UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

HUMAN-IST
RESEARCH CENTER

Evolution driven, multivariate testing on the web

Jan Kuehni[1]

Supervisor: Prof. Dr Denis Lalanne

August 28, 2015

Department of Informatics - Master Project Report

Département d'Informatique - Departement für Informatik • Université de Fribourg - Universität Freiburg • Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland

phone +41 (26) 300 84 65 fax +41 (26) 300 97 31 Diuf-secr-pe@unifr.ch http://diuf.unifr.ch

[1] jan.kuehni@unifr.ch, DIUF, University of Fribourg

Abstract

This master project discusses current approaches in AB testing and proposes the use of evolutionary techniques to improve and automate multivariate, controlled experiments on the web. In order to simplify the testing process, we have developed an open sourced framework that is quick to deploy and facilitates the initialisation and monitoring process of evolutionary driven testing.

The project aim is to find out whether the implemented testing suite is a valid alternative to classic multivariate testing and whether we can find significant difference in the performance in a series of evolutionary algorithms (basic evolution, trend supported evolution, score based evolution and breadth first search). We consider our proposed solution to be successful if the testing suite is able to consistently produce converging solutions for the specified optimisation problem.

In order to validate the evolutionary approach and compare the four evolutionary search algorithms, we performed an online evaluation featuring four tests in which user behaviour was tracked to improve a web page in terms of a specified goal. After an initial pre test, the final evaluation was performed in a time frame of ten days. Through the use of scientific mailing lists we were able to recruit over a thousand participants. Our initial estimates assumed a much lower participation rate which led to the selection of non-optimal evolutionary parameters. All users performed the same four tests but were evenly distributed among the four evolutionary algorithms.

The recorded data shows that evolutionary algorithms represent a valid approach for performing automated testing on the web. We were able to find converging values in most of the tests and thus improve the overall success rate of all test pages. Due to the low amount of fitness tests that were performed for each mutation as well as the varying nature of the tests, comparing the performance of the four implemented evolutionary approaches has proven to be difficult. Nonetheless we were able to gain a good understanding of the advantages and possible downsides of using each of the testing procedures.

Based on the identified issues as well as the data gathered from the evaluation, we think that the framework will not necessarily serve as a replacement for classical AB or multivariate testing but should rather be utilised as a tool for fine tuning specific elements on web pages. Further research will have to demonstrate how the framework performs with a higher number of evolved variables that have a lower impact on the specified success criteria. Additional work is also necessary to improve the automated adaptation of evolutionary parameters in order to reduce the frameworks conversion time and react to changes in the flow of users.

Contents

List of Figures

Listings

1 Introduction

Today, many web pages are still designed based on the assumption that developers and user experience designers are able to come up with the best and most appropriate solution for a given problem. In general, developers work on a few possible implementations, internal user testing then decides which version is implemented and published on the web. The instantiated solution remains static and is only adjusted through periodic updates or when a completely new version is designed. Without testing, this new version may even be worse in terms of the goal the page tries to achieve.

The main problem of this non iterative approach lies in the nature of web interaction. Having to create an optimal solution that works for an ever changing and strongly heterogeneous user group is very challenging. Fluctuation in user behaviour over time as well as the large number of possible visual configurations make the task of creating and optimal page virtually impossible [17].

Within the last five years, AB testing (also called split testing) has become increasingly popular and stands for a change in philosophy as well as in business strategy. Instead of creating one static solution, more and more companies start to shift their focus away from the opinions of developers towards the actual behaviour of their user base by asking a simple question: is the current implementation optimal for the problem we would like to solve? The solution to this question lies in an evidence-based practice for which some form of AB or multivariate testing is implemented [13]. In its core, this means that designers implement not one but a multitude of possible solutions for a given problem and measure the performance of each version based on a pre defined success criterion. This all happens with the intention of find the most appropriate implementation for that specific goal. When we talk about a goal, it can literally mean anything. It could be a company trying to maximise the amount of users that access their web store, that click on a subscribe button, that scroll down on a landing page or click on an advertisement. Since this process is quite tedious and needs a programmer to implement, a variety of testing frameworks have emerged in the past few years.

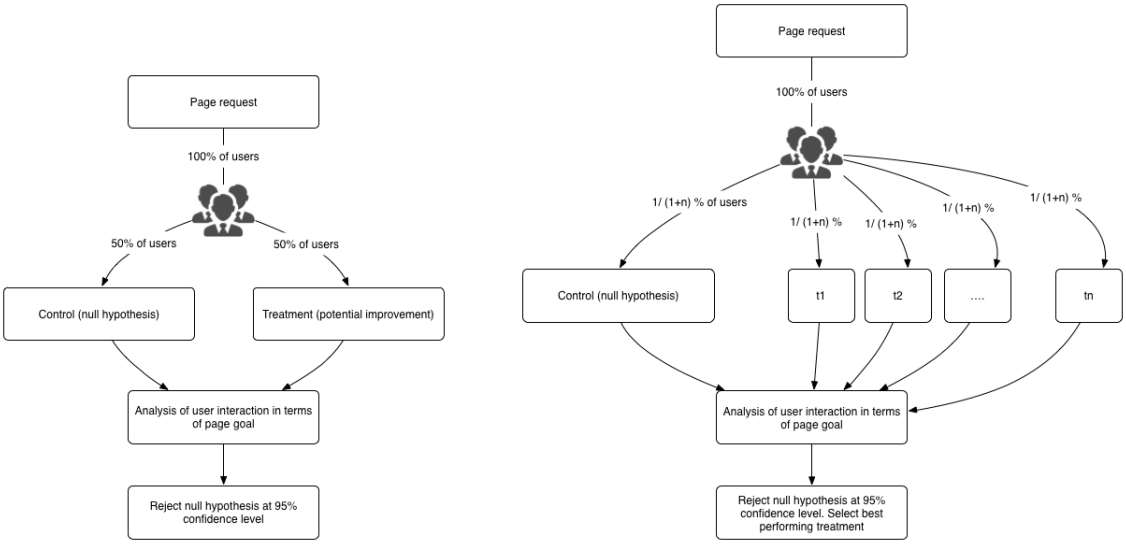
While the AB testing approach is very promising and can deliver great results if done correctly, it does not fully adhere to the described paradigm shift. In the standard AB or multivariate testing approach, developers are still manually creating the versions that will be tested. Even if they create three, ten or even more versions they are forcing their view of how the interaction is best designed upon the users. Experiments with political web pages has shown that even the slightest adjustments such as changes in colour, positioning and even font size can have a significant impact on how successful a certain page is in terms of the target metric it tries to achieve [6] .

When we look what AB testing actually does, it boils down to finding the best materialisation of user interface related variables. We can thus ask the question: why should we manually create all these different versions when the chances of finding the perfect configuration of all possible variables that are in play is very slim? We will first get a better understanding at how AB testing works and discuss possible solutions to further automate the testing process. Our aim is to find an automated search procedure that will find the optimal configuration of the variables in play and incorporate the solution into a reusable and extendable framework.

2 AB Testing

2.1 Introduction

AB testing consists of performing a randomised experiment in order to test a statistical hypothesis. The testing approach can vary and falls into two basic categories. The most basic form of AB testing is achieved by comparing a current implementation (control) to a another version (treatment). Users are randomly distributed to either the control or the treatment and information about user behaviour is logged (see fig. 1a). After a certain number of measurements (the exact number depending on the required confidence interval), we compare both versions in regards to the goal we want to achieve. The goal in the AB testing process is called Overall Evaluation Criterion (OEC) and represents a quantitative measure of the experiments objective. In statistical terms this is commonly referred to as the response or the dependent variable [15]. We associate the initial version as being the null hypothesis and try rejecting it based on the evidence we have gathered



(a) Classic AB testing

(b) Multivariate testing

Figure 1: Two types of controlled experiments

from the treatment [19]. Instead of the usual hypothesis testing procedure (t-test) we can also use Bayes theorem to compute the odds of the treatment having an impact when compared to the the control version [1].

When conducting split tests, we are usually interested to find out if the statistical means of the metric we would like to improve (usually the success criterion) is different for the control and the treatment.

$E(B) = \bar{x}_B - \bar{x}_A$

The null hypothesis should only be rejected when there is a significant difference of the means between the between both versions at a 95 percent confidence level [14]. The confidence level can be lowered depending on the type of testing that is performed as well as on the consequences a false selection would induce.

Instead of comparing only a single treatment to the null hypothesis, we can increase the number of treatments and by doing so move into the field of multivariate testing (also called multinomial testing). With this second approach, a single control version is compared to multiple treatments. Again we split up users into groups that correspond to the implemented versions and compare the performance in terms of the specified goal (see fig. 1b). This testing process can be iterative by continuously developing new treatments that are tested against the new null hypothesis.

When performing controlled experiments on the web, it is important to minimise influences which could have a negative effect on the test outcome or even skew the recorded user data in a way that would make it unusable. The most important factor which determines the success of a an AB test lies in the selection of the overall evaluation criterion [7]. The criterion is the central metric that is used for comparing the control and its treatment (or multiple treatments in a multivariate setting). If poorly chosen, none of the subsequent findings about a treatments performance has any meaning.

Another pitfall addresses issues about gathering user data. If a controlled experiment is repeatedly accessed by a robot and makes up a significant part of all recored interaction, we need to find a way to discard non human interaction. Since the differences between the treatment(s) and the control version might not have an influence on the robot, we end up with data that does no properly reflect user behaviour.

2.2 Origins

The practice of conducting controlled experiments originates in the 1920s when Sir Ronald A. Fisher conducted trials at Rothamstad Agricultural Experimental Station on England. He was an english statistician and evolutionary biologist who almost single-handedly created the foundations for modern statistical science. [4]. With the rise of the internet in the 1990s and the rapid growth of eCommerce, online controlled experiments started to increase in popularity. Today, a wide range of large online businesses such as Google, Amazon and eBay routinely conduct a large number of experiments in order to test their user interface and business processes [12]. But why is the web such a great source for performing controlled experiments? It has been shown that controlled experiments are especially useful when they are combined with agile software development [20]. Web pages have the advantage of being accessed world wide as soon as they are uploaded to a server. Changes to a page are live instantly and do not require a complicated installation/update and distribution process. And since users have to be online to view a web page, information about their behaviour can easily be tracked and stored on the server for later analysis.

2.3 Motivation

The main philosophy that underlines the theory of controlled experiments comes from the observation that often, our intuition about user behaviour as well as about their intentions is wrong. Colin McFarland said in his book called Experiment! that "No matter how much you think it's a no brainer, how much research you've done, or how many competitors are doing it, sometimes, more often than you think, experiment ideas simply fail." [16]. No matter how hard developers try to improve their solutions, they still might miss important factors, wrongly assess user intentions and implement useless features that negatively influence the success rate of their products. In order to maximise success, we need to perform controlled experiments with actual users. In the end, even if the gain is small, improving a page beyond the initial design leads to more customers and potentially an increased revenue.

2.4 Examples

Companies such as Google and Bing perform intensive user testing. Considering that both search engines generate most of their annual revenue by placing advertisements throughout their products and have millions of page views each day, it makes sense for them trying to maximise the amount of users that click on these ads. We will look at two small examples on how these two companies have used controlled experiments in the past in order to improve the revenue generated by advertisements.

(a) Bing Control (b) Bing Treatment

Figure 2: Example AB testing at Bing

2.4.1 Bing

The development at Bing wanted to allow companies to provide more information about their services when displaying advertisements in the search results [12]. In order to test the impact this feature would have on the percentage of users actually clicking on these ads, they performed a simple split test. The control version being the current implementation which does not display any additional information below the ad (see fig. 2a) and the treatment being the version with the new feature implemented (see fig. 2b). While even through the additional information increased the page load times and required more vertical space to be taken (thus showing less ads on the screen), the overall click rate went up drastically and the tested feature was implemented.

2.4.2 Google

Googles search result page could be called the companies cash cow. Each time a user clicks on one of the sponsored ads after searching for a term on the web, Google gets paid a certain amount of money. With more than one hundred million searches per month [18], even the slightest increase in user behaviour towards clicking ads can lead to a large additional income.

The blue colour that Google uses for it's search bar as well as for its advertisement links were initially chosen by a designer. The selected shade of blue was chosen because a majority of people at the Google office seemed to like it. But how could a designer know what exact shade of blue would lead to a maximal amount of ads being clicked upon? The obvious answer being "he can't", Googles former CEO Marissa Meyers decided to test which one out of 41 proposed shades of blue would lead to the most users clicking on displayed ads [11]. In order to find the best colour tone, they set up a multivariate test with 41 treatments being compared to the initial shade of blue and split up users equally (2.5 percent) to each page version. The results actually showed an significantly large enough change in user behaviour for them to modify the blue tone throughout their web page.

2.5 Testing Approaches

2.5.1 Manual testing

Manually setting up AB or multivariate tests can be a tedious process that requires developers to implement a fully working testing infrastructure which is able to distribute users equally to each treatment and log user behaviour in regards to the specified evaluation criterion. When iterating or creating completely new tests, the process has to be repeated. This is certainly not a satisfying solution, especially for smaller corporations. This is why several testing suites have emerged in the last few years, that allow performing controlled experiments without having to manually set up all the infrastructure.

2.5.2 Automated testing

We looked at several open source as well as commercial testing suites in the process of setting up our own testing library. Among the most widely used were Optimizely, Petri, Fusion and Maxymiser.

A common aspect to all these testing libraries is the ability to easily set up treatments with the help of a visual editor tool. It is to note that not all evaluated testing libraries support multivariate tests, many simply allow for the creation of a single treatment which then automatically tested against the control version after users are evenly distributed to both versions for some time. None of the frameworks seemed to support an iterative approach. If multiple split tests are being used, each test has to be set up manually from scratch.

The most widely used testing suite found on the web is Optimizely [2]. It features classic AB split tests as well as fully fledged multivariate tests. Different treatments can easily be set up with the help of a visual editor. The basic features are free to use, more advanced features such as multiple project support and custom Javascript require a paid subscription. With the paid version, Optimizely offers advanced features such as integration with Google analytics as well as the generation of heat maps based on user pointer behaviour. Other testing suites such as Petri do not feature a visual editor but offer an API that can can be used to initialise and run split tests.

2.5.3 Evolutionary testing

Our intention with this project is to further automate multivariate testing and fully adhere to the user centred approach. Instead of manually creating one or multiple treatments, we will discuss solutions that are able to find optimal variable configurations on their own. We envision the creation of a library in which the user can simply define the bounds in which a user interface element is able to change and leave it up to the library to find an optimal solution within that frame.

With multivariate testing, even if supported by one of the discussed testing libraries, designers still have to come up with and implement all individual versions manually. Logging user behaviour

and performing statistical tests will certainly be able to find the most successful out of the proposed solutions, but what if the best configuration for reaching the success criterion lies somewhere between two successful treatments? What if the most optimal treatment looks completely different than any of the proposed versions? In this case, no matter how many tests we run, we will not be able to reach that ideal configuration.

Another issue of multivariate testing is the possibility of producing versions that perform significantly worse than the original null hypothesis. In this case, we attribute a large amount of users to a page that will be less successful than the original for some time. In the scenario of a web store this could mean losing potential customers which can have a significant impact on revenue. Another important factor is page consistency. If we perform AB testing on the order button of a web shop, we might confuse users that visit the page on several occasions from different machines and find the button at different position every time the page loads. Mechanisms exist that can prevent a user from seeing multiple treatments trough the use of cookies, but the issue remains if different devices are used to access the page or cookies are disabled.

We have seen that the problem AB testing tries to solve, boils down to a search problem. We have a large search space which consists of a multitude of variables and we are trying to find the best possible variable configuration in regards to a specified metric. Looking at different types of search algorithms, taking into account the importance of process automation as well as page consistency, evolutionary programming seems to be the perfect match for the described search problem. Instead of manually creating varying versions of a web page, why not automate the process of creating treatments and use evolution to gradually approach the optimal solution?

If we look at the Bing example and how they AB tested displaying additional information at the bottom of every ad, we can imagine how evolutionary testing could lead to improved results. When improving the way advertisements are displayed, only two versions were compared to one another while leaving a lot of other potentially significant variables out of the equation. With evolutionary testing, we could further find out what the optimal font size, letter spacing or opacity of the added information would look like. We could also use the evolutionary approach to test for different amounts of information displayed with the advertisements as well as optimal positioning on the page. With the multivariate test conducted by Google, instead of having to set up 41 different treatments of the page, we could simply have initialised an evolutionary testing process that gradually approaches the optimal blue tone on all relevant page elements.

2.5.4 Project aim

The main goal of this project will be to create a library of framework which enables fully automated evolutionary testing on the web. The created library will be easy to deploy and feature a fully implemented user interface through which evolutionary processes can be initialised, monitored and stopped. We will implement a series of evolutionary algorithms in order to asses which approaches offer the best performance and how well they are suited for a wide range of optimisation problems found on the web. A final evaluation will put the developed framework to the test. Our goal is to find out if evolutionary algorithms are indeed suited for automating multivariate testing and to which extent they might be able to replace or supplement classical AB testing in the future.

3 Evolution driven, multivariate testing

3.1 Introduction

Inspired by biological processes that happen within the DNA of living organisms, the idea of evolutionary processes was transferred to the computational domain. While the exact processes behind the theory of natural selection are quite complex and some even unknown to this day, the underlying idea is rather simple: all animals and plants that still exist today are the result of millions of years of adaptation [8]. The organisms which are able to acquire the most resources in order to survive in their environment and procreate will pass down their genetic information to their descendants. Less fit organisms will have a shorter lifespan and will thus be less likely to reproduce. With time, the population is said to evolve and only contain organisms that feature

characteristics which increase their chances of survival. The process of adaption is only possible because of a phenomenon called mutation. Mutation occurs when an organism is procreating and genetic information is damaged in the copying process. In some cases, this damaged DNA can lead to features in organisms which make them better adapted to their environment.

3.2 Evolutionary computation

The first proposal for using evolution as a mean of solving optimisation problems, dates back to the mid-1960s, when John Holland of the University of Michigan started his experiments on evolutionary programming [10]. Evolutionary computation techniques abstract natural processes into algorithms that can be used to search for solutions to complex problems where little is known about the underlying search space. Unlike traditional search algorithms, such as random or heuristic sampling, evolutionary algorithms are population based and perform an efficient, directed search. The general process that evolutionary computation uses to solve optimisation problems is based on an iterative approach that approximates the optimal solution by gradually changing characteristics of individuals [3].

In order to achieve an efficient, directed search, mutated individuals are subjected to a function which determines their survivability. This fitness function is an essential part of evolutionary computing and must be specific to the problem being solved. Similar to a biological organism procreating, individuals that successfully pass the fitness test are selected for reproduction. New individuals (offspring) which make up a new generation based on the genetic information of their parents, are generated by using reproduction operators. These operators specify how an individuals genetic information is modified during the reproductive process.

Selecting winners for reproduction, defining how many fitness tests an individual has to pass as well as specifying the exact amount of offspring that is created to form a new generation are part of the selection method. There exists a wide range of selection strategies varying in complexity. A common aspect among most of them is the constant value of the population size for each generation.

There are several types of approaches that can be taken when carrying over bio inspired evolution into the digital world. The implementation that is the closest to actual biological processes uses genetic algorithms which are based on a digital DNA that encodes all instructions about behaviour, morphology etc. of the virtual organism as one or several binary strings [10]. When an individual survives by passing the fitness test, it reproduces and passes on its genetic information to its children. We simulate mutations in the DNA copying process to the offspring by changing individual bits on the binary string that carries the genetic information. The main reproduction operators used with genetic algorithms are called bit-string crossover. With this method, the DNA of two successful individuals is combined by switching out a sub sequence of bits at some random point. Another approach is the use of bit flipping where individual bits of the DNA are either flipped form 0 to 1 or vice versa at a random location. The amount of bits that are swapped depends on the specified variability. Usually, a very low variability will lead to a slow evolutionary process while a very high variability might produce monsters that have nothing in common with their ancestors and show no survivability at all. It is thus important to think about selection method and adapt it towards the problem that is being solved.

While ideal for finding well adapted individuals in an environment where there is no limit to the amount of fitness test performed, this approach is not transferable to the web domain. In its purest form, we would have to encode a web page's markup and source code as a binary string and mutate individual instructions. This scenario might work with an infinite amount of users that build up and transform the page over a large time span, but what we want is a much more targeted approach that can be used with a much lower visitor count and shorter time span.

3.3 Translating the biological evolution metaphor

The best way we have found to carry over the biological evolution process to optimise user interfaces on the web is by using what is called evolutionary programming [9]. While still similar to genetic algorithms, the structure of the evolutionary design is much more targeted at the specific problem domain. Instead of completely encoding the page's appearance as a binary string, we mutate user interface parameters of individual page elements. In terms of biological evolution, a single

materialisation of a web page is considered an individual, it's appearance is encoded as a simplified DNA that stores the element's visual parameters. This approach does not fully adhere to the principles of evolutionary computing but represents a good compromise for applying evolutionary inspired processes to the domain of online controlled experiments.

There is another classification of evolutionary algorithms that differentiates weather the reproducing organism passes on its genetic information by itself or if it is merged with another organism. Both of these variations can be observed in nature and both have their advantages and disadvantages. For the purpose of evolving web pages, we will use a single organism approach in which the most successful variation will pass down it's properties to its offspring.

When we consider a web page as being an individual organism, we have to define in which case a certain individual will be considered successful and be selected for reproduction. This means that similar to an animal being able to survive in the wild long enough for being able to reproduce, we will check weather a certain version of a web page survives. To measure the survivability of a mutation, we must first define under which conditions we consider a page to have passed the fitness test. The fitness test is similar to the notion of the success criterion which was introduced when talking about AB testing. The specifics of the fitness test will vary between web pages and depends on the intentions of the developer on which metric should be optimised.

A limiting factor that we have to consider when using evolutionary approaches for multivariate testing, is the fact that an individual (in the case of online controlled experiments a version of a page) can only try passing a fitness test if it is viewed by a visitor. In the end, it is the recorded user behaviour which will decide if a mutation fails or succeeds. Since variations between mutations can be small and the impact the evolved parameters have on the fitness test are not always obvious, a single user is not enough to determine whether a certain variation of an individual is in fact better than another one. In order to take decisions that are statistically significant results, we will need to perform multiple fitness tests per mutation until a certain cutoff value is reached. The amount of fitness tests that is required per generation is difficult to determine. It will depend on the selection method which defines the number of variables that are being evolved simultaneously as well as the variability that is used to generate new offspring.

In order to reject the null hypothesis, with classic AB testing, we would perform one of the previously discussed statistical analysis. With multivariate testing things can get even more complicated as we have to look at variable interaction as well. Using the evolutionary approach, we leave the domain of statistical hypothesis testing and utilise the iterative process for finding successful treatments. This means that we will reject the null hypothesis (which is the last generations winning mutation) in any case and generate new offspring based off a generations most successful mutation. The underlying idea is that we approach the optimal solution through the means of iterative and gradual improvements. One of the main advantages of this approach lies in its potential for automation. As we evolve user interface properties, we don't have to manually generate all possible treatments. The second benefit lies in the potential to truly find optimal solutions when the mutated variables start converging.

3.4 Possible issues

While evolutionary search algorithms seem to be a perfect fit for automating multivariate testing, we have identified several issues have that to be taken into account:

- We cannot evolve every variable. The combinatorial explosion of possible variable combinations would quickly grow and make meaningful winner selection impossible. We thus have to focus on the variables that we think have the most impact on a pages success criterion.

- Since we cannot mutate everything, the developer or usability designer will still have to make a choice about which variables should be included in the evolution process. The page may never reach it's full potential simply because the wrong visual parameters were chosen.

- Mutations require a large enough amount of fitness test before they can be compared to one another. Since a fitness test corresponds to one user opening the page, the amount of mutated variables is limited by the visitor count. Evolving twenty variables with three visitors a day does simply not offer enough significant data to take a decision about the fittest mutation.

- Another issue arises by the very nature of evolutionary algorithms. If when approaching an optimal solution the survivability of intermediate mutations drop, the evolutionary process will not be able to reach that optimum.
- The evolutionary approach is also limited to modifying numerical or discrete properties that are supported by the browser. This means that we are limited to evolving visual parameters. Comparing images or textual content is outside the reach of the evolutionary approach.
- Responsive web pages use different visual page layouts for different screen sizes. This means that the framework will have to be able to track screen sizes and run multiple evolutions for each layout. Other currently available frameworks struggle with this issue as well.

3.5 Evolutionary algorithms

Since we are applying evolutionary algorithms to multivariate testing, we were interested to see if variations to the classic evolutionary approach would lead to an overall better performance in terms of convergence rates and improving page consistency. After experimenting with several concepts derived from approaches used in AB testing, we decided to implement and evaluate the following four evolutionary variations:

3.5.1 Classic evolution

This approach implements the most basic form of evolutions as described in the previous section. Mutation selection as well as offspring generation happen on a random basis within the bounds of the specified variability.

3.5.2 Score based evolution

One issue that is common to many AB testing frameworks, is the fact that poor performing treatments are subjected to the same amount of fitness tests as successful ones before a decision is taken about their performance. With the basic evolutionary approach, when a user opens a web page that runs an evolutionary test, mutation selection from the current generation happens on a purely random basis. This means that until a winner is selected, all mutations are equally likely to be displayed to a user. This approach leads to situations in which an obviously bad treatment (not passing the fitness test multiple times) is just as likely being picked as a very successful variant.

In order to improve the pages consistency and favour successful mutations over unsuccessful ones, we have implemented a probabilistic adaptation which increases the chances of selecting a mutation based on its success rate. This change to the selection method is implemented by attributing a score value to each generated mutation. Initially, the score is set to zero, only when a user performs an action that allows the mutation to pass the fitness test, the score is incremented.

3.5.3 Trend supported evolution

When an evolutionary process is initialised, values for selected mutations can either increase or decrease. In many cases, the optimal configuration for these variables will require the user interface property to reach a specific value over several generations. With the basic evolutionary approach, variable mutation happens randomly when a new generation is spawned. But if a variable has been observed to be growing over several generations, it would be counter productive to spawn new mutations with property values that counteract this trend.

This is where the trend analysis comes in. By looking at the evolution history of a mutation, we can identify trends and use this information to bias the the mutation process in the direction of that rend. We think that the trend based approach offers several advantages compared to the basic evolution: For one, it will make the page more consistent. Generating mutations that follow the detected bias will lead to more successful variations. Further, we should be able to increase conversion time before an optimal configuration is reached.

3.5.4 Breadth first evolution

The breadth first approach attempts to counteract one of the identified issues with evolutionary testing. Based on the nature of evolutionary algorithms, which uses small iterations to reach an optimal solution, the search will fail to produce usable results if the initial and optimal value cannot be found through the means of small iterations. In nature, this compares to the reason why no known species has developed a combustion engine with moving gears. We have a good understanding of how an organ as complex as the human eye was able to evolve, but some solutions simply cannot be reached through gradual improvement. In terms of user interface evolution on the web, this means that a hypothetical button cannot grow in size if intermediate mutations are not able to pass the fitness test.

Since we are not bound by biological processes, the breadth first approach tries to overcome this issue by starting the evolutionary process with a variability that is much higher than with the other algorithms. The intended effect of this approach is to let the evolution quickly find a rough approximation of the optimal variable configuration, which is then refined in subsequent generations. This refinement can be achieved by lowering the variability with each passing generation.

4 Evolution Framework

4.1 Concept

In order to simplify deployment and testing, and to create a reusable software package, we decided to develop a prototype for an evolutionary testing library based on open sourced web technologies. The implemented testing suite allows for an easy initialisation of the evolutionary process and offers an intuitive way of specifying and restricting mutation bounds for selected user interface properties. In addition, the framework implements all of the previously described evolutionary variations and offers a simple way to observe and monitor running evolutions. The framework was also designed in a way that allows multiple tests to run simultaneously on different pages with different algorithms.

4.2 User Interface

The application is separated into two elementary views. The first view is used to select page elements that will serve as targets for the evolutionary process, specify evolution parameters and set evolution bounds for selected user interface properties. The second view is designed to provide information about a currently running evolution, offer possibilities to stop or delete running experiments as well as a monitoring tab which displays the evolution history for each generation. The testing suite also includes a simple user agent which has been implemented for testing purposes.

4.2.1 Aesthetics and usability

Figure 3: First prototype

Even if it is not the primary focus of this project, we wanted to provide developers with a simple, usable and well designed tool to initialise and monitor evolutions. The design of the user interface focused on streamlining the initialisation process as much as possible and started as a simple paper

prototype before going through several stages of improvements and being fully implemented (see fig. 3). As testing our implementation led to multiple iterations in the program development, we adhered to the principles of agile software development right from the start of the project [20]. We will look at two of the main design decisions that emerged from this process:

- We tested out several approaches that would dock the user interface to one side of the browser window when launched. Since this always ended up covering parts of the web page that potentially needed to be clicked upon in the process of initialising an evolution, we decided to create a floating window that could be freely dragged around so it would never obstruct any page elements.
- Initialising an evolution can be seen as a step by step process that starts by selecting properties to be evolved, defining a success criterion and specifying evolution parameters. Yet multiple page elements can serve as targets and they themselves define multiple properties to evolve. After designing a step by step initialisation, in which the user interface would switch through the different stages one by one, we found this approach to be rather tedious. Instead, we went for a tabbed solution that would simply stack all steps of the initialisation or monitoring process on top of one another. In addition to its simplicity, a major advantage of this design decision lies in its extensibility. New functionality can simply be added as an additional tab. Since the stack of tabs can quickly grow vertically and overflow the page bounds, we implemented collapsable tab headers that would hide and reveal information when being clicked upon.

4.2.2 Initialisation

The initialisation view is used to prepare and launch an evolution process on a page. The four stage process is launched when any a right click is detected on any element of the web page.

(a) Selecting evolution properties (b) Defining evolution bounds (c) Defining a success criterion

Figure 4: Initialisation user interface

The selection is highlighted with the help of a dashed, red border. After the click, the initialisation view appears and allows the user to specify which user interface properties he wishes to evolve. All supported properties can be accessed through the tab called "Add Properties" which reveals a search box. Once clicked, the input field enables searching evolvable user interface properties (see fig. 4a). When a property has been selected, it is added to the list of properties belonging to the selected element. There is no limitation to the number of properties that can be evolved simultaneously, but in regard to issue of combinatorial explosion, the selection of a large number of properties can have consequences on the selection method and time the framework needs to find converging values. Once a property is added, a tab is created which contains a slider that lets the user define upper and lower bounds. These bounds restrict the evolved properties from leaving a predefined range and allow users to limit the evolution to specific areas of a web page. The specification of bounds also prevents the generation of unusable values (see fig. 4b).

Once the initialisation is done, the user can specify which of the available success criteria the framework should try to optimise (see fig. 4c). As we have seen, this success criterion corresponds to an evolutionary fitness function and specifies which metric a mutation must reach in order to survive. Selecting a meaningful fitness test is a crucial step of the initialisation process and needs to be carefully thought through.

We defined and implemented a series of possible success criteria we considered most useful for being used in online controlled experiments. Of course this list could be extended to suit the needs of a broader range of web pages. In its current state, the framework supports click, click speed, hover and view duration as means of identifying successful mutations. We will quickly describe each of these success criteria and provide an usage example.

- The click option is probably the most used success criterion on AB tested web pages. When selected, the framework will evolve the specified properties in order to maximise the number of visitors that click on a specified element of the page (for example a subscription button on a blog).
- When selecting the "Click Speed" option, a mutation will be regarded as being successful, not by the amount of fitness tests it has successfully passed, but by the shortest time that users have used before clicking on the specified target element.
- Page view duration is another supported success criterion. If selected, the framework will track how long a user stays on a page and select mutations as winners that lead to the longest average page view duration. This option might be useful for pages that contain advertisements.
- Hover is another, maybe less obvious, success criterion. When picked, the framework will track how long the mouse cursor hovers above a certain area. This might be useful for elements that can be interacted with not by clicking but by hovering or for improving heat maps.
- Another success criterion which was not yet implemented but could make for a nice addition in the future is scrolling. With many companies implementing landing page type of websites, it could make sense to modify certain page elements in order to increase the amount of visitors that scroll down.

Depending on the selected fitness test, an additional input element is added to the view, letting the user specify which element should serve as the target for the success criterion. When, for example, "Click" is selected, the user will have to specify which button should be used for tracking user interaction.

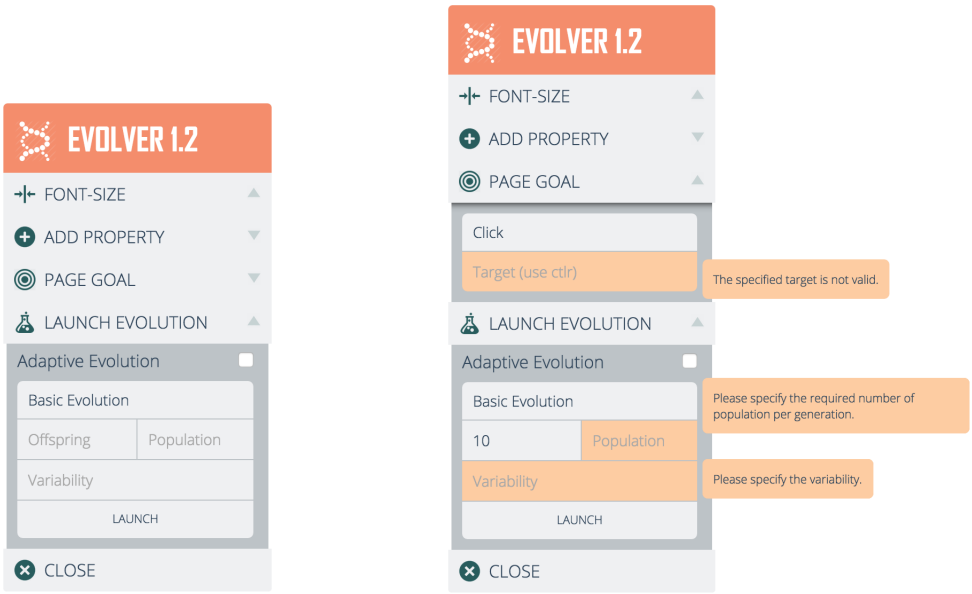
Once all properties have been set and one or multiple success criteria have been defined, the user can access the "Launch Evolution" tab. This section of the initialisation process offers the possibility to select one of the four implemented evolutionary approaches. Right below, we find the "Adaptive Evolution" toggle button which is selected by default. When unchecked, four fields appear that let the user specify evolutionary properties (selection method) such as population per generation, variability and offspring (see fig. 5a). It is recommended to use the adaptive evolution which will try to automatically adjust these parameters based on the amount of properties being evolved.

When initialising a testing process, the framework performs input validation and prompts the user to correct wrong or missing data (see fig. 9). Only when all input is valid, the framework will ask the server to initialise and run the evolution.

4.2.3 Monitoring

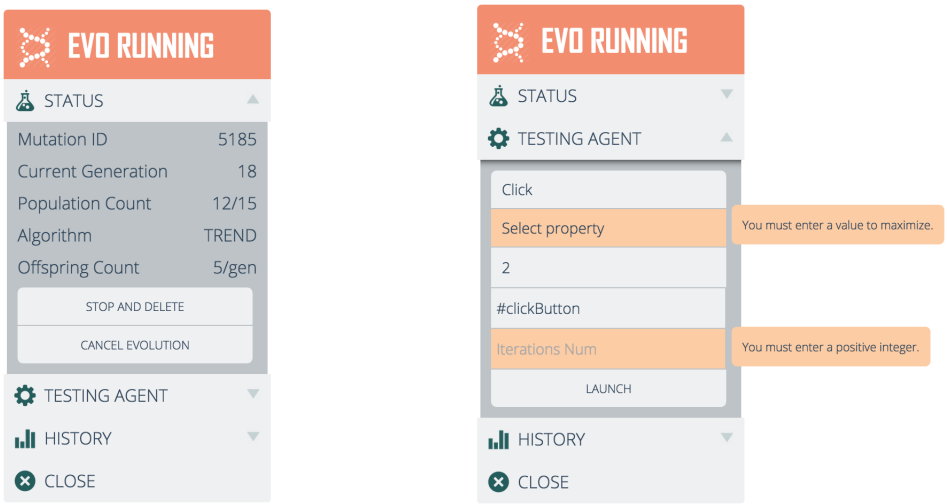
When an evolution is running and the frameworks user interface is opened, a view containing three different tabs appears. The topmost tab gives a short overview of the current state of the evolution. When expanded, it displays the currently running generation as well as status information about the mutation i.e. the number of times it has been viewed by a unique user as well as the number of times this treatment has been successful in regards to the specified fitness test (see fig. 6a).

The next tab within the monitoring tool gives access to the testing agent (see fig. 6b). This agent is used to simulate user behaviour in regards to a specified target value. It was implemented



(a) Launch an evolution (b) Input validation

Figure 5: Launching an evolution



(a) Monitoring a running evolution (b) User Agent Initialisation

Figure 6: Monitoring and agent initialisation

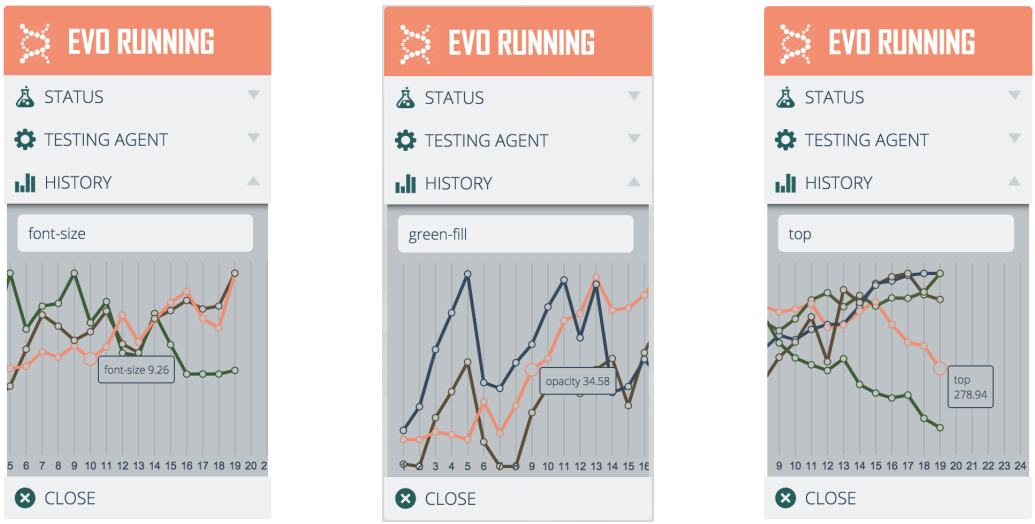


Figure 7: Monitoring different evolutions

for debugging purposes during development but can also serve for simulating user input to see whether the framework as well as the selected evolutionary approach behave as expected.

The third tab is used for more in depth monitoring and displays the complete evolution history. As can be seen in figure 7, the generations are displayed as labeled, vertical bars. The entire view is scrollable horizontally and allows us to view up to two hundred generations (if that should ever be necessary). All mutated properties are visualised as interconnected dots. The graph gives us a visual representation of all winners of all mutated properties for each generation. Additionally, hovering above a property point in the graph will display the actual value of that property as well as instantly revert the version of hovered generation, allowing us to travel back in time and view each step of the iterative process in detail.

Since some evolutions might use a large amount of evolved variables, a drop down menu above the history graph allows us to select individual mutation properties in order to inspect them in isolation. More details about loading the evolution history and the technology used to draw the graph can be found in the technical discussion of the front end.

The last tab contains two buttons which we can use to cancel or stop and delete the currently running evolution. Cancelling will simply set the active flag of the evolution to false while keeping the evolution history as well as all current mutations in tact. When a user now visits the page, the framework will return the last winner of the last generation that was fully populated. Stopping and deleting the evolution will clear all data associated with it and restore the initial page state. Once deleted, the framework can be used to create new evolutions for that page.

4.3 Client/server architecture

The implemented testing suite runs on two distinct systems that are responsible of handling different aspects of the evolution process. On one side, an apache web server handles requests, runs PHP scrips and interacts with the database by querying and writing data. On the other side the client application displays and manages the user interface and requests resources from the server. This three tier architecture of client, server and database requires us to think about distribution of concerns, security as well as the data interchange format the individual tiers use to communicate.

Figure 8 illustrates how the evolutionary process is integrated into this three tier architecture and what data is exchanged between each of them. Based on this process diagram, we will describe each step of the interaction between the client, the server and the database when a user navigates to a web page that runs a test.

- 1. The process is initiated when a user types in the URL of a page running an evolution (or uses a link toward that a page).

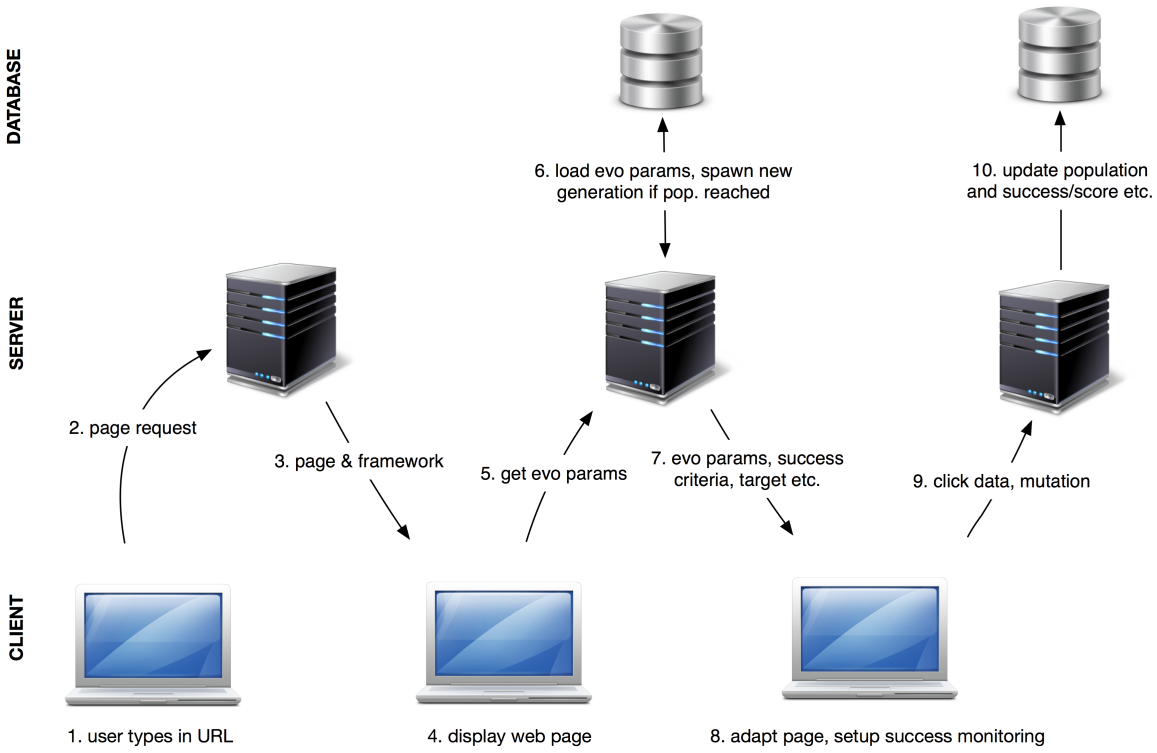


Figure 8: Client server interaction

- 2. The browser uses DNS lookup tables to find the IP address of the web server. Once a valid IP address is found, the browser requests the page from the server.
- 3. The server responds by sending the html document back to the client. The browser then traverses all file references and again requests the server to return each one of those files.
- 4. The browser has now resolved and received all required scripts, CSS files as well as other dependencies. The loaded Javascript files are executed as soon as they they are loaded. When the framework initialises, it does two things. At first, it instantly sends an asynchronous request to the server in order to check if an evolution is running for that page. It then sets up an event listener for the loaded event of the page. Once that event fires, the framework initialises and waits for further events.
- 5. The sent request contains the pages URI which the server will need to query for running evolutions.
- 6. As soon as the server gets this request, it accesses the database and queries for a running evolution as specified by the transmitted parameter. At this point, multiple paths can be taken based on the data returned by the database. If no evolution can be found, the server simply answers the clients request with an empty result. The client receives the notification and stops. The framework is still running but will not get any further data from the server.

  If the database returns a running evolution, the server will select a mutation from the current generation based on the specified algorithm of the running evolution. At this stage the server checks if the population limit of the current generation has been reached. If the sum of all mutations for the evolution surpasses the specified population count, the server goes ahead and spawns a new generation based on the winner of the current generation.
- 7. Once done, the server encodes its response containing all mutation information as well as the success criterion and all target elements into a string in JSON format and sends it back to the client.

- 8. The whole process of requesting and receiving evolution data is extremely fast and happens even before the page has been rendered on the client machines. This is important since we don't want visitors to spot visual changes to the site once it has been drawn on screen. This means that the evolution is completely invisible to a page visitor. When the front end gets the server response, it iterates though all mutation properties, traverses the page to find all elements that are involved in the evolution and adjusts their values to visualise the selected mutation. Additionally, the application also has to iterate though the specified success criterion and set in place all event listeners needed for identifying user interaction that leads to a mutation passing the fitness test. Since most of those interactions trigger page changes, it is crucial to detect and fire of all server requests before the browser navigates away from the page and garbage collects the testing library.

- 9. When a user does not perform any actions to fulfil the monitored success criterion, the frameworks job is done and it will no longer communicate with the server. All necessary information such as incrementing the population count has already been performed when the mutation was loaded. If, on the other hand, the user does click, hover or scroll on a monitored element as specified by the fitness test, the framework sends informs the server by sending it all relevant information such as which success criterion has been met as well as additional information like click time or scroll distance etc.

- 10. If the success criterion has been met, the server stores information about the user interaction into the database

4.3.1 Data interchange format

Since server (PHP) and client (Javascript) are based on different technologies, special care had to be taken to find a format suitable for transferring data between the two. While most web page URL encode their data in http requests, we had to find a format that would allow us to send more complex data structures to the server. After looking into XML initially, we decided to use JSON (Javascript object notation) for sending and receiving Javascript encoded objects. Since Javascript objects can easily be translated to JSON strings, sending and receiving data from the server is simple on the client side. On the server side, all incoming JSON have first to be translated into an object structure which the PHP script can understand,

Listing 1: JSON data send to server for evolution initialisation

```
{
   algoType:EVO
   offspring:10
   url:http://www.fryx.ch/evo/eval/bla.html
   dynamic:false
   reqPopulation:20
   target:subscribeButton
   success:speed
   variability:5
   properties:
      [
        {
           "targetElement":"subscribeButton",
           "property":"width",
           "currentValue":"200",
           "minValue":177.1,
           "maxValue":1051.3
        },
        {
           "targetElement":"subscribeButton",
           "property":"height",
           "currentValue":"11",
           "minValue":183.8,
```

```
            "maxValue":1020.9
        }
    ]
}
```

Listing 1 shows the JSON object that is being sent to server when a new evolution is initialised. Next to simple string properties, we can see that this method of communication allows us to send arrays of objects to the server without having to URL encode them. When the server receives in data in this format, it will parse the valuse and make them available in the executed PHP script.

4.4 Front end

The term "Front end" describes the part of the application that is loaded from the server and executed on the client machine. In regards to the framework, the front end is responsible for setting up and driving the user interface as well as loading and initialising evolutions by communicating with the server. In this section we will discuss the design and more technical aspects of the implemented testing suite.

4.4.1 Technology

Since the framework is intended to be used on the web and to improve web pages as well as web applications, we implemented the testing library based on technologies that make up the web today. The view uses HTML markup and CSS stylesheets to drive its appearance while all the internal logic is implemented using Javascript.

Since many operations the framework performs have to access and manipulate DOM elements, we decided to use jQuery. This library is currently one of the most used libraries on web for creating web pages and web applications. One of the main benefits of jQuery lies in its wide browser support which reduces the tedious process of finding bugs in browsers that might not support some parts of the Javascript specification. We wanted to keep the development library as light weight as possible. Since many web pages already include jQuery, no significant impact on loading times should be feared.

The only other third party library used in the project is rapahel.js. Rapahel was mostly used for simplified SVG drawing when displaying the evolution history.

4.4.2 Architecture

The front end of the evolution framework adheres to the principles of the model view controller (MVC). This structuring approach allows us to clearly separate the view which defines the user interface and it's actions, the controller which holds logic to manipulate the view, as well as the model which includes the business logic and encapsulates all communication with the back end. Because the view of the testing library is injected into the DOM tree based on the state of an evolution, some variations to the classical MVC design pattern were inevitable. Since the inner workings of the framework are rather complex, we will not traverse all of the involved classes step by step but rather discuss the workings of the individual components based on processes. Starting from the initial load, initialising the framework, setting up event listeners and so on.

The front end is composed of the following Javascript files:

- loader.js
- global.js
- editorController.js
- editorModel.js
- editorView.js
- agent.js

- cssRule.js
- maxPair.js
- mutationProperty.js
- ruleDictionary.js
- treeHelper.js

4.4.3 Initialisation

When the code has been loaded from the server, the frameworks entry point lies in the loader.js file. Here we wait for the document to be loaded and perform several crucial operations to the frameworks initialisation. First we inject the CSS file needed to correctly display the frameworks user interface into the header of the page. This prevents the user from having to include the styling rules into every html file he wishes to run the framework on. Once injected, the browser automatically loads the CSS rules and applies them as soon as the user interface is added to the DOM tree. In order to prevent CSS collisions with styling information concerning the web page on which the framework runs, all the libraries styling instructions use the "evo-" prefix when targeting DOM classes and ids.

We then initialise the MVC by instantiation the the frameworks model, view and controller. At this stage we also instantiate the user agent which is used for evaluating running evolutions. The main logic for the agent lives in another file, we will thus discuss how the agent works in more detail in a later section. After initialising the agent, other initialisation is performed for overriding jQuery CSS parsing rules which will not be further discussed here.

4.4.4 Setting up

Once the framework is initialised, the next crucial step is to check whether an evolution is actually running on that specific page or not. As soon as the model is loaded, it sends an asynchronous http request to the backend in order to load evolution data. The sent parameters simply include the sites URL. The server will check if an evolution is currently running on that address by querying the database. If it finds a running evolution, it loads a mutation from the database, otherwise the framework asserts that no evolution is currently running. The view component now sets up event listeners for keyboard strokes in order to respond to the the user launching the framework. When the key combination of ctrl + v is detected, the views event handler is called and prompts the user to type in a password. This password will be used for authenticating all future requests sent from the front end to ensure only authorised users can launch, monitor and cancel evolutions.

Listing 2: DOM tree traversal

```
TreeHelper.prototype.FindParentWithId = function(DOMElement){
    if($(DOMElement).attr('id')){
        return DOMElement;
    }
    else{
        return this.FindParentWithId($(DOMElement).parent());
    }
}
```

The user interface is now ready to launch. As soon as the user right clicks any part of the page, the view is set up by injecting a predefined html string into the sites DOM tree. This injected markup will be targeted by the previously loaded CSS file and be absolutely positioned in order to overlay the page as a draggable window. The view component now sets up all required event listeners to handle input on the generated user interface. The DOM element which was right clicked is highlighted with a dashed red border. It is important to note that only uniquely identifiable DOM elements can be user as evolution targets. This constraint is necessary as when an evolution runs, the framework needs to be able to find the DOM element to which to apply selected mutation

properties. In order to achieve this, we implemented a recursive tree search routine which performs hit testing on the clicked page location and travels up the hierarchical DOM tree in order to find an element which specifies a valid id tag (see list. 2).

The framework is now ready to add evolution properties to the selected element. By clicking the "Add Properties" Tab, a search box appears which lets the user enter the name of a property he or she wishes to evolve. The framework asks the model to return all available properties defined in the ruleDictionary data structure. The rule dictionary is easily extendable and holds all supported CSS properties that can be used in the evolution process. Next to the name, it specifies meaningful minimal and maximum values that can be selected as the evolutionary bounds. The entries in the rule dictionary are matched against the search input and displayed in the UI. By clicking on a property, two things happen:

- 1. The views event listener asks the controller to inject a property editor tab into the UI and set up a slider that can be used for visualising selected evolutionary bounds
- 2. The selected property is added to a data structure in the model. This array stores evolution property objects that hold information about the property name, the specified bounds as well as the target element to which these properties apply for.

When all properties are added to the selected element, the user can either continue with the initialisation setup or click on another page element in order to specify further properties to evolve. It is always possible to select DOM elements for which evolution properties have been added previously. When clicked, the controller will load all mutation properties having the same target as the clicked element and add it to the view for manipulation.

Listing 3: Selecting a valid id for the success target

```
            // listen for key on document, get div ID
            $(document).keydown(function(event){

                // if a box has focus, and ctrl is pressed
                if(view.targetSelectorFocus == true && event.which == 17){

                    // create a tree helper
                    var treeHelper = new TreeHelper();

                    // get the closes DOM element with ID
                    var closestDOMParentWithID =
                        treeHelper.FindParentWithId(view.lastHovered);

        // update the value in the target input field with target ID
                    $(view.focusedBox).val('#' + $(closestDOMParentWithID).attr('id'));
                }
            });
```

In a next step, the user can specify a success criterion for the page. The user interface changes based on the users selection. When "click" or "click speed" is specified as a success criterion, the view adds a second input field in which the user can enter the target elements id. For convenience, we added the option to simply click the input field, then hover the mouse over the desired target and press ctrl. The controller will again perform hit testing and find the first element in the DOM tree that contains a valid id attribute. After selection, it is automatically added to the input field (see list. 3).

The last step consists of selecting an evolutionary approach and specifying evolution parameters. The user can click a toggle button to indicate whether he would like the framework to automatically adjust these parameters or whether he wants to manually input values for the population count, offspring by generation as well as variability.

When the user finally clicks the "Launch" button, the model checks all input for it's validity before contacting the server. This security measure prevents the user from initialising bad evolutions that are prone to fail. The input validation is quite simple: it checks if all required inputs

Figure 9: Input validation errors displayed by the framework

have been given and further verifies if their type is as expected (number, string etc.). The validation also examines if the selected id specified as the success target exists within the page DOM. Figure 9 shows the user interface containing feedback based on multiple invalid input. As soon as all input has been corrected, the model sends off an asynchronous http request to the server. If authentication succeeds and the evolution initialises correctly, the page displays a confirmation message and then reloads.

4.4.5 Monitoring

When the framework initialises on a page for which an active evolution is running, a different path is selected by the framework. Again, the controller asks the model to get all evolved elements with their respective properties from the back end. Once loaded, the model informs the controller about the successful request and passes all evolutionary properties as well as information about the success criterion back to the controller.

The controller now has to perform several actions in order to modify the page and reflect the loaded mutations parameters:

- 1. The framework parses the passed JSON string from the database and populates a an array with all received evolution properties as well as their targets
- 2. It then iterates through this array, finding the corresponding elements in the DOM tree and applying the visual properties as defined by the mutations
- 3. Depending on the specified success criteria, the controller sets up event listeners and timers to catch user interaction leading to the mutation successfully passing the fitness test. For example: if the loaded evolution specifies click speed as a parameter, the controller sets up a click listener on the specified element (this could be a button or any other page area). It then initialises a timer that registers how much time has passed since the page has been initialised. This value will eventually be sent back to the server should the success criterion be reached.

Once the page has been adjusted to represent the selected mutation and the framework is able to intercept user action fulfilling the success criterion, the controller again injects the DOM with the monitoring view. Other than the initialisation view, this window contains tabs used for informing users about the evolution state.

When opening the history tab on a currently running evaluation, the frameworks controller will ask the model to perform an asynchronous call to the server and prompt for the complete evolution history of the currents page mutation. Since all mutations with all their properties are kept in the database in a structured format, we will ask the server to query the database for all winners by generation. Of course, the implementation for getting a history of all winning mutations varies based on the selected algorithm.

Listing 4: Simplified code illustrating the drawing process with raphael

```
// create the SVG canvas and attach it
var paper = Raphael($('#graph').get(0), 1000, 200);

// for every distinct property in the evolution history
for(var k = 0; k < mutProperties.length; k++){

    /* for every generation */
    for(var i = 0; i < model.evolutionHistory.length; i++){

        /* if we get the currently being drawn property */
        if(model.evolutionHistory[i].property == mutProperties[k]){

            var propMax = 0;

            // get the property max
            for(var m = 0; m < maxValues.length; m++){
                if(maxValues[m].property == mutProperties[k]){
                    propMax = maxValues[m].maxValue;
                }
            }

            // compute x position
            var x = currentGeneration * 15;

            // compute y position relative to property max
            var y = 190 - (model.evolutionHistory[i].currentValue / propMax * 180);

            // if a previous value exists, draw a line connecting the data points
            if(previousPos != null){
                var line = paper.path("M" + ((currentGeneration - 1) * 15) + "," +
                    previousPos + " L" + x + "," + y);
                line.node.setAttribute("class", model.evolutionHistory[i].property);
                line.toBack();
            }

            // draw a circle for the data point
            var circle = paper.circle(x, y, 3);
            circle.node.setAttribute("class",
                String(model.evolutionHistory[i].property));

            // bring the circle to the front
            circle.toFront();

            // setup the hover in effect
            var hoverIn = function() {
                // add highlight
            };

            // setup hover out effect
            var hoverOut = function() {
                // remove highlight
            };
```

```
            // bind the hover effects
            circle.hover(hoverIn, hoverOut);
        }
    }
}
```

Once the history is loaded, we parse the JSON string in order to draw the evolution history on screen. For easier SVG drawing, we use a library called raphael. When we look at the simplified code of the drawing process in the controller (see list: 4), we see that the rapahel canvas object is generated at the beginning. We then iterate over all evolution property winner arrays. An inner loop further iterates over every winner property by generation. Since all properties can have very different value ranges, we vertically scale the history in order to fill the height of the tab. This means that the y axis is skewed and different for all properties. Since we still want to be able to get individual values, we add hover event listeners to all of the data points and show a tooltip containing the value. At the same time we highlight the complete evolution history for better visualisation. When the mouse cursor hovers above a data point, the page is instantly changed in order to represent the winner mutation at that generation.

Listing 5: Code illustrating an user agent iteration

```
Agent.prototype.DoIteration =function(){

    // get the agent for jQuery inner
    var agent = this;

    // store new iterations count in cookie
    document.cookie = 'iterations' + '=' + this.iterations - 1;;

    // reload the current page (default link has been prevented)
    setTimeout(function() {
        location.href = $(location).attr('href');
    }, agent.actionTimeout);
};
```

The last tab contains the evolution agent which can be used to simulate user input in order to test the evolution. The agent code lives in the agent.js file and is actually initialised by the controller. Once an evolution is running, the agent can be accessed through the UI tab "Agent". It's main purpose is to simulate user behaviour based on a certain goal.

Before the agent can be launched, a series of parameters have to be set. The agent needs to know about which page element needs to be clicked in order to pass the fitness test. Further, the agent need a series of properties from which he will decide whether to perform the specified action or not. Other parameters include the iterations count, which define how many mutations the agent should evaluate as well as the target value a certain mutation parameter should reach.

The agents parameters are stored as a session cookie on the clients machine when the process is launched. Upon a page load, the agent reads the stored cookie and determines the remaining iteration count. If it is above zero, the agent will decide whether to perform a specified action or not and simply decrement the iterations counter by one, store the new value and reload the page (see list. 5).

Whether the agent performs the specified action is decided by some degree of chance and the target value the agent tries to reach. As a simple example, if we launch an evolution on the width of a button on our website. We setup the agent to reach a button with of 400 pixels and start iterating. Once launched, cookies with the specified parameters are stored on the client and the specified values are extracted. The closer the actual value of the element is to the target value, the higher the probability of the agent actually simulating a click and thus fulfilling the success criterion. Since we don't want the generated click to take us off the page or trigger any other action, the agent stops the propagation of the event before actually clicking the target. Only the code for updating the success criteria is executed. After a short delay, the page reloads and if there

are still iterations left the agent continues.

4.5 Back end

The back end refers to all the code which is managed and executed by the server. In contrast to the front end Javascript code, the back end was written in PHP and is considered a secure environment. This means that users cannot directly execute code that exists as a PHP file on the server. They only way to interact with the server side application is through http calls. The server listens for incoming requests, parses and validates data sent with the request and executes the corresponding code.

In this section we will discuss the backend architecture and illustrate some of the more interesting and challenging aspects of the code.

4.5.1 Database

In order to store store all data associated to multiple evolutions running on different pages, a server side database was needed. Initially, the framework used a simple XML database to provide a server side storage of evolution properties. With increasing complex data relations, it was decided to switch to a classic MySQL database. The consequence of this decision was a complete rewrite of the back end. The XML approach was initially chosen to reduce the frameworks dependency on external components. But after a certain development stage was reached, if was found that implementing a flat file XML database from scratch would take way to much time. Switching to a proven database architecture was the only viable solution.

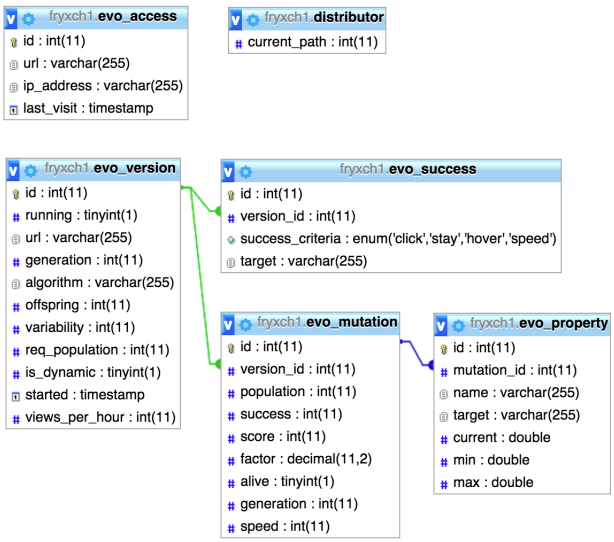


Figure 10: Database schema

At the current state of development (which is the version used for the final evaluation), the database features seven tables used to store all currently running evolutions for a a certain domain. Figure 10 illustrates the implemented database schema. We will shortly go through each table and describe its use in regard to the evolutionary testing process.

The version table is at the core of each running evolution. When an evolution is initialised, this table stores on which URL the evolution is running. It also contains all evolutionary variables which are needed to spawn new generations.

The table holds the selected algorithm used to create offspring and select mutations (EVO, SCORE, TREND and BREADTH). In addition, it stores the offspring count per generation, the variability used when spawning a new generation as well as the required population count that

determines when a generation switch happens. When the field "isDynamic" is set, the framework will initialise and adapt evolutionary parameters automatically.

The "version" table also keeps track of weather an evolution is still running. If this value is set to false, the framework will only return the most successful mutation of the last generation. The field is basically used to stop the evolutionary process and select the final version of the page after it has converged towards an optimal solution. The complete evolution history with all generations remains in storage.

The "success" table is linked to the versions table and specifies the running evolutions success criterion. The many to one relation allows an evolution to define multiple success criteria. A single success criterion is defined though a foreign key link to the "version" table, an enum of the actual criterion (speed, hover, stay, speed etc.) as well as the success target. The target is stored as a DOM elements id or class tag.

The "mutations" table is where we keep track of each single offspring. It contains a foreign key link to the "version" table. Each mutation has a population count that stores how often this mutation has been subjected to the fitness test. The "success" field contains the value for how often the mutation has passed the fitness test. Each time one of the two values if updated, the "factor" field is updated to represent the ratio of successful mutations in regards to the amount of performed fitness tests. The fields "score" and "speed" are used to store data relevant to evolutionary algorithms other than the basic evolution. A mutations score is linked to the success field and used to favour successful mutations over unsuccessful ones. The "speed" property is not directly linked to the selection process but holds an average value in case a success criterion is specified that uses a duration for identifying successful mutations. Last but not least the mutation holds data about the current generation and whether it is alive, meaning it is part of the current generation, or not.

The "property" table is where all data is stored in regards to a mutations evolved properties. The field "mutationId" links the table to its corresponding mutation and allows for a one to many relation. In order to reconstruct the page according to the specified variables, we store which page characteristic is being targeted (font-weight, colour, position etc). What the current value of that property is and within what bounds it is able to change once a new generation is spawned. The target can contain either a DOM id or class.

The last two tables are not directly linked to the evolutionary process but server for security and abuse protection. In the "access" table, we store which IP has accessed which evolution. The "distributer" table is only used for the final evaluation, ensuring an even distribution of users to all four tests.

4.5.2 Architecture

The back end consists of a series of PHP scrips that respond to http requests, receive data, interact with the database and send back a response to the client. We will quickly discuss each of the scrips and have a closer look at how the different evolutionary approaches were implemented.

- util.php
- initEvolution.php
- cancelEvolution.php
- updateSuccess.php
- getMutation.php
- getMutationHistory.php
- getMutationSuccess.php
- evoRunning.php
- rand.php

When an evolution is initialised from the front end, the application calls the initEvolution.php script through a http request, passing all required parameters in JSON format. The initialisation process starts by parsing all values passed with the request. It then goes on to create the main entry in database table evoVersion and populates the passed success criterion in it's respective table. In the next step, the script is responsible for generating the first generation. Based on the population count, it produces offspring which is stored in the evoMutation table. For each mutation, the script now mutates the specified property values based on the selected evolutionary approach and the specified variability. When successful, the script sends an acknowledgment back to the client machine.

The process is different when the user initialising the evolution has selected to perform an adaptive evolution. In this case, the back end automatically selects the evolution properties based on how many properties are being evolved simultaneously. The default variability will always be initialised at a value of five percent. With an increasing amount of properties being evolved, the framework increases the amount of offspring that is generated by each generation as well as the total population count that is required before spawning a new generation. By default, the current implementation generates five new offspring and requires twenty fitness tests to be performed before spawning a new generation. The offspring count (as well as the required population) is doubled with each additional property being evolved. This approach takes into account the fact the combinatorial growth when using multiple variables.

Listing 6: Getting a mutation for a score based evolution

```
$totalScore = 0;
     $scores = array();

        while($row = mysqli_fetch_assoc($noRandQueryResult)){
             $rowScore = $row['score'];
                $totalScore += $rowScore;
                array_push($scores, $rowScore);
            }

            $random = rand (0, $totalScore);
            $iterativeScore = 0;

            foreach ($scores as $score){
                if($random <= $iterativeScore){
                    break;
                }
                $iterativeScore += $score;
                $index++;
            }

            if($index > $mutationCount){
                $index = $mutationCount;
            }
```

The getMutation.php script is the most complex script of the back end. It's first purpose is to fetch a mutation from the database, encode the data and send it back to the client. Before the selection executes, it checks if an evolution is actually running at the specified URL. If an entry is found, the script then examines whether the clients IP address has already been recorded for this evolution instance. It then goes on to select a mutation for the passed URI of the last currently living generation. How this selection is performed depends on the type of evolution that is specified for that test.

If it is a score based evolution, the script will not randomly select a mutation but rather use the specified evolution scores to weight the probability of a mutation being picked (see list. 6). The weighting is done by ordering all mutations by their score in ascending order. The script then generate a random number between zero and the sum of all scores. Subsequently we iterate over the list of mutations by checking the random number against the mutations score value. If the

generated number is below the score, we select that mutation to be passed back to the client. This process ensures that a mutation with twice the score as another mutation is twice as likely to be picked while still allowing for the possibility of less successful mutations to be selected. Since the score values are doubled each time as score based mutation is successful, well performing mutations will be selected at a much higher rate as their score increases. All other evolutionary approaches perform a random mutation selection.

Listing 7: Selecting the winner from the current generation

```
$winner = "SELECT name, target, current, min, max, success
                FROM evo_mutation
                LEFT JOIN evo_property ON evo_mutation.id = evo_property.mutation_id
                WHERE speed = (
                    SELECT MIN( NULLIF( speed, 0 ) )
                    FROM evo_mutation
                    WHERE alive = 1
                    AND version_id = '".$versionID."')
                 AND alive = 1
                 AND version_id = '".$versionID."'
                 GROUP BY name";
```

The script then continues to increment the population count of the selected mutation and checks whether the specified population cap has been reached. Incrementing the population count only happens if the clients IP has not yet been recorded for that evolution. If the population cap is reached and the IP passes, the script will invalidate all current mutations by setting their alive value to zero and proceed to spawn a new generation. In order to generate new offspring, we first need to select the winner of the current generation. The winner selection varies based on the specified success criteria. If, for example, we select the winner of an evolution with the click speed success criteria, we select a mutation of the last last generation for which the average click speed is lowest but not equal to zero. The query (see list. 7 performs this selection by joining the evoProperty with the evoMutation table and selecting a single mutation together with all its properties. Other success criteria use different queries for selecting evolution winners.

Once a winner is picked, the script proceeds to spawn new mutations as specified by the evolutions offspring count. The process by which this is achieved takes the last generations winning mutation and iterates over all it's properties. The exact way in which the property values are mutated varies strongly based on the specified evolutionary approach. The logic being encapsulated in the util.php file, we will have a quick look at how the different evolutionary approaches perform this task:

- The basic as well as the score based evolution work the same way. Based on the winner, we simply iterate over all evolution properties and mutate them by a random value that lies within the specified variability. The process also ensures that the values stay within the specified evolution bounds.

- Before mutating the properties, the trend based approach looks at past generation winners in order to determine if the value progression can be identified as a trend (incrementing or decrementing). A trend is only detected if a mutation property shows a steady increase or decrease in it's value over at least three generations. This value was chosen in an attempt to prevent identifying false trends. If a trend is detected, the window in which a mutation property can evolve is shifted in the trend direction relative to the specified variability.

- The breadth first approach will ignore the specified value for variability and compute a variability based on the current generation. This means that in the first generation the variability will be at hundred percent. With each subsequent generation the variability will be halved.

Once all values for the new offspring is generated we increment the generation count and write the new mutations together wit their mutation properties into the database. At this stage of the evolution, we also check if the user has selected to perform an adaptive evolution. If this is the case,

we analyse the evolution history in order to determine the impact each of the mutated variables has on the success criteria. The purpose of this step is to adjust the variability as well as the offspring count of a running evolution based on how the evolved variable has been acting throughout the evolution. Since this process was not required for the evaluation (which used static evolutionary properties), we did not fully implement and test the dynamic adaptation yet. We believe that further examination is needed in order to create a solution that correctly adapts to how evolved variables behave throughout an evolutionary testing process.

Another important script file is the updateSuccess.php. This script is executed each time a mutation successfully passes the fitness test. As we've seen, setting up and detecting a successful mutation is performed at the front end. As soon as a successful mutation is identified, the front end calls the update success script passing along data that specifies which mutation has passed the fitness test. After checking the users IP address in order to prevent abuse, the script itself gets the successful mutation from the database and increments the success counter. At the same time we compute the factor value which holds the percentage of successful mutations in regard to the total number of performed fitness tests. Generating this value will later simplify querying for generation winners. If the mutation is linked to a score based evolution, the script will also duplicate the evolutions score value. Further, if the evolutions success criterion specifies a speed value, the script will update the average speed based on the current population of this mutation before storing the value.

Listing 8: Querying the evolution history from the database

```
$history = "SELECT x.id, x.version_id, x.success, x.generation
                p.current, p.target, p.name
               FROM (SELECT id, version_id, success, generation
                FROM evo_mutation
                       ORDER BY evo_mutation.generation, factor ASC) x
                       LEFT JOIN evo_version v ON v.id = x.version_id
                LEFT JOIN evo_property p ON p.mutation_id = x.id
                WHERE v.running = 1 AND v.url = '".$url."'
                GROUP BY x.generation, p.name";
```

The last script file we will look at is getEvolutionHistory.php. This file is called by the font end as soon as the history tab is opened. Depending on the specified success criteria, the script queries the database for the complete evolution history (see list. 8). The query boils down to joining the tables evoVersion, evoMutation and evoProperty in order to select the all mutation from all generations which were the most successful based on a certain metric.

4.6 Deployment

In this section, we will have a closer look at how the framework can be deployed and go though a simple example in order to illustrate the process.

4.6.1 Setting up

Listing 9: Deploying the framework

```
<head>
   <!-- include default theme -->
   <link rel="stylesheet" href="http://www.fryx.ch/evo/evolutron-golden.css"/>

   <!-- include jQuery -->
    <script src="http://code.jquery.com/jquery-latest.min.js"> </script>
      <script src="http://code.jquery.com/ui/1.10.3/jquery-ui.js"></script>

   <!-- include framework -->
   <script src="http://www.fryx.ch/evo/evolutron-min.js"></script>
</head>
```

Setting up the framework on any running web page is a quick and easy process. The front end part of the framework comes as a single minified Javascript file called evolutron-min.js. It can either be downloaded or simply added as a script reference on pages wanting to run an evolution (see list. 9). Since we all like things to be nice and shiny, the framework provides different themes for the user interface. The library will automatically inject the default CSS file reference when loading. In order to load other themes, we need to add a reference to one of the CSS files that is available from the same directory. Since the testing suite requires jQuery to run, the library has to be included above the minified Javascript file.

Once the modified html file is uploaded, the deployment process is completed. Once the browser loads the modified html page, the server will load and initialise the framework. There is no need for setting up a database and loading PHP files on the server. When running, the framework will make cross domain calls to a central Apache server running on www.fryx.ch. While setting up the back end locally is entirely possible, we chose this option to allow for a much easier deployment.

Currently it is not possible to set up a user account with a separate password on the main server. This means that any installation will need to use the main password to run evolutions. We decided not to implement such a feature, since it is now know yet if the framework will be made available to a broader audience. For testing purposes the current solution works fine.

The full, unminified code of the framework which contains both the front and backend logic is available at http://www.fryx.ch/evo/evolutron-full.zip.

4.6.2 Running an evolution

Listing 10: Adding an id tag to the page header

```
<header id="page-header">
   <div>
      ....
   </div>
</header>
```

At this point we have looked at the user interface of the the testing library and discussed the inner workings of the back as well as the front end. We will now look through simple example on how to set up an evolution based on colours (similar to the what Google has done for finding the best shade of blue). Imagine a simple web page where we want to increase the number of visitors clicking a link based on the colour of the web pages header. Once the framework is loaded, we can find our header element in the html code and add an id tag to uniquely identify it as shown in list. 10.

Once the tag is created, we open the page in the browser and activate the framework by pressing ctrl + v. After entering the password, we can now right click on the header element which is subsequently marked with a red border. The framework UI opens and displays the initialisation view. Lets imagine that we are quite happy with the scale and position of our blue header, but want to tweak which shade of blue might be best suited for our page. This means that we only want to evolve the red and green components of our header colour.

In order to this, we click on "Add Properties" and type in "red". We click on the only search result which is automatically added to the list of evolved properties. We do the same thing for "green". Since we don't want to impose any bounds on the range of the two variables, we draw the sliders out to both ends to 0 and 255. We then click on the next tab in the UI and specify "click" as the evolutions success criterion, selecting the link as the target. We select the basic evolutionary algorithm with a dynamic selection strategy and launch the evolution.

When the page refreshes, the browser loads a mutation from the server and shows that colour variation. The framework will stick to our initial blue value for the header background and only evolve the specified properties. When we open the monitoring view, we can see that we have loaded a random mutation from the first generation. The history tab is still empty as no user data has been recorded yet.

4.7 Security

Since the framework had to be field tested with a large user base, paying attention to security concerns was crucial and had to bee taken into account early in the development process. As the implemented testing library changes the way web pages look and can even modify content, we have to make sure the system cannot be abused. There are two aspects that require protection.

4.7.1 Password protection

The first security concern is unauthorised initialisation and cancelation of evolutions. Since the framework is loaded by every client accessing the web page, all users could potentially trigger calls to initialise an evolution. In order to prevent such behaviour, we added a password protection.

Before the framework is loaded, a prompt asks for a password that is only stored on the server side and thus and unaccessible for a page visitor. The provided password is now added to every backend call made by the front-end. The server only performs the required actions if the transmitted password is correct. This measure prevents unauthorised users from maliciously initialising, stoping and deleting evolutions.

4.7.2 Multiple access abuse

Another vulnerability of the framework becomes evident when an evolution is running. Since the framework utilises user interaction to modify page content, it is imaginable that a user, once knowing that such a mechanism is in place, would try to influence the evolution by repeatedly sending malicious input. Someone could for example snoop the http calls made by the framework and, using an automated system, send hundreds of calls to the backend and thus spawning new generations and modifying the content to his liking. To prevent such attacks, the server logs every interaction by a client and stores it inside an access table together with his IP address and the page the user sent the request from. Subsequent calls that already have an entry in the access table are simply ignored. A nice side effect of this approach is the fact that page interaction generated by a robot can only influence the testing process with the first request.

5 Evaluation

Designing a framework based on a novel idea is one thing, but without testing the application in the field, not much can be said about its actual usefulness. That is why we decided to design a small test setup in which the evolutionary testing library would be used to tackle a series of optimisation problems. We also wanted to see how the four implemented evolutionary variations would perform when compared to one another. The aim of the evaluation was to answer the following set of questions:

- Is the framework able to improve the success rate of a specified metric with an increasing number of generations?
- Is the testing suite able to converge on an optimal solution??
- Which of the four implemented evolutionary approaches offers the best performance in terms of generations needed to find optimal (converging) values?
- Which type of approach is better suited for what test scenario and why?
- Can evolutionary testing supplement or even replace traditional online controlled experiments?

5.1 Methodology

The testing setup is was separated into two separate stages. At first, we wanted to find out if the evolutionary algorithms worked inside a controlled environment. For this purpose, we utilised the implemented user agent which is able to interact with a website based on a certain target condition. The seconds test setup consisted in field testing the framework with hundreds of users in a series of real world scenarios.

5.2 User agent

The user agent was heavily used during development for testing purposed and for finding and fixing bugs. With it's help we were able to quickly generate a large number of mutations and track weather the evolutionary process was working as intended. Without the agent, we would have had to manually click through dozens of tests. It was especially useful for validating the trend analysis and other aspects of the evolutionary process. With its help we were able to make sure the created algorithms were working as intended and able to converge.

In its current state, the testing agent is still quite simple. With further development it could be extended in order to test more complex scenarios and support different success criteria.

5.3 Real word evaluation

Field testing the framework proved to be a challenging task. The whole idea of automated testing requires having a large user base to gather input from - the evolutionary approach needing even more users to produce meaningful results. We considered two possible solutions for generating a large number of page views:

The first option consisted in include the framework into a sub site of the universities web page and trying to improve a certain aspect of the page based on a success criteria. After discussing this idea with the web master, we quickly decided to pursue other options. Not only would it have been difficult to find isolated pages where we could have deployed the framework, but many sub pages of the university do not generate enough traffic to perform meaningful experiments. Not an ideal setup for performing the evaluation within the limited time frame of this project. Another negative aspect of performing the evaluation on a university page was the issue of having no direct control over the tests. Our goal was not only to test the framework in a real live scenario but to compare different evolutionary approaches against one another in a series of distinct tests scenarios.

This is why we decided to set up our own testing website and generate traffic through the means of scientific mailing lists. The first list we used is called CHI-WEB. Through this distributer we were able to reach 1897 recipients. After some discussion with the administration of the university, we were also able to send the participation request to the distribution system of the University of Fribourg. The exact number of recipients was not disclosed, but we estimate that the email was sent to roughly ten thousand people.

5.3.1 Testing the evaluation process

Before starting the evaluation by sending out emails through the provided mailing lists, we wanted to make sure that the test setup was bullet proof in terms of security, ran all algorithms as intended, correctly logged all results into the database and was able to handle a large number of concurrent server requests. For this purpose, all members of the department of computer science at the University of Fribourg were asked to participate. We got positive feedback and decided to launch the final evaluation a few days later.

A particularity when developing applications for the web consist in the multitude of rendering engines as well as browsers and screen sized that could be used to view and interact with a page. We took great care to ensure that the framework executes correctly on all modern browsers such as Firefox, Chrome, Opera and Internet Explorer. Foregoing such testing could have resulted in successful mutations not passing a fitness test, simply because it was not correctly displayed on a client machines, an unacceptable situation considering the small population count used in the evaluation. The pre test helped us identify such flaws and correct them before running the final evaluation.

Performing a pre test was certainly worth the effort. We were able to track and solve several issues concerning spawning new mutations in later generations and improve security. As an additional security measure, we set in place an automated, daily database backup. Since all data related to a running evolution as well as its history is stored in the database, in case anything would go wrong, we could simply load the last backup and all tests would continue running as intended.

5.3.2 Test setup

While the testing framework is capable of initialising and modifying evolutionary parameters such as variability and offspring count based on certain parameters, it was decided to deactivate this automatic adaptation throughout the evaluation process. The main reason or this decision being the way users will access the testing page: While a normal web page has a rather steady stream of users with a few peaks, the testing page will most likely be accessed by a large amount of people over a short amount of time as soon as the emails are sent out to the mailing lists.

This leads us to one of the main problem we faced when setting up the tests: estimating the number of users we would get to participate. Since the evaluation has to be performed in a limited time frame, the amount of participants has a large impact on the evolutionary selection strategy as well as the number of variables we are be able to evolve in each test. Since we will be running four evaluations in parallel, this means that the actual number of users per test is reduced even further. Based on similar requests that were sent out for inquiries by colleagues in the past, we estimate a five percent participation rate which would result in about 650 page views. Divided by four this would leave us with less about 160 users per test, not that great of a number to perform evolutionary testing over several generatioins.

The specification of the evolution variables was the hardest part to set up. We had to make an estimation of the optimal variability, offspring and population count based on the type of variables used in the evolution process as well as the projected visitor count. All these considerations led us to the following selection strategy:

We set up the tests to use a variation rate of ten percent. This means that each mutated variable is able to diverge up to a tenth of the specified range in each generation. Based on this variability, in the optimal case, it would take ten generations for a variable to evolve from one to the other end of the specified range. Since the total amount of evolved variables will be low, we select an offspring count of five which is large enough to cover all possible variable combinations. In order for the framework to generate ten generations, we have to specify a population count of fifteen. As every treatment can only be tested three times on average, we were eager to see if the evolutionary strategy would still be able to find a solution to the specified optimisation problem.

In order to test the framework under field conditions, a simple website was set up consisting of four individual test scenarios. It was first envisioned to set up a series of complex tests with a large amount of mutated variables. Since one important aspect of the evaluation was to compare different algorithms against one another and thus having to split page visitors into different groups, we decided to simplify the tests and focus on a handful of variables.

When setting up these tests, the developed user interface for initialising and monitoring running evolutions came very handy. Test were easy to initialise, maintain and monitor.

To ensure good results, a series of precautions hat to be taken to prevent abuse. The worst case scenario would be a user refreshing a page over and over again in order to abuse the evolutionary process and modify the site to his liking. The previously discussed security measures should prevent such an abuse. If a user has worked through a test and uses the back button of his browser, the server loads mutations just as before but does not update the population count of said mutation.

Since we are running four different tests in parallel, we also have to prevent a user from accessing all four test and only interact with the algorithms attributed to him. That is why each test runs on a page that is identified by a series of random characters, this ensures that a user cannot manually change the URL and access a test not attributed to him.

Once a user opens the testing web page, the server is asked which algorithm the user is attributed to. The server balances the distribution and sends back the link to where the test is actually running. The user is then redirected to the initial test page and will pass all tests with the algorithm attributed to him.

5.3.3 Test pages

On the first test page (Test A), users are shown a very brief introduction to the testing procedure and a button prompting them to start the evaluation (see. fig. 11). At this stage, the framework is already at work, trying to maximise the amount of users that click on the button to participate. This is really the paragon of AB testing and very close to a possible real world application of the

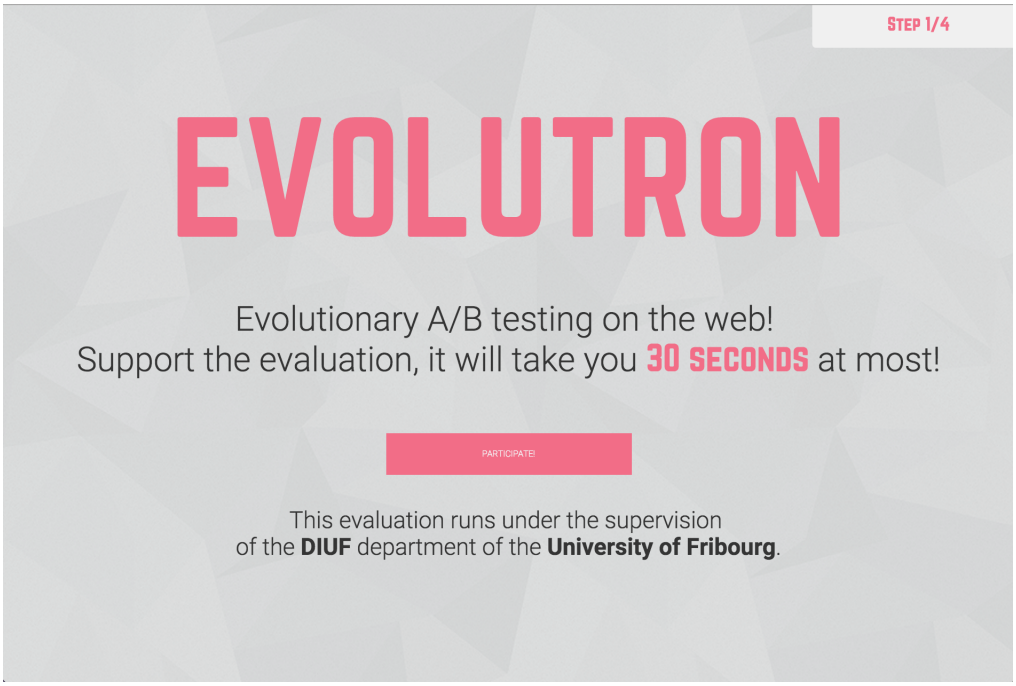


Figure 11: Participation test scenario (test A)

testing suite. The web page is designed with a clear goal in mind, which is for the user to click on the button to participate. We were interested to see if variations of ordinary properties such as font size and border radius of the participation button would change user behaviour and improve the pages average success rate. The evolutions bounds for the text scale were set between seven and seventy pixels, the border radius was able to vary between zero and sixty pixels.

With a larger user base, it would have been possible to add more variables such as font width, positioning and colour of the button or modify other parts of the site.

The second test (Test B) consists of a page with a single button labeled "Click me fast" (see. fig. 12). Instead of just measuring the success rate of users clicking the button, we were interested in minimising the time a user takes before actually clicking the button. The properties which are evolved in this test are: the left offset ranging from 0 to 1200 pixels and the top offset ranging from 0 to 800 pixels. We were also interested to see which background colour would lead to the fastest clicks. To achieve this, we also selected the RGB values for the button background (ranging from 0 to 255) to be part of the testing process.

Contrary to the first test setup, we can guess the expected, optimal value of at least two of the variables in play. The button can for sure be clicked the fastest, when it is positioned at a similar location as the participation button from the first test. This will reduce the time a user needs to move his cursor to the button in order to perform the click [5].It will be interesting to see if the best position is actually at the exact same location as the participation button on the first test or if it is simpler to identify and click the button if its position is slightly off. The colour is expected to change from this initial grey background to a colour featuring a strong contrast in regard to the page background. A bright colour should reduce the time a user needs to visually find the button and read its label.

The main goal of this test is to see if all four evolutionary approaches converge towards a single solution, if that solution is identical and what performance differences we are able to observe. The test also serves for validating the framework as a whole and show that the evolutionary approach does in fact work.

The third test (Test C) setup introduces more complexity and aims at optimising text legibility for information scanning (see. fig. 13). When reaching the third test, users are presented with the following text snippet:

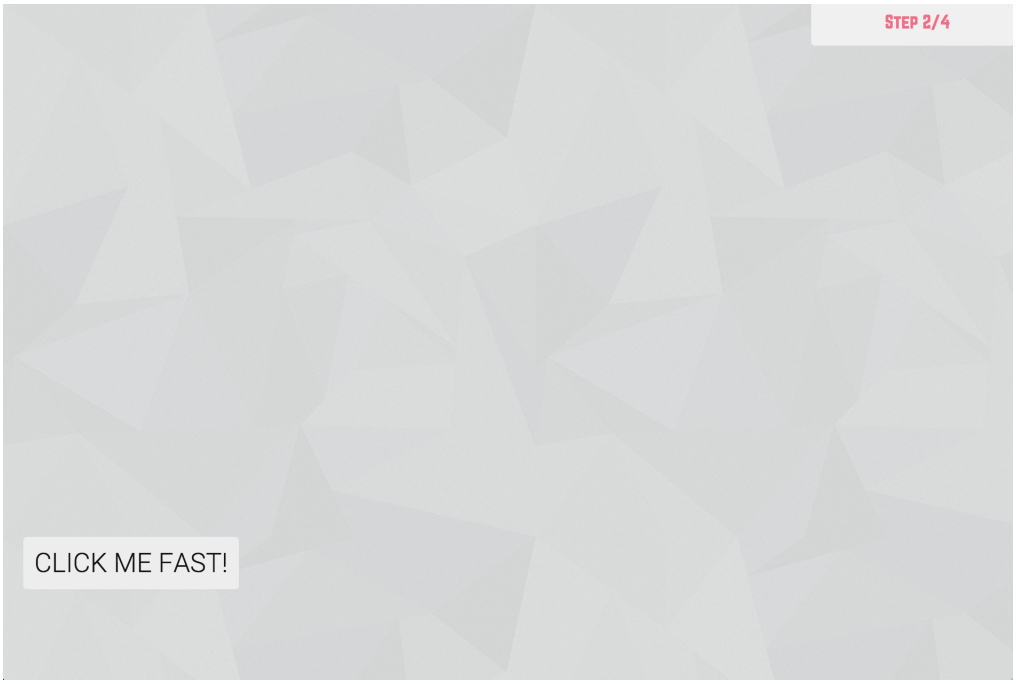


Figure 12: Click speed test scenario (test B)

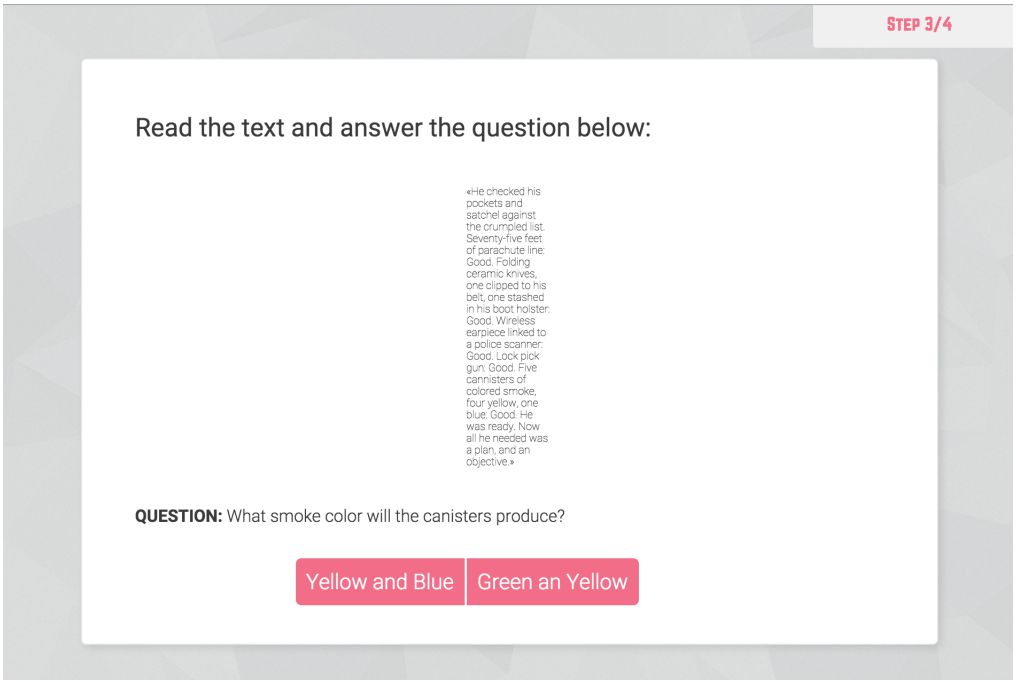


Figure 13: Text scanning speed test scenario (test C)

> He checked his pockets and satchel against the crumpled list. Seventy-five feet of parachute line: Good. Folding ceramic knives, one clipped to his belt, one stashed in his boot holster: Good. Wireless earpiece linked to a police scanner: Good. Lock pick gun: Good. Five canisters of coloured smoke, four yellow, one blue: Good. He was ready. Now all he needed was a plan, and an objective.

The user is then asked which smoke colour the five canisters contain. In order to answer the question, the user has to go back to the text and scan for the context in which the answer appears and identify the correct statement. With this test setup, we again try to minimise the time a user takes to correctly answer the question. The variables in play are the texts font size, paragraph width as well as font weight. We are interested to find out what variable combination is best suited for fast text scanning and weather the testing suite is able to even find an optimal solution.

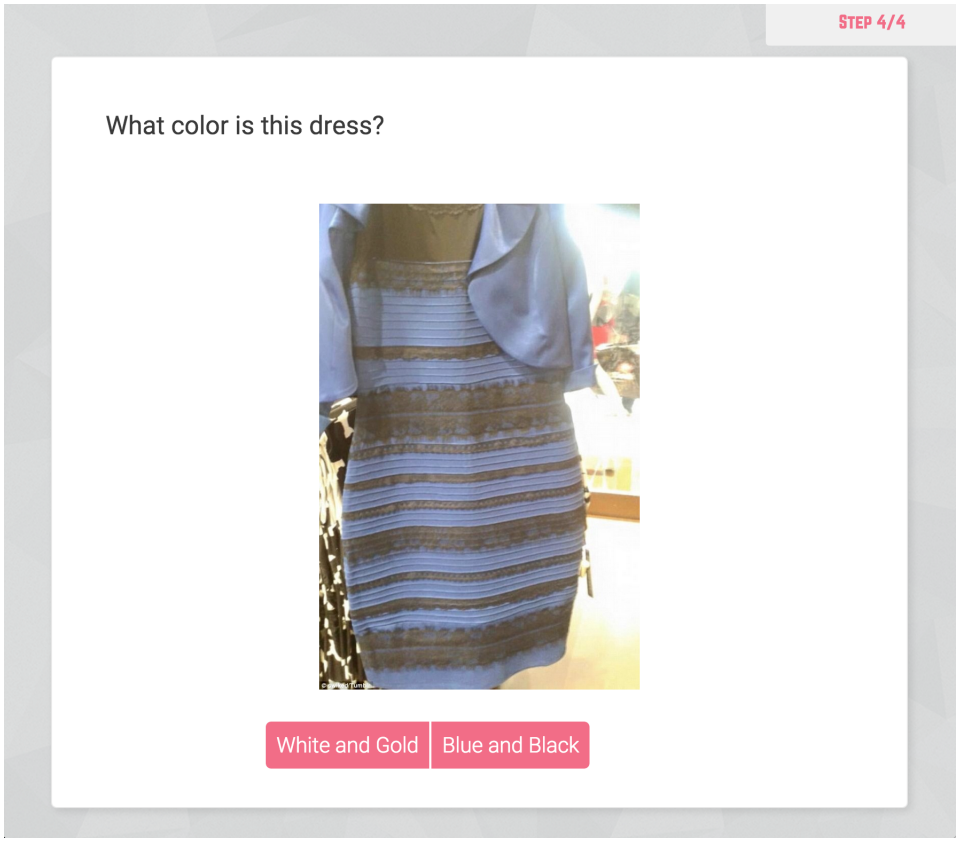


Figure 14: Color illusion test scenario (test D)

The last test (Test D) might be the most interesting one because it occupies itself with a visual colour illusion. The page shows the famous picture of the blue/black white/gold dress illusion (see. fig. 14). The photograph depicts a dress that, for some people, appears in dark blue shade with black stripes while others perceive the dress as being white and having golden stripes. The illusion boils down to how our brains perceive colour and whether the observers visual cortex interprets a blue dress as being lit by orange, evening sunlight or a white dress being covered by a pale blue shadow. We were interested to find out if we could modify the colour of the dress in such a way that all of the users would perceive it as being black and blue. Polls have shown that about forty percent of people see a white/gold coloured dress (the real dress colour being in fact blue and black).

We set up this test by overlaying the image with an SVG path of the same shape and size as the dress. We can then target this overlay with the testing suite to vary the opacity as well as the red and blue parts of the RBG colour spectrum. Initially, the dress appears in its original colour, mutating the described variables allows the dress colour as well as the colour intensity to change.

The goal of this test lies in demonstrating that the evolutionary testing suite is flexible enough for being applied to more exotic optimisation problems.

5.3.4 Evaluation process

6 Results

6.1 Real world evaluation

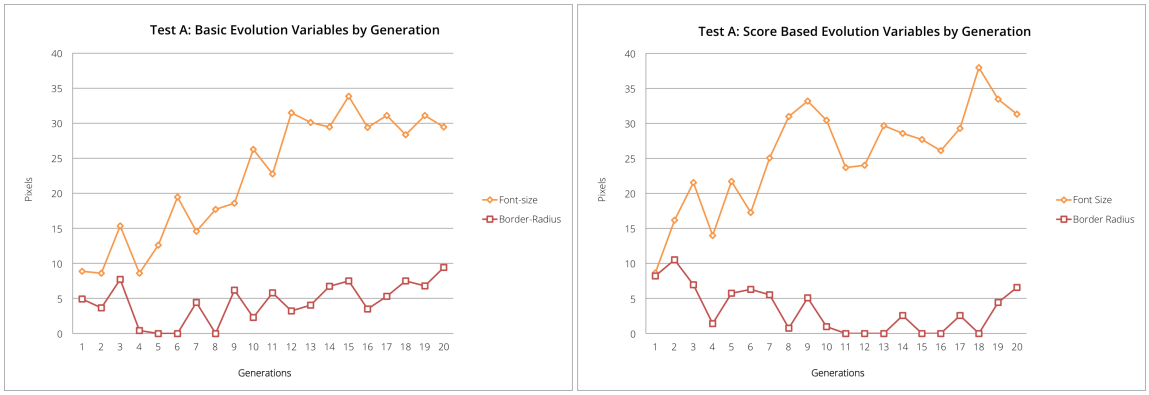
The evaluation ran for about a a week. With more than 1200 page visits, the number was almost the double of our initial estimation. This meant that instead of ten, the framework was able to generate up to twenty generations. In retrospect it would probably have been better to increase the population count per generation in order to select more significant winner mutations.

The data analysis focussed on answering the previously stated questions about the frameworks functionality as well as the performance of the four used evolutionary approaches.

For this purpose, we look for converging variable values and compare the number of generations the four evolutionary approaches need to reach that solution. We define a convergence as a variable not exceeding a data range corresponding to it's specified variability for several generations. This means that if the font size of some text is allowed to evolve from six to eighty pixels, the is considered to converge if it does not change by more than 7.4 pixels.

We are also interested by the development of average success rates as well as click speeds. If the evolutionary approach is working, we should be able to observe a steady increase of success rates throughout the testing process. Likewise, the click times should decrease as the framework find more optimal variable configurations with each generation.

6.1.1 Participation



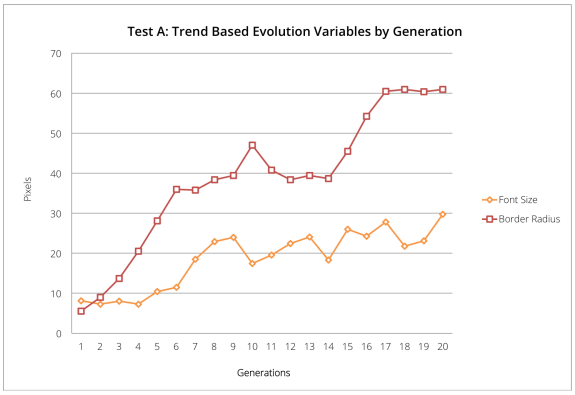
(a) Basic evolution

(b) Score evolution

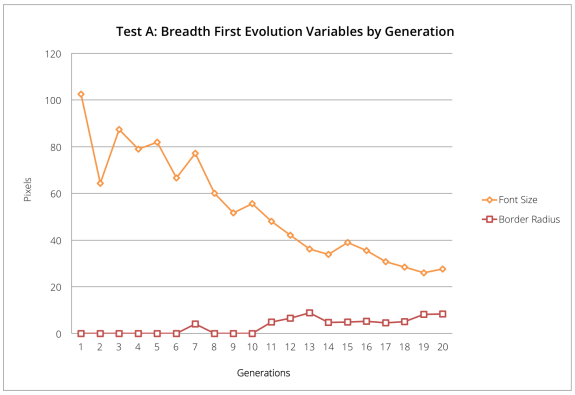
Figure 15: Test A variable evolution (basic/score)

As for all test scenarios, we will quickly go through each of the four used algorithms, discuss the gathered data and compare the different approaches.

- With the basic evolution, the font size continuously grew at an average rate of three pixels per generations and started to converge at generation ten, when values remained stable at around 28px (see fig. 15a). The border radius shows a slight tendency of growth and converges around a value of five to ten pixels. When looking at the average success rate per generation (see fig. 17a), we see that the amount of treatments passing the fitness test was very high right from the beginning and converged around a hundred percent after generation eight.
- With the score based evolution, font size quickly grew from 7 to 24px in five generations. After peaking around generation ten it converged around 30px in size (see fig 15b). We can observe

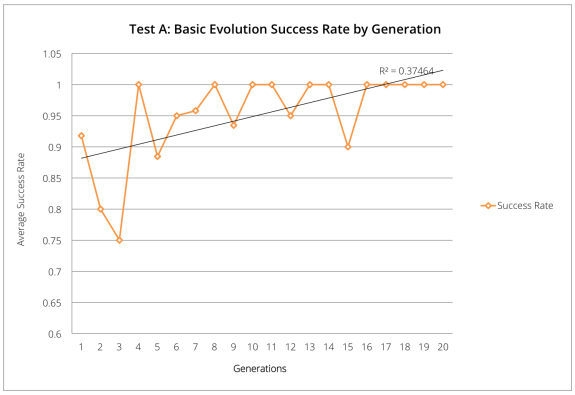


(a) Trend evolution

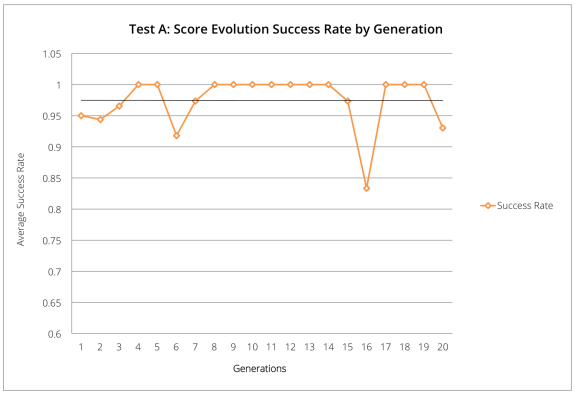


(b) Breath first evolution

Figure 16: Test A variable evolution (trend/breadth)

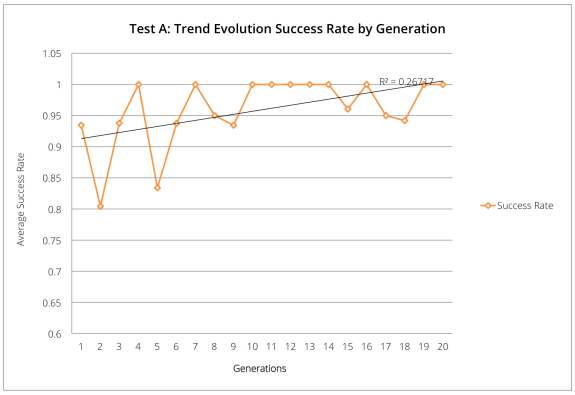


(a) Basic evolution

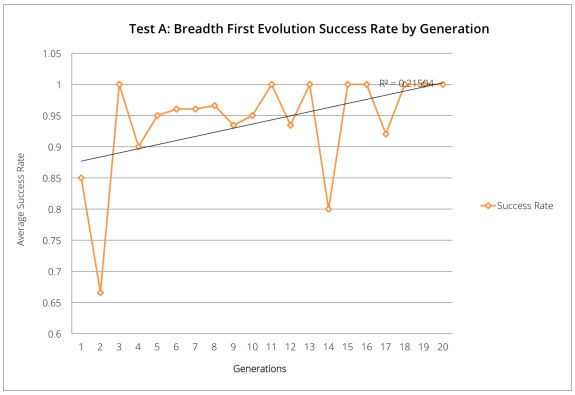


(b) Score evolution

Figure 17: Test A success rate (basic/score)



(a) Trend evolution



(b) Breath first evolution

Figure 18: Test A success rate (trend/breadth)

that compared to the basic evolution, values for font size are a lot less stable. The boarder radius shows no conversion, dropping to zero after generation ten and growing slightly towards the end. Again, the average success rate was very high, barely leaving the maximum value after generation eight (see fig. 17b).

- The trend based evolution shows similar results. At generation eight a value of 25px pixels in font-size is reached (see fig. 16a). From there on the values don't vary by more than the evolutions specified variability. Interestingly, with the trend based approach, the border-radius of the button continuously grows until reaching the evolution bound at 60 pixels at generation seventeen. The average success rates for the trend based approach show similar values, reaching a hundred percent around generation nine (see fig. 18a).

- The values for the breadth first evolution are quite interesting. We can see that in the first two generations, the font size quickly increased to over eighty pixels (see fig. 16b). When we look at the values in the database, we see that all generated mutations where attributed a very large font size. This value was certainly not optimal and started to continuously decrease from there on, reaching similar values as the other algorithms by converging at a font size of about 28 pixels. The values for the boarder radius stayed put at the beginning and started to increase slightly after generation ten. The breadth first approach needed more iterations to reach similar success rates as the other algorithms, only converging towards generation fifteen (see fig. 18b).

The extremely high success rate certainly had a negative impact on the frameworks performance in the first test scenario, yet the data shows a clear bias towards a larger font size throughout all four evolutionary approaches. The test indicates that the optimal font size in order to reach the pages success criterion lies at around 30px. We can be confident about that value when we look at the breadth first approach, in which an initially large font size continuously decreased to reach roughly the same value as the other algorithms.

Things are not as clear cut when we look at the border radius. It seems as changes in this property have a significantly smaller impact on the success criterion as the font size. This assertion is enforced by the fact that the trend based approach is the only algorithm that produced a border radius that strongly diverged from the initial value. It appears as the values randomly formed a trend in the initial stages of the evolution. The trend was picked up by the framework and since no negative impact was detected in terms of the success criterion, the values continuously increased before reaching the upper bound of sixty pixels.

When we look at the values for the basic and score based evolution, values for the border radius only started to increase once the font size had reached a convergence. It is thus possible that with a larger sample size per generation, the evolution would have further increased those values as small variations would have become more significant.

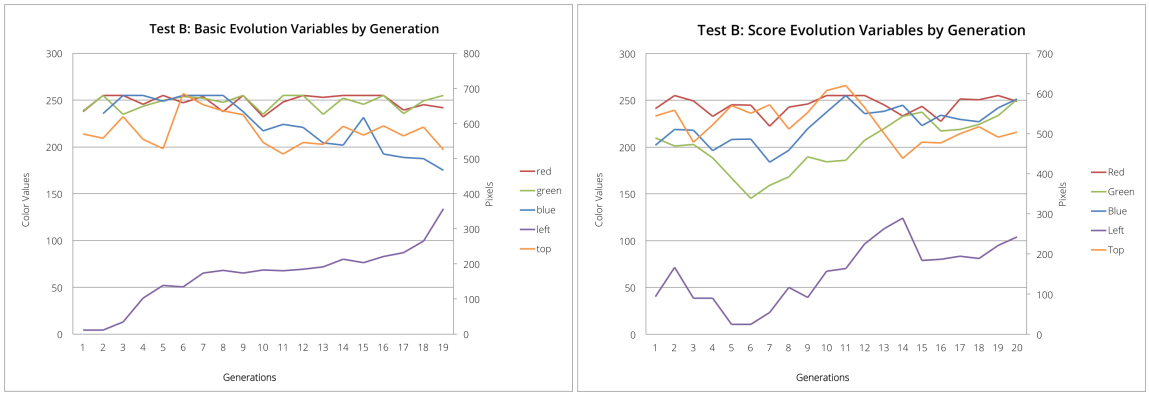
One issue we were able to observe occurred when the success rate of mutations approached a hundred percent. When all generated mutations are equally successful, the framework has to randomly chose one of the mutations as a winner. This can leads to values randomly drifting into a direction without any data justifying the change.

Overall, it is difficult to compare the algorithms performance due to the high success rates. We can say that the score based approach performed really well, as it quickly reached a converging font size. Bad mutations were quickly eliminated based on their low score. The poorest performance was observed with the breadth first algorithm. It took almost sixteen generations for the font size to converge. This behaviour clearly accentuates one of the flaws of the breadth first approach. When a low amount of offspring is generated, it is possible that none of the initial mutations is close to the optimal value. Due to decreasing variability, it can then take many generations for the property to converge.

In retrospect, it might have been better to perform this test on the last page where visitors were able to leave their email address to subscribe to the test results. With the latter test setup, success rates would have been much lower (we got about 150 requests) and the framework would have had better evidence for selecting winning mutations.

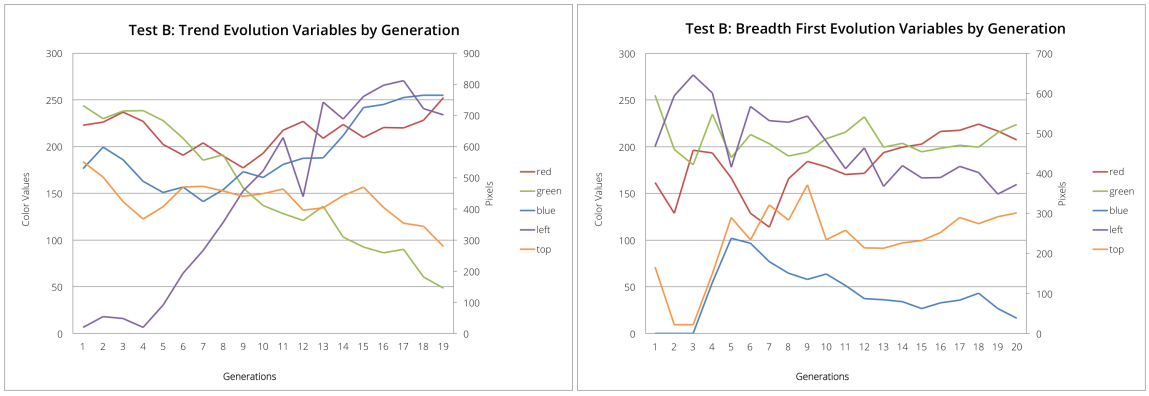
6.1.2 Speed click

Since the second test did not aim at increasing a mutations click rate but it's average click speed, the results from this test can give us more insight into the working of the algorithms. In contrast to the first test, we know what variable combination is needed to achieve the best performance. We expect the button to end up close to the centre of the screen, having a bright colour that contrasts the white page background.



(a) Basic evolution (b) Score evolution

Figure 19: Test B variable evolution (basic/score)



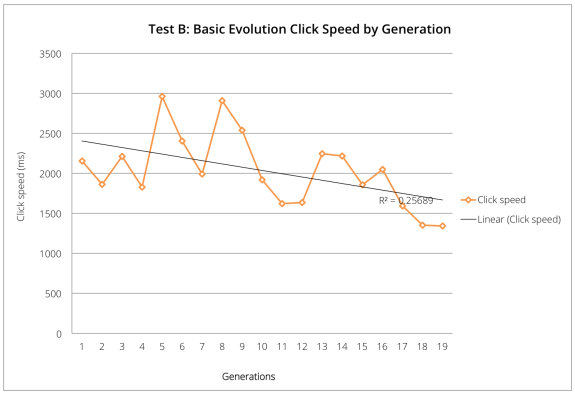
(a) Trend evolution (b) Breath first evolution

Figure 20: Test B variable evolution (trend/breadth)

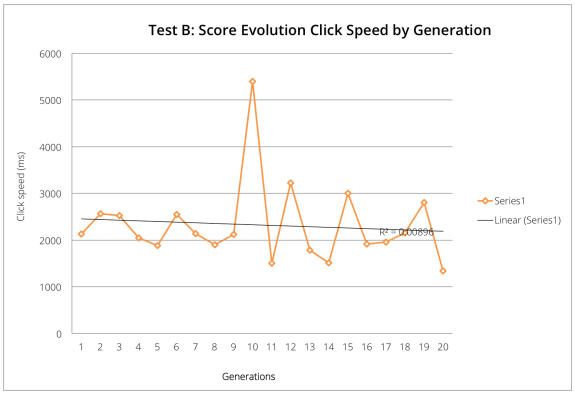
Again, we will compare the data generated by the four different approaches:

- The recorded data for the basic evolution shows a value progression that is very close to the expected result. Throughout all generations the button is moving towards the centre of the screen, starting on the left screen edge and ending up at an offset of around 300 pixels (see fig. 19a). We can see that the transition happened at a rather slow pace, reaching the final value only at generation twenty. The top offset did not change as much and modified the buttons position by about seventy pixels. The colour values remained on a constant level throughout the first ten generations. Only as the button approached its final position, the background colour started to change from a pale to a bright yellow. This behaviour indicates that the colour variables are subordinate to the positioning and only come into play once the button approaches a position with converging values. Positioning is simply much more important in terms of the success criterion than variations in colour.

  The evolution of the click speeds shows that the improvements in click time did no occur as uniformly as expected (see fig. 21a). This might be linked to the fact that with the limited

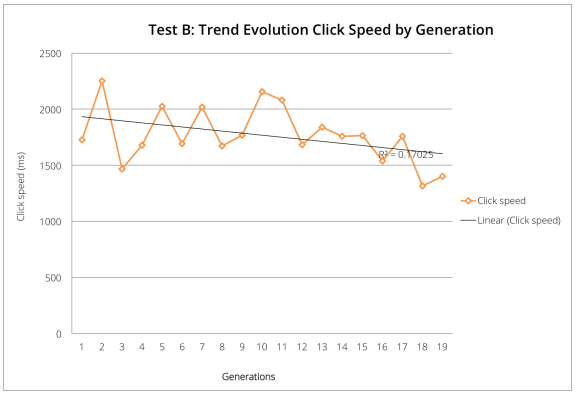


(a) Basic evolution

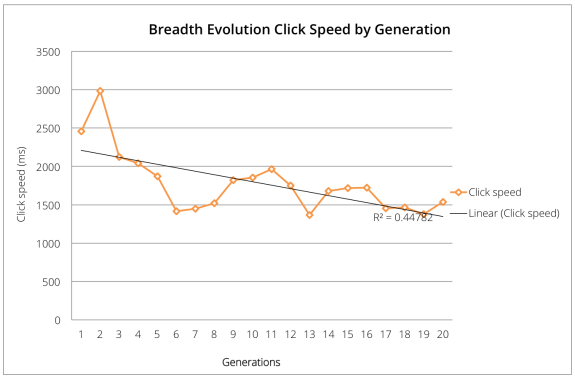


(b) Score evolution

Figure 21: Test B click speeds (basic/score)



(a) Trend evolution



(b) Breath first evolution

Figure 22: Test B click speed (trend/breadth)

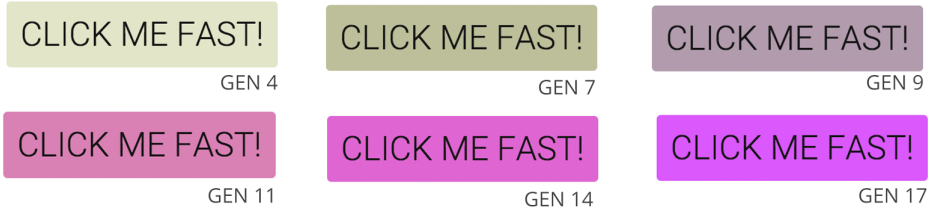


Figure 23: Trend based colour evolution

amount of fitness tests that were performed in each generation, individual user performance (outliers) could have a strong impact on the average click time of a certain mutation. The framework was still able to reduce the amount of time a user needed to click on the button from an initial 2500 ms to less than 1500 ms towards the end of the evolution.

- The score based evolution was very slow in moving the button towards the screen centre (see fig. 19b). When we look at the evolution of the left button offset, we see that in the first few generations, the button actually moved into the wrong direction. The same behaviour can be seen towards the end of generation fourteen.

  The top offset stays more or less at a constant value but also moves down instead of up at some points. The reason for this poor performance can be linked to the fact, that more than 95 percent of all button variations were clicked upon and thus passed the fitness test. This allowed potentially slower mutations to accumulate a lot of points, preventing other mutations from being selected.

  As for the buttons colour, we see a slight drop of the blue component at the beginning of the evolution but the values never converge. The click speeds reflect the bad performance of the algorithm. The average time a user takes before clicking the button only decreases by a very small amount throughout the testing process (see fig. 21b).

- The trend based evolution gave us pretty interesting results (see fig. 20a). Throughout the evolution, the button kept moving to the right. The values show that this trend was picked up around the sixth generation. Its seems as the trend kept pushing the values toward the detected bias which led to the values overshooting the goal. Only after generation fifteen the values for the left offset started to drop. The button also kept moving up to about the same position as the participate button on the previous tests. No trend was ever picked up on that variable.

  In terms of colour, the trend based approach shows a clear tendency towards a bright colour with a lot of contrast. Starting at a rather pale yellow, the green part of the RGB colour space continuously decreased and ending up at a bright pink at generation fourteen (see fig. 23). Similar to the other approaches used, we can observer that the colour values only started to change when the button had reached converged towards the centre of the page and closer to its final position. The evolution of the click speeds show a almost constant decrease. Starting at values over 2000 ms, the trend based approach was able to reduce the time a user needed to click the button to about 1500 ms (see fig. 21b).

- The breadth first approach did have some issues with this test scenario (see fig. 20b). When we look at the data for the the first generation, we see that non of the created mutations positioned the button close to the centre of the screen. This is a similar issue as with the first test. Things got even worse in the second generation when the button moved further away from the screen. Again, the data for this generation shows us that none of the generated mutations (still having a hight variability at this point) moved towards the centre of the screen. Only in the third generation a more central value was generated and the values started to converge.

  The breath first approach and the initially bad mutations show their footprint in the click speeds (see fig. 22b). The first three generations show slow click speeds between 2500 and 3000 ms. Only as better values were found at generation four the click times started to decrease.

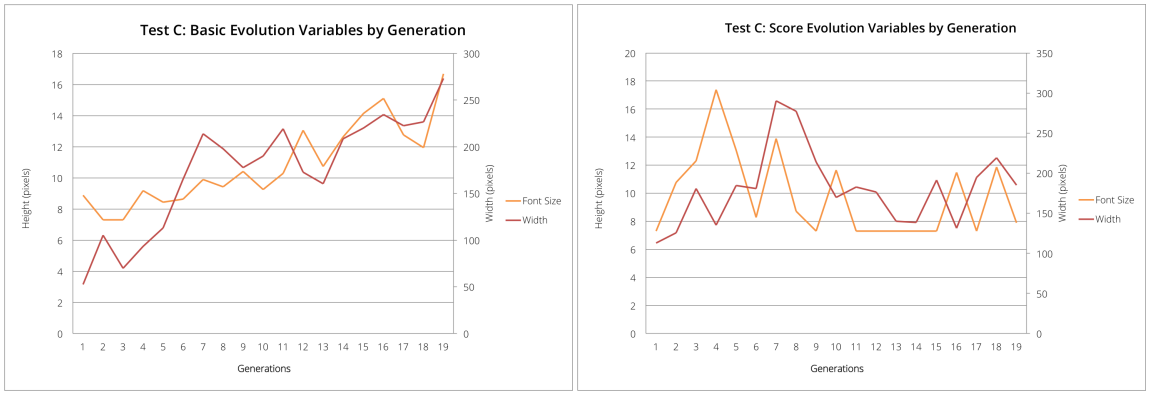
  As expected, the values for colour wildly jump around in the initial stages of the evolution. As variability decrease, the colour started to change from a pale green and converge to a bright yellow towards generation fourteen. Interestingly, this is a different colour as found by the trend evolution which produced a pink button. There are two possibilities for explaining this discrepancy. One might be linked to the starting values of the evolution. When we look at the colour progression oft the blue value, we see that in the initial stages of the evolution it dropped to zero. This change was only possible due to the hight variability in the initial stages of the evolution. With the variability decreasing over time, the framework converged

towards the best solution that could be reached based on those initial values. The other explanation could be linked to the possibility of multiple colours performing equally well, with the actual colour tone being less important than the contrast from the button in regard to its background.

We can see that all evolutionary approaches were able to move the button closer to the centre of the screen and increase the contrast of the evolved button in regard to the background of the page. Comparing performance is again quite difficult because of the small population count which led outliers to greatly influence the average recorded click times. The trend based approach was the quickest to position the button in the centre of the screen. The worst performance was observed with the score based approach. Due to the hight success rates of all mutation, scores were often randomly attributed to mutations which led to mutations with a potentially low click speed not to be picked for the fitness test.

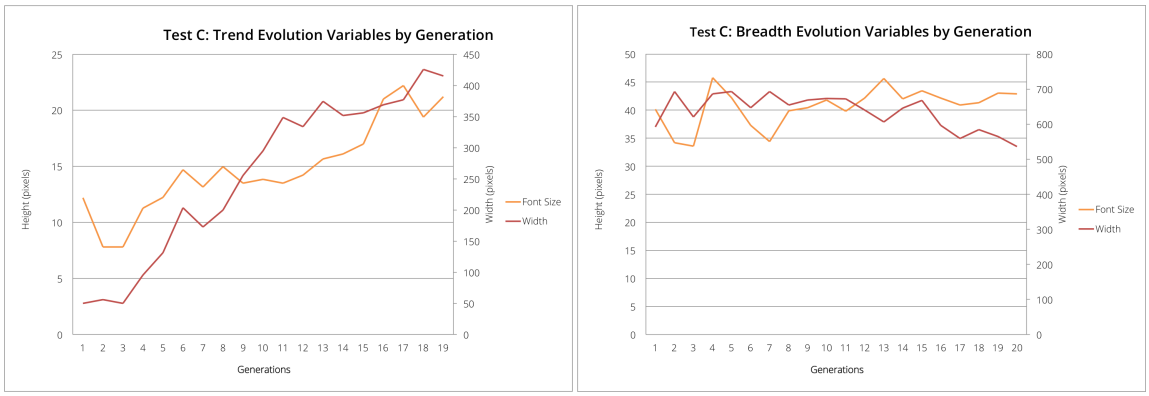
6.1.3 Text readability

The data from the third test gave us mixed results. It seems as the chosen variables had less of an impact on the selected success criterion as in other tests. We also suspect that a larger population count would have benefitted the overall performance of the framework in regard to this test. Let us look at the data and compare the four different evolutionary approaches:



(a) Basic evolution (b) Score evolution

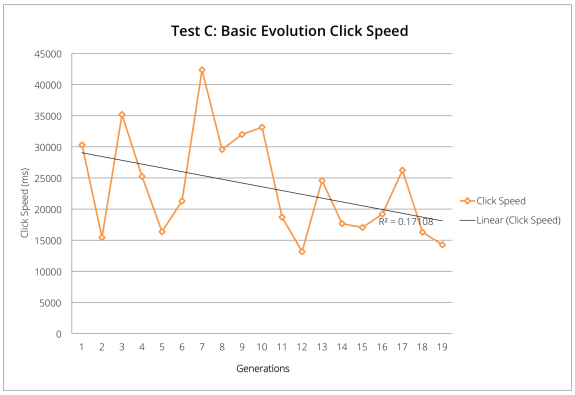
Figure 24: Test D variable evolution (basic/score)



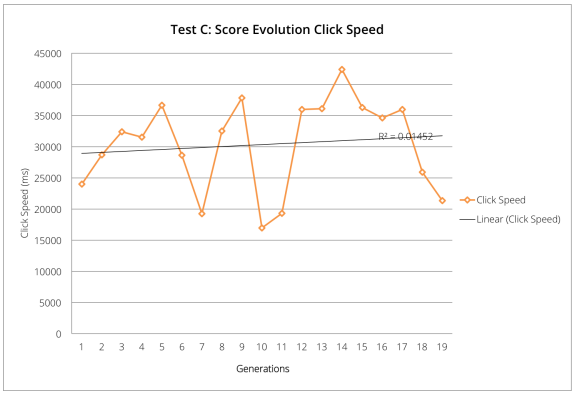
(a) Trend evolution (b) Breath first evolution

Figure 25: Test D variable evolution (trend/breadth)

- The basic evolution preformed rather well. When looking at the data, we can see that both font size as well as width continuously grew over the course of the evolution. (see fig. 24a).

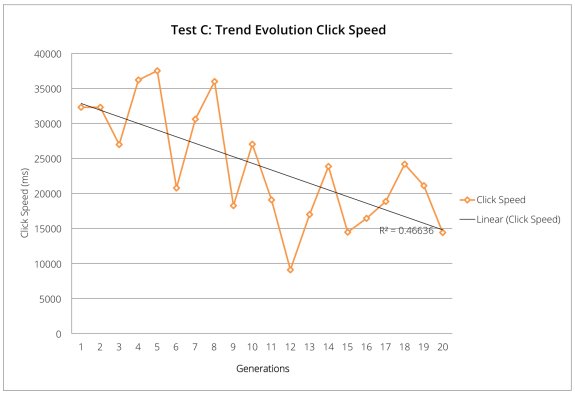


(a) Test C Click Speeds

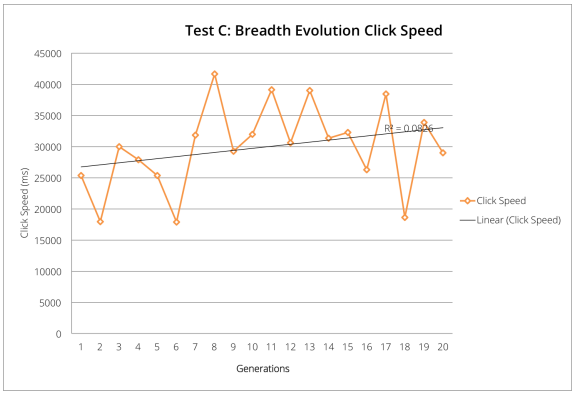


(b) Score evolution

Figure 26: Test C click speeds (basic/score)



(a) Trend evolution



(b) Breath first evolution

Figure 27: Test C click speeds (trend/breadth)

With both variables still rising at the last generation, the evolution was no able to find a convergence for either of the two evolved properties. Click speed revealed a much less steady decline than expected (see fig. 26a) and decreased from an initial average of about 25 seconds to less than twenty seconds towards the end of the evolution

- Similar than the results of the score based evolution in the previous test, the algorithm did not perform well when used with click speed as the success criterion. When we look at the recorded winners by generation, we see that none of the evolved variables converge. The paragraphs width as well as the font size are rising initially but drop back down in subsequent generations (see fig. 24b). The bad performance is reflected in the average click speeds (see fig. 26b) Instead of reducing the time a user takes to answer to question about the colour of the used canisters we observe a slight increase.

- The trend based approach was quicker to increase values for font size and width than the basic evolution (see fig. 25a). A font size of 16px and a width of 250px is already reached at generation ten (in contrast to generation 20 with the basic evolution). The faster variable growth is clearly caused by the trend analysis which is able to pick up trends on both evolved variables.

  The trend based approach was able to decrease click times from an initial thirty seconds to under fifteen seconds at the end of the evolution. (see fig. 27a).

- The breadth first surprised us by finding a completely different solution than seen with the other approaches (see fig. 25b). We can see that the font size started at around 40 pixels and stayed at that value throughout the evolution. The same goes for the width which remained at around 700 pixels.

  The click speeds show that the evolution was not able to significantly reduce the click speeds (see fig. 27b). How can we explain this behaviour? Looking at the database, we see that in the first three generations, all mutations for font size as well as width were very large. Even through the mutations with the smallest font size was picked as a winner, the decreasing variability prevented the font size from converging.

The data shows that the basic as well as the trend based evolution were able to reduce the time a user needed before answering the question. Yet, the graph is riddled with outliers. A possible explanation can be found in the small population count used in the evaluation. Since only fifteen fitness tests are performed per generation, very slow or fast readers can have a large impact on the average click times.

After analysing the data, we have to concede that the test design reveals a series of flaws. Neither the basic nor the trend based approach were able to reach converging values. They would have eventually - but the large discrepancy between the initial and the optimal values, combined with a small variability led to the issue of variables not being able to change fast enough. Secondly, the font weight had too small of an impact on the fitness test in order to converge towards a value. We would need a much larger population count in order to find significant differences in in terms of click speed criterion based on the font weight alone. Other factors were simply much more important and dominated the outcome of the fitness test.

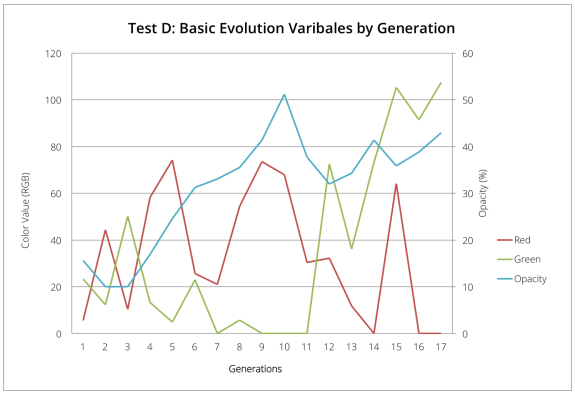
Overall, the trend based evolution gave us the best performance of all four approaches. By identifying trends on both evolved variables, the framework was able to accelerate testing process.

6.1.4 Dress illusion

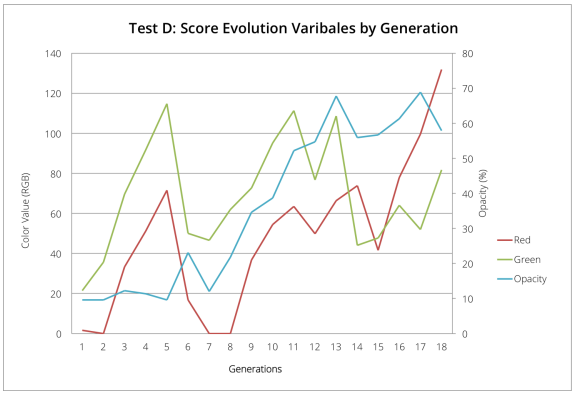
The data from the dress illusion gave us some unexpected results. Looking at polls around the internet, it seemed that about sixty percent of people perceived the dress as being black and blue. The testing data shows that in fact most people (about 80 percent) saw the dress as being black/blue right from the start.

Let us again look at results the four evolutionary approaches produced:

- With the basic evolution, we can see that the opacity of the overlay quickly rose to about fifty percent at generation ten, then dropped back down and stayed at around the forty percent

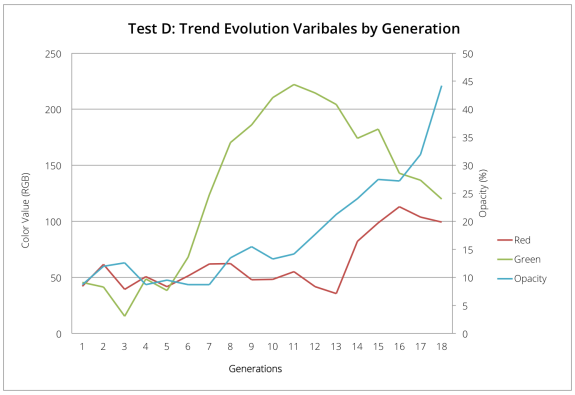


(a) Basic evolution

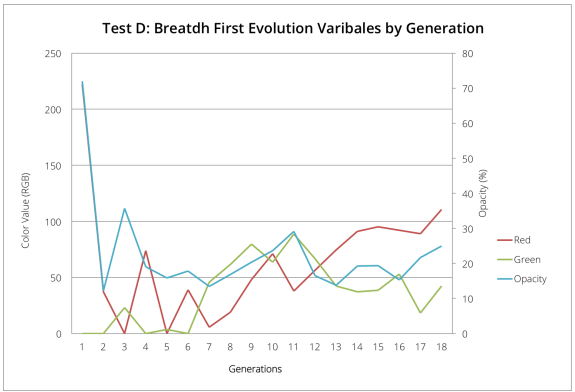


(b) Score evolution

Figure 28: Test D variable evolution (basic/score)

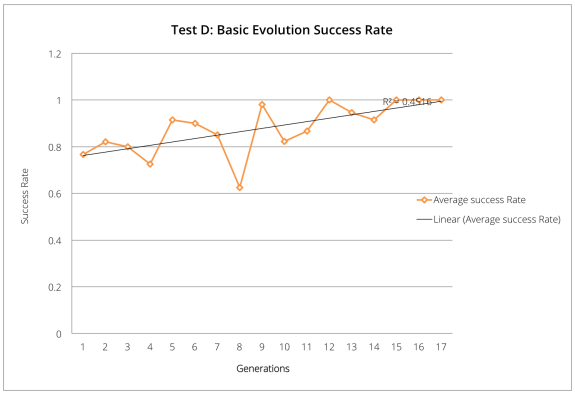


(a) Trend evolution

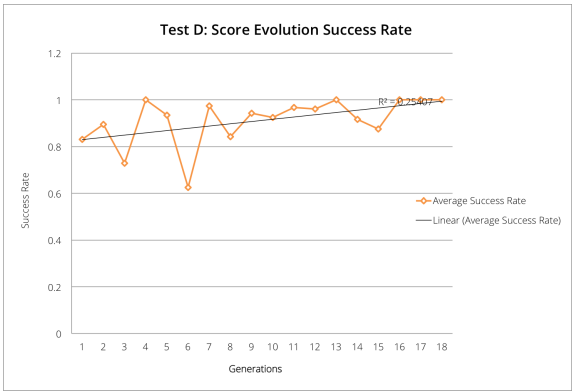


(b) Breath first evolution

Figure 29: Test D variable evolution (trend/breadth)



(a) Basic evolution



(b) Score evolution

Figure 30: Test D success rates (basic/score)

(a) Trend evolution

(b) Breath first evolution

Figure 31: Test D success rates (trend/breadth)

mark. The red part of the overlays RGB fill rose to about seventy before falling back down towards the end of the evolution. The green fill started low and started to rise at generation ten (see fig. 28a). When analysing the average success rates by generation we see that right from the start, almost eighty percent of visitors perceived the dress as being black and blue.

As the opacity increased to about 40 percent, at generation nine, 98 percent of the fitness tests were successful (see fig. 30a) and remained at a very high rate from there on. The drop in opacity at the end as well as the change in colour can be explained by the perfect success rate of all mutations. When every version is successful, the algorithm simply choses one randomly in order to generate new offspring.

Based on the missing convergence of colour values, we can assume that they have a subordinate effect on the fitness test. Also, they only start to change as soon as the opacity rises. This makes sense, since the colour tone of the overlay becomes more and more apparent as the opacity increases.

- The score evolution behaved in a similar way as the basic evolution. We can see that the opacity converged to about seventy percent at generation fourteen. A major difference to the basic evolution can be seen in the colour variables. The first thing we observe is the way they seem to be synchronised. They both start low, quickly rise and end in a range between sixty and a hundred in terms of RGB values (see fig. 28b). The rise in the success rate is very similar to the basic evolution, reaching almost perfect success rates around generation eleven when the overlay reaches an opacity of 40 percent (see fig. 30b). It seems as the algorithm had a slow start but quickly picked up speed at generation seven when the opacity climbed by fifty percent in six generation. This slow start can be explained by looking at the mutations generated in the initial phases of the evolution. There we see that by chance, the framework only generated very low opacity values. This is certainly an issue that will have to be addressed in future releases.

- The trend evolution revealed the same issues that were seen in the previous tests. At generation six, a false trend is detected in the green value of the overlay. At this time, the opacity is very low and the materialisation of this variable does not have any effect on the success criterion. As the opacity starts to increase, the green values have already climbed to over 200, giving the overlay a whitish appearance (see fig. 29a). This behaviour actually leads to a drop in the success rates of subsequent mutations (see fig. 31a). Only as the trend is overcome and values for the green fill start drop (and the red fill starts to rise, giving the overlay stronger blue appearance) the success rates improve and reach almost perfect values towards generation fifteen. This example illustrates well how false trends can have a negative impact on an evolution.

- We were initially rather confused when looking at the data from the breadth first test. It appears

that neither opacity nor the colour values are converging towards any specific value (see fig. 29b). The success rates where even dropping over the course of the evolution (see fig. 31b). We were able to solve the mystery about what had happened by looking at the database. A wrongly initialised success criterion was at fault. Instead of a "click" success criterion, the evolution was set to use "click speed".

Of course this success criterion does not make any sense in this situation and is responsible for the bad results. Fortunately we can still draw some conclusions from this test. A click speed success criterion means that the algorithm tried to reduce the time a person needed to click on the button labeled "black and blue". With both the red and green fill rising throughout the evolution could indicate that people made up their mind faster about the colour of the dress when it was tinted in a stronger blue tone.

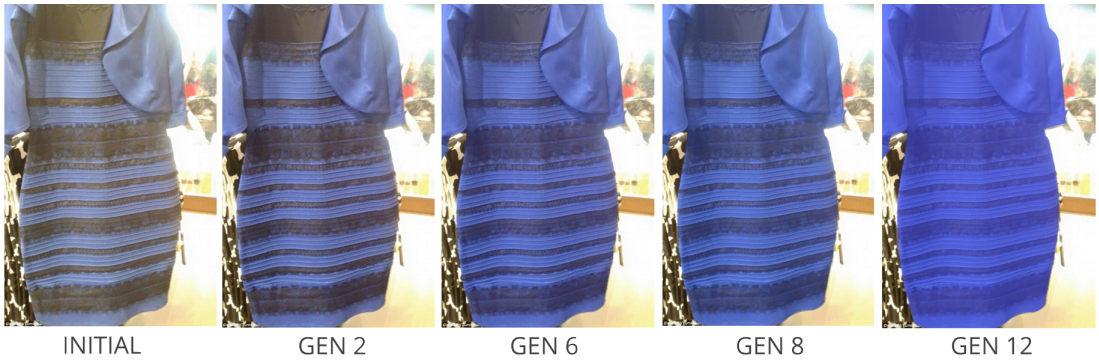


Figure 32: Dress illusion colour evolution (Score based)

The point at which all of the visitors perceive the dress as being black and blue is reached when the overlay reaches an opacity of around forty percent. The highest success rates are achieved when the red and green fill rise to a value between fifty and eighty. Figure 32 shows how the dress appearance changed through the generations (score based approach). We can clearly see how the the blue tone continuously increases and shifts to a strong blue.

Initially, we would have expected the opacity of the overlay to continuously rise before reaching a value of a hundred percent. Why has this not happened? The answer becomes clear when we look at the progression of the success rates. As they reach perfect values (meaning that every generated mutation is successful) the framework can no longer pick a real winner. This is why the values for the overlay don't increase further than the required forty percent mark.

7 Discussion

Based on the data gathered from the tests, we will now discuss all four evolutionary approaches and look at their positive aspects as well as shortcomings when used in evolutionary testing.

7.1 Evolutionary approaches

7.1.1 Basic evolution

The basic evolution performs evolution in its most basic form. Mutation generations as well as selection remain completely random. The data from the evaluation shows that even through this seemingly random process was able to optimise web pages in terms of a specified success criterion. The evaluation indicates that this approach has no significant drawbacks and works for pretty much every scenario. Compared to the other approaches, it seems as the basic approach requires a higher number of generations before converging. We expect the basic evolution to perform better in scenarios with larger populations as well as more fitness tests.

7.1.2 Score based evolution

When looking at the evaluation data, we can see that the score based approach works very well for removing unsuccessful mutations quickly and favour successful variations. As a positive side effect, this approach also increases page consistency as only very few users get to see the bad mutations. Over time, a single version acquires a larger score and is more likely to be shown to a user visiting the page.

We were able to identify tow possible issues when using this approach.

Since in it's current state, the score based evolution only attributes scores based on success rates, using this approach led to unsatisfying results in both tests that used the "click speed" success criterion. Both tests had very large success rates which made it impossible for the algorithm to attribute any meaningful scores to mutations. Often times, this led to a random mutation acquiring a large score and prevent other, possibly better mutations from being picked for the fitness test.

It also seems as this approach has a tendency to ignore subordinate variables (for example border radius in the first test). This behaviour is not surprising when we think of the following scenario: An evolution is performed on a dominant as well as a subordinate variable. Let us assume a generation is spawned with two mutations with similar dominant but different subordinate variable materialisations. If both version are successful, but the the one with the worse subordinate variable gets selected first, its chance of being chosen again increases. It is thus possible that the successful, but slightly less optimal mutation is selected as a winner.

7.1.3 Trend based evolution

The trend based approach seemed to work quite well thought the tests that were performed where the optimal solution was far away from the initial state. Taking the click speed test as an example, after the fourth generation, the algorithm was able to identify a trend in the left offset value of the button and thus bias the creation of offspring to follow this trend.

The trend analysis also lead to two issues:

The first one being that the algorithm seems to be overshooting optimal solutions. The evaluation has shown that when trends are detected and the optimal value reached, it can take a certain amount of iterations before the trend is stopped and the values bounce back.

Another observed issue was false trend identification on subordinate variables. With the dress illusion, the approach identified a random change on a subordinate variable as a trend. This false trend slowed down the evolutionary process as several iterations were needed to eliminate the wrong values. We can possibly solve both issues by adapting the trend based offspring creation by decreasing the shift of the target window used to mutate variables.

7.1.4 Breadth first evolution

The breadth first approach is not a classical evolutionary algorithm and behaves quite differently than the others. One major advantage of a breath first search is the possibility to find optimal solutions, classic evolutionary approaches are not able to reach. In our testing scenario, the breadth first approach did not perform as well as the other approaches.

In theory, when the variable count is low, it is likely for the algorithm to find a good solution within the first few iterations. But as the variable count increases, that chance drops significantly and might reach a point where all initial mutations fail. Since the variability halves with each generation, it is possible that the optimal solution will never be reached because all initial mutations where too far off from the optimal variable configuration. This issue was most prominent in the test about improving text scanning speed were the font size was not able to converge due to a series of bad initial mutations.

One way to improve the algorithms performance would certainly be by increasing the amount of offspring that is generated for each generation, this way, we would greatly increase the chance of finding a variable configuration that is not too far off from the optimal solution.

One limitation for using this algorithm is the need for the user to define evolution bounds. If not set correctly, the algorithm might produce horrid variations that would make no sense to the user in the first few generations. Of course such variations would not be successful and die off eventually, but developers would cringe at the idea of losing visitors to unusable version of their

page. We got a glimpse of this, when during the evaluation a user was presented a variation that contained barely readable text and asked us if something was wrong with the browser he used to view the tests. This means that if no bounds can be set, using the breadth first approach is not sensible.

7.2 Conclusion

The implemented testing library which we used to initialise, run and monitor evolution driven experiments has certainly demonstrated its utility when performing online controlled experiments. Deploying the testing suite only requires a few lines of code and by using the visual editor, evolutionary tests can be set up within minutes. The framework was intensively used during the evaluation process. Setting up all evolutionary tests by hand would have been much more error prone and taken significantly more time.

The performed evaluation has shown that introducing evolutionary aspects into the domain of multivariate testing does in fact work. We have seen that even with a small number of users it is possible to utilise differences in user behaviour for improving web pages in terms of the specified success criterion. Wile comparing the performance of the four implemented evolutionary approaches was difficult due to the low amount of fitness tests that were performed for each mutation, we were able to get a good understanding of the advantages and possible downsides of using each of the testing procedures. The data gathered in the final evaluation could certainly be used to improve the proposed algorithms and even lead to a hybrid solution which would combine benefits from all four approaches.

The framework will still have to prove its usability in testing scenarios where a large number of variables are evolved simultaneously. Especially when mixing dominant and subordinate variables in the optimisation process, the testing suite might need a lot of generations before converging on all specified properties. Further work also needs to be done in order to improve the automated adjustment of evolutionary parameters throughout a running evolution. The frameworks performance could certainly be improved if we can give better answers to questions like: When do we need to adjust the variability of an evolved property? In which situations do we decrease or increase the population count? What is the optimal number offspring produced when spawning a new generation?

Based on the identified limitations and the data gathered from the evaluation, we think that the framework will not necessarily serve as a replacement for classical AB or multivariate testing but should rather be utilised as a tool for fine tuning specific elements on web pages. On of the key advantages of the implemented approach is the that even if a pages success rate cannot be improved based on the selected variables, the framework will not produce mutations which perform worse than the initial control version. If the specified variables have no impact on the selected success criteria, the evolved variables will simply stay around their initial values. We think that our automated approach is best suited for test cases, in which a series of user interface properties are selected for optimisation that display a measurable effect on the target metric which is being used.

7.3 Issues and improvements

The evolution framework has proven its utility throughout the tests. But in order to be actually deployed on large web pages, several adjustments would have to be made. The following list discusses issues and useful improvements that were we've encountered during the development and evaluation process.

- Randomly selecting variations from the database when a user opens a page is in line with the evolution metaphor. As the random selection will balance out, this is not a problem for large populations. With fewer fitness tests being performed it is possible for potentially successful mutations not to be subjected to the fitness test at all. That is why a mechanism should be introduced which evenly distributes variations onto visitors.

- The evaluation process has shown that subordinate variables only started to change after the more dominant variables had converged. A low population count certainly accentuates this

behaviour. We could potentially improve the way subordinate variables behave by stopping the evolution of properties that have already converged. This would allow properties with less of an impact on the success criterion to be weighted in more.

- In order to have better control of how an evolved property behaves, it would be useful if the specified variability would not target all properties of a running evolution but could be defined for each individual property. This would fix issues where some properties need to change more in order to have a measurable effect on the success criterion.

- When discussing the framework with web developers, it was quickly pointed out, that a single range slider is not enough to limit the possible variable manifestations. In its current state it is not possible to for example limit the range of allowed positions from 0 to 100 and from 500 to 800. Having multiple sliders per variable in order to specify multiple ranges would solve that problem.

- The framework currently only allows for the evolution of numeric CSS properties such as width, height, font size etc. Other properties such as font-family, floating, text alignment are not accessible in the current version. Implementing this feature would allow for a much wider range of possibilities and give developers more freedom on the types of properties that should be optimised.

- In its current state, the framework can only be used to run one evolution per URL. In order to test business processes and use the framework in single page web applications, it would be useful if we could specify that an evolutions can affect multiple pages.

- Another feature that was requested is the possibility to change evolution variables on the run. Currently, the evolution properties are either static or automatically adjusted when the dynamic evolution properties is enabled.

- The framework already supports a wide range of success criteria. In a real world scenario it would make sense to add others such as scrolling, scroll speed etc.

- One complaint we got when developers used the framework was the fact that when evolving properties on multiple DOM elements, the testing suite would only ever highlight the currently selected element. It would definitely make sense to mark all DOM elements for which properties have been added.

8 Future development

By incorporating the knowledge gathered form the evaluation process and improving the testing suite, it could be made available to a broader audience. This would requires setting up the appropriate infrastructure and implement a registration process through which users would be able to get private keys allowing them to run tests on their own web pages. This approach would also open up a potential commercial use of the framework. We could allow users to utilise the testing suite at no cost up to some fixed traffic.

References

[1] *Bayes' Rule: A Tutorial Introduction to Bayesian Analysis.* Sebtel Press, 2013.

[2] Optimizely, http://www.optimizely.com, 2015.

[3] Daniel Ashlock. *Evolutionary Computation for Modeling and Optimization.* Springer Science and Business, 233 Spring Street, New York, NY 10013, USA, 2006.

[4] George E.P. Box, Stuart J. Hunter, and William G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery.* John Wiley and Sons, Inc., 2005.

[5] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of The Psychology of The Psychology of Human-Computer Interaction.* Lawrence Earlbaum Associates, Hillsdale, New Jersey, 1983.

[6] Brian Christian. The a/b test: Inside the technology that's changing the rules of business. *WIRED*, 2012.

[7] Thomas Crook, Brian Frasca, Ron Kohavi, and Roger Longbotham. Seven pitfalls to avoid when running controlled experiments on the web. *KDD'09*, page 9, 2009.

[8] Charles Darwin. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life.* Murray, London, 1859.

[9] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley and Sons, Inc., 1966.

[10] J.H Holland. Adaptation in natural and artificial systems. *University of Michigan Press*, 1975.

[11] Laura M. Holson. Putting a bolder face on google. *NY Times*, 2009.

[12] Ron Kohavi, Alex Denk, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. pages 1168–1176, 2013.

[13] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. *KDD'07*, page 9, 2007.

[14] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. Controlled experiments on the web: survey and practical guide. *Springer Science+Business Media*, 2008.

[15] Robert L. Mason, Richard F. Gunst, and James L. Hess. *Statistical Design and Analysis of Experiments with Applications to Engineering and Science.* John Wiley and Sons, Inc., 1989.

[16] Colin McFarland. *Experiment!: Website Conversion Rate Optimization with A/B and Multivariate Testing.* New Riders, first edition edition, 2012.

[17] Mike Moran. *Do It Wrong Quickly: How the Web Changes the Old Marketing Rules.* IBM Press, 2007.

[18] Larry Page. Founders letter, https://investor.google.com/corporate/2013/founders-letter.html, 2013.

[19] Eric T. Peterson. *Web Analytics Demystified: Marketer's Guide to Understaning How Your Web Site Affects Your Business.* Celilo Group Media, 20014.

[20] Martin C. Robert. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall., 2008.