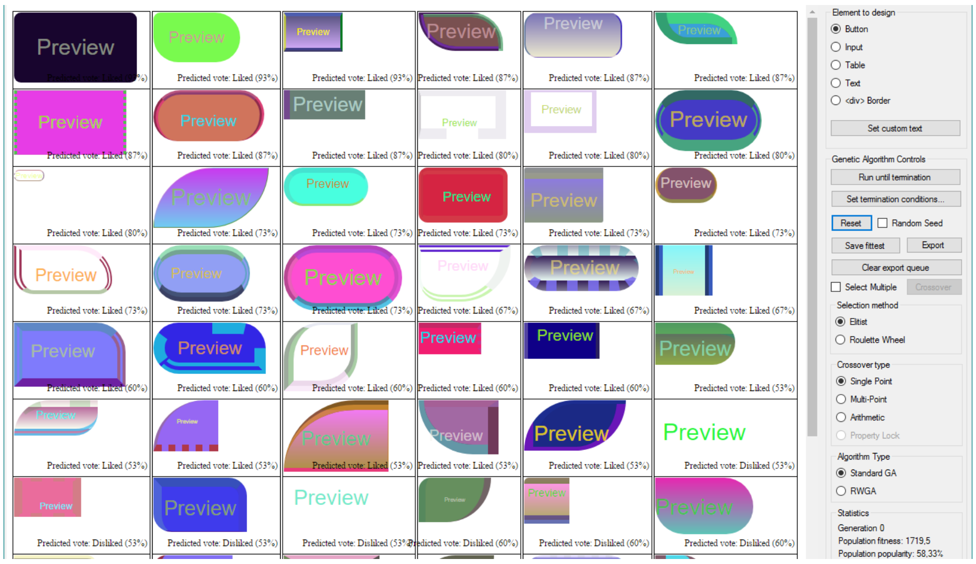
(a) K = 5



Predicted vote: Liked (100%) | Predicted vote: Liked (100%) | Predicted vote: Liked (90%) | Predicted vote: Liked (90%) | Predicted vote: Liked (90%) | Predicted vote: Liked (90%)
Predicted vote: Liked (90%) | Predicted vote: Liked (90%) | Predicted vote: Liked (90%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%)
Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%) | Predicted vote: Liked (80%)
Predicted vote: Liked (80%) | Predicted vote: Liked (70%) | Predicted vote: Liked (70%) | Predicted vote: Liked (70%) | Predicted vote: Liked (70%) | Predicted vote: Liked (70%)
Predicted vote: Liked (70%) | Predicted vote: Liked (70%) | Predicted vote: Liked (70%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (60%)
Predicted vote: Liked (60%) | Predicted vote: Liked (60%) | Predicted vote: Liked (50%) | Predicted vote: Liked (50%) | Predicted vote: Liked (50%) | Predicted vote: Liked (50%)
Predicted vote: Liked (50%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%) | Predicted vote: Disliked (60%)

Element to design: [✓ Button] Input Table Text <div> Border
Random Seed
Select Multiple
Selection method: [✓ Elitist] Roulette Wheel
Crossover type: [✓ Single Point] Multi-Point Arithmetic Property Lock
Algorithm Type: [✓ Standard GA] RWGA
Statistics
Generation 0
Population fitness: 1670.52
Population popularity: 61.67%

(b) K = 10



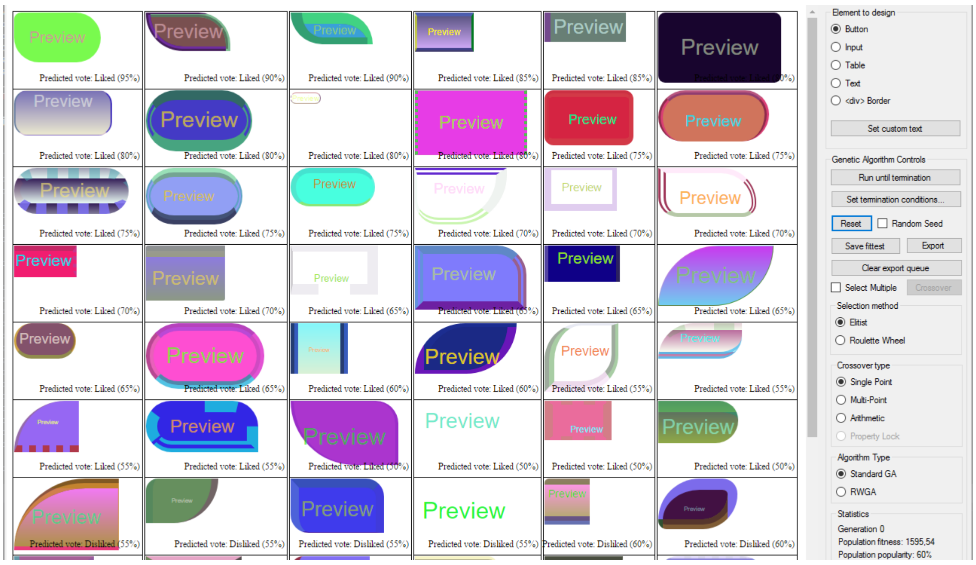
```
%)                          Predicted vote: Liked (93%)    Predicted vote: Liked (93%)    Predicted vote: Liked (87%)    Predicted vote: Liked (87%)    Predicted vote: Liked (87%)
Predicted vote: Liked (87%) Predicted vote: Liked (87%)    Predicted vote: Liked (87%)    Predicted vote: Liked (80%)    Predicted vote: Liked (80%)    Predicted vote: Liked (80%)
Predicted vote: Liked (80%) Predicted vote: Liked (73%)    Predicted vote: Liked (73%)    Predicted vote: Liked (73%)    Predicted vote: Liked (73%)    Predicted vote: Liked (73%)
Predicted vote: Liked (73%) Predicted vote: Liked (73%)    Predicted vote: Liked (73%)    Predicted vote: Liked (67%)    Predicted vote: Liked (67%)    Predicted vote: Liked (67%)
Predicted vote: Liked (60%) Predicted vote: Liked (60%)    Predicted vote: Liked (60%)    Predicted vote: Liked (60%)    Predicted vote: Liked (60%)    Predicted vote: Liked (53%)
Predicted vote: Liked (53%) Predicted vote: Liked (53%)    Predicted vote: Liked (53%)    Predicted vote: Liked (53%)    Predicted vote: Liked (53%)    Predicted vote: Disliked (53%)
Predicted vote: Disliked (53%) Predicted vote: Disliked (53%) Predicted vote: Disliked (53%) Predicted vote: Disliked (60%) Predicted vote: Disliked (60%) Predicted vote: Disliked (60%)
```

Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls
Select Multiple: unchecked
Random Seed: unchecked
Selection method: [✓ Elitist] Roulette Wheel
Crossover type: [✓ Single Point] Multi-Point Arithmetic Property Lock
Algorithm Type: [✓ Standard GA] RWGA

Statistics
Generation 0
Population fitness: 1719,5
Population popularity: 58,33%

(c) K = 15 (default)



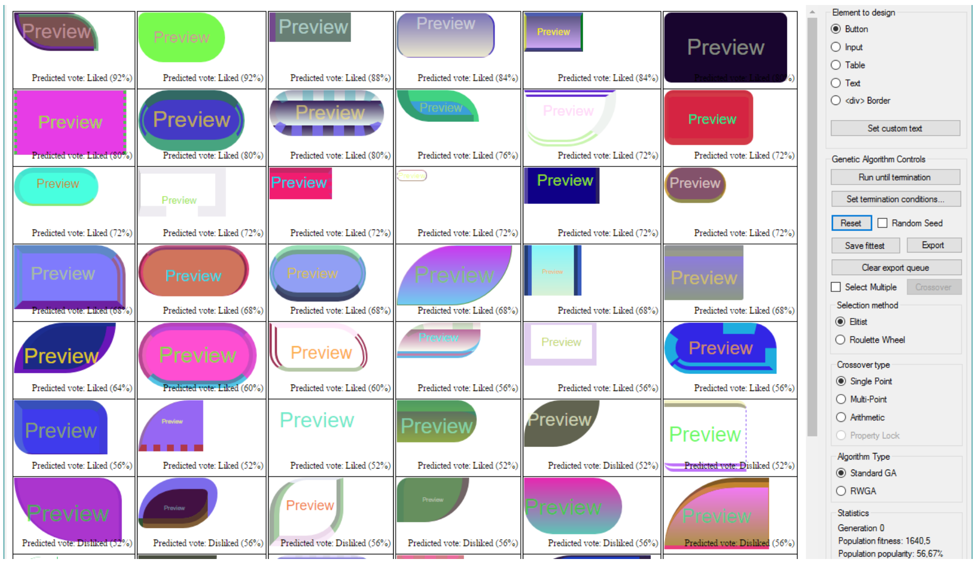
```
Predicted vote: Liked (95%) Predicted vote: Liked (90%)    Predicted vote: Liked (90%)    Predicted vote: Liked (85%)    Predicted vote: Liked (85%)    0%)
Predicted vote: Liked (80%) Predicted vote: Liked (80%)    Predicted vote: Liked (80%)    Predicted vote: Liked (80%)    Predicted vote: Liked (75%)    Predicted vote: Liked (75%)
Predicted vote: Liked (75%) Predicted vote: Liked (75%)    Predicted vote: Liked (75%)    Predicted vote: Liked (70%)    Predicted vote: Liked (70%)    Predicted vote: Liked (70%)
Predicted vote: Liked (70%) Predicted vote: Liked (70%)    Predicted vote: Liked (65%)    Predicted vote: Liked (65%)    Predicted vote: Liked (65%)    Predicted vote: Liked (65%)
Predicted vote: Liked (65%) Predicted vote: Liked (65%)    Predicted vote: Liked (60%)    Predicted vote: Liked (60%)    Predicted vote: Liked (55%)    Predicted vote: Liked (55%)
Predicted vote: Liked (55%) Predicted vote: Liked (55%)    Predicted vote: Liked (50%)    Predicted vote: Liked (50%)    Predicted vote: Liked (50%)    Predicted vote: Liked (50%)
Predicted vote: Disliked (55%) Predicted vote: Disliked (55%) Predicted vote: Disliked (55%) Predicted vote: Disliked (55%) Predicted vote: Disliked (60%) Predicted vote: Disliked (60%)
```

Element to design
[✓ Button] Input Table Text <div> Border

Genetic Algorithm Controls
Select Multiple: unchecked
Random Seed: unchecked
Selection method: [✓ Elitist] Roulette Wheel
Crossover type: [✓ Single Point] Multi-Point Arithmetic Property Lock
Algorithm Type: [✓ Standard GA] RWGA

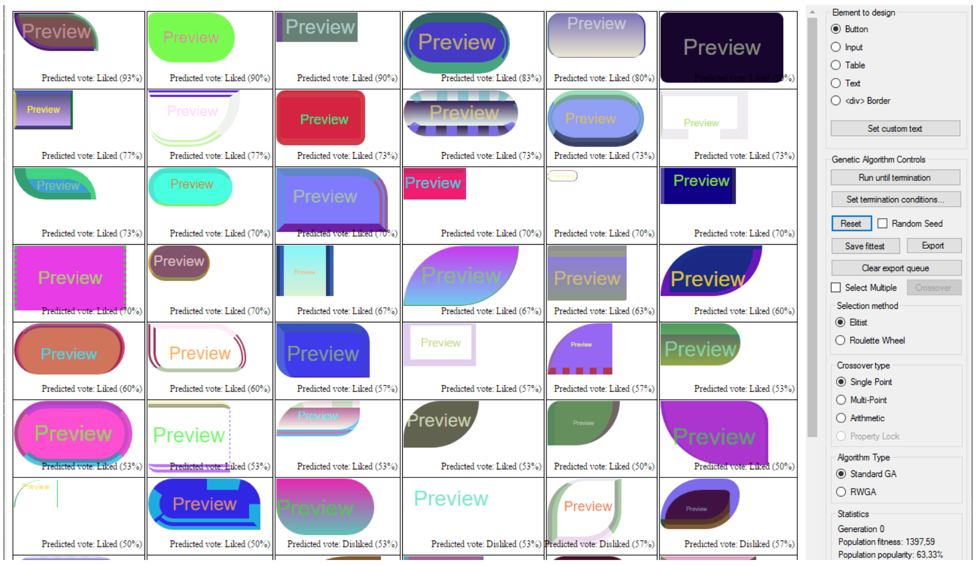
Statistics
Generation 0
Population fitness: 1595,54
Population popularity: 60%

(d) K = 20


Predicted vote: Liked (92%)
Predicted vote: Liked (92%)
Predicted vote: Liked (88%)
Predicted vote: Liked (84%)
Predicted vote: Liked (84%)
Predicted vote: Liked (80%)
Predicted vote: Liked (80%)
Predicted vote: Liked (80%)
Predicted vote: Liked (76%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (72%)
Predicted vote: Liked (68%)
Predicted vote: Liked (68%)
Predicted vote: Liked (68%)
Predicted vote: Liked (68%)
Predicted vote: Liked (68%)
Predicted vote: Liked (68%)
Predicted vote: Liked (64%)
Predicted vote: Liked (60%)
Predicted vote: Liked (60%)
Predicted vote: Liked (56%)
Predicted vote: Liked (56%)
Predicted vote: Liked (56%)
Predicted vote: Liked (56%)
Predicted vote: Liked (52%)
Predicted vote: Liked (52%)
Predicted vote: Liked (52%)
Predicted vote: Disliked (52%)
Predicted vote: Disliked (52%)
Predicted vote: Disliked (52%)
Predicted vote: Disliked (56%)
Predicted vote: Disliked (56%)
Predicted vote: Disliked (56%)
Predicted vote: Disliked (56%)
Predicted vote: Disliked (56%)
Element to design
Button
Input
Table
Text
<div> Border
Set custom text
Genetic Algorithm Controls
Run until termination
Set termination conditions...
Reset
Random Seed
Save fittest
Export
Clear export queue
Select Multiple
Crossover
Selection method
Elitist
Roulette Wheel
Crossover type
Single Point
Multi-Point
Arithmetic
Property Lock
Algorithm Type
Standard GA
RWGA

Statistics
Generation 0
Population fitness: 1640.5
Population popularity: 56.67%

(e) K = 25


Predicted vote: Liked (93%)
Predicted vote: Liked (90%)
Predicted vote: Liked (90%)
Predicted vote: Liked (83%)
Predicted vote: Liked (80%)
Predicted vote: Liked (77%)
Predicted vote: Liked (77%)
Predicted vote: Liked (73%)
Predicted vote: Liked (73%)
Predicted vote: Liked (73%)
Predicted vote: Liked (73%)
Predicted vote: Liked (73%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (70%)
Predicted vote: Liked (67%)
Predicted vote: Liked (67%)
Predicted vote: Liked (63%)
Predicted vote: Liked (60%)
Predicted vote: Liked (60%)
Predicted vote: Liked (60%)
Predicted vote: Liked (57%)
Predicted vote: Liked (57%)
Predicted vote: Liked (57%)
Predicted vote: Liked (53%)
Predicted vote: Liked (53%)
Predicted vote: Liked (53%)
Predicted vote: Liked (53%)
Predicted vote: Liked (53%)
Predicted vote: Liked (50%)
Predicted vote: Liked (50%)
Predicted vote: Liked (50%)
Predicted vote: Liked (50%)
Predicted vote: Disliked (53%)
Predicted vote: Disliked (53%)
Predicted vote: Disliked (57%)
Predicted vote: Disliked (57%)
Element to design
Button
Input
Table
Text
<div> Border
Set custom text
Genetic Algorithm Controls
Run until termination
Set termination conditions...
Reset
Random Seed
Save fittest
Export
Clear export queue
Select Multiple
Crossover
Selection method
Elitist
Roulette Wheel
Crossover type
Single Point
Multi-Point
Arithmetic
Property Lock
Algorithm Type
Standard GA
RWGA

Statistics
Generation 0
Population fitness: 1397.59
Population popularity: 63.33%

(f) K = 30

Figure 39: Classification results for static population based on varying K-values

As seen in the above figures, the classification of designs changes based on the value of K. Designs that may have been classified lower are put higher up on the list, and vice versa. This suggests that the value of K should be selected carefully to achieve the best classification. The implications of this will be discussed further.

Using K-NN based fitness for 1000 generations resulted in the following execution times:

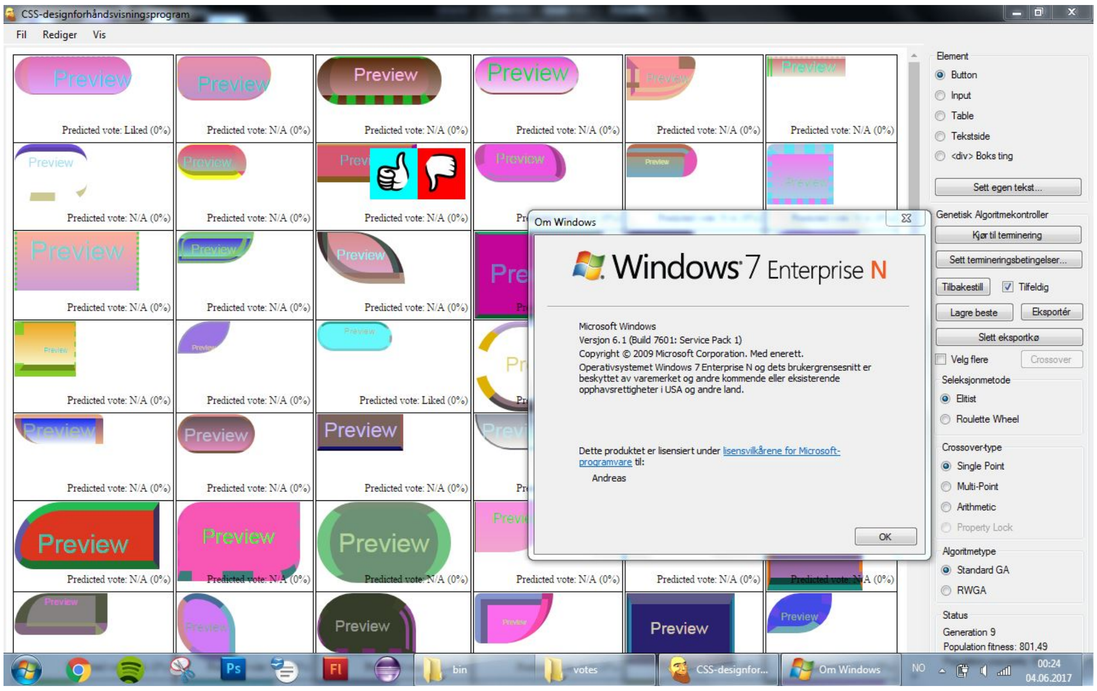
Crossover/selection	Elitist selection	Roulette wheel
Single point	3.71s	3.98s
Multi-point	3.72s	3.86s
Arithmetic	3.89s	4.21s

Table 18: Execution time for 1000 generations

Running K-NN classification for every design in every generation resulted in severely slower program run-times, as discussed in section 4.8. Using roulette wheel selection with multi-point crossover resulted in a total run-time of 31.28s, before 80% of the population was classified as Liked. Convergence took 76 generations. After the classification was divided into partitions and run in multiple threads, performance increased noticeably: 80% was reached after 6.37 seconds using the same initial population, needing 54 generations. A data set of 525 collected votes was used during this classification benchmark.

5.13 Testing outside the development environment

The program executable has been tested and confirmed working on a computer with Windows 7 without Visual Studio installed:



Predicted vote: Liked (0%)

Predicted vote: N/A (0%)

Element: [✓ Button] Input Table Tekstside <div> Boks ting

[✓ Tilfeldig]

Velg flere

Seleksjonmetode: [✓ Elitist] Roulette Wheel

Crossover-type: [✓ Single Point] Multi-Point Arithmetic Property Lock

Algoritmetype: [✓ Standard GA] RWGA

Status

Generation 9

Population fitness: 801,49

Om Windows

Windows 7 Enterprise N

Microsoft Windows
Versjon 6.1 (Build 7601: Service Pack 1)
Copyright © 2009 Microsoft Corporation. Med enerett.
Operativsystemet Windows 7 Enterprise N og dets brukergrensesnitt er beskyttet av varemerket og andre kommende eller eksisterende opphavsrettigheter i USA og andre land.

Dette produktet er lisensiert under lisensvilkårene for Microsoft-programvare til:
Andreas

00:24
04.06.2017

Figure 40: Program tested outside the development environment

However, in order to run the program, version 4.5.2 or newer of the Microsoft .Net framework as well as the Visual C++ Redistributable Package for Visual Studio 2013 (x86) are required. Both should be included on newer versions of Windows, or may be downloaded for free through Microsoft's pages.

6 Discussion of results

The first implementation was simple, yet it served as the foundation of the implementation throughout the project. Its simplicity made it easy to implement, and produced quick results. This was beneficial in order to visualize further development and extensions of the program. Focus was on making the program as visual as possible, and time was spent on implementing a responsive GUI that updated the color of every individual for every generation. This was fascinating to watch, however not really related to the genetic algorithm. As later implementations omitted this visualizer all together, in hindsight more time could have been spent on genetic algorithm research.

Adding more elements to the basic chromosome using the existing framework proved to be a good way to visualize a simple web page layout. The restructuring of the code made it easier to work with, and to add new properties. However, the transition from Windows Forms-based graphics to a proper web page rendering system added a new layer of complexity to the program. Preview of designs now depended on multiple steps of reading and writing files, and finally displaying them using the browser. On the other hand, previewing the designs directly in the program resulted in a more user-friendly and agile iterative design process. Overall, this was a necessary step to avoid the use of en external program to view the designs.

The duplicate designs observed in the initial populations of the first implementations was a result of the default random seed. In C#, this is based on the system clock, meaning that generating multiple random numbers in rapid succession may lead to duplicates, especially on weaker hardware. As the *Guid*-class is more or less a unique value, using it as the random seed increases randomness.

The poor performance of the elitist selection scheme observed in table 13 and table 14 is due to the implementation. Crossover is performed by randomly selecting one of the top 50%, and crossing it with the least fit members. This eliminates the less fit member, but results in lackluster performance. This was not given much attention during development, as the roulette selection scheme combined with multi-point crossover was sufficiently fast.

Choosing to only focus on design of single elements only, as opposed to the full web page, was another necessity to reduce program complexity and maintain user friendliness. As discussed earlier, a web page may be composed of multiple buttons, forms and tables, in addition to background design, layout and menus. Combining all these properties into a single chromosome, and adding support for all possible combinations would exceed the scope of this thesis by far, and take development time away from the genetic algorithm part. Early testing showed that matching buttons and menus were few and far between, not to mention an appropriate layout. By concentrating on the design of single elements, it became easier for the user to reach a set of solutions matching their preference.

The addition of the export-feature was more as a test, to be able to view generated designs when using the Windows Forms-based rendering system. The feature was extended and kept to let future users of the program export and use the design further.

Using the limiter helper variable made the population more uniform, and arguably, more aesthetically pleasing. Linking each multi-property feature with an associated limiter was the result of observing existing web design, and reaching the conclusion that most elements have only one border style, color and radius. This could have been solved by simply removing the code, but some designs with multiple unique features were deemed visually interesting, and thus the support was kept. As seen in the resulting screenshots, the addition of the variable turned out to be an adequate solution to this challenge.

Being able to manually select multiple individuals for crossover resulted in more control for the user. Using the arithmetic crossover implementation for this was considered logical, as it was assumed the user wanted to create a mix of the selected design, as an analog to nature. This implementation worked as expected, due to the way properties were represented. This was especially true for colors and border roundness, as they are linear, with extremes at each end of the value range. Properties like border style and font are not, and are thus averaged in a non-standard way. A way to make this more consistent will be discussed later.

Using weights to create population diversity was not a main part of this thesis, and was not given much attention. In short, a limited set of results show that this approach produced satisfactory results in the context of this project.

Being able to parse a known design and basing an initial population on this was a good way to give the user even more control over the design process. Using crossover to introduce the genetic material resulted in an adequate mix of features from the known design, in addition to random exploration provided by the algorithm.

Limiting the range of the random number generators using the initial conditions produced expected results considering the initialization process of the chromosome class. However, the current implementation only limits the initial population, and does not guarantee that the population will stay in the set part of the solution space due to random mutations. This is handled indirectly, as individuals with that greatly differs from the fittest are given a lower fitness score and thus less priority during the selection process.

As one may see from fig. 31b, the resulting population after 4455 generations is arguably more uniform than the one of fig. 31a as a result of the normalization of distance. This is because all properties are weighted equally, unlike the previous implementation where properties with bigger ranges became indirectly prioritized. For example, it it possible to see the padding property of fig. 31a is more varied than the one found seen in fig. 31b. Similarly, the font size and border style of the fittest individual, which are represented by smaller values, are present in a larger part of the resulting population, which implies that all properties are treated equally. It should be noted that the sample size of fig. 31 is relatively small, which may lead to false assumptions about the above statement. However, this is sufficient in respect to the topic of the thesis, as it is proven for a small set of solutions.

As seen in fig. 39, the value of K may severely impact the classification of designs. By examining each set of classifications and their K-values, it may be possible to determine the optimal number of neighbors. Furthermore, the high number of dimensions may influence the classification. In their work, K. Beyer et al.[37], suggest that the classification becomes less effective dealing with high dimensional feature vectors. In the implementation discussed in this thesis, the total number of dimensions of the base chromosome is 61. This may explain the unexpected results observed in fig. 35. However, it is still possible to observe the effect of the classification algorithm. A suggested method of approach to improve this will be discussed later.

In terms of computation time, the results in table 18 outperforms the web page designer by Oliver et al. in section 2.1. However, this is most likely due to outdated hardware compared to the test system used in this thesis. In addition, since rendering is done on the client, preview is virtually instant when the algorithm terminates. The only limitation is the speed of the hard drive, as the CSS-file is written to disk and then read back into the browser.

7 Further development

The program performs the operations as required by the tests and benchmarks of the implementation. However, there is still room for improvements. First of all, the user interface was designed with the purpose of testing as many functions as possible using buttons. Testing shows that a more intuitive and user friends interface should be considered. Secondly, the current structure of the code is not the best. To achieve clean and more readable code, restructuring is needed. Leftover code from the earlier versions of the program is still present. This does not impact performance, however. Some functions could on the other hand be rewritten to boost performance.

An important issue with the current implementations in regards to the helper variables discussed in section 4.5 is the fact that even if the helper variable says an element only has one unique border, the fitness function and the classification algorithm still sees the "hidden" values, which impacts the results in an undesirable way. To fix this, setting all these unused variables to 0 after the chromosome is generated is expected to solve this fault.

Another issue regarding representation effecting the arithmetic crossover are properties using mapping functions. In order to make the average function more consistent with their visual representation, values should be manually arranged in such a way that averaging them makes sense in respect to the visual result. An example is fonts. Strategically ordering fonts in the mapping function in such a way that the average of their indices correspond to an "average" of their visual forms is considered a solution to this.

As it currently stands, the votes are collected for a single user only. By distributing the votes to multiple users, it would be possible to incorporate the preferences of different people into the classification algorithm, introducing an element of crowd intelligence.

As seen, changing the K number of neighbors for the classification algorithm resulted in different classifications. Currently, this value is hard-coded to 15. In order to achieve the best classification possible, being able to change the K value should be investigated. This could be done either manually, or by implementing a second simple GA to optimize this value based on various parameters. Furthermore, introducing a decay factor for votes could be looked into. By doing this, older designs will count less during classification. Newer, and more frequently occurring designs would therefore be prioritized. This allows for the user's taste to change over time, as the user starts to consciously select different design styles than before. This may also alleviate some of the work for the hardware, as the number of votes to sort and go trough will remain somewhat constant.

Another hard-coded variable is the mutation chance. Allowing the user to experiment with different mutation rates to control population diversity could be added.

As discussed, the high number of dimensions of the chromosome may impact the classification performance. Beyer et al. recommend no more than 10 dimension for the feature vector. Principal Component Analysis (PCA) is a technique often used in AI to reduce the complexity of data to be analyzed by an algorithm. Running PCA on the votes before classification in order to reduce the dimensions is expected to have a positive impact on the performance of the classification.

The program only supports a limited set of design variables. Extending the chromosome to cover a broader solution space is a possibility.

8 Conclusion

In the introduction, three major goals were defined:

1. Creation of a web page design system able to demonstrate independent creativity while learning to absorb quality measures in regards to human preference
2. Implementation and testing of a dynamic genetic algorithm fitness function able to change over time based on user feedback
3. Improve performance compared to previous work of similar nature

Results show that using genetic algorithms to produce unique visual designs by recombining features and explore the solution space may serve as an inspiration for aspiring designers. Furthermore, by introducing learning in the form of voting and classification, it is possible to capture trends and personal preference over time. Using classification as a fitness function in the genetic algorithm has the potential of producing aesthetically pleasing results without human interaction. However, this is dependent on a proper set of votes and the correct K-value. Results show a variance in classification when changing the number of neighbors, which means that selecting a correct value is important in order to achieve a good classification. The program produced lets the user explore and review generations of designs in an iterative process, in addition exporting and to manually fine-tuning the results for further use. The work produced extends the implementations discussed in state-of-the-art in multiple ways, including the addition of more CSS features and faster results, in addition to a new fitness function in the form of the aforementioned classification algorithm. This is also true in regards to representation of solutions. For crossover, single point, multi point, arithmetic and a ”property lock” scheme were implemented and benchmarked. Selection was carried out using either elitist selection, roulette wheel or manual by the user. Experiments show that the fastest convergence is achieved by combining multi point crossover with roulette wheel selection, while manual crossover is great for controlling the direction of the search in finer detail.

As mentioned in the introduction, combining the social and creative intelligence of human designers with the speed and accuracy of intelligent system are expected to produce interesting results. Allowing the designer to explore a large set of design variations with minimal effort is expected to boost overall productivity. The results observed during the project work may help confirm this, as the implemented features are able to serve as a tool during a creative process. Due to its random nature, features are recombined with no concern regarding established norms, only limited by the fitness function. When classifying completely random solutions, designs that have been left unexplored by humans due to practical or aesthetic constraints may show up and be surprisingly original. In theory, this will happen if the classification algorithm is able to pick up hidden patterns for good designs previously unknown to humans. In addition, having large sets of proposed solutions is imagined to reduce time further.

As seen in the results, the implementation of this thesis outperforms similar work in regards to execution time and available design features. However, the program is still a prototype, and requires more work to be more user friendly and intuitive to be able to be used in a professional setting.

To sum up, the solution is imagined to benefit from investigating and optimizing the theory of K-NN classification to boost performance. Giving the user even more control over the GA parameters could also be tested. Perfecting the classification of design using aforementioned suggestions is expected to result in a program capable of assisting human designers in regards to exploration and testing of new radical designs, in addition to serve as a way of observing design trends based on data from multiple users. Finally, making the user interface of the program more intuitive could be done.

9 List of appendices

References

[1] M. Brian, “Google’s alphago is the best go player in the world.” https://www.engadget.com/2017/05/25/google-alphago-ai-deepmind-ke-jie-go-win/. Accessed: 2017-01-06.

[2] A. Joglekar and M. Tungare, “Genetic algorithms and their use in the design of evolvable hardware,” 2000.

[3] C. Prenzler, “Tesla’s latest autopilot 2.0 update handles ‘curvy’ highways like a boss.” http://www.teslarati.com/tesla-autopilot-highway-turns-local-driving-video/. Accessed: 2017-01-06.

[4] R. Girling, “Ai and the future of design: What skills do we need to compete against the machines?.” https://www.oreilly.com/ideas/ai-and-the-future-of-design-what-skills-do-we-need-to-compete-against-the-machines. Accessed: 2017-29-05.

[5] R. Girling, “Ai and design: What will the designer of 2025 look like?.” https://www.artefactgroup.com/articles/ai-designer-2025/. Accessed: 2017-29-05.

[6] S. Kemp, “Digital in 2017: Global overview.” https://wearesocial.com/uk/special-reports/digital-in-2017-global-overview. Accessed: 2017-29-05.

[7] Cisco Systems Inc., “The zettabyte era — trends and analysis – cisco.” http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html. Accessed: 2017-29-05.

[8] Netcraft Ltd., “May 2017 web server survey.” https://news.netcraft.com/archives/category/web-server-survey/. Accessed: 2017-29-05.

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[10] C. K. Fung, C. Kwong, K. Y. Chan, and H. Jiang, “A guided search genetic algorithm using mined rules for optimal affective product design,” 2013.

[11] M. Khajeh, P. Payvandy1, and S. J. Derakhshan, “Fashion set design with an emphasis on fabric composition using the interactive genetic algorithm,” 2016.

[12] M. Perkowitz and O. Etzioni, “Adaptive web site - examining the potential use of automated adaptation to improve web sites for visitors.,” 2000.

[13] A. Oliver, N. Monmarché, and G. Venturini, “Interactive design of web sites with a genetic algorithm,” 2003.

[14] Prisma Labs Inc., “Prisma.” https://prisma-ai.com/. Accessed: 2017-29-05.

[15] Unknown, “The grid.” https://thegrid.io/. Accessed: 2017-29-05.

[16] G. Hobbins, “The first grid.io sites have surfaced and they’re kinda terrible.” https://www.designernews.co/stories/65265-the-first-gridio-sites-have-surfaced-and-theyre-kinda-terrible. Accessed: 2017-29-05.

[17] Y. Vetrov, "Algorithm-driven design: How artificial intelligence is changing design." https://www.smashingmagazine.com/2017/01/algorithm-driven-design-how-artificial-intelligence-changing-design/. Accessed: 2017-29-05.

[18] T. H. Nätt, *Snarveien til Dreamweaver CS3*. Gyldendal Norsk Forlag AS, 2007.

[19] w3schools.com, "Css reference." https://www.w3schools.com/cssref/default.asp. Accessed: 2017-29-05.

[20] A. Jahanian, S. V. N. Vishwana, and J. P. Allebach, "Colors —messengers of concepts: Visual design mining for learning color semantics," 2015.

[21] The Qt Company, "Qt — cross-platform software development for embedded & desktop." https://www.qt.io/. Accessed: 2015-21-08.

[22] Oracle, "Trail: Creating a gui with jfc/swing." http://docs.oracle.com/javase/tutorial/uiswing/. Accessed: 2017-01-06.

[23] Microsoft, "Windows forms." https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx. Accessed: 2017-01-06.

[24] D. Whitley, "A genetic algorithm tutorial," 1994.

[25] V. Pieterse and P. E. Black, ""fisher-yates shuffle", in dictionary of algorithms and data structures." https://www.nist.gov/dads/HTML/fisherYatesShuffle.html. Accessed: 2017-12-02.

[26] Microsoft, "Guid structure." https://msdn.microsoft.com/en-us/library/system.guid(v=vs.110).aspx. Accessed: 2017-05-06.

[27] Microsoft, "Webbrowser.version property." https://msdn.microsoft.com/en-us/library/system.windows.forms.webbrowser.version(v=vs.110).aspx. Accessed: 2017-03-05.

[28] w3schools.com, "Css3 browser support." https://www.w3schools.com/cssref/css3_browsersupport.asp. Accessed: 2017-29-05.

[29] I. Hickson, "Acid tests - the web standards project." http://www.acidtests.org/. Accessed: 2017-29-05.

[30] Awesomuim, "Awesomuim." http://www.awesomium.com/. Accessed: 2017-03-05.

[31] A. Young, "An open-source component for embedding mozilla gecko (firefox) in .net applications.." https://code.google.com/archive/p/geckofx/. Accessed: 2017-03-05.

[32] A. Maitland, H. Gupta, and P. Lundberg, "Cefsharp - fast web browser for winforms and wpf apps." http://cefsharp.github.io/. Accessed: 2017-03-05.

[33] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms," 2006.

[34] W3C, "Checkpoint 2.2 - ensure that foreground and background color combinations provide sufficient contrast when viewed by someone having color deficits or when viewed on a black and white screen." https://www.w3.org/TR/AERT#color-contrast. Accessed: 2017-06-04.

[35] Google Inc., "Google fonts." https://fonts.google.com/. Accessed: 2017-12-04.

[36] L. E. Peterson, “K-nearest neighbor,” *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009. revision #136646.

[37] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, “When is “nearest neighbor” meaningful?,” 1999.

Appendix 1 - Source code

The complete source code may be found at `https://source.uit.no/aja073/ProjectDazzle/tree/master` or, alternatively, `https://bitbucket.org/ADJansson/projectdazzle`.

Appendix 2 - Project description

Faculty of Engineering Science and Technology

Department of Computer Science and Computational Engineering

UiT The Arctic University of Norway

Genetic algorithms (GA) for adaptable design

Andreas Dyrøy Jansson

Thesis for Master of Science in Computer Science

Problem description

This project proposal seeks to explore creative AI systems. This implies systems that can create new design and develop unprecedented solutions to problems posed to them. Emphasis will be placed on Genetic Algorithms. The application of AI methods in web design and smart phone app production is of special interest.

The project work encompasses three main parts

1. Literature study
2. Conceptualization and implementation
3. Test and verification

The proposal will explore state-of-the-art for AI based design systems. Emphasis should be placed on literature that applies genetic algorithms (GA) alone or as part of a suit of methods. The principles for using GAs to produce formative, adaptable and creative solutions should be stressed. The first idea is to study representation issues. This should include design features as well as functional constraints, requirements and priorities. The question of how a web page should adapt to a particular user group or certain circumstances should be investigated. Studying methods for evaluating and ranking tentative design solutions are also important. In this respect references to analogy-based reasoning and case-based reasoning could be useful.

Based on these investigations the student will develop a GA concept that will be able to create novel solutions that demonstrate creative capabilities through extensive recombination of many features. The student should conceptualize his own method for producing an adaptable and artistic design of web pages by means of a GA with or without other type of AI support. The choice of GA type will be essential. Finally, decisions regarding operators and their use must be settled.

The student must define a set of criteria for evaluation and verification of the quality of the design results generated by his system. These could be general in nature, specific for a particular user group or circumstantial. Tests should be designed accordingly and performed on a rich set of cases.

The work should be documented and the software should be delivered in the form source code of and executables tested beyond the development environment. It should also be accompanied by a video showing different demonstrations of the system as well as tests performed.

A starting point for the literature study could sources like:

1. A guided search genetic algorithm using mined rules for optimal affective product design, CKY Fung, CK Kwong, KY Chan… - Engineering …, 2014 - Taylor & Francis

2. Fashion and Textiles December 2016, 3:8 Fashion set design with an emphasis on fabric composition using the interactive genetic algorithm by Mahnaz Khajeh, Pedram Payvandy, Seyyed Javad Derakhshan

3. Quantifying Aesthetics of Visual Design Applied to Automatic Design,Part of the series Springer Theses pp 15-55, Date: 21 June 2016

4. Design Mining Color Semantics, Ali Jahanian, The Art of Artificial Evolution, Springer, Part of the series Natural Computing Series pp 3-37, Evolutionary Visual Art and Design

5. Pervasive Computing in the Supermarket: Designing a Context-Aware Shopping Trolley, Darren Black (Systematic, Denmark), Nils Jakob Clemmensen (Nordjyske Medier, Denmark) and Mikael B. Skov (Aalborg University, Denmark) Source Title: International Journal of Mobile Human Computer Interaction (IJMHCI) 2010

6. Adaptive Web sites By Mike Perkowitz, Oren Etzioni
Communications of the ACM, Vol. 43 No. 8, Pages 152-158

7. Adaptive Web Sites: an AI Challenge, By Mike Perkowitz, Oren Etzioni , IJCAI 199

8. INTERACTIVE DESIGN OF WEB SITES WITH A GENETIC ALGORITHM A. Oliver, N. Monmarché, G. Venturini, IADIS International Conference WWW/Internet 2002

Dates

Date of distributing the task:	17.01.2017
Date for submission (deadline):	06.06.2017

Contact information

Candidate	Andreas Dyrøy Jansson aja073@post.uit.no
Advisor at UiT-IVT	Bernt Bremdal bernt.bremdal@uit.no

General information

This master thesis should include:

- ❄ Preliminary work/literature study related to actual topic
  - A state-of-the-art investigation
  - An analysis of requirement specifications, definitions, design requirements, given standards or norms, guidelines and practical experience etc.
  - Description concerning limitations and size of the task/project
  - Estimated time schedule for the project/ thesis
- ❄ Selection & investigation of actual materials
- ❄ Development (creating a model or model concept)
- ❄ Experimental work (planned in the preliminary work/literature study part)

❄ Suggestion for future work/development

Preliminary work/literature study

After the task description has been distributed to the candidate a preliminary study should be completed within 4 weeks. It should include bullet pints 1 and 2 in "The work shall include", and a plan of the progress. The preliminary study may be submitted as a separate report or "natural" incorporated in the main thesis report. A plan of progress and a deviation report (gap report) can be added as an appendix to the thesis.

In any case the preliminary study report/part must be accepted by the supervisor before the student can continue with the rest of the master thesis. In the evaluation of this thesis emphasis will be placed on the thorough documentation of the work performed.

Reporting requirements

The thesis should be submitted as a research report and could include the following parts; Abstract, Introduction, Material & Methods, Results & Discussion, Conclusions, Acknowledgements, Bibliography, References and Appendices. Choices should be well documented with evidence, references, or logical arguments.

The candidate should in this thesis strive to make the report survey-able, testable, accessible, well written, and documented.

Materials which are developed during the project (thesis) such as software/codes or physical equipment are considered to be a part of this paper (thesis). Documentation for correct use of such information should be added, as far as possible, to this paper (thesis).

The text for this task should be added as an appendix to the report (thesis).

General project requirements

If the tasks or the problems are performed in close cooperation with an external company, the candidate should follow the guidelines or other directives given by the management of the company.

The candidate does not have the authority to enter or access external companies' information system, production equipment or likewise. If such should be necessary for solving the task in a satisfactory way a detailed permission should be given by the management in the company before any action are made.

Any travel cost, printing and phone cost must be covered by the candidate themselves, if and only if, this is not covered by an agreement between the candidate and the management in the enterprises.

If the candidate enters some unexpected problems or challenges during the work with the tasks and these will cause changes to the work plan, it should be addressed to the supervisor at the UiT or the person which is responsible, without any delay in time.

Submission requirements

This thesis should result in a final report with an electronic copy (i.e. CD/DVD, memory stick) of the report included appendices and necessary software codes, simulations and calculations. The final report with its appendices will be the basis for the evaluation and grading of the thesis. The report with all materials should be delivered in one signed loose leaf copy, together with three bound. If there is an external company that needs a copy of the thesis, the candidate must arrange this. A standard front page, which can be found on the UiT internet site, should be used. Otherwise, refer to the “General guidelines for thesis” and the subject description for master thesis.

The final report with its appendices should be submitted no later than the decided final date. The final report should be delivered to the adviser at the office of the IVT Faculty at the UiT.